HTML5

Хуан Диего Гоше

MINK BOOKS

ПИТЕР

HTML5

for Masterminds

J.D. Gauchat
www.jdgauchat.com

Edited by: Laura Edlund
www.lauraedlund.ca

Хуан Диего Гоше

HTML5

ПИТЕР®

Москва · Санкт-Петербург · Нижний Новгород · Воронеж
Ростов-на-Дону · Екатеринбург · Самара · Новосибирск
Киев · Харьков · Минск
2013

ББК 32.988-02-018.1

УДК 004.43

Г74

Гоше Х. Д.

Г74 HTML5. Для профессионалов. — СПб.: Питер, 2013. — 496 с.: ил.

ISBN 978-5-496-00099-4

Мы стоим на пороге революции в веб-разработке и программировании для мобильных устройств, и в основе всех этих изменений лежит формат HTML5.

Эта книга поможет вам получить необходимые знания об этом стандарте и освоить сложные темы, включенные в спецификации HTML5. Вы узнаете, как организовать ваши документы с HTML5, как оформлять их стилями с помощью CSS3 и как работать с самыми продвинутыми JavaScript API.

Данное издание не является введением в HTML5, а представляет собой полноценный учебный курс, который научит вас создавать с помощью HTML5 современные сайты и веб-приложения. Каждая глава посвящена определенной ключевой теме HTML5, также рассмотрены сложные вопросы HTML5, CSS3 и JavaScript. Книга содержит множество примеров программного кода, благодаря чему вы сможете легко усвоить и применить знания по каждому тегу, стилю и функции, включенным в спецификации HTML5.

ББК 32.988-02-018.1

УДК 004.43

Все права защищены. Никакая часть данной книги не может быть воспроизведена в какой бы то ни было форме без письменного разрешения владельцев авторских прав.

Информация, содержащаяся в данной книге, получена из источников, рассматриваемых издательством как надежные. Тем не менее, имея в виду возможные человеческие или технические ошибки, издательство не может гарантировать абсолютную точность и полноту приводимых сведений и не несет ответственности за возможные ошибки, связанные с использованием книги.

Права на издание этой книги получены по соглашению с MARCOMBO, S.A. of Gran Via Corts Catalanes 594, 08007, Barcelona, Spain.

This edition of EL GRAN LIBRO DE HTML5, CSS3 Y JAVASCRIPT (Spanish version) and HTML5 FOR MASTERMINDS (English version) by J.D. Gauchat is published by arrangement with MARCOMBO, S.A. of Gran Via Corts Catalanes 594, 08007, Barcelona, Spain.

ISBN 978-1463604059 англ.
ISBN 978-5-496-00099-4

© 2011 by Juan Diego Gauchat
© Перевод на русский язык ООО Издательство «Питер», 2013
© Издание на русском языке, оформление ООО Издательство «Питер», 2013

Краткое оглавление

Оглавление

Введение

HTML5 — это не новая версия старого языка разметки и даже не усовершенствованный вариант этой уже перешедшей в разряд «древних» технологии. Нет, это совершенно новая концепция построения веб-сайтов и приложений в эру мобильных устройств, облачных вычислений и сетевой передачи данных.

Все началось очень давно с простейшей версии HTML, предназначенной для создания базовой структуры страниц, организации их содержимого и распространения информации. Основной целью создания этого языка, как и самой Сети, была передача текстовой информации.

Ограниченный потенциал HTML подтолкнул компании к разработке новых языков и программного обеспечения, благодаря которым веб-страницы обрели невиданные доселе характеристики. Первые наработки превратились в мощные и популярные плагины. Из простых игр и смешных анимированных картинок выросли замысловатые приложения, новые возможности которых навсегда изменили саму концепцию Сети.

Изо всех новинок наиболее успешными оказались Java и Flash, которые получили самое широкое распространение. Считалось, что будущее Интернета за ними. Однако по мере того как число пользователей увеличивалось, а Интернет из простого механизма соединения компьютеров превращался в глобальное поле для ведения бизнеса и социального взаимодействия, недостатки этих двух технологий становились все более явными, и в итоге им был подписан смертный приговор.

Основной недостаток Java и Flash можно описать как отсутствие интеграции. Обе технологии с самого начала воспринимались как плагины — то, что вставляется в уже существующую структуру и делит с ней некоторое экранное пространство. Между приложениями и документами невозможно было установить эффективные коммуникационные и интеграционные каналы.

Отсутствие интеграции стало определяющим фактором и положило начало развитию языка, который умеет делить пространство документа с HTML и на который при этом не распространяются ограничения плагинов. JavaScript, встраиваемый в браузеры интерпретируемый язык, очевидно, двигался в правильном направлении к цели — расширению функциональности Сети и улучшению взаимодействия с пользователем. Однако даже через несколько лет рынок так и не смог полностью принять этот язык, а его популярность упала из-за неверного применения и неудачных попыток продвижения. У критиков JavaScript были все основания для того, чтобы ругать его. В те времена JavaScript не мог заменить собой функциональность Flash и Java. И даже когда стало очевидно, что Java и Flash ограничивают масштаб веб-приложений и изолируют содержимое от контекста, пользующиеся всеобщей любовью возможности, такие как потоковая передача видео, не потеряли ведущее роли в жизни Сети, а их эффективное предоставление было возможно только с помощью этих технологий.

Несмотря на определенный успех язык Java терял позиции. Сложная природа, медленное развитие и отсутствие интеграции снизили значимость этого языка до такой степени, что сегодня он почти не используется в распространенных веб-приложениях. Вычеркнув Java из активного арсенала, рынок взглянул в сторону Flash. Однако Flash обладает теми же базовыми характеристиками, что и его конкурент в сфере построения сетевых приложений, и, таким образом, подвержен тем же рискам и обречен на ту же судьбу.

Тем временем развитие программного обеспечения, предлагающего доступ к Сети, продолжалось. Одновременно с внедрением новых возможностей и более быстрых способов доступа к Интернету разработчики браузеров непрерывно совершенствовали механизмы JavaScript. Увеличение мощи дало жизнь новым возможностям, и этот скриптовый язык готов был воплотить их.

В какой-то момент этого эволюционного процесса части разработчиков стало очевидно, что ни Java, ни Flash не смогут в итоге предоставить инструменты, необходимые для создания приложений, отвечающих требованиям неуклонно увеличивающейся пользовательской аудитории. Эти разработчики стали внедрять JavaScript в свои приложения новыми, невиданными доселе способами. Инновации и их поразительные результаты привлекли внимание еще большего числа программистов. Скоро на свет появилось то, что знакомо нам под именем Web 2.0, а взгляды на JavaScript в сообществе разработчиков кардинальным образом изменились.

JavaScript стал тем языком, благодаря которому разработчики смогли предложить огромное количество новшеств и реализовать то, чего раньше в Сети попросту не существовало. За прошедшие годы программисты и веб-дизайнеры всего мира придумали множество невероятных трюков, позволявших преодолеть ограничения данной технологии и первоначальную нехватку мобильности. HTML, CSS и язык JavaScript создали ту идеальную комбинацию, которой недоставало для очередного рывка в эволюции Сети.

Фактически, HTML5 представляет собой усовершенствованную версию этой комбинации — клей, надежно скрепляющий все фрагменты. HTML5 предлагает стандарты для каждого аспекта Сети и ясно описывает предназначение всех участвующих технологий. Теперь HTML обеспечивает структурные элементы, CSS фокусируется на том, как превратить эту структуру в нечто визуально привлекательное и удобное в использовании, а JavaScript предлагает мощь, необходимую для обеспечения функциональности и построения полноценных веб-приложений.

Границы между веб-сайтами и приложениями наконец-то растворились. Необходимые технологии готовы к применению. Будущее Сети выглядит очень многообещающим, а совместная эволюция и описание этих трех технологий (HTML, CSS и JavaScript) в одной мощной спецификации превращает Интернет в лидирующую платформу для разработки. Без сомнения, HTML5 идет во главе современных сетевых технологий.

ВНИМАНИЕ

Сейчас не все браузеры поддерживают функциональность HTML5, и большинство из возможностей, описанных в спецификациях HTML5, пока находятся на стадии разработки. Мы рекомендуем по мере знакомства с главами этой книги исполнять предложенные примеры кода в новейших версиях Google Chrome и Firefox. Google Chrome основывается на WebKit — браузерном механизме с открытым кодом, который поддерживает почти все возможности, уже реализованные в HTML5. Это отличная платформа для тестирования. Firefox — один из лучших браузеров для разработчиков, а его механизм Gecko также превосходно поддерживает функциональность HTML5.

Вне зависимости от того, какой браузер вы используете, всегда помните о том, что хороший разработчик устанавливает и тестирует свой код во всех программах, предлагаемых на рынке. Тестируйте примеры кода из этой книги во всех доступных вам браузерах.

продолжение ⇩

Загрузить новейшие версии браузеров вы сможете, посетив следующие веб-сайты:

http://www.google.com/chrome;

http://www.apple.com/safari/download;

http://www.mozilla.com;

http://windows.microsoft.com;

http://www.opera.com.

В конце мы рассмотрим альтернативные технологии, помогающие создавать веб-сайты и приложения, которые будут доступны и в старых версиях браузеров, и в браузерах, несовместимых с HTML5.

1 Документы HTML5

Базовые компоненты

Фактически, при построении веб-приложений HTML5 охватывает три направления: структуру, стиль и функциональность. Об этом никогда официально не объявлялось, но даже с учетом того, что некоторые API и вся спецификация CSS3 лежат за пределами описания HTML5, язык HTML5 считается результатом объединения HTML, CSS и JavaScript. Между этими технологиями существуют сильные зависимости, и они функционируют как единое целое под общим названием «спецификация HTML5». HTML отвечает за структуру, CSS выводит эту структуру и ее содержимое на экран, а JavaScript делает все остальное, что тоже немаловажно, как мы убедимся далее в этой книге.

Несмотря на взаимную интеграцию этих технологий, неотъемлемой частью документа является его структура. Она определяет элементы, необходимые для размещения статического или динамического содержимого, а также служит основной платформой для приложений. Из-за разнообразия устройств с доступом в Интернет и интерфейсов для взаимодействия с Сетью такой базовый аспект, как структура, является жизненно важной частью документа. Структура должна обеспечивать форму, порядок и гибкость, и она должна быть прочной, как фундамент здания.

Для того чтобы создавать веб-сайты и приложения с помощью HTML5, нам сначала необходимо узнать, из чего состоит структура. Построив

прочный фундамент, позднее мы сможем применить остальные компоненты, чтобы воспользоваться всеми преимуществами новых возможностей.

Итак, начнем с самого начала и будем продвигаться вперед шаг за шагом. В первой главе вы узнаете, как построить шаблон для будущих проектов с использованием новых элементов HTML, появившихся в HTML5.

САМОСТОЯТЕЛЬНО

В своем любимом текстовом редакторе создайте новый документ. С помощью этого документа вы будете тестировать в браузере примеры кода из этой главы. Так вам будет проще запомнить новые теги и привыкнуть к использованию новой разметки.

ПОВТОРЯЕМ ОСНОВЫ

HTML-документ — это текстовый файл. Если у вас нет специальных приложений, то вы можете использовать простой редактор Блокнот в Windows или любой другой текстовой редактор. Файл необходимо сохранить с расширением .html*, а имя файла может быть любым (например, mycode.html).*

ВНИМАНИЕ

Чтобы получить дополнительную информацию и примеры кода, посетите наш веб-сайт: *www.minkbooks.com*.

Общая структура

К структуре документов HTML предъявляются строгие требования. Все части документа отделены друг от друга, каждая из них объявлена и заключена в определенные теги. В этом разделе мы рассмотрим построение общей структуры HTML-документа и научимся использовать новые семантические элементы, являющиеся частью HTML5.

Тип документа

В первую очередь необходимо указать тип создаваемого нами документа. В HTML5 это делается чрезвычайно просто (листинг 1.1).

Листинг 1.1. Использование элемента <doctype>

```
<!DOCTYPE html>
```

ВНИМАНИЕ

Эта строка должна идти самой первой в файле HTML, и перед ней не должно быть ни пробелов, ни пустых строк. Она активирует стандартный режим браузеров для интерпретации HTML5, когда это возможно (или игнорирует его в противном случае).

САМОСТОЯТЕЛЬНО

Вы можете сразу начать писать код в своем HTML-файле и постепенно добавлять в него элементы, которые мы будем изучать далее.

<html>

После объявления типа документа мы переходим к построению древовидной структуры HTML. Как всегда, в корне дерева находится элемент `<html>`. Внутри этого элемента помещается весь остальной код HTML (листинг 1.2).

Листинг 1.2. Использование элемента <html>

```
<!DOCTYPE html>
<html lang="ru">

</html>
```

Единственный атрибут, указания которого требует HTML5, — это атрибут `lang` открывающего тега `<html>`. Он определяет язык содержимого

в создаваемом нами документе, и в данном случае мы выбрали русский — значение `ru` (английскому языку соответствует значение `en`).

ПОВТОРЯЕМ ОСНОВЫ

HTML — это язык разметки для построения веб-страниц. Теги HTML представляют собой ключевые слова и атрибуты, окруженные угловыми скобками, например, <html lang="ru">. В данном случае ключевое слово — это html, а lang — атрибут со значением ru. Большинство тегов HTML парные: один открывающий, а второй закрывающий. Содержимое находится между тегами. В нашем примере <html lang="ru"> указывает на начало кода HTML, а </html> — на завершение кода. Сравните открывающий и закрывающий теги, и вы увидите, что закрывающий тег отличается только слешем (/) перед ключевым словом (например, </html>). Весь наш остальной код будет находиться между этими двумя тегами: <html> ... </html>.

ВНИМАНИЕ

Что касается структуры и составляющих ее элементов, HTML5 — чрезвычайно гибкий инструмент. Элемент <html> можно использовать без атрибутов или вообще не добавлять в код. Но в целях обеспечения совместимости, а также по другим причинам, которые не стоит здесь рассматривать, мы рекомендуем вам следовать некоторым базовым правилам. Мы научим вас создавать документы с использованием хорошего HTML-кода.

Значения атрибута `lang` для других языков вы можете найти на веб-странице: www.w3schools.com/tags/ref_language_codes.asp.

<head>

Продолжаем построение документа. HTML-код, который добавляется между тегами `<html>`, делится на два основных раздела. Как и в предыдущих версиях HTML, первый раздел представляет собой «голову», а второй — «тело» кода. Таким образом, нужно создать эти два раздела, используя уже знакомые элементы `<head>` и `<body>`.

Разумеется, первым идет элемент <head>, и, как и у других структурных элементов, у него есть открывающий и закрывающий теги (листинг 1.3).

Листинг 1.3. Использование элемента <head>

```
<!DOCTYPE html>
<html lang="ru">
<head>

</head>

</html>
```

Сам тег по сравнению с предыдущими версиями языка не изменился, и его назначение остается таким же. Внутри тега <head> мы определяем заголовок веб-страницы, объявляем кодировку символов, добавляем общую информацию о документе и приводим ссылки на внешние файлы, содержащие стили, сценарии и даже изображения, необходимые для отображения страницы.

За исключением заголовка и некоторых значков, вся остальная информация, содержащаяся между тегами <head>, на экране обычно не отображается.

<body>

Следующий большой раздел, который является частью основной структуры документа HTML, — «тело». Это видимая часть документа, содержимое которой описывается внутри тега <body>. Данный тег также не изменился по сравнению с предыдущими версиями HTML.

Листинг 1.4. Использование элемента <body>

```
<!DOCTYPE html>
<html lang="ru">
<head>

</head>
<body>

</body>
</html>
```

ПОВТОРЯЕМ ОСНОВЫ

Пока что мы написали очень простой код, но уже с составной структурой. Смысл в том, что HTML-код — это не последовательный набор инструкций. HTML — это язык разметки, то есть набор тегов или элементов, чаще всего парных, которые можно вкладывать друг в друга (одни элементы могут целиком находиться внутри других). В первой строке кода из листинга 1.4 находится одиночный тег с определением документа, сразу за ним следует открывающий тег <html lang=»ru»>. Этот тег и закрывающий тег </html> в конце документа указывают на начало и конец HTML-кода. Между ними мы добавили теги, определяющие два важных раздела основной структуры документа: <head> и <body>. Это также парные теги. Позже в этой главе вы узнаете о других тегах, которые можно вставлять внутрь пар <head> и <body>. В результате получается древовидная структура, в корне которой находится тег <html>.

<meta>

Настало время вплотную заняться «головой» документа. Здесь было добавлено несколько изменений и новшеств, среди которых тег, определяющий кодировку символов в документе. Это тег `<meta>`, который указывает на то, каким образом текст должен отображаться на экране (листинг 1.5).

Листинг 1.5. Использование элемента <meta>

```
<!DOCTYPE html>
<html lang="ru">
<head>
  <meta charset="utf-8">

</head>
<body>

</body>
</html>
```

В HTML5 данный элемент (как и многие другие) упрощен. Новый тег `<meta>` для обозначения кодировки символов стал короче. Разумеется, вы можете поменять кодировку UTF-8 на любую другую и использовать другие теги `<meta>`, например `description` или `keywords`, как показано в листинге 1.6.

Листинг 1.6. Добавление различных элементов <meta>

```
<!DOCTYPE html>
<html lang="ru">
<head>
  <meta charset="utf-8">
  <meta name="description" content="Это пример кода HTML5">
  <meta name="keywords" content="HTML5, CSS3, JavaScript">

</head>
<body>

</body>
</html>
```

ПОВТОРЯЕМ ОСНОВЫ

В документе может быть несколько тегов <meta>, содержащих общие объявления, однако эта информация не отображается в окне браузера. Она используется только поисковыми системами и устройствами, которым требуются предварительные данные о нашем документе. Как уже говорилось, кроме заголовка и некоторых значков, никакая другая информация между тегами <head> пользователям не видна. В коде листинга 1.6 атрибут name внутри тега <meta> определяет тип тега, а в атрибуте content объявляется его значение, однако эти сведения на экран не выводятся. Чтобы побольше узнать о теге <meta>, зайдите на наш веб-сайт и просмотрите ссылки для этой главы.

В HTML5 одинарные теги не обязательно должны иметь закрывающий слеш (`/`), но мы все же рекомендуем использовать его для совместимости. Предыдущий пример кода можно записать, как показано в листинге 1.7.

Листинг 1.7. Одинарные теги с закрывающим слешем

```
<!DOCTYPE html>
<html lang="ru">
<head>
  <meta charset="utf-8" />
  <meta name="description" content="Это пример" />
  <meta name="keywords" content="HTML5, CSS3, JavaScript" />

</head>
<body>

</body>
</html>
```

<title>

Тег <title>, как и раньше, используется для определения заголовка документа (листинг 1.8), и ничего нового о нем сказать нельзя.

Листинг 1.8. Использование элемента <title>

```
<!DOCTYPE html>
<html lang="ru">
<head>
  <meta charset="utf-8">
  <meta name="description" content="Это пример HTML5">
  <meta name="keywords" content="HTML5, CSS3, JavaScript">
  <title>Этот текст — заголовок документа</title>

</head>
<body>

</body>
</html>
```

ПОВТОРЯЕМ ОСНОВЫ

Текст между тегами <title> представляет собой общий заголовок для создаваемого документа. Чаще всего браузеры отображают его в заголовке окна.

<link>

Еще один важный элемент, являющийся частью «головы» документа, — <link>. Он используется для подключения к документу стилей, сценариев, изображений или значков из внешних файлов. Чаще всего с помощью <link> подключают таблицу стилей из внешнего CSS-файла (листинг 1.9).

Листинг 1.9. Использование элемента <link>

```
<!DOCTYPE html>
<html lang="ru">
<head>
  <meta charset="utf-8">
  <meta name="description" content="Это пример HTML5">
  <meta name="keywords" content="HTML5, CSS3, JavaScript">
  <title>Этот текст — заголовок документа</title>
  <link rel="stylesheet" href="mystyles.css">
</head>
<body>

</body>
</html>
```

ПОВТОРЯЕМ ОСНОВЫ

Таблица стилей — это набор правил форматирования, позволяющий менять внешний вид документа, например размер и цвет текста. Без таких правил текст и другие элементы отображаются на экране с использованием стандартных стилей браузера (размер, цвет и другие характеристики устанавливаются по умолчанию). Стили представляют собой простые правила, занимающие обычно всего несколько строк кода, и их можно объявить в самом документе HTML. Как мы увидим далее, не обязательно подгружать эту информацию из внешних файлов, но использование отдельных файлов для таблиц стилей — хорошая практика. Загрузка стилей CSS из внешнего файла упрощает структуру основного документа, увеличивает скорость загрузки веб-сайта, а также позволяет воспользоваться преимуществами новых возможностей HTML5.

В HTML5 избавились от необходимости указывать тип подключаемой таблицы стилей, таким образом, атрибут `type` больше не используется. Теперь для внедрения файла со стилями достаточно двух атрибутов: `rel` и `href`. Название атрибута `rel` происходит от англ. *relation* (отношение), и его значение определяет, чем для нашего документа является подключаемый файл. В данном случае атрибуту `rel` присвоено значение `stylesheet`, то есть мы сообщаем браузеру, что mystyles.css — это CSS-файл со стилями, необходимыми для отображения страницы (в следующей главе мы займемся стилями CSS и научимся подключать их к документу).

С добавлением этого последнего элемента работу над «головой» документа можно считать законченной. Теперь мы можем как следует поработать над телом, где и происходит все самое интересное.

Структура тела документа

Структура тела документа (код между тегами `<body>`) определяет его видимую часть. Именно в этом коде находится содержимое веб-страницы.

Язык HTML с самого начала предлагал разные способы построения и организации информации в теле документа. Один из первых элементов, выполняющих эту функцию, — `<table>`. Таблицы позволяют организовывать данные, текст, изображения и инструменты в строки и столбцы, хотя первоначально этот элемент создавался для других целей.

Во времена появления Сети использование таблиц стало настоящей революцией, большим шагом вперед в области визуализации документов и работы с веб-страницами. Позднее эти функции таблиц постепенно взяли на себя другие элементы, позволявшие добиваться тех же результатов быстрее с использованием меньшего объема кода. Создавать, поддерживать и приспосабливать страницы к новым требованиям стало намного проще.

Основным элементом стал `<div>`. С появлением интерактивных веб-приложений и взаимной интеграции HTML, CSS и JavaScript использование `<div>` стало привычной практикой. Однако этот элемент, как и `<table>`, не дает достаточно информации о той части тела документа, которую он представляет. Между открывающим и закрывающим тегами `<div>` может находиться все, что угодно: меню, текст, ссылки, сценарии, формы и т. д. Другими словами, ключевое слово `div` всего лишь указывает на некую

составляющую тела документа, например на ячейку в таблице, однако ничего не говорит о том, что это за составляющая, каково ее назначение и что находится внутри нее.

Для пользователей такие подсказки не слишком важны, но для браузеров правильная интерпретация содержимого обрабатываемого документа может иметь критическое значение. После революции в области портативных устройств и возникновения новых способов подключения к Сети возможность распознавания всех частей документа обрела особую значимость.

Именно поэтому в HTML5 появились новые элементы, помогающие определить каждую часть документа и упорядочить его тело. В HTML5 документ делится на несколько важных разделов, и основная структура больше не зависит от тегов `<div>` и `<table>`.

Как использовать эти новые элементы — решает сам разработчик, однако ключевые слова, выбранные для каждого такого элемента, дают подсказку относительно его назначения. Обычно веб-страница или веб-приложение делится на несколько визуальных областей, что улучшает взаимодействие с пользователем и интерактивность. Как мы скоро убедимся, ключевые слова, представляющие каждый из новых элементов HTML5, тесно связаны с этими визуальными областями.

Макет страницы

На рис. 1.1 показан обычный макет страницы, на основе которого построено большинство современных веб-сайтов. Несмотря на то что все дизайнеры создают собственные шаблоны страниц, почти любой веб-сайт можно разбить на следующие основные разделы.

В верхней области, обозначенной «Заголовок», обычно находится логотип, название, подзаголовок и краткое описание веб-сайта или веб-страницы.

Под этой областью находится панель навигации, на которую почти все разработчики помещают меню или ссылки для перемещения по сайту. С помощью панели навигации пользователи переходят к различным страницам или документам, обычно в пределах одного веб-сайта.

Основное содержимое страницы размещается, как правило, в середине макета. В этой области представлена самая важная информация и ссылки. Чаще всего ее делят на несколько строк и столбцов. На рис. 1.1 вы видите только два столбца, «Основная информация» и «Боковая панель», но данная область является очень гибкой, и дизайнеры настраивают ее

Рис. 1.1. Визуальное представление типичного макета веб-сайта

в соответствии с поставленными требованиям, добавляя новые строки, разбивая столбцы на блоки меньшего размера и создавая разнообразные комбинации и варианты размещения информации. Содержимое этой части макета, как правило, имеет наибольший приоритет. В нашем примере в области «Основная информация» могли бы содержаться список статей, описания продуктов, записи блога и любые другие сведения. В область «Боковая панель» можно было бы поместить список ссылок на каждый из этих элементов. Например, на странице блога здесь обычно находится список ссылок на каждую запись, на страницу с информацией об авторе и т. п.

В нижней части типичного макета можно увидеть одну или несколько панелей со служебной информацией. Эта область называется служебной, так как она чаще всего содержит общие сведения о веб-сайте, его авторе или компании-владельце, а также здесь можно найти ссылки на правила и условия, карты и любые дополнительные данные, которые разработчик

посчитал необходимым указать. Панель со служебной информацией дополняет заголовок и считается неотъемлемой составляющей структуры страницы.

Рис. 1.2. Визуальное представление типичного макета блога

На рис. 1.2 приведен пример обычного блога. Здесь вы можете легко определить все составляющие дизайна, перечисленные ранее:

1. **Заголовок**.
2. **Панель навигации**.
3. **Основная информация**.
4. **Боковая панель**.
5. **Служебная информация** или нижний колонтитул.

Это простое представление блога помогает понять, что у каждого раздела веб-сайта есть определенное назначение. Не всегда деление так очевидно, но перечисленные разделы можно обнаружить на каждом веб-сайте.

В HTML5 учитывается эта базовая структура и макет, и для каждого из разделов существуют новые элементы, позволяющие объявлять и различать их. Теперь мы можем указать браузеру, для чего предназначены разделы страницы.

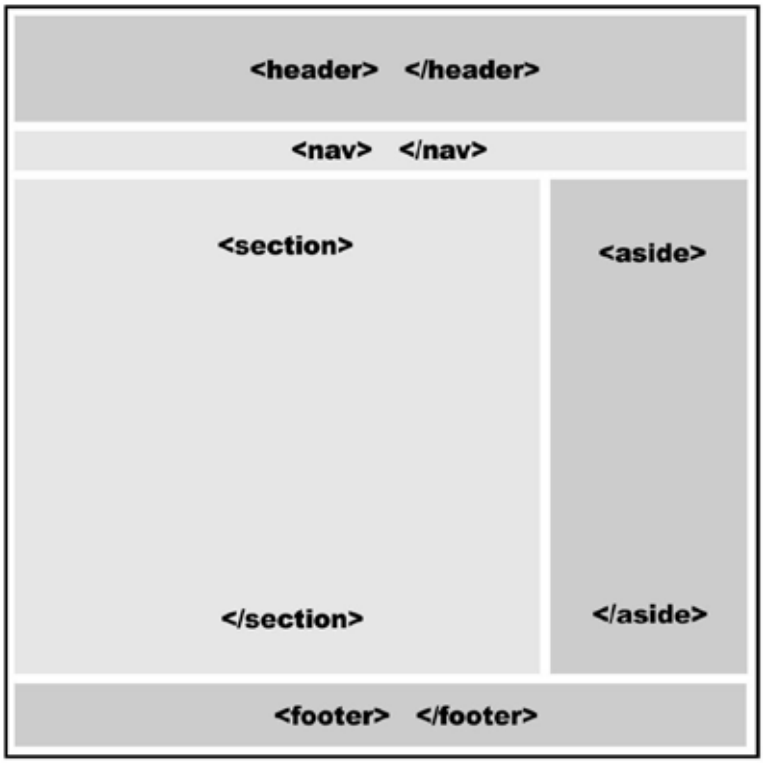

Рис. 1.3. Визуальное представление структуры макета с помощью тегов HTML5

На рис. 1.3 показан тот же типичный макет, который мы использовали ранее, но здесь все разделы обозначены с помощью соответствующих элементов HTML5 (приведены как открывающие, так и закрывающие теги).

<header>

Один из новых элементов, появившихся в HTML5, — `<header>`. Не путайте `<header>` с тегом `<head>`, о котором мы говорили ранее и который описывает «голову» документа. Аналогично `<head>`, тег `<header>` содержит вводную информацию (такую как заголовки, подзаголовки или

логотипы), однако области применения этих двух тегов различаются. Тег <head> предназначен для хранения информации обо всем документе, тогда как <header> используется только для тела документа или для его разделов.

Листинг 1.10. Использование элемента <header>

```
<!DOCTYPE html>
<html lang="ru">
<head>
  <meta charset="utf-8">
  <meta name="description" content="Это пример HTML5">
  <meta name="keywords" content="HTML5, CSS3, JavaScript">
  <title>Этот текст — заголовок документа</title>
  <link rel="stylesheet" href="mystyles.css">
</head>

<body>
  <header>
    <h1>Это главный заголовок веб-сайта</h1>
  </header>

</body>
</html>
```

ВНИМАНИЕ

Если вы выполняли задания с самого начала главы, то у вас уже должен быть готов к тестированию текстовый файл, содержащий все рассмотренные элементы кода. Если же это не так, просто скопируйте код из листинга 1.10 в пустой текстовый файл, использовав какой-нибудь текстовый редактор (например, Блокнот в Windows), и сохраните полученный документ под любым именем с расширением .html. Чтобы проверить результат выполнения кода, откройте этот файл в браузере, совместимом с HTML5. Это можно сделать прямо из браузера, выбрав пункт меню File (Файл), или дважды щелкнуть на нужном файле в файловом менеджере.

В листинге 1.10 с помощью тега <header> мы определяем заголовок веб-страницы. Не забывайте, что этот заголовок и общий заголовок,

определенный ранее в «голове» документа, — это разные вещи. Тег `<header>` указывает на начало основного содержимого документа, его видимой части. Начиная с этого тега, мы будем видеть результаты нашего кода в окне браузера.

ПОВТОРЯЕМ ОСНОВЫ

Между тегами <header> в листинге 1.10 используется тег, который вам может быть незнаком. Тег <h1> — это старый элемент из спецификации HTML, определяющий заголовок. Цифра в названии тега указывает на уровень заголовка. Элемент <h1> определяет самый важный заголовок, а <h6> — наименее значимый. Таким образом, <h1> используется для отображения главного заголовка страницы, а с помощью остальных можно описывать подзаголовки документа. Позже мы увидим, как эти элементы работают в HTML5.

<nav>

Следующий раздел нашего примера — это панель навигации. В HTML5 для ее описания используется тег `<nav>`.

Листинг 1.11. Использование элемента <nav>

```
<!DOCTYPE html>
<html lang="ru">
<head>
  <meta charset="utf-8">
  <meta name="description" content="Это пример HTML5">
  <meta name="keywords" content="HTML5, CSS3, JavaScript">
  <title>Этот текст — заголовок документа</title>
  <link rel="stylesheet" href="mystyles.css">
</head>

<body>
  <header>
    <h1>Это основной заголовок веб-страницы</h1>
  </header>
  <nav>
    <ul>
```

```
      <li>домой</li>
      <li>фотографии</li>
      <li>видео</li>
      <li>контакты</li>
    </ul>
  </nav>

</body>
</html>
```

Как вы видите в листинге 1.11, элемент `<nav>` находится между тегами `<body>`, после закрывающего тега заголовка `</header>`, но не между тегами `<header>`. Смысл в том, что `<nav>` — это не часть заголовка, а отдельный раздел.

Мы уже говорили ранее, что структура и порядок использования элементов в HTML5 определяются разработчиком. HTML5 — очень гибкий язык, он всего лишь предоставляет параметры и базовые элементы, а как их применять, решаем мы сами. Например, тег `<nav>` можно было бы поместить внутрь элемента `<header>` или в любой другой раздел тела документа. Однако необходимо учитывать, что эти новые теги создавались для того, чтобы предоставлять браузерам больше информации и помогать любым новым программам и устройствам распознавать наиболее важные части документа. Для создания понятного и переносимого кода мы рекомендуем следовать стандартам и писать чистый код, без лишних хитростей. Элемент `<nav>` предназначен для определения навигационных элементов, таких как главное меню или основные панели навигации. Используйте его только для этих целей.

ПОВТОРЯЕМ ОСНОВЫ

В листинге 1.11 мы создали список пунктов меню для нашей веб-страницы. Между тегами <nav> используются два элемента для создания списка. Элемент <ul> определяет сам список. Между открывающим и закрывающим тегами <ul> вы видите несколько тегов <li> с различным текстом, который представляет собой пункты нашего меню. Как вы уже догадались, теги <li> применяются для определения элементов списка. В этой книге мы не ставим себе целью научить вас основам HTML. Если вам необходима более подробная информация о стандартных элементах языка, зайдите на наш веб-сайт и просмотрите ссылки для этой главы.

<section>

Далее в нашем типовом дизайне нужно определить разделы, обозначенные на рис. 1.1 как «Основная информация» и «Боковая панель». Как уже говорилось ранее, в области основной информации выводится главное содержимое документа. Эта область может принимать самые разные формы, например, ее можно поделить на несколько блоков или столбцов. Поскольку эти блоки и столбцы не имеют конкретного назначения, то элемент HTML5 для их определения имеет общее название `<section>` (раздел).

Листинг 1.12. Использование элемента <section>

```
<!DOCTYPE html>
<html lang="ru">
<head>
  <meta charset="utf-8">
  <meta name="description" content="Это пример HTML5">
  <meta name="keywords" content="HTML5, CSS3, JavaScript">
  <title>Этот текст — заголовок документа</title>
  <link rel="stylesheet" href="mystyles.css">
</head>
<body>
  <header>
    <h1>Это главный заголовок веб-сайта</h1>
  </header>
  <nav>
    <ul>
      <li>домой</li>
      <li>фотографии</li>
      <li>видео</li>
      <li>контакты</li>
    </ul>
  </nav>
  <section>

  </section>

</body>
</html>
```

Как и панель навигации, область основной информации представляет собой еще один независимый раздел документа. Поэтому он определяется после закрывающего тега `</nav>`.

САМОСТОЯТЕЛЬНО

Сравните код в листинге 1.12 и макет на рис. 1.3 и попробуйте понять, в каком порядке расположены теги в коде и каким разделам они соответствуют в визуальном представлении веб-страницы.

ВНИМАНИЕ

Теги, определяющие разделы документа, в коде следуют один за другим, но на веб-странице некоторые из них могут располагаться рядом, а не один под другим (например, «Основная информация» и «Боковая панель»). В HTML5 визуальное отображение элементов на экране определяется с помощью CSS. Каждому элементу назначается свой CSS-стиль. Изучением CSS мы займемся в следующей главе.

<aside>

В типичном макете веб-сайта (см. рис. 1.1) область под названием «Боковая панель» находится сбоку от основной информации. Данные в этой области обычно связаны с основной информацией страницы, но не так важны.

В примере стандартного макета блога боковая панель содержит список ссылок (см. рис. 1.2, область с номером 4). Ссылки указывают на каждую запись блога, а также на страницу с дополнительной информацией об авторе. Таким образом, содержимое боковой панели связано с основной информацией страницы, но само по себе не имеет никакого значения. Кроме того, можно сказать, что главной информацией являются записи блога, а ссылки и краткие анонсы этих записей представляют собой лишь средства навигации, а не то, что будет интересовать пользователя в первую очередь.

В HTML5 можно обозначать такую вспомогательную информацию с помощью элемента `<aside>` (листинг 1.13).

Листинг 1.13. Использование элемента <aside>

```
<!DOCTYPE html>
<html lang="ru">
<head>
  <meta charset="utf-8">
  <meta name="description" content="Это пример HTML5">
  <meta name="keywords" content="HTML5, CSS3, JavaScript">
  <title>Этот текст — заголовок документа</title>
  <link rel="stylesheet" href="mystyles.css">
</head>
<body>
  <header>
    <h1>Это главный заголовок веб-сайта</h1>
  </header>
  <nav>
    <ul>
      <li>домой</li>
      <li>фотографии</li>
      <li>видео</li>
      <li>контакты</li>
    </ul>
  </nav>
  <section>

  </section>
  <aside>
    <blockquote>Статья номер 1</blockquote>
    <blockquote>Статья номер 2</blockquote>
  </aside>

</body>
</html>
```

Элемент `<aside>` может располагаться на странице справа или слева, он не имеет конкретного местоположения. Этот тег описывает всего лишь заключенную в него информацию, а не ее место в структуре документа. Таким образом, элемент `<aside>` можно добавлять в любую область макета и использовать для любого содержимого, не относящегося к основной информации веб-страницы. Например, элемент `<aside>` можно поместить внутрь элемента `<section>` или даже внутрь основной информации документа (один из способов оформления цитат).

<footer>

Для завершения шаблона и структуры документа HTML5 нам осталось добавить только один элемент. У нас уже есть заголовок тела документа, панель навигации и раздел основной информации, а также боковая панель со вспомогательной информацией. Завершающий элемент должен визуально закончить дизайн и обозначить конец тела документа. В HTML5 для этого используется специальный тег `<footer>` (листинг 1.14).

Листинг 1.14. Использование элемента <footer>

```
<!DOCTYPE html>
<html lang="ru">
<head>
  <meta charset="utf-8">
  <meta name="description" content="Это пример HTML5">
  <meta name="keywords" content="HTML5, CSS3, JavaScript">
  <title>Этот текст — заголовок документа</title>
  <link rel="stylesheet" href="mystyles.css">
</head>
<body>
  <header>
    <h1>Это главный заголовок веб-сайта</h1>
  </header>
  <nav>
    <ul>
      <li>домой</li>
      <li>фотографии</li>
      <li>видео</li>
      <li>контакты</li>
    </ul>
  </nav>
  <section>

  </section>
  <aside>
    <blockquote>Статья номер 1</blockquote>
    <blockquote>Статья номер 2</blockquote>
  </aside>
  <footer>
```

продолжение ⇩

Листинг 1.14 *(продолжение)*

```
    Copyright &copy; 2010-2011
  </footer>

</body>
</html>
```

В типичном макете веб-страницы (см. рис. 1.1) для описания раздела «Служебная информация» используется элемент `<footer>`. С его помощью мы определяем «подвал» нашего документа, в котором, как правило, содержатся общие сведения об авторе или компании, владеющей проектом, а также авторское право, условия использования и т. д.

Обычно элемент `<footer>` добавляется в конец документа и выполняет функцию, описанную ранее. Однако его можно использовать в теле документа несколько раз — в конце разных разделов (точно так же в документе можно неоднократно использовать тег `<header>`). Но об этом мы поговорим чуть позже.

Строение тела документа

Описание тела нашего документа готово. Базовая структура определена, но необходимо еще поработать над содержанием. Пока мы изучили только элементы HTML5, помогающие определять разделы макета и указывать их назначение, однако все самое важное находится внутри этих разделов.

Большинство уже рассмотренных элементов нужны для определения структуры HTML-документа, которую смогут распознать любые браузеры и новые устройства. Мы узнали, что с помощью тега `<body>` объявляется тело видимой части документа, тег `<header>` включает в себя важную информацию о теле, тег `<nav>` определяет средства навигации, тег `<section>` описывает содержимое самого документа, а в теги `<aside>` и `<footer>` заключается вспомогательная информация. Но ни один из указанных элементов не объявляет непосредственно содержимое документа. Все они относятся исключительно к описанию структуры документа.

Чем дальше мы углубляемся в разбор документа, тем ближе подходим к определению его содержимого. Информация в документе может включать в себя разные визуальные элементы, такие как заголовки,

тексты, изображения, видео и интерактивные приложения. Нам необходимо различать эти элементы и устанавливать взаимоотношения между ними.

<article>

Приведенный на рис. 1.1 макет представляет собой простую и обобщенную структуру современных веб-сайтов, а также показывает, каким образом основное содержимое документа располагается на экране. Так же, как блог делится на записи, содержимое веб-сайтов чаще всего делится на фрагменты, обладающие схожими характеристиками. Определить каждый из этих фрагментов помогает элемент `<article>` (статья).

Листинг 1.15. Использование элемента <article>

```
<!DOCTYPE html>
<html lang="ru">
<head>
  <meta charset="utf-8">
  <meta name="description" content="Это пример HTML5">
  <meta name="keywords" content="HTML5, CSS3, JavaScript">
  <title>Этот текст — заголовок документа</title>
  <link rel="stylesheet" href="mystyles.css">
</head>
<body>
  <header>
    <h1>Это главный заголовок веб-сайта</h1>
  </header>
  <nav>
    <ul>
      <li>домой</li>
      <li>фотографии</li>
      <li>видео</li>
      <li>контакты</li>
    </ul>
  </nav>
  <section>
    <article>
      Это текст моей первой статьи
    </article>
```

продолжение ⇩

Листинг 1.15 *(продолжение)*

```
    <article>
      Этот текст моей второй статьи
    </article>
  </section>
  <aside>
    <blockquote>Статья номер 1</blockquote>
    <blockquote>Статья номер 2</blockquote>
  </aside>
  <footer>
    Copyright &copy; 2010-2011
  </footer>

</body>
</html>
```

Как вы видите в коде листинга 1.15, теги `<article>` располагаются между тегами `<section>` — статьи принадлежат разделу, частью которого они являются. Можно сказать, что тег `<article>` — дочерний элемент тега `<section>`, так же как все элементы внутри тега `<body>` являются дочерними элементами тела документа. Как и все дочерние элементы тела документа, теги `<article>` следуют один за другим, так как они независимы друг от друга, что и показано на рис. 1.4.

> **ПОВТОРЯЕМ ОСНОВЫ**
>
> *Как уже говорилось, документ HTML имеет древовидную структуру, корнем которой является элемент <html>. Элемент структуры может быть предком, потомком или братом другого элемента, в зависимости от того, какое место относительно друг друга они занимают в дереве. Например, в обычном HTML-документе элемент <body> является потомком элемента <html> и братом элемента <head>. Для обоих элементов, <body> и <head>, элемент <html> является родителем или предком.*

Название элемента `<article>` никак не ограничивает его применение, то есть не обязательно описывать с помощью него только новостные статьи. Элементы `<article>` могут описывать любые независимые части содержимого документа: публикации на форуме, статьи в журнале, записи блога, комментарии пользователей и т. п. Данный элемент всего лишь

группирует связанные друг с другом фрагменты информации, независимо от характера этой информации.

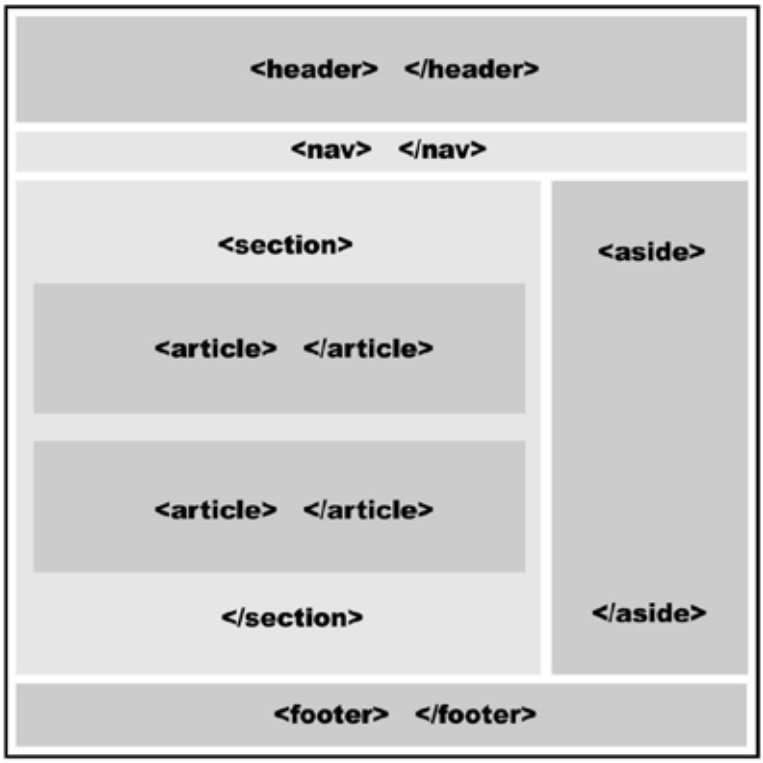

Рис. 1.4. Визуальное представление тегов <article> внутри основного раздела веб-страницы

Как любая независимая часть документа, содержимое каждого элемента `<article>` обладает собственной структурой. При определении этой структуры мы можем пользоваться преимуществами, которые дает нам универсальность тегов `<header>` и `<footer>`, рассмотренных ранее. Это переносимые теги, и их можно использовать не только в теле документа, но и внутри любого его раздела.

Листинг 1.16. Построение структуры элемента <article>

```
<!DOCTYPE html>
<html lang="ru">
<head>
```

продолжение ⇩

Листинг 1.16 *(продолжение)*

```
    <meta charset="utf-8">
    <meta name="description" content="Это пример HTML5">
    <meta name="keywords" content="HTML5, CSS3, JavaScript">
    <title>Этот текст — заголовок документа</title>
    <link rel="stylesheet" href="mystyles.css">
  </head>
  <body>
    <header>
      <h1>Это главный заголовок веб-сайта</h1>
    </header>
    <nav>
      <ul>
        <li>домой</li>
        <li>фотографии</li>
        <li>видео</li>
        <li>контакты</li>
      </ul>
    </nav>
    <section>
      <article>
        <header>
          <h1>Заголовок статьи 1</h1>
        </header>
        Это текст моей первой статьи
        <footer>
          <p>комментарии (0)</p>
        </footer>
      </article>
      <article>
        <header>
          <h1>Заголовок статьи 2</h1>
        </header>
        Это текст моей второй статьи
        <footer>
          <p>комментарии (0)</p>
        </footer>
      </article>
    </section>
    <aside>
      <blockquote>Статья номер 1</blockquote>
```

```
    <blockquote>Статья номер 2</blockquote>
  </aside>
  <footer>
    Copyright &copy; 2010-2011
  </footer>

</body>
</html>
```

Обе статьи в листинге 1.16 описаны с помощью элемента `<article>`, и у каждой статьи задана своя структура. Вначале определен тег `<header>` с заголовком `<h1>`. Далее идет основное содержимое, то есть текст статьи. Завершает каждую статью тег `<footer>` с указанием количества комментариев.

<hgroup>

Внутри каждого элемента `<header>` — в начале тела документа и в начале каждой статьи — мы использовали тег `<h1>` для описания заголовка. В принципе, тега `<h1>` достаточно для создания заголовка части документа. Но иногда нам нужно добавить подзаголовок или другую вспомогательную информацию, чтобы пояснить назначение веб-страницы или какого-то ее раздела. Очень удобно, что элемент `<header>` может содержать другие элементы, например: оглавление, формы поиска, короткие текстовые фрагменты и логотипы.

Для создания заголовков мы можем использовать теги `<h>`: `<h1>`, `<h2>`, `<h3>`, `<h4>`, `<h5>` и `<h6>`. Однако для ускорения обработки документа и для того, чтобы во время его интерпретации не создавались множественные разделы и подразделы, эти теги необходимо группировать. Для этого в HTML5 используется элемент `<hgroup>`.

Листинг 1.17. Использование элемента <hgroup>

```
<!DOCTYPE html>
<html lang="ru">
<head>
  <meta charset="utf-8">
  <meta name="description" content="Это пример HTML5">
  <meta name="keywords" content="HTML5, CSS3, JavaScript">
  <title>Этот текст — заголовок документа</title>
```

продолжение ⇩

Листинг 1.17 *(продолжение)*

```
    <link rel="stylesheet" href="mystyles.css">
  </head>
  <body>
    <header>
      <h1>Это главный заголовок веб-сайта</h1>
    </header>
    <nav>
      <ul>
        <li>домой</li>
        <li>фотографии</li>
        <li>видео</li>
        <li>контакты</li>
      </ul>
    </nav>
    <section>
      <article>
        <header>
          <hgroup>
            <h1>Заголовок статьи 1</h1>
            <h2>подзаголовок статьи 1</h2>
          </hgroup>
          <p>опубликовано 10.12.2011</p>
        </header>
        Это текст моей первой статьи
        <footer>
          <p>комментарии (0)</p>
        </footer>
      </article>
      <article>
        <header>
          <hgroup>
            <h1>Заголовок статьи 2</h1>
            <h2>подзаголовок статьи 2</h2>
          </hgroup>
          <p>опубликовано 15.12.2011</p>
        </header>
        Это текст моей второй статьи
        <footer>
          <p>комментарии (0)</p>
        </footer>
      </article>
```

```
  </section>
  <aside>
    <blockquote>Статья номер 1</blockquote>
    <blockquote>Статья номер 2</blockquote>
  </aside>
  <footer>
    Copyright &copy; 2010-2011
  </footer>

</body>
</html>
```

Необходимо соблюдать иерархию тегов `<h>`, то есть сначала должен быть объявлен тег `<h1>`, затем для подзаголовка тег `<h2>` и т. д. Но в отличие от предыдущих версий HTML, в HTML5 теги `<h>` можно использовать в каждом разделе и заново строить такую иерархию. В листинге 1.17 мы добавили подзаголовок и метаданные в каждую статью и сгруппировали заголовок и подзаголовок с помощью `<hgroup>`. Иерархия тегов `<h1>` и `<h2>` используется в каждом элементе `<article>`.

ВНИМАНИЕ

Элемент <hgroup> следует добавлять только в том случае, если в одном теге <header> есть заголовки разных уровней. Данный элемент может содержать только теги <h> — вот почему в нашем примере метаданные находятся за его пределами. Если в вашем коде используется только тег <h1> или тег <h1> с метаданными, то эти элементы группировать не нужно. Например, в заголовке тела документа элемент <hgroup> отсутствует, потому что элемент <h> только один. Элемент <hgroup> предназначен исключительно для группировки тегов <h>, на что и указывает его название.

Браузеры и программы, которые исполняют и отображают веб-сайты, считывают HTML-код и создают собственную внутреннюю структуру для интерпретации и обработки каждого элемента. Эта внутренняя структура состоит из разделов, не имеющих ничего общего с разделами документа, которые мы определили в HTML-коде, например, с помощью элементов `<section>`. Речь идет о концептуальных разделах, генерируемых во время интерпретации кода. Элемент `<header>` сам по себе не создает отдельного концептуального раздела. Это означает, что элементы внутри `<header>`, представляющие разные уровни заголовков, могут

привести к формированию нескольких концептуальных разделов. Элемент <hgroup> был создан специально для группировки тегов <h>, чтобы избежать ошибок интерпретации HTML-кода браузерами.

ПОВТОРЯЕМ ОСНОВЫ

Метаданные — это набор данных, который описывает другой набор данных и предоставляет дополнительную информацию о нем. В нашем примере метаданные содержат дату публикации статей.

<figure> и <figcaption>

Элемент <figure> предназначен для более точного определения содержимого документа. До его появления невозможно было объявить содержимое, которое представляет собой изолированную частью документа: иллюстрации, рисунки, видео и т. п.Как правило, такие элементы являются частью основного раздела, но их можно спокойно удалять из документа, не нарушая его структуру. Если такая информация присутствует в документе, то она определяется тегом <figure>.

Листинг 1.18. Использование элементов <figure> и <figcaption>

```
<!DOCTYPE html>
<html lang="ru">
<head>
  <meta charset="utf-8">
  <meta name="description" content="Это пример HTML5">
  <meta name="keywords" content="HTML5, CSS3, JavaScript">
  <title>Этот текст — заголовок документа</title>
  <link rel="stylesheet" href="mystyles.css">
</head>
<body>
  <header>
    <h1>Это главный заголовок веб-сайта</h1>
  </header>
  <nav>
    <ul>
      <li>домой</li>
      <li>фотографии</li>
      <li>видео</li>
```

```
    <li>контакты</li>
  </ul>
</nav>
<section>
  <article>
    <header>
      <hgroup>
        <h1>Заголовок статьи 1</h1>
        <h2>подзаголовок статьи 1</h2>
      </hgroup>
      <p>опубликовано 10.12.2011</p>
    </header>
    Это текст моей первой статьи
    <figure>
      <img src="http://minkbooks.com/content/myimage.jpg">
      <figcaption>
        Это изображение для первой статьи
      </figcaption>
    </figure>
    <footer>
      <p>комментарии (0)</p>
    </footer>
  </article>
  <article>
    <header>
      <hgroup>
        <h1>Заголовок статьи 2</h1>
        <h2>подзаголовок статьи 2</h2>
      </hgroup>
      <p>опубликовано 15.12.2011</p>
    </header>
    Это текст моей второй статьи
    <footer>
      <p>комментарии (0)</p>
    </footer>
  </article>
</section>
<aside>
  <blockquote>Статья номер 1</blockquote>
  <blockquote>Статья номер 2</blockquote>
</aside>
```

продолжение ⇩

Листинг 1.18 *(продолжение)*

```
    <footer>
      Copyright &copy; 2010-2011
    </footer>

</body>
</html>
```

В листинге 1.18 в первой статье сразу после текста мы добавили изображение (`<img src="http://minkbooks.com/content/myimage.jpg">`). Это распространенный прием, текст часто сопровождается изображениями или видео. Теги `<figure>` определяют эти визуальные дополнения и помогают отличать их от остальной информации документа.

Кроме того, в листинге 1.18 внутри тега `<figure>` используется еще один дополнительный элемент. Такие блоки информации, как изображения и видео, принято подписывать. В HTML5 предусмотрен специальный элемент для создания таких подписей. Тег `<figcaption>` определяет текст, относящийся к содержимому `<figure>`, и устанавливает отношение между этими элементами и их содержимым.

Новые и старые элементы

Целью разработки HTML5 было упрощение, уточнение и организация кода. Для этого были добавлены новые теги и атрибуты, а сам язык HTML объединен с CSS и JavaScript. Но такие усовершенствования не только привели к появлению новых элементов, они коснулись и уже существующих.

<mark>

Тег `<mark>` был добавлен для подсветки фрагмента текста, который изначально ничем не выделялся, но из-за текущей активности пользователя обрел определенную значимость. Лучшим примером использования данного тега является отображение результатов поиска. С помощью элемента `<mark>` на странице подсвечиваются слова, совпадающие со строкой поиска.

Листинг 1.19. Подсветка слова «car» с помощью элемента <mark>

```
<span>My <mark>car</mark> is red</span>
```

Если пользователь выполнит поиск по слову «car», то для оформления результата поиска можно применить код, показанный в листинге 1.19. Этот короткий код выводит результат поиска, а тег `<mark>` выделяет текст, который использовался в качестве строки поиска (слово «car»). В некоторых браузерах это слово по умолчанию будет подсвечиваться желтым фоном, но вы можете переопределить стиль подсветки с помощью CSS, и мы научимся делать это в последующих главах.

В предыдущих версиях для выделения текста обычно использовался элемент `<b>`. Однако появление элемента `<mark>` изменило назначение и определило новые способы использования этого и других похожих элементов:

- `<em>`. Следует применять для акцентирования фрагмента текста (вместо тега `<i>`, который использовался раньше);
- `<strong>`. Подчеркивает значимость текста (браузеры выделяют такой текст полужирным шрифтом);
- `<mark>`. Подсвечивает текст, имеющий особое значение при сложившихся обстоятельствах;
- `<b>`. Рекомендуется использовать только в тех случаях, когда нет другого, более подходящего элемента.

<small>

Новая специфика HTML проявляется особенно явно на таких элементах, как `<small>`. Раньше данный элемент предназначался для вывода текста шрифтом меньшего размера. Это ключевое слово относилось только к размеру текста, без учета его смысла. В HTML5 у элемента `<small>` появилось новое назначение — вывод дополнительной информации, которая обычно пишется мелким шрифтом, например юридическая информация, отказ от ответственности и т. п. (листинг 1.20).

Листинг 1.20. Юридическая информация, оформленная с помощью элемента <small>

```
<small>Copyright &copy; 2011 MinkBooks</small>
```

<cite>

Еще один элемент, изменивший свое назначение на еще более специфичное в HTML5, — `<cite>`. Теперь тегом `<cite>` следует выделять названия источников, таких как книги, фильмы, песни и т. п. (листинг 1.21).

Листинг 1.21. Цитирование фильма с помощью элемента <cite>

```
<span>I love the movie <cite>Temptations</cite></span>
```

<address>

Элемент `<address>` существует уже очень давно, но теперь он стал структурным. Мы не стали использовать его в нашем шаблоне, но в некоторых ситуациях он отлично подходит для оформления контактной информации внутри элемента `<article>` или всего `<body>`.

Этот элемент можно помещать внутрь элемента `<footer>`, как в листинге 1.22.

Листинг 1.22. Добавление контактной информации в раздел <article>

```
<article>
  <header>
    <h1>Заголовок статьи 2</h1>
  </header>
  Это текст статьи
  <footer>
    <address>
      <a href="http://www.jdgauchat.com">JD Gauchat</a>
    </address>

  </footer>
</article>
```

<time>

В оба раздела `<article>` нашего шаблона из листинга 1.18 мы добавили дату публикации статьи. Для этого мы использовали простой элемент `<p>` внутри элемента `<header>`. Однако для данной цели есть специаль-

ный элемент HTML5. Элемент `<time>` позволяет объявить дату в машинном формате, а также добавить читабельный текст с указанием даты и времени.

Листинг 1.23. Отображение даты с помощью элемента <time>

```
<article>
  <header>
    <h1>Заголовок статьи 1</h1>
    <time datetime="2011-10-12" pubdate>опубликовано 10.12.2011</time>
  </header>
  Это текст статьи
</article>
```

В листинге 1.23 для вывода даты публикации статьи мы заменили элемент `<p>` из предыдущих примеров новым элементом `<time>`. Значение атрибута `datetime` представляет собой временную отметку в машинном формате, который должен соответствовать следующему шаблону: `2011-10-12T12:10:45`. Мы также добавили атрибут `pubdate`, который всего лишь сообщает, что значение атрибута `datetime` определяет дату публикации.

Краткий справочник. Документы HTML5

В спецификации HTML5 ответственность за структуру документа лежит на HTML, и для этих целей в языке HTML появился целый набор новых элементов. Кроме того, есть также несколько элементов, используемых исключительно для оформления. Далее приведен список наиболее важных, по нашему мнению, элементов:

> **ВНИМАНИЕ**
>
> Чтобы ознакомиться с полным справочником элементов HTML, включенных в новую спецификацию, зайдите на наш веб-сайт и просмотрите ссылки для этой главы.

- `<header>`. Определяет вводную информацию и может использоваться в разных разделах документа. Он содержит заголовки раздела, но может также представлять алфавитные указатели, формы поиска, логотипы и т. п.;

- `<nav>`. Содержит набор ссылок для навигации, таких как меню или алфавитные указатели. Не все ссылки на веб-странице следует помещать внутрь элемента `<nav>` — только те, которые относятся к главной панели навигации;
- `<section>`. Представляет собой простой, не специализированный раздел документа. Обычно его используют для построения нескольких блоков (например, столбцов), чтобы группировать содержимое по темам, например, главы или страницы книги, группа новостных статей, набор продуктов и т. п.;
- `<aside>`. Определяет вспомогательную информацию, связанную с основным содержимым страницы. Это могут быть цитаты, сведения на боковой панели, реклама и т. п.;
- `<footer>`. Представляет дополнительную информацию, относящуюся к содержимому родительского элемента. Например, в «подвале» тела документа может находиться вспомогательная информация о документе, так же как и в обычном колонтитуле страницы. Данный элемент можно использовать не только внутри тега `<body>`, но и в других разделах документа. В таком случае он предоставляет сведения, относящиеся только к конкретному разделу;
- `<article>`. Представляет самостоятельный фрагмент информации, например, отдельную статью в газете или запись в блоге. Элементы `<article>` можно вкладывать друг в друга и использовать для отображения списков внутри другого, более крупного списка связанных элементов. Например, так можно оформлять пользовательские комментарии к записи блога;
- `<hgroup>`. Используется для группировки элементов `<h>` в случае многоуровневого заголовка, например, основного заголовка и подзаголовка;
- `<figure>`. Представляет независимый фрагмент содержимого (например, изображение, диаграмму или видео), на который ссылается основное содержимое страницы. Этот независимый блок можно легко удалить из основного содержимого, не нарушив структуру документа;
- `<figcaption>`. Предназначен для отображения подписи или легенды, относящейся к элементу `<figure>`. Например, это может быть подпись к фотографии;
- `<mark>`. Подсвечивает текст, который приобретает особую значимость в какой-то конкретной ситуации или в ответ на действия пользователя;
- `<small>`. Определяет служебную информацию, выводимую мелким шрифтом (например, отказ от ответственности, юридические ограничения, сведения об авторском праве);

- `<cite>`. Представляет названия произведений (книг, фильмов, стихотворений и т. д.);
- `<address>`. Содержит контактную информацию для раздела `<article>` или для всего документа. Его необходимо вставлять внутрь элемента `<footer>`;
- `<time>`. Необходим для отображения даты и времени в форматах, удобных для восприятия как компьютером, так и человеком. Метка в формате, предназначенном для человека, помещается между тегами, а метка в машинном формате устанавливается в качестве значения атрибута `datetime`. Второй необязательный атрибут под названием `pubdate` указывает, что значение атрибута `datetime` представляет собой дату публикации соответствующего содержимого.

2 Стили CSS и блочные модели

CSS и HTML

Как мы уже объясняли, новая спецификация HTML охватывает не только теги, она выходит за пределы обычного кода HTML. Сеть предъявляет высокие требования не только к структурной организации и определению разделов, но и к дизайну и функциональности. В этой новой парадигме HTML объединяется с CSS и JavaScript в один интегрированный инструмент. Мы уже рассмотрели функции каждой из этих технологий и изучили новые элементы HTML, отвечающие за структуру документа. Настало время взглянуть на роль CSS в этом стратегическом союзе и узнать, в какой мере визуализация HTML-документов зависит от данной технологии.

Официально технология CSS никак не связана с HTML5. Она не является и никогда не была частью спецификации HTML5. В действительности это вспомогательная технология, которая разрабатывалась с целью преодоления ограничений и уменьшения сложности HTML. Первоначально некие базовые стили связывались с каждым элементом с помощью атрибутов в тегах HTML, однако по мере того, как язык развивался, код становилось все сложнее разрабатывать и поддерживать, и вскоре оказалось, что одного HTML недостаточно для удовлетворения всех требований веб-дизайнеров. В результате на вооружение была взята технология CSS, позволяющая отделить структуру от представления. С тех пор CSS

успешно развивалась, однако разработка данной технологии шла параллельно HTML и фокусировалась на нуждах дизайнеров, а не на необходимости поддерживать эволюцию HTML.

Третья версия CSS следует по аналогичному пути, однако ее разработчики принимают намного больше компромиссных решений. Спецификация HTML5 подразумевает, что за дизайн теперь отвечает CSS, и из-за этого интеграция между HTML и CSS3 стала критически важным элементом веб-разработки. Вот почему всегда, когда мы говорим о HTML5, мы также упоминаем CSS3. Это естественно, несмотря на то что официально данные технологии независимы.

В настоящее время возможности CSS3 внедряются и реализуются в браузерах, совместимых с HTML5, наряду с остальной функциональностью, описанной в спецификации. В этой главе мы изучим базовые концепции CSS и новые техники CSS3, уже доступные для структурирования и представления веб-страниц. Мы также узнаем о новых селекторах и псевдоклассах, упрощающих выбор и идентификацию элементов HTML.

ПОВТОРЯЕМ ОСНОВЫ

CSS — это язык, работающий совместно с HTML. Он связывает с элементами документа разнообразные визуальные стили, определяющие их размер, цвет, фон, рамки и т. п.

ВНИМАНИЕ

Сейчас возможности CSS3 уже встроены в последние версии большинства популярных браузеров, однако некоторые из них все еще находятся на стадии разработки. По этой причине для обеспечения их эффективной работы в названиях новых стилей необходимо использовать браузерные префиксы, такие как moz или webkit (в зависимости от механизма используемого браузера). Об этой тонкости мы поговорим чуть позже в данной главе.

Стили и структура

Каждый браузер по умолчанию связывает определенные стили с элементами HTML, однако эти стили не всегда соответствуют ожиданиям

дизайнера. Если честно, в большинстве случаев они чрезвычайно далеки от того, что нам хотелось бы видеть на наших веб-сайтах. Дизайнерам и разработчикам зачастую приходится применять собственные стили, чтобы добиться желаемого оформления и организации данных на экране.

ВНИМАНИЕ

В этом разделе главы мы познакомимся со стилями CSS и изучим базовые техники определения структуры документа. Если эти концепции вам знакомы, можете пропустить первый раздел и сразу перейти к новому материалу.

Блочные элементы

Что касается структуры, почти все браузеры по умолчанию располагают элементы, ориентируясь на их тип: блочный или строчной. Данная классификация определяет, как элементы выстраиваются на экране:

- блочные элементы (block element) располагаются на странице один под другим;
- строчные элементы (inline element) располагаются бок о бок на одной строке, и разрыв строки не вставляется до тех пор, пока на экране хватает пространства по горизонтали.

Практически каждый структурный элемент документа по умолчанию считается блочным. Это означает, что каждый тег HTML, представляющий собой некий компонент визуальной организации (например, `<section>`, `<nav>`, `<header>`, `<footer>`, `<div>`), будет помещаться ниже предыдущего.

В главе 1 мы создали HTML-документ, повторяющий макет типичного веб-сайта. Дизайн предусматривает горизонтальные полосы и два столбца в центре. Однако из-за использования правил отображения, принятых в браузерах по умолчанию, визуальный результат оказывается весьма далеким от ожидаемого. Открыв HTML-файл с кодом из листинга 1.18 в браузере, вы заметите, что два столбца, описанные с помощью тегов `<section>` и `<aside>`, располагаются на экране неправильно: один под другим, а не бок о бок, как мы планировали. По умолчанию каждый блок визуализируется максимальной ширины, а его высота соответствует содержащейся в нем информации. Схематическое представление результата, в котором блоки отображаются вертикально, показано на рис. 2.1.

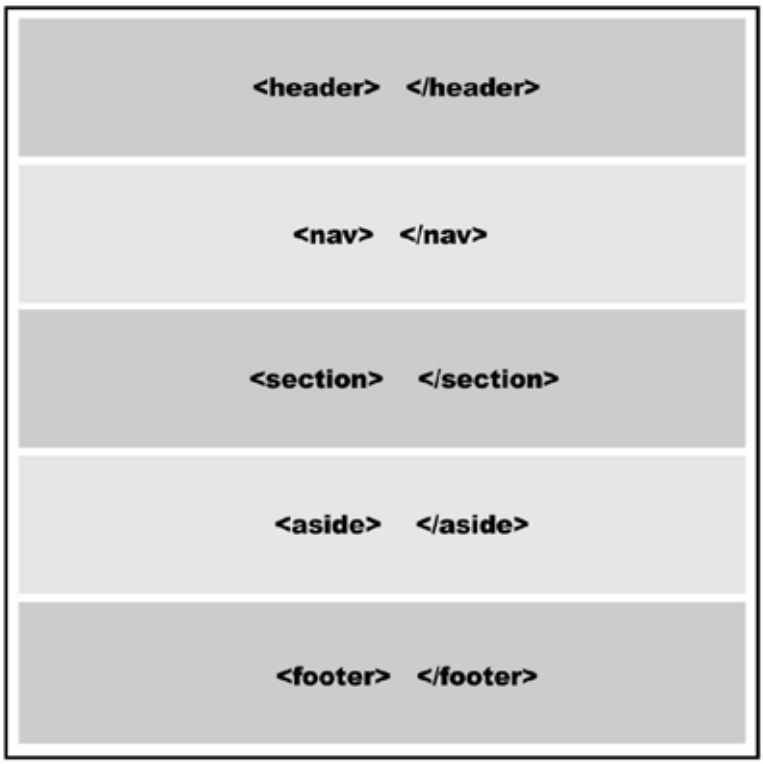

Рис. 2.1. Визуальное представление макета страницы, созданное с использованием стилей по умолчанию

Блочные модели

Для того чтобы научиться создавать собственные хорошие макеты, необходимо сначала разобраться, каким образом браузеры обрабатывают код HTML. Каждый элемент HTML браузеры считают отдельным блоком. Веб-страница, фактически, представляет собой группу блоков, собранную воедино в соответствии с некоторыми правилами. Эти правила определяются стилями — либо встроенными в браузеры, либо поставляемыми дизайнерами в форме кода CSS.

В CSS существует предустановленный набор свойств, позволяющий переопределять стили браузеров и создавать желаемые варианты дизайна. Это не какие-то узкоспециализированные свойства: их можно и нужно объединять для формирования правил, на основе которых блоки и будут группироваться, определяя правильный дизайн. Комбинацию таких

правил обычно называют моделью или системой макета. Все вместе правила определяют блочную модель.

Сегодня только одна блочная модель считается стандартной, кроме нее существует еще несколько экспериментальных. Допустимая и повсеместно используемая модель называется традиционной блочной моделью (Traditional Box Model), и она существует со времен первой версии CSS. Несмотря на доказанную эффективность данной модели, разрабатываются экспериментальные модели, нацеленные на преодоление ее недостатков. Наиболее значимой среди них является новая гибкая блочная модель (Flexible Box Model), появившаяся в CSS3 и полагаемая частью HTML5.

Основы применения стилей

Прежде чем начинать кодировать правила CSS в собственных таблицах стилей и экспериментировать с блочными моделями, следует вспомнить базовые концепции применения стилей CSS, к которым мы будем обращаться до конца книги.

ПОВТОРЯЕМ ОСНОВЫ

В этой книге вы найдете лишь краткое введение в стили CSS. Мы упоминаем только те техники и свойства, которые необходимы для понимания тем, изучаемых в книге, и чтения представленных кодов. Если вы никогда раньше не работали с CSS, посетите наш веб-сайт и ознакомьтесь с ресурсами, перечисленными в ссылках к данной главе.

САМОСТОЯТЕЛЬНО

Создайте пустой текстовый файл и копируйте туда HTML-код из всех следующих листингов. Обновляйте этот файл в браузере для проверки работы кода. Обратите внимание: для того чтобы файл обрабатывался браузером правильно, его необходимо сохранить с расширением .html.

Применение стилей к элементам HTML меняет способ их отображения на экране. Как уже говорилось, браузеры предоставляют набор стилей по

умолчанию, которого в большинстве случаев оказывается недостаточно для реализации замыслов дизайнеров. Чтобы исправить ситуацию, можно переопределить стандартные стили собственными стилями, применив для этого различные техники.

Строчные стили

Одна из простейших техник определения стилей внутри элемента включает в себя использование атрибутов. В листинге 2.1 представлен код простого HTML-документа, в котором элемент `<p>` модифицируется атрибутом `style`, имеющим значение `font-size: 20px`. Атрибут `style` меняет размер по умолчанию для текста внутри элемента `<p>`, увеличивая его до 20 пикселов.

Листинг 2.1. Стили CSS внутри тегов HTML

```
<!DOCTYPE html>
<html lang="ru">
<head>
  <title>Этот текст — заголовок документа</title>
</head>
<body>
  <p style="font-size: 20px">Мой текст</p>
</body>
</html>
```

Описанная техника — отличный способ тестирования стилей и быстрой проверки результата, однако ее не рекомендуется применять для всего документа. Причина проста: данная техника подразумевает, что стиль нужно описать и повторить для каждого отдельного элемента, из-за чего документ раздувается до недопустимого размера, а поддерживать и обновлять его становится невозможно. Представьте всего лишь, что вам понадобится изменить размер текста во всех элементах `<p>`: вместо 20 пикселов установить 24 пиксела. Вам придется отредактировать каждый стиль в каждом теге `<p>` во всем документе.

Встроенные стили

Намного лучшая альтернатива — вставлять стили в «голову» документа, а затем ссылаться на них в соответствующих элементах HTML.

Листинг 2.2. Стили перечисляются в «голове» документа

```
<!DOCTYPE html>
<html lang="ru">
<head>
  <title>Этот текст — заголовок документа</title>
  <style>
    p { font-size: 20px }
  </style>
</head>
<body>
  <p>Мой текст</p>
</body>
</html>
```

Элемент `<style>`, показанный в листинге 2.2, позволяет авторам вставлять стили CSS в код документа. В предыдущих версиях HTML необходимо было указывать, какой тип стилей будет использоваться. В HTML5 стилем по умолчанию считается CSS, следовательно, никакие другие атрибуты к открывающему тегу `<style>` добавлять не нужно.

Выделенный в листинге 2.2 жирным шрифтом код выполняет ту же функцию, что и выделенная строка кода в листинге 2.1, однако в листинге 2.2 нам не пришлось описывать стиль внутри каждого тега `<p>` документа, так как общее определение в начале кода распространяется на все элементы `<p>`. Благодаря этой технике можно сокращать объем кода и назначать желаемые стили определенным элементам с помощью ссылок (их мы изучим чуть позже в этой главе).

Внешние файлы

Объявление стилей в «голове» документа экономит пространство и делает код более единообразным, а его поддержку — удобной, однако требует создания копии стилей в каждом документе веб-сайта. Гораздо лучшее решение — переместить все стили во внешний файл. После этого с помощью элемента `<link>` данный файл можно будет вставить в любой документ, требующий применения стилей. Этот метод также позволяет быстро поменять весь набор стилей, всего лишь добавив ссылку на другой файл. Кроме того, становится проще модифицировать документы и адаптировать их к различным условиям и устройствам — мы научимся делать это в конце книги.

В главе 1 мы познакомились с тегом `<link>` и научились вставлять файлы CSS в наши документы. В строке кода `<link rel="stylesheet" href="mystyles.css">` мы приказываем браузеру загрузить файл mystyles.css, так как он содержит все стили, необходимые для визуализации страницы. Этот прием широко практикуют дизайнеры, уже применяющие в своей работе HTML5. Тег `<link>` со ссылкой на файл CSS вставляется во все документы, в которых требуются стили (листинг 2.3).

Листинг 2.3. Применение стилей CSS из внешнего файла

```
<!DOCTYPE html>
<html lang="ru">
<head>
  <title>Этот текст — заголовок документа</title>
  <link rel="stylesheet" href="mystyles.css">
</head>
<body>
  <p>Мой текст</p>
</body>
</html>
```

САМОСТОЯТЕЛЬНО

Начиная с этого момента, мы будем добавлять стили CSS в файл под названием mystyles.css. Создайте этот файл в той же папке, где находится ваш файл HTML, и копируйте в него стили CSS, чтобы на экране проверять, как они работают.

ПОВТОРЯЕМ ОСНОВЫ

Файлы CSS представляют собой обычные текстовые файлы. Как и файлы HTML, их можно создавать в любых текстовых редакторах, например в Блокноте в Windows.

Ссылки

Собирать все стили в одном внешнем файле и вставлять этот файл во все нужные документы довольно удобно. Однако нам также нужен механизм

установления определенного взаимоотношения между этими стилями и элементами документа, на которые они должны влиять.

Когда мы обсуждали встраивание стилей в документ, то познакомились с одной из техник, которая часто используется в CSS для создания ссылок на элементы HTML. В листинге 2.2 стиль, меняющий размер шрифта, благодаря ключевому слову `p` ссылался на все элементы `<p>`. Таким образом, стиль, описанный между тегами `<style>`, ссылался на каждый тег `<p>` в документе и связывал с ним определенный стиль CSS.

Существуют и другие способы выбора элементов HTML, которые должны меняться под влиянием правила CSS:

- по ключевому слову элемента;
- атрибуту `id`;
- атрибуту `class`.

Однако, как мы увидим далее, спецификация CSS3 довольно гибкая в этом отношении и включает в себя новые, более точные способы определения ссылок на элементы HTML.

Ссылка по ключевому слову

Объявление правила CSS с упоминанием ключевого слова элемента позволяет изменить все соответствующие элементы в документе. Например, следующее правило (листинг 2.4) меняет стиль элементов `<p>`.

Листинг 2.4. Ссылка по ключевому слову

```
p { font-size: 20px }
```

Мы уже познакомились с этой техникой в листинге 2.2. Указывая ключевое слово `p` перед правилом, мы сообщаем браузеру, что данное правило следует применить к каждому элементу `<p>`, который обнаружится в документе HTML. Теперь весь текст, окруженный тегами `<p>`, будет отображаться шрифтом нового размера — 20 пикселов.

Разумеется, то же самое можно сделать для любого другого элемента HTML в документе. Если, например, вместо `p` указать ключевое слово `span`, то размер шрифта любого текста, заключенного между тегами `<span>`, изменится со стандартного на 20 пикселов (листинг 2.5).

Листинг 2.5. Ссылка по другому ключевому слову

```
span { font-size: 20px }
```

Но что делать, если перед нами стоит задача сослаться только на один определенный тег? Нужно ли использовать атрибут `style` внутри этого тега? Ответ — нет. Как мы узнали раньше, метод строчных стилей (использование атрибута `style` прямо внутри тегов HTML) устарел, и его следует избегать. Для выбора определенного элемента HTML в правилах файла CSS можно пользоваться двумя следующими атрибутами: `id` и `class`.

Ссылка по атрибуту id

Атрибут `id` — это что-то вроде имени, идентификатор элемента. Это означает, что значение данного атрибута не может повторяться в документе. Во всем документе имя элемента должно быть уникальным.

Для того чтобы сослаться из файла CSS на определенный элемент по его атрибуту `id`, нужно объявить правило с символом `#` перед значением идентификатора.

Листинг 2.6. Ссылка по значению атрибута id

```
#text1 { font-size: 20px }
```

Правило из листинга 2.6 будет применяться только к элементу HTML, идентифицируемому по атрибуту `id="text1"`. Наш код HTML будет выглядеть, как показано в листинге 2.7.

Листинг 2.7. Идентифицируем элемент <p> по его атрибуту id

```
<!DOCTYPE html>
<html lang="ru">
<head>
  <title>Этот текст – идентификатор документа</title>
  <link rel="stylesheet" href="mystyles.css">
</head>
<body>
  <p id="text1">Мой текст</p>
</body>
</html>
```

Теперь любая ссылка в файле CSS, содержащая идентификатор `text1`, будет влиять только на элемент с этим значением идентификатора. Остальные элементы `<p>`, так же как и прочие элементы в документе, меняться не будут.

Это очень точный способ определения ссылок на элементы, и он обычно используется для элементов общего назначения, таких как структурные теги. В действительности атрибут `id` в силу своей специфичности больше подходит для использования в коде JavaScript, как мы увидим в следующих главах.

Ссылка по атрибуту class

Более предпочтительный вариант, чем использование атрибута `id`, — ссылаться при определении стилей элементов на атрибут `class`. Это более гибкий атрибут, который можно связать со всеми содержащимися в документе элементами HTML, требующими одинакового дизайна (листинг 2.8).

Листинг 2.8. Ссылка по значению атрибута class

```
.text1 { font-size: 20px }
```

Для работы с атрибутом `class` необходимо объявить правило, название которого начинается с точки. Преимущество данного подхода состоит в том, что добавления атрибута `class` со значением `text1` будет достаточно для назначения нужного стиля всем требующим этого элементам.

Листинг 2.9. Назначение стилей с помощью атрибута class

```
<!DOCTYPE html>
<html lang="ru">
<head>
  <title>Этот текст — заголовок документа</title>
  <link rel="stylesheet" href="mystyles.css">
</head>
<body>
  <p class="text1">Мой текст</p>
  <p class="text1">Мой текст</p>
  <p>Мой текст</p>
</body>
</html>
```

У элементов <p> в первых двух строках тела листинга 2.9 есть атрибут class со значением text1. Как уже упоминалось, один и тот же класс можно связать с разными элементами одного документа. Таким образом, первые два элемента <p> относятся к одному классу и, следовательно, оформляются с использованием стиля из листинга 2.8. К последнему элементу <p> применяются стили по умолчанию.

Смысл добавления точки в начале имени класса заключается в том, что одно название класса можно связывать с разными элементами и назначать каждому из них собственный стиль.

Листинг 2.10. Определение ссылки по значению атрибута class только на элементы <p>

```
p.text1 { font-size: 20px }
```

В листинге 2.10 мы создали правило, ссылающееся на класс под названием text1, но предназначенное исключительно для элементов <p>. Даже если атрибут class любого другого элемента будет содержать то же значение, к этому элементу данный стиль применяться не будет.

Ссылка по любому атрибуту

Описанных способов обращения к элементам HTML-документа достаточно для обработки большинства ситуаций, однако иногда возникает необходимость найти определенный элемент и поменять стиль только для него. В последних версиях CSS появились новые способы определения ссылок на элементы HTML. Один из них подразумевает использование селектора атрибутов. Теперь сослаться на документ можно не только по id или class, но и по любому другому его атрибуту.

Листинг 2.11. Определение ссылки только на элементы <p> и только имеющие атрибут name

```
p[name] { font-size: 20px }
```

Правило из листинга 2.11 меняет стиль только элементов <p> и только имеющих атрибут name. Для имитации описанных ранее решений, включающих в себя атрибуты id и class, нужно добавить значение атрибута (листинг 2.12).

Листинг 2.12. Определение ссылки на элементы <p>, имеющие атрибут name со значением mytext

```
p[name="mytext"] { font-size: 20px }
```

CSS3 позволяет комбинировать символ равенства (=) с другими, определяя еще более детальные правила выбора (листинг 2.13).

Листинг 2.13. Новые селекторы в CSS3

```
p[name^="my"] { font-size: 20px }
p[name$="my"] { font-size: 20px }
p[name*="my"] { font-size: 20px }
```

Если вы уже знакомы с концепцией регулярных выражений (например, после изучения других языков, таких как JavaScript и PHP), то узнаете селекторы, использованные в листинге 2.13. В CSS3 эти селекторы дают следующие результаты:

- правило с селектором ^= применяется ко всем элементам <p>, имеющим атрибут name, значение которого начинается с my (например, mytext, mycar);
- правило с селектором $= применяется ко всем элементам <p>, имеющим атрибут name, значение которого заканчивается на my (например, textmy, carmy);
- правило с селектором *= применяется ко всем элементам <p>, имеющим атрибут name, значение которого содержит подстроку my (в данном случае подстрока может находиться в середине значения, например, textmycar).

В этих примерах мы использовали элемент <p>, атрибут name и случайный текст «my», однако этот же прием можно применять с любыми подходящими для вашей задачи атрибутами и значениями. Просто запишите в квадратных скобках название атрибута и желаемое значение —так вы сможете определить ссылку на любой элемент HTML.

Определение ссылок по псевдоклассам

В CSS3 появились также новые псевдоклассы, еще больше увеличивающие точность выбора элементов.

Листинг 2.14. Шаблон для тестирования псевдоклассов

```
<!DOCTYPE html>
<html lang="ru">
<head>
  <title>Этот текст — заголовок документа</title>
  <link rel="stylesheet" href="mystyles.css">
</head>
<body>
  <div id="wrapper">
    <p class="mytext1">Мой текст1</p>
    <p class="mytext2">Мой текст2</p>
    <p class="mytext3">Мой текст3</p>
    <p class="mytext4">Мой текст4</p>
  </div>
</body>
</html>
```

Давайте рассмотрим новый HTML-код, использованный в листинге 2.14. Здесь определяются четыре элемента `<p>`, которые в структуре данного документа являются братьями и потомками одного и того же элемента `<div>`.

Благодаря псевдоклассам мы можем воспользоваться преимуществами такой организации и сослаться на конкретный элемент, ничего не зная о его атрибутах и их значениях.

Листинг 2.15. Псевдокласс :nth-child()

```
p:nth-child(2)
  { background: #999999;
}
```

Псевдокласс определяется двоеточием, которое находится между ссылкой и названием псевдокласса. В правиле из листинга 2.15 мы ссылаемся на элементы `<p>`. Данное правило можно изменить следующим образом: `.myclass:nth-child(2)` — и обратиться к каждому элементу, являющемуся потомком другого элемента, значение атрибута `class` которого равно `myclass`. Допускается присоединение псевдоклассов к любым типам ссылок, которые мы изучили ранее.

Псевдокласс `nth_child()` позволяет найти определенного потомка. Как уже говорилось, в документе HTML из листинга 2.14 присутствуют

четыре элемента <p>, и в данной структуре документа они братья. Это означает, что у них общий предок — элемент . В действительности данный псевдокласс означает что-то вроде «выбрать потомка под номером...», то есть число в круглых скобках представляет собой позицию потомка или его индекс. Правило из листинга 2.15 ссылается на все элементы <p> в документе, являющиеся вторыми потомками своих предков.

САМОСТОЯТЕЛЬНО

В своем файле HTML замените предыдущий код кодом из листинга 2.14 и откройте этот файл в браузере. Добавьте правила из листинга 2.15 в файл mystyles.css и протестируйте этот пример.

Разумеется, применив данный ссылочный метод, можно выбрать любого потомка, просто поменяв индекс в описании псевдокласса. Например, следующее правило (листинг 2.16) распространяется только на последний элемент <p> в нашем шаблоне.

Листинг 2.16. Псевдокласс nth-child()

```
p:nth-child(4){
  background: #999999;
}
```

Как вы, вероятно, уже догадались, стили можно связать со всеми элементами, создав для каждого собственное правило.

Листинг 2.17. Создание списка с псевдоклассом nth-child()

```
*{
  margin: 0px;
}
p:nth-child(1){
  background: #999999;
}
p:nth-child(2){
  background: #CCCCCC;
}
p:nth-child(3){
  background: #999999;
}
```

```
p:nth-child(4){
  background: #CCCCCC;
}
```

В первом правиле в листинге 2.17 используется универсальный селектор * (звездочка), позволяющий связать один и тот же стиль со всеми элементами в документе. Этот новый селектор представляет каждый элемент в теле документа и полезен при определении некоторых базовых правил. В данном случае мы для каждого элемента определяем ширину поля 0 пикселов, для того чтобы избежать незаполненного пространства и пустых строк, которые обычно возникают по умолчанию вокруг элементов `<p>`.

САМОСТОЯТЕЛЬНО

Скопируйте последний пример кода в CSS-файл и откройте HTML-документ в браузере, чтобы проверить результат.

Дальше в листинге 2.17 несколько раз упоминается псевдокласс `nth-child()` — в этом перечислении мы создаем меню или список вариантов, хорошо различимых на экране за счет разноцветных строк.

Чтобы добавить в меню еще несколько пунктов, необходимо вставить в HTML-код новые элементы `<p>`, а в таблицу стилей — новые правила с псевдоклассом `nth-child()` и правильным индексом. Однако такой подход означает создание огромного объема кода, к тому же его невозможно применять на веб-сайтах с динамическим генерированием содержимого. Для того чтобы получить тот же результат более эффективным способом, нужно применить в определении псевдокласса ключевые слова `odd` и `even` (листинг 2.18).

Листинг 2.18. Использование ключевых слов odd и even

```
*{
  margin: 0px;
}
p:nth-child(odd){
  background: #999999;
}
p:nth-child(even){
  background: #CCCCCC;
}
```

Теперь весь список описывается с помощью всего двух правил. Даже если мы добавим в него множество новых вариантов или пунктов, стили к ним будут применяться автоматически согласно порядковому номеру. Ключевое слово `odd` в определении псевдокласса `nth-child()`влияет на элементы `<p>`, являющиеся потомками другого элемента и находящиеся на нечетной позиции, а ключевое слово `even` — на элементы, занимающие четные позиции.

Существуют и другие важные псевдоклассы с аналогичной функциональностью. Часть из них добавлена совсем недавно, например `first-child`, `last-child` и `only-child`. Псевдокласс `first-child` ссылается только на первого потомка, `last-child` — только на последнего, а `only-child` относится к элементу, который для своего предка является единственным потомком. Эти псевдоклассы не требуют ключевых слов или дополнительных параметров, а пример их использования вы найдете в следующем фрагменте кода (листинг 2.19).

Листинг 2.19. Использование псевдокласса last-child для модифицирования только последнего элемента <p> в списке

```
*{
  margin: 0px;
}
p:last-child{
  background: #999999;
}
```

Еще один важный псевдокласс позволяет описать отрицание — это псевдокласс `not()`.

Листинг 2.20. Применение стилей ко всем элементам, за исключением <p>

```
:not(p){
  margin: 0px;
}
```

Правило, использованное в листинге 2.20, создает поле шириной 0 пикселов вокруг всех элементов в документе, за исключением `<p>`. В отличие от универсального селектора, который мы использовали ранее, псевдокласс `not()` позволяет задать исключение. Стили в правилах, содержащих данный псевдокласс, назначаются всем элементам HTML-документа, кроме элемента, указанного в скобках.

Вместо ключевого слова элемента можно также указывать ссылку. Например, правило из листинга 2.21 применяется ко всем элементам, кроме тех, значение атрибута `class` которых равно `mytext2`.

Листинг 2.21. Определение исключения для атрибута class

```
:not(.mytext2){
  margin: 0px;
}
```

Если применить последнее правило к HTML-коду из листинга 2.14, то браузер свяжет с элементом `<p>`, атрибут `class` которого равен `mytext2`, стили по умолчанию, а поля остальных элементов `<p>` будут уменьшены до 0 пикселов.

Новые селекторы

Есть еще несколько селекторов, добавленных в CSS3 или считающихся частью этой спецификации, которые могут оказаться весьма полезными при создании дизайна. В этих селекторах с помощью символов `>`, `+` и `~` описываются отношения между двумя элементами.

Листинг 2.22. Селектор >

```
div > p.mytext2{
  color: #990000;
}
```

Селектор `>` указывает, что правило применяется ко второму элементу при условии, что он является потомком первого. Правило, использованное в листинге 2.22, модифицирует элементы `<p>`, для которых элемент `<div>` — родительский. В данном случае мы добавили еще одно условие и ссылаемся только на элементы `<p>`, у которых значение атрибута `class` равно `mytext2`.

Листинг 2.23. Селектор +

```
p.mytext2 + p{
  color: #990000;
}
```

Следующий селектор конструируется с использованием символа `+`. Он позволяет сослаться на второй элемент при условии, что прямо перед

ним находится первый элемент. Предок обоих элементов должен быть одним и тем же.

Правило, использованное в листинге 2.23, влияет на элемент `<p>`, следующий за другим элементом `<p>`, который, в свою очередь, идентифицируется классом `mytext2`. Если вы откроете в браузере HTML-файл с кодом из листинга 2.14, то увидите, что красным цветом выделен текст в третьем элементе `<p>`, так как прямо перед ним находится элемент `<p>` с атрибутом `class`, равным `mytext2`.

Для конструирования последнего селектора используется символ `~`. Данный селектор похож на предыдущий, однако элемент, на который распространяется правило, не обязательно должен следовать сразу за первым элементом. Кроме того, элементов может быть несколько.

Листинг 2.24. Селектор ~

```
p.mytext2 ~ p{
  color: #990000;
}
```

Правило, использованное в листинге 2.24, меняет представление третьего и четвертого элементов `<p>` в нашем примере. Стиль применяется ко всем элементам `<p>`, являющимся братьями и расположенным после элемента `<p>` с классом `mytext2`. Если между ними встретятся другие элементы разных типов, это не будет играть роли — все равно стиль поменяется только для третьего и четвертого элементов `<p>`. Поэкспериментируйте с HTML-кодом из листинга 2.14: вставьте элемент `<span></span>` после элемента `<p>` с классом `mytext2`, и вы сможете убедиться в том, что данное правило модифицирует только элементы `<p>`.

Применение таблиц стилей CSS к шаблону

Как говорилось ранее в этой главе, каждый структурный элемент считается блоком и структура документа представляет собой группу блоков. Собранные вместе, блоки определяют то, что называется блочной моделью.

Мы изучим две разные блочные модели: традиционную блочную модель и новую гибкую блочную модель. Традиционная блочная модель используется со времен появления первой версии CSS. В настоящее время ее

поддерживают все браузеры, присутствующие на рынке, и она считается стандартом веб-дизайна. В отличие от нее гибкая блочная модель, включенная в состав CSS3, находится на этапе разработки, однако ее преимущества вполне могут позволить ей занять место отраслевого стандарта, сместив с него традиционную блочную модель. Таким образом, она становится важным предметом изучения.

ВНИМАНИЕ

Представленное далее описание традиционной блочной модели — это не введение в HTML5. Данная модель доступна разработчикам очень давно, и вы, вероятно, уже умеете ее реализовывать. В таком случае можете сразу же перейти к следующему разделу.

Обе модели можно применять к одним и тем же структурам HTML, однако структура документа должна быть подготовлена для использования конкретной модели, и тогда стили будут накладываться правильно. Наша задача — адаптировать документы HTML к выбранной блочной модели.

Традиционная блочная модель

Все началось с таблиц. Это произошло случайно, но таблицы стали теми самыми инструментами, используя которые, разработчики начали создавать и располагать блоки содержимого на экране. Можно сказать, что так появилась первая блочная модель в Сети. Блоки создавались путем расширения ячеек, объединения строк, столбцов и целых таблиц и даже посредством вкладывания одной таблицы в другую. Когда веб-сайты стали больше и сложнее, эта практика превратилась в серьезную проблему: объем кода, необходимого для создания подходящих таблиц, сильно разрастался, из-за чего возникали трудности с его поддержкой.

Стремление решить эту проблему дало жизнь тому, что сегодня воспринимается как совершенно естественное явление, — отделению структуры от представления. Благодаря тегам `<div>` и стилям CSS появилась возможность избавиться от таблиц, в то же время сохранив их функциональность, и эффективно разделить структуру HTML и представление на экране. Используя элементы `<div>` и CSS-код, мы могли создавать на экране блоки, размещая их в любом подходящем месте и определяя нужные размеры, рамки, цвета и т. д. Спецификация CSS предлагала

определенные свойства, помогающие организовывать блоки нужным образом. Эти свойства обладали достаточной мощью для того, чтобы в итоге зародилась целая новая модель — традиционная блочная.

Из-за того что данная модель имела определенные недостатки, таблицы еще некоторое время активно использовались, однако благодаря успеху Ajax и появлению множества новых интерактивных приложений основной тенденцией стало все же использование тегов `<div>` и стилей CSS. Эти две составляющие стали отраслевым стандартом, а традиционная блочная модель получила общее признание.

Шаблон

В главе 1 мы уже построили шаблон документа HTML5. Он включал в себя необходимые элементы для определения структуры нашего документа, однако для того, чтобы подготовить его к стилизации с помощью CSS и применению традиционной блочной модели, требуются некоторые усовершенствования.

Традиционная блочная модель требует наличия оберток вокруг блоков, для того чтобы их можно было размещать горизонтально. Поскольку все содержимое тела документа создается из группы блоков, которые должны быть определенным образом выровнены и подогнаны по размеру, следует всегда добавлять элемент `<div>` в качестве обертки.

Новый шаблон будет выглядеть так.

Листинг 2.25. Новый шаблон HTML5, готовый к добавлению стилей CSS

```
<!DOCTYPE html>
<html lang="ru">
<head>
  <meta charset="utf-8">
  <meta name="description" content="Это пример HTML5">
  <meta name="keywords" content="HTML5, CSS3, JavaScript">
  <title>Этот текст — заголовок документа</title>
  <link rel="stylesheet" href="mystyles.css">
</head>
<body>
<div id="wrapper">
  <header id="main_header">
    <h1>Это главный заголовок веб-сайта</h1>
  </header>
```

```
<nav id="main_menu">
  <ul>
    <li>домой</li>
    <li>фотографии</li>
    <li>видео</li>
    <li>контакты</li>
  </ul>
</nav>
<section id="main_section">
  <article>
    <header>
      <hgroup>
        <h1>Заголовок статьи 1</h1>
        <h2>подзаголовок статьи 1</h2>
      </hgroup>
     <time datetime="2011-12-10" pubdate>опубликовано 10.12.2011</time>
    </header>
    Этот текст моей первой статьи
    <figure>
      <img src="http://minkbooks.com/content/myimage.jpg">
      <figcaption>
        это изображение для первой статьи
      </figcaption>
    </figure>
    <footer>
      <p>комментарии (0)</p>
    </footer>
  </article>
  <article>
    <header>
      <hgroup>
        <h1>Заголовок статьи 2</h1>
        <h2>подзаголовок статьи 2</h2>
      </hgroup>
     <time datetime="2011-12-15" pubdate>опубликовано 15.12.2011</time>
    </header>
    Это текст моей второй статьи
    <footer>
      <p>комментарии (0)</p>
    </footer>
  </article>
```

продолжение ⇩

Листинг 2.25 *(продолжение)*

```
    </section>
    <aside id="main_aside">
      <blockquote>Статья номер один</blockquote>
      <blockquote>Статья номер два</blockquote>
    </aside>
    <footer id="main_footer">
      Copyright &copy 2010-2011
    </footer>
</div>
</body>
</html>
```

В листинге 2.25 содержится новый шаблон, готовый к добавлению стилей. В этом коде можно заметить два важных изменения по сравнению с листингом 1.18. Во-первых, присутствует несколько элементов с атрибутами `id` и `class`. Это означает, что теперь у нас есть возможность сослаться на конкретный элемент по значению его атрибута `id`, а также модифицировать одновременно несколько элементов, указав в правилах CSS значение их атрибута `class`.

Второе важное изменение по сравнению с предыдущим шаблоном заключается в добавлении элемента `<div>`, упомянутого ранее. Этот элемент идентифицируется атрибутом `id="wrapper"`, а в конце тела документа закрывается тегом `</div>`. Данная обертка позволяет связать блочную модель с содержимым тела документа и определить горизонтальное расположение элементов.

САМОСТОЯТЕЛЬНО

Сравните код из листинга 1.18 с кодом из листинга 2.25 и найдите открывающий и закрывающий теги элемента , который мы добавили в качестве обертки. Также проверьте, какие элементы теперь идентифицируются атрибутом id, а какие — атрибутом class. Удостоверьтесь в том, что значения атрибутов id уникальны во всем документе.

Кроме того, вам нужно будет заменить код в созданном ранее HTML-файле кодом из листинга 2.25.

Теперь, когда HTML-документ готов, настало время создать собственную таблицу стилей.

Универсальный селектор *

Начнем с простейших правил, которые обеспечат единообразие дизайна.

Листинг 2.26. Общее правило CSS

```
* {
  margin: 0px;
  padding: 0px;
}
```

Обычно приходится для большинства элементов в документе настраивать размер поля или, по крайней мере, пытаться минимизировать его. Поля некоторых элементов по умолчанию ненулевой ширины и иногда оказываются даже слишком большими. По мере разработки дизайна мы убедимся в том, что нулевая ширина полей лучше всего подходит почти для любого элемента. Для того чтобы избежать необходимости в дублирующихся стилях, здесь удобно применить универсальный селектор * («звездочка»), с которым мы познакомились ранее.

Первое правило для CSS-файла, показанное в листинге 2.26, гарантирует, что у всех элементов поля будут нулевой ширины. Дальше в коде будем модифицировать поля только в том случае, если для конкретного элемента их ширина должна быть больше нуля.

ПОВТОРЯЕМ ОСНОВЫ

Помните, что каждый элемент считается отдельным блоком? Поле элемента — это пространство вокруг данного элемента, то есть лежащее за пределами блока. (Забивка — это пространство вокруг содержимого элемента, но внутри блока, например между заголовком и границей виртуальной рамки, определяемой элементом <h1> для этого заголовка. Мы подробнее поговорим о забивке далее в этой главе.) Размер поля можно определять сразу для всех сторон элемента или для каждой стороны отдельно. Правило margin: 0px в нашей таблице стилей определяет нулевую ширину полей со всех сторон вокруг блока. Если бы мы указали значение 5 пикселов, то блок был бы окружен пустым пространством шириной 5 пикселов, то есть от соседних блоков его всегда отделяло бы поле размером 5 пикселов.

САМОСТОЯТЕЛЬНО

Будем записывать все необходимые правила CSS в файле, содержащем таблицу стилей. Мы уже указали ссылку на этот файл в «голове» HTML-кода в теге <link>, поэтому вам всего лишь нужно создать в любом текстовом редакторе новый пустой файл под названием mystyles.css и скопировать туда правило из листинга 2.26. В дальнейшем вы будете копировать туда правила из следующих листингов.

Новая иерархия заголовков

В нашем шаблоне использованы элементы `<h1>` и `<h2>` для объявления заголовков и подзаголовков различных разделов документа. Стили по умолчанию для этих элементов всегда далеки от того, что нам хотелось бы видеть на экране. Зато, как мы узнали из предыдущей главы, в HTML5 иерархию элементов `H` можно выстраивать заново в каждом разделе. Например, в одном документе элемент `<h1>` может встречаться несколько раз — не только для определения главного заголовка документа, как было ранее, но и во внутренних разделах. Таким образом, необходимо определить подходящие стили для всех нужных уровней заголовка.

Листинг 2.27. Добавление стилей для элементов <h1> и <h2>

```
h1 {
  font: bold 20px verdana, sans-serif;
}
h2 {
  font: bold 14px verdana, sans-serif;
}
```

Свойство `font`, которое в листинге 2.27 мы связываем с элементами `<h1>` и `<h2>`, позволяет задать все текстовые стили в одной строке объявления. С помощью `font` можно объявлять следующие свойства: `font-style`, `font-variant`, `font-weight`, `font-size/line-height` и `font-family` в указанном порядке. В наших правилах мы меняем стиль начертания, размер и семейство шрифта для любого текста, содержащегося внутри элементов `<h1>` и `<h2>`, на желаемые.

Объявление новых элементов HTML5

Еще одно базовое правило, которое необходимо добавить в самом начале, — это определение структурных элементов HTML5 по умолчанию. Некоторые браузеры не распознают их или обрабатывают как строчные элементы. Нужно указать, что данные элементы являются блочными и их следует обрабатывать как теги `<div>`, для того чтобы мы смогли построить нужную блочную модель.

Листинг 2.28. Правило по умолчанию для элементов HTML5

```
header, section, footer, aside, nav, article, figure, figcaption,
hgroup{
  display: block;
}
```

Теперь элементы, на которые распространяется правило из листинга 2.28, будут располагаться один над другим, если только позже мы не объявим другое правило.

Выравнивание тела документа по центру

Первый элемент в блочной модели — это всегда `<body>`. Обычно содержимое данного элемента по тем или иным причинам необходимо размещать горизонтально. Кроме того, чтобы дизайн сохранял единообразие в разных конфигурациях, нужно указать размер или максимальный размер содержимого.

Листинг 2.29. Выравнивание тела документа по центру

```
body {
  text-align: center;
}
```

По умолчанию ширина тега `<body>` составляет 100 %. Это означает, что тело документа растягивается на всю ширину окна браузера. Для того чтобы выровнять страницу по центру экрана, необходимо выровнять по центру содержимое тела документа. Благодаря правилу, добавленному в листинге 2.29, выравниваться по центру экрана будет все, что мы поместим внутрь элемента `<body>`, то есть вся веб-страница.

Создание главного блока

Теперь нам нужно указать максимальный либо фиксированный размер содержимого «тела» документа. Как вы наверняка помните, в листинге 2.25 мы добавили к шаблону элемент `<div>`, обрамляющий все содержимое. Он и будет считаться главным блоком, внутри которого находятся все основные элементы. Таким образом, размер данного блока будет определять максимальный размер остальных элементов.

Листинг 2.30. Определение свойств главного блока

```
#wrapper {
  width: 960px;
  margin: 15px auto;
  text-align: left;
}
```

Правило из листинга 2.30 впервые ссылается на элемент по значению его атрибута `id`. Символ решетки (`#`) говорит браузеру, что данные стили должны применяться только к элементу, у которого есть атрибут `id` со значением `wrapper`.

Это правило содержит три стиля для главного блока. Первый устанавливает фиксированную ширину 960 пикселов. Таким образом, ширина блока всегда будет равна 960 пикселам (ширина современных веб-сайтов чаще всего колеблется между 960 и 980 пикселами, однако значения этого параметра в разные времена могут сильно различаться).

Второй стиль относится к тому, что называется традиционной блочной моделью. В предыдущем правиле (листинг 2.29) мы указали, что содержимое тела документа будет выравниваться по центру, использовав для этого правило `text-align: center`. Однако это распространяется только на строчное содержимое, такое как текст и изображения. Для блочного содержимого, такого как элементы `<div>`, необходимо задавать конкретную ширину поля, обеспечивающую автоматическую подгонку под размер родительского элемента. Для этого предназначено свойство `margin`, поддерживающее четыре значения: ширину верхнего, правого, нижнего и левого полей в указанном порядке. Это означает, что первое указанное значение описывает ширину верхнего поля элемента, второе относится к полю справа и т. д. Однако если записать только первые два параметра, то остальные примут такое же значение. В примере мы делаем именно так.

В листинге 2.30 стиль `margin: 15px auto` задает для верхнего и нижнего полей соответствующего элемента `<div>` ширину 15 пикселов, а для

левого и правого устанавливается автоматическое значение ширины. Таким образом, сверху и снизу тела документа всегда будет оставаться пустое пространство шириной 15 пикселов. В то же время размер пустого пространства справа и слева от этого блока будет рассчитываться автоматически в зависимости от размера тела документа и размера блока `<div>`. В результате содержимое блока будет выравниваться по центру окна браузера.

Итак, веб-страница выровнена по центру, и для нее установлена фиксированная ширина 960 пикселов. Теперь нужно сделать кое-что для предотвращения проблемы, которая возникает во многих браузерах. Свойство `text-align` наследуется иерархически. Это означает, что по центру будут выравниваться все элементы, составляющие тело документа, а не только главный блок. Действие данного стиля будет распространяться на всех потомков элемента `<body>`. Таким образом, необходимо для остального содержимого документа восстановить исходное значение стиля. С этой задачей справляется третье и последнее правило, использованное в листинге 2.30. В результате мы получаем следующую картину: содержимое тела документа выровнено по центру, однако содержимое главного блока (обертки `<div>`) выровнено по левому краю. Следовательно, остальной код наследует данный стиль и также по умолчанию выравнивается по левому краю.

САМОСТОЯТЕЛЬНО

Если вы еще этого не сделали, скопируйте все приведенные ранее правила в пустой файл под названием mystyles.css. Этот файл нужно поместить в ту же папку или каталог, где находится HTML-файл с кодом из листинга 2.25. На этом этапе у вас должно быть два файла: файл с кодом HTML и файл mystyles.css, содержащий все стили CSS, которые мы изучили, начиная с листинга 2.26. Откройте HTML-файл в своем браузере, и вы сможете увидеть созданный блок на экране (в большинстве систем, для того чтобы открыть файл, нужно дважды щелкнуть на его названии).

Заголовок

Перейдем теперь к настройке остальных структурных элементов. За открывающим тегом обертки `<div>` следует первый структурный элемент HTML5, `<header>`. Этот элемент описывает главный заголовок нашей

веб-страницы, который выводится вверху экрана. Для идентификации заголовка в коде используется атрибут `id="main_header"`.

По умолчанию ширина каждого блочного элемента, включая тело документа, составляет 100 %. Это означает, что элемент растягивается по горизонтали, занимая все доступное пространство. Как уже говорилось, для тела документа это вся видимая область экрана, однако для остальных элементов максимальная доступная ширина определяется размером родительского элемента. В нашем примере элементам внутри главного блока выделяется не более 960 пикселов, так как ранее мы установили для главного блока максимальную ширину, равную этому значению.

Листинг 2.31. Добавление стилей для элемента <header>

```
#main_header {
  background: #FFFBB9;
  border: 1px solid #999999;
  padding: 20px;
}
```

Поскольку элемент `<header>` будет занимать в главном блоке все доступное по горизонтали пространство и поскольку он и так считается блочным элементом и располагается вверху страницы, нам остается только связать с ним стили, которые помогут узнать его на экране. В правиле из листинга 2.31 мы создаем для элемента `<header>` желтый фон, сплошную рамку толщиной 1 пиксел, а также внутренние поля шириной 20 пикселов (для этого используется свойство `padding`).

Навигационная полоса

Следом за `<header>` идет очередной структурный элемент, `<nav>`, предназначение которого — обеспечивать навигацию по веб-сайту. Ссылки, сгруппированные внутри этого элемента, представляют собой меню нашего сайта. Оно должно выглядеть как простая полоска под заголовком, так что получается, что почти все нужные стили уже связаны с данным элементом (как и с элементом `<header>`). `<nav>` относится к блочным элементам, следовательно, располагается прямо под предыдущим элементом, его ширина по умолчанию составляет 100 %, поэтому он будет на экране таким же широким, как родительский элемент (обертка `<div>`). Кроме того (и это тоже стиль по умолчанию), его высота будет равна высоте содержимого плюс поля. Таким образом, нам осталось только сделать

его привлекательнее внешне. Для этого добавим серый фон и маленькие внутренние поля, отделяющие пункты меню от рамки.

Листинг 2.32. Добавление стилей для элемента <nav>

```
#main_menu {
  background: #CCCCCC;
  padding: 5px 15px;
}
#main_menu li {
  display: inline-block;
  list-style: none;
  padding: 5px;
  font: bold 14px verdana, sans-serif;
}
```

В листинге 2.32 первое правило ссылается на элемент `<nav>` по его атрибуту `id`. В этом правиле мы меняем цвет фона и с помощью свойства `padding` добавляем внутренние поля шириной 5 и 15 пикселов.

ПОВТОРЯЕМ ОСНОВЫ

Свойство padding работает точно так же, как margin. Можно указать четыре значения: для верхнего, правого, нижнего и левого полей в указанном порядке. Если указано только одно значение, то оно присваивается полям со всех сторон элемента. Если вы задаете два значения, то первое связывается с верхним и нижним полями, а второе — с правым и левым.

Внутри навигационной полосы находится список, созданный на основе тегов `<ul>` и `<li>`. По умолчанию элементы списка располагаются в столбик, один под другим. Чтобы поменять это поведение и вывести пункты меню на одной строке бок о бок, мы ссылаемся на элементы `<li>` данного конкретного тега `<nav>` через селектор `#main_menu li` и связываем с ними стиль `display: inline-block`, превращая их в строчные блоки. В отличие от блочных элементов, элементы, для которых определяется стандартизованный в CSS3 параметр `inline-block`, не создают разрывов строк. Тем не менее их также можно обрабатывать как блоки и объявлять для них значение ширины. Если ширина не задается явно, то этот параметр устанавливает размер элементов равным размеру их содержимого.

Кроме того, мы убрали небольшие рисунки, которые по умолчанию выводятся перед каждым пунктом списка (маркеры списка), применив свойство `list-style`.

Раздел и боковая врезка

Следующие структурные элементы в нашем коде — это два блока, находящиеся по соседству. Традиционная блочная модель строится на базе стилей CSS, позволяющих задать местоположение каждого блока. С помощью свойства `float` блоки можно поместить у правого или левого края экрана в зависимости от того, какие перед нами стоят задачи. Для создания этих блоков в шаблоне HTML использованы теги `<section>` и `<aside>`, а идентифицируются они атрибутом `id` со значениями `main_section` и `main_aside` соответственно.

Листинг 2.33. Создание двух столбцов с помощью свойства float

```
#main_section {
  float: left;
  width: 660px;
  margin: 20px;
}
#main_aside {
  float: left;
  width: 220px;
  margin: 20px 0px;
  padding: 20px;
  background: #CCCCCC;
}
```

Свойство CSS `float` — одно из наиболее часто используемых в реализации традиционной блочной модели. Оно заставляет элемент прикрепляться к одному или другому краю доступного пространства. Элементы, для которых определено свойство `float`, работают, как блочные элементы, — различия определяются лишь значением данного свойства, а не нормальным потоком документа. В зависимости от значения свойства `float` элементы переносятся к самому краю доступной для размещения области — как можно левее или правее.

В правилах из листинга 2.33 мы объявляем позиции и размеры обоих блоков и таким образом создаем на экране столбцы содержимого. Свойство `float` перемещает блок к тому краю доступного пространства, на

который указывает его значение, свойство `width` определяет размер по горизонтали, а `margin` — размер полей элемента.

Что касается стилей, создаваемых нашими правилами, то содержимое элемента `<section>` будет располагаться у левого края экрана. Для него отводится 660 пикселов по ширине плюс 40 пикселов на поля, итого 700 пикселов по горизонтали.

Значение свойства `float`, связанного с элементом `<aside>`, — `left`. Это означает, что данный блок будет располагаться у левого края доступного пространства. Поскольку предыдущий блок, определяемый элементом `<section>`, также отодвигается к левому краю экрана, в данный момент мы рассматриваем только пространство, остающееся справа от него. Новый блок будет находиться в той же строке, что и первый, справа от него. Занимая оставшееся на этой строке место, он будет создавать второй столбец в нашем дизайне.

ПОВТОРЯЕМ ОСНОВЫ

Для получения реального значения складываем размер элемента с шириной его полей. Если ширина элемента равна 200 пикселам, а справа и слева у него есть поля шириной 10 пикселов каждое, то в действительности он займет область шириной 220 пикселов. Общая ширина полей (20 пикселов) прибавляется к ширине элемента (200 пикселов): итоговое значение и показывает, сколько места для него потребуется на экране. То же самое относится к свойствам padding (внутренние поля или забивка) и border (рамка). Добавляя рамку к элементу или определяя пустое пространство между содержимым и рамкой с помощью свойства padding, не забывайте прибавлять эти значения к ширине элемента, для того чтобы понять его реальный размер на экране. Реальная ширина вычисляется по формуле: размер + поля + забивка + рамки.

САМОСТОЯТЕЛЬНО

Вернитесь к листингу 2.25. Пройдитесь по всем созданным до этого момента правилам CSS и найдите в шаблоне элементы HTML, соответствующие каждому из этих правил. Обращайте внимание на ссылки: ключевые слова элементов (например, h1) и атрибуты id (такие, как main_header) помогают понять, как работают ссылки и как стили назначаются различным элементам.

Мы также определили размер второго блока — 220 пикселов, добавили серый фоновый цвет и внутренние поля шириной 20 пикселов. В результате ширина поля будет равна 220 пикселам плюс еще 40, добавленных за счет применения свойства `padding` (внешние поля справа и слева от элемента имеют нулевую ширину).

Нижний колонтитул

Для завершения применения традиционной блочной модели нужно создать и связать с элементом `<footer>` еще одно свойство CSS. Это свойство восстановит нормальную последовательность документа и позволит поместить `<footer>` ровно под последним элементом, а не в стороне.

Листинг 2.34. Добавление стиля для элемента <footer> и восстановление нормальной последовательности документа

```
#main_footer {
  clear: both;
  text-align: center;
  padding: 20px;
  border-top: 2px solid #999999;
}
```

В листинге 2.34 мы создаем вверху элемента `<footer>` рамку шириной 2 пиксела и внутреннее поле (забивку) шириной 20 пикселов. Также выравниваем текст внутри элемента по центру. Кроме того, мы восстанавливаем нормальную последовательность документа с помощью свойства `clear`. Это свойство всего лишь очищает область, занимаемую элементом, и запрещает его отображение сбоку от плавающего блока (блока со свойством `float`). Чаще всего используется значение `both`, означающее, что очищаются обе стороны и теперь документ должен следовать обычному потоку (то есть не должен считаться плавающим). Для блочного элемента это означает, что он отображается ниже последнего элемента на новой строке.

Свойство `clear` выталкивает элементы по вертикали, заставляя их ограничиваться размером реальной области экрана. Без этого свойства браузер визуализирует документ так, словно плавающих элементов не существует, и блоки накладываются друг на друга.

Когда в традиционной блочной модели есть располагающиеся бок о бок блоки, всегда необходимо добавлять после них элемент со стилем `clear: both`, чтобы иметь возможность размещать новые блоки под ними

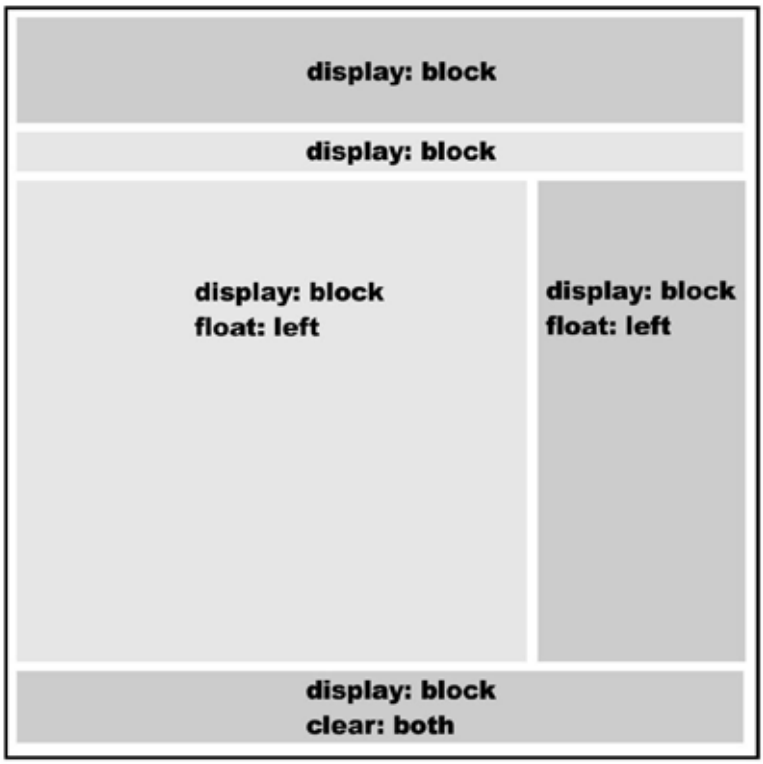

Рис. 2.2. Визуальное представление традиционной блочной модели

в естественном течении документа. На рис. 2.2 показано визуальное представление данной модели и обозначены базовые стили CSS, позволяющие создать такой макет.

Значения `left` и `right` свойства `float` не гарантируют, что блок будет располагаться у левого или правого края окна соответственно. В действительности эти значения всего лишь делают плавающей определенную сторону документа, разрушая нормальный поток документа. Например, если значение свойства равно `left`, то браузер попытается расположить элемент у левого края доступного пространства. Если рядом с предыдущим элементом есть место, то новый элемент окажется на экране справа, так как именно левая сторона у него плавающая. Получается, что он «плывет» налево, пока не натыкается на что-то, что его заблокирует, — это может быть элемент-брат или край родительского элемента. Об этом важно помнить при создании макетов с несколькими столбцами. Каждому столбцу в свойстве `float` присваивается значение `left`, для того чтобы

все столбцы располагались в ряд в правильном порядке. Каждый столбец «плывет» налево до тех пор, пока не натолкнется на другой столбец или границу родительского элемента. Данный метод позволяет расположить блоки рядом друг с другом на одной строке, создав в визуальном представлении на экране набор столбцов.

Последние штрихи

Единственное, над чем нам осталось поработать, — это дизайн содержимого. Для этого определим стили еще нескольких элементов HTML5.

Листинг 2.35. Добавляем последние штрихи к базовому дизайну

```
article {
  background: #FFFBCC;
  border: 1px solid #999999;
  padding: 20px;
  margin-bottom: 15px;
}
article footer {
  text-align: right;
}
time {
  color: #999999;
}
figcaption {
  font: italic 14px verdana, sans-serif;
}
```

Первое правило в листинге 2.35 ссылается на все элементы `<article>` и определяет для них несколько стилей (фоновый цвет, сплошная рамка шириной 1 пиксел, забивка, поле снизу). Нижнее поле шириной 15 пикселов предназначено для того, чтобы визуально отделять статьи друг от друга по вертикали.

Кроме того, в каждой статье есть элемент `<footer>`, показывающий количество полученных комментариев. Для того чтобы сослаться на `<footer>`, находящийся внутри элемента `<article>`, мы использовали селектор `article footer`, означающий, что каждый `<footer>`, находящийся внутри `<article>`, будет оформлен с использованием следующих стилей. Мы применили здесь этот прием, чтобы выровнять текст нижнего колонтитула каждой статьи по правому краю блока.

В конце кода в листинге 2.35 мы поменяли цвет содержимого для всех элементов `<time>` и определили новый стиль для подписи под изображением. Это необходимо для того, чтобы шрифт текста, добавляемого с помощью элемента `<figcaption>`, отличался от шрифта остального текста статьи.

САМОСТОЯТЕЛЬНО

Если вы еще этого не сделали, последовательно скопируйте все приведенные в этой главе правила, начиная с листинга 2.26, и вставьте их в файл mystyles.css. Затем откройте файл HTML, содержащий шаблон из листинга 2.25. Так вы сможете увидеть, как работает традиционная блочная модель и как структурные элементы организованы на экране.

ВНИМАНИЕ

Вы можете загрузить эти фрагменты кода и запустить их выполнение одним щелчком мыши на нашем веб-сайте. Заходите на *http://www.minkbooks.com*.

Свойство box-sizing

В CSS3 встроено еще одно свойство, относящееся к структуре и традиционной блочной модели. Свойство `box-sizing` позволяет менять способ расчета размера элемента: например, можно заставить браузер считать ширину забивки и рамки частью исходного размера элемента.

Как уже говорилось, для вычисления общей ширины области, занимаемой элементом, браузер применяет следующую формулу: размер + поля + забивка + рамки. Таким образом, если значение свойства `width` равно 100 пикселам, `margin` — 20 пикселам, `padding` — 10 пикселам, а `border` — 1 пикселу, то ширина области, занимаемой документом, получается следующей: 100 + 40 + 20 + 2 = 162 пиксела. Обратите внимание на то, что значения `margin`, `padding` и `border` удваиваются, так как необходимо принять во внимание и левую, и правую стороны блока. Следовательно, при каждом объявлении размера элемента с использованием свойства `width` нужно учитывать, что реальная площадь, которая потребуется для размещения элемента, скорее всего, будет больше.

В зависимости от вашего стиля кодирования неплохой идеей может быть внедрение значений `padding` и `border` в значение `width`, для того чтобы формула расчета реальной ширины упрощалась до такой: размер + поля.

Листинг 2.36. Возможность учесть забивку и рамку в исходном размере элемента

```
div {
  width: 100px;
  margin: 20px;
  padding: 10px;
  border: 1px solid #000000;

  -moz-box-sizing: border-box;
  -webkit-box-sizing: border-box;
  box-sizing: border-box;
}
```

Свойство `box-sizing` может принимать два значения. По умолчанию оно равно `content-box` — это означает, что браузеры прибавляют значения `padding` и `border` к размеру, задаваемому свойством `width`. Если же вместо этого указать значение `border-box`, то поведение изменится: забивка и рамка будут учитываться в размере самого элемента.

Листинг 2.36 демонстрирует применение данного свойства к элементу `<div>`. Это всего лишь пример, который мы в нашем шаблоне использовать не будем. Однако он может оказаться полезным для некоторых дизайнеров — это зависит от того, насколько хорошо они знакомы с традиционными методами расчета размера элементов, унаследованными от предыдущих версий CSS.

ВНИМАНИЕ

Сейчас существуют только экспериментальные реализации свойства box-sizing в некоторых браузерах. Для эффективного использования в своих документах данное свойство нужно объявлять с подходящим префиксом. К этой теме мы вернемся чуть позже.

Гибкая блочная модель

Главное назначение блочной модели — обеспечивать механизм разбиения оконного пространства на отдельные блоки и создания строк и столбцов, которые и составляют основу дизайна обычной веб-страницы. Однако традиционная блочная модель, используемая еще со времен первой версии CSS и широко распространенная по сей день, не слишком хорошо справляется с этой задачей. Например, в этой модели невозможно определить расположение блоков и задать их ширину и высоту, не прибегая к помощи разнообразных трюков или сложных наборов правил, которые придумал башковитый парень откуда-то с другого континента.

Сложность создания популярных дизайнерских эффектов (таких, как увеличение ширины столбцов в зависимости от размера свободного пространства, выравнивание содержимого по вертикали, вытягивание столбцов по вертикали независимо от содержимого) заставила разработчиков задуматься о возможности применения к документам каких-то новых моделей. Было разработано несколько образцов, но ни один из них не сумел привлечь к себе столько внимания, как гибкая блочная модель.

Гибкая блочная модель элегантно решает проблемы предыдущей модели. Благодаря новой реализации у нас наконец-то появились блоки, представляющие виртуальные строки и столбцы, которые в действительности больше всего и нужны дизайнерам и пользователям. Теперь мы получили полный контроль над макетом, позициями и размером блоков, распределением блоков внутри других блоков и размещением их в общем оконном пространстве. Код раз и навсегда принял форму, удовлетворяющую требованиям дизайнеров.

ВНИМАНИЕ

Несмотря на то что гибкая блочная модель обладает массой преимуществ по сравнению с предыдущей моделью, она все еще находится на экспериментальной стадии разработки. Еще по меньшей мере несколько лет придется ждать полного внедрения ее в браузеры и процессы создания веб-сайтов и приложений. В настоящее время существует две спецификации, и только одну поддерживают браузеры на базе WebKit и Gecko, такие как Firefox и Google Chrome. Вот почему мы также подробно рассмотрели применение традиционной блочной модели.

В этом разделе мы познакомимся с работой гибкой блочной модели, узнаем, как применить ее к нашему шаблону, а также изучим предоставляемые этой моделью новые возможности.

Одна из главных особенностей данной модели заключается в том, что некоторые возможности (например, ориентация — вертикальная или горизонтальная) объявляются в родительских блоках. Из этого следует, что для правильной организации блоков необходимо одни блоки вставлять в другие. В новой модели у каждого набора блоков обязательно должен быть родительский блок.

Если вы взглянете на шаблон, с которым мы до сих пор работали, то увидите, что несколько родительских блоков там уже определены. Элемент `<body>` и обертку `<div>` можно с успехом превратить в родительские блоки. Однако родительский блок требуется и для еще одной составляющей структуры. Мы добавим новый элемент `<div>` для обрамления поднабора блоков, представляющего два столбца в середине страницы (в коде документа они определяются элементами `<section>` и `<aside>`).

Вот как наш шаблон будет выглядеть после добавления новой обертки.

Листинг 2.37. Добавление родительского блока для обрамления <section> и <aside>

```
<!DOCTYPE html>
<html lang="ru">
<head>
  <meta charset="utf-8">
  <meta name="description" content="Это пример HTML5">
  <meta name="keywords" content="HTML5, CSS3, JavaScript">
  <title>Этот текст — заголовок документа</title>
  <link rel="stylesheet" href="mystyles.css">
</head>
<body>
<div id="wrapper">
  <header id="main_header">
    <h1>Это главный заголовок веб-сайта</h1>
  </header>
  <nav id="main_menu">
    <ul>
      <li>домой</li>
      <li>фотографии</li>
      <li>видео</li>
      <li>контакты</li>
```

```
  </ul>
</nav>
<div id="container">
  <section id="main_section">
    <article>
      <header>
        <hgroup>
          <h1>Заголовок статьи 1</h1>
          <h2>подзаголовок статьи 1</h2>
        </hgroup>
       <time datetime="2011-12-10" pubdate>опубликовано 10.12.2011</time>
      </header>
      Это текст моей первой статьи
      <figure>
        <img src="http://minkbooks.com/content/myimage.jpg">
        <figcaption>
          это изображение для первой статьи
        </figcaption>
      </figure>
      <footer>
        <p>комментарии (0)</p>
      </footer>
    </article>
    <article>
      <header>
        <hgroup>
          <h1>Заголовок статьи 2</h1>
          <h2>подзаголовок статьи 2</h2>
        </hgroup>
       <time datetime="2011-12-15" pubdate>опубликовано 15.12.2011</time>
      </header>
      Это текст моей второй статьи
      <footer>
        <p>комментарии (0)</p>
      </footer>
    </article>
  </section>
  <aside id="main_aside">
    <blockquote>Статья 1</blockquote>
    <blockquote>Статья 2</blockquote>
  </aside>
```

продолжение ⇩

Листинг 2.37 *(продолжение)*

```
    </div>
    <footer id="main_footer">
      Copyright &copy 2010-2011
    </footer>
  </div>
  </body>
  </html>
```

САМОСТОЯТЕЛЬНО

В оставшейся части главы мы будем использовать шаблон из листинга 2.37. Замените шаблон в своем HTML-файле представленным ранее кодом. Также вам понадобится очистить файл mystyles.css, для того чтобы добавить туда правила CSS, которые мы рассмотрим далее.

Прежде чем применять свойства CSS, относящиеся только к гибкой блочной модели, давайте добавим в новую таблицу стилей базовые правила, общие для обеих моделей.

Листинг 2.38. Общие правила CSS для обеих моделей

```
* {
  margin: 0px;
  padding: 0px;
}
h1 {
  font: bold 20px verdana, sans-serif;
}
h2 {
  font: bold 14px verdana, sans-serif;
}
header, section, footer, aside, nav, article, figure, figcaption,
hgroup{
  display: block;
}
```

Здесь происходит то же самое, чем мы уже пользовались раньше: первое правило в листинге 2.38 устанавливает нулевую ширину полей для всех

элементов в коде. После этого определяются свойства шрифта для текста в тегах H, а затем элементы HTML5 описываются как блочные элементы (это необходимо для браузеров, которые не умеют по умолчанию назначать стили).

Теперь, когда базовые правила готовы, можно связать с нашим шаблоном гибкую блочную модель. В этой модели каждый элемент, внутри которого находятся структурные элементы, должен быть объявлен как родительский блок. Первый родительский блок в нашем документе — это само тело документа.

Листинг 2.39. Объявление элемента <body> в качестве родительского блока

```
body {
  width: 100%;

  display: -moz-box;
  display: -webkit-box;

  -moz-box-pack: center;
  -webkit-box-pack: center;
}
```

Для того чтобы сделать элемент родительским блоком, нужно связать с ним свойство `display` и присвоить ему значение `box`.

В листинге 2.39 помимо объявления тела документа родительским блоком мы определили стиль, выравнивающий содержимое элемента `<body>` по центру. Значение `center` свойства `box-pack` выравнивает по центру дочерние элементы родительского блока. В нашем случае у `<body>` только один дочерний элемент, а именно `<div id="wrapper">`, поэтому настройка выравнивания распространяется на всю веб-страницу.

Еще одна важная характеристика данной модели заключается в возможности с легкостью увеличивать и уменьшать размер любого элемента вебстраницы в зависимости от доступного свободного пространства. Однако для того, чтобы блок обладал подобной гибкостью, он должен быть потомком такого же гибкого родительского элемента. Размер элементов невозможно корректировать, если неизвестен размер их родительских элементов, поэтому необходимо указать, что тело документа займет все пространство окна браузера. Для этого мы в предыдущем правиле объявили ширину 100 %.

ВНИМАНИЕ

Поскольку свойства CSS, которые мы здесь изучаем, находятся на стадии разработки, большинство их них необходимо объявлять с использованием специального префикса, соответствующего механизму визуализации. В будущем можно будет использовать простую запись display: box, однако пока экспериментальный этап не завершился, приходится делать так, как показано в листинге 2.39. Далее перечислены префиксы для большинства распространенных браузеров:

-moz- — Firefox;

-webkit- — Safari и Chrome;

-o- — Opera;

-khtml- — Konqueror;

-ms- — Internet Explorer;

-chrome- — только Google Chrome.

Существует множество свойств, предназначенных для определения позиции каждого блока на экране. Можно поставить блоки один на другой, вывести их в одной строке, поменять порядок вывода на противоположный или даже объявить конкретную последовательность отображения блоков. Некоторые из этих свойств мы применим в нашем шаблоне прямо сейчас, а чуть позже рассмотрим их подробнее.

Одно из новых свойств называется `box-orient`. Оно определяет вертикальную или горизонтальную ориентацию дочерних элементов. Значение по умолчанию равно `horizontal`, поэтому для элемента `<body>` его указывать не нужно, однако для обертки оно требуется.

Листинг 2.40. Задание для родительского блока максимального размера и вертикальной ориентации

```
#wrapper{
  max-width: 960px;
  margin: 15px 0px;

  display: -moz-box;
  display: -webkit-box;

  -moz-box-orient: vertical;
  -webkit-box-orient: vertical;
```

```
  -moz-box-flex: 1;
  -webkit-box-flex: 1;
}
```

Блок, для которого объявляется свойство `display: box`, обладает свойствами блочного элемента и занимает все свободное пространство в своем контейнере. В предыдущей модели мы использовали для главного блока фиксированное значение ширины, равное 960 пикселам. Это значение фиксировало размер не только данного блока, но и всей веб-страницы. Для того чтобы воспользоваться преимуществами гибких свойств новой блочной модели, необходимо снизить уровень точности. Если `<body>` уже занимает 100 % окна, то обертка будет вести себя точно так же и в некоторых случаях это приведет к искажению пропорций страницы. Чтобы избежать этого, но все же иметь на странице гибкое содержимое, мы в листинге 2.40 использовали свойство `max-width` со значением `960px`. Таким образом мы определили переменную ширину для обертки (и, следовательно, для всей веб-страницы), которая никогда не будет превышать 960 пикселов. Теперь размер веб-страницы будет адаптироваться под каждое устройство и любые условия, а за тем, чтобы единообразие дизайна не нарушалось, будет следить свойство, определяющее максимальную ширину.

Так как блок `<div>`, стиль для которого мы создали в последнем фрагменте кода, обрамляет все содержимое веб-страницы, его также необходимо объявить родительским блоком, применив для этого `display: box`. На этот раз его дочерние элементы будут располагаться один над другим, поэтому мы присвоили свойству `box-orient` значение `vertical`.

Для того чтобы сделать родительский блок гибким, мы добавили свойство `box-flex`. Без этого свойства блок `<div>` невозможно было бы увеличить или уменьшить: его ширина всегда совпадала бы с шириной его содержимого. Значение `1` определяет гибкий блок, а значение `0` — фиксированный. Это свойство может принимать и другие значения, с которыми мы познакомимся чуть позже при других обстоятельствах.

Далее в нашем HTML-документе встречаемся с блоками, создаваемыми элементами `<header>` и `<nav>`. Это первые потомки родительского блока `<div id="wrapper">`. В листинге 2.41 мы всего лишь объявляем некоторые визуальные стили; что касается позиционирования, эти элементы уже обладают свойствами вертикальных блоков, унаследованными от родительского блока.

Под блоком, созданным тегом `<nav>`, находится еще один блок, являющийся предком для двух потомков. Это новый блок `<div>`, который мы добавили в листинге 2.37 для обрамления столбцов в центре веб-страницы. Для идентификации данного блока используется атрибут `id="container"`.

Листинг 2.41. Простые правила для отображения заголовка и меню на экране

```
#main_header {
  background: #FFFBB9;
  border: 1px solid #999999;
  padding: 20px;
}
#main_menu {
  background: #CCCCCC;
  padding: 5px 15px;
}
#main_menu li {
  display: inline-block;
  list-style: none;
  padding: 5px;
  font: bold 14px verdana, sans-serif;
}
```

Листинг 2.42. Создание еще одного родительского блока

```
#container {
  display: -moz-box;
  display: -webkit-box;

  -moz-box-orient: horizontal;
  -webkit-box-orient: horizontal;
}
```

Правило из листинга 2.42 определяет родительский блок для двух столбцов в середине страницы. Первый стиль создает блок, а второй определяет горизонтальную ориентацию его дочерних элементов. Этот контейнер не нужно делать гибким, так же как мы не определяли гибкие блоки для заголовка и строки меню. Его размер по умолчанию равен 100 %, поэтому он занимает все доступное пространство, предоставляемое контейнером, внутри которого он находится. Кроме того, поскольку родительский элемент данного блока гибкий, то и сам блок тоже гибкий. Следовательно, это свойство опускаем.

Подготовив стили для родительских элементов, можно переходить к работе над столбцами. После применения правил из листинга 2.43 первый столбец, определяемый элементом `<section>`, станет гибким, то есть потеряет фиксированный размер. Благодаря свойству `box-flex` он будет

адаптироваться к свободному пространству, предоставляемому его родительским элементом. В противоположность этому для правого столбца, создаваемого элементом `<aside>`, определяется фиксированный размер: 220 пикселов плюс 40 пикселов внутреннего поля (забивки). Разумеется, это всего лишь наше решение для конкретного шаблона. Мы могли бы сделать гибкими оба столбца или разделить доступное пространство между ними пропорционально, применив для этого различные инструменты, доступные в данной модели. Однако вместо этого мы решили сделать размер одного столбца фиксированным, для того чтобы вы смогли сравнить поведение разных блочных моделей на примере одного и того же документа. Кроме того, такая практика широко распространена. Обычно в столбец фиксированной ширины дизайнеры помещают меню, объявления или важную информацию, исходный размер которой важно сохранять.

Листинг 2.43. Создание гибкого и фиксированного столбцов

```
#main_section {
  -moz-box-flex: 1;
  -webkit-box-flex: 1;

  margin: 20px;
}
#main_aside {
  width: 220px;
  margin: 20px 0px;
  padding: 20px;
  background: #CCCCCC;
}
```

Комбинация гибких и фиксированных столбцов — не уникальное свойство гибкой блочной модели, однако ее возможности весьма широки, и некоторые эффекты, для разработки которых когда-то требовалась не одна строка кода, теперь можно реализовать с помощью нескольких простых свойств.

Есть еще одно важное свойство, которое нам объявлять не требуется, так как оно автоматически связывается с обоими столбцами. Свойство `box-align` со значением по умолчанию `stretch` растягивает столбцы по вертикали так, чтобы они занимали все доступное пространство. Благодаря ему правый столбец в нашем шаблоне занимает столько же места по вертикали, сколько и левый.

Результат применения этих свойств — явным образом или по умолчанию — заключается в том, что ширина и высота определенного содержимого веб-страницы могут меняться в зависимости от того, сколько свободного места осталось в окне. Столбец, создаваемый элементом `<section>`, сможет растягиваться и сжиматься по горизонтали, как и его содержимое. Кроме того, если мы уменьшим или увеличим окно браузера, то автоматически поменяется размер заголовка, навигационной полосы и нижнего колонтитула нашего шаблона. Второй столбец, создаваемый элементом `<aside>`, растянется по вертикали на все пространство между навигационной полосой и нижним колонтитулом. Всех этих эффектов было довольно сложно достичь в старой блочной модели. Простота реализации эффектов делает гибкую блочную модель отличным кандидатом для замены традиционной блочной модели.

Теперь, с добавлением последних правил из листинга 2.43, у нас есть все необходимые стили для организации блоков и применения гибкой блочной модели к нашему шаблону. Осталось только добавить немного визуальных стилей к оставшимся элементам, чтобы восстановить предыдущий вариант оформления.

Листинг 2.44. Завершение таблицы стилей

```
#main_footer {
  text-align: center;
  padding: 20px;
  border-top: 2px solid #999999;
}
article {
  background: #FFFBCC;
  border: 1px solid #999999;
  padding: 20px;
  margin-bottom: 15px;
}
time {
  color: #999999;
}
article footer {
  text-align: right;
}
figcaption {
  font: italic 14px verdana, sans-serif;
}
```

САМОСТОЯТЕЛЬНО

Вы уже должны были заменить код шаблона в HTML-файле новым кодом из листинга 2.37. Также вы должны были собрать все CSS-стили, начиная с листинга 2.38, в файле mystyles.css. Теперь этот HTML-файл нужно открыть в браузере и визуально проверить результаты. Разверните и сожмите окно браузера и посмотрите, размер каких блоков будет меняться вслед за изменением доступной площади. Проверьте вертикальный размер правого столбца. На нашем веб-сайте вы сможете одним щелчком мыши загрузить все эти коды и примеры — заходите на *http://www.minkbooks.com*.

Правила из листинга 2.44 схожи с теми, что мы использовали раньше. Единственное различие заключается в том, что больше не нужно добавлять свойство `clear: both` для нижнего колонтитула, потому что плавающих элементов, требующих «очистки», в данной модели нет.

На рис. 2.3 вы видите схематическое визуальное представление того, что нам удалось сделать в предыдущих фрагментах кода. Блоки с номерами

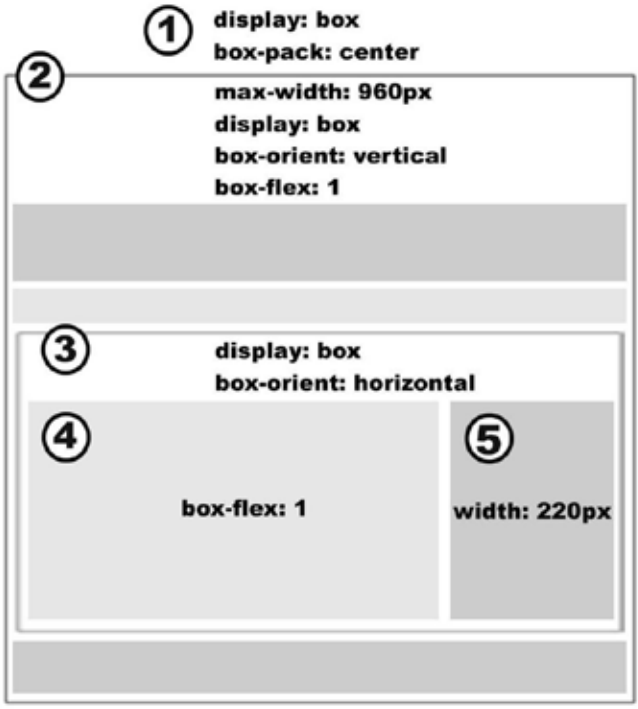

Рис. 2.3. Применение гибкой блочной модели к шаблону

1, 2 и 3 — родительские. Остальные блоки представляют собой дочерние элементы, указанные в порядке, определяемом унаследованными от предков свойствами.

Принципы работы гибкой блочной модели

Пока что мы всего лишь применили гибкую блочную модель к нашему шаблону, однако для того, чтобы полностью раскрыть ее потенциал, необходимо хорошо понимать, как эта модель работает, а также внимательно изучить ее свойства.

Одной из главных причин создания гибкой блочной модели была необходимость иметь возможность эффективно распределять экранное пространство между элементами. Приходилось увеличивать или уменьшать элементы в зависимости от того, сколько свободного места было в соответствующем контейнере. Чтобы понимать, какое количество пространства доступно для распределения, нужно было точно определять размер контейнера, и это привело к возникновению понятия родительских блоков.

Родительские блоки определяются свойством `display`, и их можно описывать как блочные элементы, используя значение `box`, либо как строчные элементы, если задать значение `inline-box`.

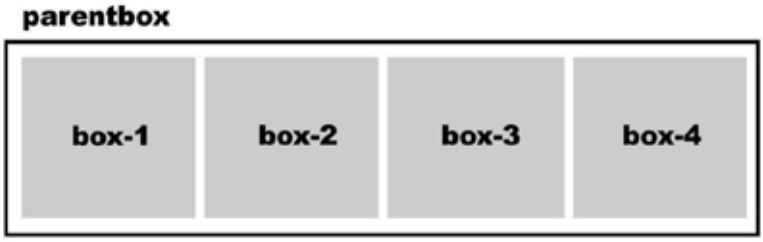

Рис. 2.4. Родительский блок с четырьмя потомками

HTML-код для генерации блоков, изображенных на рис. 2.4, может выглядеть примерно так, как показано в листинге 2.45.

Листинг 2.45. Базовый HTML-код, с которым мы будем работать

```
<!DOCTYPE html>
<html lang="en">
<head>
  <title>Flexible Box Model sample</title>
```

```
  <link rel="stylesheet" href="test.css">
</head>
<body>
<section id="parentbox">
  <div id="box-1">Box 1</div>
  <div id="box-2">Box 2</div>
  <div id="box-3">Box 3</div>
  <div id="box-4">Box 4</div>
</section>
</body>
</html>
```

САМОСТОЯТЕЛЬНО

Создайте еще один пустой текстовый файл с любым названием и расширением .html. Скопируйте в него код из листинга 2.45. Используя этот файл, мы поэкспериментируем со свойствами гибкой блочной модели. Правила CSS будут подключаться из внешнего файла под названием test.css. Создайте и этот файл тоже и добавляйте в него правила из следующих листингов. Проверяйте результаты применения каждого правила, открывая HTML-файл в окне браузера.

Свойство display

Как уже говорилось, свойство `display` может принимать два значения: `box` и `inline-box`. Мы определим родительский блок как блочный элемент, используя значение `box`.

ВНИМАНИЕ

Не забывайте о том, что пока все эти свойства находятся на этапе разработки. Для того чтобы применять их, объявлять каждое из свойств необходимо с добавлением префикса -moz- или -webkit- в зависимости от того, с каким браузером вы работаете. Например, пока что display: box следует записывать как display: -moz-box для механизма Gecko и display: -webkit-box для механизма WebKit. Посмотрите, как это делается, в предыдущих листингах или зайдите на наш вебсайт, чтобы загрузить полные версии этих примеров. Мы решили не добавлять префиксы в листинги из этого раздела, для того чтобы вам было проще читать код.

Листинг 2.46. Объявление элемента parentbox в качестве родительского блока

```
#parentbox {
  display: box;
}
```

Свойство box-orient

По умолчанию родительский блок определяет горизонтальную ориентацию для всех своих потомков. С помощью свойства `box-orient` можно объявить определенную ориентацию элементов.

Листинг 2.47. Объявление ориентации дочерних элементов

```
#parentbox {
  display: box;
  box-orient: horizontal;
}
```

Свойство `box-orient` может принимать одно из четырех значений: `horizontal`, `vertical`, `inline-axis` или `block-axis`. По умолчанию устанавливается значение `inline-axis`, поэтому блоки выстраиваются в ряд по горизонтали (как показано на рис. 2.4). Аналогичного эффекта можно добиться с помощью значения `horizontal`, которое мы применили в листинге 2.47.

> **ВНИМАНИЕ**
>
> Этот и последующие примеры будут работать в браузере, только если вы будете объявлять каждый новый стиль с добавлением префикса -moz- или -webkit- в соответствии с тем, какой браузер используете (соответственно, Firefox или Google Chrome).

Свойство box-direction

Как видите, блоки соответствуют нормальному потоку документа и выводятся слева направо по горизонтали и сверху вниз по вертикали. Одна-

ко нормальный поток можно поменять на обратный, применив свойство `box-direction` (листинг 2.48, рис. 2.5).

Листинг 2.48. Изменение потока документа на противоположный

```
#parentbox {
  display: box;
  box-orient: horizontal;
  box-direction: reverse;
}
```

Свойство `box-direction` может принимать следующие значения: `normal`, `reverse` и `inherit`. Разумеется, значением по умолчанию является `normal`.

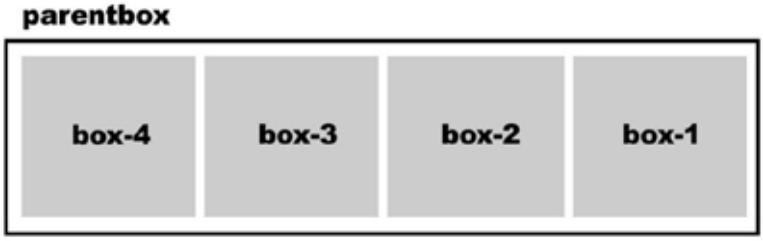

Рис. 2.5. Порядок следования блоков изменен на обратный

Свойство box-ordinal-group

Порядок дочерних блоков можно настраивать совершенно произвольно. Свойство `box-ordinal-group` позволяет определить место каждого отдельного блока.

Листинг 2.49. Настройка позиции отдельных блоков

```
#parentbox {
  display: box;
  box-orient: horizontal;
}
#box-1{
  box-ordinal-group: 2;
}
```

продолжение ⇩

Листинг 2.49 *(продолжение)*

```
#box-2{
  box-ordinal-group: 4;
}
#box-3{
  box-ordinal-group: 3;
}
#box-4{
  box-ordinal-group: 1;
}
```

Для настройки произвольного порядка отображения блоков это свойство необходимо связывать с самими дочерними блоками. В случае повторения порядкового номера соответствующие блоки выводятся в той последовательности, которая определена в структуре HTML (рис. 2.6).

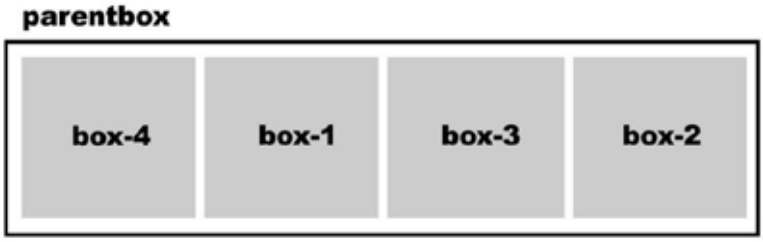

Рис. 2.6. Блоки выстроены в нужном порядке в соответствии с кодом из листинга 2.49

Как уже говорилось, самая важная особенность гибкой блочной модели заключается в возможности эффективно распределять пространство между экранными элементами. Возможны самые разные ситуации, когда доступное пространство приходится делить между дочерними блоками, например: они принадлежат гибкому родительскому блоку, они сами гибкие, и для них не определены конкретные размеры или же ширина родительского блока больше суммы ширины всех его дочерних блоков.

Давайте рассмотрим пример.

Листинг 2.50. Определение фиксированного размера родительского блока и его потомков

```
#parentbox {
  display: box;
```

```
  box-orient: horizontal;
  width: 600px;
}
#box-1{
  width: 100px;
}
#box-2{
  width: 100px;
}
#box-3{
  width: 100px;
}
#box-4{
  width: 100px;
}
```

САМОСТОЯТЕЛЬНО

Добавьте для дочерних блоков какие-нибудь свойства, например фоновый цвет, значение высоты или рамку, для того чтобы они лучше смотрелись и их было проще отличать друг от друга на экране.

В листинге 2.50 размер родительского блока фиксирован и равен 600 пикселам, а размер каждого дочернего блока 100 пикселов. Таким образом, у нас остается 200 пикселов свободного пространства, которое нужно разделить между дочерними блоками.

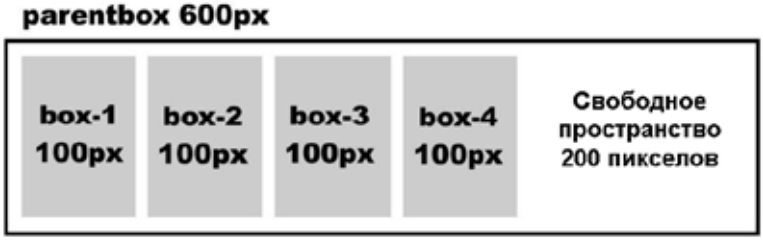

Рис. 2.7. Блоки точных размеров и общее свободное пространство

Распределить свободное пространство можно разными способами. По умолчанию дочерние блоки выводятся в порядке, показанном на рис. 2.7,

слева направо и вплотную друг к другу. Таким образом, в конце остается свободное место. Однако есть и альтернативные варианты, которые мы рассмотрим далее.

Свойство box-pack

Свойство `box-pack` указывает, каким образом в родительском блоке размещаются дочерние блоки и учитывается свободное пространство (листинг 2.51). Данное свойство может принимать четыре значения: `start`, `end`, `center` и `justify`.

Листинг 2.51. Распределение свободного пространства с помощью свойства box-pack

```
#parentbox {
  display: box;
  box-orient: horizontal;
  width: 600px;
  box-pack: center;
}
#box-1{
  width: 100px;
}
#box-2{
  width: 100px;
}
#box-3{
  width: 100px;
}
#box-4{
  width: 100px;
}
```

Свойство `box-pack` по-разному влияет на блоки в зависимости от их ориентации. Если блоки ориентированы горизонтально, то свойство `box-pack` меняет вариант распределения только горизонтального свободного пространства. Точно так же для вертикальных блоков оно определяет, как будет использоваться свободное вертикальное пространство.

Потенциал данного свойства и возможности гибкой блочной модели иллюстрируют рис. 2.8–2.10.

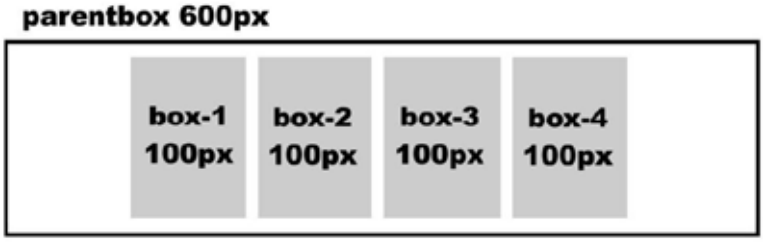

Рис. 2.8. Распределение свободного пространства со свойством box-pack: center

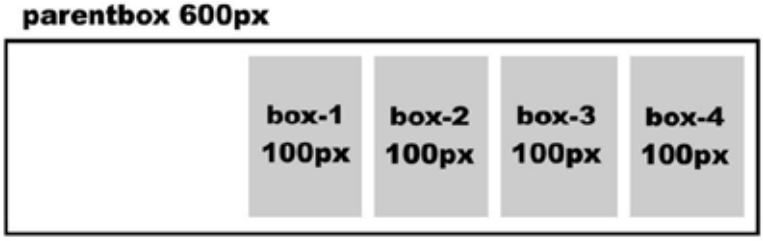

Рис. 2.9. Распределение свободного пространства со свойством box-pack: end

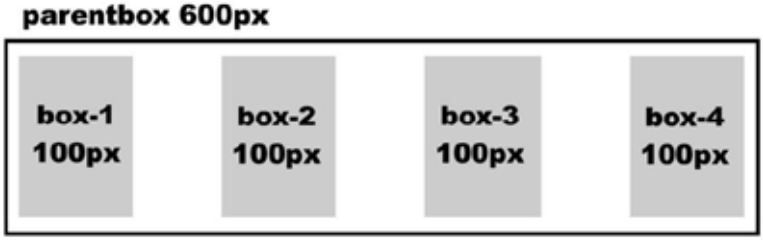

Рис. 2.10. Распределение свободного пространства со свойством box-pack: justify

Свойство box-flex

До сих пор мы делали то, что в определенном смысле противоречит принципам рассматриваемой модели. Мы не пользовались преимуществами, предлагаемыми гибкими элементами. Свойство `box-flex` поможет нам исследовать эти возможности.

Свойство `box-flex` позволяет объявить блок как гибкий или негибкий, что помогает эффективно распределять пространство. По умолчанию блоки считаются негибкими, а значение данного свойства равно 0. Присваивая ему значение `1` или больше, вы объявляете гибкие блоки. Такие блоки растягиваются или сжимаются для заполнения дополнительного пространства. Их размер может меняться как по горизонтали, так и по вертикали — в зависимости от того, какая ориентация была объявлена в свойствах родительского блока.

Распределение пространства зависит также от свойств остальных блоков. Если все дочерние блоки объявлены как гибкие, то размер каждого из них определяется размером родительского блока и значением свойства `box-flex`. Давайте рассмотрим пример (листинг 2.52, рис. 2.11).

Листинг 2.52. Объявление гибких блоков с помощью свойства box-flex

```
#parentbox {
  display: box;
  box-orient: horizontal;
  width: 600px;
}
#box-1{
  box-flex: 1;
}
#box-2{
  box-flex: 1;
}
#box-3{
  box-flex: 1;
}
#box-4{
  box-flex: 1;
}
```

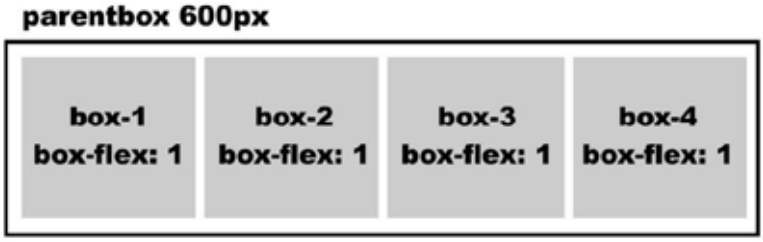

Рис. 2.11. Пространство поровну делится между блоками

Размер каждого из блоков рассчитывается так: размер родительского блока умножается на значение свойства `box-flex` данного дочернего блока, а затем делится на сумму значений свойств `box-flex` всех дочерних блоков. В примере из листинга 2.52 формула для блока 1 будет выглядеть так: $600 \times 1 / 4 = 150$. Значение 600 — это размер родительского блока, 1 — значение свойства `box-flex` блока `box-1`, а 4 — сумма значений свойств `box-flex` всех дочерних блоков. Так как в нашем примере у всех дочерних блоков значение свойства `box-flex` равно 1, их итоговые размеры также равны между собой и составляют 150 пикселов.

Потенциал данного свойства становится очевиден, когда для разных блоков определяются разные значения `box-flex`, когда гибкие блоки сочетаются с негибкими и когда мы задаем точные размеры гибких блоков.

Листинг 2.53. Неравномерное распределение свободного пространства

```
#parentbox {
  display: box;
  box-orient: horizontal;
  width: 600px;
}
#box-1{
  box-flex: 2;
}
#box-2{
  box-flex: 1;
}
#box-3{
  box-flex: 1;
}
#box-4{
  box-flex: 1;
}
```

В листинге 2.53 для блока `box-1` мы поменяли значение свойства `box-flex` — новое значение равно 2. Теперь формула для вычисления размера данного блока выглядит так: $600 \times 2 / 5 = 240$. Поскольку мы не меняли размер родительского блока, первая составляющая формулы осталась прежней, однако вторая теперь равна 2 (новое значение свойства `box-flex` для блока `box-1`). И, разумеется, сумма значений данного свойства всех дочерних блоков теперь равна 5 (2 для `box-1` и 1 для трех

остальных). Применив ту же формулу для оставшихся дочерних блоков, мы с легкостью определяем их размер: $600 \times 1 / 5 = 120$.

Сравнивая результаты, мы видим, что теперь пространство распределено по-другому. Доступное пространство было разделено на количество частей, равное сумме значений свойства `box-flex` всех дочерних блоков (в нашем примере сумма равна 5). После этого получившиеся части распределены между блоками. Блок номер 1 получил две части, а остальные потомки по одной, так как у них значение свойства `box-flex` равно 1.

Результат применения данного метода проиллюстрирован на рис. 2.12. Его основное преимущество состоит в том, что, когда вы добавите новый дочерний блок, не придется пересчитывать все размеры, как в случае, когда они заданы процентными значениями. В гибкой блочной модели размеры пересчитываются автоматически.

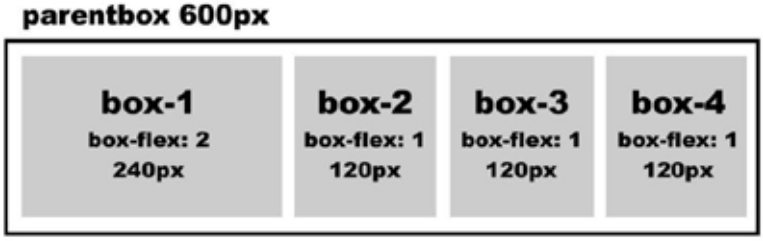

Рис. 2.12. Свободное пространство разделено между блоками в соответствии со значениями свойства box-flex

Возможны и другие сценарии. Например, когда один из дочерних блоков по умолчанию негибкий и для него явно задан размер, остальные дочерние блоки делят между собой оставшееся свободным пространство.

Листинг 2.54. Негибкие и гибкие потомки

```
#parentbox {
  display: box;
  box-orient: horizontal;
  width: 600px;
}
#box-1{
  width: 300px;
}
#box-2{
  box-flex: 1;
}
```

```
#box-3{
  box-flex: 1;
}
#box-4{
  box-flex: 1;
}
```

Размер первого блока в примере из листинга 2.54 равен 300 пикселам, поэтому оставшееся пространство, которое будет делиться между остальными потомками, также имеет ширину 300 пикселов (600 – 300 = 300). Браузер вычислит размер каждого гибкого блока по формуле, которую мы уже изучили: $300 \times 1 / 3 = 100$ (рис. 2.13).

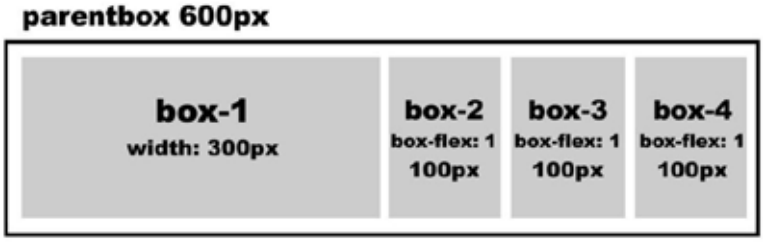

Рис. 2.13. Распределяется только свободное пространство

Размер может быть задан явно как для одного блока, так и для нескольких. Тем не менее принцип не меняется: между остальными блоками распределяется только свободное пространство.

Кроме того, можно определять гибкие блоки, обладающие конкретными размерами (листинг 2.55).

Листинг 2.55. Для гибких блоков заданы предпочтительные размеры

```
#parentbox {
  display: box;
  box-orient: horizontal;
  width: 600px;
}
#box-1{
  width: 200px;
  box-flex: 1;
}
#box-2{
```

продолжение ⇗

Листинг 2.55 *(продолжение)*

```
  width: 100px;
  box-flex: 1;
}
#box-3{
  width: 100px;
  box-flex: 1;
}
#box-4{
  width: 100px;
  box-flex: 1;
}
```

В данном случае для каждого блока указана предпочтительная ширина, однако после размещения всех блоков у нас остается еще 100 пикселов свободного пространства. Это свободное место делится между гибкими блоками. Для вычисления доли, которая достанется каждому блоку, используется уже знакомая нам формула: $100 \times 1 / 4 = 25$. Это означает, что к предпочтительной ширине каждого блока будет добавлено еще 25 пикселов (рис. 2.14).

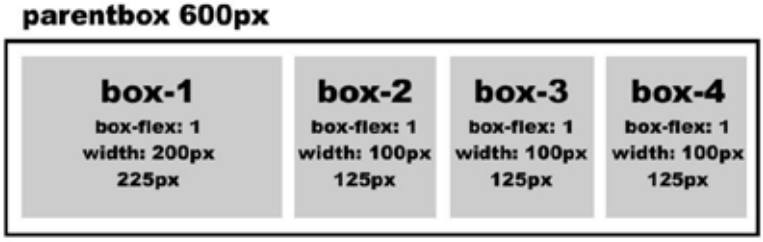

Рис. 2.14. Добавление свободного пространства к значениям ширины всех блоков

Свойство box-align

Еще одно свойство, помогающее распределять пространство, носит название `box-align`. Оно работает аналогично `box-pack`, однако выравнивает блоки в направлении, не совпадающем с ориентацией блоков. Для блоков с вертикальной ориентацией оно определяет позиционирование по горизонтали и наоборот. Данное свойство удобно применять для выравнивания блоков по вертикали — этой возможности очень не хватало разработчикам после того, как повсеместно отказались от таблиц.

Листинг 2.56. Распределение пространства по вертикали

```
#parentbox {
  display: box;
  box-orient: horizontal;
  width: 600px;
  height: 200px;
  box-align: center;
}
#box-1{
  height: 100px;
  box-flex: 1;
}
#box-2{
  height: 100px;
  box-flex: 1;
}
#box-3{
  height: 100px;
  box-flex: 1;
}
#box-4{
  height: 100px;
  box-flex: 1;
}
```

В листинге 2.56 мы указали точное значение высоты для всех блоков, включая родительский. Оставшееся свободное пространство высотой 100 пикселов будет распределяться в соответствии со значением свойства `box-align` (рис. 2.15).

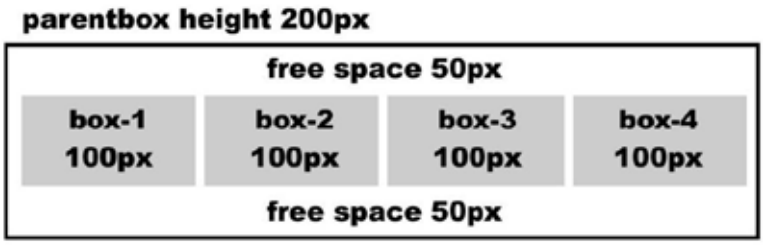

Рис. 2.15. Выравнивание по вертикали с помощью свойства box-align

Свойство `box-align` может принимать следующие значения: `start`, `end`, `center`, `baseline` и `stretch`. Последнее значение растягивает блоки по

вертикали, подгоняя размер дочерних блоков под размер свободного пространства (рис. 2.16). Эта характеристика настолько важна, что именно значение `stretch` выбрано значением по умолчанию. В результате применения данного значения все дочерние блоки автоматически подгоняются под высоту их родительского блока, независимо от того, какая высота задана для них самих.

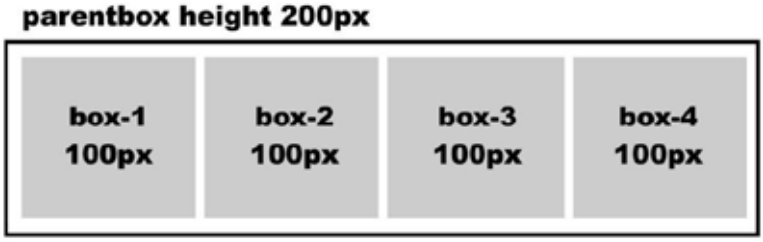

Рис. 2.16. Растягивание дочерних блоков с помощью свойства box-align: stretch

Вы, вероятно, уже обратили внимание на использование этого свойства в нашем шаблоне. Правый столбец, создаваемый элементом `<aside>`, автоматически растягивается на всю высоту, и для этого не приходится определять специальное свойство или добавлять сценарий. Если бы мы попытались реализовать то же самое в традиционной блочной модели, очень скоро у нас случилась бы истерика от нервного перенапряжения!

Краткий справочник. Стили CSS и блочные модели

В HTML5 значительная часть ответственности за структуру ложится на плечи CSS. В последнюю версию CSS было добавлено много нового, а существующие возможности усовершенствованы для обеспечения лучшей организации документов и работы с составляющими их элементами.

Гибкая блочная модель

Гибкая блочная модель — это новая модель, предназначенная для замены используемой в настоящее время традиционной блочной модели.

В CSS3 входит несколько свойств, помогающих применять новую модель к структуре HTML-документов:

- `display` — это свойство было реализовано раньше, однако теперь оно может принимать два дополнительных значения: `box` и `inline-box`. Они превращают элемент в родительский блок, то есть блок, который может содержать другие блоки и определять способ их организации;
- `box-orient` — у этого свойства два допустимых значения: `horizontal` и `vertical`. Они предназначены для ориентации дочерних блоков внутри родительского блока;
- `box-direction` — это свойство с присвоенным ему значением `reverse` меняет нормальный поток документа (слева направо, сверху вниз) на противоположный. По умолчанию свойство принимает значение `normal`. В зависимости от значения свойства блок разрешает своему содержимому «переливаться» через правый или левый край (это происходит в ситуациях, когда дочерние блоки занимают больше места, чем доступно внутри родительского);
- `box-ordinal-group` — это свойство позволяет указать точное положение всех дочерних блоков. Оно должно связываться с дочерними блоками, и ему необходимо присваивать целочисленное значение, представляющее позицию соответствующего блока в группе. Можно использовать совместно с `box-direction` для инвертирования позиций блоков;
- `box-pack` — это свойство помогает браузеру принимать решение, каким образом распределять свободное пространство, оставшееся в родительском блоке после размещения негибких дочерних блоков или дочерних блоков, для которых указана максимальная ширина. Значение по умолчанию `start`, что означает, что блоки выводятся слева направо, если только направление вывода не было инвертировано с помощью `box-direction`. Другие возможные значения — `end`, `center` и `justify`. Последний вариант (`justify`) указывает, что пространство будет разделено поровну и добавлено между дочерними блоками;
- `box-flex` — это свойство делает блок гибким. При этом размер блока рассчитывается на основе того, сколько свободного пространства доступно в родительском блоке и какие размер и свойства определены для остальных дочерних блоков. Свойство `box-flex` влияет только на размер блока по оси, совпадающей с ориентацией. Его значение — это число с плавающей запятой, представляющее долю доступного пространства. Оно рассчитывается по следующей формуле: размер

родительского блока × `box-flex` дочернего блока/сумму `box-flex` всех дочерних блоков. Значение 0 указывает, что блок негибкий, а значение 1 или больше делает блок гибким;

- `box-align` — это свойство определяет способ распределения свободного пространства вдоль оси, перпендикулярной ориентации блоков. Значение по умолчанию `stretch`. В случае горизонтальной ориентации это означает, что блоки растягиваются по вертикали и занимают все доступное пространство сверху донизу. Именно это свойство в дизайне типичной веб-страницы заставляет столбцы автоматически вытягиваться в соответствии с размером их соседей по строке. Другие возможные значения — `start`, `end`, `center` и `baseline`, они делают возможным еще один полезный фокус — выравнивание по вертикали.

Псевдоклассы и селекторы

В CSS3 также были добавлены новые механизмы определения ссылок и выбора элементов HTML.

Селекторы атрибутов. Теперь мы можем использовать и другие атрибуты, помимо `id` и `class`, для поиска элементов в документе и связывания с ними различных стилей. С помощью конструкции вида `ключевое_слово[атрибут=значение]` можно сослаться на элемент, имеющий определенный атрибут с определенным значением. Например, `p[name="text"]` создает ссылку на все элементы `<p>`, у которых есть атрибут с названием `name` и значением `"text"`. В CSS3 также существуют техники для определения менее точных ссылок. Используя комбинации символов `^=`, `$=` и `*=`, мы можем находить элементы, начинающиеся или заканчивающиеся указанными значениями или включающие их в произвольном месте. Например, `p[name^="text"]` находит все элементы `<p>`, у которых есть атрибут с названием `name` и значением, начинающимся на `"text"`.

Псевдокласс `:nth-child()`. Находит определенного потомка в древовидной структуре. Например, с помощью стиля `span:nth-child(2)` можно сослаться на элемент `<span>`, у которого есть братья — другие элементы `<span>` — и который среди них расположен на позиции с номером 2. Это число считается индексом. Кроме того, вместо числа можно использовать такие ключевые слова, как `odd` и `even`, и ссылаться на все элементы с нечетным или четным индексом, например `span:nth-child(odd)`.

Псевдокласс `:first-child`. Как и `:nth-child(1)`, позволяет сослаться на первого потомка.

Псевдокласс `:last-child`. Позволяет сослаться на последнего потомка.

Псевдокласс `:only-child`. Этот псевдокласс позволяет сослаться на элемент, являющийся единственным потомком своего родителя.

Псевдокласс `:not()`. Используется для того, чтобы сослаться на все элементы, за исключением указанного в скобках.

Селектор `>`. Ссылается на второй элемент при условии, что первый элемент является его предком. Например, `div > p` позволяет выбрать только те элементы `<p>`, которые являются потомками элементов `<div>`.

Селектор `+`. Ссылается на элементы, являющиеся братьями. Ссылка указывает на второй элемент при условии, что он расположен сразу за первым элементом. Например, `span + p` позволяет выбрать только те элементы `<p>`, которые следуют за элементами `<span>` и находятся с ними на одном уровне (то есть являются братьями).

Селектор `~`. Похож на предыдущий, но в данном случае второй элемент не обязательно должен следовать за первым, между ними могут находиться другие элементы.

3 Свойства CSS3

Новые правила

Новые приложения на базе реализаций Ajax, впервые появившиеся в начале 2000-х годов, отличались улучшенным дизайном и функциями взаимодействия с пользователями, и эти усовершенствования навсегда изменили Сеть. Версия 2.0 — это название присвоили Сети, перешедшей на следующий уровень развития, — описывала изменения не только в способах передачи данных, но также в дизайне веб-сайтов и приложений.

Коды, реализованные в веб-сайтах нового поколения, быстро превратились в общепризнанный стандарт. Инновация оказалась настолько важной для создания удачных интернет-приложений и сайтов, что программисты разработали целые библиотеки, помогающие преодолевать ограничения и воплощать в жизнь требования дизайнеров.

Отсутствие адекватной поддержки браузерами было очевидно, однако организация, ответственная за веб-стандарты, не принимала маркетинговые тенденции всерьез и пыталась идти собственным путем. К счастью, несколько смышленых ребят одновременно занялись разработкой новых стандартов, и скоро на свет появилась спецификация HTML5. После того как пыль улеглась, объединение HTML, CSS и JavaScript в едином стандарте HTML5 превратило их в храбрых победоносных рыцарей, сумевших привести свои войска к вражеской резиденции.

Несмотря на то что вся эта суматоха происходила совсем недавно, сама битва началась много лет назад, когда была опубликована первая спецификация третьей версии CSS. Когда наконец в 2005 году эту технологию официально признали стандартом, она уже была готова предоставить разработчикам функциональность, которую программисты годами создавали с использованием сложных и не всегда совместимых кодов JavaScript.

В этой главе мы узнаем о вкладе, который спецификация CSS3 внесла в развитие HTML5, а также обо всех новых свойствах, упрощающих жизнь дизайнеров и программистов.

CSS3 сходит с ума

Спецификация CSS всегда относилась только к внешнему виду и форматированию, но теперь все поменялось. В попытке сократить использование кода JavaScript и в целях стандартизации популярных возможностей разработчики CSS3 описали не только дизайн и веб-стили, но также форму и движение. Спецификация CSS3 разбита на модули, обеспечивающие стандартизацию каждого аспекта, участвующего в визуальном представлении документа. От скругленных углов и теней до трансформаций и реорганизации элементов, уже визуализированных на экране, — разработчикам предоставляется возможность реализовать любые эффекты, при создании которых раньше было не обойтись без JavaScript. Благодаря таким обширным изменениям спецификация CSS3 фактически превратилась в совершенно новую технологию по сравнению с предыдущими версиями.

Когда спецификация HTML5 еще находилась в процессе написания, одно то, что ее создатели приняли решение возложить ответственность за дизайн на CSS, позволило им обойти конкурентов сразу на несколько корпусов.

Шаблон

Новые свойства CSS3 обладают чрезвычайной мощью, и их следует изучать последовательно, одно за другим. Однако для того, чтобы упростить процесс, мы свяжем их все с одним и тем же шаблоном. Поэтому давайте начнем с документа HTML и определим некоторые базовые стили (листинг 3.1).

Листинг 3.1. Простой шаблон для тестирования новых свойств

```
<!DOCTYPE html>
<html lang="ru">
<head>
  <title>Новые стили CSS3</title>
  <link rel="stylesheet" href="newcss3.css">
</head>
<body>
<header id="mainbox">
  <span id="title">CSS Styles Web 2.0</span>
</header>
</body>
</html>
```

В нашем документе содержится всего один раздел с коротким текстом. Элемент `<header>`, который мы использовали в данном шаблоне, можно было бы заменить на `<div>`, `<nav>`, `<section>` или любой другой структурный элемент в зависимости от его местоположения и выполняемой функции. После того как мы определим стили, блок из примера в листинге 3.1 будет выглядеть как заголовок — вот почему в код добавлен именно элемент `<header>`.

Так как от элемента `<font>` в HTML5 отказались, для отображения текста обычно используют тег `<span>`, если речь идет о короткой строке, или `<p>`, если это целый абзац. Можно оформлять текст и с помощью других тегов. Поскольку мы выводим всего несколько слов, в нашем шаблоне текст заключен в теги `<span>`.

> **САМОСТОЯТЕЛЬНО**
>
> Используйте код из листинга 3.1 в качестве шаблона для упражнений из этой главы. Вам также понадобится создать новый файл CSS с именем newcss3.css, в котором вы будете сохранять CSS-стили.

Теперь перейдем к базовым стилям для нашего документа.

Листинг 3.2. Базовые правила CSS, с которых все начинается

```
body {
  text-align: center;
}
```

```
#mainbox {
  display: block;
  width: 500px;
  margin: 50px auto;
  padding: 15px;
  text-align: center;
    border: 1px solid #999999;
background: #DDDDDD;
}
#title {
  font: bold 36px verdana, sans-serif;
}
```

Ничего нового в правилах из листинга 3.2 нет, только стили, необходимые для придания формы нашему заголовку. Создаем длинный блок, находящийся в центре окна, добавляем серый фон, рамку и внутри крупный текст «CSS Styles Web 2.0».

Когда вы увидите этот блок на экране, то сразу же заметите, что у него прямые углы. Мне это не нравится. Не знаю, может быть, это какая-то особенность человеческой психологии, но среди веб-разработчиков вряд ли можно найти человека, которому прямые углы были бы по душе. Давайте изменим углы этого поля.

Свойство border-radius

Много лет я испытывал огромные страдания, создавая скругленные углы для полей на своих веб-страницах. Я не самый хороший графический дизайнер, поэтому процесс всегда был крайне утомительным и мучительным. И я знал, что не одинок. Просмотрите любую видеопрезентацию, посвященную новым возможностям HTML5, и вы увидите, что аудитория начинает неистовствовать при каждом упоминании свойства CSS, позволяющего с легкостью создавать скругленные углы. Казалось бы, что может быть проще — это всего лишь банальные скругленные углы! Однако в течение многих лет они представляли собой невероятную сложность для разработчиков.

Вот почему среди всех новых возможностей и поразительных свойств, добавленных в CSS3, в первую очередь я хочу познакомить вас с `border-radius` (листинг 3.3).

Листинг 3.3. Создание скругленных углов

```
body {
  text-align: center;
}
#mainbox {
  display: block;
  width: 500px;
  margin: 50px auto;
  padding: 15px;
  text-align: center;
  border: 1px solid #999999;
  background: #DDDDDD;

  -moz-border-radius: 20px;
  -webkit-border-radius: 20px;
  border-radius: 20px;
}
#title {
  font: bold 36px verdana, sans-serif;
}
```

Свойство `border-radius` пока что находится на этапе разработки, поэтому мы добавили префиксы `-moz-` и `-webkit-` (так же, как делали это для свойств, которые изучали в главе 2). Если все углы должны быть одинаковыми, то свойству можно передать только одно значение. Однако аналогично свойствам `margin` и `padding`, данное свойство позволяет указывать индивидуальные значения для каждого угла.

Листинг 3.4. Разные значения для всех углов поля

```
body {
  text-align: center;
}
#mainbox {
  display: block;
  width: 500px;
  margin: 50px auto;
  padding: 15px;
  text-align: center;
  border: 1px solid #999999;
  background: #DDDDDD;
```

```
  -moz-border-radius: 20px 10px 30px 50px;
  -webkit-border-radius: 20px 10px 30px 50px;
  border-radius: 20px 10px 30px 50px;
}
#title {
  font: bold 36px verdana, sans-serif;
}
```

Как вы видите в листинге 3.4, четыре значения, связанные со свойством `border-radius`, представляют четыре местоположения в следующем порядке: верхний левый угол, верхний правый угол, нижний правый угол и нижний левый угол. Значения всегда перечисляются, начиная с верхнего левого угла по часовой стрелке.

Так же, как свойства `margin` и `padding`, свойство `border-radius` способно принимать набор всего из двух значений. В таком случае первое значение связывается с первым и третьим углами (верхний левый, нижний правый), а второе — со вторым и четвертым углами (верхний правый, нижний левый). Повторю еще раз — углы всегда отсчитываются по часовой стрелке, начиная с левого верхнего.

Для определения формы углов можно также указывать двойные значения, разделенные косой чертой. В таком случае значение слева от косой черты представляет горизонтальный радиус, а значение справа — вертикальный радиус. Сочетание таких значений определяет форму эллипса (листинг 3.5).

Листинг 3.5. Эллиптические углы

```
body {
  text-align: center;
}
#mainbox {
  display: block;
  width: 500px;
  margin: 50px auto;
  padding: 15px;
  text-align: center;
  border: 1px solid #999999;
  background: #DDDDDD;

  -moz-border-radius: 20px / 10px;
  -webkit-border-radius: 20px / 10px;
```

продолжение ⇩

Листинг 3.5 *(продолжение)*

```
  border-radius: 20px / 10px;
}
#title {
  font: bold 36px verdana, sans-serif;
}
```

САМОСТОЯТЕЛЬНО

Скопируйте в CSS-файл с именем newcss3.css стили, которые вы хотели бы протестировать, и откройте HTML-файл из листинга 3.1 в своем браузере.

Свойство box-shadow

Итак, у нас есть симпатичные углы, и мы готовы попробовать нечто большее. Еще один великолепный эффект, которого было чрезвычайно трудно добиться, — это тени. Годами дизайнерам приходилось комбинировать изображения, элементы и некоторые свойства CSS для создания эффекта тени. Благодаря CSS3 и новому свойств `box-shadow` мы добавим тень к нашему полю, написав всего одну строку кода.

Листинг 3.6. Добавление к полю тени

```
body {
  text-align: center;
}
#mainbox {
  display: block;
  width: 500px;
  margin: 50px auto;
  padding: 15px;
  text-align: center;
  border: 1px solid #999999;
  background: #DDDDDD;

  -moz-border-radius: 20px;
  -webkit-border-radius: 20px;
  border-radius: 20px;
```

```
  -moz-box-shadow: rgb(150,150,150) 5px 5px;
  -webkit-box-shadow: rgb(150,150,150) 5px 5px;
  box-shadow: rgb(150,150,150) 5px 5px;
}
#title {
  font: bold 36px verdana, sans-serif;
}
```

Свойству `box-shadow` необходимо передать минимум три значения. Первое, которое вы видите в листинге 3.6, — это цвет. Для формирования данного значения мы использовали функцию `rgb()`, передав ей необходимые аргументы в десятичной системе счисления, однако вы можете просто записать нужный цвет в шестнадцатеричном представлении (как мы делали раньше в этой книге для других параметров).

Следующие два значения, указываемые в пикселах, задают смещение для тени. Оно может быть положительным или отрицательным. Эти значения определяют расстояние от тени до элемента по горизонтали и вертикали соответственно. Отрицательные значения сдвигают тень влево и вверх, и наоборот, положительные значения создают тень справа и ниже элемента. Нулевое значение позволяет поместить тень прямо за элемент, например, для создания эффекта размытости вокруг него.

САМОСТОЯТЕЛЬНО

Для тестирования различных параметров и возможностей создания тени рядом с полем скопируйте код из листинга 3.5 в CSS-файл и откройте HTML-файл с шаблоном из листинга 3.1 в своем браузере. Теперь поэкспериментируйте со значениями свойства box-shadow. Тот же самый код вы сможете использовать для проверки новых параметров, которые мы изучим в дальнейшем.

Пока что тень получается сплошной, без градиентов или эффекта прозрачности, и, следовательно, не очень похожей на настоящую. Мы можем добавить еще несколько параметров и улучшить ее внешний вид.

Четвертое значение, которое принимает данное свойство, задает радиус размытия. Благодаря этому эффекту тень выглядит как настоящая. Попробуйте применить данный параметр со значением 10 пикселов, добавив его в правило из листинга 3.6, как в листинге 3.7.

Листинг 3.7. Добавление радиуса размытия к свойству box-shadow

```
box-shadow: rgb(150,150,150) 5px 5px 10px;
```

Еще одно значение в пикселах, добавляемое в конце свойства, определяет рассеяние тени. Этот эффект немного меняет саму природу тени, расширяя охватываемую область. Как правило, использовать его не рекомендуется, однако в некоторых случаях он может оказаться уместным.

САМОСТОЯТЕЛЬНО

Добавьте значение 20px в конце стиля из листинга 3.7, объедините этот код с кодом из листинга 3.6 и протестируйте в браузере.

ВНИМАНИЕ

Не забывайте, что эти свойства пока что находятся на этапе разработки. Если вы планируете использовать их, всегда добавляйте к ним префикс -moz- или -webkit- в зависимости от того, какой браузер используете (то есть Firefox или Google Chrome).

Последнее возможное значение свойства `box-shadow` — это не число, а ключевое слово `inset`. Оно превращает внешнюю тень во внутреннюю, создавая эффект вдавленного поля.

Листинг 3.8. Внутренняя тень

```
box-shadow: rgb(150,150,150) 5px 5px 10px inset;
```

Стиль из листинга 3.8 определяет внутреннюю тень на расстоянии 5 пикселов от рамки поля, которая дополнительно размывается на 10 пикселов.

САМОСТОЯТЕЛЬНО

Стили из листингов 3.7 и 3.8 — это всего лишь примеры. Для проверки эффекта в браузере вам понадобится внести соответствующие изменения в полный набор правил из листинга 3.6.

ВНИМАНИЕ

Тени не растягивают элемент и не увеличивают его размер, поэтому вам необходимо тщательно проверять доступное пространство. Всегда убеждайтесь, что свободного места достаточно для того, чтобы тень можно было увидеть на экране.

Свойство text-shadow

Итак, мы узнали о тенях все, и теперь вы, вероятно, захотите создать тени для всех элементов в своем документе. Свойство `box-shadow` специально предназначено для добавления теней к полям. Если вы попробуете применить данный эффект, например, к элементу `<span>`, то тень появится у невидимого поля, занимаемого этим элементом, а не у его содержимого. Для добавления теней к фигурам неправильной формы, то есть текстовым символам, используется особое свойство `text-shadow`.

Листинг 3.9. Добавление тени к заголовку

```
body {
  text-align: center;
}
#mainbox {
  display: block;
  width: 500px;
  margin: 50px auto;
  padding: 15px;
  text-align: center;
  border: 1px solid #999999;
  background: #DDDDDD;

  -moz-border-radius: 20px;
  -webkit-border-radius: 20px;
  border-radius: 20px;

  -moz-box-shadow: rgb(150,150,150) 5px 5px 10px;
  -webkit-box-shadow: rgb(150,150,150) 5px 5px 10px;
  box-shadow: rgb(150,150,150) 5px 5px 10px;
}
```

продолжение ⇩

Листинг 3.9 *(продолжение)*

```
#title {
  font: bold 36px verdana, sans-serif;
  text-shadow: rgb(0,0,150) 3px 3px 5px;
}
```

Значения, которые принимает свойство `text-shadow`, аналогичны значениям `box-shadow`. Можно задать цвет тени, расстояние от объекта по горизонтали и вертикали, а также радиус размытия.

В листинге 3.9 синяя тень добавлена к заголовку из нашего шаблона. Она находится на расстоянии 3 пиксела от букв, а радиус ее размытия равен 5 пикселам.

Свойство @font-face

Текстовая тень — отличный трюк, который, кстати, было непросто реализовать методами предыдущего поколения. Однако он всего лишь добавляет эффект объема, а не меняет сам текст. Создание тени — как покраска старого автомобиля: в результате у вас остается все та же старая машина. А у нас — все тот же шрифт.

Проблема со шрифтами стара, как сама Сеть. На компьютерах обычных пользователей Сети чаще всего установлен небольшой набор шрифтов. У разных пользователей на компьютерах можно найти разные шрифтовые гарнитуры, а некоторые умудряются установить шрифты, которых нет у большинства других пользователей. Годами на веб-сайтах приходилось ограничиваться минимальным набором надежных шрифтов — базовой группой, которая наверняка встретится на любом компьютере. Вся информация выводилась на экран исключительно с использованием таких шрифтов.

Свойство `@font-face` позволяет дизайнерам подключать определенный шрифт и применять его для отображения текста на веб-странице. Теперь любой желаемый шрифт можно встроить в веб-сайт, просто добавив файл этого шрифта.

Листинг 3.10. Новый шрифт для заголовка

```
body {
  text-align: center;
}
#mainbox {
```

```
  display: block;
  width: 500px;
  margin: 50px auto;
  padding: 15px;
  text-align: center;
  border: 1px solid #999999;
  background: #DDDDDD;

  -moz-border-radius: 20px;
  -webkit-border-radius: 20px;
  border-radius: 20px;

  -moz-box-shadow: rgb(150,150,150) 5px 5px 10px;
  -webkit-box-shadow: rgb(150,150,150) 5px 5px 10px;
  box-shadow: rgb(150,150,150) 5px 5px 10px;
}
#title {
  font: bold 36px MyNewFont, verdana, sans-serif;
  text-shadow: rgb(0,0,150) 3px 3px 5px;
}
@font-face {
  font-family: 'MyNewFont';
  src: url('font.ttf');
}
```

САМОСТОЯТЕЛЬНО

Загрузите файл font.ttf с нашего веб-сайта или выберите один из имеющихся на вашем компьютере. Скопируйте его в ту же папку, где находится CSS-файл. (Для загрузки файла перейдите по ссылке *http://minkbooks.com/content/font.ttf*.)

Еще больше бесплатных шрифтов можно загрузить со страницы *http://www.moorstation.org/typoasis/designers/steffmann/*.

Свойство `@font-face` требует указания еще по меньшей мере двух свойств для объявления шрифта и загрузки файла. Свойство `font-family` задает название, с помощью которого мы будем ссылаться на данный шрифт, а свойство `src` — URL-адрес файла с кодами для визуализации шрифта.

В листинге 3.10 мы присвоили шрифту имя `MyNewFont` и указали в качестве источника файл font.ttf.

ВНИМАНИЕ

Файл шрифта должен находиться в том же домене, что и веб-страница (или в нашем случае на том же компьютере). Такое ограничение действует в нескольких браузерах, например Firefox.

После того как шрифт загружен, его можно использовать с любыми элементами документа, всего лишь указав имя шрифта (`MyNewFont`). Мы определили стиль `font` в правиле из листинга 3.10 так, чтобы заголовок выводился с использованием нового шрифта. Если же загрузка файла этого шрифта завершится ошибкой, то в качестве альтернативы будут применяться шрифты Verdana и Sans-serif.

Линейный градиент

Градиенты — одна из самых привлекательных новинок в CSS3. Их почти невозможно было реализовать с помощью техник предыдущего поколения, но в CSS3 это делается легко и просто. Для того чтобы придать документу или веб-странице профессиональный облик, достаточно добавить свойство `background` с несколькими параметрами (листинг 3.11).

Листинг 3.11. Добавление симпатичного фонового градиента к нашему полю

```
body {
  text-align: center;
}
#mainbox {
  display: block;
  width: 500px;
  margin: 50px auto;
  padding: 15px;
  text-align: center;
  border: 1px solid #999999;
  background: #DDDDDD;

  -moz-border-radius: 20px;
  -webkit-border-radius: 20px;
```

```
  border-radius: 20px;

  -moz-box-shadow: rgb(150,150,150) 5px 5px 10px;
  -webkit-box-shadow: rgb(150,150,150) 5px 5px 10px;
  box-shadow: rgb(150,150,150) 5px 5px 10px;

  background: -webkit-linear-gradient(top, #FFFFFF, #006699);
  background: -moz-linear-gradient(top, #FFFFFF, #006699);
}
#title {
  font: bold 36px MyNewFont, verdana, sans-serif;
  text-shadow: rgb(0,0,150) 3px 3px 5px;
}
@font-face {
  font-family: 'MyNewFont';
  src: url('font.ttf');
}
```

Градиенты определяются как фоны, поэтому для их добавления на страницу можно использовать свойства `background` и `background-image`. Синтаксис значений, присваиваемых этим свойствам, таков: `linear-gradient (начальная_позиция, начальный_цвет, конечный_цвет)`. Атрибуты функции `linear-gradient()` указывают начальную точку и цвета градиента. Первым может идти значение в пикселах или процентах или одно из ключевых слов: `top`, `bottom`, `left` или `right` (как в нашем примере). В качестве начальной позиции можно также указать угол, для того чтобы задать точное направление градиента (листинг 3.12).

Листинг 3.12. Градиент распространяется под углом 30 градусов

```
background: linear-gradient(30deg, #FFFFFF, #006699);
```

ВНИМАНИЕ

Существует несколько способов реализации эффекта градиента. В этой главе мы изучили стандартный способ, предложенный комиссией W3C. В таких браузерах, как Firefox и Google Chrome, данный стандарт успешно реализован, однако Internet Explorer и другие все еще пытаются справиться с задачей внедрения. Как всегда, прежде чем продавать свой код, тестируйте его во всех доступных браузерах и проверяйте актуальность реализации.

Помимо этого для каждого цвета можно определить конечную точку (листинг 3.13).

Листинг 3.13. Определение конечных точек

```
background: linear-gradient(top, #FFFFFF 50%, #006699 90%);
```

Радиальный градиент

Стандартный синтаксис для радиальных градиентов отличается от рассмотренного синтаксиса линейного градиента всего в нескольких аспектах. Необходимо использовать функцию `radial-gradient()` и новый атрибут, определяющий форму.

Листинг 3.14. Радиальный градиент

```
background: radial-gradient(center, circle, #FFFFFF 0%, #006699
200%);
```

Начальное положение — это источник градиента, и его можно задать как значение в пикселах или процентах или как комбинацию ключевых слов `center`, `top`, `bottom`, `left` и `right`. Для определения фигуры доступны два ключевых слова — `circle` и `ellipse`, а описание промежуточных точек включает в себя значение цвета и позиции, в которой начинается переход.

САМОСТОЯТЕЛЬНО

Замените кодом из листинга 3.14 соответствующий код в листинге 3.11 и проверьте результат в своем браузере (не забудьте добавить префикс -moz- или -webkit- в зависимости от того, какой браузер используете).

RGBA

Пока что мы объявляли только сплошные цвета, используя шестнадцатеричные значения или функцию `rgb()`, возвращающую полное значение цвета для указанных десятичных компонентов. В CSS3 была добавлена новая функция под названием `rgba()`, упрощающая определение цветов

и прозрачности. Кроме того, она решает упомянутую ранее проблему со свойством `opacity`.

У функции `rgba()` четыре атрибута. Первые — такие же, как у `rgb()`, и они просто описывают комбинацию цветовых компонентов. Последний атрибут позволяет описывать прозрачные цвета. Он может принимать значения от 0 до 1, где 0 означает полностью прозрачный цвет, а 1 — совершенно непрозрачный.

Листинг 3.15. Усовершенствование тени за счет прозрачности

```
#title {
  font: bold 36px MyNewFont, verdana, sans-serif;
  text-shadow: rgba(0,0,0,0.5) 3px 3px 5px;
}
```

Листинг 3.15 содержит простой пример, демонстрирующий, насколько привлекательнее становится эффект, если добавить к нему прозрачность. В определении тени заголовка мы заменили функцию `rgb()` функцией `rgba()` и задали значение непрозрачности, равное 0,5. Теперь тень заголовка будет постепенно сливаться с фоном поля, создавая более естественный эффект.

В предыдущих версиях CSS приходилось в каждом браузере применять собственную технику, позволяющую сделать элемент прозрачным. Однако у всех них была одна общая проблема: значение непрозрачности, определенное для элемента, наследовалось всеми его потомками. Функция `rgba()` решила эту проблему: теперь вы можете запросто связать значение непрозрачности с фоном поля, и это никак не затронет его содержимое.

САМОСТОЯТЕЛЬНО

Замените кодом из листинга 3.15 соответствующий код в листинге 3.11 и проверьте результат в своем браузере.

HSLA

Точно так же, как `rgba()` добавляет к старой функции `rgb()` значение непрозрачности, `hsla()` делает это для функции предыдущего поколения `hsl()`.

Функция `hsla()` — всего лишь очередной инструмент генерации цвета элемента, но его преимущество в том, что он интуитивно понятнее, чем `rgba()`. Многим дизайнерам проще создавать собственные наборы цветов с использованием `hsla()`. Синтаксис функции таков: `hsla(тон, насыщенность, яркость, непрозрачность)`.

Согласно синтаксису, тон представляет один из цветов воображаемого цветового круга — это значение от 0 до 360, выраженное в градусах. Вокруг значений 0 и 360° находятся красные цвета, вокруг 120° — зеленые, а вокруг 240° — синие. Насыщенность — это процентное значение в диапазоне от 0 % (шкала серых тонов) до 100 % (полный цвет или полное насыщение). Яркость — это также процентное значение от 0 % (абсолютно темный цвет) до 100 % (абсолютно яркий цвет). Значение 50 % определяет среднюю яркость. Последний аргумент функции `hsla()`, так же, как и в функции `rgba()`, указывает уровень непрозрачности.

Листинг 3.16. Новый цвет для заголовка определяется с помощью hsla()

```
#title {
  font: bold 36px MyNewFont, verdana, sans-serif;
  text-shadow: rgba(0,0,0,0.5) 3px 3px 5px;
  color: hsla(120, 100%, 50%, 0.5);
}
```

САМОСТОЯТЕЛЬНО

Замените кодом из листинга 3.16 соответствующий код в листинге 3.11 и проверьте результат в своем браузере.

Свойство outline

Свойство `outline` — это старое свойство из спецификации CSS, которое в CSS3 было улучшено и теперь включает в себя значение сдвига. Данное свойство применяется для создания второй рамки, причем эта рамка может находиться на расстоянии от края элемента.

Листинг 3.17. Добавление контура к полю заголовка

```
#mainbox {
  display: block;
```

```
  width: 500px;
  margin: 50px auto;
  padding: 15px;
  text-align: center;

border: 1px solid #999999;
  background: #DDDDDD;

  outline: 2px dashed #000099;
  outline-offset: 15px;
}
```

В листинге 3.17 к стилям, которые мы раньше привязали к полю в нашем шаблоне, добавили рамку шириной 2 пиксела со смещением 15 пикселов. Свойство `outline` обладает теми же характеристиками и работает с теми же параметрами, что и свойство `border`. Свойству `outline-offset` передается только одно значение, выраженное в пикселах.

САМОСТОЯТЕЛЬНО

Замените кодом из листинга 3.17 соответствующий код в листинге 3.11 и проверьте результат в своем браузере.

Свойство border-image

Эффекты, предлагаемые свойствами `border` и `outline`, ограничиваются одинарными линиями и несколькими конфигурационными параметрами. Новое свойство `border-image` позволяет преодолеть это ограничение и предоставляет дизайнерам возможность создавать качественные и разнообразные рамки, не используя для этого специальных изображений.

САМОСТОЯТЕЛЬНО

Для тестирования этого свойства будем использовать изображение ромбов в формате PNG. Загрузите файл diamonds.png с нашего веб-сайта, перейдя по ссылке *http://www.minkbooks.com/content/diamonds.png*, и скопируйте его в папку, где находится ваш CSS-файл.

Свойство `border-image` берет в качестве основы для узора предоставленное вами изображение. Затем оно нарезает изображение на куски согласно переданным аргументам и размещает полученные куски вокруг объекта, выстраивая таким образом его рамку (рис. 3.1).

Чтобы добиться желаемого эффекта, нужно указать три атрибута: имя файла изображения и его местоположение, размер фрагментов, которые должны быть вырезаны из изображения, и несколько ключевых слов, описывающих размещение фрагментов вокруг объекта.

Модификации, описанные в листинге 3.18, дают следующий результат: вокруг поля заголовка создается рамка шириной 29 пикселов, а для построения ее содержимого загружается изображение diamonds.png. Значение свойства `border-image`, равное 29, объявляет размер фрагментов, а `stretch` — это один из методов размещения фрагментов вокруг поля.

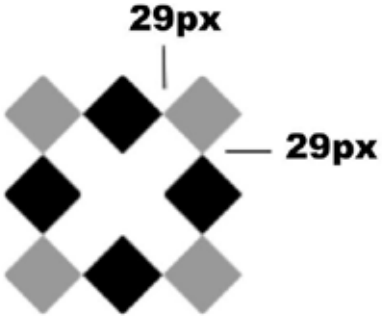

Рис. 3.1. Узор, на основе которого будем строить рамку; ширина каждого фрагмента 29 пикселов

Листинг 3.18. Создаем собственную рамку для поля заголовка

```
#mainbox {
  display: block;
  width: 500px;
  margin: 50px auto;
  padding: 15px;
  text-align: center;
  border: 29px;
  -moz-border-image: url("diamonds.png") 29 stretch;
  -webkit-border-image: url("diamonds.png") 29 stretch;
  border-image: url("diamonds.png") 29 stretch;
}
```

Последний атрибут может принимать одно из трех значений. Ключевое слово `repeat` означает повторение фрагментов, вырезанных из изображения, необходимое число раз для того, чтобы полностью закрыть ими сторону элемента. Размер фрагментов не меняется, если места для фрагмента не хватает, он будет обрезан. Ключевое слово `round` заставляет браузер учитывать длину стороны рамки, на которую накладываются фрагменты, и при необходимости растягивать их, чтобы ни один из фрагментов не был обрезан. Наконец, ключевое слово `stretch` (которое мы использовали в листинге 3.18) заставляет браузер закрывать сторону рамки одним-единственным фрагментом, растягивая его до нужного размера.

Для определения ширины рамки мы использовали свойство `border`, однако можно также применить свойство `border-with` и задать разные размеры для каждого элемента рамки (свойство `border-with` принимает четыре параметра, а его синтаксис аналогичен синтаксису `margin` и `padding`.). То же самое можно сделать и для вырезаемых фрагментов: добавить до четырех параметров, для того чтобы получить из исходного изображения фрагменты разных размеров.

САМОСТОЯТЕЛЬНО

Замените кодом из листинга 3.18 соответствующий код в листинге 3.11 и проверьте результат в своем браузере.

Свойства transform и transition

Сразу после создания элементы HTML подобны неподъемным и прочно закрепленным блокам. Конечно, их можно перемещать с помощью кода JavaScript или возможностей из некоторых популярных библиотек, таких как jQuery (http://www.jquery.com), однако до появления свойств `transform` и `transition` в CSS3 стандартной процедуры для этого не существовало.

Теперь не приходится задумываться, как это делать. Нужно всего лишь выучить несколько параметров, и вы сможете создавать потрясающе динамические веб-сайты, о которых раньше могли только мечтать.

Свойство `transform` умеет выполнять четыре базовых трансформации объекта: масштабировать (функция `scale`), вращать (функция `rotate`), наклонять (функция `skew`) и перемещать или транслировать (функция `translate`). Давайте посмотрим, как это работает.

Функция transform: scale

Листинг 3.19. Масштабирование поля заголовка

```
#mainbox {
  display: block;
  width: 500px;
  margin: 50px auto;
  padding: 15px;
  text-align: center;
  border: 1px solid #999999;
  background: #DDDDDD;

  -moz-transform: scale(2);
  -webkit-transform: scale(2);
}
```

В примере из листинга 3.19 мы взяли базовые стили, которые ранее создали для поля заголовка в листинге 3.2, и трансформировали их, увеличив размер элемента в два раза. Функция `scale` принимает два параметра: значение `X` для горизонтального масштабирования и значение `Y` — для вертикального. Если указано только одно значение, оно используется для обоих параметров.

Масштаб может быть описан как целым, так и дробным значением, а расчет масштабирования выполняется с помощью матрицы. Значения от 0 до 1 уменьшают размер элемента, значение 1 сохраняет исходные пропорции, а значения больше 1 пошагово увеличивают габариты объекта.

Интересного эффекта позволяют добиться отрицательные значения параметров функции.

Листинг 3.20. Создание зеркального отображения с помощью функции scale

```
#mainbox {
  display: block;
  width: 500px;
  margin: 50px auto;
  padding: 15px;
  text-align: center;
  border: 1px solid #999999;
  background: #DDDDDD;
```

```
  -moz-transform: scale(1,-1);
  -webkit-transform: scale(1,-1);
}
```

В листинге 3.20 мы использовали два параметра для масштабирования элемента `mainbox`. Первое значение, равное 1, сохраняет исходные пропорции по горизонтали, а второе значение, равное –1, сохраняя исходные пропорции, инвертирует элемент по вертикали, создавая эффект зеркального отображения.

Существует еще две функции, аналогичные `scale`, но работающие только в одном измерении — либо по горизонтали, либо по вертикали. Эти функции, `scaleX` и `scaleY`, разумеется, принимают только один параметр.

САМОСТОЯТЕЛЬНО

Замените кодом из листингов 3.19 и 3.20 соответствующий код в листинге 3.11 и проверьте результат в своем браузере.

Функция transform: rotate

Функция `rotate` поворачивает элемент по часовой стрелке. Значение необходимо задавать в градусах и указывать единицу измерения — `deg`.

Листинг 3.21. Поворот поля

```
#mainbox {
  display: block;
  width: 500px;
  margin: 50px auto;
  padding: 15px;
  text-align: center;
  border: 1px solid #999999;
  background: #DDDDDD;

  -moz-transform: rotate(30deg);
  -webkit-transform: rotate(30deg);
}
```

Отрицательное значение угла поворота меняет направление вращения элемента.

САМОСТОЯТЕЛЬНО

Замените кодом из листинга 3.21 соответствующий код в листинге 3.11 и проверьте результат в своем браузере.

Функция transform: skew

Функция `skew` влияет на симметрию элемента, сдвигая его в обоих измерениях на указанный угол.

Листинг 3.22. Горизонтальный скос

```
#mainbox {
  display: block;
  width: 500px;
  margin: 50px auto;
  padding: 15px;
  text-align: center;
  border: 1px solid #999999;
  background: #DDDDDD;

  -moz-transform: skew(20deg);
  -webkit-transform: skew(20deg);
}
```

Функция `skew` принимает два параметра, но здесь, в отличие от других функций, каждый из них влияет только на одно измерение; следовательно, эти параметры независимы друг от друга. В листинге 3.22 мы выполнили операцию трансформации поля заголовка, задав скос на 20° Использован только первый параметр, поэтому искажение происходит лишь в горизонтальном измерении. Если бы мы задали оба параметра, то могли бы исказить объект в обоих измерениях. В качестве альтернативы можно также применять независимые функции `skewX` и `skewY` для каждого из измерений.

САМОСТОЯТЕЛЬНО

Замените кодом из листинга 3.22 соответствующий код в листинге 3.11 и проверьте результат в своем браузере.

Функция transform: translate

Аналогично старым свойствам top и left, функция translate перемещает элемент на экране на новое место.

Листинг 3.23. Перемещение поля заголовка вправо

```
#mainbox {
  display: block;
  width: 500px;
  margin: 50px auto;
  padding: 15px;
  text-align: center;
  border: 1px solid #999999;
  background: #DDDDDD;

  -moz-transform: translate(100px);
  -webkit-transform: translate(100px);
}
```

Функция `translate` воспринимает экран как сетку пикселов, а точкой отсчета считает исходное положение элемента. Координаты верхнего левого угла элемента (0, 0), поэтому отрицательные значения параметра задают перемещение объекта левее или выше исходного местоположения, а положительные — правее или ниже.

В листинге 3.23 мы переместили поле заголовка на 100 пикселов вправо относительно исходного положения. Этой функции можно передавать два значения, чтобы передвинуть элемент как по горизонтали, так и по вертикали. Также можно воспользоваться функциями `translateX` и `translateY` для независимого перемещения в обоих измерениях.

САМОСТОЯТЕЛЬНО

Замените кодом из листинга 3.23 соответствующий код в листинге 3.11 и проверьте результат в своем браузере.

Одновременное использование всех видов трансформации

Иногда возникает необходимость одновременно применить к элементу несколько трансформаций. Для того чтобы получить составное свойство transform, нужно всего лишь разделить функции пробелом.

Очень важно помнить о том, что порядок перечисления функций играет огромную роль. Некоторые функции передвигают точку отсчета и центр объекта, изменяя таким образом параметры, с которыми будут работать функции, перечисленные далее.

Листинг 3.24. Перемещение, масштабирование и поворот элемента: трансформации определяются в одной строке

```
#mainbox {
  display: block;
  width: 500px;
  margin: 50px auto;
  padding: 15px;
  text-align: center;
  border: 1px solid #999999;
  background: #DDDDDD;

  -moz-transform: translateY(100px) rotate(45deg) scaleX(0.3);
  -webkit-transform: translateY(100px) rotate(45deg) scaleX(0.3);
}
```

САМОСТОЯТЕЛЬНО

Замените кодом из листинга 3.24 соответствующий код в листинге 3.11 и проверьте результат в своем браузере.

Динамические трансформации

Возможности, которые мы изучали до сих пор, в значительной степени изменили Сеть, но никак не повлияли на ее статичность. Однако мы можем воспользоваться преимуществами комбинирования трансформаций и псевдоклассов и превратить нашу страницу в динамическое приложение.

Листинг 3.25. Реагирование на действия пользователя

```
#mainbox {
  display: block;
  width: 500px;
```

```
  margin: 50px auto;
  padding: 15px;
  text-align: center;
  border: 1px solid #999999;
  background: #DDDDDD;
}
#mainbox:hover{
  -moz-transform: rotate(5deg);
  -webkit-transform: rotate(5deg);
}
```

В листинге 3.25 мы сохранили в исходном виде правило из листинга 3.2, описывающее поле заголовка, но добавили новое правило, применяющее эффект трансформирования через псевдокласс `:hover`. В результате каждый раз, когда указатель мыши будет подводиться к полю заголовка, свойство `transform` повернет заголовок на 5°, а когда указатель мыши выводится за пределы поля, заголовок возвращается в исходное положение. Таким образом мы создали простейшую, но довольно полезную анимацию, не применяя ничего, кроме свойств CSS.

САМОСТОЯТЕЛЬНО

Замените кодом из листинга 3.25 соответствующий код в листинге 3.11 и проверьте результат в своем браузере.

Переходы

Теперь мы можем с легкостью украшать наши страницы прекрасными эффектами с динамическими трансформациями. Однако для создания настоящей анимации необходимо определять переходы между соседними шагами процесса.

Свойство `transition` было создано специально для упрощения этой нетривиальной задачи. Оно волшебным образом создает недостающие шаги, подразумевающиеся в нужном направлении движения. Добавляя всего одно это свойство, мы приказываем браузеру взять на себя заботу об анимации, создать вместо нас все эти невидимые шаги и сгенерировать плавный переход из одного состояния в другое.

Листинг 3.26. Красивое вращение с использованием перехода

```
#mainbox {
  display: block;
  width: 500px;
  margin: 50px auto;
  padding: 15px;
  text-align: center;
  border: 1px solid #999999;
  background: #DDDDDD;

  -moz-transition: -moz-transform 1s ease-in-out 0.5s;
  -webkit-transition: -webkit-transform 1s ease-in-out 0.5s;
}
#mainbox:hover{
  -moz-transform: rotate(5deg);
  -webkit-transform: rotate(5deg);
}
```

Как видно из листинга 3.26, свойство `transition` принимает до четырех параметров, разделенных пробелами. Первое значение представляет свойство, на базе которого будет создаваться переход. Его необходимо указывать, так как одновременно могут меняться несколько свойств, а нам, вероятно, нужно создать шаги только для одного из них. Второй параметр устанавливает время, за которое должен произойти переход из начального положения в конечное. Третьим параметром может выступать любое из пяти ключевых слов: `ease`, `linear`, `ease-in`, `ease-out` и `ease-in-out`. Эти ключевые слова определяют, на основе какой кривой Безье будет выполняться процесс перехода. Каждое ключевое слово представляет отдельную кривую Безье, и единственный способ узнать, какая из них лучше всего подходит именно для вашего перехода, — протестировать все варианты на экране. Последний параметр свойства `transition` — это задержка. Он определяет, какова будет пауза перед началом перехода.

Для того чтобы переход охватывал все меняющиеся свойства данного объекта, необходимо указать ключевое слово all. Также можно явно объявить все участвующие свойства, просто перечислив их через запятую.

САМОСТОЯТЕЛЬНО

Замените кодом из листинга 3.26 соответствующий код в листинге 3.11 и проверьте результат в своем браузере.

ВНИМАНИЕ

В листинге 3.26 мы создали переход для изменений, определяемых свойством transform. Однако свойство transition поддерживает не все свойства CSS, а список поддерживаемых, вероятно, со временем будет меняться. Вам придется самостоятельно тестировать их или изучать веб-сайты производителей соответствующих браузеров для получения дополнительной информации о поддержке.

Краткий справочник. Свойства CSS3

В CSS3 появились новые свойства, позволяющие создавать визуальные и динамические эффекты, которые сегодня считаются неотъемлемой частью Сети:

- `border-radius`. Создает скругленные углы для блочных элементов. Принимает два параметра, определяющих форму каждого угла поля. Первый параметр задает горизонтальную кривую, второй — вертикальную кривую, благодаря чему появляется возможность создавать даже углы эллиптической формы. Если указать только одно значение (например, `border-radius: 20px`), то все углы будут скругляться одинаково. Можно также объявить собственное значение для каждого угла, перечислив их по часовой стрелке от верхнего левого к нижнему левому. Если задаются оба параметра настройки кривой, то эти значения должны быть разделены косой чертой (например, `border-radius: 15px / 20px`);
- `box-shadow`. Создает тени для блочных элементов. Принимает до пяти параметров: цвет, сдвиг по горизонтали, сдвиг по вертикали, радиус размытия и ключевое слово `inset`, если тень должна находиться внутри, а не снаружи элемента. Значение сдвига может быть отрицательным, а радиус размытия и ключевое слово `inset` указывать необязательно (например, `box-shadow: #000000 5px 5px 10px inset`);
- `text-shadow`. Аналогично `box-shadow`, однако предназначено специально для текста. Принимает четыре параметра: цвет, сдвиг по горизонтали, сдвиг по вертикали и радиус размытия (например, `text-shadow: #000000 5px 5px 10px`);
- `@font-face`. Позволяет загрузить и использовать любой желаемый шрифт. В первую очередь необходимо создать шрифт, указав его название в свойстве `font-family` и путь к файлу в свойстве `src` (например,

`@font-face{ font-family: Myfont; src: url('fontfile.ttf') }`). После этого вы сможете связать данный шрифт (в нашем примере — `Myfont`) с любым элементом в документе;

- `linear-gradient(начальная_позиция, начальный_цвет, конечный_цвет)`. Эту функцию можно применять к свойству `background` или `background-image` для создания линейного градиента. Ее атрибуты указывают начальную точку градиента и участвующие цвета. Первое значение может быть в пикселах, процентах или представлять собой одно из ключевых слов: `top`, `bottom`, `left` или `right`. Начальную позицию можно заменить углом, для того чтобы указать точное направление распространения градиента (например, `linear-gradient(top, #FFFFFF 50%, #006699 90%);`);
- `radial-gradient(начальная_позиция, форма, начальный_цвет, конечный_цвет)`. Эту функцию можно применять к свойству `background` или `background-image` для создания радиального градиента. Начальная позиция — это точка отсчета, указанная как значение в пикселах, процентах или комбинация ключевых слов `center`, `top`, `bottom`, `left` и `right`. Форма определяется одним из двух допустимых значений: `circle` для круга и `ellipse` для эллипса, — а цветовые точки включают в себя значение цвета и позицию начала перехода (например, `radial-gradient(center, circle, #FFFFFF 0%, #006699 200%);`);
- `rgba()`. Усовершенствованная версия старой функции `rgb()`. Принимает четыре значения: уровень красного цвета (0–255), уровень зеленого цвета (0–255), уровень синего цвета (0–255) и уровень непрозрачности (значение от 0 до 1);
- `hsla()`. Усовершенствованная версия старой функции `hsl()`. Она принимает четыре значения: тон (значение от 0 до 360), насыщение (значение в процентах), яркость (значение в процентах) и уровень непрозрачности (значение от 0 до 1);
- `outline`. Было улучшено путем добавления еще одного свойства под названием `outline-offset`. В сочетании два свойства создают вторую рамку на расстоянии от первой (например, `outline: 1px solid #000000; outline-offset: 10px;`);
- `border-image`. Создает рамку с определенным изображением. Для этого необходимо сначала создать саму рамку с помощью свойства `border` или `border-with`. Свойство `border-image` принимает минимум три параметра: URL-адрес изображения, размер фрагментов, которые будут вырезаны из изображения для построения рамки, а также ключевое слово, описывающее способ размещения фрагментов (например, `border-image: url("file.png") 15 stretch;`);

- `transform`. Меняет форму элемента. Работает с четырьмя базовыми функциями: `scale`, `rotate`, `skew` и `translate`. Функция `scale` принимает только одно значение. Отрицательное значение инвертирует элемент, значение от 0 до 1 уменьшает его, а значение больше 1 — увеличивает (например, `transform: scale(1.5);`). Функция `rotate` поворачивает элемент и принимает только одно значение, выраженное в градусах (например, `transform: rotate(20deg);`). Функция `skew` принимает два значения, также выраженные в градусах, которые определяют степень трансформации по горизонтали и по вертикали (например, `transform: skew(20deg, 20deg);`). Функция `translate` перемещает объект на количество пикселов, переданное с помощью двух ее параметров (например, `transform: translate(20px);`);
- `transition`. Может применяться к другим свойствам для создания перехода между двумя состояниями элемента. Принимает до четырех параметров: задействованное свойство, длительность перехода, ключевое слово, определяющее тип перехода (`ease`, `linear`, `ease-in`, `ease-out`, `ease-in-out`), и значение задержки, то есть величину паузы, предшествующей переходу (например, `transition: color 2s linear 1s;`).

4 JavaScript

Значение JavaScript

HTML5 можно представлять себе как здание, стоящее на трех опорах: HTML, CSS и JavaScript. Мы уже познакомились с элементами, входящими в HTML, и новыми свойствами, превращающими CSS в идеальный инструмент дизайнера. Теперь настало время торжественно представить одну из сильнейших составляющих спецификации — JavaScript.

JavaScript — это интерпретируемый язык, применяемый для множества целей, но которому пока что отводилась исключительно вспомогательная роль. Одной из инноваций, благодаря которым взгляд на JavaScript кардинально изменился, стали новые механизмы браузеров, специально разработанные для ускорения обработки сценариев. Разработчики большинства успешных механизмов приняли общее ключевое решение: превращать сценарии в машинный код. Это позволило достичь скоростей, сравнимых со скоростями настольных приложений. Улучшение производительности помогло преодолеть существовавшие ранее ограничения, и теперь JavaScript по праву считается лучшим выбором для кодирования в Сети.

Для того чтобы пользоваться преимуществами новой многообещающей инфраструктуры, JavaScript улучшили, приняв во внимание переносимость и интеграцию. Кроме того, по умолчанию в каждый браузер

были встроены полные прикладные интерфейсы программирования (application programming interface, API), дополняющие базовую функциональность языка. Эти новые API (такие как Web Storage (Веб-хранилище), Canvas (Холст) и другие) представляют собой интерфейсы для встроенных в браузеры библиотек. Замысел заключается в том, чтобы обеспечить повсеместную доступность мощных возможностей через простые стандартизированные техники программирования, расширяя таким образом масштаб применения языка и способствуя созданию необходимого и полезного программного обеспечения для Сети.

В этой главе вы научитесь встраивать JavaScript-код в документы HTML и познакомитесь с новейшими усовершенствованиями этого языка программирования, то есть полностью подготовитесь к изучению остального материала книги.

ВНИМАНИЕ

В этой книге познакомим вас лишь с основами JavaScript. Мы будем прорабатывать довольно сложные вопросы, но использовать минимальный объем кода, необходимый для внедрения новых возможностей. Для того чтобы глубже вникнуть в эту тему, зайдите на наш веб-сайт и изучите ссылки для этой главы.

Внедрение JavaScript

Как и при использовании CSS, существуют разные техники встраивания JavaScript-кода в документы HTML. Однако точно так же, как и в случае CSS, в HTML5 рекомендуется применять только одну из них — добавление ссылок на внешние файлы.

ВНИМАНИЕ

В этой главе мы рассказываем о новых возможностях и простейших техниках, которые необходимо знать, чтобы успешно разбирать примеры кода в книге. Если вы уже знакомы с этой базовой информацией, то можете сразу переходить к следующей главе.

Строчные сценарии

Это простая техника добавления JavaScript-кода в документ, основанная на преимуществах атрибутов элементов HTML. Речь идет об атрибутах, являющихся обработчиками событий, которые исполняют код в зависимости от действий пользователя.

Чаще всего используются обработчики событий, относящиеся к движениям мыши, такие как `onclick`, `onMouseOver` и `onMouseOut`. Однако на некоторых сайтах применяются ключевые слова и события окна (например, `onload` и `onfocus`), то есть действия выполняются после нажатия клавиши или изменения состояния окна.

Листинг 4.1. Строчный сценарий JavaScript

```
<!DOCTYPE html>
<html lang="ru">
<head>
  <title>Этот текст — заголовок документа</title>
</head>
<body>
  <div id="main">
  <p onclick="alert('Ты на мне щелкнул!')">Щелкни на мне</p>
  <p>На мне нельзя щелкать</p>
  </div>
</body>
</html>
```

Код в листинге 4.1, использующий обработчик события `onclick`, исполняется каждый раз, когда пользователь щелкает на тексте «Щелкни на мне». Обработчик события `onclick` говорит браузеру примерно следующее: «Когда пользователь щелкнет на этом элементе, выполни этот код», — а кодом является стандартная функция JavaScript, отображающая небольшое окно с сообщением «Ты на мне щелкнул!».

Попробуйте заменить обработчик события `onclick`, например, обработчиком события `onMouseOver`, и код будет запускаться простым подведением указателя мыши к элементу.

В HTML5 допускается использование JavaScript внутри элементов HTML, однако по тем же причинам, которые были перечислены при обсуждении CSS, прибегать к подобному методу не рекомендуется. Код HTML из-за этого сильно раздувается, его становится сложно

поддерживать и обновлять. Кроме того, разбрасывание фрагментов кода по всему документу сильно затрудняет построение эффективных приложений.

Были разработаны новые методы и техники определения ссылок на элементы HTML и регистрации обработчиков событий, не требующие вставки строчных сценариев. Мы вернемся к этому вопросу и подробнее поговорим о событиях и обработчиках событий позже в этой главе.

САМОСТОЯТЕЛЬНО

Скопируйте код из листинга 4.1 и продолжайте копировать последующие примеры кода из этой главы в пустой HTML-файл. Открывайте этот файл в своем браузере для проверки результата.

Встроенный код

Для того чтобы работать с объемным кодом и писать собственные функции, необходимо группировать сценарии между тегами `<script>`. Элемент `<script>` играет ту же роль, что и элемент `<style>` в CSS, то есть позволяет собрать код в одном месте и обращаться к остальным элементам в документе посредством ссылок.

Так же, как и в случае с элементом `<style>`, в HTML5 не требуется использовать атрибут `type` для указания языка в теге `<script>`. В HTML5 по умолчанию считается, что код будет написан на языке JavaScript.

Листинг 4.2. Встроенный сценарий JavaScript

```
<!DOCTYPE html>
<html lang="ru">
<head>
  <title>Этот текст — заголовок документа</title>
  <script>
    function showalert(){
      alert('Ты на мне щелкнул!');
    } function clickme(){
      document.getElementsByTagName('p')[0].onclick=showalert;
```

продолжение ⇗

Листинг 4.2 *(продолжение)*

```
    } window.onload=clickme;
  </script>
</head>
<body>
  <div id="main">
    <p>Щелкни на мне</p>
    <p>На мне нельзя щелкать</p>
  </div>
</body>
</html>
```

Элемент `<script>` и его содержимое могут находиться в любом месте документа, внутри других элементов или между ними. В целях унификации рекомендуем всегда помещать сценарии в голову документа (как в примере в листинге 4.2) и ссылаться на элементы с помощью корректных методов JavaScript, предусмотренных именно для этой цели.

Сейчас в JavaScript существует три метода определения ссылок на элементы HTML:

- `getElementsByTagName` (используется в листинге 4.2) ссылается на элемент по его ключевому слову;
- `getElementById` ссылается на элемент по значению его атрибута `id`;
- `getElementsByClassName` — это новый метод, который позволяет сослаться на элемент по значению его атрибута `class`.

Даже если вы следуете рекомендациям (и помещаете сценарии в голову документа), необходимо учитывать следующее: браузер считывает код последовательно, поэтому невозможно сослаться на элемент, который еще не был создан.

В листинге 4.2 сценарий находится в голове документа и считывается до того, как создаются элементы `<p>`. Если бы мы попытались в этом коде сделать что-либо с элементами `<p>`, то получили бы сообщение об ошибке, объясняющее, что такого элемента не существует. Во избежание данной проблемы мы превратили код в функцию под названием `showalert()`, а ссылку на элемент `<p>` и обработчик событий поместили во вторую функцию, носящую название `clickme()`.

Функции вызываются из последней строки сценария с помощью еще одного обработчика событий (в данном случае связанного с окном), который называется `onload`. Этот обработчик выполняет указанную функцию, когда окно полностью загружается и все элементы полностью создаются.

Давайте рассмотрим процесс выполнения всего документа из листинга 4.2. Сначала функции загружаются, но еще не исполняются. Затем создаются элементы HTML, включая элементы `<p>`. И наконец, когда документ загружен целиком, срабатывает событие `load` и вызывается функция `clickme()`.

В этой функции метод `getElementsByTagName` ссылается на элементы `<p>`. Он возвращает массив, содержащий список элементов, найденных в документе. Однако, добавив в конце метода индекс [0], мы указали, что выбрать нужно только первый элемент. После того как элемент идентифицирован, код регистрирует для него обработчик события `onclick`. Когда соответствующее событие срабатывает, выполняется функция `showalert()`. При этом она открывает на экране небольшое окошко с сообщением «Ты на мне щелкнул!».

Может показаться, что мы использовали слишком много кода, чтобы воспроизвести эффект, который в примере из листинга 4.1 создали всего одной строкой. Однако с учетом потенциала HTML5 и того, какие изощренные решения можно создавать на JavaScript, подобный подход — с кодом, сосредоточенным в одном месте, и тщательно продуманной организацией документов — дает большое преимущество для будущих реализаций и значительно упрощает поддержку и обслуживание веб-сайтов и приложений.

ПОВТОРЯЕМ ОСНОВЫ

Функция — это фрагмент кода, который выполняется только после вызова (активизации) данной функции по имени. Обычно для вызова функции указывают ее имя и передают какие-то значения, заключенные в круглые скобки, например, clickme(1,2). Исключение из этого синтаксиса можно наблюдать в листинге 4.2. Здесь мы не используем скобки, так как передаем обработчику событий ссылку на функцию, а не результат ее выполнения. Для того чтобы больше узнать о функциях JavaScript, зайдите на наш веб-сайт и изучите ссылки для этой главы.

Внешний файл

Коды JavaScript увеличиваются в объеме экспоненциально с добавлением новых функций и обращением к некоторым из упомянутых ранее API. Таким образом, встроенные коды значительно увеличивают документы, к тому же немалая часть кода повторяется. Для того чтобы сократить время загрузки, повысить эффективность и получить возможность распространять и повторно использовать коды в новых документах, не жертвуя эффективностью, мы рекомендуем сохранять коды JavaScript в одном или нескольких внешних файлах и вызывать их через атрибут `src`.

Листинг 4.3. Получение кода из внешних файлов

```
<!DOCTYPE html>
<html lang="ru">
<head>
  <title>Этот текст — заголовок документа</title>
  <script src="mycode.js"></script>
</head>
<body>
  <div id="main">
    <p>Щелкни на мне</p>
    <p>На мне нельзя щелкать</p>
  </div>
</body>
</html>
```

Элемент `<script>` в листинге 4.3 загружает код JavaScript из внешнего файла под названием `mycode.js`. Теперь мы можем вставить этот файл в каждый документ нашего веб-сайта и в любой момент повторно использовать содержащийся в нем код. С точки зрения пользователя, такая практика сокращает время загрузки и доступа к веб-сайту. Для нас же важно то, что при этом значительно упрощаются организация и поддержка.

САМОСТОЯТЕЛЬНО

Скопируйте код из листинга 4.3 в ранее созданный HTML-файл. Создайте новый пустой файл с именем mycode.js и скопируйте в него JavaScript-код из листинга 4.2. Обратите внимание на то, что следует скопировать только код между тегами <script>, исключая сами теги.

Новые селекторы

Как мы уже видели, для того чтобы как-то влиять на элементы HTML из кода JavaScript, необходимо создавать ссылки на них. Если помните, в предыдущих главах мы уже говорили о том, что в CSS, в частности, в CSS3, реализована мощная система определения ссылок и выбора элементов, не сравнимая с теми немногими методами, которые предоставляет JavaScript. Методов `getElementById`, `getElementsByTagName` и `getElementsByClassName` недостаточно для поддержания должного уровня интеграции и актуальности в спецификации HTML5. Для того чтобы перевести JavaScript на уровень, соответствующий современной ситуации, были придуманы и внедрены альтернативные решения. Теперь мы можем выбирать элементы HTML, применяя всевозможные селекторы CSS и используя новые методы `querySelector()` и `querySelectorAll()`.

Метод querySelector()

Этот метод возвращает первый элемент, соответствующий описанной в круглых скобках группе селекторов. Селекторы объявляются в кавычках с использованием синтаксиса CSS, как в листинге 4.4.

Листинг 4.4. Использование метода querySelector()

```
function clickme(){
  document.querySelector("#main p:first-child").onclick=showalert;
}
function showalert(){
  alert('Ты на мне щелкнул!');
}
window.onload=clickme;
```

В листинге 4.4 мы заменили метод `getElementsByTagName`, который использовали раньше, новым методом `querySelector()`. Селектор в этом методе ссылается на первый элемент `<p>`, являющийся потомком элемента, имеющего атрибут `id` со значением `main`.

Ранее уже говорилось, что данный метод возвращает только первый найденный элемент, и вы наверняка заметили лишний псевдокласс `first-child`. Метод `querySelector()` в нашем примере вернет только первый

элемент `<p>` внутри `<div>`, являющийся, разумеется, его первым потомком. Назначение данного примера — показать, что `querySelector()` распознает все допустимые селекторы CSS, так что теперь JavaScript, как и CSS, предоставляет важные инструменты для обращения к любому элементу в документе.

Можно также объявить несколько групп селекторов, разделив их запятыми. Метод `querySelector()` вернет только первый элемент в документе, соответствующий одному из них.

САМОСТОЯТЕЛЬНО

Замените код в файле mycode.js кодом из листинга 4.4 и откройте HTML-файл с кодом из листинга 4.3 в своем браузере, для того чтобы проверить работу querySelector().

Метод querySelectorAll()

В отличие от предыдущего метода, `querySelectorAll()` возвращает все элементы, соответствующие группе селекторов, указанной в круглых скобках. Возвращаемое значение представляет собой массив, содержащий все найденные элементы в том порядке, в котором они находятся в документе.

Листинг 4.5. Использование метода querySelectorAll()

```
function clickme(){
  var list=document.querySelectorAll("#main p");
  list[0].onclick=showalert;
}
function showalert(){
  alert('Ты на мне щелкнул!');
}
window.onload=clickme;
```

Группа селекторов, описанная в методе `querySelectorAll()` в листинге 4.5, вернет все элементы `<p>` в HTML-документе из листинга 4.3, которые являются потомками элемента `<div>`. После выполнения первой строки в массив `list` будут помещены два значения: ссылки на первый и второй элементы `<p>`. Поскольку ключевые слова любых

автоматически создаваемых массивов начинаются с 0, в следующей строке мы ссылаемся на первый найденный элемент, добавив 0 в квадратных скобках.

Обратите внимание на то, что этот пример не демонстрирует весь потенциал `querySelectorAll()`. Обычно данный метод используется для выбора сразу нескольких элементов, а не одного, как в нашем случае. Для прохождения по списку элементов, возвращенных методом, удобно использовать цикл `for`.

Листинг 4.6. Выбор всех элементов, обнаруженных с помощью querySelectorAll()

```
function clickme(){
  var list=document.querySelectorAll("#main p");
  for(var f=0; f<list.length; f++){
    list[f].onclick=showalert;
  }
}
function showalert(){
  alert('Ты на мне щелкнул!');
}
window.onload=clickme;
```

В листинге 4.6 вместо того, чтобы выбирать только первый найденный элемент, мы зарегистрировали обработчик событий `onclick` для всех найденных подходящих элементов, применив для этого цикл `for`. Теперь все элементы `<p>` внутри `<div>` будут реагировать на щелчок на них отображением небольшого окна с сообщением.

Метод `querySelectorAll()`, как и `querySelector()`, может содержать одну или несколько групп селекторов, разделенных запятыми. Эти и предыдущие методы можно комбинировать, для того чтобы точно выбирать нужные элементы. Например, в листинге 4.7 мы выполняем ту же задачу, что и в листинге 4.6, но используем как `querySelectorAll()`, так и `getElementById()`.

Данная техника демонстрирует, насколько точными могут быть эти методы. Мы можем скомбинировать методы в одной строке, а можем выбрать группу элементов и применить к ней второй метод, для того чтобы выбрать только некоторые из полученных на первом этапе. Далее в этой главе познакомимся с дополнительными примерами.

Листинг 4.7. Комбинирование методов

```
function clickme(){
  var list=document.getElementById("main").querySelectorAll("p");
  list[0].onclick=showalert;
}
function showalert(){
  alert('Ты на мне щелкнул!');
}
window.onload=clickme;
```

Обработчики событий

Как говорилось ранее, код JavaScript обычно выполняется после того, как пользователь выполняет какие-то действия. Эти действия и другие события обрабатываются обработчиками событий и связанными с ними функциями JavaScript.

Существует три разных способа регистрации обработчика событий для элемента HTML: можно добавить к элементу новый атрибут, зарегистрировать обработчик событий как свойство элемента или использовать новый стандартный метод `addEventListener()`.

ПОВТОРЯЕМ ОСНОВЫ

В JavaScript действия пользователя называются событиями. Когда пользователь выполняет действие, такое как щелчок клавишей мыши или нажатие на клавишу на клавиатуре, срабатывает событие, связанное с этим конкретным действием. Помимо пользователя, инициатором событий может выступать и система. Например, к числу системных относится событие load, которое срабатывает после полной загрузки документа. Подобные события обрабатываются кодами или целыми функциями. Код, реагирующий на событие, называется обработчиком событий. Когда мы регистрируем обработчик, то, по сути, определяем, как наше приложение будет реагировать на конкретное событие. После того как был стандартизирован метод addEventListener(), данную процедуру стали называть прослушиванием события, а подготовку кода, который будет реагировать на событие, — добавлением прослушивателя события к определенному элементу.

Строчные обработчики событий

Мы уже применяли данную технику в коде в листинге 4.1, где создали атрибут `onclick` для элемента `<p>`. Этот прием считается устаревшим, но в определенных ситуациях нередко оказывается весьма полезным.

Обработчики событий как свойства

Во избежание сложностей, связанных с применением строчной техники, события следует регистрировать в коде JavaScript. Используя селекторы JavaScript, мы можем сослаться на элемент HTML и связать с этим элементом желаемый обработчик события, представив его как свойство.

Мы применили эту технику на практике в коде из листинга 4.2. Двум разным элементам были назначены обработчики событий, оформленные как свойства. Обработчик события `onload` был зарегистрирован для окна с помощью конструкции `window.onload`, а обработчик события `onclick` был зарегистрирован для первого элемента `<p>` в документе с помощью селектора `getElementsByTagName` в строке кода `document.getElementsByTagName('p')[0].onclick`.

> **ПОВТОРЯЕМ ОСНОВЫ**
>
> *Названия обработчиков событий составляют добавлением префикса on к названию события. Например, имя обработчика события click — onclick. Говоря об обработчике onclick, мы имеем в виду код, который будет выполнен после того, как произойдет событие click.*

Ранее в HTML5 техника реализации обработчиков событий посредством JavaScript-кода была единственной, которую поддерживали все браузеры. Некоторые разработчики браузеров создавали собственные системы, но ни одна из них не получила широкого распространения. А после этого появился новый стандарт. Таким образом, данный прием можно рекомендовать для обеспечения совместимости со старыми браузерами, однако в приложениях HTML5 его применять не следует.

Метод addEventListener()

Использование метода `addEventListener()` — это идеальная техника, к тому же стандартизированная в спецификации HTML5. Данный метод

принимает три аргумента: тип события, исполняемую функцию и булево значение (значение логического типа).

Листинг 4.8. Добавление обработчиков событий с помощью addEventListener()

```
<!DOCTYPE html>
<html lang="ru">
<head>
  <title>Этот текст — заголовок документа</title>
  <script>
    function showalert(){
      alert('Ты на мне щелкнул!');
    }
    function clickme(){
      var pelement=document.getElementsByTagName('p')[0];
      pelement.addEventListener('click', showalert, false);
    }
    window.addEventListener('load', clickme, false);
  </script>
</head>
<body>
  <div id="main">
    <p>Щелкни на мне</p>
    <p>На мне нельзя щелкать</p>
  </div>
</body>
</html>
```

В листинге 4.8 вы видите тот же код, что и в листинге 4.2, однако теперь прослушиватель добавляется для каждого события с помощью метода `addEventListener()`. Для упорядочивания кода мы в функции `clickme()` присвоили ссылку на элемент переменной под названием `pelement`, а затем, используя эту переменную, добавили прослушиватель для события `click`.

Синтаксис метода `addEventListener()` иллюстрируется в листинге 4.8. Первый атрибут — это имя события. Второй — функция, которая будет исполняться при наступлении события. Это может быть ссылка на функцию (как в нашем случае) или просто анонимная функция. Третий атрибут, принимающий значение `true` или `false`, указывает, каким образом срабатывают множественные события. Например, мы прослушиваем событие `click` на двух вложенных элементах (один элемент вложен

в другой). Когда пользователь щелкает на этих элементах, два события `click` срабатывают в порядке, определяемом этим атрибутом. Если его значение для одного из элементов равно `true`, то считается, что событие для этого элемента произошло первым, а для другого элемента — вторым по счету. Обычно для большинства ситуаций хватает значения `false`.

ПОВТОРЯЕМ ОСНОВЫ

Анонимные функции — это функции, объявляемые динамически и не имеющие имени (поэтому они и называются анонимными). В JavaScript подобные функции чрезвычайно полезны: они помогают содержать код в порядке и не перегружать глобальный контекст независимыми функциями. В следующих главах мы еще несколько раз будем использовать анонимные функции.

Даже если в результате применения этого и предыдущего методов получаются одинаковые результаты, у `addEventListener()` все же остается огромное преимущество по сравнению с любыми другими подходами — он позволяет добавить неограниченное число прослушивателей для одного и того же элемента. Таким образом, это идеальная реализация для приложений HTML5.

Поскольку события — это краеугольный камень построения интерактивных веб-сайтов и веб-приложений, в спецификации HTML5 описаны несколько полезных событий. Мы изучим каждое из них в соответствующем контексте далее в этой книге.

API-интерфейсы

Если вы раньше уже занимались программированием или хотя бы внимательно прочитали первую половину главы, то обратили внимание на то, сколько кода требуется для выполнения простых задач. Теперь представьте, сколько времени и усилий вам пришлось бы потратить на построение с нуля системы управления базами данных, визуализацию сложных рисунков или создание приложения для обработки фотографий.

Язык JavaScript обладает такой же мощью, как и любой другой современный язык разработки. По той же причине, по которой для профессиональных языков программирования предлагаются библиотеки для создания графических элементов, 3D-механизмы для видеоигр и интерфейсы

доступа к базам данных (и это лишь несколько из множества вспомогательных инструментов), в JavaScript существуют API-интерфейсы, помогающие программистам справляться со сложными задачами.

В HTML5 появилось несколько API, позволяющих из простого кода JavaScript обращаться к различным мощным библиотекам. Потенциал этих нововведений настолько важен, что в следующих главах они превратятся в основной объект исследования. Давайте взглянем на новые возможности, чтобы получить представление, о чем пойдет речь в оставшихся главах книги.

API Canvas (Холст)

API Canvas (Холст) — это API рисования, предоставляющий разработчику простую, но с большими возможностями поверхность для создания изображений. Это самый удивительный и многообещающий API из всех. Возможность динамически создавать и визуализировать рисунки, создавать анимированные изображения, манипулировать изображениями и видео в сочетании с другой функциональностью HTML5 открывает пути к реализации любых графических эффектов, какие только можно представить.

API Canvas генерирует растровое изображение, то есть изображение, состоящее из пикселов. Для его создания и редактирования используется набор функций и методов, предназначенных специально для работы с графическими объектами.

API Drag and Drop (Перетаскивание)

API Drag and Drop (Перетаскивание) разработан для упрощения реализации в Сети самого распространенного действия из настольных приложений. Теперь, добавляя короткие строки кода, мы можем разрешить пользователю перетаскивать элементы по экрану, в том числе помещать их на другие элементы. Под элементами имеются в виду не только рисунки, но также текст, ссылки, файлы и данные.

API Geolocation (Геолокация)

API Geolocation (Геолокация) используется для установления физического местоположения устройства, с помощью которого пользователь обратился к приложению. Существует несколько способов извлечения этой

информации — от сетевых сигналов, таких как IP-адрес, до возможностей глобальной системы позиционирования (Global Positioning System, GPS). Методы API возвращают значения широты и долготы, благодаря которым его можно интегрировать с внешними API географических карт (например, GoogleMaps) или использовать для доступа к специфической локальной информации и построения полезных практических приложений.

API хранения

Для реализации возможностей хранения были созданы два API: Web Storage (Веб-хранилище) и Indexed Database (Индексированная база данных). Эти API отвечают в основном за обработку данных, передаваемых с серверов на пользовательский компьютер, однако веб-хранилище и атрибут `sessionStorage` также делают веб-приложения более управляемыми и повышают их эффективность.

У API Web Storage (Веб-хранилище) есть два важных атрибута, которые сами порой рассматриваются как отдельные небольшие API: `sessionStorage` и `localStorage`.

Атрибут `sessionStorage` обеспечивает корректность данных в течение сеанса работы со страницей, а также отвечает за безопасное хранение временной информации, такой как содержимое корзины в интернет-магазине. Например, он не допускает утечки такой информации в случае ошибки или при неверном использовании (скажем, когда в браузере открывается другое окно).

И наоборот, атрибут `localStorage` API хранения позволяет сохранять на пользовательском компьютере большие файлы. Эта информация сохраняется навсегда и никогда автоматически не стирается, если только это не указано в настройках безопасности.

Оба атрибута, `sessionStorage` и `localStorage`, заменяют собой функциональность файлов cookie и позволяют эффективно преодолевать их ограничения.

Второй API из группы API хранения, независимый от уже перечисленных, — API Indexed Database (Индексированная база данных). Эта база данных предназначена для хранения индексированной информации. Упомянутые ранее API работают с хранилищами больших файлов или временных данных, но данные в них не структурированы. Такая возможность доступна только в системах баз данных — собственно, именно поэтому был создан отдельный API базы данных.

API индексированной базы данных разрабатывался в качестве замены API Web SQL Database (База данных Web SQL). Из-за несоответствия стандарту ни один из этих двух API не был окончательно утвержден. В действительности на момент написания этой главы от API Web SQL Database (который поначалу встречали с распростертыми объятиями) уже отказались.

Поскольку API Indexed Database, также известный как IndexedDB, кажется более многообещающим и поддерживается разработчиками Microsoft, Firefox и Google, в этой книге мы решили использовать именно его. Однако не забывайте, что сейчас рассматривается также возможность внедрения других реализаций SQL.

Файловые API

Под общим названием API File (Файл) спецификация предлагает несколько API для управления файлами. В настоящее время доступны три: API File (Файл), API File: Directories & System (Файл: каталоги и система) и API File: Writer (Файл: запись).

Благодаря этой группе API мы можем считывать, обрабатывать и создавать файлы на пользовательском компьютере.

Коммуникационные API

Некоторые API можно объединить в группу по какому-либо признаку. Это справедливо и для следующих трех: XMLHttpRequest Level 2 (XMLHttpRequest 2-го уровня), Cross Document Messaging (Обмен сообщениями между документами) и Web Sockets (Веб-сокеты).

Коммуникация всегда была основной задачей Интернета, но не все было идеально, и оставалось несколько нерешенных проблем, которые делали обмен данными слишком сложным, а подчас и невозможным. Необходимо было справиться с тремя конкретными проблемами: API для Ajax-приложений был неполным и слишком сложным для межбраузерной реализации, коммуникационные каналы между несвязанными приложениями отсутствовали, и невозможно было установить эффективную двустороннюю связь для доступа к серверной информации в режиме реального времени.

Первая из перечисленных проблем была решена благодаря созданию API XMLHttpRequest 2-го уровня. XMLHttpRequest — это API для создания

Ajax-приложений, то есть приложений, умеющих обращаться к серверу без перезагрузки страницы. Второй уровень этого API включает в себя новые события, предоставляет более широкую функциональность (события позволяют отслеживать прогресс), он переносим (теперь API стандартизован) и высокодоступен (поддерживаются запросы из разных источников).

Для решения второй проблемы был создан API Cross Document Messaging (Обмен сообщениями между документами). Этот API помогает разработчикам преодолевать ограничения на обмен информацией между разными фреймами и окнами. Благодаря этому стал возможен безопасный обмен данными в сообщениях между разными источниками.

Последний коммуникационный API, появившийся в HTML5, — это Web Sockets (Веб-сокеты). Его назначение — предоставлять инструменты, необходимые для приложений реального времени (например, чатов). Этот API позволяет приложениям получать информацию с сервера и отправлять ее на сервер за минимальное время, обеспечивая таким образом функциональность приложений реального времени.

API Web Workers (Рабочие процессы)

Этот уникальный API переводит JavaScript на совершенно новый уровень. JavaScript никогда не был многопотоковым языком, то есть одновременно он способен выполнять только одну задачу. API рабочих процессов обеспечивает возможность обработки кода в фоновом режиме в отдельных потоках, не мешая активности на самой веб-странице. Благодаря этому API JavaScript теперь поддерживает многозадачность.

API History (История)

Ajax изменил способ взаимодействия пользователей с веб-сайтами и веб-приложениями. Браузеры оказались не готовыми к этой ситуации. API истории реализован с целью адаптации современных приложений к способам отслеживания пользовательской активности, принятым в браузерах. Этот API включает в себя техники искусственного создания URL-адресов для каждого шага процесса, что обеспечивает возможность возврата к предыдущим состояниям через стандартные навигационные процедуры.

API Offline (Автономная работа)

Даже сегодня, когда доступ к Интернету есть почти везде, иногда можно оказаться в ситуации, когда подключение будет невозможным. Портативные устройства можно принести куда угодно, однако не везде присутствует сигнал, необходимый для установления связи. А настольные компьютеры могут оставить нас без Интернета в самые неожиданные моменты. Благодаря комбинации атрибутов HTML, событий, управляемых JavaScript, и текстовых файлов API автономной работы теперь позволяет нашим приложениям работать как при наличии подключения, так и в его отсутствие в зависимости от ситуации.

Внешние библиотеки

Целью разработки HTML5 было расширение Сети за счет стандартного набора технологий, который будет поддерживаться любым браузером. И эта спецификация действительно включает в себя все, что требуется разработчику. В действительности концепция HTML5 задумывалась так, чтобы спецификация не зависела от сторонних технологий. Разумеется, по различным причинам к помощи со стороны все же порой приходится прибегать.

До появления HTML5 было разработано несколько библиотек JavaScript, помогающих преодолевать существовавшие в то время ограничения технологии. Некоторые из этих библиотек создавались с конкретными целями — от обработки и валидации форм до генерирования графических объектов и манипулирования ими. Эти библиотеки завоевали огромную популярность, и независимым разработчикам вряд ли под силу будет воспроизвести многие из них (например, Google Maps).

Даже когда новые реализации будут предоставлять более эффективные методы или позволять совершенствовать уже существующие приложения, программисты всегда будут стремиться к поиску самых простых путей решения задач. Библиотеки, упрощающие сложные задачи, всегда будут востребованы, а их число будет только увеличиваться.

Эти библиотеки не входят в спецификацию HTML5, но представляют собой важную составляющую Сети, а некоторые из них уже сегодня используются на самых успешных веб-сайтах и в приложениях. Вместе с прочими возможностями новой спецификации библиотеки делают JavaScript еще лучше и помогают любому желающему идти в ногу с самыми современными технологиями.

jQuery

Это самая популярная библиотека из существующих сегодня. Библиотека jQuery предоставляется бесплатно, а ее назначение — упрощать создание современных приложений на базе JavaScript. Она делает проще выбор элементов HTML и создание анимированных изображений, а кроме того, обрабатывает события и помогает внедрять в приложения возможности Ajax.

Библиотека jQuery представляет собой всего лишь небольшой файл, который можно загрузить с веб-сайта http://www.jquery.com и вставить в документы с помощью тега `<script>`. Она предоставляет простой API, который любой программист может без труда изучить и немедленно применить в своей работе.

Подключив jQuery к своим документам, мы можем пользоваться простыми методами, входящими в библиотеку, превращая статичные веб-сайты в современные и практичные приложения.

Еще одно преимущество jQuery заключается в том, что она поддерживается устаревшими браузерами и делает привычные задачи проще и доступнее для любого разработчика. Можно использовать ее с кодом HTML5 или, что еще проще, заменять функциональностью jQuery некоторые базовые возможности HTML5, что особенно актуально для браузеров, еще не готовых к новым технологиям.

Google Maps

Доступный через API JavaScript и посредством других технологий Google Maps — это уникальный сложный набор инструментов, позволяющий нам, разработчикам, создавать в Сети любые географические сервисы, какие только можно представить. Компания Google стала лидером в сфере предоставления сервисов подобного типа, и посредством технологии Google Maps она открывает разработчикам и пользователям доступ к очень точной и детализированной карте мира. Можно искать конкретное местоположение, вычислять расстояния, находить популярные заведения и даже рассматривать определенные виды так, словно мы находимся там вживую.

API Google Maps предоставляется бесплатно, и этим прикладным интерфейсом может воспользоваться любой разработчик. Готовые к использованию версии API доступны по адресу http://code.google.com/apis/maps/.

Краткий справочник. JavaScript

В HTML5 функциональность JavaScript была расширена путем добавления новых возможностей и усовершенствования существующих методов.

Элементы

- `<script>`. Теперь этот элемент считает JavaScript языком сценариев по умолчанию, атрибут `type` больше указывать не нужно.

Селекторы

Возможность динамически обращаться из кода JavaScript к любым элементам документа лежит в основе построения любого качественного веб-приложения. Для этой цели были реализованы новые методы:

- `getElementsByClassName`. Позволяет находить в документе элементы по значению их атрибутов `class`. Стал дополнением к уже существующим селекторам `getElementsByTagName` и `getElementById`;
- `querySelector(селекторы)`. Позволяет выбрать элементы в документе, используя селекторы CSS. Селекторы объявляются внутри круглых скобок, а сам метод можно комбинировать с другими методами, конструируя более точные ссылки. Он возвращает первый найденный элемент;
- `querySelectorAll(селекторы)`. Похож на `querySelector()`, однако возвращает все элементы, соответствующие перечисленным в скобках селекторам.

События

Важность событий в веб-приложениях привела к необходимости стандартизации методов, которые до этого уже использовались в популярных браузерах:

- `addEventListener(тип, прослушиватель, useCapture)`. Используется для добавления прослушивателя события. Принимает три значения: имя события, функция для обработки события и булево значение, указывающее порядок исполнения для нескольких событий, сработавших

одновременно. Чаще всего третьему атрибуту присваивается значение `false`;

- `removeEventListener(тип, прослушиватель, useCapture)`. Удаляет прослушиватели событий и деактивирует соответствующие обработчики событий. Принимает те же аргументы, что и `addEventListener()`.

API

Масштаб JavaScript значительно расширился благодаря набору модных встроенных приложений, доступ к которым осуществляется посредством API-интерфейсов:

- Canvas (Холст). Это API рисования, позволяющий создавать растровые изображения и манипулировать ими. Для работы с ним используются стандартные методы JavaScript;
- Drag and Drop (Перетаскивание). Делает доступным в Сети наиболее распространенное действие, выполняемое в настольных приложениях. Он позволяет пользователям перетаскивать любые элементы в сетевых документах;
- Geolocation (Геолокация). Этот API предоставляет доступ к информации о физическом местоположении устройства, на котором выполняется приложение. С помощью различных механизмов (например, сетевая информация и GPS) он может извлекать такие данные, как широта и долгота;
- Web Storage (Веб-хранилище). Предоставляет два атрибута для постоянного хранения данных на компьютере пользователя: `sessionStorage` и `localStorage`. Атрибут `sessionStorage` позволяет разработчикам отслеживать действия пользователя, сохраняя информацию, относящуюся к каждому из окон, ограниченное время, а именно до конца сеанса. В противоположность этому атрибут `localStorage` позволяет разработчикам использовать закрытую область хранилища, создаваемую для каждого приложения и имеющую объем до нескольких мегабайт, навсегда записывая информацию и данные в память компьютера пользователя;
- Indexed Database (Индексированная база данных). Добавляет в веб-приложения возможности базы данных на пользовательской стороне. Данная система была разработана независимо от предыдущих технологий и представляет собой простую реализацию базы данных, предназначенную именно для сетевых приложений. База данных сохраняется на компьютере пользователя, она не затирается автоматически (данные

сохраняются навсегда), и, разумеется, пользоваться ею может только приложение, создавшее эту базу данных;

- File (Файл). Это группа API, предназначенная для считывания, записи и обработки файлов пользователя;
- XMLHttpRequest Level 2 (XMLHttpRequest 2-го уровня). Представляет собой усовершенствованную версию старого API XMLHttpRequest и предназначен для построения приложений Ajax. Он включает в себя новые методы для управления ходом выполнения приложения и обработки запросов из разных источников;
- Cross Document Messaging (Обмен сообщениями между документами). Предоставляет новую коммуникационную технологию, позволяющую приложениям обмениваться данными друг с другом в разных фреймах и окнах;
- WebSockets (Веб-сокеты). Предоставляет механизм двусторонней коммуникации между клиентами и серверами. Это необходимо для приложений реального времени, таких как чаты или интерактивные видеоигры;
- Web Workers (Рабочие процессы). Благодаря этому API сценарии в JavaScript могут выполняться в фоновом режиме, не прерывая работу текущей страницы;
- History (История). Этот API предоставляет альтернативное решение для сохранения каждого шага выполнения приложения в истории браузера;
- Offline (Автономная работа). Предназначен для создания приложений, способных продолжать работать, даже когда пользователь отключен от сети.

5 Видео и аудио

Воспроизведение видео с помощью HTML5

Одна из наиболее часто упоминаемых возможностей HTML5 — обработка видео. Однако эта шумиха связана не с новыми инструментами, предоставляемыми HTML5 для этой цели, а с тем фактом, что в момент превращения видео в центральный элемент Интернета все автоматически предположили, что теперь следует ожидать его встроенной поддержки в любых браузерах. Создавалось впечатление, что значимость видео осознавали все, кроме разработчиков технологий для Сети.

Но когда встроенная поддержка и даже стандарт, позволяющий создавать кросс-браузерные приложения для обработки видео, появились, выяснилось, что все гораздо сложнее, чем ожидалось. Причины отказаться от реализации видео оказывались намного серьезнее, чем разработка необходимых для этого кодеков.

Но несмотря на все сложности создатели HTML5 наконец представили элемент, предназначенный для вставки видеофайлов в документы HTML и их воспроизведения. Элементу `<video>` для выполнения своих функций требуются лишь открывающий и закрывающий теги и несколько несложных параметров. Синтаксис чрезвычайно прост, а обязательным является только атрибут `src`.

Листинг 5.1. Базовый синтаксис элемента <video>

```
<!DOCTYPE html>
<html lang="ru">
<head>
  <title>Видеопроигрыватель</title>
</head>
<body>
<section id="player">
  <video src="http://minkbooks.com/content/trailer.mp4" controls>
  </video>
</section>
</body>
</html>
```

Теоретически кода из листинга 5.1 должно быть достаточно — подчеркиваю теоретически. Однако, как уже упоминалось, в реальной жизни все немного сложнее, чем в проектах. Во-первых, необходимо предоставлять по меньшей мере два файла в разных форматах видео: OGG и MP4. Причина заключается в том, что, хотя элемент `<video>` стандартизован, стандартного формата видео не существует. Одни браузеры поддерживают одну группу кодеков, другие — другую, и эти группы между собой могут совершенно не пересекаться. Во-вторых, у кодека, используемого в формате MP4 (единственном поддерживаемом такими важными браузерами, как Safari и Internet Explorer), коммерческая лицензия.

Форматы OGG и MP4 представляют собой контейнеры для видео и аудио. OGG включает в себя видеокодек Theora и аудиокодек Vorbis, а MP4 — видеокодек H.264 и аудиокодек AAC. Сейчас OGG поддерживают браузеры Firefox, Google Chrome и Opera, а MP4 работает в Safari, Internet Explorer и Google Chrome.

Элемент <video>

Давайте пока что постараемся позабыть о трудностях и насладиться простотой элемента `<video>`. У этого элемента несколько атрибутов, предназначенных для установки свойств и конфигурации по умолчанию. Атрибуты `width` и `height`, как и у любого другого элемента HTML,

объявляют размеры элемента `<video>` или окна проигрывателя. Размер самого видео автоматически подгоняется под эти значения, однако они не предназначены для растягивания картинки, поэтому их используют для ограничения области видеосодержимого и сохранения единообразия дизайна. Атрибут `src`, как говорилось ранее, позволяет указать источник видео. Этот атрибут можно заменить элементом `<source>` и его собственным атрибутом `src` и объявить несколько источников для видео в разных форматах (как в следующем примере).

Листинг 5.2. Работающий в разных браузерах видеопроигрыватель с элементами управления по умолчанию

```
<!DOCTYPE html>
<html lang="ru">
<head>
  <title>Видеопроигрыватель</title>
</head>
<body>
<section id="player">
  <video id="media" width="720" height="400" controls>
    <source src="http://minkbooks.com/content/trailer.mp4">
    <source src="http://minkbooks.com/content/trailer.ogg">
  </video>
</section>
</body>
</html>
```

В листинге 5.2 мы расширили определение элемента `<video>`, и теперь между его тегами находятся два элемента `<source>`. Они определяют разные источники видео, для того чтобы браузер мог сделать правильный выбор. Браузер считывает теги `<source>` и, основываясь на поддерживаемых форматах (MP4 или OGG), принимает решение, какой из файлов следует воспроизвести.

САМОСТОЯТЕЛЬНО

Создайте новый пустой HTML-файл с именем video.html (или любым другим), скопируйте в него код из листинга 5.2 и откройте в разных браузерах, чтобы проверить функционирование элемента <video>.

Атрибуты элемента <video>

Вы наверняка заметили в теге `<video>` один атрибут, который появился как в листинге 5.1, так и в листинге 5.2. Атрибут `controls` — один из нескольких специфических атрибутов, доступных для данного элемента. Он отображает элементы управления видео, предоставляемые самим браузером. Каждый браузер активирует собственный интерфейс, при помощи которого пользователь может начать воспроизведение видео, приостановить его, перейти к определенному кадру и т. п.

Помимо `controls` можно использовать также следующие атрибуты:

- `autoplay`. Когда этот атрибут присутствует, браузер автоматически запускает воспроизведение видео, как только это становится возможным;
- `loop`. Если этот атрибут присутствует, браузер начинает воспроизведение видео с начала, как только ролик достигает конца;
- `poster`. Этот атрибут позволяет указать URL-адрес изображения, которое будет отображаться на экране, пока воспроизведение видео не запущено;
- `preload`. Этот атрибут может принимать три значения: `none`, `metadata` и `auto`. Первое указывает, что видео не должно кэшироваться и обычно используется для минимизации расходования трафика. Второе значение, `metadata`, рекомендует браузеру загрузить определенную информацию о ресурсе, например ширину и высоту картинки, продолжительность, первый кадр. Третье значение, `auto`, устанавливается по умолчанию и заставляет браузер загружать файл целиком, как только это становится возможным.

Листинг 5.3. Использование атрибутов тега <video>

```
<!DOCTYPE html>
<html lang="ru">
<head>
  <title>Видеопроигрыватель</title>
</head>
<body>
  <section id="player">
    <video id="media" width="720" height="400" preload controls
      loop poster="http://minkbooks.com/content/poster.jpg">
      <source src="http://minkbooks.com/content/trailer.mp4">
      <source src="http://minkbooks.com/content/trailer.ogg">
```

```
    </video>
  </section>
</body>
</html>
```

В листинге 5.3 мы заполнили элемент `<video>` атрибутами. Из-за того что поведение элемента в разных браузерах может различаться, некоторые атрибуты будут по умолчанию включены или выключены, а часть в зависимости от обстоятельств вообще не будет работать. Для того чтобы получить полный контроль над элементом `<video>` и воспроизводимым видеороликом, необходимо запрограммировать собственный видеопроигрыватель на JavaScript, пользуясь возможностями новых методов, свойств и событий, появившихся в спецификации HTML5.

Программирование видеопроигрывателя

Если вы тестировали предыдущие примеры кода в разных браузерах, то наверняка обратили внимание на то, что в каждом из них используется собственный графический дизайн элементов управления воспроизведением. В каждом браузере собственные кнопки и индикатор прогресса, даже собственный набор функций. Такое положение дел в определенных ситуациях может нас устраивать, однако в профессиональной среде, где каждая деталь имеет значение, единообразие дизайна обязательно должно сохраняться во всех механизмах и приложениях, и мы должны полностью контролировать процесс от начала и до конца.

Спецификация HTML5 предоставляет новые события, свойства и методы манипулирования видео и интегрирования его в документы. Теперь мы можем создавать собственные видеопроигрыватели и наполнять их желаемой функциональностью, применяя для этого HTML, CSS и JavaScript. Теперь видео стало частью документа.

Дизайн

Каждому видеопроигрывателю требуется панель управления, предоставляющая хотя бы базовые возможности. В новом шаблоне в листинге 5.4 после элемента `<video>` мы добавили новый элемент `<nav>`. Элемент `<nav>` включает в себя два элемента `<div>`, `buttons` и `bar`, представляющих кнопку Play и индикатор прогресса.

Листинг 5.4. HTML-шаблон для нашего видеопроигрывателя

```
<!DOCTYPE html>
<html lang="ru">
<head>
  <title>Видеопроигрыватель</title>
  <link rel="stylesheet" href="player.css">
  <script src="player.js"></script>
</head>
<body>
<section id="player">
  <video id="media" width="720" height="400">
    <source src="http://minkbooks.com/content/trailer.mp4">
    <source src="http://minkbooks.com/content/trailer.ogg">
  </video>
  <nav>
    <div id="buttons">
      <button type="button" id="play">Play</button>
    </div>
    <div id="bar">
      <div id="progress"></div>
    </div>
    <div style="clear: both"></div>
  </nav>
</section>
</body>
</html>
```

Шаблон также включает в себя две ссылки на внешние файлы с кодом. Один из файлов — это player.css, содержащий CSS-стили, представленные в листинге 5.5.

Листинг 5.5. CSS-стили для проигрывателя

```
body{
  text-align: center;
}
header, section, footer, aside, nav, article, figure, figcaption,
hgroup{
  display : block;
}
#player{
  width: 720px;
```

```
  margin: 20px auto;
  padding: 5px;
  background: #999999;
  border: 1px solid #666666;

  -moz-border-radius: 5px;
  -webkit-border-radius: 5px;
  border-radius: 5px;
}
nav{
  margin: 5px 0px;
}
#buttons{
  float: left;
  width: 85px;
  height: 20px;
}
#bar{
  position: relative;
  float: left;
  width: 600px;
  height: 16px;
  padding: 2px;
  border: 1px solid #CCCCCC;
  background: #EEEEEE;
}
#progress{
  position: absolute;
  width: 0px;
  height: 16px;
  background: rgba(0,0,150,.2);
}
```

В коде листинга 5.5 для создания блока, который будет включать в себя все составляющие видеопроигрывателя, мы применили техники традиционной блочной модели. С помощью возможностей данной модели блок выравнивается по центру окна. Обратите внимание на третий тег `<div>` в конце элемента `<nav>` нашего шаблона. Он содержит строчный стиль, восстанавливающий нормальный поток документа.

В последнем примере кода нет никаких новых свойств и прочих сюрпризов — это всего лишь группа свойств CSS (которые вы уже изучили), описывающих стили элементов проигрывателя. Однако один стиль все же

можно назвать необычным: атрибут `width` для элемента `<div> progress` инициализируется со значением 0. Это необходимо потому, что мы будем использовать данный элемент для имитации индикатора прогресса, который меняется по мере воспроизведения видео.

САМОСТОЯТЕЛЬНО

Скопируйте новый шаблон из листинга 5.4 в HTML-файл (video.html). Создайте два новых пустых файла для CSS-стилей и JavaScript-кода и присвойте им названия player.css и player.js соответственно. Скопируйте код из листинга 5.5 в CSS-файл, а все следующие фрагменты кода JavaScript — в JavaScript-файл.

Код

Теперь настало время написать код JavaScript для проигрывателя. Запрограммировать видеопроигрыватель можно разными способами, однако мы продемонстрируем применение только самых необходимых событий, методов и свойств, с помощью которых обрабатывается видео. Остальное можете изучить самостоятельно и реализовать любые усовершенствования, которые подскажет воображение.

Мы будем работать с несколькими простыми функциями для запуска и приостановки воспроизведения видео, отображения индикатора прогресса воспроизведения, а также реализации возможности щелкнуть на индикаторе и перейти к соответствующему кадру вперед или назад.

События

В HTML5 появились новые события, относящиеся к конкретным API. Для обработки видео и аудио были добавлены события, информирующие о состоянии мультимедиа — например, о том, какая доля видео уже загружена, завершилось ли воспроизведение файла, воспроизводится видео или приостановлено и т. п. Мы не будем использовать в примере все существующие события, однако для построения более сложных приложений они могут понадобиться. Далее перечислены наиболее часто используемые:

- `progress`. Это событие срабатывает периодически и обновляет информацию о состоянии загрузки мультимедиа. Для доступа к этой инфор-

мации используется атрибут `buffered`, с которым мы познакомимся чуть позже;

- `canplaythrough`. Это событие срабатывает, когда становится известно, что весь файл мультимедиа может быть воспроизведен без перерывов. Статус устанавливается с учетом текущей скорости загрузки и в предположении, что она сохранится до завершения загрузки файла. Существует и другое событие, выполняющее аналогичную функцию, — `canplay`. Однако оно не учитывает ситуацию в целом и срабатывает, когда доступными становятся всего пара кадров;
- `ended`. Срабатывает, когда мультимедиа воспроизводится до конца;
- `pause`. Срабатывает, когда воспроизведение приостанавливается;
- `play`. Срабатывает при запуске воспроизведения мультимедиа;
- `error`. Срабатывает, когда происходит ошибка. Это событие доставляется в элемент `<source>`, соответствующий источнику мультимедиа, на котором произошла ошибка.

Посредством нашего проигрывателя будем прослушивать только обычные события `click` и `load`.

ВНИМАНИЕ

События, методы и свойства для API-интерфейсов пока что находятся на этапе разработки. В этой книге мы изучаем только те из них, которые важны для выполнения задачи и без которых создать приложение попросту не получится. Для того чтобы проверить, каких успехов достигла спецификация HTML5 в этом отношении, посетите наш веб-сайт и пройдитесь по ссылкам для этой главы.

Листинг 5.6. Первая функция

```
function initiate() {
  maxim=600;
  mmedia=document.getElementById('media');
  play=document.getElementById('play');
  bar=document.getElementById('bar');
  progress=document.getElementById('progress');

  play.addEventListener('click', push, false);
  bar.addEventListener('click', move, false);
}
```

В листинге 5.6 представлена первая функция для нашего видеопроигрывателя. Функция называется `initiate`, так как она инициирует приложение, то есть запускает его выполнение после завершения загрузки окна.

Так как эта функция исполняется первой, необходимо установить глобальные переменные для настройки проигрывателя. Используя селектор `getElementById`, мы определили ссылки на все элементы проигрывателя, для того чтобы позднее обращаться к ним в коде. Мы также установили переменную `maxim`, чтобы всегда знать максимальный размер индикатора прогресса (600 пикселов).

Наш проигрыватель должен принимать во внимание два действия: пользователь щелкает на кнопке Play (Воспроизведение) и пользователь щелкает на индикаторе прогресса, для того чтобы перейти к кадру ближе к началу или к концу медиафайла. Для этой цели мы добавили два прослушивателя событий. Сначала добавили к элементу `play` прослушиватель события `click`. Этот прослушиватель будет запускать функцию `push()` каждый раз, когда пользователь щелкнет на элементе (кнопке Play). Второй прослушиватель относится к элементу `bar`. Он указывает, что каждый раз, когда пользователь щелкает на индикаторе прогресса, будет выполняться функция `move()`.

Методы

Функция `push()`, код которой представлен в листинге 5.7, — это первая из функций, выполняющих конкретные действия. В зависимости от ситуации она вызывает специальные методы `pause()` и `play()`.

Листинг 5.7. Эта функция запускает и приостанавливает воспроизведение видео

```
function push(){
  if(!mmedia.paused && !mmedia.ended) {
    mmedia.pause();
    play.innerHTML='Play';
    window.clearInterval(loop);
  }else{
    mmedia.play();
    play.innerHTML='Pause';
    loop=setInterval(status, 1000);
  }
}
```

Специальные методы `play()` и `pause()` входят в список методов, добавленных в HTML5 специально для обработки мультимедиа. Далее перечислены наиболее часто используемые:

- `play()`. Запускает воспроизведение медиафайла с самого начала, если только он не воспроизводился до этого и не был приостановлен;
- `pause()`. Приостанавливает воспроизведение;
- `load()`. Загружает медиафайл. Его полезно применять для того, чтобы в динамических приложениях загружать мультимедиа заранее;
- `canPlayType(тип)`. Благодаря этому методу мы узнаем, поддерживается ли определенный формат файла браузером.

Свойства

В функции `push()` также используются некоторые свойства, позволяющие получать информацию о мультимедиа. Далее перечислены наиболее популярные:

- `paused`. Возвращает значение `true`, если воспроизведение мультимедиа приостановлено или еще не начиналось;
- `ended`. Возвращает значение `true`, если воспроизведение мультимедиа завершилось;
- `duration`. Возвращает длительность мультимедиа в секундах;
- `currentTime`. Способно получать и возвращать значение. Оно либо информирует о текущей позиции воспроизведения мультимедиа, либо устанавливает новую позицию воспроизведения;
- `error`. Возвращает значение ошибки;
- `buffered`. Предоставляет информацию о том, какая доля файла уже загружена в буфер. Благодаря этому мы получаем возможность создавать индикаторы прогресса загрузки. Обычно данное свойство считывают после срабатывания события `progress`. Поскольку пользователи могут запрашивать загрузку мультимедиа не только с начала, но и с любой другой позиции, свойство `buffered` возвращает массив, содержащий все уже загруженные части мультимедиа, а не только ту, начало которой совпадает с началом файла. Для обращения к элементам массива используются атрибуты `end()` и `start()`. Например, код `buffered.end(0)` возвращает продолжительность (в секундах) первой из содержащихся в буфере частей мультимедиа. Поддержка данной возможности еще окончательно не реализована.

Выполнение кода

Теперь, когда мы познакомились со всеми участвующими в обработке элементами, давайте внимательно рассмотрим процесс выполнения функции `push()`.

Эта функция выполняется, когда пользователь щелкает на кнопке Play (Воспроизведение). Сама кнопка играет две роли: в зависимости от ситуации на ней отображается текст «Play», и тогда пользователь может запустить воспроизведение, или «Pause» — в этом состоянии она позволяет приостановить видео. Таким образом, если видео приостановлено или еще не воспроизводилось, эта кнопка включает воспроизведение. И наоборот, если видео уже воспроизводится, то нажатие на кнопку его приостанавливает.

Для того чтобы добиться такого поведения, мы в коде распознаем состояние мультимедиа, проверяя свойства `paused` и `ended`. В первой строке кода находится оператор `if`, проверяющий именно это условие. Если значения `mmedia.paused` и `mmedia.ended` равны `false`, значит, видео воспроизводится, и тогда вызывается метод `pause()`, приостанавливающий видеофайл, а текст на кнопке меняется на «Play» (для этого мы используем `innerHTML`).

Если же истинны противоположные условия, то видео стоит на паузе или его воспроизведение завершилось. Тогда условный оператор возвращает значение `false` и вызывается метод `play()`, запускающий воспроизведение видео с начала или с того момента, когда оно было поставлено на паузу. В этом случае мы дополнительно выполняем еще одно важное действие: начинаем вызывать функцию `status()` каждую секунду, определяя время с помощью `setInterval()`.

Листинг 5.8. Эта функция обновляет вид индикатора прогресса

```
function status(){
  if(!mmedia.ended){
    var size=parseInt(mmedia.currentTime*maxim/mmedia.duration);
    progress.style.width=size+'px';
  }else{
    progress.style.width='0px';
    play.innerHTML='Play';
    window.clearInterval(loop);
  }
}
```

Функция `status()` из листинга 5.8 вызывается каждую секунду, пока видео воспроизводится. В этой функции также присутствует условный оператор `if`, проверяющий статус мультимедиа. Если свойство `ended` возвращает значение `false`, то мы вычисляем требуемую длину индикатора прогресса в пикселах и устанавливаем соответствующий размер для представляющего наш индикатор блока `<div>`. Если же значение свойства равно `true` (то есть воспроизведение видео закончилось), то мы устанавливаем нулевой размер индикатора прогресса, меняем текст на кнопке на «Play» и очищаем цикл с помощью `clearInterval`. Периодический вызов функции `status()` отменяется.

Давайте подробнее остановимся на вычислении размера индикатора прогресса. Поскольку функция `status()` во время воспроизведения видео вызывается каждую секунду, текущее значение позиции воспроизведения постоянно меняется. Это значение (представляющее собой количество секунд) извлекается через свойство `currentTime`. Мы также знаем продолжительность видео — это значение предоставляет свойство `duration` — и максимальный размер индикатора прогресса, сохраненный в переменной `maxim`. Имея эти три значения, несложно вычислить длину (в пикселах) индикатора прогресса, указывающего, сколько секунд видео уже воспроизведено. Формула `текущая позиция времени` × `максимальная длина/общая продолжительность` позволяет перевести секунды в пикселы и соответствующим образом изменить размер блока `<div>`, представляющего индикатор прогресса.

Функцию для обработки события `click` элемента `play` (кнопки) мы создали раньше. Теперь нужно сделать то же самое для индикатора прогресса.

Листинг 5.9. Начинаем воспроизведение с позиции, выбранной пользователем

```
function move(e){
  if(!mmedia.paused && !mmedia.ended){
    var mouseX=e.pageX-bar.offsetLeft;
    var newtime=mouseX*mmedia.duration/maxim;
    mmedia.currentTime=newtime;
    progress.style.width=mouseX+'px';
  }
}
```

Прослушиватель события `click` добавляется к элементу `bar` для проверки, не щелкнул ли пользователь на индикаторе прогресса, чтобы начать

воспроизведение видео с новой позиции. Для обработки события, когда оно происходит, прослушиватель применяет функцию `move()`. Код этой функции представлен в листинге 5.9. Так же, как и предыдущие функции, она начинается с условного оператора `if`, однако на этот раз оператор нужен для того, чтобы следующий код выполнялся только для воспроизводящегося видео. Если значения свойств `paused` и `ended` равны `false`, значит, видео воспроизводится и код необходимо выполнить.

Для вычисления позиции времени, с которой нужно начать воспроизведение видео, нам приходится сделать несколько вещей. Нужно определить точное местоположение мыши, в котором произошло событие `click`, вычислить расстояние в пикселах от этой точки до начала индикатора прогресса и определить, какому количеству секунд соответствует это расстояние на временной шкале.

Процессы, регистрирующие обработчики событий (или прослушиватели событий), такие как `addEventListener()`, всегда генерируют ссылку на событие. Ссылка передается функции-обработчику в виде атрибута. Традиционно для сохранения этого значения используется переменная `e`. В функции из листинга 5.9 мы воспользовались значением данной переменной и свойства `pageX` для определения точного местоположения указателя мыши в момент срабатывания события. Свойство `pageX` возвращает значение, указывающее точку в системе координат всей страницы, а не конкретного индикатора прогресса или окна. Для того чтобы узнать расстояние между началом индикатора прогресса и указателем мыши, необходимо вычесть из значения `pageX` расстояние между краем страницы и началом полосы индикатора. Вы наверняка помните, что наш индикатор прогресса находится внутри поля, выровненного по центру экрана. Предположим, что от левого края страницы его отделяет 421 пиксел, а щелчок выполнен точно по центру полосы индикатора. Длина полосы индикатора равна 600 пикселам, следовательно, щелчок сделан на расстоянии 300 пикселов от начала индикатора. Однако свойство `pageX` вернет другое значение, отличное от 300, а конкретно 721. Для получения точной позиции на полосе индикатора прогресса нужно вычесть из значения `pageX` расстояние между левым краем страницы и началом полосы (в нашем примере 421 пиксел). Получить это значение можно благодаря свойству `offsetLeft`. Таким образом, формула `e.pageX – bar.offsetLeft` дает точное местоположение указателя мыши относительно начала полосы индикатора прогресса. В нашем примере итоговая формула будет выглядеть так: 721 – 421 = 300.

Получив данное значение, необходимо преобразовать его в секунды. Мы берем значение свойства `duration`, точное местоположение указа-

теля мыши на полосе индикатора прогресса и максимальный размер полосы и создаем следующую формулу: `mouseX` × `video.duration`/`maxim`. Она дает нам искомое значение, которое мы сохраняем в переменной `newtime`. Результат представляет собой время в секундах, соответствующее позиции на индикаторе прогресса, в которой пользователь щелкнул мышью.

После этого мы начинаем воспроизведение видео с новой позиции. Свойство `currentTime`, как говорилось ранее, возвращает текущую позицию воспроизведения видео, оно способно также переместить ползунок в новую позицию, если передать свойству новое значение. Установка свойства с использованием переменной `newtime` позволяет перенести ползунок воспроизведения в желаемую позицию.

Осталось только поменять размер элемента `progress` в соответствии с новой ситуацией на экране. Используя значение переменной `mouseX`, мы можем поменять значение элемента так, чтобы точно попасть в точку, где произошел щелчок указателем мыши.

Код видеопроигрывателя почти готов. У нас есть все события, методы, свойства и функции, необходимые для работы приложения. Остались всего одна строка кода и одно дополнительное событие, которое нам также необходимо прослушивать для успешной реализации задачи.

Листинг 5.10. Прослушивание события load

```
window.addEventListener('load', initiate, false);
```

Мы могли бы применить проверенный временем стандартный подход с `window.onload` и зарегистрировать обработчик события — это обеспечило бы намного лучшую совместимость со старыми браузерами. Однако поскольку книга посвящена HTML5, мы решили прибегнуть к стандартной возможности HTML5 — методу `addEventListener()`.

САМОСТОЯТЕЛЬНО

Скопируйте все фрагменты кода JavaScript, начиная с листинга 5.6, в файл player.js. Откройте файл video.html, содержащий шаблон из листинга 5.4, в своем браузере и щелкните на кнопке Play. Протестируйте приложение в разных браузерах.

Форматы видео

Пока что единый стандарт формата видео и аудио для Сети не определен. Существуют несколько контейнеров и разнообразные кодеки, однако ни один из них не получил всеобщего признания, и производители браузеров не могут прийти к консенсусу относительно того, как добиться этого в ближайшем будущем.

Наиболее распространенные контейнеры — это OGG, MP4, FLV и новый контейнер WEBM, предложенный Google. Видео для этих контейнеров обычно кодируется с помощью кодеков Theora, H.264, VP6 и VP8 соответственно. Список кодеков приведен далее:

- OGG — видеокодек Theora и аудиокодек Vorbis;
- MP4 — видеокодек H.264 и аудиокодек AAC;
- FLV — видеокодек VP6 и аудиокодек MP3. Также поддерживает H.264 и AAC;
- WEBM — видеокодек VP8 и аудиокодек Vorbis.

Кодеки для OGG и WEBM бесплатные, однако использование кодеков для MP4 и FLV ограничивается патентами. Это означает, что, если вы желаете встраивать в свои приложения мультимедиа в формате MP4 и FLV, придется платить. Правда, некоторые ограничения снимаются для бесплатных приложений.

Проблема также в том, что в настоящее время Safari и Internet Explorer не поддерживают бесплатные технологии. Они работают только с MP4. Разработчики Internet Explorer объявили о том, что планируется также добавление поддержки видеокодека VP8 (об аудио ничего сказано не было). Список поддерживаемых разными браузерами кодеков представлен далее:

- Firefox — видеокодек Theora и аудиокодек Vorbis;
- Google Chrome — видеокодек Theora и аудиокодек Vorbis. Также поддерживает видеокодек H.264 и аудиокодек AAC;
- Opera — видеокодек Theora и аудиокодек Vorbis;
- Safari — видеокодек H.264 и аудиокодек AAC;
- Internet Explorer — видеокодек H.264 и аудиокодек AAC.

Когда в будущем появится поддержка открытых форматов, таких как WEBM, жизнь станет проще, однако в ближайшие два-три года единого стандарта выработано, скорее всего, не будет, и нам придется все так же

делать выбор среди альтернативных вариантов, ориентируясь на природу наших приложений и бизнеса.

Воспроизведение аудио с помощью HTML5

В Интернете аудио не так популярно, как видео. Мы можем снять на персональную камеру видео, которое наберет миллионы просмотров на вебсайтах, подобных http://www.youtube.com, но добиться того же результата, выложив в сеть звуковой файл, почти невозможно. Однако аудио все же присутствует, и у него есть свой рынок в Сети, включающий в себя радиошоу и подкасты.

В HTML5 появился новый элемент, позволяющий воспроизводить аудио в документах HTML. Разумеется, элемент носит название `<audio>`, а его характеристики почти идентичны характеристикам элемента `<video>`.

Листинг 5.11. Простейший HTML-код для встраивания аудио

```
<!DOCTYPE html>
<html lang="ru">
<head>
  <title>Аудиопроигрыватель</title>
</head>
<body>
<section id="player">
  <audio src="http://minkbooks.com/content/beach.mp3" controls>
  </audio>
</section>
</body>
</html>
```

Элемент <audio>

Элемент `<audio>` работает так же, как и элемент `<video>`, и у них много общих атрибутов:

- `src`. Задает URL-адрес файла, который будет воспроизводиться на странице. Так же, как при использовании элемента `<video>`, этот атрибут обычно заменяют элементом `<source>`, для того чтобы перечислить

ссылки на несколько файлов в разных аудиоформатах, из которых браузер сможет выбрать наиболее подходящий;

- `controls`. Активирует интерфейс, по умолчанию предоставляемый каждым браузером;
- `autoplay`. Если этот атрибут присутствует, браузер автоматически начинает воспроизведение аудио, как только это становится возможным;
- `loop`. Если этот атрибут присутствует, браузер начнет воспроизведение аудио с начала, как только оно достигнет конца;
- `preload`. Может принимать три значения: `none`, `metadata` и `auto`. Первое указывает, что аудио кэшировать не нужно (обычно используется с целью минимизации расхода трафика). Второе значение, `metadata`, рекомендует браузеру загрузить некоторую информацию о ресурсе (например, продолжительность). Третье значение, `auto`, устанавливается по умолчанию и заставляет браузер загружать файл сразу же, как только это становится возможным.

И снова нам приходится говорить о кодеках, потому что кода из листинга 5.11 должно быть более чем достаточно для встраивания аудио, однако в реальной жизни это не так. MP3 распространяется с коммерческой лицензией, следовательно, не поддерживается такими браузерами, как Firefox и Opera. Vorbis (аудиокодек из контейнера OGG) поддерживается вышеупомянутыми браузерами, но не Safari и не Internet Explorer. Поэтому нам снова приходится прибегать к помощи элемента `<source>` и добавлять ссылки минимум на два файла в разных форматах, из которых браузер сможет выбрать подходящий.

Листинг 5.12. Два источника одного и того же аудио

```
<!DOCTYPE html>
<html lang="ru">
<head>
  <title>Аудиопроигрыватель</title>
</head>
<body>
<section id="player">
  <audio id="media" controls>
    <source src="http://minkbooks.com/content/beach.mp3">
    <source src="http://minkbooks.com/content/beach.ogg">
  </audio>
```

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

Код из листинга 5.12 будет воспроизводить аудио в любом браузере, используя элементы управления по умолчанию. Если невозможно воспроизвести MP3, то браузер загрузит версию OGG, и наоборот. Не забывайте только, что использование MP3, так же как и MP4 для видео, ограничивается коммерческими лицензиями, поэтому встраивать медиафайлы в этих форматах можно только в определенных обстоятельствах, определяемых конкретной лицензией.

Поддержка бесплатных аудиокодеков (таких как Vorbis) становится все обширнее, однако для того, чтобы доселе неизвестный формат завоевал звание сетевого стандарта, необходимо довольно много времени.

Программирование аудиопроигрывателя

API-интерфейс мультимедиа поддерживает как видео, так и аудио. Каждое событие, метод и свойство, добавленные для поддержки видео, работают также и с аудиофайлами. Следовательно, в нашем шаблоне нужно только заменить элемент `<video>` элементом `<audio>`, и мы моментально получим готовый аудиопроигрыватель.

Листинг 5.13. Шаблон для аудиопроигрывателя

```
<!DOCTYPE html>
<html lang="ru">
<head>
  <title>Audio Player</title>
  <link rel="stylesheet" href="player.css">
  <script src="player.js"></script>
</head>
<body>
<section id="player">
  <audio id="media">
    <source src="http://minkbooks.com/content/beach.mp3">
    <source src="http://minkbooks.com/content/beach.ogg">
  </audio>
  <nav>
    <div id="buttons">
```

продолжение ⇩

Листинг 5.13 *(продолжение)*

```
      <button type="button" id="play">Play</button>
    </div>
    <div id="bar">
      <div id="progress"></div>
    </div>
    <div style="clear: both"></div>
  </nav>
</section>
</body>
</html>
```

В новом шаблоне из листинга 5.13 появилось только одно изменение: мы добавили элемент `<audio>` и соответствующие источники. Весь остальной код остался прежним, включая ссылки на внешние файлы. Больше ничего менять не нужно, так как события, методы и свойства одинаковы для обоих типов медиафайлов.

САМОСТОЯТЕЛЬНО

Создайте новый файл с именем audio.html, скопируйте код из листинга 5.13 в этот файл и откройте его в своем браузере. Для запуска аудиопроигрывателя можно использовать те же файлы player.css и player.js, которые вы создали раньше.

Краткий справочник. Видео и аудио

Видео и аудио — неотъемлемые составляющие Сети. В спецификацию HTML5 входят все элементы, необходимые для встраивания мультимедиа в веб-приложения.

Элементы

В HTML5 появились специальный API-интерфейс для доступа к библиотеке мультимедиа, а также два новых элемента HTML, предназначенные для обработки мультимедиа.

- `<video>`. Позволяет вставлять в HTML-документы видеофайлы;

- `<audio>`. Позволяет вставлять в HTML-документы аудиофайлы.

Атрибуты

Спецификация также предусматривает специальные атрибуты для элементов `<video>` и `<audio>`:

- `src`. Определяет URL-адрес встраиваемого файла мультимедиа. Можно также использовать внутри выбранного мультимедийного элемента (`<video>` или `<audio>`) элемент `<source>`, для того чтобы указать несколько источников и позволить браузеру выбрать подходящий формат для воспроизведения;
- `controls`. Этот атрибут (если он присутствует) активирует элементы управления мультимедиа, предоставляемые браузером по умолчанию. В каждом браузере предусмотрен собственный набор функциональности, включающий в себя, например, кнопки Play (Воспроизвести) и Pause (Пауза) или индикатор прогресса;
- `autoplay`. Этот атрибут (если он присутствует) заставляет браузер начинать воспроизведение мультимедиа, как только это становится возможным;
- `loop`. Заставляет браузер бесконечно циклически воспроизводить мультимедиа;
- `preload`. Подсказывает браузеру, как правильно обрабатывать загрузку медиафайла. Может принимать три значения: `none`, `metadata` и `auto`. Значение `none` приказывает браузеру подождать с загрузкой файла до тех пор, пока пользователь не инициирует загрузку мультимедиа. Значение `metadata` заставляет браузер загрузить основную информацию о мультимедийном файле. Значение `auto` приказывает браузеру начать загрузку файла как можно скорее.

Атрибуты элемента <video>

Некоторые атрибуты относятся только к элементу `<video>`:

- `poster`. Позволяет указать изображение, которое будет отображаться на экране до тех пор, пока пользователь не инициирует воспроизведение видео;
- `width`. Определяет ширину окна видеопроигрывателя (в пикселах);
- `height`. Определяет высоту окна видеопроигрывателя (в пикселах).

События

Чаще всего используются следующие события этого API:

- `progress`. Срабатывает периодически и информирует о прогрессе загрузки медиафайла;
- `canplaythrough`. Срабатывает в момент, когда становится понятно, что медиафайл целиком можно воспроизвести без перерывов;
- `canplay`. Срабатывает, когда медиафайл готов к воспроизведению. Но, в отличие от предыдущего события, готовностью к воспроизведению считает загрузку всего лишь пары кадров;
- `ended`. Срабатывает, когда воспроизведение мультимедиа завершается;
- `pause`. Срабатывает, когда пользователь приостанавливает воспроизведение;
- `play`. Срабатывает в момент начала воспроизведения;
- `error`. Срабатывает при возникновении ошибки. Событие доставляется в элемент `<source>` (если таковой существует), соответствующий источнику медиафайла, вызвавшего ошибку.

Методы

Чаще всего используются следующие методы этого API:

- `play()`. Запускает или возобновляет воспроизведение медиафайла;
- `pause()`. Приостанавливает воспроизведение;
- `load()`. Загружает медиафайл в динамические приложения;
- `canPlayType(тип)`. Позволяет узнать, поддерживается данный формат файла браузером или нет. Если браузер не способен воспроизвести медиафайл, он возвращает пустую строку; в противном случае возвращает строку `maybe` или `probably` в зависимости от того, какова вероятность успешного воспроизведения файла.

Свойства

Чаще всего используются следующие свойства этого API:

- `paused`. Возвращает значение `true`, если воспроизведение мультимедиа приостановлено или еще не начиналось;
- `ended`. Возвращает значение `true`, если видео было воспроизведено до конца;

- `duration`. Возвращает продолжительность мультимедиа в секундах;
- `currentTime`. Может как возвращать, так и принимать значение. Оно либо информирует о текущей позиции воспроизведения медиафайла, либо устанавливает новую позицию, с которой продолжается воспроизведение;
- `error`. Возвращает значение ошибки в случае, если произошел сбой;
- `buffered`. Предоставляет информацию о том, какая часть файла уже загружена в буфер. Возвращаемое значение представляет собой массив, содержащий данные обо всех загруженных фрагментах мультимедиа. Если пользователь переходит к части медиафайла, которая еще не была загружена, браузер продолжает загрузку с этой позиции. Для доступа к элементам массива можно использовать атрибуты `end()` и `start()`. Например, код `buffered.end(0)` вернет продолжительность (в секундах) первой загруженной части мультимедиа, содержащейся в буфере.

6 Формы и API Forms

Веб-формы HTML

В Web 2.0 все делается для пользователя и ради пользователя. А это автоматически означает, что центром внимания разработчика становятся интерфейсы. Мы начинаем задумываться, как сделать их более интуитивно понятными, более естественными, более практичными и, разумеется, более красивыми. Формы — это самый важный вид интерфейсов из существующих. Они позволяют пользователям вводить данные, принимать решения, обмениваться информацией и менять поведение приложения. За последние несколько лет создано множество библиотек и примеров кода, предназначенных для обработки форм на компьютере пользователя. HTML5 стандартизирует эти возможности, предоставляет новые атрибуты, элементы и целые API-интерфейсы. Теперь функциональность, предназначенная для обработки данных в формах в режиме реального времени, встраивается прямо в браузер и полностью описывается стандартами.

Элемент <form>

Сами формы не претерпели особых изменений. Структура осталась все той же, но в HTML5 были добавлены новые элементы, типы ввода

и атрибуты, расширяющие функциональность форм и добавляющие возможности, которые раньше приходилось программно реализовывать в веб-приложениях.

Листинг 6.1. Обычная структура формы

```
<!DOCTYPE html>
<html lang="ru">
<head>
  <title>Формы</title>
</head>
<body>
  <section id="form">
    <form name="myform" id="myform" method="get">
      <input type="text" name="name" id="name">
      <input type="submit" value="Send">
    </form>
  </section>
</body>
</html>
```

В листинге 6.1 мы создали шаблон простейшей формы. Как видите, структура формы и атрибуты по сравнению с предыдущими спецификациями не поменялись. Однако теперь с элементом `<form>` можно использовать новые атрибуты:

- `autocomplete`. Это старый атрибут, который теперь описан в стандарте. Он может принимать два значения: `on` и `off`. Значение по умолчанию равно `on`. Когда значение атрибута равно `off`, составляющие форму элементы `<input>` не заполняются автоматически, то есть для них не отображаются списки введенных ранее значений. Атрибут можно добавлять как к элементу `<form>`, так и независимо к любому элементу `<input>`;
- `novalidate`. Одна из особенностей форм HTML5 — встроенная возможность валидации. Правильность содержимого форм проверяется автоматически. Чтобы запретить такое поведение, можно добавить атрибут `novalidate`. Ту же задачу, но для отдельных элементов `<input>`, выполняет атрибут `formnovalidate`. Оба атрибута относятся к логическому типу, поэтому для них не нужно указывать значения.

Элемент <input>

Самый важный элемент формы — `<input>`, способный менять свои характеристики благодаря атрибуту `type`. Этот атрибут определяет, данные какого типа ожидают от пользователя. Раньше доступны были лишь несколько типов: многофункциональный `text` и парочка более специфических, таких как `password` и `submit`. В HTML5 количество вариантов увеличилось, а функциональность элемента значительно расширилась.

В HTML5 эти новые типы не только указывают, какие именно данные должен ввести пользователь, но также сообщают браузеру, как обрабатывать получаемую информацию. Браузер обрабатывает введенные данные в зависимости от значения атрибута `type` и выполняет или не выполняет их валидацию согласно тому, какие настройки формы вы задали.

Атрибут `type` можно использовать совместно с другими атрибутами для более тонкой настройки ограничений и управления вводимыми данными в режиме реального времени.

САМОСТОЯТЕЛЬНО

Создайте новый HTML-файл и скопируйте в него шаблон из листинга 6.1. Для проверки работы каждого из рассматриваемых далее типов ввода заменяйте элементы <input> в шаблоне соответствующими элементами из следующих листингов и открывайте файл в браузере. Способы обработки вводимых типов в разных браузерах могут различаться, поэтому мы рекомендуем проверять код во всех доступных вам браузерах.

Тип email

Почти в каждой форме, существующей в Сети, можно обнаружить поле для ввода адреса электронной почты. До сих пор для обработки данных такого типа можно было использовать только тип `text`. Тип `text` представляет собой текст любого вида, а не какие-то специфические строки, поэтому обрабатывать вводимые данные приходилось в коде JavaScript. В противном случае мы не могли быть уверены в том, что пользователь ввел адрес электронной почты в допустимом формате. Однако теперь благодаря новому типу `email` браузер сам может позаботиться о валидации указанного пользователем адреса.

Листинг 6.2. Тип email

```
<input type="email" name="myemail" id="myemail">
```

Когда пользователь введет текст в поле, сгенерированном кодом из листинга 6.2, эти данные будут проверены браузером на соответствие формату адреса электронной почты. Если валидация завершится неудачей, содержимое формы не будет отправлено серверу.

Спецификация HTML5 не описывает, как браузеры должны реагировать на ввод недопустимых данных. В одних браузерах вокруг элемента `<input>` с ошибочными данными появляется красная рамка, в других — синяя. Всегда существует способ настроить обработку ошибок, и мы познакомимся с доступными вариантами чуть позже.

Тип search

Тип `search` никак не контролирует ввод данных — это всего лишь способ сообщить браузеру о предназначении поля. Некоторые браузеры меняют представление по умолчанию этого элемента, давая пользователям дополнительную подсказку, для чего используется данное поле.

Листинг 6.3. Тип search

```
<input type="search" name="mysearch" id="mysearch">
```

Тип URL

Этот тип работает точно так же, как и `email`, но предназначен для сетевых адресов. Он считает допустимыми только абсолютные URL-адреса и возвращает ошибку, если значение не соответствует этому формату (листинг 6.4).

Листинг 6.4. Тип url

```
<input type="url" name="myurl" id="myurl">
```

Тип tel

Этот тип предназначен для обработки телефонных номеров. В отличие от типов `email` и `url`, тип `tel` не требует какого-то определенного

синтаксиса. Он всего лишь сообщает браузеру о том, ввод данных какого вида в это поле ожидается, на случай, если приложению потребуется скорректировать представление поля с учетом особенностей устройства, на котором оно выполняется (листинг 6.5).

Листинг 6.5. Тип tel

```
<input type="tel" name="myphone" id="myphone">
```

Тип number

Как несложно догадаться исходя из названия, тип `number` допускает ввод только числовых данных. С этим типом можно использовать несколько полезных новых атрибутов:

- `min`. Значение этого атрибута определяет минимальное допустимое значение для данного поля;
- `max`. Значение этого атрибута определяет максимальное допустимое значение для данного поля;
- `step`. Значение этого атрибута определяет шаг увеличения или уменьшения значения данного поля. Например, вы задали шаг, равный 5, минимальное значение 0 и максимальное значение 10. В этом случае браузер не позволит указать значения между 0 и 5 и между 5 и 10 (листинг 6.6).

Листинг 6.6. Тип number

```
<input type="number" name="mynumber" id="mynumber" min="0"
max="10" step="5">
```

Оба атрибута, `min` и `max`, необязательные, а значение по умолчанию для атрибута `step` равно 1.

Тип range

Этот тип ввода заставляет браузер создавать на странице новый тип элемента управления, не существовавший до сих пор. В соответствии со своим названием он позволяет пользователю выбрать значение из некоего числового диапазона. Обычно на экране этот элемент принимает вид

ползунка или стрелочек, увеличивающих и уменьшающих значение, однако стандартного общего для всех дизайна пока не существует.

Для определения границ диапазона используются атрибуты типа `range` `min` и `max`. Помимо этого можно добавить атрибут `step`, чтобы задать определенный шаг увеличения или уменьшения значения элемента (листинг 6.7).

Листинг 6.7. Тип range

```
<input type="range" name="mynumbers" id="mynumbers" min="0"
max="10" step="5">
```

Первоначальное значение можно установить с помощью давно существующего атрибута `value`, а в коде JavaScript можно запрограммировать отображение чисел на экране, для того чтобы пользователю было проще ориентироваться. Мы поэкспериментируем с этими возможностями, а также с новым элементом `<output>` чуть позже.

Тип date

Это еще один тип ввода, создающий на экране новый элемент управления. Он был добавлен в спецификацию для упрощения ввода дат. В браузерах этот элемент управления выглядит как поле, при каждом щелчке на котором открывается календарь. В календаре пользователь также щелчком выбирает день, и полное значение даты вставляется в поле. Этот элемент используется, например, в приложениях, где пользователь выбирает дату полета или другого события. Мы всего лишь вставляем в свой документ элемент `<input>`, и благодаря типу `date` браузер автоматически создает удобный для пользователя календарь (листинг 6.8).

Листинг 6.8. Тип date

```
<input type="date" name="mydate" id="mydate">
```

Интерфейс данного элемента управления в спецификации не описывается. В каждом браузере генерируется собственный интерфейс, и иногда дизайн календаря меняется, адаптируясь к устройству, на котором выполняется приложение. Чаще всего для генерации значения используется синтаксис «год–месяц–день», и от пользователя ожидают ввода в таком же формате.

Тип week

Этот тип генерирует интерфейс, аналогичный создаваемому `date`, но позволяет выбирать только определенную неделю. Чаще всего используется синтаксис «2011–W50», где 2011 — год, а 50 — номер недели (листинг 6.9).

Листинг 6.9. Тип week

```
<input type="week" name="myweek" id="myweek">
```

Тип month

Этот тип аналогичен предыдущему, но позволяет выбирать только конкретный месяц. Обычно он ожидает значение в формате «год–месяц» (листинг 6.10).

Листинг 6.10. Тип month

```
<input type="month" name="mymonth" id="mymonth">
```

Тип time

Тип `time` аналогичен типу `date`, но предназначен для обработки значения времени. Он принимает данные в формате часов и минут, но его поведение в настоящее время сильно зависит от конкретного браузера. Обычно ожидается значение, соответствующее синтаксису «часы:минуты:секунды», допустимо также сокращенное значение «часы:минуты» (листинг 6.11).

Листинг 6.11. Тип time

```
<input type="time" name="mytime" id="mytime">
```

Тип datetime

Тип `datetime` предназначен для ввода полной даты, включая время и часовой пояс (листинг 6.12).

Листинг 6.12. Тип datetime

```
<input type="datetime" name="mydatetime" id="mydatetime">
```

Тип datetime-local

Тип `datetime-local` — то же самое, что `datetime`, но без часовой зоны (листинг 6.13).

Листинг 6.13. Тип datetime-local

```
<input type="datetime-local" name="mylocaldatetime"
id="mylocaldatetime">
```

Тип color

Помимо типов для обработки даты и времени, существует также тип, предоставляющий стандартный интерфейс выбора цвета. Обычно в таком поле ожидается значение в шестнадцатеричном формате, такое как `#00FF00` (листинг 6.14).

Листинг 6.14. Тип color

```
<input type="color" name="mycolor" id="mycolor">
```

HTML5 не описывает стандартного интерфейса для типа `color`, однако возможно, что во всех браузерах будет использоваться обычная сетка с набором базовых цветов.

Новые атрибуты

Для того чтобы некоторые типы ввода выполняли свою задачу, к ним необходимо добавлять атрибуты, например изученные ранее `min`, `max` и `step`. Другим типам ввода помощь атрибутов требуется для более эффективной работы или для того, чтобы указать на их значимость в процессе валидации. Мы уже познакомились с некоторыми атрибутами, такими как `novalidate`, который запрещает валидацию во всей форме, и `formnovalidate`, запрещающим валидацию отдельных элементов.

Атрибут `autocomplete`, также описанный ранее, обеспечивает дополнительную безопасность формы целиком или отдельных ее элементов. Теперь давайте познакомимся с остальными атрибутами, появившимися в HTML5.

Атрибут placeholder

Обычно используемый для элементов ввода типа `search`, но также допустимый для текстовых полей, атрибут `placeholder` представляет собой короткую подсказку — слово или фразу, помогающие пользователю правильно ввести ожидаемые данные. Значение данного атрибута визуализируется браузером внутри поля и исчезает, когда пользователь переводит фокус ввода на соответствующий элемент (листинг 6.15).

Листинг 6.15. Атрибут placeholder

```
<input type="search" name="mysearch" id="mysearch"
placeholder="введите критерий поиска">
```

Атрибут required

Этот атрибут логического типа не позволяет подтвердить отправку формы, если поле осталось пустым. Например, ранее мы привели пример поля типа `email`, предназначенного для ввода адреса электронной почты. Браузер проверяет, соответствует ли введенная информация формату адреса электронной почты, но при этом считает допустимым и пустое значение поля. Если же добавить атрибут `required`, то браузер будет требовать обязательного заполнения этого поля в дополнение к проверке форматирования введенных данных (листинг 6.16).

Листинг 6.16. Поле ввода типа email теперь стало обязательным для заполнения

```
<input type="email" name="myemail" id="myemail" required>
```

Атрибут multiple

Атрибут `multiple` — это еще один булев атрибут, который можно использовать с некоторыми типами ввода (например, `email` и `file`), чтобы

разрешать пользователю вводить в одном поле несколько значений. Для того чтобы введенная информация успешно прошла валидацию, значения необходимо разделить запятой.

Листинг 6.17. Поле ввода типа email допускает ввод нескольких адресов, разделенных запятой

```
<input type="email" name="myemail" id="myemail" multiple>
```

Код в листинге 6.17 создает поле, в котором пользователь может ввести несколько значений, разделив их запятыми. Каждое из значений будет проверено браузером на соответствие формату адреса электронной почты.

Атрибут autofocus

Эту возможность большинство разработчиков раньше реализовывали с помощью метода JavaScript `focus()`. Такой подход был довольно эффективным, но фокус при этом переключался всегда, даже если пользователь работал с другим элементом. Это раздражало, но бороться с этим до сих пор было невозможно. Атрибут `autofocus` переводит фокус вебстраницы на выбранный элемент с учетом текущей ситуации: фокус не меняется, если пользователь уже выбрал другой элемент и работает с ним (листинг 6.18).

Листинг 6.18. Атрибут autofocus в поле поиска

```
<input type="search" name="mysearch" id="mysearch" autofocus>
```

Атрибут pattern

Атрибут `pattern` предназначен для проверки вводимых данных. Он позволяет настраивать правила валидации добавлением регулярных выражений. Некоторые из рассмотренных типов ввода проверяют определенные форматы строк, но предположим, например, что перед вами стоит задача проверить пятизначный почтовый индекс. Стандартного типа ввода для данных в таком формате не существует. Атрибут `pattern` позволяет создавать собственные правила для проверки подобных значений. Также можно добавить сообщение об ошибке в атрибуте `title` (листинг 6.19).

Листинг 6.19. Настройка проверки нестандартных типов с помощью атрибута pattern

```
<input pattern="[0-9]{5}" name="pcode" id="pcode" title="insert
the 5 numbers of your postal code">
```

ВНИМАНИЕ

Регулярные выражения — это весьма сложная тема. Для того чтобы узнать о них больше, зайдите на наш веб-сайт и пройдите по ссылкам для этой главы.

Атрибут form

Атрибут `form` — это очень удобное нововведение, позволяющее объявлять элементы формы за пределами тегов `<form>`. До настоящего времени для построения формы приходилось записывать открывающий и закрывающий теги `<form>` и между ними объявлять все нужные элементы. В HTML5 элементы можно вставлять в любое место документа и привязывать их к форме, ссылаясь на ее название в атрибуте `form` (листинг 6.20).

Листинг 6.20. Объявление элементов формы в разных местах документа

```
<!DOCTYPE html>
<html lang="ru">
<head>
  <title>Формы</title>
</head>
<body>
  <nav>
    <input type="search" name="mysearch" id="mysearch" form="myform">
  </nav>
  <section id="form">
    <form name="myform" id="myform" method="get">
      <input type="text" name="name" id="name">
      <input type="submit" value="Send">
    </form>
  </section>
</body>
</html>
```

Новые элементы форм

Мы изучили появившиеся в HTML5 новые типы ввода, и нам осталось познакомиться с новыми элементами HTML, улучшающими и расширяющими функциональность форм.

Элемент <datalist>

Элемент `<datalist>` используется только в формах. Он позволяет заранее построить список пунктов, которые в дальнейшем будут предлагаться в качестве вариантов заполнения полей ввода (для этого нужно добавить атрибут `list`) (листинг 6.21).

Листинг 6.21. Построение списка

```
<datalist id="mydata">
  <option value="123123123" label="Phone 1">
  <option value="456456456" label="Phone 2">
</datalist>
```

После того как вы объявили элемент `<datalist>`, остается лишь сослаться на этот список пунктов из элемента `<input>`, используя атрибут `list`.

Листинг 6.22. Список возможных значений предлагается с помощью атрибута list

```
<input type="tel" name="myphone" id="myphone" list="mydata">
```

В элементе из листинга 6.22 отображается список возможных значений, из которых пользователь может выбрать подходящее.

> **ВНИМАНИЕ**
>
> В настоящее время элемент <datalist> реализован только в браузерах Opera и Firefox Beta.

Элемент <progress>

Этот элемент не обязательно использовать только с формами, но поскольку он иллюстрирует прогресс выполнения задачи, а задачи чаще

всего инициируются и обрабатываются с помощью форм, его можно включить в группу элементов форм.

Элемент `<progress>` принимает два атрибута, устанавливающие его статус и лимиты. Атрибут `value` указывает, какая доля задачи уже выполнена, а атрибут `max` содержит значение, достижение которого соответствует завершению задачи.

Элемент <meter>

Аналогично элементу `<progress>`, элемент `<meter>` используется для отображения шкалы, однако это не шкала выполнения задачи. Данный элемент предназначен для отображения известного диапазона, например, использования пропускной способности.

С элементом `<meter>` связаны несколько атрибутов: `min` и `max` устанавливают границы диапазона, `value` определяет измеряемое значение, а `low`, `high` и `optimum` используются для сегментирования диапазона и определения позиции, которая будет соответствовать оптимальному значению.

Элемент <output>

Этот элемент представляет собой результат вычисления. Обычно он используется для отображения результатов обработки каких-то значений элементами формы. Атрибут `for` позволяет связать элемент `<output>` с исходными элементами, участвующими в расчетах, однако чаще всего ссылки на элементы создаются и модифицируются в коде JavaScript. Синтаксис данного элемента таков: `<output>значение</output>`.

API-интерфейс Forms (Формы)

Вы вряд ли удивитесь тому, что, как и у любых других составляющих HTML5, у форм есть собственный API JavaScript, позволяющий настраивать любые аспекты обработки и валидации их содержимого.

Существуют разные подходы к управлению процессом валидации в HTML5. Можно использовать типы ввода, по умолчанию требующие валидации (например, `email`), или превращать обычное поле типа `text` в обязательное добавлением атрибута `required`. Кроме того, можно при-

менять особые типы, позволяющие настраивать детали валидации, такие как `pattern`. Но когда дело доходит до более сложных механизмов валидации (например, объединения полей или проверки результатов вычислений), приходится полагаться на возможности новых ресурсов, предоставляемых API-интерфейсом.

Метод setCustomValidity()

Браузеры с поддержкой HTML5 умеют выводить сообщения об ошибке, когда пользователь пытается отправить форму с недопустимым содержимым в каком-либо поле. Однако благодаря методу `setCustomValidity(сообщение)` мы можем создавать собственные сообщения об ошибках, отвечающие требованиям к валидации наших приложений.

Этот метод позволяет настраивать обработку ошибок с выводом сообщения при попытке отправить форму. Пустое сообщение в методе очищает состояние ошибки.

Листинг 6.23. Настройка сообщений об ошибках

```
<!DOCTYPE html>
<html lang="ru">
<head>
  <title>Формы</title>
  <script>
    function initiate(){
      name1=document.getElementById("firstname");
      name2=document.getElementById("lastname");
      name1.addEventListener("input", validation, false);
      name2.addEventListener("input", validation, false);
      validation();
    }
    function validation(){
      if(name1.value=='' && name2.value==''){
        name1.setCustomValidity('введите хотя бы одно имя');
        name1.style.background='#FFDDDD';
      }else{
        name1.setCustomValidity('');
        name1.style.background='#FFFFFF';
      }
    }
```

продолжение ⇗

Листинг 6.23 *(продолжение)*

```
    window.addEventListener("load", initiate, false);
  </script>
</head>
<body>
  <section id="form">
    <form name="registration" method="get">
      Имя:
      <input type="text" name="firstname" id="firstname">
      Фамилия:
      <input type="text" name="lastname" id="lastname">
      <input type="submit" id="send" value="отправить">
    </form>
  </section>
</body>
</html>
```

Код из листинга 6.23 иллюстрирует ситуацию комплексной валидации. Создаются два поля ввода, в которых пользователь должен указать свои имя и фамилию. Однако ошибка валидации возникает только в одном случае — когда оба поля пусты. Чтобы форма была успешно проверена и отправлена, пользователь должен ввести хотя бы одно значение — имя или фамилию.

В подобных случаях невозможно использовать атрибут `required`, так как мы не знаем, какое поле решит заполнить пользователь. Только с помощью JavaScript-кода и индивидуального программирования обработки ошибок можно создать эффективный механизм валидации для подобного сценария.

Наш код начинает выполняться, когда срабатывает событие `load`. Для обработки этого события вызывается функция `initiate()`. Она создает ссылки на два элемента `<input>` и для обоих добавляет прослушиватели для отслеживания события `input`. Прослушиватели вызывают функцию `validation()` каждый раз, когда пользователь вводит данные в соответствующие поля.

Поскольку при первоначальной загрузке документа элементы `<input>` не содержат никаких данных, необходимо задать условие, запрещающее отправку формы до того, как хотя бы одно из полей будет заполнено. По этой причине в начале кода вызывается функция `validation()`, проверяющая описанное условие. Если вместо обоих имен она находит пустые строки, то создает ошибку и меняет фоновый цвет элемента `firstname` на светло-красный.

Однако если позднее условие ошибки перестает выполняться, так как пользователь заполнил по меньшей мере одно поле, ошибка сбрасывается, а фоновый цвет элемента `firstname` вновь меняется на белый.

Важно понимать, что единственное изменение, происходящее во время обработки содержимого формы, заключается в переключении фонового цвета. Сообщение об ошибке, объявленное с помощью `setCustomValidity()`, будет отображаться только в том случае, если пользователь попытается отправить форму.

САМОСТОЯТЕЛЬНО

В целях тестирования мы добавили JavaScript-код прямо в документ. Таким образом, для тестирования примера вам нужно всего лишь скопировать код из листинга 6.23 в пустой HTML-файл и открыть этот файл в своем браузере.

ВНИМАНИЕ

API Forms (Формы) пока что находится на этапе разработки, поэтому в зависимости от того, насколько продвинется процесс к тому моменту, как вы станете читать эту книгу, вам может потребоваться множество проверок примеров кода из этой главы, чтобы увидеть, как они будут работать в разных браузерах.

Событие invalid

Каждый раз, когда пользователь подтверждает отправку формы, в случае обнаружения поля с недопустимым содержимым срабатывает событие. Оно называется `invalid` и связано с элементом, ставшим причиной ошибки. Для настройки собственного варианта реагирования на данное событие можно зарегистрировать специальный обработчик события, как в листинге 6.24.

Листинг 6.24. Наша собственная система валидации

```
<!DOCTYPE html>
<html lang="ru">
<head>
  <title>Формы</title>
```

продолжение ⇩

Листинг 6.24 *(продолжение)*

```
  <script>
    function initiate(){
      age=document.getElementById("myage");
      age.addEventListener("change", changerange, false);

      document.information.addEventListener("invalid", validation, true);
      document.getElementById("send").addEventListener("click",
                                sendit, false);
    }
    function changerange(){
      var output=document.getElementById("range");
      var calc=age.value-20;
      if(calc<20){
        calc=0;
        age.value=20;
      }
      output.innerHTML=calc+' to '+age.value;
    }
    function validation(e){
      var elem=e.target;
      elem.style.background='#FFDDDD';
    }
    function sendit(){
      var valid=document.information.checkValidity();
      if(valid){
        document.information.submit();
      }
    }
    window.addEventListener("load", initiate, false);
  </script>
</head>
<body>
  <section id="form">
    <form name="information" method="get">
      Имя:
      <input pattern="[A-Za-z]{3,}" name="nickname" id="nickname"
maxlength="10" required>
      Email:
      <input type="email" name="myemail" id="myemail" required>
      Возрастной диапазон:
```

```
      <input type="range" name="myage" id="myage" min="0" max="80"
step="20" value="20">
      <output id="range">от 0 до 20</output>
      <input type="button" id="send" value="отправить">
    </form>
  </section>
</body>
</html>
```

В листинге 6.24 мы создали новую форму с тремя полями ввода, в которых запрашиваются имя, адрес электронной почты и возраст в диапазоне 20 лет.

У поля `nickname` три дополнительных атрибута валидации: `pattern`, разрешающий ввод не менее 3 символов и проверяющий, что это буквы от A до Z (в верхнем или нижнем регистре), `maxlength`, ограничивающий количество вводимых символов десятью, и `required`, создающий ошибку в случае, если поле остается незаполненным. На поле `email` накладываются естественные ограничения, диктуемые его природой, к нему также применяется атрибут `required`. Для поля ввода `range` мы добавили атрибуты `min`, `max`, `step` и `value`, устанавливающие условия диапазона.

Кроме того, мы объявляем элемент `<output>`, в целях информирования показывающий выбранный диапазон на экране.

JavaScript-код в этом примере выполняет очень простую задачу: когда пользователь щелкает на кнопке Отправить, для всех полей с недопустимым содержимым срабатывает событие `invalid`, а функция `validation()` меняет фоновый цвет этих полей на светло-красный.

Давайте внимательнее рассмотрим процесс обработки. Код начинает выполняться после срабатывания типичного события `load`, то есть после полной загрузки документа. Выполняется функция `initiate()`, и добавляются три прослушивателя для событий `change`, `invalid` и `click`.

Каждый раз, когда элемент формы по какой-либо причине меняется, для него срабатывает событие `change`. Когда это происходит для поля ввода `range`, мы захватываем данное событие и вызываем функцию `changerange()`. Таким образом, когда пользователь выбирает новый возрастной диапазон, передвигая ползунок этого элемента управления, функция `changerange()` вычисляет новые граничные значения диапазона. С этим элементом ввода данных связаны двадцатилетние периоды, например от 0 до 20 или от 20 до 40. Однако элемент возвращает только одно значение, такое как 20, 40, 60 или 80. Для вычисления начального значения диапазона мы вычитаем 20 из текущего значения элемента

`range` с помощью формулы `age.value` – 20 и сохраняем результат в переменной `calc`. Минимальный возможный период — от 0 до 20; таким образом, в условном операторе `if` мы проверяем это условие и не позволяем устанавливать период меньшей длины (для того чтобы понять, как это работает, изучите код функции `changerange()`).

Второй прослушиватель добавляется в функцию `initiate()` и также отслеживает событие `invalid`, а в случае, если событие срабатывает, вызывает функцию `validation()`. Эта функция меняет фоновый цвет полей с недопустимыми значениями. Вспомните, что событие срабатывает при щелчке на кнопке отправки формы и ссылается не на форму и не на кнопку отправки, а на элемент ввода, в котором произошла ошибка. В функции `validation()` мы захватываем эту ссылку и сохраняем в переменной `elem`, используя для этого переменную `e` и свойство `target`. Таким образом, конструкция `e.target` возвращает ссылку на элемент ввода с недопустимым значением. В следующей строке функции `validation()` фоновый цвет меняется только для этого элемента.

Вернемся к функции `initiate()`, для того чтобы проанализировать еще один прослушиватель. Чтобы обладать полным контролем над подачей формы и моментом валидации, мы создали обычную кнопку, а не кнопку со специальной функциональностью `submit`. Когда пользователь щелкает на нашей кнопке, форма отправляется на сервер, однако происходит это только в том случае, если все элементы заполнены допустимыми значениями. Прослушиватель события `click`, который мы связали с кнопкой внутри функции `initiate()`, после щелчка на кнопке вызывает функцию `sendit()`. Используя метод `checkValidity()`, мы заставляем браузер выполнить валидацию и отправляем форму через традиционный метод `submit()` только в том случае, если условий ошибки не осталось.

Благодаря JavaScript-коду, добавленному в этот документ, мы взяли в свои руки контроль над всем процессом валидации, настроив по своему усмотрению каждый аспект и поведение браузера.

Валидация в режиме реального времени

Открыв файл с шаблоном из листинга 6.24 в браузере, вы заметите, что валидация данных формы происходит не в режиме реального времени. Информация в полях проверяется только после того, как вы нажимаете кнопку отправки формы. Для того чтобы сделать процесс валидации более практичным, мы можем воспользоваться возможностями нескольких атрибутов, предоставляемых объектом `ValidityState`.

Листинг 6.25. Валидация вводимых данных в режиме реального времени

```
<!DOCTYPE html>
<html lang="ru">
<head>
  <title>Формы</title>
  <script>
    function initiate(){
      age=document.getElementById("myage");
      age.addEventListener("change", changerange, false);

     document.information.addEventListener("invalid", validation, true);
     document.getElementById("send").addEventListener("click", sendit, false);
     document.information.addEventListener("input", checkval, false);
    }
    function changerange(){
      var output=document.getElementById("range");
      var calc=age.value-20;
      if(calc<20){
        calc=0;
        age.value=20;
      }
      output.innerHTML=calc+' to '+age.value;
    }
    function validation(e){
      var elem=e.target;
      elem.style.background='#FFDDDD';
    }
    function sendit(){
      var valid=document.information.checkValidity();
      if(valid){
        document.information.submit();
      }
    }
    function checkval(e){
      var elem=e.target;
      if(elem.validity.valid){
        elem.style.background='#FFFFFF';
      }else{
        elem.style.background='#FFDDDD';
       }
     }
```

продолжение ⇩

Листинг 6.26 *(продолжение)*

```
    window.addEventListener("load", initiate, false);
  </script>
</head>
<body>
  <section id="form">
    <form name="information" method="get">
      Имя:
      <input pattern="[A-Za-z]{3,}" name="nickname" id="nickname"
             maxlength="10" required>
      Email:
      <input type="email" name="myemail" id="myemail" required>
      Возрастной диапазон:
      <input type="range" name="myage" id="myage" min="0" max="80"
                  step="20" value="20">
      <output id="range">от 0 до 20</output>
      <input type="button" id="send" value="отправить">
    </form>
  </section>
</body>
</html>
```

В листинге 6.25 мы добавили в нашу форму новый прослушиватель события `input`. Каждый раз, когда пользователь модифицирует поля, записывая в них новую информацию или меняя существующую, вызывается функция `checkval()`, обрабатывающая данное событие.

Функция `checkval()` также использует свойство `target` для создания ссылки на элемент, на котором сработало событие. Помимо этого, она контролирует допустимость значения в поле, проверяя состояние `valid` с помощью атрибута `validity`, для чего используется конструкция `elem.validity.valid`.

Состояние `valid` равно `true`, если элемент содержит допустимое значение, и `false` — в противном случае. Имея эту информацию, мы можем менять фоновый цвет элемента, в котором срабатывает событие `input`. Белый цвет указывает на допустимое значение, а светло-красный — на ошибочное.

Благодаря этому простому усовершенствованию теперь каждый раз, когда пользователь будет изменять информацию в элементах формы, она будет сразу же проходить валидацию, и результат (подсветка элемента

в зависимости от успешности валидации) будет отображаться на экране в режиме реального времени.

Ограничения валидности

В примере из листинга 6.25 мы проверяли состояние `valid`. Это состояние представляет собой атрибут объекта `ValidityState`, который возвращает информацию о том, допустимо ли значение элемента, с учетом всех существующих статусов валидности. Если все проверки на валидность выполняются, то атрибут `valid` возвращает значение `true`.

Существует восемь различных статусов валидности для различных условий:

- `valueMissing`. Равен `true`, если к элементу добавлен атрибут `required`, но поле ввода пусто;
- `typeMismatch`. Равен `true`, когда синтаксис введенных данных не соответствует указанному типу, например, если текст в поле ввода типа `email` не является адресом электронной почты;
- `patternMismatch`. Равен `true`, если введенные данные не соответствуют указанному для элемента шаблону;
- `tooLong`. Равен `true`, если для элемента объявлен атрибут `maxlength`, но длина введенных данных больше, чем значение атрибута;
- `rangeUnderflow`. Равен `true`, если для элемента объявлен атрибут `min`, но введенное значение меньше значения атрибута;
- `rangeOverflow`. Равен `true`, если для элемента объявлен атрибут `max`, но введенное значение больше значения атрибута;
- `stepMismatch`. Равен `true`, если для элемента объявлен атрибут `step`, но его значение не соответствует значениям других атрибутов, таких как `min`, `max` и `value`;
- `customError`. Равен `true`, когда мы настраиваем обработку специальных ошибок, например применяем изученный ранее метод `setCustomValidity()`.

Для проверки перечисленных статусов валидности необходимо использовать синтаксис `элемент.validity.статус` (где `статус` может принимать одно из значений из предыдущего списка). Эти атрибуты позволяют устанавливать точные причины ошибок в формах, как, например, в следующем фрагменте кода (листинг 6.26).

Листинг 6.26. Отображение специального сообщения об ошибке после проверки статуса валидности

```
function sendit(){
  var elem=document.getElementById("nickname");
  var valid=document.information.checkValidity();
  if(valid){
    document.information.submit();
  }else if(elem.validity.patternMismatch ||
           elem.validity.valueMissing){
    alert('длина имени должна быть не менее 3 символов');
  }
}
```

В листинге 6.26 мы модифицировали функцию `sendit()`, переведя контроль над вводимыми данными на более высокий уровень. Форма валидируется методом `checkValidity()`, и, если оказывается, что все элементы содержат допустимые значения, она отправляется на сервер с помощью метода `submit()`. В противном случае проверяются статусы валидности `patternMismatch` и `valueMissing` для элемента ввода `nickname`, и, если один из них или оба возвращают значение `true`, выводится сообщение об ошибке.

САМОСТОЯТЕЛЬНО

Замените функцию sendit() в шаблоне из листинга 6.25 новой функцией из листинга 6.26 и откройте этот HTML-файл в своем браузере для проверки.

Атрибут willValidate

В динамических приложениях возможны ситуации, когда определенные элементы вообще не требуют валидации. Например, так может случиться с кнопками, скрытыми элементами ввода или элементом `<output>`. Распознать это условие можно с помощью атрибута `willValidate`, используя синтаксис `элемент.willValidate`.

Краткий справочник. Формы и API Forms

Формы — это основной путь коммуникации между пользователями и веб-приложениями. В HTML5 были добавлены новые типы элементов `<input>`, новый API для валидации и обработки форм, а также атрибуты, совершенствующие данный интерфейс.

Типы

В некоторые из новых типов ввода, появившихся в HTML5, по умолчанию встроены условия валидации. Другие всего лишь объявляют о назначении соответствующего поля, что помогает браузеру выбрать наиболее подходящий способ визуализации:

- `email`. Проверяет предоставленные пользователем данные на соответствие формату адреса электронной почты;
- `search`. Предоставляет браузеру информацию о назначении поля для упрощения его визуализации на странице;
- `url`. Проверяет предоставленные пользователем данные на соответствие формату адреса веб-страницы;
- `tel`. Предоставляет браузеру информацию о назначении поля (здесь должен быть указан телефонный номер) для упрощения его визуализации на странице;
- `number`. Проверяет, что пользователь ввел числовое значение. Его можно комбинировать с другими атрибутами (такими, как `min`, `max` и `step`) для ограничения поддерживаемого диапазона чисел;
- `range`. Создает на странице ползунок, позволяющий выбрать числовое значение. Доступный диапазон ограничивается атрибутами `min`, `max` и `step`. Атрибут `value` устанавливает начальное значение элемента;
- `date`. Проверяет предоставленные пользователем данные на соответствие формату даты «год–месяц–день»;
- `month`. Проверяет предоставленные пользователем данные на соответствие формату даты «год–месяц»;

- `week`. Проверяет предоставленные пользователем данные на соответствие формату даты «год–неделя», где второе значение представляет собой букву `W`, за которой следует номер недели;
- `time`. Проверяет предоставленные пользователем данные на соответствие формату времени «часы:минуты:секунды». Также поддерживаются другие варианты синтаксиса, такие как «часы:минуты»;
- `datetime`. Проверяет предоставленные пользователем данные на соответствие полному формату даты, включая часовой пояс;
- `datetime-local`. Проверяет предоставленные пользователем данные на соответствие полному формату даты, но без часового пояса;
- `color`. Проверяет, что предоставленные пользователем данные представляют собой строку с закодированным значением одного цвета.

Атрибуты

Новые атрибуты были добавлены в HTML5 для расширения функциональности форм и управления валидацией:

- `autocomplete`. Указывает, должны ли вводимые значения сохраняться и в будущем предлагаться пользователю в виде списка возможных вариантов. Он может принимать два значения: `on` и `off`;
- `autofocus`. Булев атрибут, он переводит фокус ввода на элемент после того, как страница полностью загрузится;
- `novalidate`. Используется только с элементами `<form>`. Это булев атрибут, указывающий, должна ли выполняться валидация данных всей формы;
- `formnovalidate`. Предназначен для отдельных элементов формы. Это булев атрибут, указывающий, должна ли выполняться валидация конкретного элемента;
- `placeholder`. Позволяет создать подсказку, помогающую пользователю при заполнении элемента. Его значением может быть одно слово или короткий текст, который будет отображаться внутри поля ввода в случае, когда фокус ввода находится не на этом элементе;
- `required`. Объявляет элемент обязательным для валидации. Это булев атрибут, который не позволяет отправлять форму до тех пор, пока для этого атрибута не будет указано подходящее значение;

- `pattern`. Позволяет добавить регулярное выражение, на соответствие которому будут проверяться предоставленные пользователем данные;
- `multiple`. Булев атрибут, позволяющий ввести несколько значений в одном поле. Например, пользователь может указать несколько адресов электронной почты, разделив их запятыми;
- `form`. Связывает элемент с формой. Его значением должно быть значение атрибута `id` нужной формы;
- `list`. Связывает элемент с элементом `<datalist>`, в котором определен список возможных значений для данного поля ввода. Значением данного атрибута должно быть значение атрибута `id` нужного элемента `<datalist>`.

Элементы

В HTML5 также появились новые элементы, улучшающие и расширяющие функциональность форм:

- `<datalist>`. Позволяет создать список вариантов, которые будут предлагаться в элементе `<input>` в качестве возможных значений. Для конструирования списка используется элемент `<option>`, а каждый из вариантов объявляется с атрибутами `value` и `label`. Список вариантов связывается с элементом `<input>` через атрибут `list`;
- `<progress>`. Отражает степень завершения задачи, например загрузки данных;
- `<meter>`. Отражает некоторое измеряемое значение, например расход пропускной способности;
- `<output>`. Представляет выходные данные динамического приложения.

Методы

В спецификацию HTML5 входит специальный API для форм, предлагающий новые методы, события и свойства. Наиболее важные методы перечислены далее:

- `setCustomValidity(сообщение)`. Позволяет объявить ошибку и задать сообщение об ошибке для специального процесса валидации. Чтобы сбросить ошибку, необходимо вызвать данный метод, передав ему в качестве атрибута пустую строку;

- `checkValidity()`. Позволяет принудительно запустить валидацию из кода сценария. Таким образом, для активации встроенного в браузер процесса валидации не требуется нажимать кнопку отправки формы на сервер. Метод возвращает значение `true`, если элемент успешно проходит валидацию.

События

Далее перечислены события, входящие в API форм:

- `invalid`. Срабатывает, если во время процесса валидации обнаруживается элемент с недопустимым содержимым;
- `forminput`. Срабатывает, когда пользователь вводит какие-либо данные в форму;
- `formchange`. Срабатывает, когда в данных формы происходят какие-либо изменения.

Статусы

В API Forms (Формы) были добавлены различные проверки статусов, помогающие настраивать специальные процессы валидации:

- `valid`. Это общий статус валидности. Он возвращает значение `true` в том случае, когда ни один из остальных статусов не истинен (их значение не равно `true`), и это означает, что элемент успешно проходит валидацию;
- `valueMissing`. Равен `true`, когда для элемента определен атрибут `required`, но пользователь не заполнил его;
- `typeMismatch`. Равен `true`, когда введенное пользователем значение не соответствует ожидаемому, например, в полях для ввода адреса электронной почты или URL-адреса;
- `patternMismatch`. Равен `true`, когда введенное пользователем значение не соответствует регулярному выражению, определенному с помощью атрибута `pattern`;
- `tooLong`. Равен `true`, когда длина введенного значения превышает значение атрибута `maxlength`;
- `rangeUnderflow`. Равен `true`, когда введенное значение меньше значения атрибута `min`;

- `rangeOverflow`. Равен `true`, когда введенное значение больше значения атрибута `max`;
- `stepMismatch`. Равен `true`, когда значение атрибута `step` не соответствует значениям атрибутов `min`, `max` и `value`;
- `customError`. Равен `true`, когда в элементе возникает специальная ошибка, настраиваемая индивидуально.

7 API холста

Подготовка холста

API Canvas (Холст) — один из самых мощных в спецификации HTML5. Он позволяет разработчикам манипулировать динамической и интерактивной визуальной средой, реализовывая в веб-приложениях функциональность, которая раньше связывалась исключительно с настольными приложениями.

В начале книги мы обсуждали, как HTML5 заменяет существовавшие ранее встраиваемые модули, например Flash и Java-апплеты. Для того чтобы сделать Сеть независимой от сторонних технологий, необходимо позаботиться о двух очень важных вещах: обработке видео и графических приложениях. Элемент `<video>` и API мультимедиа хорошо справляются с первой задачей, но никак не могут помочь в решении проблемы с графикой. О графическом аспекте заботится API холста, и делает он это чрезвычайно эффектно и эффективно. API Canvas (Холст) позволяет рисовать графические элементы, выводить на экран изображения из файла, анимировать и другими способами обрабатывать рисунки и текст. Используя его совместно с другими составляющими HTML5, вы можете создавать полноценные приложения и даже двухмерные и трехмерные игры для Сети.

Элемент <canvas>

Этот элемент создает на веб-странице пустой прямоугольник, внутри которого затем визуализируются результаты применения методов, входящих в API холста. Добавляя в код страницы данный элемент, вы генерируете пустое пространство, аналогичное пустому элементу `<div>`, но имеющее совершенно иное предназначение.

Листинг 7.1. Синтаксис элемента <canvas>

```
<!DOCTYPE html>
<html lang="ru">
<head>
  <title>API холста</title>
  <script src="canvas.js"></script>
</head>
<body>
  <section id="canvasbox">
    <canvas id="canvas" width="500" height="300">
      В вашем браузере элемент canvas не поддерживается
    </canvas>
  </section>
</body>
</html>
```

Как видно из листинга 7.1, для этого элемента нужно указать буквально пару атрибутов. Атрибуты `width` и `height` определяют размер блока, их необходимо добавлять всегда, так как любые объекты, которые визуализируются поверх данного элемента, ориентируются на эти значения. Если также указать атрибут `id`, то вы сможете с легкостью сослаться на элемент `<canvas>` из кода JavaScript.

Итак, мы изучили всю функциональность элемента `<canvas>`: он всего лишь создает на экране пустой блок. Для того чтобы извлечь из этой поверхности практическую пользу, нужно добавить JavaScript-код и воспользоваться новыми методами и свойствами, появившимися в API.

ВНИМАНИЕ

В целях обеспечения совместимости на случай, если браузер не поддерживает API Canvas (Холст), мы добавили между тегами <canvas> альтернативное содержимое. Оно будет отображаться на экране в устаревших и несовместимых браузерах.

Метод getContext()

Для того чтобы подготовить элемент `<canvas>` к работе, в первую очередь необходимо вызвать метод `getContext()`. Этот метод генерирует контекст рисования, который будет связан с указанным холстом. Мы будем использовать ссылку на получившийся контекст рисования, работая с остальными возможностями API холста.

Листинг 7.2. Создание контекста рисования для холста

```
function initiate(){
  var elem=document.getElementById('canvas');
  canvas=elem.getContext('2d');
}
window.addEventListener("load", initiate, false);
```

В листинге 7.2 мы сохранили ссылку на элемент `<canvas>` в переменной `elem`, а затем создали контекст рисования, вызвав метод `getContext('2d')`. Этот метод принимает одно из двух значений, `2d` или `3d`, описывающих, соответственно, двухмерную и трехмерную среды рисования. Пока что поддерживается только вариант `2d`, однако разработка трехмерного API для холста уже идет полным ходом.

САМОСТОЯТЕЛЬНО

Скопируйте код HTML-документа из листинга 7.1 в новый пустой файл. Также понадобится создать файл с именем canvas.js — в нем вы будете сохранять примеры кода, начиная с листинга 7.2. Все фрагменты кода, представленные в этой главе, независимы друг от друга; каждый последующий нужно будет копировать на место предыдущего.

ПОВТОРЯЕМ ОСНОВЫ

Когда переменная объявляется внутри функции без ключевого слова var, она становится глобальной. Это означает, что обратиться к ней можно из любого места в коде, включая другие функции. В примере кода из листинга 7.2 мы объявили глобальную переменную canvas, для того чтобы всегда иметь возможность обращаться к контексту холста.

Контекст рисования холста представляет собой сетку пикселов, выстроенных в строки (слева направо) и столбцы (сверху вниз), с точкой отсчета (пикселом с координатами (0, 0)) в верхнем левом углу прямоугольника.

Рисование на холсте

Теперь, когда элемент `<canvas>` и его контекст рисования подготовлены, можно приступать к созданию настоящих рисунков и манипулированию графическими объектами. Для этих целей API предоставляет обширный арсенал инструментов и методов рисования, начиная с простых фигур и заканчивая текстом, тенями и сложными трансформациями. Мы изучим их последовательно, один за другим.

Рисование прямоугольников

Чаще всего разработчикам приходится сначала готовить фигуры, которые они хотели бы поместить на холст, а затем отправлять их в контекст (мы скоро рассмотрим соответствующие процессы), однако существует несколько методов, позволяющих рисовать прямо на холсте, минуя подготовительный этап. Эти методы ограничиваются прямоугольной формой и генерируют на экране примитивную фигуру (для получения других фигур придется научиться комбинировать различные техники рисования и сложные пути). Далее перечислены доступные простейшие методы:

- `fillRect(x, y, width, height)`. Предназначен для рисования прямоугольника, залитого цветом. Верхний левый угол фигуры будет находиться в точке, заданной атрибутами `x` и `y`. Атрибуты `width` и `height` определяют размер прямоугольника;
- `strokeRect(x, y, width, height)`. Аналогичен предыдущему, но создает на холсте пустой прямоугольник, то есть прямоугольный контур;
- `clearRect(x, y, width, height)`. Предназначен для вычитания пикселов из области, определяемой его атрибутами. Другими словами, это прямоугольный ластик.

Это уже знакомая вам функция из листинга 7.2, но с несколькими дополнительными методами, которые создают на холсте настоящие фигуры. Как видите, мы связали контекст с переменной `canvas`, и теперь эта

переменная используется для обращения к контексту в любом методе рисования.

Листинг 7.3. Рисование прямоугольников

```
function initiate(){
  var elem=document.getElementById('canvas');
  canvas=elem.getContext('2d');

  canvas.strokeRect(100,100,120,120);
  canvas.fillRect(110,110,100,100);
  canvas.clearRect(120,120,80,80);
}
window.addEventListener("load", initiate, false);
```

Первый метод, `strokeRect(100, 100, 120, 120)`, создает на холсте пустой прямоугольный контур с левым верхним углом в точке (100, 100) и стороной размером 120 пикселов (фактически это квадрат со стороной 120 пикселов). Второй метод, `fillRect(110, 110, 100, 100)`, создает прямоугольник, заполненный цветом, и на этот раз рисование начинается в точке холста (110, 110). Наконец, последний метод, `clearRect(120, 120, 80, 80)`, очищает квадратную область со стороной 80 пикселов в центре предыдущего прямоугольника.

На рис. 7.1 представлено всего лишь схематическое изображение того, что вы увидите после исполнения кода из листинга 7.3. Элемент `<canvas>` визуализируется как сетка с точкой отсчета в левом верхнем углу страницы и габаритами, определяемыми его атрибутами. Прямоугольники выводятся на холсте в точках, заданных атрибутами `x` и `y`, один поверх другого в том порядке, как они перечислены в коде (прямоугольник, описанный первым, будет нарисован первым, второй — поверх первого и т. д.). Существует метод, позволяющий настраивать способ рисования фигур, но о нем мы поговорим позже.

Цвета

Пока что мы рисовали только цветом по умолчанию — непрозрачным черным. Однако для определения любых желаемых цветов можно применять синтаксис CSS со следующими свойствами:

- `strokeStyle`. Определяет цвет линий фигуры;
- `fillStyle`. Определяет цвет внутренней области фигуры;
- `globalAlpha`. Описывает не цвет, а прозрачность. Устанавливает уровень прозрачности для всех фигур, которые визуализируются на холсте.

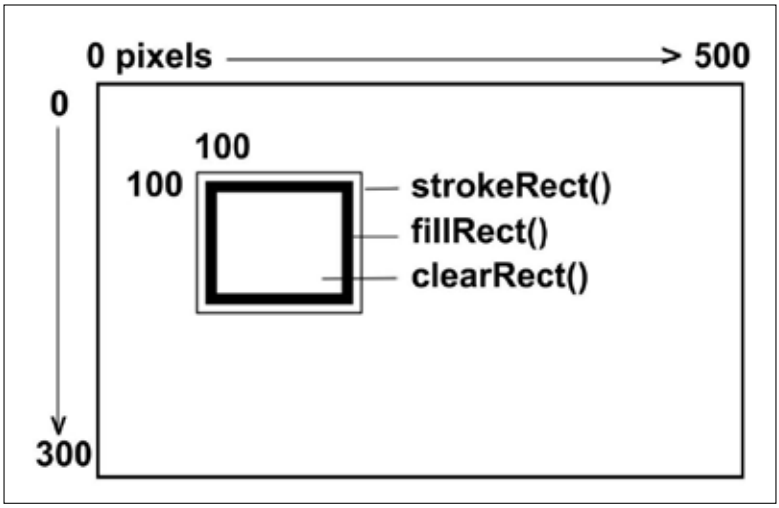

Рис. 7.1. Схема холста и прямоугольников, создаваемых кодом из листинга 7.3

Листинг 7.4. Добавляем цвет

```
function initiate(){
  var elem=document.getElementById('canvas');
  canvas=elem.getContext('2d');

  canvas.fillStyle="#000099";
  canvas.strokeStyle="#990000";

  canvas.strokeRect(100,100,120,120);
  canvas.fillRect(110,110,100,100);
  canvas.clearRect(120,120,80,80);
}
window.addEventListener("load", initiate, false);
```

В листинге 7.4 мы определили цвета как шестнадцатеричные значения. Но для этого также можно использовать методы, например `rgb()`, или даже определять степень прозрачности фигуры, как в методе `rgba()`. В таком случае необходимо добавлять кавычки: `strokeStyle="rgba(255, 165, 0, 1)"`.

Когда цвет определяется посредством перечисленных методов, он становится цветом по умолчанию для всех последующих методов рисования.

Помимо функции `rgba()` существует еще один способ настройки степени прозрачности: свойство `globalAlpha`. Его синтаксис таков: `globalAlpha=значение`, где значением может быть любое число от 0,0 (абсолютная непрозрачность) до 1,0 (абсолютная прозрачность).

Градиенты

Градиенты — неотъемлемая составляющая любой современной программы рисования, и холсты HTML5 не исключение. Так же как в CSS3, градиенты на холсте могут быть линейными или радиальными, кроме того, разработчик может определить несколько цветовых остановок, создав плавные переходы между множеством цветов:

- `createLinearGradient(x1, y1, x2, y2)`. Создает объект градиента для последующей визуализации на холсте;
- `createRadialGradient(x1, y1, r1, x2, y2, r2)`. Создает для последующей визуализации на холсте объект градиента, состоящий из двух окружностей. Значения в скобках представляют собой координаты центров окружностей и их радиусы;
- `addColorStop(position, color)`. Определяет цвета, которые будут использоваться для создания градиента. Атрибут `position` — это значение от 0,0 до 1,0, определяющее, в какой позиции начнется затухание цвета `color`.

Листинг 7.5. Наложение линейного градиента на холст

```
function initiate(){
  var elem=document.getElementById('canvas');
  canvas=elem.getContext('2d');

  var grad=canvas.createLinearGradient(0,0,10,100);
  grad.addColorStop(0.5, '#0000FF');
  grad.addColorStop(1, '#000000');
  canvas.fillStyle=grad;
```

```
  canvas.fillRect(10,10,100,100);
  canvas.fillRect(150,10,200,100);
}
window.addEventListener("load", initiate, false);
```

В листинге 7.5 мы создали градиент, распространяющийся от точки (0, 0) до точки (10, 100) с небольшим наклоном влево. Для определения цветов использовали метод `addColorStop()`, а для итогового применения градиента в качестве стиля заливки — свойство `fillStyle` (так же, как для любого другого цвета).

Обратите внимание на то, что позиция градиента определяется относительно холста, а не фигур, для заливки которых этот градиент затем используется. Таким образом, если в конце кода вы переместите прямоугольники на новое место, градиент внутри них станет выглядеть подругому.

САМОСТОЯТЕЛЬНО

Радиальный градиент для холста определяется аналогично тому, как мы это делали в правилах CSS3. Попробуйте заменить линейный градиент в коде из листинга 7.5 радиальным градиентом, используя выражение, подобное createRadialGradient(0, 0, 30, 0, 0, 300). Также вы можете поэкспериментировать с местоположением прямоугольников, чтобы посмотреть, каким образом градиент будет визуализироваться в каждой новой позиции.

Создание путей

Пока что мы изучали методы, позволяющие рисовать прямо на холсте, однако это не всегда возможно или уместно. Чаще всего приходится сначала в фоновом режиме обрабатывать фигуры и изображения, а после этого отправлять результат в контекст, для того чтобы рисунок в итоге появился на странице. Для этого в API Canvas (Холст) предусмотрено несколько методов генерации путей.

Путь — это контур, вдоль которого следует кончик воображаемого пера, оставляя за собой след. Сначала нам нужно определить путь, затем отправить его в контекст, и только после этого он будет постоянно визуализирован на экране. Путь может включать в себя различные виды штрихов:

прямые линии, дуги, прямоугольники и т. п. Из всех этих разнообразных компонентов можно составлять весьма сложные фигуры.

Следующие два метода предназначены для создания путей и их закрытия:

- `beginPath()`. Объявляет о начале определения новой фигуры. Вызывается в первую очередь, еще до того, как мы начнем создавать новый путь;
- `closePath()`. Закрывает путь, добавляя прямую линию между последней точкой и исходной точкой пути. Существуют способы избежать появления этой линии в ситуациях, когда требуется открытый путь или когда для рисования пути используется метод `fill()`.

Следующие три метода позволяют визуализировать путь на холсте:

- `stroke()`. Визуализирует путь в виде контура;
- `fill()`. Визуализирует путь в виде залитой цветом фигуры. Если вы используете данный метод, то закрывать путь посредством `closePath()` не требуется — он автоматически закрывается прямой линией, соединяющей последнюю точку с первой;
- `clip()`. Определяет область обрезки для контекста. В момент инициализации контекста область обрезки совпадает с областью холста. Метод `clip()` позволяет задать область обрезки произвольной формы, создав маску. Все, что оказывается за пределами маски, на странице не визуализируется.

Листинг 7.6. Базовые правила для определения пути

```
function initiate(){
  var elem=document.getElementById('canvas');
  canvas=elem.getContext('2d');

  canvas.beginPath();
  // здесь описывается путь
  canvas.stroke();
}
window.addEventListener("load", initiate, false);
```

Код из листинга 7.6 не создает никаких рисунков, он лишь сигнализирует о начале описания пути в контексте конкретного холста и визуализирует контур этого пути на экране с помощью метода `stroke()`. Для описания пути и создания реальной фигуры предназначены следующие методы:

- `moveTo(x, y)`. Перемещает кончик пера в указанную позицию. Позволяет начать или продолжить рисование пути с новой точки сетки, создав таким образом разрыв в контуре;
- `lineTo(x, y)`. Создает прямой отрезок линии между текущей позицией пера и точкой с координатами `x` и `y`;
- `rect(x, y, width, height)`. Создает прямоугольник. В отличие от методов, описанных ранее, в данном случае прямоугольник становится частью пути (а не сразу же визуализируется на холсте). Тем не менее атрибуты остаются неизменными;
- `arc(x, y, radius, startAngle, endAngle, direction)`. Создает дугу или окружность с центром в точке с координатами `x` и `y`, радиусом и угловым значением, объявленными в атрибутах. Последний атрибут — это булево значение, задающее направление рисования: по часовой стрелке или против нее;
- `quadraticCurveTo(cpx, cpy, x, y)`. Создает квадратичную кривую Безье, начинающуюся в текущей позиции пера и заканчивающуюся в позиции с координатами `x` и `y`. Атрибуты `cpx` и `cpy` представляют собой контрольные точки, управляющие формой кривой;
- `bezierCurveTo(cp1x, cp1y, cp2x, cp2y, x, y)`. Аналогичен предыдущему, но задействует два дополнительных атрибута, позволяющих определить кубическую кривую Безье. В данном случае для управления формой кривой используются две контрольные точки на сетке, описываемые значениями `cp1x`, `cp1y`, `cp2x` и `cp2y`.

Давайте попробуем создать простой путь, для того чтобы понять, как работают эти методы.

Листинг 7.7. Наш первый путь

```
function initiate(){
  var elem=document.getElementById('canvas');
  canvas=elem.getContext('2d');

  canvas.beginPath();
  canvas.moveTo(100,100);
  canvas.lineTo(200,200);
  canvas.lineTo(100,200);
  canvas.stroke();
}
window.addEventListener("load", initiate, false);
```

Рекомендуем всегда задавать начальную позицию пера сразу же после объявления начала пути. В коде из листинга 7.7 мы сначала перенесли перо в позицию (100, 100), а затем создали линию между этой точкой и точкой (200, 200). После этого текущей позицией пера уже считается точка (200, 200), а следующий отрезок создается между ней и точкой (100, 200). Наконец, мы визуализируем путь в виде контура, применяя для этого метод `stroke()`.

Протестировав этот код в своем браузере, вы увидите на странице острый угол (или треугольник без одной стороны). Этот треугольник можно закрыть и даже залить цветом или узором, применив различные методы, как в следующих примерах (листинги 7.8 и 7.9).

Листинг 7.8. Завершение создания треугольника

```
function initiate(){
  var elem=document.getElementById('canvas');
  canvas=elem.getContext('2d');

  canvas.beginPath();
  canvas.moveTo(100,100);
  canvas.lineTo(200,200);
  canvas.lineTo(100,200);
  canvas.closePath();
  canvas.stroke();
}
window.addEventListener("load", initiate, false);
```

Метод `closePath()` всего лишь добавляет к пути прямой отрезок, соединяющий последнюю точку с первой и таким образом закрывающий фигуру.

Применив метод `stroke()` в конце пути, мы нарисовали на холсте пустой треугольник. Чтобы получить треугольник, залитый цветом, необходимо использовать метод `fill()`.

Листинг 7.9. Заливаем треугольник сплошным цветом

```
function initiate(){
  var elem=document.getElementById('canvas');
  canvas=elem.getContext('2d');

  canvas.beginPath();
  canvas.moveTo(100,100);
```

```
  canvas.lineTo(200,200);
  canvas.lineTo(100,200);
  canvas.fill();
}
window.addEventListener("load", initiate, false);
```

Теперь на экране отображается сплошной черный треугольник. Метод `fill()` автоматически закрывает путь, поэтому добавлять в код метод `closePath()` не требуется.

Ранее мы упоминали еще один метод визуализации контура на холсте, а именно `clip()`. Этот метод ничего не рисует, он всего лишь создает маску в форме пути и таким образом позволяет определить, что будет нарисовано на холсте, а что нет. Ничего из того, что оказывается за пределами маски, на экране не отображается.

Листинг 7.10. Используем треугольник в качестве маски

```
function initiate(){
  var elem=document.getElementById('canvas');
  canvas=elem.getContext('2d');

  canvas.beginPath();
  canvas.moveTo(100,100);
  canvas.lineTo(200,200);
  canvas.lineTo(100,200);
  canvas.clip();

  canvas.beginPath();
  for(f=0; f<300; f=f+10){
    canvas.moveTo(0,f);
    canvas.lineTo(500,f);
  }
  canvas.stroke();
}
window.addEventListener("load", initiate, false);
```

Для того чтобы продемонстрировать работу метода `clip()`, мы добавили в листинг 7.10 цикл, создающий горизонтальные линии через каждые 10 пикселов. Линии пересекают холст слева направо, но на странице мы видим только те их фрагменты, которые попадают внутрь треугольной маски.

Теперь, когда мы научились рисовать пути, настало время изучить альтернативные варианты. Пока что мы создавали только прямые линии и прямоугольные фигуры. Для создания фигур, включающих в себя различные дуги, в API предусмотрено три новых метода: `arc()`, `quadraticCurveTo()` и `bezierCurveTo()`. Первый относительно прост и предназначен для рисования полных окружностей или дуг, что иллюстрирует листинг 7.11.

Листинг 7.11. Создание окружности с помощью метода arc()

```
function initiate(){
  var elem=document.getElementById('canvas');
  canvas=elem.getContext('2d');

  canvas.beginPath();
  canvas.arc(100,100,50,0,Math.PI*2, false);
  canvas.stroke();
}
window.addEventListener("load", initiate, false);
```

Вы наверняка обратили внимание на значение `PI` в определении метода. Данный метод ориентируется на значения угла в радианах, а не в градусах. Значение `PI` в радианах соответствует 180°, таким образом, формула `PI` × 2 означает удвоение `PI`, что в итоге дает 360°.

Код из листинга 7.11 создает дугу с центром в точке (100, 100) радиусом 50 пикселов, начинающуюся от угла 0° и заканчивающуюся углом `Math.PI` × 2 градусов, что соответствует полной окружности. Свойство `PI` объекта `Math` дает точное значение числа π.

Для перевода значения в градусах в радианы используйте формулу `Math.PI`/180 × градусы, как в следующем примере.

Листинг 7.12. Дуга размером 45°

```
function initiate(){
  var elem=document.getElementById('canvas');
  canvas=elem.getContext('2d');

  canvas.beginPath();
  var radians=Math.PI/180*45;
  canvas.arc(100,100,50,0,radians, false);
  canvas.stroke();
}
window.addEventListener("load", initiate, false);
```

Выполнив код из листинга 7.12, мы увидим на экране дугу, составляющую только 45° окружности. Попробуйте изменить значение атрибута `direction` в этом примере на `true`. В результате на экране появится дуга между точками 0° и 315°, выглядящая как окружность без небольшого фрагмента.

Очень важно учитывать следующее: если вы продолжите работать с этим путем после создания дуги, то начальной точной будет считаться конец дуги. Если такое поведение нежелательно, нужно при помощи метода `moveTo()` изменить позицию пера. Кстати, если следующей кривой также должна быть дуга, например, вы хотите нарисовать полный круг, то не стоит забывать, что метод `moveTo()` переносит виртуальное перо в точку, с которой начинается рисование, а не в центр круга. Предположим, что центр вашего круга должен находиться в точке (300, 150), а его радиус равен 50 пикселам. Тогда для получения нужной фигуры следует перевести перо в точку (350, 150).

Помимо `arc()` есть еще два метода рисования сложных кривых. Метод `quadraticCurveTo()` генерирует квадратичную кривую Безье, а метод `bezierCurveTo()` предназначен для рисования кубической кривой Безье. Различие между этими методами заключается в том, что первый предлагает только одну контрольную точку, а второй — две, таким образом, они создают кривые разных типов.

Листинг 7.13. Сложные кривые

```
function initiate(){
  var elem=document.getElementById('canvas');
  canvas=elem.getContext('2d');

  canvas.beginPath();
  canvas.moveTo(50,50);
  canvas.quadraticCurveTo(100, 125, 50, 200);
  canvas.moveTo(250,50);
  canvas.bezierCurveTo(200, 125, 300, 125, 250, 200);
  canvas.stroke();
}
window.addEventListener("load", initiate, false);
```

Для рисования квадратичной кривой мы переместили виртуальное перо в позицию (50, 50), а координаты конечной точки определили как (50, 200). Координаты контрольной точки этой кривой (100, 125).

Кубическая кривая, создаваемая методом `bezierCurveTo()`, немного сложнее. У нее две контрольные точки, одна с координатами (200, 125), вторая — (300, 125) (рис. 7.2).

Значения на рис. 7.2 указывают позиции контрольных точек наших кривых. Перемещая контрольные точки, вы можете менять форму кривых.

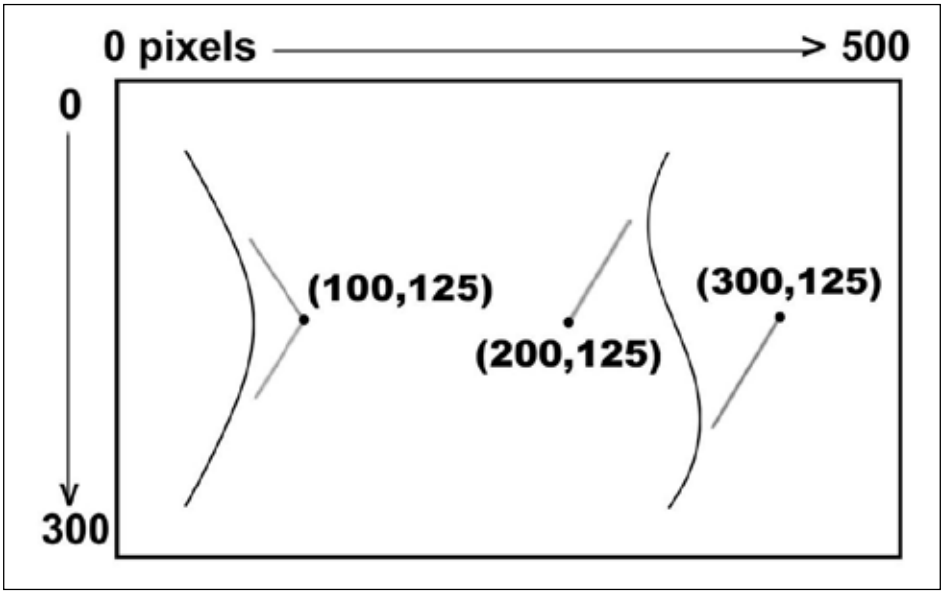

Рис. 7.2. Вид кривых Безье на холсте с указанием их контрольных точек

САМОСТОЯТЕЛЬНО

Вы можете добавить сколь угодно много кривых для построения желаемых фигур. Попробуйте поменять значения контрольных точек в коде из листинга 7.13 и проверьте в браузере, как изменится форма кривых. Стройте сложные фигуры, объединяя разные кривые и линии, для того чтобы понять, каким образом конструируют пути.

Стили линий

Пока что мы использовали один и тот же стиль для всех линий на холсте. Ширину, вид окончания и другие характеристики линии можно

настраивать, создавая линии в точности такого типа, какой требуется для завершения рисунка.

Для этой цели применяются четыре свойства:

- `lineWidth`. Определяет толщину линии. Значение по умолчанию единица;
- `lineCap`. Определяет форму окончания линии. Оно может принимать одно из трех значений: `butt`, `round` или `square`;
- `lineJoin`. Определяет форму соединения двух линий. Возможные значения: `round`, `bevel` и `miter`;
- `miterLimit`. Используется совместно со свойством `lineJoin` и определяет протяженность соединения двух линий в случае, если свойству `lineJoin` присвоено значение `miter`.

Перечисленные свойства влияют на весь путь. После каждого изменения характеристик линии необходимо создавать новый путь с новыми значениями свойств.

Листинг 7.14. Тестируем свойства линий

```
function initiate(){
  var elem=document.getElementById('canvas');
  canvas=elem.getContext('2d');
  canvas.beginPath();
  canvas.arc(200,150,50,0,Math.PI*2, false);
  canvas.stroke();

  canvas.lineWidth=10;
  canvas.lineCap="round";
  canvas.beginPath();
  canvas.moveTo(230,150);
  canvas.arc(200,150,30,0,Math.PI, false);
  canvas.stroke();
  canvas.lineWidth=5;
  canvas.lineJoin="miter";
  canvas.beginPath();
  canvas.moveTo(195,135);
  canvas.lineTo(215,155);
  canvas.lineTo(195,155);
  canvas.stroke();
}
window.addEventListener("load", initiate, false);
```

В коде 7.14 мы начали рисование с создания пути со свойствами по умолчанию, определяющего полный круг. Затем, используя `lineWith`, установили новую ширину линии, равную 10 единицам, а свойству `lineCap` присвоили значение `round`. Таким образом, следующий путь стал толще и со скругленными окончаниями. Для визуализации этого пути мы переместили перо в точку (230, 150) и создали полукруг. Скругленные окончания помогают создать иллюзию улыбающегося рта.

Наконец, мы добавили на холст еще один путь, включающий в себя две линии и напоминающий по форме нос. Обратите внимание на то, что ширина линий этого пути равна 5 единицам, а для определения формы их соединения мы присвоили свойству `lineJoin` значение `miter`. Благодаря этому нос получается заостренным: внешние края угла в точке соединения вытягиваются и сходятся в одну точку.

САМОСТОЯТЕЛЬНО

Поэкспериментируйте со свойствами линий носа. Например, определите значение свойства miterLimit, добавив инструкцию miterLimit=2. Поменяйте значение свойства lineJoin на round или bevel. Можете также попробовать изменить форму рта, использовав другие значения свойства lineCap.

Текст

Для того чтобы добавить на холст какой-то текст, нужно всего лишь определить несколько свойств и вызвать подходящий метод. Для настройки текста используются три свойства:

- `font`. Синтаксис этого свойства аналогичен синтаксису свойства `font` из CSS, и оно принимает такие же значения;
- `textAlign`. У этого свойства несколько допустимых значений. Возможные варианты выравнивания описываются значениями `start`, `end`, `left`, `right` и `center`;
- `textBaseline`. Предназначено для настройки выравнивания по вертикали. Устанавливает позицию текста (включая текст в кодировке Unicode). Возможные значения: `top`, `hanging`, `middle`, `alphabetic`, `ideographic` и `bottom`.

Для вывода текста на холст используются два метода:

- `strokeText(text, x, y)`. Аналогично методам для рисования пути, выводит на холст в точке (`x`, `y`) переданный ему в первом параметре текст, превращая его в контурную фигуру. В список параметров можно добавить и четвертое значение, определяющее максимальный размер. Если текст длиннее этого значения, то он сжимается, чтобы полностью поместиться в пространстве указанного размера;
- `fillText(text, x, y)`. Аналогичен предыдущему, но визуализирует текст как залитые цветом фигуры.

Листинг 7.15. Рисование текста

```
function initiate(){
  var elem=document.getElementById('canvas');
  canvas=elem.getContext('2d');

  canvas.font="bold 24px verdana, sans-serif";
  canvas.textAlign="start";
  canvas.fillText("my message", 100,100);
}
window.addEventListener("load", initiate, false);
```

Как видно из листинга 7.15, свойству `font` можно передавать одновременно несколько значений, а его синтаксис аналогичен синтаксису свойства `font` из CSS. Свойство `textAling` определено таким образом, чтобы очередная порция текста выводилась на холст, начиная с точки (100, 100) (если бы, например, мы присвоили данному свойству значение `end`, то текст заканчивался бы в позиции (100, 100)). Наконец, метод `fillText` визуализирует текст на холсте, заливая его сплошным цветом.

Листинг 7.16. Измерение текста

```
function initiate(){
  var elem=document.getElementById('canvas');
  canvas=elem.getContext('2d');

  canvas.font="bold 24px verdana, sans-serif";
  canvas.textAlign="start";
  canvas.textBaseline="bottom";
```

продолжение ⇩

Листинг 7.16 *(продолжение)*

```
    canvas.fillText("My message", 100,124);

    var size=canvas.measureText("My message");
    canvas.strokeRect(100,100,size.width,24);
}
window.addEventListener("load", initiate, false);
```

Помимо методов, перечисленных ранее, этот API предоставляет еще один важный метод, предназначенный для работы с текстом, — `measureText()`. Он возвращает информацию о размере указанного текста. Его полезно применять в случаях, когда текст комбинируется с другими фигурами на холсте, а также для вычисления точных позиций элементов и даже для создания столкновений в анимированных рисунках.

Здесь мы начинаем с того же кода, который использовали в листинге 7.15, но добавляем выравнивание по вертикали. Свойству `textBaseline` присваивается значение `bottom`, и это означает, что координата нижнего края текста равна 124 пикселам. Таким образом, мы точно знаем, на каком уровне по вертикали текст находится на холсте.

Благодаря методу `measureText()` и свойству `width` мы узнаем размер текста по горизонтали. Выполнив все необходимые измерения, рисуем аккуратный прямоугольник точно вокруг текста.

САМОСТОЯТЕЛЬНО

Используя код из листинга 7.16, протестируйте разные значения свойств textAlign и textBaseline. Ориентируйтесь на прямоугольник: его размер будет автоматически подгоняться к размеру текста.

Тени

Разумеется, тени — очень важная составляющая API Canvas (Холст). Тени можно создавать для любых путей и даже для текста. В API для этого предусмотрены четыре свойства:

- `shadowColor`. Объявляет цвет тени. Синтаксис такой же, как в CSS;
- `shadowOffsetX`. Ему необходимо передать числовое значение, указывающее, на какое расстояние тень будет отстоять от объекта по горизонтали;

- `shadowOffsetY`. Ему необходимо передать числовое значение, указывающее, на какое расстояние тень будет отстоять от объекта по вертикали;
- `shadowBlur`. Создает для тени эффект размытости.

Листинг 7.17. Добавляем тени

```
function initiate(){
  var elem=document.getElementById('canvas');
  canvas=elem.getContext('2d');

  canvas.shadowColor="rgba(0,0,0,0.5)";
  canvas.shadowOffsetX=4;
  canvas.shadowOffsetY=4;
  canvas.shadowBlur=5;

  canvas.font="bold 50px verdana, sans-serif";
  canvas.fillText("my message", 100,100);
}
window.addEventListener("load", initiate, false);
```

Для определения цвета тени в листинге 7.17 мы используем функцию `rgba()`. Наша тень полупрозрачная черного цвета. Расстояние между тенью и объектом составляет 4 пиксела, кроме того, тень размывается на 5 пикселов.

> **САМОСТОЯТЕЛЬНО**
>
> Добавьте тени к другим, нетекстовым объектам. Например, попробуйте создать тень для контурной фигуры, затем для фигуры, залитой цветом. Протестируйте ее на прямоугольниках и кругах.

Трансформации

API холста поддерживает сложные операции с графическими объектами и с самим холстом. Для выполнения операций используются пять методов трансформации, каждый из которых выполняет определенную задачу:

- `translate(x, y)`. Применяется для переноса начала координат холста. Начало координат, то есть точка с координатами (0, 0), по умолчанию

находится в левом верхнем углу холста, в обоих направлениях координаты `x` и `y` на холсте увеличиваются. Точки с отрицательными координатами находятся за пределами холста. Однако иногда для создания сложных фигур удобно использовать отрицательные значения. Метод `translate()` позволяет переместить начало координат в другую точку холста и использовать новое местоположение в качестве точки отсчета для создания рисунков;

- `rotate(angle)`. Поворачивает холст вокруг начала координат на указанный угол;
- `scale(x, y)`. Увеличивает или уменьшает единицы измерения, использующиеся на холсте, увеличивая или уменьшая масштаб всех нарисованных на нем элементов. Масштаб можно менять независимо по горизонтали и по вертикали, указывая соответствующие значения `x` и `y`. Также значения параметров могут быть отрицательными — это позволяет создавать зеркальные отражения. Значения по умолчанию равны 1,0;
- `transform(m1, m2, m3, m4, dx, dy)`. Холст характеризуется матрицей значений, описывающих его свойства. Метод `transform()` применяет новую матрицу трансформаций поверх текущей, модифицируя таким образом весь холст;
- `setTransform(m1, m2, m3, m4, dx, dy)`. Отменяет текущую трансформацию и определяет новую на основе значений, переданных в атрибутах метода.

Листинг 7.18. Трансляция, поворот и масштабирование

```
function initiate(){
  var elem=document.getElementById('canvas');
  canvas=elem.getContext('2d');

  canvas.font="bold 20px verdana, sans-serif";
  canvas.fillText("TEST",50,20);

  canvas.translate(50,70);
  canvas.rotate(Math.PI/180*45);
  canvas.fillText("TEST",0,0);

  canvas.rotate(-Math.PI/180*45);
  canvas.translate(0,100);
  canvas.scale(2,2);
```

```
  canvas.fillText("TEST",0,0);
}
window.addEventListener("load", initiate, false);
```

Нет лучшего способа понять, как работают трансформации, чем использование их в программном коде. В листинге 7.18 мы применили методы `translate()`, `rotate()` и `scale()` к одному и тому же тексту. Сначала нарисовали этот текст на холсте в состоянии по умолчанию. Текст выводится в точке (50, 20), а его размер равен 20 пикселам. После этого мы с помощью `translate()` перенесли начало координат холста в точку (50, 70) и повернули холст на 45°, использовав метод `rotate()`. Таким образом, во второй раз текст выводится с учетом нового начала координат и наклона 45°. После этого определенные на предыдущем шаге трансформации считаются значениями по умолчанию, поэтому для тестирования метода `scale()` возвращаем исходное значение поворота, а начало координат переносим еще на 100 пикселов ниже. Наконец, мы удваиваем масштаб холста, поэтому при визуализации очередной строки текст оказывается в два раза крупнее оригинала.

Каждая последующая трансформация накладывается на предыдущие, то есть они не изолированы друг от друга. Например, если выполнить две трансформации `scale()` одну за другой, то второй метод масштабирует холст с учетом изменений, произошедших после выполнения первого. Применение `scale(2, 2)` после еще одного такого же `scale(2, 2)` увеличивает масштаб холста в четыре раза. Матричные методы трансформации также не исключение. Именно поэтому для определения трансформаций на матрице характеристик также существует два метода: `transform()` и `setTransform()`.

Листинг 7.19. Кумулятивная трансформация на матрице

```
function initiate(){
  var elem=document.getElementById('canvas');
  canvas=elem.getContext('2d');

  canvas.transform(3,0,0,1,0,0);

  canvas.font="bold 20px verdana, sans-serif";
  canvas.fillText("TEST",20,20);

  canvas.transform(1,0,0,10,0,0);
```

продолжение ⇗

Листинг 7.19 *(продолжение)*

```
  canvas.font="bold 20px verdana, sans-serif";
  canvas.fillText("TEST",100,20);
}
window.addEventListener("load", initiate, false);
```

Так же, как и в предыдущем примере кода, в листинге 7.19 мы применили методы трансформации к одному и тому же тексту, для того чтобы сравнить результаты. Значения матрицы холста по умолчанию равны (1, 0, 0, 1, 0, 0). Поменяв первое значение на 3 в первой трансформации, мы растянули холст по горизонтали. Сразу после этого текст, нарисованный на холсте, оказывается шире, чем при значениях матрицы по умолчанию.

Благодаря следующей трансформации холст растягивается по вертикали: мы меняем четвертое значение на 10, сохраняя значения по умолчанию для остальных параметров.

Важно помнить, что все последующие трансформации применяются к матрице холста, уже модифицированной предыдущими трансформациями. Поэтому второй текст, который выводится кодом из листинга 7.19, растягивается как по вертикали, так и по горизонтали. Для того чтобы отменить изменения матрицы и определить совершенно новые значения трансформации, нужно применить метод `setTransform()`.

САМОСТОЯТЕЛЬНО

Попробуйте заменить последний метод transform() в этом примере методом setTransform() и проверьте результат в браузере. Все так же работая только с текстом, поменяйте все значения в методе transform() и посмотрите, какую трансформацию определяет каждое из них.

Восстановление состояния

Из-за накопительного эффекта трансформаций возвращаться к предыдущим состояниям холста становится довольно сложно. Например, для написания кода в листинге 7.18 приходилось запоминать, на какой угол мы повернули холст, для того чтобы в следующем методе вернуть его в исходное положение. Решить эту проблему помогают предоставляемые API Canvas (Холст) методы сохранения и восстановления состояния холста:

- `save()`. Сохраняет состояние холста, включая все определенные для него ранее трансформации, значения свойств стилизации и текущий контур обрезки (область, определяемую методом `clip()`, если он использован);
- `restore()`. Восстанавливает последнее сохраненное состояние.

Листинг 7.20. Сохранение состояния холста

```
function initiate(){
  var elem=document.getElementById('canvas');
  canvas=elem.getContext('2d');

  canvas.save();
  canvas.translate(50,70);
  canvas.font="bold 20px verdana, sans-serif";
  canvas.fillText("TEST1",0,30);
  canvas.restore();

  canvas.fillText("TEST2",0,30);
}
window.addEventListener("load", initiate, false);
```

Выполнив код из листинга 7.20 в своем браузере, вы увидите текст «TEST1», написанный крупными буквами в центре холста, и текст «TEST2» более мелким шрифтом рядом с началом координат. Мы сохранили состояние холста по умолчанию, а затем определили новое начало координат и стили оформления текста. Прежде чем выводить на холст вторую порцию текста, мы восстановили сохраненное состояние. Таким образом, второй текст визуализируется с учетом стилей по умолчанию, а не тех, которые мы объявили для первой строки.

Независимо от того, как много трансформаций вы применяете к холсту, при вызове метода `restore()` в точности восстанавливается предыдущее сохраненное состояние холста.

Свойство globalCompositeOperation

Обсуждая пути, мы сказали, что существует свойство, определяющее, каким образом фигура выводится на холст и комбинируется с фигурами, созданными ранее. Это свойство называется `globalCompositeOperation`, а его значение по умолчанию — `source-over`, это означает, что новая

фигура визуализируется поверх уже добавленных на холст. Можно использовать 11 других значений данного свойства:

- `source-in`. Визуализируется только та часть новой фигуры, которая перекрывает предыдущую фигуру. Остальные фрагменты новой и предыдущей фигур становятся прозрачными;
- `source-out`. Визуализируется только та часть новой фигуры, которая не перекрывает предыдущую фигуру. Остальные фрагменты новой и даже предыдущей фигуры становятся прозрачными;
- `source-atop`. Визуализируется только та часть новой фигуры, которая перекрывает предыдущую фигуру. Предыдущая фигура сохраняется целиком, но остальные фрагменты новой фигуры становятся прозрачными;
- `lighter`. Визуализируются обе фигуры, но цвет перекрывающихся путей определяется сложением цветовых значений;
- `xor`. Визуализируются обе фигуры, но перекрывающиеся фрагменты становятся прозрачными;
- `destination-over`. Это прямая противоположность значению по умолчанию. Новые фигуры визуализируются позади фигур, уже добавленных на холст;
- `destination-in`. Сохраняются только те фрагменты существующих фигур, которые перекрываются новой фигурой. Все остальное, включая новую фигуру, становится прозрачным;
- `destination-out`. Сохраняются только те фрагменты существующих фигур, которые не перекрываются новой фигурой. Все остальное, включая новую фигуру, становится прозрачным;
- `destination-atop`. Существующие фигуры и новая фигура становятся прозрачными, за исключением тех фрагментов, где они перекрываются;
- `darker`. Визуализируются обе фигуры, но цвет перекрывающихся фрагментов определяется вычитанием значений цвета;
- `copy`. Визуализируется только новая фигура, а предыдущие становятся прозрачными.

Листинг 7.21. Проверка свойства globalCompositeOperation

```
function initiate(){
  var elem=document.getElementById('canvas');
  canvas=elem.getContext('2d');

  canvas.fillStyle="#990000";
```

```
  canvas.fillRect(100,100,300,100);

  canvas.globalCompositeOperation="destination-atop";

  canvas.fillStyle="#AAAAFF";
  canvas.font="bold 80px verdana, sans-serif";
  canvas.textAlign="center";
  canvas.textBaseline="middle";
  canvas.fillText("TEST",250,110);
}
window.addEventListener("load", initiate, false);
```

Только визуальная иллюстрация всех возможных значений свойства `globalCompositeOperation` поможет вам понять, как они работают. Для этого мы подготовили код в листинге 7.21. При его выполнении в центре холста создается красный прямоугольник, но благодаря значению `destination-atop` мы видим только ту часть прямоугольника, поверх которой выводится текст.

САМОСТОЯТЕЛЬНО

Замените значение destination-atop любым другим значением данного свойства и проверьте результат в своем браузере. Протестируйте код в разных браузерах.

Обработка изображений

API Canvas (Холст) не представлял бы никакой ценности, если бы не предлагал возможности обработки изображений. Но несмотря на значимость изображений для веб-дизайна, для работы с графическими файлами предусмотрен только один встроенный метод.

Метод drawImage()

Метод `drawImage()` — единственный предназначенный для вывода изображений на холст. Однако он способен принимать различные значения, возвращая разные результаты в зависимости от переданных аргументов. Далее перечислены всевозможные варианты использования:

- `drawImage(image, x, y)`. Такой синтаксис позволяет вывести изображение на холст в позиции, определяемой параметрами `x` и `y`. Первый параметр должен содержать ссылку на изображение;
- `drawImage(image, x, y, width, height)`. Этот синтаксис позволяет масштабировать изображение перед тем, как помещать его на холст. Новые значения ширины и высоты задаются в параметрах `width` и `height`;
- `drawImage(image, x1, y1, width1, height1, x2, y2, width2, height2)`. Это самый сложный вариант синтаксиса. Для каждого параметра передаются по два значения. Смысл в том, что вы можете отрезать часть изображения и вывести его в указанной точке холста, одновременно поменяв его размер. Значения `x1` и `y1` определяют координаты верхнего левого угла отрезаемого фрагмента изображения. Значения `width1` и `height1` задают размер этого фрагмента изображения. Остальные значения (`x2`, `y2`, `width2` и `height2`) объявляют точку, в которой будет выводиться изображение, а также его размер (который может отличаться от исходного).

В любом случае первый атрибут — это ссылка на изображение в том же документе, сгенерированная с помощью такого метода, как `getElementById()`, или созданием нового объекта изображения обычными методами JavaScript. Использовать URL-адреса или загружать файлы из внешних источников напрямую в данном методе запрещается.

Листинг 7.22. Работа с изображениями

```
function initiate(){
  var elem=document.getElementById('canvas');
  canvas=elem.getContext('2d');

  var img=new Image();
  img.src="http://www.minkbooks.com/content/snow.jpg";
  img.addEventListener("load", function(){
    canvas.drawImage(img,20,20)
  }, false);
}
window.addEventListener("load", initiate, false);
```

Начнем с простого примера. В коде из листинга 7.22 мы всего лишь загружаем изображение и выводим его на холст. Поскольку холст способен принимать только заранее загруженные изображения, эту ситуацию

необходимо контролировать, отслеживая событие `load`. Мы добавили прослушиватель и объявили анонимную функцию для обработки данного события. Метод `drawImage()` внутри функции выводит изображение на холст только после того, как его загрузка завершается.

ПОВТОРЯЕМ ОСНОВЫ

В листинге 7.22 в методе addEventListener() мы использовали анонимную функцию, а не ссылку на функцию. В подобных случаях, когда функция совсем небольшая, такая техника позволяет упростить код и сделать его более легким для понимания. Чтобы побольше узнать об этом, откройте наш веб-сайт и изучите ссылки для данной главы.

Листинг 7.23. Корректировка размера изображения по размеру холста

```
function initiate(){
  var elem=document.getElementById('canvas');
  canvas=elem.getContext('2d');

  var img=new Image();
  img.src="http://www.minkbooks.com/content/snow.jpg";
  img.addEventListener("load", function(){
    canvas.drawImage(img,0,0,elem.width,elem.height)
  }, false);
}
window.addEventListener("load", initiate, false);
```

В листинге 7.23 мы добавили к уже знакомому нам методу `drawImage()` два новых значения, меняющих размер изображения. Свойства `width` и `height` возвращают размеры холста, поэтому наш код растянет изображение так, чтобы оно закрыло холст полностью.

Листинг 7.24. Извлечение, изменение размера и визуализация

```
function initiate(){
  var elem=document.getElementById('canvas');
  canvas=elem.getContext('2d');

  var img=new Image();
  img.src="http://www.minkbooks.com/content/snow.jpg";
```

продолжение ⇩

Листинг 7.24 *(продолжение)*

```
    img.addEventListener("load", function(){
      canvas.drawImage(img,135,30,50,50,0,0,200,200)
    }, false);
}
window.addEventListener("load", initiate, false);
```

В листинге 7.24 представлен самый сложный синтаксис метода `drawImage()`. Для того чтобы отрезать часть исходного изображения, поменять его размер, а затем вывести на холст, методу переданы девять аргументов. Мы вырезали из исходного изображения квадратный фрагмент, начинающийся с позиции (135, 50). Размеры квадрата 50×50 пикселов. После этого вырезанный блок увеличен до размера 200×200 пикселов и, наконец, визуализирован на холсте в точке (0, 0).

Данные изображений

Сказав, что `drawImage()` — единственный метод, способный выводить изображения на холст, мы солгали. Существует еще несколько мощных методов, предназначенных для обработки изображений, которые также умеют визуализировать результат. Но поскольку они работают не с изображениями, а с данными, предыдущее заявление следует считать истинным. И все же зачем нам обрабатывать данные вместо изображений?

Каждое изображение можно представить в виде последовательности целых чисел, соответствующих компонентам RGBA (по четыре значения на каждый пиксел). Группа значений, несущих такую информацию, составляет одномерный массив, на основе которого можно сгенерировать изображение. API Canvas (Холст) предлагает три метода для манипулирования такими данными и обработки изображений:

- `getImageData(x, y, width, height)`. Считывает прямоугольную часть холста с размерами, определенными в свойствах, и преобразует ее в массив данных. Возвращает объект, к которому затем можно обращаться через свойства `width`, `height` и `data`;
- `putImageData(imagedata, x, y)`. Превращает данные, на которые ссылается `imagedata`, в изображение и выводит его на холст в позиции с координатами `x` и `y`. Таким образом, это противоположность метода `getImageData()`;

- `createImageData(width, height)`. Создает данные для пустого изображения. Все пикселы пустого изображения черного цвета и полностью прозрачные. Он также может принимать данные изображения через атрибут (который указывается вместо атрибутов `width` и `height`) и в таком случае возвращает размеры изображения, описываемого этими данными.

Позиция каждого из значений в массиве вычисляется по формуле (`width` · 4 · `y`) + (`x` · 4). Результат вычисления соответствует первому компоненту пиксела (красному цвету). Для получения позиций остальных компонентов к результату необходимо прибавлять по единице для каждого компонента. Например, (`width` · 4 · `y`) + (`x` · 4) + 1 указывает на зеленый компонент, (`width` · 4 · `y`) + (`x` · 4) + 2 — на синий, а (`width` · 4 · `y`) + + (`x` · 4) + 3 — на компонент альфа-канала. Давайте рассмотрим пример.

ВНИМАНИЕ

Из-за ограничений безопасности из элемента холста невозможно извлечь никакой информации после того, как на нем визуализируется изображение из внешнего источника. Метод getImageData() работает правильно только в том случае, когда документ и изображение принадлежат одному и тому же источнику (URL). Таким образом, для тестирования этого примера вам необходимо сохранить изображение с нашего сервера, расположенное по адресу http://www.minkbooks.com/content/snow.jpg, на своем компьютере (или использовать собственное изображение), а затем загрузить изображение, HTML-файл и JavaScript-файл на собственный сервер. Если вы всего лишь сохраните файл на своем компьютере и попробуете открыть пример в браузере, он не сработает.

Листинг 7.25. Создание негатива изображения

```
function initiate(){
  var elem=document.getElementById('canvas');
  canvas=elem.getContext('2d');

  var img=new Image();
  img.src="snow.jpg";
```

продолжение ⇩

Листинг 7.25 *(продолжение)*

```
  img.addEventListener("load", modimage, false);
}
function modimage(e){
  img=e.target;
  canvas.drawImage(img,0,0);
  var info=canvas.getImageData(0,0,175,262);

  var pos;
  for(x=0;x<=175;x++){
    for(y=0;y<=262;y++){
      pos=(info.width*4*y)+(x*4);
      info.data[pos]=255-info.data[pos];
      info.data[pos+1]=255-info.data[pos+1];
      info.data[pos+2]=255-info.data[pos+2];
    }
  }
  canvas.putImageData(info,0,0);
}
window.addEventListener("load", initiate, false);
```

На этот раз для обработки изображения после загрузки нам пришлось создать новую функцию (вместо того чтобы использовать анонимную). Сначала функция `modimage()` создает ссылку на изображение посредством свойства `target`, которое мы уже использовали в предыдущих примерах. Затем с помощью этой ссылки и метода `drawImage()` изображение выводится на холст в точке (0, 0). В этой части кода ничего необычного не происходит, но скоро все изменится.

> **ВНИМАНИЕ**
>
> Для того чтобы этот пример работал правильно, вам понадобится загрузить на свой сервер все файлы для него (включая файл snow.jpg, который вы можете найти по адресу *http://www.minkbooks.com/content/snow.jpg*).

Ширина изображения в примере равна 350 пикселам, а высота — 262 пикселам, поэтому, передавая методу `getImageData()` координату верхнего левого угла (0, 0) и смещение по горизонтали и вертикали 175 и 262 пиксела соответственно, мы вырезаем только левую половину исходного

изображения. Вырезанные данные изображения сохраняются в переменной `info`.

После того как нужная информация собрана, необходимо обработать каждый пиксел, для того чтобы получить желаемый результат (мы хотим получить негатив исходного изображения).

Для описания каждого цвета используется значение от 0 до 255; таким образом, для того чтобы получить негатив любого цвета, нужно вычесть его значение из 255, используя формулу `цвет` = 255 – `цвет`. Необходимо выполнить эти вычисления для каждого пиксела изображения, и с этой целью мы создали два цикла: один для столбцов, второй для строк. В циклах мы обрабатываем каждый пиксел, вычисляя соответствующее значение негатива. Обратите внимание на то, что цикл `for` для значений `x` начинается с 0 и заканчивается 175 (что соответствует ширине той части изображения, которую мы вырезали из холста), а цикл для значений `y` начинается с 0 и заканчивается 262 (что соответствует высоте исходного изображения и высоте вырезанной части).

После того как все пикселы пройдут обработку, переменная `info` с новыми данными изображения отправляется на холст посредством метода `putImageData()`. Обработанное изображение выводится в той же позиции, в которой находится исходное. Таким образом, только что созданный негатив заменяет левую половину исходного изображения.

Метод `getImageData()` возвращает объект, который можно обработать, обратившись к его свойствам (`width`, `height` и `data`), или сразу же передать методу `putImageData()`, не добавляя шаг обработки. Существует еще один способ извлечения данных холста, который возвращает его содержимое в форме закодированной в системе base64 строки. Эту строку в дальнейшем можно использовать в качестве источника для другого холста или HTML-элементов, подобных `<img>`, а можно отправить на сервер или сохранить в виде файла. Для этой цели в API существует специальный метод `toDataURL(type)`. У элемента `<canvas>` есть два свойства, `width` и `height`, и два метода, `getContext()` и `toDataURL()`. Второй метод возвращает URL данных (data:url), содержащий представление содержимого холста в формате PNG (или в другом формате изображения, указанном в атрибуте `type`).

Чуть позже мы рассмотрим несколько примеров использования `toDataURL()` и узнаем, как с помощью этого метода интегрировать API Canvas (Холст) с другими API.

Узоры

Узоры — это очень простое усовершенствование, которое, однако, позволяет сделать пути намного привлекательнее. С помощью узоров можно добавлять текстуру создаваемым на холсте фигурам. Процедура добавления аналогична работе с градиентами: нужно создать узор с помощью метода `createPattern()`, а затем связать его с путем, как любой другой цвет.

В `createPattern(image, type)` атрибут `image` представляет собой ссылку на изображение, а атрибут `type` может принимать одно из четырех значений: `repeat`, `repeat-x`, `repeat-y` или `no-repeat`.

Листинг 7.26. Добавление узора к путям на холсте

```
function initiate(){
  var elem=document.getElementById('canvas');
  canvas=elem.getContext('2d');
  var img=new Image();
  img.src="http://www.minkbooks.com/content/bricks.jpg";
  img.addEventListener("load", modimage, false);
}
function modimage(e){
  img=e.target;
  var pattern=canvas.createPattern(img,'repeat');
  canvas.fillStyle=pattern;
  canvas.fillRect(0,0,500,300);
}
window.addEventListener("load", initiate, false);
```

САМОСТОЯТЕЛЬНО

Поэкспериментируйте с другими значениями аргументов метода createPattern() и другими фигурами.

Анимация на холсте

Для создания анимации приходится прибегать к помощи обычного кода JavaScript. Не существует ни специальных методов для анимирования

объектов на холсте, ни четко определенной последовательности действий для решения этой задачи. По сути, нужно очищать область холста, на которой должна происходить анимация, рисовать там новые фигуры и повторять этот процесс снова и снова. Когда вы нарисуете фигуры на холсте, их уже нельзя будет передвинуть. Строить анимированное изображение можно только одним способом — стирая его часть и снова рисуя фигуры на этом месте. Вот почему в играх или приложениях, требующих анимирования большого количества объектов, лучше использовать изображения, а не фигуры, построенные на базе сложных путей (для создания игр обычно применяются изображения в формате PNG).

Разработчики программного обеспечения придумали множество техник создания анимированных изображений. Одни чрезвычайно просты, а другие не уступают по сложности самим приложениям, для которых создается анимация. Мы сконструируем простой пример, в котором будем очищать холст методом `clearRect()` и снова рисовать на нем фигуры. Таким образом, для создания анимации нам понадобится всего одна функция. Однако помните: если вы намереваетесь работать со сложными анимационными эффектами, стоит для начала хорошенько изучить учебник продвинутого уровня по программированию на JavaScript.

Листинг 7.27. Наша первая анимация

```
function initiate(){
  var elem=document.getElementById('canvas');
  canvas=elem.getContext('2d');

  window.addEventListener('mousemove', animation, false);
}
function animation(e){
  canvas.clearRect(0,0,300,500);

  var xmouse=e.clientX;
  var ymouse=e.clientY;
  var xcenter=220;
  var ycenter=150;
  var ang=Math.atan2(xmouse-xcenter,ymouse-ycenter);
  var x=xcenter+Math.round(Math.sin(ang)*10);
  var y=ycenter+Math.round(Math.cos(ang)*10);

  canvas.beginPath();
  canvas.arc(xcenter,ycenter,20,0,Math.PI*2, false);
```

продолжение ⇗

Листинг 7.27 *(продолжение)*

```
    canvas.moveTo(xcenter+70,150);
    canvas.arc(xcenter+50,150,20,0,Math.PI*2, false);
    canvas.stroke();

    canvas.beginPath();
    canvas.moveTo(x+10,y);
    canvas.arc(x,y,10,0,Math.PI*2, false);
    canvas.moveTo(x+60,y);
    canvas.arc(x+50,y,10,0,Math.PI*2, false);
    canvas.fill();
}
window.addEventListener("load", initiate, false);
```

В коде из листинга 7.27 мы создаем рисунок глаз, следящих за указателем мыши. Для перемещения зрачков обновляем позицию соответствующих элементов каждый раз, когда указатель мыши сдвигается. Для этого в функции `initiate()` используется прослушиватель события `mousemove`: когда событие срабатывает, происходит вызов функции `animation()`.

Выполнение функции начинается с очистки холста инструкцией `clearRect(0, 0, 300, 500)`. После этого считывается позиция указателя мыши, а в переменных `xcenter` и `ycenter` сохраняется местоположение первого глаза.

После инициализации переменных настает время вычислений. Используя местоположение указателя мыши и позицию центра левого глаза, мы вычисляем угол наклона невидимого отрезка, соединяющего две эти точки. Для этого используется стандартный метод JavaScript `atan2`. Затем на основе значения угла по формуле `xcenter + Math.round(Math.sin(ang) × 10)` вычисляются точные координаты центра зрачка. Число 10 в формуле представляет собой расстояние от центра глаза до центра зрачка (зрачок никогда не перемещается в центр, он всегда остается у края глаза).

Наконец, получив все значения, мы можем нарисовать на холсте глаза. Первый путь объединяет две окружности, соответствующие глазам. Первый метод `arc()` рисует окружность с координатами `xcenter` и `ycenter`, а второй `arc()` создает аналогичную окружность на 50 пикселов правее первой, для чего ему передается инструкция `arc(xcenter+50, 150, 20, 0, Math.PI*2, false)`.

Анимированная часть рисунка определяется вторым путем. Для создания этого пути используются переменные x и y со значениями, вычислен-

ными ранее на основе величины угла. Оба зрачка визуализируются как черные круги с помощью метода `fill()`.

Процесс повторяется, и значения пересчитываются при каждом срабатывании события `mousemove`.

САМОСТОЯТЕЛЬНО

Скопируйте код из листинга 7.27 в JavaScript-файл с именем canvas.js и откройте в браузере HTML-файл, содержащий шаблон из листинга 7.1.

Обработка видео на холсте

Как и в случае с анимацией, не существует специального метода для отображения видео на элементе холста. Единственный способ воспроизвести видео — поочередно считать все кадры из элемента `<video>` и нарисовать их на холсте как отдельные изображения, используя `drawImage()`. Таким образом, для обработки видео нужно всего лишь скомбинировать техники, которые мы изучили ранее.

Давайте сначала построим новый шаблон, а затем напишем код, чтобы вывести наше видео на холст.

Листинг 7.28. Шаблон для видео на холсте

```
<!DOCTYPE html>
<html lang="ru">
<head>
  <title>Видео на холсте</title>
  <style>
    .boxes{
      display: inline-block;
      margin: 10px;
      padding: 5px;
      border: 1px solid #999999;
    }
  </style>
  <script src="canvasvideo.js"></script>
```

продолжение ⇩

Листинг 7.28 *(продолжение)*

```
</head>
<body>
  <section class="boxes">
    <video id="media" width="483" height="272">
      <source src="http://www.minkbooks.com/content/trailer2.mp4">
      <source src="http://www.minkbooks.com/content/trailer2.ogg">
    </video>
  </section>
  <section class="boxes">
    <canvas id="canvas" width="483" height="272">
      В вашем браузере элемент canvas не поддерживается
    </canvas>
  </section>
</body>
</html>
```

В шаблоне из листинга 7.28 присутствуют два специальных компонента: элемент `<video>` и элемент `<canvas>`. Благодаря их комбинации мы сможем обработать видео и вывести его на холст.

Кроме того, шаблон включает в себя встроенные CSS-стили для блоков и ссылается на новый JavaScript-файл с именем canvasvideo.js, содержащий следующий код.

Листинг 7.29. Преобразование цветного видео в черно-белое

```
function initiate(){
  var elem=document.getElementById('canvas');
  canvas=elem.getContext('2d');
  video=document.getElementById('media');

  video.addEventListener('click', push, false);
}
function push(){
  if(!video.paused && !video.ended){
    video.pause();
    window.clearInterval(loop);
  }else{
    video.play();
    loop=setInterval(processFrames, 33);
  }
}
```

```
function processFrames(){
  canvas.drawImage(video,0,0);

  var info=canvas.getImageData(0,0,483,272);
  var pos;
  var gray;
  for(x=0;x<=483;x++){
    for(y=0;y<=272;y++){
      pos=(info.width*4*y)+(x*4);
  gray=parseInt(info.data[pos]*0.2989 + info.data[pos+1]*0.5870 +
      +info.data[pos+2]*0.1140);
      info.data[pos]=gray;
      info.data[pos+1]=gray;
      info.data[pos+2]=gray;
    }
  }
  canvas.putImageData(info,0,0);
}
window.addEventListener("load", initiate, false);
```

САМОСТОЯТЕЛЬНО

Создайте новый HTML-файл с кодом из листинга 7.28 и JavaScript-файл под названием canvasvideo.js с кодом из листинга 7.29. Для запуска видео щелкните в левом поле на экране.

ВНИМАНИЕ

В этом примере для обработки данных изображения используются методы getImageData() и putImageData(). Как уже говорилось, эти методы извлекают информацию из элемента холста. По соображениям безопасности возможность извлечь данные холста отключается, если элемент получает содержимое из источника, не совпадающего с источником документа (документ принадлежит одному домену, а видео — другому). По этой причине для тестирования данного примера понадобится загрузить видеофайлы с нашего вебсайта и сохранить все задействованные в примере файлы на собственном сервере.

Давайте рассмотрим код в листинге 7.29. Напомню, что для обработки видео на холсте мы всего лишь комбинируем изученные ранее примеры кода и приемы. В этом коде мы используем функцию `push()` из главы 5, для того чтобы запускать и останавливать видео по щелчку на элементе. Мы также создали функцию под названием `processFrames`, содержащую код из листинга 7.25 (единственное различие заключается в том, что вместо инвертирования изображения мы с помощью определенной формулы превращаем все цвета в оттенки серого).

Функция `push()` выполняет две задачи: запускает или останавливает видео, а также инициирует интервал, который каждые 33 мс вызывает функцию `processFrames()`. Эта функция считывает кадр из элемента `<video>` и выводит его на холст посредством инструкции `drawImage(video, 0, 0)`. Затем данные холста извлекаются методом `getImageData()`, и каждый пиксел этого кадра обрабатывается в двойном цикле.

Мы применяем один из самых распространенных способов превращения цветов в оттенки серого, описание которого несложно найти в Интернете. Формула выглядит так: `красный` × 0,2989 + `зеленый` × 0,587 + `синий` × 0,114. Результат вычисления присваивается всем цветовым компонентам (красному, зеленому и синему) каждого пиксела. В коде для этого используется переменная `gray`.

Завершается процесс выводом кадра на холст в методе `putImageData()`.

ВНИМАНИЕ

Это учебный пример, а в действительности обрабатывать видео в режиме реального времени не рекомендуется. В зависимости от конфигурации вашего компьютера и от того, в каком браузере выполняется приложение, вы можете заметить, что обработка выполняется с задержкой. При создании реальных приложений JavaScript всегда необходимо особое внимание обращать на быстродействие.

Краткий справочник. API Canvas (Холст)

API Canvas (Холст), вероятно, самый сложный и объемный во всей спецификации HTML5. Он предоставляет несколько методов и свойств для создания графических приложений на базе элемента `<canvas>`.

Методы

В API Canvas (Холст) входят следующие методы:

- `getContext(context)`. Создает контекст для холста, на котором вы затем будете рисовать. Принимает одно из двух значений: `2d` или `3d` для двухмерной и трехмерной графики соответственно;
- `fillRect(x, y, width, height)`. Позволяет нарисовать прямо на холсте, в точке (x, y), залитый цветом прямоугольник со сторонами `width` и `height`;
- `strokeRect(x, y, width, height)`. Рисует прямо на холсте, в точке (x, y), прямоугольный контур со сторонами `width` и `height`;
- `clearRect(x, y, width, height)`. Очищает на холсте прямоугольную область, координаты и размеры которой заданы его атрибутами;
- `createLinearGradient(x1, y1, x2, y2)`. Создает линейный градиент, который затем можно привязать к фигуре, как любой другой цвет, — через свойство `fillStyle`. В атрибутах необходимо указать только начальную и конечную позиции градиента (относительно холста). Для объявления цветов градиента данный метод необходимо сочетать с методом `addColorStop()`;
- `createRadialGradient(x1, y1, r1, x2, y2, r2)`. Создает радиальный градиент, который затем можно привязать к фигуре, как любой другой цвет, — через свойство `fillStyle`. Градиент состоит из двух окружностей. Атрибуты метода определяют только позиции и радиусы окружностей (относительно холста). Для объявления цветов градиента данный метод необходимо сочетать с методом `addColorStop()`;
- `addColorStop(position, color)`. Применяется для объявления цветов градиента. Атрибут `position` принимает значение от 0,0 до 1,0 и указывает, в какой точке цвет начинает растворяться;
- `beginPath()`. Используется для объявления начала нового пути;
- `closePath()`. Можно добавить в конце пути, для того чтобы закрыть его. Создает прямую линию между последней позицией пера и начальной точкой пути. Метод не обязательно применять в случаях, когда путь должен оставаться открытым или когда для визуализации пути используется метод `fill()`;
- `stroke()`. Предназначен для визуализации контура пути;
- `fill()`. Предназначен для визуализации пути как фигуры, залитой сплошным цветом;

- `clip()`. Применяется для создания новой области обрезки в форме пути. Если графические объекты отправляются на холст после вызова данного метода, то отображаются только те их фрагменты, которые попадают внутрь фигуры;
- `moveTo(x, y)`. Переносит виртуальное перо в новое местоположение. Следующий метод продолжит рисование пути с этой точки;
- `lineTo(x, y)`. Добавляет к пути прямую линию, начиная от текущей позиции пера и до точки, определяемой атрибутами `x` и `y`;
- `rect(x, y, width, height)`. Добавляет к пути прямоугольник в точке (x, y) со сторонами `width` и `height`;
- `arc(x, y, radius, startAngle, endAngle, direction)`. Добавляет к пути дугу. Центр дуги находится в точке (x, y), начальный и конечный углы объявляются в радианах, а `direction` — это булево значение, указывающее направление: по часовой стрелке или против часовой стрелки. Для преобразования градусов в радианы используйте формулу `Math.PI`/180 × градусы;
- `quadraticCurveTo(cpx, cpy, x, y)`. Добавляет к пути квадратичную кривую Безье. Кривая начинается в текущей позиции пера и заканчивается в точке (x, y). Атрибуты `cpx` и `cpy` определяют позицию контрольной точки, которая задает форму кривой;
- `bezierCurveTo(cp1x, cp1y, cp2x, cp2y, x, y)`. Добавляет к пути кубическую кривую Безье. Кривая начинается в текущей позиции пера и заканчивается в точке (x, y). Атрибуты `cp1x`, `cp1y`, `cp2x` и `cp2y` определяют позиции двух контрольных точек, которые задают форму кривой;
- `strokeText(text, x, y, max)`. Рисует контурный текст прямо на холсте. Атрибут `max` необязательный, он задает максимальный размер текста;
- `fillText(text, x, y, max)`. Рисует прямо на холсте текст, представляющий собой залитые цветом фигуры. Атрибут `max` необязательный, он задает максимальный размер текста;
- `measureText(text)`. Вычисляет размер области, которую текст займет на холсте при визуализации с текущими стилями. Для обращения к получившемуся значению используется свойство `width`;
- `translate(x, y)`. Переносит начало координат холста в точку (x, y). Первоначально точка с координатами (0, 0) находится в верхнем левом углу области, соответствующей элементу `<canvas>`;

- `rotate(angle)`. Поворачивает холст вокруг начала координат. Величину угла нужно указывать в радианах. Для преобразования градусов в радианы используйте формулу `Math.PI`/180 × градусы;
- `scale(x, y)`. Меняет масштаб холста. Значения по умолчанию (1,0, 1,0). Можно также задавать отрицательные значения;
- `transform(m1, m2, m3, m4, dx, dy)`. Модифицирует матрицу трансформации холста. Новые значения матрицы вычисляются на основе предыдущих;
- `setTransform(m1, m2, m3, m4, dx, dy)`. Модифицирует матрицу трансформации холста. Отменяет предыдущие модификации матрицы и задает новые значения;
- `save()`. Сохраняет текущее состояние холста, включая значения матрицы трансформации, свойства стилизации и маску обрезки;
- `restore()`. Восстанавливает последнее сохраненное состояние холста, включая матрицу трансформации, свойства стилизации и маску обрезки;
- `drawImage()`. Выводит на холст изображение и поддерживает три варианта синтаксиса. Синтаксис `drawImage(image, x, y)` позволяет вывести изображение на холст в точке (x, y). Синтаксис `drawImage(image, x, y, width, height)` позволяет вывести изображение на холст в точке (x, y), с новыми размерами, определяемыми параметрами `width` и `height`. Синтаксис `drawImage(image, x1, y1, width1, height1, x2, y2, width2, height2)` позволяет вырезать из исходного изображения часть, определяемую параметрами `x1`, `y1`, `width1` и `height1`, и вывести ее на холст в точке (x_2, y_2) с новыми размерами `width2` и `height2`;
- `getImageData(x, y, width, height)`. Считывает содержимое фрагмента холста и сохраняет его в виде объекта. К значениям объекта можно обращаться через свойства `width`, `height` и `data`. Первые два свойства возвращают размер вырезанного фрагмента изображения, а свойство `data` возвращает массив значений, описывающих цвета пикселов. Для доступа к значениям используйте формулу (`width` × y) + (x × 4);
- `putImageData(imagedata, x, y)`. Визуализирует данные, сохраненные в объекте `imagedata`, как изображение на холсте;
- `createImageData(width, height)`. Создает новое изображение в формате набора данных. Первоначально все пикселы изображения черные прозрачные. Вместо параметров `width` и `height` метод может принимать в качестве атрибута данные изображения. В таком случае размер

нового изображения соответствует размеру, закодированному в этом наборе данных;

- `createPattern(image, type)`. Создает на основе данных изображения узор, который затем можно привязать к фигуре с помощью свойства `fillStyle`, как любой другой цвет. Возможные значения атрибута `type` — `repeat`, `repeat-x`, `repeat-y` и `no-repeat`.

Свойства

Далее перечислены свойства, входящие только в API Canvas (Холст):

- `strokeStyle`. Определяет цвет линий фигур. Может принимать любые значения, поддерживаемые CSS, включая такие функции, как `rgb()` и `rgba()`;
- `fillStyle`. Определяет цвет заливки сплошных фигур. Может принимать любые значения, поддерживаемые CSS, включая такие функции, как `rgb()` и `rgba()`. Также используется для привязки к фигурам градиентов и узоров (эти стили сначала сохраняются в переменной, а затем переменная связывается с данным свойством, как любой другой цвет);
- `globalAlpha`. Позволяет задать уровень прозрачности любой фигуры. Принимает значения от 0,0 (полностью непрозрачный) до 1,0 (абсолютно прозрачный);
- `lineWidth`. Устанавливает толщину линии. Значение по умолчанию 1,0;
- `lineCap`. Определяет форму окончания линий. Существует три возможных значения: `butt` (обычное окончание), `round` (полукруглое окончание) и `square` (квадратное окончание);
- `lineJoin`. Определяет форму соединения двух линий. Может принимать одно из трех значений: `round` (скругленное соединение), `bevel` (срезанное соединение) и `miter` (вытянутое соединение — линии сходятся в одну точку);
- `miterLimit`. Принимает числовое значение, указывающее, как сильно будет вытягиваться соединение двух линий в случае, когда свойству `lineJoin` присвоено значение `miter`;
- `font`. Аналогично свойству `font` из CSS и принимает те же значения для определения стилей текста;
- `textAlign`. Определяет способ выравнивания текста. Возможные значения — `start`, `end`, `left`, `right` и `center`;

- `textBaseline`. Описывает выравнивание текста по вертикали. Возможные значения — `top`, `hanging`, `middle`, `alphabetic`, `ideographic` и `bottom`;
- `shadowColor`. Устанавливает цвет тени. Принимает значения, разрешенные в CSS;
- `shadowOffsetX`. Объявляет расстояние между объектом и тенью по горизонтали в количестве единиц;
- `shadowOffsetY`. Объявляет расстояние между объектом и тенью по вертикали в количестве единиц;
- `shadowBlur`. Принимает числовое значение, описывающее эффект размытия тени;
- `globalCompositeOperation`. Определяет способ рисования новых фигур на холсте поверх уже имеющихся там фигур. Может принимать разные значения: `source-over`, `source-in`, `source-out`, `source-atop`, `lighter`, `xor`, `destination-over`, `destination-in`, `destination-out`, `destination-atop`, `darker` и `copy`. Значение по умолчанию `source-over`, при этом новые фигуры просто выводятся поверх существующих.

8 API перетаскивания

Перетаскивание в Сети

В настольных приложениях мы постоянно перетаскиваем элементы с одного места на другое, но не ожидаем встретить эту функциональность в веб-приложениях. И причина не в том, что веб-приложения чем-то кардинально отличаются от настольных, а в том, что разработчикам никогда не предлагали стандартную технологию для реализации возможностей перетаскивания.

Теперь благодаря появлению в спецификации HTML5 API Drag and Drop (Перетаскивание) мы наконец-то получили возможность создавать для Сети программное обеспечение, ничем не отличающееся от настольных приложений.

Новые события

Одна из самых важных составляющих нового API — набор из семи специальных событий, предназначенных для обработки различных ситуаций перетаскивания. Часть событий срабатывает на источнике (перетаскиваемом элементе), часть — на цели (элементе, на который перетаскивается элемент-источник). Когда пользователь выполняет операцию перетаскивания, на источнике срабатывают следующие три события:

- `dragstart`. Срабатывает в момент, когда операция перетаскивания начинается. Система настраивает данные, связанные с элементом-источником, именно в этот момент;

- `drag`. Похоже на событие `mousemove`, но срабатывает во время операции перетаскивания на элементе-источнике;
- `dragend`. Срабатывает, когда операция перетаскивания заканчивается (успешно или неудачно).

Следующие события срабатывают на целевом элементе на протяжении той же операции:

- `dragenter`. Срабатывает, когда во время операции перетаскивания указатель мыши оказывается в области предполагаемого целевого элемента;
- `dragover`. Похоже на событие `mousemove`, но срабатывает во время операции перетаскивания на возможных целевых элементах;
- `drop`. Срабатывает, когда во время операции перетаскивания пользователь отпускает элемент-источник над целевым элементом;
- `dragleave`. Срабатывает, когда указатель мыши покидает область возможного целевого элемента во время операции перетаскивания. Используется совместно с событием `dragenter` и обеспечивает взаимодействие с объектами приложения, помогая идентифицировать целевые элементы.

Прежде чем вы начнете внедрять перетаскивание в своих приложениях, следует продумать важный момент. Во время операции перетаскивания браузеры выполняют действия по умолчанию. Для того чтобы получать желаемые результаты, часто приходится запрещать поведение по умолчанию и настраивать действия нужным образом. Для некоторых событий, таких как `dragenter`, `dragover` и `drop`, запрещать поведение по умолчанию необходимо всегда, даже если вы настраиваете для соответствующих действий какую-то специальную обработку. Давайте рассмотрим простой пример, для того чтобы на практике разобраться, как это происходит.

Листинг 8.1. Шаблон для операции перетаскивания

```
<!DOCTYPE html>
<html lang="ru">
<head>
  <title>Перетаскивание</title>
  <link rel="stylesheet" href="dragdrop.css">
  <script src="dragdrop.js"></script>
</head>
<body>
```

продолжение ⇩

Листинг 8.1 *(продолжение)*

```
    <section id="dropbox">
      Перетащите изображение сюда
    </section>
    <section id="picturesbox">
      <img id="image"
       src="http://www.minkbooks.com/content/monster1.gif">
    </section>
  </body>
  </html>
```

В HTML-документе из листинга 8.1 присутствует элемент `<section>` с идентификатором `dropbox`. Он будет использоваться в качестве цели, а изображение послужит источником. Также в коде можно увидеть ссылки на два файла: со стилями CSS и с кодом JavaScript, в котором происходит обработка операции.

Листинг 8.2. Стили для нашего шаблона (файл dragdrop.css)

```
#dropbox{
float: left;
width: 500px;
height: 300px;
margin: 10px;
border: 1px solid #999999;
}
#picturesbox{
float: left;
width: 320px;
margin: 10px;
border: 1px solid #999999;
}
#picturesbox > img{
float: left;
padding: 5px;
}
```

Правила в листинге 8.2 всего лишь описывают стили блоков, помогающих идентифицировать поле-источник и целевое поле операции перетаскивания.

Листинг 8.3. Элементарный код для операции перетаскивания

```
function initiate(){
  source1=document.getElementById('image');
  source1.addEventListener('dragstart', dragged, false);

  drop=document.getElementById('dropbox');
  drop.addEventListener('dragenter', function(e){
    e.preventDefault(); }, false);
  drop.addEventListener('dragover', function(e){
    e.preventDefault(); }, false);
  drop.addEventListener('drop', dropped, false);
}
function dragged(e){
  var code='<img src="'+source1.getAttribute('src')+'">';
  e.dataTransfer.setData('Text', code);
}
function dropped(e){
  e.preventDefault();
   drop.innerHTML=e.dataTransfer.getData('Text');
}
window.addEventListener('load', initiate, false);
```

Для настройки процедуры перетаскивания можно применить пару атрибутов элементов HTML, но основное действие происходит все же в коде JavaScript. В листинге 8.3 представлены три функции: `initiate()` добавляет прослушиватели событий для операции перетаскивания, а `dragged()` и `dropped()` генерируют и получают информацию, которая передается между объектами в процессе обработки действия.

Для того чтобы обработать обычную операцию перетаскивания, нужно подготовить информацию, которую будут передавать друг другу элемент-источник и целевой элемент. Для этого мы добавили прослушиватель события `dragstart`. Он при срабатывании события вызывает функцию `dragged()`, а подготавливает необходимую информацию входящий в функцию метод `setData()`.

На большинстве элементов документа прием перетаскиваемых объектов по умолчанию не поддерживается и никак не обрабатывается. Таким образом, для того чтобы наша «зона приема» могла принимать элемент, необходимо запретить поведение по умолчанию. Мы сделали это, добавив прослушиватели событий `dragenter` и `dragover`, а также анонимную функцию, которая выполняет метод `preventDefault()`.

Наконец, мы создали прослушиватель события `drop`, который вызывает функцию `dropped()`, — эта функция принимает и обрабатывает данные, отправленные источником.

ПОВТОРЯЕМ ОСНОВЫ

Для событий dragenter и dragover мы использовали анонимную функцию, содержащую отменяющий эти события метод preventDefault(). Для того чтобы можно было сослаться на событие внутри функции, ей передается переменная e. Если вы хотите побольше узнать об анонимных функциях, зайдите на наш веб-сайт и изучите ссылки для этой главы.

Когда пользователь начинает перетаскивать рисунок, срабатывает событие `dragstart` и вызывается функция `dragged()`. В этой функции мы извлекаем значение атрибута `src` перетаскиваемого элемента и настраиваем передаваемые в элементе данные с помощью метода `setData()` объекта `dataTransfer`. На другой стороне процесса, когда пользователь отпускает элемент над зоной приема, срабатывает событие `drop` и вызывается функция `dropped()`. Эта функция всего лишь модифицирует содержимое зоны приема, добавляя в нее информацию, полученную с помощью метода `getData()`. При срабатывании этого события браузеры также выполняют определенные действия по умолчанию (например, открывают ссылку или обновляют содержимое окна, для того чтобы показать, что изображение перенесено в другое место), поэтому мы запрещаем поведение по умолчанию методом `preventDefault()` — так же, как сделали это для других событий чуть раньше.

САМОСТОЯТЕЛЬНО

Создайте HTML-файл с шаблоном из листинга 8.1, CSS-файл с именем dragdrop.css, содержащий стили из листинга 8.2, и JavaScript-файл с именем dragdrop.js, содержащий код из листинга 8.3. Для тестирования примера откройте файл HTML в своем браузере.

Объект dataTransfer

Это объект, который содержит информацию, задействованную в операции перетаскивания. С объектом `dataTransfer` связаны несколько мето-

дов и свойств. Мы уже использовали `setData()` и `getData()` в примере из листинга 8.3. Именно эти методы, а также дополнительный метод `clearData()` отвечают за передаваемую информацию:

- `setData(type, data)`. Используется для объявления передаваемых данных и их типа. Принимает данные обычных типов (таких, как `text/plain`, `text/html` и `text/uri-list`), специальных типов (таких, как `URL` и `Text`) и даже персонализированных типов. Метод `setData()` необходимо отдельно вызывать для каждого типа данных, которые мы хотим отправить в ходе одной операции;
- `getData(type)`. Возвращает отправленные элементом-источником данные указанного типа;
- `clearData()`. Удаляет данные указанного типа.

В функции `dragged()` в листинге 8.3 содержится короткий HTML-код, включающий в себя значение атрибута `src` элемента, на котором сработало событие `dragstart`. Мы сохраняем этот код в переменной `code` и отправляем ее целевому элементу посредством метода `setData()`. Поскольку отправляем текст, в качестве значения первого аргумента указан соответствующий тип, `Text`.

ВНИМАНИЕ

Можно было бы использовать в нашем примере более подходящий тип, например text/html или даже персонализированный тип, однако пока что многие браузеры поддерживают крайне ограниченный набор типов. Таким образом, тип Text делает наше приложение лучше совместимым с существующими браузерами, и примеры из этой главы можно тестировать без какой-либо дополнительной подготовки.

ВНИМАНИЕ

Чтобы побольше узнать о типах данных для операции перетаскивания, зайдите на наш веб-сайт и изучите ссылки для этой главы.

Когда в функции `dropped()` мы извлекаем переданные данные с помощью метода `getData()`, необходимо указывать, данные какого типа мы считываем. Смысл в том, что одновременно один элемент может отправить данные разных типов. Например, объект изображения может отправить само

изображение, URL-адрес и текст с описанием. Для отправки и получения данных каждого из типов применяются отдельные методы `setData()` и `getData()` с указанием требуемого типа.

У объекта `dataTransfer` есть еще несколько методов и свойств, которые также могут быть полезны при построении приложений:

- `setDragImage(element, x, y)`. Некоторые браузеры показывают небольшой эскиз перетаскиваемого элемента. Этот метод применяется для настройки эскиза и выбора его точного местоположения относительно указателя мыши (с помощью атрибутов `x` и `y`);
- `types`. Возвращает массив, содержащий типы, которые были отправлены в событии `dragstart` (нашим кодом или самим браузером). Можно сохранить этот массив в переменной (`list=dataTransfer.types`), а затем считать его содержимое в цикле;
- `files`. Возвращает массив, содержащий информацию о перетаскиваемых файлах;
- `dropEffect`. Возвращает тип операции, которая выбрана в данный момент для выполнения после того, как пользователь отпускает элемент. Также может принимать значение, и в этом случае меняет выбранную операцию. Допустимые значения: `none`, `copy`, `link` и `move`;
- `effectAllowed`. Возвращает типы допустимых операций. Ему можно передать значение, для того чтобы поменять допустимые операции. Возможные значения: `none`, `copy`, `copyLink`, `copyMove`, `link`, `linkMove`, `move`, `all` и `uninitialized`.

Применим некоторые из этих свойств и методов в следующих примерах.

События dragenter, dragleave и dragend

Пока что с событием `dragenter` мы ничего не делали — всего лишь отменили его, для того чтобы запретить поведение браузера по умолчанию. Точно так же не воспользовались возможностями событий `dragleave` и `dragend`. Однако это важные события, обеспечивающие обратную связь, благодаря чему мы можем давать пользователям подсказки относительно перемещения элементов по экрану.

Листинг 8.4. Управление процессом перетаскивания от и до

```
function initiate(){
  source1=document.getElementById('image');
```

```
  source1.addEventListener('dragstart', dragged, false);
  source1.addEventListener('dragend', ending, false);

  drop=document.getElementById('dropbox');
  drop.addEventListener('dragenter', entering, false);
  drop.addEventListener('dragleave', leaving, false);
  drop.addEventListener('dragover', function(e){
    e.preventDefault(); }, false);
  drop.addEventListener('drop', dropped, false);
}
function entering(e){
  e.preventDefault();
  drop.style.background='rgba(0,150,0,.2)';
}
function leaving(e){
  e.preventDefault();
  drop.style.background='#FFFFFF';
}
function ending(e){
  elem=e.target;
  elem.style.visibility='hidden';
}
function dragged(e){
  var code='<img src="'+source1.getAttribute('src')+'">';
  e.dataTransfer.setData('Text', code);
}
function dropped(e){
  e.preventDefault();
  drop.style.background='#FFFFFF';
  drop.innerHTML=e.dataTransfer.getData('Text');
}
window.addEventListener('load', initiate, false);
```

JavaScript-код из листинга 8.4 заменяет собой код из листинга 8.3. В новом коде мы добавили две функции для зоны приема и одну — для источника. Функции `entering()` и `leaving()` меняют фоновый цвет зоны приема каждый раз, когда при перетаскивании элемента указатель мыши оказывается над ней или покидает ее (эти действия инициируют события `dragenter` и `dragleave`). Помимо этого прослушиватель события `dragend` вызывает функцию `ending()` в тот момент, когда пользователь отпускает перетаскиваемый элемент. Обратите внимание на то, что эти

события и функция не проверяют, завершился процесс успешно или нет, вам это нужно делать самостоятельно.

Благодаря этим функциям каждый раз, когда в процессе перетаскивания указатель мыши заходит в зону приема, соответствующий блок подсвечивается светло-зеленым цветом, а когда пользователь отпускает элемент, исходное изображение скрывается. Наблюдаемые изменения никак не влияют на процесс перетаскивания, но они обеспечивают обратную связь от приложения, направляя пользователя во время операции.

Для запрещения поведения по умолчанию нам приходится применять метод `preventDefault()` в каждой функции, даже если мы дополнительно настраиваем какие-то специальные действия.

САМОСТОЯТЕЛЬНО

Скопируйте код из листинга 8.4 в свой JavaScript-файл, откройте HTML-документ из листинга 8.1 в браузере и перетащите изображение на зону приема.

Выбор допустимого источника

Не существует никакого специального метода, который позволил бы проверить, допускается перетаскивание данных из источника или нет. На информацию, возвращаемую методом `getData()`, положиться нельзя, так как, даже если мы получим данные указанного в соответствующем атрибуте типа, может оказаться так, что их передал другой источник, просто тип случайно совпал. У объекта `dataTransfer` есть свойство под названием `types`. Оно возвращает массив со списком типов, определенных в событии `dragstart`, однако тоже не помогает решить задачу валидации.

По этой причине техники выбора и валидации данных, передаваемых в операции перетаскивания, могут в значительной степени варьироваться. Создавайте настолько простые или сложные процедуры проверки, насколько необходимо в конкретной ситуации.

Листинг 8.5. Новый шаблон с дополнительными источниками

```
<!DOCTYPE html>
<html lang="ru">
<head>
  <title>Перетаскивание</title>
  <link rel="stylesheet" href="dragdrop.css">
```

```
  <script src="dragdrop.js"></script>
</head>
<body>
  <section id="dropbox">
    Перетаскивайте изображения сюда
  </section>
  <section id="picturesbox">
    <img id="image1" src="http://www.minkbooks.com/content/monster1.gif">
    <img id="image2" src="http://www.minkbooks.com/content/monster2.gif">
    <img id="image3" src="http://www.minkbooks.com/content/monster3.gif">
    <img id="image4" src="http://www.minkbooks.com/content/monster4.gif">
  </section>
</body>
</html>
```

Используя новый HTML-документ из листинга 8.5, мы будем фильтровать источники, проверяя атрибут `id` каждого изображения. Следующий JavaScript-код подсказывает, какое изображение можно отпустить на зоне приема, а какое нельзя.

Листинг 8.6. Отправка атрибута id

```
function initiate(){
  var images=document.querySelectorAll('#picturesbox > img');
  for(var i=0; i<images.length; i++){
    images[i].addEventListener('dragstart', dragged, false);
}

  drop=document.getElementById('dropbox');
  drop.addEventListener('dragenter', function(e){
    e.preventDefault(); }, false);
  drop.addEventListener('dragover', function(e){
    e.preventDefault(); }, false);
  drop.addEventListener('drop', dropped, false);
}
function dragged(e){
  elem=e.target;
  e.dataTransfer.setData('Text', elem.getAttribute('id'));
}
function dropped(e){
  e.preventDefault();
```

продолжение ⇩

Листинг 8.6 *(продолжение)*

```
    var id=e.dataTransfer.getData('Text');
    if(id!="image4"){
      var src=document.getElementById(id).src;
      drop.innerHTML='<img src="'+src+'">';
    }else{
      drop.innerHTML='не разрешается';
    }
}
window.addEventListener('load', initiate, false);
```

В листинге 8.6 поменялось не так уж много по сравнению с предыдущими примерами кода. Мы используем метод `querySelectorAll()` для добавления прослушивателя события `dragstart` к каждому изображению внутри элемента `picturesbox`, отправляем атрибут `id` посредством метода `setData()` каждый раз, когда пользователь начинает перетаскивать любое из изображений, и проверяем значение этого атрибута `id` в функции `dropped()`, для того чтобы не допускать перетаскивания изображения с `id="image4"` (если пользователь отпустит это изображение над зоной приема, вместо картинки там появится надпись «Не разрешается»).

Это очень простой фильтр. В качестве альтернативных примеров вы могли бы применить метод `querySelectorAll()` в функции `dropped()` для проверки того, что полученное изображение действительно принадлежит элементу `picturesbox`, или поработать со свойствами объекта `dataTransfer` (такими, как `types` и `files`), но в любом случае каждый раз это уникальный процесс. Другими словами, всегда необходимо настраивать проверку самостоятельно.

Метод setDragImage()

Изменение эскиза, который отображается рядом с указателем мыши в процессе перетаскивания, может казаться бесполезным занятием, но иногда это помогает избежать лишней головной боли. Метод `setDragImage()` не только позволяет менять эскиз, но также принимает два атрибута, `x` и `y`, устанавливающих позицию эскиза относительно указателя мыши. Браузер обычно сам генерирует эскиз источника, а его позиция относительно указателя устанавливается в зависимости от того, в какой точке находился указатель мыши, когда пользователь начал перетаскивание. Метод `setDragImage()` позволяет задать конкретную позицию, которая не будет меняться между операциями перетаскивания.

Листинг 8.7. Использование элемента <canvas> в качестве зоны приема

```
<!DOCTYPE html>
<html lang="ru">
<head>
  <title>Перетаскивание</title>
  <link rel="stylesheet" href="dragdrop.css">
  <script src="dragdrop.js"></script>
</head>
<body>
  <section id="dropbox">
    <canvas id="canvas" width="500" height="300"></canvas>
  </section>
  <section id="picturesbox">
    <img id="image1" src="http://www.minkbooks.com/content/monster1.gif">
    <img id="image2" src="http://www.minkbooks.com/content/monster2.gif">
    <img id="image3" src="http://www.minkbooks.com/content/monster3.gif">
    <img id="image4" src="http://www.minkbooks.com/content/monster4.gif">
  </section>
</body>
</html>
```

Используя новый HTML-документ из листинга 8.7, мы продемонстрируем важность метода `setDragImage()`. Для этого настроим элемент `<canvas>` в качестве зоны приема.

Листинг 8.8. Маленькое приложение с функциональностью перетаскивания

```
function initiate(){
  var images=document.querySelectorAll('#picturesbox > img');
  for(var i=0; i<images.length; i++){
    images[i].addEventListener('dragstart', dragged, false);
    images[i].addEventListener('dragend', ending, false);
  }

  drop=document.getElementById('canvas');
  canvas=drop.getContext('2d');

  drop.addEventListener('dragenter', function(e){
    e.preventDefault(); }, false);
  drop.addEventListener('dragover', function(e){
    e.preventDefault(); }, false);
```

продолжение ⇩

Листинг 8.8 *(продолжение)*

```
  drop.addEventListener('drop', dropped, false);
  }
  function ending(e){
    elem=e.target;
    elem.style.visibility='hidden';
  }
  function dragged(e){
    elem=e.target;
    e.dataTransfer.setData('Text', elem.getAttribute('id'));
    e.dataTransfer.setDragImage(e.target, 0, 0);
  }
  function dropped(e){
    e.preventDefault();
    var id=e.dataTransfer.getData('Text');
    var elem=document.getElementById(id);

    var posx=e.pageX-drop.offsetLeft;
    var posy=e.pageY-drop.offsetTop;

    canvas.drawImage(elem,posx,posy);
}
window.addEventListener('load', initiate, false);
```

Создав этот пример, мы подобрались совсем близко к приложениям из реального мира. Код в листинге 8.8 управляет тремя аспектами процесса. Когда изображение перетаскивают, вызывается функция `dragged()`. Она определяет, каким будет эскиз перетаскиваемого изображения (для чего применяется метод `setDragImage()`). Кроме того, в коде задается контекст, для того чтобы можно было работать с холстом. Отпущенное пользователем над зоной приема изображение визуализируется посредством метода `drawImage()`, которому передается ссылка на источник. Наконец, в функции `ending()` изображение на источнике скрывается.

В качестве уникального эскиза используем само изображение, которое перетаскивает пользователь. В этом отношении мы ничего не изменили, только установили позицию эскиза (0, 0) — это означает, что мы теперь точно знаем, где эскиз находится относительно указателя мыши. Эту информацию учитываем в функции `dropped()`. Применяя технику, уже знакомую по предыдущим главам, мы вычисляем точку на холсте, в которой пользователь отпустил элемент-источник, и визуализируем изображение точно в этом месте. Протестируйте этот пример в браузерах,

поддерживающих метод `setDragImage()` (например, Firefox 4), и вы увидите, что изображение на холсте появляется именно под эскизом. Таким образом, пользователю не составляет труда подобрать точное местоположение, в котором он отпустит изображение.

ВНИМАНИЕ

В коде из листинга 8.8, для того чтобы скрывать исходное изображение по завершении операции, используется событие dragend. Оно срабатывает на источнике, когда операция перетаскивания завершается — успешно или нет. В нашем примере изображение скрывается в обоих случаях. Для того чтобы скрывать изображение только в случае успешного перетаскивания, вам придется запрограммировать соответствующую обработку.

Файлы

Вероятно, самая интересная особенность API Drag and Drop (Перетаскивание) — это возможность работать с файлами. Этот API доступен не только изнутри документа — он интегрируется с системой, позволяя пользователям перетаскивать элементы из браузера в другие приложения и наоборот. Чаще всего возникает необходимость перетаскивать из внешних источников именно файлы.

Мы уже упоминали специальное свойство объекта `dataTransfer`, возвращающее массив со списком перетаскиваемых файлов. Эту информацию можно применять для построения сложных сценариев, помогающих пользователям работать с файлами или загружать их на сервер.

Листинг 8.9. Простой шаблон для перетаскивания файлов

```
<!DOCTYPE html>
<html lang="ru">
<head>
  <title>Перетаскивание</title>
  <link rel="stylesheet" href="dragdrop.css">
  <script src="dragdrop.js"></script>
</head>
<body>
```

продолжение ⇩

Листинг 8.9 *(продолжение)*

```
  <section id="dropbox">
    Перетаскивайте сюда файлы
  </section>
</body>
</html>
```

HTML-документ из листинга 8.9 всего лишь определяет зону приема. Файлы на нее нужно перетаскивать из внешних приложений (например, Проводника Windows). Для обработки данных в файлах предназначен следующий код.

Листинг 8.10. Обработка данных в файлах

```
function initiate(){
  drop=document.getElementById('dropbox');
  drop.addEventListener('dragenter', function(e){
    e.preventDefault(); }, false);
  drop.addEventListener('dragover', function(e){
    e.preventDefault(); }, false);
  drop.addEventListener('drop', dropped, false);
}
function dropped(e){
  e.preventDefault();
  var files=e.dataTransfer.files;
  var list='';
  for(var f=0;f<files.length;f++){
    list+='File: '+files[f].name+' '+files[f].size+'<br>';
  }
  drop.innerHTML=list;
}
window.addEventListener('load', initiate, false);
```

Информацию, возвращаемую свойством `files`, можно сохранить в переменной и затем считать в цикле `for`. В коде из листинга 8.10 мы всего лишь выводим на экран название и размер каждого файла, попавшего в зону приема. Для того чтобы применить эту информацию при построении более сложных приложений, нам придется обратиться к помощи других API и техник программирования. Об этом поговорим в следующих главах книги.

САМОСТОЯТЕЛЬНО

Создайте новые файлы с примерами кода из листингов 8.9 и 8.10, а затем откройте шаблон в своем браузере. Перетащите файлы из Проводника Windows или другой аналогичной программы на зону приема в шаблоне. Вы должны увидеть список названий и размеры всех выбранных файлов.

Краткий справочник. API Drag and Drop (Перетаскивание)

API Drag and Drop (Перетаскивание) предоставляет специальные события, методы и свойства, помогающие строить приложения с поддержкой перетаскивания элементов.

События

В этом API появилось семь новых событий:

- `dragstart`. Срабатывает на источнике, когда операция перетаскивания начинается;
- `drag`. Срабатывает на источнике во время выполнения операции перетаскивания;
- `dragend`. Срабатывает на источнике, когда операция перетаскивания завершается: либо перетаскиваемый элемент успешно попадает на зону приема, либо перетаскивание отменяется;
- `dragenter`. Срабатывает на целевом элементе, когда указатель мыши с перетаскиваемым элементом оказывается над ним. Данное событие всегда необходимо отменять методом `preventDefault()`;
- `dragover`. Срабатывает на целевом элементе, когда указатель мыши с перетаскиваемым элементом находится над ним. Данное событие всегда необходимо отменять методом `preventDefault()`;
- `drop`. Срабатывает на целевом элементе, когда пользователь отпускает на нем перетаскиваемый элемент. Данное событие всегда необходимо отменять методом `preventDefault()`;
- `dragleave`. Срабатывает на целевом элементе, когда указатель мыши с перетаскиваемым элементом покидает данную зону.

Методы

Далее приведен список наиболее важных методов, появившихся в данном API:

- `setData(type, data)`. Применяется для подготовки данных, которые будут отправлены при срабатывании события `dragstart`. В качестве атрибута `type` можно передавать название любого обычного типа данных (например, `text/plain` или `text/html`) или персонализированного типа данных;
- `getData(type)`. Возвращает данные указанного типа. Используется после того, как срабатывает событие `drop`;
- `clearData(type)`. Удаляет данные указанного типа;
- `setDragImage(element, x, y)`. Заменяет эскиз по умолчанию, создаваемый браузером, указанным изображением, а также устанавливает его позицию относительно указателя мыши.

Свойства

У объекта `dataTransfer`, в котором содержатся данные, передаваемые во время операции перетаскивания, также есть несколько полезных свойств:

- `types`. Возвращает массив, содержащий все типы, определенные в событии `dragstart`;
- `files`. Возвращает массив с информацией о перетаскиваемых файлах;
- `dropEffect`. Возвращает тип выбранной в данный момент операции. Его также можно использовать для смены выбранной операции. Возможные значения — `none`, `copy`, `link` и `move`;
- `effectAllowed`. Возвращает допустимые типы операций. Также с помощью него можно менять список допустимых операций. Возможные значения — `none`, `copy`, `copyLink`, `copyMove`, `link`, `linkMove`, `move`, `all` и `uninitialized`.

9 API геолокации

Определение своего местоположения

API Geolocation (Геолокация) разработан в качестве стандартного детекторного механизма браузеров, с помощью которого разработчики могут определять физическое местоположение пользователя. Раньше у нас был только один способ узнавать, где находится пользователь: строить большую базу данных IP-адресов и использовать на сервере пожирающие компьютерные ресурсы сценарии, обеспечивающие лишь приблизительную оценку местоположения (обычно с точностью до страны).

Этот API работает на базе таких новых систем, как сетевая триангуляция и GPS, и возвращает точное местоположение устройства, на котором выполняется приложение. Такая информация позволяет создавать приложения, адаптирующиеся к конкретным требованиям пользователя и автоматически возвращающие локализованную информацию.

Для того чтобы можно было пользоваться возможностями API, добавлены три специальных метода:

- `getCurrentPosition(location, error, configuration)`. Применяется для одиночных запросов. Принимает три атрибута: функцию для обработки возвращенного значения местоположения, функцию для обработки возвращенных значений ошибки и объект для настройки способа получения информации. Для того чтобы метод работал правильно, обязательно указывать только первый атрибут, остальные можно опустить;

- `watchPosition(location, error, configuration)`. Похож на предыдущий, но в отличие от него не срабатывает разово, а запускает процесс слежения за местоположением. Принцип работы аналогичен использованию метода `setInterval()` из JavaScript: процесс автоматически повторяется на протяжении определенного времени в зависимости от конфигурации по умолчанию или значений атрибутов;
- `clearWatch(id)`. Метод `watchPosition()` возвращает значение, которое можно сохранить в переменной, а затем, когда потребуется остановить слежение, передать его в качестве идентификатора методу `clearWatch()`. Принцип аналогичен использованию метода `clearInterval()` для остановки процесса, запущенного с помощью `setInterval()`.

Метод getCurrentPosition(location). Синтаксис № 1

Как уже упоминалось, для того чтобы метод `getCurrentPosition()` работал правильно, ему достаточно передать только первый атрибут. Это должна быть функция обратного вызова, которая получит объект с именем `Position`, содержащий всю информацию, извлеченную геолокационными системами.

У объекта `Position` два атрибута:

- `coords`. Содержит набор значений, определяющих позицию устройства, а также другую важную информацию. Для доступа к значениям предназначены семь внутренних атрибутов: `latitude` (широта), `longitude` (долгота), `altitude` (высота в метрах), `accuracy` (точность в метрах), `altitudeAccuracy` (точность определения высоты в метрах), `heading` (направление в градусах) и `speed` (скорость в метрах в секунду);
- `timestamp`. Указывает время, когда была получена эта информация.

Этот объект передается функции обратного вызова, после чего можно обращаться к перечисленным значениям внутри этой функции. Давайте рассмотрим практический пример использования данного метода.

Листинг 9.1. HTML-документ для тестирования геолокации

```
<!DOCTYPE html>
<html lang="ru">
<head>
  <title>Геолокация</title>
```

```
  <script src="geolocation.js"></script>
</head>
<body>
  <section id="location">
    <button id="getlocation">Узнать мое местоположение</button>
  </section>
</body>
</html>
```

Код из листинга 9.1 будет служить шаблоном для проверки остальных примеров из этой главы. Это всего лишь элемент `<button>` внутри элемента `<section>`, который мы будем использовать для отображения информации, извлеченной геолокационной системой.

Листинг 9.2. Получение информации о местоположении

```
function initiate(){
  var get=document.getElementById('getlocation');
  get.addEventListener('click', getlocation, false);
}
function getlocation(){
  navigator.geolocation.getCurrentPosition(showinfo);
}
function showinfo(position){
  var location=document.getElementById('location');
  var data='';
  data+='Широта: '+position.coords.latitude+'<br>';
  data+='Долгота: '+position.coords.longitude+'<br>';
  data+='Точность: '+position.coords.accuracy+'mts.<br>';
  location.innerHTML=data;
}
window.addEventListener('load', initiate, false);
```

Внедрять возможности API геолокации очень просто: мы применяем метод `getCurrentPosition()` и создаем функцию, которая будет выводить на экран возвращенные значения. Метод `getCurrentPosition()` принадлежит объекту `geolocation`. Это новый объект, входящий в объект `navigator` — объект JavaScript, который ранее предназначался для возвращения информации о браузере и системе. Таким образом, для того чтобы обратиться к методу `getCurrentPosition()`, нужно воспользоваться следующим синтаксисом: `navigator.geolocation.getCurrentPosition(function)`, где `function` — это пользовательская

функция, роль которой — получить объект `Position` и обработать возвращенную информацию.

В коде из листинга 9.2 мы назвали такую функцию `showinfo`. Когда происходит вызов метода `getCurrentPosition()`, новый объект `Position` со всей необходимой информацией создается и отправляется функции `showinfo()`. Мы ссылаемся на этот объект внутри функции через переменную `position`, а затем с помощью той же переменной выводим сведения на экран.

У объекта `Position` есть два важных атрибута: `coords` и `timestamp`. В нашем примере для доступа к необходимой информации (широта, долгота и точность) мы используем только атрибут `coords`. Эти значения сохраняются в переменной `data`, а затем выводятся на экран в качестве нового содержимого элемента `location`.

САМОСТОЯТЕЛЬНО

Создайте файлы с фрагментами кода из листингов 9.1 и 9.2, загрузите их на свой сервер, а затем откройте HTML-документ в браузере. Когда вы щелкнете на кнопке, браузер запросит разрешение на активацию геолокационной системы для данного приложения. Если вы разрешите приложению обращаться к этой информации, на экране появится информация о значениях широты, долготы и точности определения координат.

Метод getCurrentPosition(location, error). Синтаксис № 2

Что же происходит, когда пользователь не разрешает браузеру обращаться к географическим данным? Добавив второй атрибут (еще одну функцию), мы можем захватывать возникающие ошибки. Одной из них будет ошибка, связанная с невозможностью доступа.

Помимо объекта `Position` метод `getCurrentPosition()` в случае обнаружения ошибки возвращает объект `PositionError`. Этот объект передается второму атрибуту метода `getCurrentPosition()`, а для считывания значения и описания ошибки предлагает два атрибута, `error` и `message`. Трем возможным ошибкам соответствуют перечисленные далее константы:

- `PERMISSION_DENIED`, значение 1. Эта ошибка возвращается, когда пользователь запрещает API геолокации доступ к информации о своем географическом положении;
- `POSITION_UNAVAILABLE`, значение 2. Эта ошибка возвращается, когда местоположение устройства определить невозможно;
- `TIMEOUT`, значение 3. Эта ошибка возвращается, когда не удается определить местоположение в течение времени, объявленного в конфигурации.

Листинг 9.3. Вывод сообщений об ошибках

```
function initiate(){
  var get=document.getElementById('getlocation');
  get.addEventListener('click', getlocation, false);
}
function getlocation(){
  navigator.geolocation.getCurrentPosition(showinfo, showerror);
}
function showinfo(position){
  var location=document.getElementById('location');
  var data='';
  data+='Широта: '+position.coords.latitude+'<br>';
  data+='Долгота: '+position.coords.longitude+'<br>';
  data+='Точность: '+position.coords.accuracy+'mts.<br>';
  location.innerHTML=data;
}
function showerror(error){
  alert('Ошибка: '+error.code+' '+error.message);
}
window.addEventListener('load', initiate, false);
```

Сообщения об ошибках предназначены для внутреннего использования. Целью их добавления было создание механизма извещения приложения о текущей ситуации, для того чтобы оно могло продолжать работу соответствующим образом. В коде из листинга 9.3 мы добавили в метод `getCurrentPosition()` второй параметр (еще одну функцию обратного вызова) и создали функцию `showerror()`, которая выводит на экран информацию из атрибутов `code` и `message`. Значением атрибута `code` может быть только число от 0 до 3, соответствующее коду ошибки (перечислены ранее).

Для того чтобы сделать пример более наглядным, мы вывели информацию на экран с помощью метода `alert()`, однако в реальных приложениях ответы API геолокации следует обрабатывать по возможности незаметно для пользователя, не извещая его об ошибках.

Объект `PositionError` отправляется функции `showerror()`, его представляет переменная `error`. Мы могли бы также отдельно выполнять проверку каждой ошибки (например, `error.PERMISSION_DENIED`) и уведомлять пользователя только в случае возникновения какого-то конкретного состояния.

Метод getCurrentPosition(location, error, configuration). Синтаксис № 3

Третий возможный параметр метода `getCurrentPosition()` — это объект, включающий в себя до трех атрибутов:

- `enableHighAccuracy`. Это булев атрибут, извещающий систему о том, что вам требуется максимально точная информация о местоположении. Для того чтобы вернуть точные координаты устройства, браузер попытается получить географическую информацию через такие системы, как GPS. Однако эти системы расходуют большое количество ресурсов устройства, поэтому их использование необходимо ограничивать. По этой причине значение по умолчанию данного атрибута равно `false`;
- `timeout`. Задает максимальную продолжительность интервала времени, отведенного на выполнение операции. Если информацию не удается получить за указанное время, возвращается ошибка `TIMEOUT`. Значение следует указывать в миллисекундах;
- `maximumAge`. Координаты предыдущих местоположений кэшируются в системе. Если в нашем приложении допускается использование последнего сохраненного набора данных вместо извлечения новой информации о местоположении (во избежание излишней траты ресурсов или для обеспечения быстрой реакции), то с помощью этого атрибута можно задать лимит возраста информации. Если последнее кэшированное местоположение «старше» указанного возраста, то выполняется запрос нового местоположения. Значение задается в миллисекундах.

Код из листинга 9.4 пытается получить самую точную информацию о местоположении устройства за время, не превышающее 10 с, при условии, что в кэше нет географических данных, полученных менее 60 с назад (если есть, то именно они возвращаются в объекте `Position`).

Листинг 9.4. Конфигурация системы

```
function initiate(){
  var get=document.getElementById('getlocation');
  get.addEventListener('click', getlocation, false);
}
function getlocation(){
  var geoconfig={
    enableHighAccuracy: true,
    timeout: 10000,
    maximumAge: 60000
  };
  navigator.geolocation.getCurrentPosition(showinfo, showerror,
            geoconfig);
}
function showinfo(position){
  var location=document.getElementById('location');
  var data='';
  data+='Широта: '+position.coords.latitude+'<br>';
  data+='Долгота: '+position.coords.longitude+'<br>';
  data+='Точность: '+position.coords.accuracy+'mts.<br>';
  location.innerHTML=data;
}
function showerror(error){
  alert('Ошибка: '+error.code+' '+error.message);
}
window.addEventListener('load', initiate, false);
```

ПОВТОРЯЕМ ОСНОВЫ

В JavaScript предусмотрено несколько способов создания объектов. Для того чтобы сделать пример понятнее, мы сначала создаем объект, сохраняем его в переменной geoconfig, а затем используем эту ссылку в методе getCurrentPosition(). Однако можно также напрямую добавить объект в метод в виде атрибута. В небольших приложениях вполне допустимо отказываться от использования объектов, однако в более сложных кодах это вряд ли возможно. Для того чтобы больше узнать об объектах, зайдите на наш веб-сайт и изучите ссылки для этой главы.

Сначала мы создали объект с конфигурационными значениями, а затем сослались на него из метода `getCurrentPosition()`. В остальном код не поменялся. Функция `showinfo()` выводит информацию на экран независимо от того, каким образом она была извлечена (из кэша или путем нового системного запроса).

В последнем примере кода мы начинаем раскрывать истинное предназначение API Geolocation (Геолокация). Самые эффективные и полезные возможности ориентированы на мобильные устройства. Например, когда значение атрибута `enableHighAccuracy` равно `true`, браузер обращается к таким системам, как GPS, чтобы получить самые точные географические данные. Методы `watchPosition()` и `clearWatch()`, с которыми мы познакомимся далее, работают с постоянно меняющимися географическими данными, а такая ситуация возможна, разумеется, только при обращении к приложению с мобильного (и перемещающегося в пространстве) устройства. Этот аспект вскрывает две важные проблемы. Во-первых, большинство наших примеров кода следует тестировать на мобильном устройстве, чтобы понимать, как они будут работать в реальных условиях. А во-вторых, к использованию данного API следует подходить очень ответственно. GPS и другие системы определения местоположения потребляют большое количество ресурсов, и, если не проявлять достаточную осмотрительность, аккумулятор устройства быстро разрядится. Что касается первой из них, есть альтернатива: зайдите на веб-сайт http://dev.w3.org/geo/api/test-suite/ и найдите комплект тестирования для API Geolocation (Геолокация). В отношении же второй можно дать небольшой совет: присваивайте атрибуту `enableHighAccuracy` значение `true` лишь в случае крайней необходимости и не используйте его только потому, что он доступен.

Метод watchPosition(location, error, configuration)

Аналогично `getCurrentPosition()`, метод `watchPosition()` принимает три атрибута и выполняет ту же задачу, а именно: устанавливает местоположение устройства, с которого пользователь обратился к приложению. Единственное различие заключается в том, что первый метод выполняет операцию один раз, в то время как `watchPosition()` автоматически возвращает новые данные при каждом изменении местоположения. Этот метод постоянно следит за координатами и при появлении новых данных отсылает информацию функции обратного вызова. Отменить этот процесс можно методом `clearWatch()`.

В следующем примере мы внедряем метод `watchPosition()` на основе предыдущих примеров кода.

Выполнив этот код на настольном компьютере, вы не заметите ничего необычного, но если запустить его на мобильном устройстве, то новая информация будет отображаться каждый раз, когда местоположение устройства изменится. Атрибут `maximumAge` определяет, как часто географическая информация отсылается функции `showinfo()`. Если новые географические данные извлекаются через 60 с (60 000 мс) после предыдущей попытки, то они выводятся на экран, в противном случае функция `showinfo()` не вызывается.

Листинг 9.5. Тестируем метод watchPosition()

```
function initiate(){
  var get=document.getElementById('getlocation');
  get.addEventListener('click', getlocation, false);
}
function getlocation(){
  var geoconfig={
    enableHighAccuracy: true,
    maximumAge: 60000
  };
  control=navigator.geolocation.watchPosition(showinfo, showerror,
          geoconfig);
}
function showinfo(position){
  var location=document.getElementById('location');
  var data='';
  data+='Широта: '+position.coords.latitude+'<br>';
  data+='Долгота: '+position.coords.longitude+'<br>';
  data+='Точность: '+position.coords.accuracy+'mts.<br>';
  location.innerHTML=data;
}
function showerror(error){
  alert('Ошибка: '+error.code+' '+error.message);
}
window.addEventListener('load', initiate, false);
```

Обратите внимание на то, что значение, возвращаемое методом `watchPosition()`, сохраняется в переменной `control`. Эта переменная служит своеобразным идентификатором данной операции. Если позднее мы захотим отменить обработку данного метода, то для этого нужно будет всего

лишь выполнить запрос `clearWatch(control)`, и метод `watchPosition()` прекратит обновлять информацию.

Если запустить этот код на настольном компьютере, то метод `watchPosition()` будет работать абсолютно так же, как `getCurrentPosition()`: никакая информация обновляться не будет. Обращение к функции обратного вызова происходит только в одном случае — когда местоположение меняется.

Практические варианты использования с Google Maps

Пока что мы выводили географические данные на экран в том виде, в котором получали их от объекта. Однако большинству обычных людей они не дают никакой полезной информации. Я бы не смог немедленно назвать широту и долготу моего текущего местоположения, не говоря уж о том, чтобы идентифицировать произвольную точку земного шара по этим значениям. У нас есть две альтернативы: обрабатывать возвращаемые данные внутри приложения, вычисляя позицию, расстояние и другие переменные, которые позволят предложить пользователям полезные результаты (такие, как магазины или рестораны по соседству), или показывать информацию, полученную с помощью API геолокации, в более понятном виде. А что может быть понятнее, чем указание нужной точки на карте?

Раньше в этой книге мы уже упоминали API Google Maps. Это внешний API JavaScript авторства Google, который никак не связан с HTML5, но широко используется на множестве современных веб-сайтов и приложений. Он предлагает различные способы взаимодействия с интерактивными картами и даже реальными видами некоторых мест посредством технологии StreetView.

Мы продемонстрируем простой пример использования одной из составляющих этого API под названием API Static Maps (Статические карты). Для того чтобы воспользоваться преимуществами этого API, нужно всего лишь сформировать URL-адрес с информацией о местоположении, и в ответ будет возвращено изображение выбранной области на карте.

Листинг 9.6. Представление географических данных на карте

```
function initiate(){
  var get=document.getElementById('getlocation');
  get.addEventListener('click', getlocation, false);
}
```

```
function getlocation(){
  navigator.geolocation.getCurrentPosition(showinfo, showerror);
}
function showinfo(position){
  var location=document.getElementById('location');
  var mapurl='http://maps.google.com/maps/api/
      staticmap?center='+position.coords.latitude+','+
      position.coords.longitude+'&zoom=12&size=400x400&sensor=
      false&markers='+position.coords.latitude+','+
      position.coords.longitude;
  location.innerHTML='<img src="'+mapurl+'">';
}
function showerror(error){
  alert('Ошибка: '+error.code+' '+error.message);
}
window.addEventListener('load', initiate, false);
```

Этот код чрезвычайно прост. Мы добавили метод `getCurrentPosition()` и отправили информацию функции `showinfo()` так же, как делали это в предыдущих примерах. Единственное различие — на этот раз внутри функции мы вставили значение объекта `Position` в URL-адрес в домене Google.com. Полученный адрес затем используется в качестве источника в элементе `<img>`, благодаря чему фрагмент карты появляется на экране.

САМОСТОЯТЕЛЬНО

Протестируйте код из листинга 9.6 в своем браузере с использованием шаблона из листинга 9.1. Зайдите на веб-страницу API Google Maps и изучите другие доступные варианты. Адрес страницы http://code.google.com/apis/maps/. Поменяйте значения атрибутов zoom и size в URL-адресе, чтобы изменить внешний вид карты, возвращаемой API.

Краткий справочник. API Geolocation (Геолокация)

Возможность определять географическое местоположение пользователя играет огромную роль в современных веб-приложениях. Популярность

мобильных устройств обусловливает почву для создания приложений, эффективно использующих эту информацию.

Методы

В API Geolocation (Геолокация) предусмотрено три метода для получения географической информации от систем устройства:

- `getCurrentPosition(location, error, configuration)`. При каждом вызове возвращает географическую информацию. Первый атрибут — это функция обратного вызова, предназначенная для получения информации, второй — еще одна функция обратного вызова для обработки ошибок, а третий — объект, содержащий конфигурационные значения (см. описание объекта конфигурации далее);
- `watchPosition(location, error, configuration)`. Автоматически возвращает географическую информацию при каждом изменении местоположения. Первый атрибут — это функция обратного вызова, предназначенная для получения информации, второй — еще одна функция обратного вызова для обработки ошибок, а третий — объект, содержащий конфигурационные значения (см. описание объекта конфигурации далее);
- `clearWatch(id)`. Отменяет процесс, запущенный методом `watchPosition()`. Атрибут `id` — это идентификатор, возвращаемый при вызове метода `watchPosition()`.

Объекты

Методы `getCurrentPosition()` и `watchPosition()` создают два объекта для передачи информации, полученной от геолокационной системы, а также статуса операции:

- `Position`. Создается для того, чтобы поместить в него информацию о местоположении. У него два атрибута: `coords` и `timestamp`:
  - `coords`. Имеет несколько внутренних атрибутов, возвращающих информацию о местоположении: `latitude` (широта), `longitude` (долгота), `altitude` (высота, в метрах), `accuracy` (точность, в метрах), `altitudeAccuracy` (точность определения высоты, в метрах), `heading` (направление, в градусах) и `speed` (скорость, в метрах в секунду);
  - `timestamp`. Возвращает время определения местоположения;

- `PositionError`. Создается, когда происходит ошибка. У него есть два обычных атрибута, содержащих код ошибки и сообщение об ошибке:
  - `message`. Возвращает сообщение с описанием обнаруженной ошибки;
  - `error`. Содержит код обнаруженной ошибки.

  Также он предлагает три специальных значения для идентификации отдельных ошибок:
  - `PERMISSION_DENIED`, значение 1 в атрибуте `error`. Равно `true`, когда пользователь запрещает API Geolocation (Геолокация) обращаться к географическим данным своего устройства;
  - `POSITION_UNAVAILABLE`, значение 2 в атрибуте `error`. Равно `true`, когда невозможно определить местоположение устройства;
  - `TIMEOUT`, значение 3 в атрибуте `error`. Равно `true`, если местоположение не удалось определить в течение времени, объявленного в конфигурации.

Методам `getCurrentPosition()` и `watchPosition()` для конфигурирования процесса требуется объект `Configuration`. Он предоставляет конфигурационные значения для методов `getCurrentPosition()` и `watchPosition()`.

- `enableHighAccuracy`. Это один из возможных атрибутов объекта `Configuration`. Если его значение равно `true`, то браузер должен вернуть максимально точную географическую информацию;
- `timeout`. Это один из возможных атрибутов объекта `Configuration`. Указывает максимальную величину интервала, в течение которого операция должна быть выполнена;
- `maximumAge`. Это один из возможных атрибутов объекта `Configuration`. Указывает, как долго последнее кэшированное местоположение считается текущим.

10 API веб-хранилища

Две системы хранения

Первоначально в Сети на первое место ставилось отображение информации — всего лишь представление ее на экране. Обработкой данных занялись позже: сначала с помощью серверных приложений, а затем также с использованием малопроизводительных сценариев и встраиваемых модулей на клиентской стороне. Тем не менее суть процессов оставалась неизменной: данные подготавливались на сервере и только после этого отображались на компьютере пользователя. Тяжелая работа почти всегда выполнялась на серверной стороне, так как система не располагала возможностями для того, чтобы задействовать ресурсы клиентского компьютера.

Благодаря спецификации HTML5 ситуация уравновесилась. Ориентируясь на особенности мобильных устройств, появление облачных вычислений и необходимость стандартизировать технологии и инновации, внедряемые в течение многих лет посредством встраиваемых модулей, разработчики HTML5 объединили в одной спецификации всю функциональность, позволяющую выполнять полноценные приложения на пользовательских компьютерах даже в отсутствие сетевого подключения.

Одна из самых востребованных возможностей любого приложения — хранение данных, однако у веб-приложений не было эффективного

механизма, позволяющего записывать данные в постоянное хранилище и по необходимости обращаться к ним. Для хранения небольших порций информации на клиентской стороне использовались файлы cookie, однако их природа такова, что они поддерживают запись лишь коротких строк и применять их удобно далеко не во всех ситуациях.

API Web Storage (веб-хранилище) — это, по сути, следующая ступень развития файлов cookie. Этот API позволяет записывать данные на жесткий диск пользователя и обращаться к ним так же, как это делается в настольных приложениях. Процессы сохранения и извлечения данных, поддерживаемые этим API, применимы в двух ситуациях: когда информация должна быть доступна только в течение одного сеанса и когда ее необходимо сохранить надолго — до тех пор, пока пользователь сам не удалит ее. Таким образом, для упрощения жизни разработчиков этот API разделили на две части — `sessionStorage` и `localStorage`:

- `sessionStorage`. Это механизм хранения, удерживающий данные только на протяжении сеанса одной страницы. В действительности, в отличие от настоящих сеансов, здесь речь идет об информации, доступ к которой есть только у одного окна или вкладки браузера, причем как только окно или вкладка закрывается, эта информация удаляется. В спецификации используется термин «сеанс», поскольку информация сохраняется даже при обновлении содержимого окна, а также в случае, когда пользователь открывает новую страницу того же веб-сайта;
- `localStorage`. Этот механизм работает аналогично системам хранения для настольных приложений. Данные записываются навсегда, и приложение, сохранившее их, может обращаться к этой информации в любой момент.

Оба механизма работают через один и тот же интерфейс и предлагают одинаковые методы и свойства. Кроме того, оба механизма завязаны на источник, то есть любая информация в хранилище доступна только через тот веб-сайт, который ее создал. У каждого веб-сайта есть собственное пространство хранилища, которое в зависимости от используемого механизма очищается при закрытии окна или не очищается никогда.

В этом API существует четкое разграничение временных и постоянных данных, благодаря чему становится намного проще конструировать как небольшие приложения, требующие временного хранения всего нескольких строк (например, содержимого корзины в интернет-магазине), так и объемные и сложные приложения, в которых нередко возникает необходимость сохранять на неопределенное время целые документы.

ВНИМАНИЕ

Большинство браузеров корректно работают с данным API только в том случае, когда источником является настоящий сервер. Мы рекомендуем для проверки следующих примеров кода загружать соответствующие файлы на сервер.

sessionStorage

Первая составляющая API, `sessionStorage`, заменяет собой сеансовые файлы cookie. Файлы cookie, так же как и `sessionStorage`, сохраняют данные ограниченное время. Однако если файлы cookie привязываются к браузеру, то объекты `sessionStorage` привязываются к конкретным окнам или вкладкам. Это означает, что к файлам cookie, созданным для определенного сеанса, можно обращаться до тех пор, пока хотя бы одно окно браузера открыто, но данные, создаваемые `sessionStorage`, доступны только до момента закрытия окна (и обращаться к ним может только конкретное окно или вкладка).

Реализация хранения данных

Поскольку обе системы, `sessionStorage` и `localStorage`, работают с одним и тем же интерфейсом, для тестирования примеров кода и экспериментов с API нам будет достаточно одного HTML-документа и простой формы.

Листинг 10.1. Шаблон для API хранения

```
<!DOCTYPE html>
<html lang="ru">
<head>
  <title>API веб-хранилища</title>
  <link rel="stylesheet" href="storage.css">
  <script src="storage.js"></script>
</head>
<body>
  <section id="formbox">
    <form name="form">
      <p>Ключевое слово:<br><input type="text"
         name="keyword" id="keyword"></p>
      <p>Значение:<br><textarea name="text"
         id="text"></textarea></p>
```

```
      <p><input type="button" name="save" id="save"
value="Сохранить"></p>
    </form>
  </section>
  <section id="databox">
    Информация недоступна
  </section>
</body>
</html>
```

Мы также создали простой набор стилей для оформления страницы, благодаря которым область формы будет визуально отделяться от поля для отображения данных.

Листинг 10.2. Стили для нашего шаблона

```
#formbox{
  float: left;
  padding: 20px;
  border: 1px solid #999999;
}
#databox{
  float: left;
  width: 400px;
  margin-left: 20px;
  padding: 20px;
  border: 1px solid #999999;
}
#keyword, #text{
  width: 200px;
} #databox > div{
  padding: 5px;
  border-bottom: 1px solid #999999;
}
```

САМОСТОЯТЕЛЬНО

Создайте HTML-файл с кодом из листинга 10.1 и CSS-файл с именем storage.css, содержащий стили из листинга 10.2. Также вам понадобится файл под названием storage.js для сохранения и тестирования примеров JavaScript-кода из следующих листингов.

Создание данных

И `sessionStorage`, и `localStorage` сохраняют данные в форме отдельных элементов. Элементом считается пара из ключевого слова и значения, при этом каждое значение перед помещением в хранилище можно конвертировать в строку. Вы можете представлять себе элементы как переменные, обладающие именем и значением, которые можно создавать, модифицировать и удалять.

Для создания и извлечения элементов из пространства хранилища предназначены два новых метода, применяемые только в этом API:

- `setItem(key, value)`. Для того чтобы создать элемент, нужно вызвать этот метод. Элемент создается с ключевым словом и значением, переданными в качестве аргументов метода. Если в хранилище уже есть элемент с таким ключевым словом, то ему присваивается новое значение, таким образом, данный метод позволяет модифицировать данные;
- `getItem(key)`. Для извлечения значения из хранилища необходимо вызвать этот метод, передав ему ключевое слово нужного элемента. Ключевое слово должно совпадать с тем, которое было объявлено при создании элемента методом `setItem()`.

Листинг 10.3. Сохранение и извлечение данных

```
function initiate(){
  var button=document.getElementById('save');
  button.addEventListener('click', newitem, false);
}
function newitem(){
  var keyword=document.getElementById('keyword').value;
  var value=document.getElementById('text').value;
  sessionStorage.setItem(keyword,value);


  show(keyword);
}
function show(keyword){
  var databox=document.getElementById('databox');
  var value=sessionStorage.getItem(keyword);
  databox.innerHTML='<div>'+keyword+' - '+value+'</div>';
}
window.addEventListener('load', initiate, false);
```

Процесс чрезвычайно прост. Эти методы входят в объект `sessionStorage`, а синтаксис их вызова всегда одинаков: `sessionStorage.setItem()`. В коде из листинга 10.3 функция `newitem()` выполняется каждый раз, когда пользователь щелкает на кнопке формы. Эта функция создает элемент и добавляет в него информацию, полученную из формы, а затем вызывает функцию `show()`. Функция `show()`, в свою очередь, извлекает элемент из хранилища по ключевому слову, используя метод `getItem()`, а затем выводит его содержимое на экран.

Помимо этих методов, API хранения также предоставляет упрощенный способ создания и извлечения элементов из пространства хранилища, в котором ключевое слово элемента используется как свойство. Для этого способа предусмотрены два варианта синтаксиса в зависимости от типа информации, на базе которой создается элемент. Можно переменную, соответствующую ключевому слову, заключить в квадратные скобки (например, `sessionStorage[ключевое_слово]=значение`), а можно передать строку в качестве имени свойства (например, `sessionStorage.myitem=значение`).

Листинг 10.4. Работа с элементами упрощенным способом

```
function initiate(){
  var button=document.getElementById('save');
  button.addEventListener('click', newitem, false);
}
function newitem(){
  var keyword=document.getElementById('keyword').value;
  var value=document.getElementById('text').value;
  sessionStorage[keyword]=value;

  show(keyword);
}
function show(keyword){
  var databox=document.getElementById('databox');
  var value=sessionStorage[keyword];
  databox.innerHTML='<div>'+keyword+' - '+value+'</div>';
}
window.addEventListener('load', initiate, false);
```

Считывание данных

В предыдущем примере мы всего лишь извлекали последний помещенный в хранилище элемент. Теперь улучшим код и сделаем его более

практичным, воспользовавшись дополнительными методами и свойствами API, позволяющими манипулировать данными:

- `length`. Возвращает число элементов, помещенных в хранилище данным приложением. Оно работает точно так же, как обычное свойство `length` для массивов из JavaScript, и его удобно использовать для последовательного считывания элементов;
- `key(index)`. Элементы записываются в хранилище последовательно, и им автоматически присваиваются порядковые номера, начиная с 0. С помощью данного метода можно извлечь определенный элемент или даже всю информацию, содержащуюся в хранилище, если пройтись по нему в цикле.

Листинг 10.5. Перечисление элементов

```
function initiate(){
  var button=document.getElementById('save');
  button.addEventListener('click', newitem, false);
  show();
}
function newitem(){
  var keyword=document.getElementById('keyword').value;
  var value=document.getElementById('text').value;

  sessionStorage.setItem(keyword,value);
  show();
  document.getElementById('keyword').value='';
  document.getElementById('text').value='';
}
function show(){
  var databox=document.getElementById('databox');
  databox.innerHTML='';
  for(var f=0;f<sessionStorage.length;f++){
    var keyword=sessionStorage.key(f);
    var value=sessionStorage.getItem(keyword);
    databox.innerHTML+='<div>'+keyword+' - '+value+'</div>';
  }
}
window.addEventListener('load', initiate, false);
```

Задача кода из листинга 10.5 — вывести полный список элементов из хранилища в правом поле на экране. Мы усовершенствовали функцию `show()`, применив свойство `length` и метод `key()`. Для этого создали

цикл `for`, начинающийся с 0 и завершающийся порядковым номером последнего элемента из хранилища. Внутри цикла мы с помощью метода `key()` извлекаем ключевое слово для каждой позиции. Например, если в позиции 0 пространства хранилища находится элемент с ключевым словом «myitem», то код `sessionStorage.key(0)` возвращает значение «myitem». Вызывая этот метод в цикле, мы проходим по всему списку и выводим на экран ключевые слова и значения всех элементов из хранилища.

Функцию `show()` мы вызываем из функции `initiate()`. Таким образом, она выводит список элементов из хранилища на экран сразу же, как только вы запускаете приложение.

САМОСТОЯТЕЛЬНО

Вы можете воспользоваться функциональностью API Forms (Формы), о котором говорилось в главе 6, и реализовать проверку содержимого полей формы ввода, для того чтобы не допустить отправки на сервер неправильно заполненных или пустых элементов.

Удаление данных

Элементы в хранилище можно создавать, считывать и, разумеется, удалять из хранилища. Для выполнения последней задачи предназначены два метода:

- `removeItem(key)`. Удаляет один элемент. Для идентификации элемента методу нужно передать ключевое слово, которое использовалось в методе `setItem()` при создании элемента;
- `clear()`. Очищает пространство хранилища. Удаляются все находящиеся в нем элементы.

Листинг 10.6. Удаление элементов

```
function initiate(){
  var button=document.getElementById('save');
  button.addEventListener('click', newitem, false);
  show();
}
function newitem(){
```

продолжение ⇗

Листинг 10.6 *(продолжение)*

```
  var keyword=document.getElementById('keyword').value;
  var value=document.getElementById('text').value;

  sessionStorage.setItem(keyword,value);
  show();
  document.getElementById('keyword').value='';
  document.getElementById('text').value='';
}
function show(){
  var databox=document.getElementById('databox');
  databox.innerHTML='<div><button onclick="removeAll()">удалить
все</button></div>';
  for(var f=0;f<sessionStorage.length;f++){
    var keyword=sessionStorage.key(f);
    var value=sessionStorage.getItem(keyword);
    databox.innerHTML+='<div>'+keyword+' - '+
    value+'<br><button onclick="remove(\''+keyword+'\')
  ">удалить</button></div>';
  }
}
function remove(keyword){
  if(confirm('Вы уверены?')){
    sessionStorage.removeItem(keyword);
    show();
  }
}
function removeAll(){
  if(confirm('Вы уверены?')){
    sessionStorage.clear();
    show();
}
}
window.addEventListener('load', initiate, false);
```

Функции `initiate()` и `newitem()` в листинге 10.6 ничем не отличаются от аналогичных функций в предыдущих примерах кода. Изменилась только функция `show()`: теперь она включает в себя обработчик события `onclick`, который вызывает функции, удаляющие отдельный элемент или все элементы хранилища. Список элементов строится точно так же, как раньше, но на этот раз к каждому элементу добавляется кнопка Удалить. Кроме того, вверху списка создается еще одна кнопка, позволяющая полностью очистить хранилище.

За удаление выбранного элемента и полную очистку хранилища отвечают функции `remove()` и `removeAll()` соответственно. Каждая из этих функций в конце вызывает функцию `show()` для обновления списка элементов на экране.

САМОСТОЯТЕЛЬНО

Используя код из листинга 10.6, вы можете протестировать обработку информации объектом sessionStorage. Откройте шаблон из листинга 10.1 в своем браузере, создайте несколько элементов, а затем снова откройте тот же шаблон, но в другом окне. Содержимое окон будет различаться: в первом окне сохранятся созданные вами элементы, а пространство хранилища для второго окна будет пустым. В отличие от других систем (таких, как файлы cookie), при использовании объекта sessionStorage каждое окно считается независимым экземпляром приложения, и информация о сеансе никогда не выходит за пределы одного окна.

Система `sessionStorage` сохраняет данные, созданные в окне, только на протяжении одного сеанса, то есть до закрытия окна. Это полезно, например, для управления содержимым корзины в интернет-магазине, а также для построения любых других приложений, требующих сохранения данных на короткое время.

Объект localStorage

Надежная система для хранения данных на протяжении сеанса окна чрезвычайно полезна во многих ситуациях, но, если вы попытаетесь имитировать в Сети мощные настольные приложения, одного лишь временного хранилища данных вам будет недостаточно.

Для решения этой проблемы API хранилища предоставляет вторую систему, которая резервирует пространство хранилища для каждого источника и сохраняет информацию навсегда. Благодаря `localStorage` мы наконец получили возможность сохранять большие объемы данных. При этом решение о том, требуется ли еще эта информация или ее можно удалить, может принимать сам пользователь.

Эта система использует тот же интерфейс, что и `sessionStorage`, поэтому для `localStorage` можно использовать те же методы и свойства, которые

мы изучили ранее. Для подготовки кода придется внести единственное изменение: заменить префикс `session` префиксом `local`.

Листинг 10.7. Работа с объектом localStorage

```
function initiate(){
  var button=document.getElementById('save');
  button.addEventListener('click', newitem, false);
  show();
}
function newitem(){
  var keyword=document.getElementById('keyword').value;
  var value=document.getElementById('text').value;

  localStorage.setItem(keyword,value);
  show();
  document.getElementById('keyword').value='';
  document.getElementById('text').value='';
}
function show(){
  var databox=document.getElementById('databox');
  databox.innerHTML='';
  for(var f=0;f<localStorage.length;f++){
    var keyword=localStorage.key(f);
    var value=localStorage.getItem(keyword);
    databox.innerHTML+='<div>'+keyword+' - '+value+'</div>';
  }
}
window.addEventListener('load', initiate, false);
```

САМОСТОЯТЕЛЬНО

Используя шаблон из листинга 10.1, протестируйте код из листинга 10.7. Этот код создает новый элемент с информацией, которую вы вводите в форме, и автоматически перечисляет все элементы из пространства хранилища, зарезервированного для данного приложения. Закройте браузер, а затем снова откройте HTML-файл. Вы все так же сможете видеть список элементов из хранилища.

В листинге 10.7 мы всего лишь взяли предыдущий пример кода и поменяли название объекта `sessionStorage` на `localStorage`. Теперь все

созданные вами элементы будут сохраняться навсегда, и вы сможете обращаться к ним из других окон и даже после того, как браузер будет закрыт и снова запущен.

Событие storage

Объект `localStorage` предоставляет информацию всем окнам, в которых выполняется одно и то же приложение, из-за чего возникают сразу две проблемы: возможность обмена сообщениями между окнами и обновления информации в окне, которое в данный момент неактивно. Для решения обеих проблем в спецификацию добавили событие `storage`. Оно срабатывает в окне каждый раз, когда в пространстве хранилища происходит изменение. Его можно использовать для информирования всех окон, где открыто одно и то же приложение, об изменении данных в хранилище. В этом случае отреагировать соответствующим образом смогут все окна.

Листинг 10.8. Прослушивание события storage и автоматическое обновление списка элементов

```
function initiate(){
  var button=document.getElementById('save');
  button.addEventListener('click', newitem, false);
  window.addEventListener("storage", show, false);

  show();
}
function newitem(){
  var keyword=document.getElementById('keyword').value;
  var value=document.getElementById('text').value;

  localStorage.setItem(keyword,value);
  show();
  document.getElementById('keyword').value='';
  document.getElementById('text').value='';
}
function show(){
  var databox=document.getElementById('databox');
  databox.innerHTML='';
  for(var f=0;f<localStorage.length;f++){
    var keyword=localStorage.key(f);
```

продолжение ⇗

Листинг 10.8 *(продолжение)*

```
    var value=localStorage.getItem(keyword);
    databox.innerHTML+='<div>'+keyword+' - '+value+'</div>';
  }
}
window.addEventListener('load', initiate, false);
```

Для того чтобы вызывать функцию `show()` каждый раз, когда элемент в хранилище создается, модифицируется или удаляется из хранилища, нам понадобилось всего лишь добавить прослушиватель события `storage` в функции `initiate()`. Теперь, если что-нибудь поменяется в одном окне, изменение автоматически отразится в остальных окнах, где выполняется то же приложение.

Пространство хранения

Информация, которая добавляется в хранилище с помощью объекта `localStorage`, сохраняется навсегда, если только пользователь не принимает решение избавиться от нее. Это означает, что при каждом запуске приложения физическое пространство жесткого диска, занимаемое его данными, может значительно увеличиваться. Сейчас спецификация HTML5 рекомендует разработчикам браузеров резервировать минимум 5 Мбайт дискового пространства для каждого источника (веб-сайта или приложения). Это всего лишь рекомендация, которая в ближайшие годы может серьезно измениться. Некоторые браузеры запрашивают у пользователя разрешение на расширение пространства хранилища, когда в этом возникает необходимость, однако вы в любом случае должны помнить о существующем ограничении и учитывать его при разработке своих приложений.

ВНИМАНИЕ

Большинство браузеров правильно работают с данным API только в том случае, когда источником является настоящий сервер. Для тестирования примеров кода из этой главы мы рекомендуем вам загружать соответствующие файлы на свой сервер.

Краткий справочник. API Web Storage (Веб-хранилище)

Благодаря API хранения веб-приложения теперь могут записывать свои данные в локальное хранилище. На компьютере пользователя сохраняются элементы данных, состоящие из ключевого слова и значения, и такой вариант хранения обеспечивает не только быстрый доступ к информации, но и возможность работы в автономном режиме.

Типы хранилищ

Для локального хранения данных предоставляются два механизма:

- `sessionStorage`. Удерживает информацию в хранилище до тех пор, пока окно браузера не будет закрыто, и предоставляет доступ к данным только этому окну;
- `localStorage`. Сохраняет информацию навсегда, а пользоваться данными в хранилище могут все окна, в которых выполняется одно и то же приложение. Информация остается в хранилище до тех пор, пока пользователь не примет решение удалить ее.

Методы

Этот API включает общий интерфейс с новыми методами, свойствами и событиями:

- `setItem(key, value)`. Создает новый элемент данных и записывает его в пространство хранилища, зарезервированное для данного приложения. Элемент представляет собой пару из ключа и значения, которая составляется на основе атрибутов `key` и `value`;
- `getItem(key)`. Извлекает содержимое элемента, ключевое слово которого совпадает с указанным в атрибуте `key`. Необходимо передавать то значение ключевого слова, которое использовалось при создании элемента в методе `setItem()`;
- `key(index)`. Возвращает ключевое слово элемента, находящегося в хранилище на позиции, соответствующей атрибуту `index`;
- `removeItem(key)`. Удаляет элемент, ключевое слово которого совпадает с указанным в атрибуте `key`. Необходимо передавать то значение клю-

чевого слова, которое использовалось при создании элемента в методе `setItem()`;

- `clear()`.Удаляет все элементы из пространства хранилища, зарезервированного для данного приложения.

Свойства

- `length`. Возвращает число элементов, содержащихся в зарезервированном для приложения пространстве хранилища.

События

- `storage`. Срабатывает каждый раз, когда в пространстве хранилища, зарезервированном для приложения, происходят изменения.

11 API индексированных баз данных

Низкоуровневый API

API хранилища, с которым мы познакомились в предыдущей главе, удобно использовать для хранения незначительных объемов данных, но, когда речь заходит о большом количестве структурированной информации, приходится прибегать к помощи систем баз данных. HTML5 предлагает собственное решение для таких ситуаций — API IndexedDB (API индексированных баз данных).

API IndexedDB представляет собой систему баз данных, предназначенную для хранения индексированной информации на компьютере пользователя. Эта система разрабатывалась как низкоуровневый API, поддерживающий широкий диапазон вариантов использования. В итоге данный API стал самым мощным, но и самым сложным в спецификации HTML5. Целью его разработки было формирование простейшей инфраструктуры, на базе которой разработчики смогут создавать собственные решения, а также добавление узкоспециализированных высокоуровневых интерфейсов. Результат таков: на изучение этого API и привыкание к нему большинству разработчиков приходится потратить немало времени, и очень часто работа с ним сводится к использованию популярных библиотек, таких как jQuery или других, которые наверняка появятся в ближайшем будущем.

Структура, предлагаемая IndexedDB, отличается от SQL и других распространенных систем баз данных, широко применяемых разработчиками. Информация в базе данных хранится в форме объектов (записей) внутри сущностей, называемых хранилищами объектов (таблиц). У хранилища объектов (Object Stores) нет четко определенной структуры, всего лишь название и индексы, позволяющие находить содержащиеся в нем объекты. С объектами также не связана никакая предопределенная структура — они могут быть совершенно разными и настолько сложными, насколько вам требуется. Единственное ограничение, которое накладывается на структуру объектов, заключается в наличии как минимум одного свойства, объявленного как индекс. Без этого объекты невозможно будет находить внутри хранилищ объектов.

База данных

Сама база данных проста. Поскольку каждая база данных связывается с одним компьютером и одним веб-сайтом или приложением, проблемы присваивания различных имен пользователя или создания других систем разграничения доступа не возникает. Нужно всего лишь указать название и версию, и база данных будет готова к работе.

Для открытия баз данных API-интерфейс предоставляет атрибут `indexedDB` и метод `open()`. Этот метод возвращает объект, на котором могут срабатывать два события, указывая на ошибку создания базы данных или успешное выполнение операции.

Второй аспект, который необходимо учитывать при попытке создать или открыть базу данных, — это версия. API требует, чтобы с базой данных всегда связывалась версия. Это необходимо для подготовки системы к будущим миграциям. При необходимости обновить структуры базы данных на серверной стороне, например добавить таблицы или индексы, вы обычно отключаете сервер, переносите информацию в новую структуру, а затем вновь включаете сервер. Однако пользовательский компьютер невозможно выключить, для того чтобы выполнить аналогичный процесс в браузере. Из-за этого приходится менять версию базы данных и переносить информацию из старой версии в новую.

Для работы с версиями баз данных в API предусмотрены свойство `version` и метод `setVersion()`. Свойство возвращает значение текущей версии базы данных, а метод связывает с используемой в данный момент базой данных новую версию. В качестве значения версии могут выступать любые число или строка.

Объекты и хранилища объектов

То, что в привычных базах данных мы называем записями, в IndexedDB носит название объектов. К числу объектов относятся также свойства, позволяющие сохранять и идентифицировать значения. Количество свойств и структура объектов определяются произвольно и могут быть любыми. Единственное ограничение: как минимум одно свойство необходимо объявить индексом хранилища объектов, для того чтобы иметь возможность выполнять поиск по хранилищу и извлекать нужные данные.

С хранилищами объектов (таблицами) также не связываются никакие конкретные требования к структуре. В момент создания требуется объявить только название и один или несколько индексов, позволяющих находить объекты внутри хранилища.

Рис. 11.1. Объекты с различными свойствами в хранилище объектов

Типичное хранилище объектов (рис. 11.1) может содержать объекты с самыми разными свойствами. У одних объектов есть свойство `DVD`, у других — свойство `Book` и т. д. У каждого объекта своя структура, но для того, чтобы их можно было находить в базе, объекты должны обладать как минимум одним свойством-индексом. В примере на рис. 11.1 в роли индекса может выступать свойство `Id`.

Для того чтобы приступить к работе с объектами и хранилищами объектов, необходимо всего лишь создать хранилище объектов и объявить

свойства, которые будут использоваться в качестве индексов. После этого можно начинать сохранять в него объекты. Пока что нам не нужно думать о структуре или содержимом объектов, а нужно всего лишь предусмотреть индексы, с помощью которых позже будем находить их.

В API предусмотрены несколько методов для манипулирования хранилищами объектов:

- `createObjectStore(name, keyPath, autoIncrement)`. Создает новое хранилище объектов с именем и настройками, определяемыми его атрибутами. Обязательно указывать только название — атрибут `name`. Атрибут `keyPath` объявляет общий индекс для всех объектов. Атрибут `autoIncrement` — это булево значение, определяющее, будет ли в хранилище объектов использоваться генератор ключей;
- `objectStore(name)`. Для доступа к объектам в хранилище объектов необходимо начать транзакцию и открыть хранилище для данной транзакции. Этот метод открывает хранилище объектов с именем, указанным в атрибуте `name`;
- `deleteObjectStore(name)`. Разрушает хранилище объектов с именем, указанным в атрибуте `name`.

Методы `createObjectStore()` и `deleteObjectStore()`, как и другие методы, ответственные за конфигурацию базы данных, применяются только при создании базы данных или обновлении ее до новой версии.

Индексы

Для поиска объектов в хранилище необходимо объявить часть свойств объектов индексами. Проще всего сделать это, добавив атрибут `keyPath` при вызове метода `createObjectStore()`. Свойство, указанное в атрибуте `keyPath`, будет считаться общим индексом для всех объектов, находящихся в данном хранилище. Если атрибут `keyPath` используется, то соответствующее свойство должен иметь каждый объект.

Для определения любых желаемых индексов в хранилище объектов помимо атрибута `keyPath` можно использовать специальные методы, перечисленные далее:

- `createIndex(name, property, unique)`. Создает индекс для определенного объекта. Атрибут `name` — это идентификатор индекса, `property` — свойство объекта, которое будет использоваться в качестве индекса,

а `unique` — булево значение, определяющее уникальность индекса, то есть можно ли будет использовать одно и то же значение индекса для двух или более объектов;

- `index(name)`. Для того чтобы воспользоваться индексом, нужно сначала создать ссылку на индекс, а затем связать ее с транзакцией. Метод `index()` создает ссылку на индекс, определяемый атрибутом `name`;
- `deleteIndex(name)`. Если индекс нам больше не нужен, его можно удалить с помощью этого метода.

Транзакции

Система баз данных, работающая в браузере, должна учитывать определенные уникальные обстоятельства, не встречающиеся на других платформах. В браузере может произойти сбой, он может внезапно закрыться, процесс может быть остановлен пользователем, в текущем окне может начаться загрузка другого веб-сайта и т. п. Есть множество ситуаций, в которых работа напрямую с базой данных приведет к ошибкам или даже повреждению информации. Для предотвращения подобных проблем каждое действие с базой данных выполняется в пределах транзакции.

Метод, предназначенный для создания транзакции, называется `transaction()`, а тип транзакции задается с помощью перечисленных далее атрибутов:

- `READ_ONLY`. Определяет транзакцию, разрешающую только чтение. Никакие модификации не допускаются;
- `READ_WRITE`. Когда используется данный тип транзакции, данные можно считывать и записывать. Модификации разрешены;
- `VERSION_CHANGE`. Используется только для обновления версии базы данных.

Чаще всего применяются транзакции, допускающие чтение и запись данных. Однако типом по умолчанию считаются транзакции только для чтения — это сделано для предотвращения ненадлежащего использования. Таким образом, если перед нами стоит задача всего лишь извлечь какую-то информацию из базы данных, то единственное, что требуется, — указать масштаб транзакции (чаще всего название хранилища объектов, в котором будет вцыполняться поиск данных).

Методы хранилища объектов

Для взаимодействия с хранилищами объектов, то есть считывания и записи информации, API предоставляет несколько методов:

- `add(object)`. Принимает пару из ключа и значения или объект, содержащий несколько пар «ключ/значение», и добавляет в выбранное хранилище объект с этой информацией. Если объект с предложенным индексом уже существует, то метод `add()` возвращает ошибку;
- `put(object)`. Похож на предыдущий. Единственное различие заключается в том, что в случае обнаружения объекта с таким же индексом он стирает старый объект и записывает в хранилище новый. Данный метод удобно применять для модификации объекта, ранее сохраненного в выбранном хранилище объектов;
- `get(key)`. С помощью этого метода можно извлечь из хранилища определенный объект. Атрибут `key` должен содержать значение индекса того объекта, который мы хотим прочитать;
- `delete(key)`. Для удаления объекта из выбранного хранилища объектов нужно всего лишь вызвать данный метод, передав ему в качестве атрибута значение индекса.

Практическое применение индексированных баз данных

Достаточно теории! Давайте создадим нашу первую базу данных и применим часть упомянутых методов. Это будет приложение для хранения информации о фильмах. Вы сможете добавить собственную информацию, но для начала поместим в базу данных следующие описания:

```
id: tt0068646 name: The Godfather date: 1972
id: tt0086567 name: WarGames date: 1983
id: tt0111161 name: The Shawshank Redemption date: 1994
id: tt1285016 name: The Social Network date: 2010
```

ВНИМАНИЕ

Здесь перечислены названия свойств (id, name и date), которые мы будем использовать во всех последующих примерах в этой главе. Значения свойств взяты с веб-сайта http://www.imdb.com, однако вы можете составить собственный список или использовать для тестирования примеров кода случайные наборы символов.

Шаблон

Как всегда, нам понадобятся HTML-документ и таблица CSS-стилей, с помощью которых мы создадим и украсим поля формы для ввода и отображения информации. Для того чтобы добавить в базу данных сведения о фильме, нужно будет ввести ключевое слово, название фильма и год его производства.

Листинг 11.1. Шаблон для работы с API IndexedDB

```
<!DOCTYPE html>
<html lang="ru">
<head>
  <title>API IndexedDB</title>
  <link rel="stylesheet" href="indexed.css">
  <script src="indexed.js"></script>
</head>
<body>
  <section id="formbox">
    <form name="form">
      <p>Ключевое слово:<br><input type="text"
         name="keyword" id="keyword"></p>
      <p>Название:<br><input type="text" name="text" id="text"></p>
      <p>Год:<br><input type="text" name="year" id="year"></p>
      <p><input type="button" name="save" id="save"
          value="Сохранить"></p>
    </form>
  </section>
  <section id="databox">
    Информация недоступна
  </section>
</body>
</html>
```

CSS-стили определяют поля формы и способ визуализации информации на экране.

Листинг 11.2. Стили полей формы

```
#formbox{
  float: left;
  padding: 20px;
```

продолжение ⇩

Листинг 11.2 *(продолжение)*

```
    border: 1px solid #999999;
  }
  #databox{
    float: left;
    width: 400px;
    margin-left: 20px;
    padding: 20px;
    border: 1px solid #999999;
  }
  #keyword, #text{
    width: 200px;
  }
  #databox > div{
    padding: 5px;
    border-bottom: 1px solid #999999;
  }
```

САМОСТОЯТЕЛЬНО

Вам понадобятся HTML-файл для шаблона из листинга 11.1 и CSS-файл с именем indexed.css для стилей из листинга 11.2. Также создайте JavaScript-файл под названием indexed.js для сохранения примеров кода, которые будут рассматриваться далее.

Открытие базы данных

В первую очередь в JavaScript-коде нам нужно открыть базу данных. Атрибут `indexedDB` и метод `open()` открывают существующую базу данных с указанным именем или создают новую, если базы с этим именем еще нет.

Листинг 11.3. Открытие базы данных

```
function initiate(){
  databox=document.getElementById('databox');
  var button=document.getElementById('save');
  button.addEventListener('click', addobject, false);
  if('webkitIndexedDB' in window){
```

```
    window.indexedDB=window.webkitIndexedDB;
    window.IDBTransaction=window.webkitIDBTransaction;
    window.IDBKeyRange=window.webkitIDBKeyRange;
    window.IDBCursor=window.webkitIDBCursor;
  }else if('mozIndexedDB' in window){
    window.indexedDB=window.mozIndexedDB;
  }

  var request=indexedDB.open('mydatabase');
  request.addEventListener('error', showerror, false);
  request.addEventListener('success', start, false);
}
```

Функция `initiate()` из листинга 11.3 подготавливает элементы шаблона и открывает базу данных. Инструкция `indexedDB.open()` выполняет попытку открытия базы данных с именем `mydatabase` и возвращает объект запроса с результатом операции. В зависимости от результата операции на этом объекте срабатывает событие `error` или `success`.

ВНИМАНИЕ

На момент написания этой главы разработка данного API находилась на стадии экспериментов. К некоторым атрибутам, включая indexedDB, необходимо добавлять префиксы, для того чтобы они правильно работали в разных браузерах. Прежде чем открывать базу данных в функции initiate(), мы проверили наличие webkitIndexedDB или mozIndexedDB, а затем подготовили атрибуты для конкретного механизма браузера. Когда экспериментальный период закончится, вы сможете удалить условный оператор if в начале кода из листинга 11.3.

События — важная часть данного API. IndexedDB обладает свойствами как синхронного, так и асинхронного API. Синхронная часть в настоящее время только разрабатывается, и она будет тесно связана с API Web Workers (Рабочие процессы). В противоположность этому асинхронная часть предназначена для обычного использования в Сети и уже доступна разработчикам приложений. Асинхронная система выполняет задачи в фоновом режиме и возвращает результаты позже. С этой целью API запускает различные события для каждой операции. Любое действие над базой данных и ее содержимым обрабатывается в фоне (пока система выполняет другие сценарии и программы), а события срабатывают позже, информируя вызвавший их код о результатах.

После того как API заканчивает обработку запроса к базе данных, срабатывает событие `error` или `success` и в нашем коде выполняется функция `showerror()` или `start()` для отображения сообщения об ошибке или перехода к определению базы данных соответственно.

Версия базы данных

Для того чтобы завершить определение базы данных, нужно сделать еще несколько вещей. Как уже говорилось, базам данных IndexedDB всегда соответствует определенный номер версии. При создании базы данных ее версии присваивается значение `null`. Таким образом, проверив это значение, мы можем точно определить, имеем мы дело с новой базой данных или нет.

Листинг 11.4. Установка версии и реагирование на события

```
function showerror(e){
  alert('Ошибка: '+e.code+' '+e.message);
}
function start(e){
  db=e.result || e.target.result;
  if(db.version==''){
    var request=db.setVersion('1.0');
    request.addEventListener('error', showerror, false);
    request.addEventListener('success', createdb, false);
  }
}
```

> **ВНИМАНИЕ**
>
> Сейчас одни браузеры отправляют результирующий объект через событие, а другие — через элемент, на котором событие сработало. Для того чтобы автоматически выбиралась нужная ссылка, мы добавили логическое выражение e.result || e.target.result. Когда спецификация будет готова, вероятно, можно будет всегда использовать только одну ссылку.

Наша функция `showerror()` очень проста (в таком маленьком приложении нет необходимости обрабатывать ошибки). Здесь мы всего лишь генерируем сообщение об ошибке, используя атрибуты `code` и `message` интерфейса `IDBErrorEvent`. В то же время функция `start()` выполняет

необходимую последовательность шагов для распознавания версии базы данных и установки правильного значения в случае, если приложение запускается впервые. Функция связывает объект `result`, созданный событием, с переменной `db`. В дальнейшем эта переменная представляет базу данных и используется для обращения к ней.

В интерфейсе `IDBDatabase` предусмотрено свойство `version`, информирующее о текущей версии базы данных, а также метод `setVersion()`, позволяющий задать новое значение версии. В функции `start()` из листинга 11.4 мы распознаем текущее значение версии, а затем в зависимости от результата задаем или не задаем новое. Если база данных уже существует, то значение свойства `version` отличается от `null`, следовательно, ничего настраивать не нужно. Однако если пользователь запускает приложение впервые, то в свойстве `version` мы обнаруживаем значение `null` и нам нужно задать новую версию и настроить базу данных.

Метод `setVersion()` принимает строку, содержащую любое число или символьную последовательность, которую вы хотите объявить в качестве версии. Необходимо позаботиться только о том, чтобы в каждом коде, где открывается эта версия базы данных, использовалась та же самая строка. Данный метод, как и любая другая процедура в этой части API, работает по асинхронному принципу. Версия устанавливается в фоновом режиме, а о результате ваш код информируется посредством событий. В случае ошибки можно снова вызывать функцию `showerror()`, но если установка номера версии проходит правильно, то вызывается функция `createdb()`. Она объявляет хранилища объектов и индексы для этой новой версии.

Хранилища объектов и индексы

В настоящее время приходится заранее продумывать, объекты какого типа будут храниться в базе данных и как вы позже будете извлекать эту информацию из хранилищ объектов. Если что-то пойдет не так или если вы захотите в будущем добавить что-то в конфигурацию базы данных, вам придется задать новую версию и перенести данные из предыдущей. Причина заключается в том, что создавать хранилища объектов и индексы можно только во время транзакции `setVersion`.

Листинг 11.5. Объявление хранилищ объектов и индексов

```
function createdb(){
  var objectstore=db.createObjectStore('movies',{keyPath:'id'});
  objectstore.createIndex('SearchYear', 'date',{unique: false});
}
```

В нашем примере нужно создать только одно хранилище объектов (для сохранения информации о фильмах) и два индекса. Первый индекс, `id`, передается методу `createObjectStore()` в атрибуте `keyPath` при создании хранилища объектов. Второй индекс связывается с хранилищем объектов в методе `createIndex()`. Этот индекс идентифицируется именем `SearchYear` и объявляется для свойства `date`. Мы будем использовать его для упорядочивания списка фильмов по году производства.

Добавление объектов

Итак, у нас есть имя базы данных, `mydatabase`, и мы знаем, что номер ее версии 1.0. Кроме того, мы создали одно хранилище объектов с именем `movies` и двумя индексами: `id` и `date`. Самое время поместить в хранилище несколько объектов.

Листинг 11.6. Добавление объектов

```
function addobject(){
  var keyword=document.getElementById('keyword').value;
  var title=document.getElementById('text').value;
  var year=document.getElementById('year').value;

  var transaction=db.transaction(['movies'],
      IDBTransaction.READ_WRITE);
  var objectstore=transaction.objectStore('movies');
  var request=objectstore.add({id: keyword, name: title,
      date: year});
  request.addEventListener('success', function(){ show(keyword) },
      false);
  document.getElementById('keyword').value='';
  document.getElementById('text').value='';
  document.getElementById('year').value='';
}
```

В начале функции `initiate()` мы добавили прослушиватель события `click`, привязанный к кнопке на форме. При срабатывании события этот прослушиватель вызывает функцию `addobject()`. Она принимает значения из полей формы (`keyword`, `text` и `year`) и генерирует транзакцию, посредством которой эта информация сохраняется в новом объекте базы данных.

Для того чтобы начать транзакцию, нужно вызвать метод `transaction()`, указав участвующие в транзакции хранилища объектов и тип транзак-

ции. В данном случае мы указываем единственное хранилище — `movies`, а тип выбираем `READ_WRITE`.

На следующем шаге необходимо выбрать хранилище объектов, с которым мы будем работать. Поскольку транзакция может быть запущена для нескольких хранилищ, обязательно следует объявить, с каким хранилищем связывается конкретная операция. Используя метод `objectStore()`, мы открываем хранилище объектов и связываем его с транзакцией:

`transaction.objectStore('movies')`.

После этого можно добавлять объект в хранилище объектов. В примере мы используем метод `add()`, так как создаем новые объекты, однако если бы перед нами стояла задача модифицировать или заменить старые объекты, можно было бы применить метод `put()`. Метод `add()` принимает свойства `id`, `name` и `date`, а также переменные `keyword`, `title` и `year` и создает объект, добавляя в него пары «ключевое слово/значение», составленные из переданной информации.

Наконец, мы прослушиваем событие, запускаемое данным запросом, и в случае успеха вызываем функцию `show()`. Разумеется, вторым возможным событием является `error`, но, поскольку ответ зависит исключительно от нашего приложения, в данном примере мы такую возможность не рассматриваем.

Извлечение объектов

Если запись объекта в хранилище завершается успешно, срабатывает событие `success` и вызывается функция `show()`. В коде из листинга 11.6 мы объявили эту функцию внутри анонимной функции. Это нужно для того, чтобы иметь возможность передавать ей переменную `keyword`. Теперь воспользуемся этим значением для считывания ранее сохраненного объекта.

Листинг 11.7. Отображение содержимого нового объекта

```
function show(keyword){
  var transaction=db.transaction(['movies']);
  var objectstore=transaction.objectStore('movies');
  var request=objectstore.get(keyword);
  request.addEventListener('success', showlist, false);
}
```

продолжение ⇩

Листинг 11.7 *(продолжение)*

```
function showlist(e){
  var result=e.result || e.target.result;
  databox.innerHTML='<div>'+result.id+' - '+result.name+' -
'+result.date+'</div>';
}
```

Код из листинга 11.7 создает допускающую только чтение транзакцию (транзакцию типа `READ_ONLY`) и с помощью метода `get()` извлекает объект с указанным ключевым словом. Мы не объявляем тип транзакции явно, так как тип `READ_ONLY` используется по умолчанию.

Метод `get()` возвращает объект из хранилища по условию `id=keyword`. Например, если мы добавим фильм «The Godfather» из нашего списка, то в переменной `keyword` окажется значение tt0068646. Функция `show()` получает это значение и передает методу `get()` для извлечения информации о фильме «The Godfather». Как видите, этот код выполняет исключительно иллюстративную функцию, поскольку всего лишь возвращает только что добавленный фильм.

Так как все операции выполняются в асинхронном режиме, для вывода этой информации на экран нам требуются две функции. Функция `show()` создает транзакцию, а функция `showlist()` в случае успешного завершения транзакции отображает значения свойств на экране. Мы снова прослушиваем только событие `success`, однако в случае ошибки данная операция также может запустить событие `error`.

Функция `showlist()` получает объект, поэтому для доступа к его свойствам нам нужны представляющая его переменная и название свойства (например, `result.id`). Переменная `result` представляет собой объект, а `id` — это одно из его свойств.

Завершение кода и тестирование

Как и в предыдущих примерах кода, для завершения этого примера нам необходимо добавить прослушиватель события `load` и, как только приложение будет полностью загружено в браузер, вызвать функцию `initiate()`.

Листинг 11.8. Инициализация приложения

```
window.addEventListener('load', initiate, false);
```

САМОСТОЯТЕЛЬНО

Скопируйте все примеры JavaScript-кода из листингов с 11.3 по 11.8 в файл indexed.js и откройте HTML-документ с кодом из листинга 11.1 в своем браузере. Добавляйте информацию о фильмах, перечисленных в начале главы, в поля формы и помещайте ее в базу данных. При добавлении данных о каждом новом фильме эти сведения будут также отображаться в поле справа от формы.

Перечисление данных

Метод `get()`, который мы используем в коде из листинга 11.7, возвращает объекты по одному (в нашем случае — только последний добавленный фильм). В следующем примере мы воспользуемся функциональностью курсоров, для того чтобы создать список всех фильмов, информация о которых есть в хранилище объектов `movies`.

Курсоры

Курсоры представляют собой альтернативный способ извлечения данных и поиска по группе объектов, которая была возвращена базой данных в одной транзакции. Курсор получает определенный список объектов из хранилища объектов и создает указатель, с помощью которого одновременно можно обратиться только к одному объекту в списке.

В API индексированных баз данных для создания курсора используется метод `openCursor()`. Он извлекает информацию из выбранного хранилища объектов и возвращает объект IDBCursor, предлагающий собственные атрибуты и методы для манипулирования курсором:

- `continue()`. Перемещает указатель курсора на одну позицию вперед, одновременно на курсоре срабатывает событие `success`. Когда указатель достигает конца списка, событие `success` также срабатывает, однако в этот раз возвращается пустой объект. Указатель можно переместить в произвольную позицию списка, передав методу значение индекса (указывается в скобках);
- `delete()`. Удаляет объект в текущей позиции курсора;
- `update(value)`. Аналогичен методу `put()`, однако обновляет значение объекта в текущей позиции курсора.

У метода `openCursor()` также есть атрибуты для указания типа возвращаемых объектов и их порядка. Со значениями по умолчанию он возвращает все доступные объекты в выбранном хранилище объектов, упорядочивая их по возрастанию. Подробнее об этом поговорим чуть позже.

Листинг 11.9. Список объектов

```
function initiate(){
  databox=document.getElementById('databox');
  var button=document.getElementById('save');
  button.addEventListener('click', addobject, false);
  if('webkitIndexedDB' in window){
    window.indexedDB=window.webkitIndexedDB;
    window.IDBTransaction=window.webkitIDBTransaction;
    window.IDBKeyRange=window.webkitIDBKeyRange;
    window.IDBCursor=window.webkitIDBCursor;
  }else if('mozIndexedDB' in window){
    window.indexedDB=window.mozIndexedDB;
}
  var request=indexedDB.open('mydatabase');
  request.addEventListener('error', showerror, false);
  request.addEventListener('success', start, false);
}
function showerror(e){
  alert('Ошибка: '+e.code+' '+e.message);
}
function start(e){
  db=e.result || e.target.result;
  if(db.version==''){
    var request=db.setVersion('1.0');
    request.addEventListener('error', showerror, false);
    request.addEventListener('success', createdb, false);
  }else{
    show();
  }
}
function createdb(){
  var objectstore=db.createObjectStore('movies',{keyPath: 'id'});
  objectstore.createIndex('SearchYear', 'date',{unique: false});
}
function addobject(){
  var keyword=document.getElementById('keyword').value;
```

```
  var title=document.getElementById('text').value;
  var year=document.getElementById('year').value;
  var transaction=db.transaction(['movies'],
      IDBTransaction.READ_WRITE);
  var objectstore=transaction.objectStore('movies');
  var request=objectstore.add({id: keyword, name: title,
      date: year});
  request.addEventListener('success', show, false);
  document.getElementById('keyword').value='';
  document.getElementById('text').value='';
  document.getElementById('year').value='';
}
function show(){
  databox.innerHTML='';
  var transaction=db.transaction(['movies']);
  var objectstore=transaction.objectStore('movies');
  var newcursor=objectstore.openCursor();
  newcursor.addEventListener('success', showlist, false);
}
function showlist(e){
  var cursor=e.result || e.target.result;
  if(cursor){
    databox.innerHTML+='<div>'+cursor.value.id+' - '+
            cursor.value.name+' - '+cursor.value.date+'</div>';
    cursor.continue();
  }
}
window.addEventListener('load', initiate, false);
```

В листинге 11.9 представлен полный JavaScript-код, необходимый для знакомства с этим примером. Из всех функций, применяемых для конфигурирования базы данных, немного изменилась только `start()`. Теперь, если версия базы данных отличается от `null` (то есть база данных была создана ранее), вызывается функция `show()`. Она выполняет всю работу по отображению списка объектов из хранилища объектов. Таким образом, если база данных уже существует, то, как только веб-страница загрузится, вы увидите список объектов в поле справа на экране.

Самые большие изменения в этом коде произошли в функциях `show()` и `showlist()`. Здесь мы впервые используем курсоры.

Даже если мы считываем информацию из базы данных с помощью курсора, это все равно нужно делать в пределах транзакции. Поэтому

в функции `show()` мы первым делом создаем разрешающую только чтение (тип `READ_ONLY`) транзакцию для хранилища объектов `movies`. Именно это хранилище объектов привязывается к транзакции, после чего мы открываем для данного хранилища курсор, применяя для этого метод `openCursor()`.

Если операция завершается успешно, то возвращается объект, содержащий всю полученную из хранилища информацию, на этом объекте срабатывает событие `success` и вызывается функция `showlist()`.

У объекта, возвращаемого операцией, есть несколько атрибутов, позволяющих читать информацию:

- `key`. Возвращает значение ключа объекта в текущей позиции курсора;
- `value`. Возвращает значение любого свойства объекта в текущей позиции курсора. Имя свойства необходимо указать как свойство атрибута, например `value.year`;
- `direction`. Объекты можно считывать в порядке по возрастанию или по убыванию. Этот атрибут возвращает текущее условие сортировки;
- `count`. Возвращает приблизительное количество объектов в курсоре.

В функции `showlist()` в листинге 11.9 есть условный оператор `if`. С помощью этого оператора мы проверяем содержимое курсора. Если никакие объекты возвращены не были, а также если указатель достиг конца списка, объект оказывается пустым и цикл завершается. Однако если указатель указывает на допустимый объект, то информация выводится на экран, а указатель с помощью метода `continue()` переводится на одну позицию вперед.

Важно запомнить, что необходимости в использовании цикла `while` здесь нет, так как метод `continue()` снова вызывает событие `success` и вся функция целиком выполняется еще раз. Так продолжается до тех пор, пока курсор не возвращает значение `null`, — в этом случае `continue()` больше не вызывается.

САМОСТОЯТЕЛЬНО

Код из листинга 11.9 заменяет собой все предыдущие примеры кода JavaScript. Очистите файл indexed.js и скопируйте в него этот новый код. Откройте шаблон из листинга 11.1 и, если вы еще этого не сделали, добавьте в базу данных информацию обо всех фильмах, перечисленных в начале главы. Вы будете видеть полный список уже сохраненных в базе фильмов в поле справа на экране. Данные будут упорядочиваться по возрастанию значения свойства id.

Изменение способа сортировки

Для того чтобы получить желаемый список, нужно изменить еще две вещи. В примере при перечислении фильмов мы сортируем их по возрастанию значения свойства `id`. Именно оно используется в качестве `keyPath` для хранилища объектов `movies`, но в то же время очевидно, что чаще всего пользователей интересует сортировка совсем по другому значению.

С учетом этого мы в функции `createdb()` создали еще один индекс, `SearchYear`, связав его со свойством `date`. Этот индекс позволит упорядочить список фильмов по году производства.

Листинг 11.10. Упорядочивание по убыванию года

```
function show(){
  databox.innerHTML='';
  var transaction=db.transaction(['movies']);
  var objectstore=transaction.objectStore('movies');
  var index=objectstore.index('SearchYear');

  var newcursor=index.openCursor(null, IDBCursor.PREV);
  newcursor.addEventListener('success', showlist, false);
}
```

Функция из листинга 11.10 заменяет собой функцию `show()` из кода в листинге 11.9. Эта новая функция создает транзакцию, затем связывает индекс `SearchYear` с используемым в транзакции хранилищем объектов и, наконец, с помощью `openCursor()` извлекает из хранилища объекты, у которых есть соответствующее данному индексу свойство (в нашем случае `date`).

Два атрибута позволяют настраивать процесс выбора и упорядочивания информации с использованием курсора. Первый атрибут определяет диапазон, в котором объекты выбираются, а значением второго может быть одна из перечисленных далее констант:

- `NEXT`. Возвращаемые объекты упорядочиваются по возрастанию (это значение по умолчанию);
- `NEXT_NO_DUPLICATE`. Возвращаемые объекты упорядочиваются по возрастанию, а дублирующиеся объекты игнорируются (в случае обнаружения повторяющихся ключевых слов возвращается только первый объект);
- `PREV`. Возвращаемые объекты упорядочиваются по убыванию;

- `PREV_NO_DUPLICATE`. Возвращаемые объекты упорядочиваются по убыванию, а дублирующиеся объекты игнорируются (в случае обнаружения повторяющихся ключевых слов возвращается только первый объект).

Благодаря указанным в листинге 11.10 атрибутам метода `openCursor()` функция `show()` возвращает список объектов хранилища, упорядоченный по убыванию даты. В качестве значения атрибута `range` пока что используется `null`. Как определить диапазон выбора значений, мы узнаем в конце этой главы.

САМОСТОЯТЕЛЬНО

Откройте код из листинга 11.9 и замените функцию show() новой функцией, код которой представлен в листинге 11.10. Эта новая функция выводит на экран список фильмов, упорядочивая их по убыванию года (самые новые находятся наверху списка). Результат должен выглядеть так:

```
id: tt1285016 name: The Social Network date: 2010
id: tt0111161 name: The Shawshank Redemption date: 1994
id: tt0086567 name: WarGames date: 1983
id: tt0068646 name: The Godfather date: 1972
```

Удаление данных

Мы научились добавлять, извлекать и перечислять данные. Настало время узнать, как удалять объекты из хранилища объектов. Как уже упоминалось, входящий в состав API метод `delete()` принимает значение и удаляет объект, ключевое слово которого совпадает с этим значением.

Код очень простой. Нужно всего лишь создать кнопку для каждого объекта в списке и сгенерировать разрешающую чтение и запись транзакцию (тип `READ_WRITE`), и мы сможем выполнять операцию удаления.

Листинг 11.11. Удаление объектов

```
function showlist(e){
  var cursor=e.result || e.target.result;
  if(cursor){
    databox.innerHTML+='<div>'+cursor.value.id+' - '+cursor.value.
name+' - '+cursor.value.date+' <button onclick="remove(\''+cursor.
value.id+'\')">удалить</button></div>';
```

```
    cursor.continue();
  }
}
function remove(keyword){
  if(confirm('Вы уверены?')){
    var transaction=db.transaction(['movies'],
        IDBTransaction.READ_WRITE);
    var objectstore=transaction.objectStore('movies');
    var request=objectstore.delete(keyword);
    request.addEventListener('success', show, false);
  }
}
```

С кнопкой, которую мы создаем для каждого объекта в функции `showlist()` в листинге 11.11, связан строчный обработчик событий. Каждый раз, когда пользователь щелкает на одной из этих кнопок, вызывается функция `remove()`, а в качестве атрибута ей передается значение свойства `id`. Эта функция сначала генерирует транзакцию для чтения и записи (тип `READ_WRITE`), а затем, используя полученное ключевое слово, удаляет соответствующий объект из хранилища объектов `movies`.

В конце, если операция завершилась успешно, срабатывает событие `success` и вызывается функция `show()` для обновления списка фильмов на экране.

САМОСТОЯТЕЛЬНО

Откройте код из листинга 11.9, замените функцию showlist() и добавьте функцию remove() из кода, приведенного в листинге 11.11. Наконец, откройте HTML-документ с кодом из листинга 11.1, чтобы протестировать приложение.

Вы снова увидите список фильмов, однако теперь в каждой строке появится кнопка, позволяющая удалить соответствующую запись из хранилища объектов.

Поиск данных

Самая важная операция, для выполнения которой предназначены системы баз данных, — это поиск. Назначение системы в том и состоит, чтобы индексировать хранящуюся в базе информацию и тем самым делать

проще ее обнаружение. Как мы узнали ранее в этой главе, метод `get()` удобно использовать для извлечения одного объекта с известным значением ключевого слова, однако операция поиска обычно все же бывает несколько сложнее.

Для того чтобы получить определенный список объектов из хранилища объектов, нужно методу `openCursor()` передать первый аргумент — диапазон поиска. В рассматриваемом API есть интерфейс IDBKeyRange, включающий в себя несколько методов и свойств для объявления диапазона и ограничения списка возвращаемых объектов:

- `only(value)`. Возвращаются только объекты, ключевое слово которых соответствует указанному значению. Например, если при поиске фильмов по году задать условие `only("1972")`, то из всего списка будет возвращен только фильм «The Godfather»;
- `bound(lower, upper, lowerOpen, upperOpen)`. Для того чтобы создать настоящий диапазон, нужно указать начальное и конечное значения и определить, должны ли граничные значения включаться в список. Значение атрибута `lower` определяет начальную точку списка. Атрибут `upper` соответствует конечной точке. Атрибуты `lowerOpen` и `upperOpen` — это логические значения, указывающие, должны ли игнорироваться объекты, в точности соответствующие значениям атрибутов `lower` и `upper`. Например, `bound("1972", "2010", false, true)` возвращает список фильмов, выпущенных в промежутке от 1972 до 2010 года, не включая фильмы 2010 года (фильмы этого года не учитываются в списке, так как булев атрибут, описывающий конечную точку, равен `true`);
- `lowerBound(value, open)`. Создает открытый диапазон, начинающийся со значения `value` и продолжающийся до конца списка. Например, `lowerBound("1983", true)` возвращает все фильмы, созданные после 1983 года (то есть фильмы 1983 года в результаты поиска не включаются);
- `upperBound(value, open)`. Представляет собой полную противоположность предыдущего — тоже создает открытый диапазон, но при этом возвращаются объекты от начала списка до значения `value`. Например, `upperBound("1983", false)` возвращает список фильмов, созданных до 1983 года, причем фильмы 1983 года также входят в список.

Давайте подготовим новый шаблон и добавим на страницу форму поиска фильмов (листинг 11.12).

Листинг 11.12. Форма поиска

```
<!DOCTYPE html>
<html lang="ru">
<head>
  <title>API IndexedDB</title>
  <link rel="stylesheet" href="indexed.css">
  <script src="indexed.js"></script>
</head>
<body>
  <section id="formbox">
    <form name="form">
      <p>Поиск фильма по году:<br><input type="text" name="year"
         id="year"></p>
      <p><input type="button" name="find" id="find"
          value="Найти"></p>
    </form>
  </section>
  <section id="databox">
    Информация недоступна
  </section>
</body>
</html>
```

В новом HTML-документе есть кнопка и текстовое поле. В текстовом поле нужно ввести год, в результате будет возвращен список, соответствующий определенному в коде диапазону.

Листинг 11.13. Поиск фильмов

```
function initiate(){
  databox=document.getElementById('databox');
  var button=document.getElementById('find');
  button.addEventListener('click', findobjects, false);

  if('webkitIndexedDB' in window){
    window.indexedDB=window.webkitIndexedDB;
    window.IDBTransaction=window.webkitIDBTransaction;
    window.IDBKeyRange=window.webkitIDBKeyRange;
    window.IDBCursor=window.webkitIDBCursor;
  }else if('mozIndexedDB' in window){
    window.indexedDB=window.mozIndexedDB;
  }
```

продолжение ⇩

Листинг 11.13 *(продолжение)*

```
  var request=indexedDB.open('mydatabase');
  request.addEventListener('error', showerror, false);
  request.addEventListener('success', start, false);
}
function showerror(e){
  alert('Ошибка: '+e.code+' '+e.message);
}
function start(e){
  db=e.result || e.target.result;
  if(db.version==''){
    var request=db.setVersion('1.0');
    request.addEventListener('error', showerror, false);
    request.addEventListener('success', createdb, false);
  }
}
function createdb(){
  var objectstore=db.createObjectStore('movies', {keyPath: 'id'});
  objectstore.createIndex('SearchYear', 'date', { unique: false
});
}
function findobjects(){
  databox.innerHTML='';
  var find=document.getElementById('year').value;

  var transaction=db.transaction(['movies']);
  var objectstore=transaction.objectStore('movies');
  var index=objectstore.index('SearchYear');
  var range=IDBKeyRange.only(find);

  var newcursor=index.openCursor(range);
  newcursor.addEventListener('success', showlist, false);
}
function showlist(e){
  var cursor=e.result || e.target.result;
  if(cursor){
    databox.innerHTML+='<div>'+cursor.value.id+' - '+
            cursor.value.name+' - '+cursor.value.date+'</div>';
    cursor.continue();
  }
}
window.addEventListener('load', initiate, false);
```

Самая важная функция в листинге 11.13 — это `findobjects()`. В этой функции мы создаем транзакцию только для чтения (тип `READ_ONLY`), ссылающуюся на хранилище объектов `movies`, открываем индекс `SearchYear`, используя в качестве ориентира свойство `date`, и определяем диапазон, начинающийся со значения переменной `find` (это год, который пользователь указал в форме). Для определения диапазона мы используем метод `only()`, но вы можете протестировать и другие методы, перечисленные ранее. Диапазон передается в качестве аргумента методу `openCursor()`. После успешного выполнения операции функция `showlist()` выводит на экран список фильмов, соответствующих выбранному году.

Метод `only()` возвращает только фильмы, в точности соответствующие значению переменной `find`. Для тестирования остальных методов вы можете указать конкретные значения атрибутов, например `bound(find, "2011", false, true)`.

Метод `openCursor()` может одновременно принимать два атрибута. Таким образом, инструкция вроде `openCursor(range, IDBCursor.PREV)` вполне допустима — она вернет объекты из указанного диапазона, упорядочив их по убыванию (относительно того же индекса).

ВНИМАНИЕ

Возможность реализации полнотекстового поиска сейчас рассматривается, но разработка пока не начиналась и даже не включена в официальную спецификацию. Для получения обновленных примеров кода для данного API зайдите на наш веб-сайт и изучите ссылки для этой главы.

Краткий справочник. API IndexedDB (Индексированные базы данных)

API индексированных баз данных, или IndexedDB, работает на низком уровне. Методы и свойства, изученные в этой главе, представляют собой лишь небольшую часть возможностей данного API. Для того чтобы не усложнять примеры, мы не придерживались никакой конкретной структуры. Однако этот API, как и другие, включает в себя несколько интерфейсов. Например, существует специальный интерфейс для управления организацией базы данных, еще один интерфейс — для создания хранилищ объектов и манипулирования ими и т. д. Каждый интерфейс имеет

собственные методы и свойства. Поэтому далее вы найдете информацию, с которой уже познакомились в этой главе, но представленную чуть по-другому: мы структурировали ее в соответствии с официальной классификацией.

ВНИМАНИЕ

В этом кратком справочнике описаны лишь самые важные аспекты интерфейсов. Чтобы найти полную спецификацию, зайдите на наш сайт и изучите ссылки для этой главы.

Интерфейс среды (IDBEnvironment и IDBFactory)

Интерфейс среды, или IDBEnvironment, включает в себя атрибут IDBFactory. Совместно эти интерфейсы предоставляют элементы, необходимые для работы с базами данных:

- `indexedDB`. Этот атрибут обеспечивает механизм доступа к системе индексированных баз данных;
- `open(name)`. Этот метод открывает базу данных с указанным именем. Если такой базы данных нет, то создается новая база данных с названием, переданным методу в атрибуте `name`;
- `deleteDatabase(name)`. Этот метод удаляет базу данных, указанную в атрибуте `name`.

Интерфейс базы данных (IDBDatabase)

Объект, возвращаемый после открытия или создания базы данных, обрабатывается именно этим интерфейсом. Для работы с объектом в интерфейсе предусмотрено несколько методов и свойств:

- `version`. Это свойство возвращает текущую версию открытой базы данных;
- `name`. Это свойство возвращает название открытой базы данных;
- `objectStoreNames`. Это свойство возвращает список названий хранилищ объектов в открытой базе данных;
- `setVersion(value)`. Этот метод устанавливает новое значение версии для открытой базы данных. В качестве атрибута `value` можно передавать любую строку;

- `createObjectStore(name, keyPath, autoIncrement)`. Этот метод создает новое хранилище объектов в открытой базе данных. Атрибут `name` представляет собой название хранилища объектов, атрибут `keyPath` — это общий индекс для всех объектов в данном хранилище, а `autoIncrement` — булево значение, позволяющее активировать генератор ключей;
- `deleteObjectStore(name)`. Этот метод удаляет хранилище объектов, имя которого передано ему в атрибуте `name`;
- `transaction(stores, type, timeout)`. Этот метод инициализирует транзакцию. Транзакция связывается с одним или несколькими хранилищами объектов, объявленными в атрибуте `stores`, и допускает различные режимы доступа в соответствии со значением атрибута `type`. Также методу можно передать атрибут `timeout` со значением в миллисекундах, чтобы ограничить время выполнения операции. Подробнее о настройке транзакции рассказывается в разделе «Интерфейс транзакций (IDBTransaction)» краткого справочника.

Интерфейс хранилища объектов (IDBObjectStore)

Этот интерфейс предоставляет все методы и свойства, необходимые для манипулирования объектами в хранилище объектов:

- `name`. Это свойство возвращает имя используемого в данный момент хранилища объектов;
- `keyPath`. Это свойство возвращает значение `keyPath`, если оно определено, для используемого в данный момент хранилища объектов;
- `IndexNames`. Это свойство возвращает список имен индексов, определенных для используемого в данный момент хранилища объектов;
- `add(object)`. Этот метод добавляет в выбранное хранилище объектов новый объект с информацией из атрибутов. Если объект с таким индексом уже существует, то возвращается ошибка. В качестве атрибута метод может принимать как пару из ключевого слова и значения, так и объект, содержащий несколько пар «ключ/значение»;
- `put(object)`. Этот метод добавляет в выбранное хранилище объектов объект с информацией из атрибутов. Если объект с таким индексом уже существует, то он перезаписывается с использованием новой информации. В качестве атрибута метод может принимать как пару из ключевого слова и значения, так и объект, содержащий несколько пар «ключ/значение»;

- `get(key)`. Этот метод возвращает объект, индекс которого соответствует значению `key`;
- `delete(key)`. Этот метод удаляет объект, индекс которого соответствует значению `key`;
- `createIndex(name, property, unique)`. Этот метод создает новый индекс для выбранного хранилища объектов. Атрибут `name` содержит название индекса, атрибут `property` объявляет свойство объектов, с которым этот индекс будет связан, а атрибут `unique` указывает, допустимо ли наличие нескольких объектов с одинаковыми значениями индекса;
- `index(name)`. Этот метод открывает индекс с названием, соответствующим атрибуту `name`;
- `deleteIndex(name)`. Этот метод удаляет индекс с названием, соответствующим атрибуту `name`;
- `openCursor(range, direction)`. Этот метод создает курсор над объектом из выбранного хранилища объектов. Атрибут `range` принимает объект диапазона, определяющий способ выбора объектов. Атрибут `direction` указывает порядок следования объектов. Подробнее о конфигурировании курсора и манипулировании им рассказывается в разделе «Интерфейс курсора (IDBCursor)» краткого справочника. Построение диапазона рассматривается в разделе «Интерфейс диапазона (IDBKeyRangeConstructors)».

Интерфейс курсора (IDBCursor)

Этот интерфейс предоставляет конфигурационные значения для настройки порядка следования объектов, выбранных из хранилища объектов. Эти константы передаются в качестве второго атрибута метода `openCursor()`, например, `openCursor(null, IDBCursor.PREV)`:

- `NEXT`. Задает для объектов, на которые указывает курсор, сортировку по возрастанию (это значение по умолчанию);
- `NEXT_NO_DUPLICATE`. Задает для объектов, на которые указывает курсор, сортировку по возрастанию, а также пропуск дублирующихся объектов;
- `PREV`. Задает для объектов, на которые указывает курсор, сортировку по убыванию;
- `PREV_NO_DUPLICATE`. Задает для объектов, на которые указывает курсор, сортировку по убыванию, а также пропуск дублирующихся объектов;

В интерфейсе также предусмотрено несколько методов и свойств для манипулирования объектами, на которые указывает курсор:

- `continue(key)`. Этот метод перемещает указатель курсора на следующий объект в списке или на объект, определяемый атрибутом `key`, если он существует;
- `delete()`. Этот метод удаляет объект, на который в данный момент указывает курсор;
- `update(value)`. Этот метод обновляет объект, на который в данный момент указывает курсор, помещая в него значение из атрибута;
- `key`. Это свойство возвращает значение индекса для объекта, на который в данный момент указывает курсор;
- `value`. Это свойство возвращает значение любого свойства объекта, на который в данный момент указывает курсор;
- `direction`. Это свойство возвращает порядок следования объектов, которые считываются курсором (по возрастанию или убыванию).

Интерфейс транзакций (IDBTransaction)

Этот интерфейс предоставляет конфигурационные значения для задания типа очередной транзакции. Эти значения передаются во втором атрибуте метода `transaction()`, например, `transaction(stores, IDBTransaction.READ_WRITE)`:

- `READ_ONLY`. Эта константа настраивает транзакцию, допускающую только чтение (значение по умолчанию);
- `READ_WRITE`. Эта константа настраивает транзакцию, допускающую чтение и запись;
- `VERSION_CHANGE`. Этот тип транзакции используется только для обновления номера версии.

Интерфейс диапазона (IDBKeyRangeConstructors)

Этот интерфейс предоставляет несколько методов построения диапазона для выбора данных с помощью курсора:

- `only(value)`. Возвращает диапазон, начальная и конечная точки которого равны `value`;
- `bound(lower, upper, lowerOpen, upperOpen)`. Возвращает диапазон, начальная точка которого равна `lower`, а конечная точка — `upper`. Также можно указать, нужно ли исключать граничные точки из возвращаемого списка объектов;

- `lowerBound(value, open)`. Возвращает диапазон, начинающийся с `value` и продолжающийся до конца списка объектов. Атрибут `open` определяет, исключается объект, соответствующий `value`, из результирующего списка или нет;
- `upperBound(value, open)`. Возвращает диапазон, начало которого совпадает с началом списка объектов и который заканчивается на `value`. Атрибут `open` определяет, исключается объект, соответствующий `value`, из результирующего списка или нет.

Интерфейс ошибок (IDBDatabaseException)

Через этот интерфейс передаются ошибки, возвращаемые операциями над базой данных:

- `code`. Это свойство представляет кодовый номер ошибки;
- `message`. Это свойство возвращает сообщение с описанием ошибки.

Также возвращенное значение можно сравнить со следующим списком, чтобы найти соответствующую ошибку:

- `UNKNOWN_ERR` — значение 0;
- `NON_TRANSIENT_ERR` — значение 1;
- `NOT_FOUND_ERR` — значение 2;
- `CONSTRAINT_ERR` — значение 3;
- `DATA_ERR` — значение 4;
- `NOT_ALLOWED_ERR` — значение 5;
- `TRANSACTION_INACTIVE_ERR` — значение 6;
- `ABORT_ERR` — значение 7;
- `READ_ONLY_ERR` — значение 11;
- `RECOVERABLE_ERR` — значение 21;
- `TRANSIENT_ERR` — значение 31;
- `TIMEOUT_ERR` — значение 32;
- `DEADLOCK_ERR` — значение 33.

12 Файловый API

Хранилище файлов

Файлы — это единицы информации, которые пользователи могут с легкостью пересылать друг другу. Невозможно передать другому пользователю значение переменной, однако не составляет труда сделать копию любого файла и записать его на DVD-, флеш- или жесткий диск, отправить по Интернету и т. п. В файле можно сохранить огромный объем данных, но при этом перемещение, копирование и передача файла никак не зависят от природы его содержимого.

Файлы всегда были неотъемлемой составляющей любого приложения, однако до сих пор работать с ними в Сети было почти невозможно. Варианты использования ограничивались копированием на пользовательский компьютер или загрузкой на сервер уже существующих файлов. До появления спецификации HTML5 не существовало способов создания, копирования или иной обработки файлов в Сети.

Спецификация HTML5 разрабатывалась с учетом всех аспектов построения и взаимодействия веб-приложений. Ее создатели позаботились обо всем, начиная с элементарной структуры данных. И разумеется, они не могли обойти вниманием файлы. Таким образом, частью глобальной спецификации стал файловый API, или API File.

У файлового API довольно много общего с API хранилищ, которые мы изучали в предыдущих главах. Например, файловый API, как и IndexedDB, характеризуется низкоуровневой инфраструктурой, хотя

и не такой сложной, и может работать в синхронном или асинхронном режиме. Синхронная часть предназначена для использования с API рабочих процессов (Web Workers) — так же, как синхронная составляющая IndexedDB и других API, а асинхронная часть применяется в обычных веб-приложениях. Это означает, что нам придется заботиться обо всех аспектах процесса, проверять успешность завершения процесса и возможные ошибки и, вероятно, принимать на вооружение (или самостоятельно разрабатывать) более простые API, которые наверняка появятся в скором будущем.

Файловый API довольно стар, но он претерпел значительные изменения и улучшения. В настоящий момент он состоит как минимум из трех отдельных спецификаций: API File (Файл), API File: Directories & System (Каталоги и система) и API File: Writer (Запись файлов), — однако ситуация может измениться буквально в течение нескольких месяцев, если появятся новые спецификации или произойдет унификация существующих. По сути, API File позволяет нам взаимодействовать с локальными файлами и обрабатывать их содержимое из приложения; расширение API File: Directories & System предоставляет инструменты для работы с небольшой файловой системой, создаваемой специально для каждого приложения, а расширение API File: Writer предназначается для записи содержимого в файлы, создаваемые или загруженные приложением.

Обработка файлов пользователя

Работать с локальными файлами из веб-приложения довольно опасно. Разработчикам браузеров необходимо учесть в своих продуктах множество аспектов безопасности, прежде чем задумываться о том, как обрабатывать обращения к пользовательским файлам из приложений. Файловый API предоставляет только два метода загрузки: тег `<input>` и операцию перетаскивания.

В главе 8 мы научились применять возможности API Drag and Drop (Перетаскивание) для перетаскивания файлов из настольных приложений в специальную зону веб-страницы. Тег `<input>` с типом `file` работает по схожему принципу. И тег, и API перетаскивания передают файлы посредством свойства `files`. Нам нужно всего лишь исследовать значение этого свойства, для того чтобы извлечь все выбранные или перетащенные пользователем файлы, — так же, как мы уже делали раньше.

ВНИМАНИЕ

Этот API и его расширения в настоящее время не работают локально, к тому же их поддержка реализована только в браузерах Chrome и Firefox. Во время написания этой главы часть функциональности настолько нова, что работает только в экспериментальных браузерах, таких как Chromium (*http://www.chromium.org*) и Firefox Beta. Для проверки примеров кода из этой главы вам понадобится загрузить все файлы на сервер и тестировать их в новейших версиях браузеров.

Шаблон

В этой части главы мы будем использовать для выбора файлов тег `<input>`, однако вы всегда можете воспользоваться знаниями, полученными в главе 8, и интегрировать представленные далее примеры кода с функциональностью API Drag and Drop (Перетаскивание) (листинг 12.1).

Листинг 12.1. Шаблон для работы с файлами пользователя

```
<!DOCTYPE html>
<html lang="ru">
<head>
  <title>Файловый API</title>
  <link rel="stylesheet" href="file.css">
  <script src="file.js"></script>
</head>
<body>
  <section id="formbox">
    <form name="form">
      <p>Файл:<br><input type="file" name="myfiles"
         id="myfiles"></p>
    </form>
  </section>
  <section id="databox">
    Файл не выбран
  </section>
</body>
</html>
```

CSS-файл включает в себя стили для этого шаблона и других шаблонов, с которыми мы будем работать позже (листинг 12.2).

Листинг 12.2. Стили для формы и поля данных

```
#formbox{
  float: left;
  padding: 20px;
  border: 1px solid #999999;
}
#databox{
  float: left;
  width: 500px;
  margin-left: 20px;
  padding: 20px;
  border: 1px solid #999999;
}
.directory{
  color: #0000FF;
  font-weight: bold;
  cursor: pointer;
}
```

Считывание файлов

Для того чтобы прочитать файлы пользователя с его компьютера, нам понадобится помощь интерфейса `FileReader`. Этот интерфейс возвращает объект с несколькими методами, позволяющими добраться до содержимого каждого файла:

- `readAsText(file, encoding)`. Можно применять для обработки текстового содержимого. Событие `load` срабатывает на объекте `FileReader` после того, как загрузка файла завершается. Содержимое возвращается в виде текста в кодировке UTF-8, если только в атрибуте `encoding` не задан какой-либо другой вариант кодирования. Данный метод пытается интерпретировать каждый байт многобайтовой последовательности как текстовый символ;
- `readAsBinaryString(file)`. Считывает информацию как последовательность целых чисел в диапазоне от 0 до 255. Он просто просматривает каждый байт, не пытаясь никак интерпретировать его значение.

Данный метод удобно применять для обработки двоичного содержимого, такого как изображения или видео;

- `readAsDataURL(file)`. Генерирует представляющий содержимое файлов URL данных (data:url) в кодировке base64;
- `readAsArrayBuffer(file)`. Генерирует на основе данных файла данные в формате ArrayBuffer.

Листинг 12.3. Считывание текстового файла

```
function initiate(){
  databox=document.getElementById('databox');
  var myfiles=document.getElementById('myfiles');
  myfiles.addEventListener('change', process, false);
}
function process(e){
  var files=e.target.files;
  var file=files[0];
  var reader=new FileReader();
  reader.onload=show;
  reader.readAsText(file);
}
function show(e){
  var result=e.target.result;
  databox.innerHTML=result;
}
window.addEventListener('load', initiate, false);
```

Поле ввода данных в HTML-документе из листинга 12.1 позволяет выбрать файл для обработки. Чтобы реакция на действия пользователя была правильной, в листинге 12.3 мы добавили в функцию `initiate()` прослушиватель события `change`, связав его с элементом `<input>`, а для обработки события выбрали функцию `process()`.

Свойство `files`, отправляемое элементом `<input>` (и API-интерфейсом Drag and Drop (Перетаскивание)), представляет собой массив, содержащий все выбранные пользователем файлы. Если для элемента `<input>` не добавлен атрибут `multiple`, то выбор нескольких файлов невозможен, в этом случае единственным доступным элементом массива становится первый. В начале функции `process()` мы извлекаем содержимое свойства `files`, помещаем его в переменную `files`, а затем выбираем первый элемент данного массива с помощью кода `var file=files[0]`.

ВНИМАНИЕ

Для того чтобы больше узнать об атрибуте multiple, откройте листинг 6.17. Еще один пример работы с несколькими файлами вы найдете в листинге 8.10.

Если мы хотим обработать файл, то в первую очередь должны с помощью конструктора `FileReader()` получить объект `FileReader`. В функции `process()` из листинга 12.3 мы присваиваем этому объекту имя `reader`. После этого для объекта `reader` регистрируется обработчик события `onload`, который позволяет распознать ситуацию, когда файл готов к обработке. Наконец, метод `readAsText()` считывает файл и извлекает его содержимое в форме простого текста.

Когда метод `readAsText()` завершает чтение файла, срабатывает событие `load` и вызывается функция `show()`. Эта функция извлекает содержимое файла из свойства `result` объекта `reader` и выводит его на экран.

САМОСТОЯТЕЛЬНО

Создайте файлы с примерами кода из листингов 12.1–12.3. Имена CSS-файла и JavaScript-файла объявлены в HTML-документе как file.css и file.js соответственно. Откройте шаблон в браузере и с помощью формы выберите один из файлов на своем компьютере. Поэкспериментируйте с текстовыми файлами и с изображениями, для того чтобы понять, как содержимое каждого типа файлов будет визуализироваться на экране.

ВНИМАНИЕ

Пока что внедрение возможностей файлового API и его спецификаций не завершено – разработчики браузеров еще работают над этим. Примеры кода из этой главы тестировались в Chrome и Firefox 4, но в последнем релизе Chrome пока не добавлен метод addEventListener() для объекта FileReader и др. По этой причине, чтобы код работал правильно, мы использовали в примерах обработчики событий, такие как onload. Например, мы добавили строчку reader.onload=show вместо reader.addEventListener('load', show, false). Как всегда, вам понадобится протестировать фрагменты кода во всех доступных браузерах, чтобы выяснить, какие реализации готовы к использованию данного API.

Разумеется, в этом примере кода мы ожидаем, что пользователь будет открывать текстовые файлы, однако метод `readAsText()` может интерпретировать как текст любые данные, включая файлы с двоичным содержимым (например, изображения). Выбрав нетекстовый файл, вы увидите на экране множество забавных, но не слишком понятных символов.

Свойства файлов

В реальных приложениях, для того чтобы информировать пользователя о проходящих обработку файлах, а также для контроля над загружаемыми файлами, всегда нужно каким-то образом извлекать такую информацию, как имя файла, его размер или даже тип. Объект файла, отправляемый тегом `<input>`, предоставляет несколько свойств, позволяющих получить эту информацию:

- `name`. Возвращает полное имя файла (название и расширение);
- `size`. Возвращает размер файла в байтах;
- `type`. Возвращает тип файла как тип MIME.

Листинг 12.4. Загрузка изображений

```
function initiate(){
  databox=document.getElementById('databox');
  var myfiles=document.getElementById('myfiles');
  myfiles.addEventListener('change', process, false);
}
function process(e){
  var files=e.target.files;
  databox.innerHTML='';
  var file=files[0];
  if(!file.type.match(/image.*/i)){
    alert('insert an image');
  }else{
    databox.innerHTML+='Имя: '+file.name+'<br>';
    databox.innerHTML+='Размер: '+file.size+' bytes<br>';

    var reader=new FileReader();
    reader.onload=show;
    reader.readAsDataURL(file);
  }
}
```

продолжение ⇙

Листинг 12.4 *(продолжение)*

```
function show(e){
  var result=e.target.result;
  databox.innerHTML+='<img src="'+result+'">';
}
window.addEventListener('load', initiate, false);
```

Пример из листинга 12.4 очень похож на предыдущий, только на этот раз для считывания файла мы применили метод `readAsDataURL()`. Он возвращает содержимое файла в формате data:url, и его в дальнейшем можно использовать в качестве источника для тега `<img>`, чтобы вывести выбранное изображение на экран.

Если перед нами стоит задача обработать файл определенного типа, вполне естественно, что первым делом мы проверяем свойство этого файла под названием `type`. В функции `process()` из листинга 12.4 такая проверка выполняется при помощи старого метода `match()`. Если выясняется, что это не файл изображения, то метод `alert()` выводит на экран сообщение об ошибке. Если же все в порядке и это файл изображения, то на экран выводятся имя и размер, а также сама картинка.

Несмотря на то что мы выбрали другой метод, а именно `readAsDataURL()`, процесс открытия файла не изменился. Сначала создается объект `FileReader`, затем регистрируется обработчик события `onload`, и, наконец, файл загружается. После завершения этого процесса содержимое свойства `result` передается в функцию `show()`, где используется в качестве источника для тега `<img>`, для того чтобы изображение появилось на странице.

ПОВТОРЯЕМ ОСНОВЫ

Для построения фильтра мы использовали регулярные выражения и давно существующий метод JavaScript match(). Этот метод ищет соответствие между регулярным выражением и строкой и возвращает либо массив совпадений, либо значение null. Тип MIME для изображения может быть, например, image/jpeg, если формат файла JPG, или image/gif, если GIF, поэтому выражение /image./i позволяет ограничить чтение файлов только файлами изображений. Чтобы получить более подробную информацию о регулярных выражениях и типах MIME, зайдите на наш веб-сайт и изучите ссылки для этой главы.*

Бинарные блоки

Рассматриваемый API умеет работать помимо файлов также с источниками, относящимися к другому типу и носящими название бинарных блоков (binary large object, blob). Бинарный блок — это объект, представляющий собой необработанные данные. Этот тип был создан для преодоления ограничений JavaScript на работу с двоичными данными. Обычно бинарный блок генерируется на основе файла, но так бывает не всегда. Это хороший способ работать с данными без необходимости загружать файл целиком в память, и он позволяет обрабатывать двоичную информацию небольшими фрагментами.

Существует несколько вариантов использования бинарных блоков, но главная их задача — обеспечивать эффективный способ обработки больших объектов сырых данных или больших файлов. Для создания бинарного блока на основе другого бинарного блока или файла в API предусмотрен метод `slice()`: `slice(start, length, type)`. Этот метод возвращает новый бинарный блок, созданный на базе другого бинарного блока или файла. Первый атрибут указывает начальную точку, второй — длину нового бинарного блока, а третий представляет собой необязательный параметр, задающий тип данных.

Листинг 12.5. Работа с бинарными блоками

```
function initiate(){
  databox=document.getElementById('databox');
  var myfiles=document.getElementById('myfiles');
  myfiles.addEventListener('change', process, false);
}
function process(e){
  var files=e.target.files;
  databox.innerHTML='';
  var file=files[0];
  var reader=new FileReader();
  reader.onload=function(e){ show(e, file); };
  var blob=file.slice(0,1000);
  reader.readAsBinaryString(blob);
}
function show(e, file){
  var result=e.target.result;
  databox.innerHTML='Имя: '+file.name+'<br>';
  databox.innerHTML+='Тип: '+file.type+'<br>';
```

продолжение ⇗

Листинг 12.5 *(продолжение)*

```
    databox.innerHTML+='Размер: '+file.size+' bytes<br>';
    databox.innerHTML+='Размер бинарного блока: '+result.length+' bytes<br>';
    databox.innerHTML+='Бинарный блок: '+result;
}
window.addEventListener('load', initiate, false);
```

ВНИМАНИЕ

Из-за несовместимости с ранее существовавшими методами сейчас ведется разработка нового метода для замены slice. До тех пор, пока новый метод не появится, для тестирования кода из листинга 12.5 в последних версиях браузеров Firefox и Google Chrome вам нужно будет заменять название slice названиями, содержащими браузерные префиксы: mozSlice или webkitSlice соответственно. Для получения более подробной информации зайдите на наш веб-сайт и изучите ссылки для этой главы.

В коде из листинга 12.5 мы делаем то же самое, что делали раньше, но на этот раз вместо считывания файла целиком с помощью метода `slice()` создаем бинарный блок. Длина блока равна 1000 байт, а начинается он с нулевого байта файла. Если пользователь загружает файл размером менее 1000 байт, то длина бинарного блока ограничивается размером файла (от начала до метки EOF, то есть End of File — конец файла).

Для отображения информации, полученной в процессе, мы регистрируем обработчик события `onload` с анонимной функцией, позволяющей отправить ссылку на объект `file`. Принимает данную ссылку функция `show()`, которая и выводит на экран значения свойств объекта.

У бинарных блоков огромное количество преимуществ. Например, можно в цикле сгенерировать несколько бинарных блоков из одного файла, а затем обработать получившиеся части одну за другой. Это удобно для создания асинхронных загрузчиков, программ для обработки изображений и решения многих других задач. Бинарные блоки открывают новые возможности кода JavaScript.

События

В зависимости от размера файла длительность процесса его загрузки в память может сильно варьироваться. Для небольших файлов это кажет-

ся мгновенной операцией, но загрузка объемных файлов порой занимает до нескольких минут. Помимо рассмотренного события `load`, наш API предоставляет несколько других специальных событий, информирующих обо всех деталях процесса:

- `loadstart`. Срабатывает на объекте FileReader в момент, когда загрузка начинается;
- `progress`. Срабатывает периодически во время считывания файла или бинарного блока;
- `abort`. Срабатывает в случае, если процесс прерывается;
- `error`. Срабатывает в случае ошибки считывания;
- `loadend`. Аналогично событию `load`, но срабатывает в обоих случаях, как при успешном завершении, так и при завершении с ошибкой.

Листинг 12.6. Контролирование процесса с помощью событий

```
function initiate(){
  databox=document.getElementById('databox');
  var myfiles=document.getElementById('myfiles');
  myfiles.addEventListener('change', process, false);
}
function process(e){
  var files=e.target.files;
  databox.innerHTML='';
  var file=files[0];
  var reader=new FileReader();
  reader.onloadstart=start;
  reader.onprogress=status;
  reader.onloadend=function(){ show(file); };
  reader.readAsBinaryString(file);
}
function start(e){
  databox.innerHTML='<progress value="0" max="100">0%</progress>';
}
function status(e){
  var per=parseInt(e.loaded/e.total*100);
  databox.innerHTML=
       '<progress value="'+per+'" max="100">'+per+'%</progress>';
}
function show(file){
```

продолжение ⇨

Листинг 12.6 *(продолжение)*

```
  databox.innerHTML='Имя: '+file.name+'<br>';
  databox.innerHTML+='Тип: '+file.type+'<br>';
  databox.innerHTML+='Размер: '+file.size+' bytes<br>';
}
window.addEventListener('load', initiate, false);
```

В коде из листинга 12.6 мы создали приложение, которое загружает файл и визуально сообщает о прогрессе операции, обновляя индикаторную полосу. Для объекта `FileReader` зарегистрировали три обработчика событий, позволяющих контролировать процесс считывания, а для реагирования на эти события создали две новые функции: `start()` и `status()`. Функция `start()` инициализирует полосу прогресса со значением 0 % и выводит ее на экран. С индикаторной полосой можно связать любые значение или диапазон, но мы решили использовать процентные показатели, для того чтобы пользователям было проще ориентироваться. В функции `status()` мы вычисляем процентное значение загрузки, основываясь на значениях свойств `loaded` и `total`, которые возвращает событие `progress`. Полоса прогресса заново создается на экране каждый раз, когда срабатывает событие `progress`.

САМОСТОЯТЕЛЬНО

Используя шаблон из листинга 12.1 и JavaScript-код из листинга 12.6, попробуйте загрузить большой файл, например видеофайл или файл данных, чтобы протестировать полосу прогресса. Если браузер не распознает элемент <progress>, то вместо визуального представления вы увидите на экране содержимое элемента.

ВНИМАНИЕ

Для добавления нового элемента <progress> к документу мы использовали innerHTML. Так делать не рекомендуется, однако подобная практика может быть удобной и полезной в таких небольших задачах, как наша. Обычно элементы добавляются в DOM с помощью методов JavaScript createElement() и appendChild().

Создание файлов

Главный API File (Файл) применяется для загрузки файлов с компьютера пользователя и их последующей обработки, но он способен работать только с файлами, уже существующими на жестком диске. Возможность создания новых файлов или каталогов не рассматривается. Этим занимается расширение основного API под названием API File: Directories and System (Каталоги и система). Данный API резервирует на жестком диске особое пространство — специальное пространство хранилища, в котором веб-приложение может создавать и обрабатывать файлы и каталоги точно так же, как это делает любое настольное приложение. Это пространство уникально, и обратиться к нему можно только из создавшего его приложения.

ВНИМАНИЕ

Во время написания этой главы единственный браузер, в котором реализовано данное расширение файлового API, — это Chrome, однако он не резервирует пространство хранилища. Если вы попытаетесь выполнить следующие примеры кода, то вернется ошибка QUOTA_EXCEEDED. Для того чтобы получить возможность использовать API File: Directories and System, Chrome нужно открыть со следующими флагами: --unlimited-quota-for-files. В Windows, чтобы добавить эти флаги, откройте Рабочий стол, щелкните правой клавишей мыши на значке Chrome и выберите пункт контекстного меню Properties (Свойства). В открывшемся окне вы увидите поле Target (Объект), содержащее путь и имя запускаемого файла для браузера. Добавьте флаг --unlimited-quota-for-files в конец этой строки. В результате путь будет выглядеть примерно так:

```
C:\Users\...\Chrome\Application\chrome.exe
--unlimited-quota-for-files
```

Шаблон

Для тестирования этой составляющей API понадобится новая форма с полем ввода и кнопкой. На этой странице будем создавать и обрабатывать файлы и каталоги (листинг 12.7).

Листинг 12.7. Новый шаблон для API File: Directories and System

```
<!DOCTYPE html>
<html lang="ru">
<head>
  <title>Файловый API</title>
  <link rel="stylesheet" href="file.css">
  <script src="file.js"></script>
</head>
<body>
  <section id="formbox">
    <form name="form">
      <p>Название:<br><input type="text" name="myentry"
         id="myentry" required></p>
      <p><input type="button" name="fbutton" id="fbutton"
          value="Создать"></p>
    </form>
  </section>
  <section id="databox">
    Записи недоступны
  </section>
</body>
</html>
```

САМОСТОЯТЕЛЬНО

В этом HTML-документе мы создали новую форму, однако сохранили прежнюю структуру и стили CSS. Для тестирования следующих примеров замените предыдущий вариант HTML-документа кодом из этого листинга и копируйте примеры JavaScript-кода в файл file.js.

ВНИМАНИЕ

Мы добавили к элементу <input> атрибут request, однако он не будет учитываться в примерах кода для этой главы. Для того чтобы задействовать процесс валидации, нужно применить возможности API Forms (Формы). Пример вы найдете в главе 10 — это листинг 10.5.

Жесткий диск

Место, резервируемое для приложения, — это «песочница», небольшой блок на жестком диске с собственными корневым каталогом и конфигурацией. Для того чтобы начать работать с ним, в первую очередь необходимо инициализировать файловую систему для нашего приложения: `requestFileSystem(type, size, функция для успешного завершения, функция для завершения с ошибкой)`. Этот метод создает файловую систему такого размера и такого типа, как указано в его атрибутах. Атрибут `type` может принимать значение либо `TEMPORARY`, либо `PERSISTENT` в зависимости от того, как долго должны сохраняться данные: временно или на неограниченный срок соответственно. Атрибут `size` определяет размер пространства, которое резервируется на жестком диске для данной файловой системы, и задается в байтах. В случае успешного выполнения операции и в случае ошибки метод вызывает соответствующую функцию обратного вызова.

Метод `requestFileSystem()` возвращает объект файловой системы (File System) с двумя свойствами:

- `root`. Значение этого свойства представляет собой ссылку на корневой каталог файловой системы. По сути, это объект DirectoryEntry, с которым также связаны определенные методы (с ними мы познакомимся чуть позже). Через это свойство можно ссылаться на пространство хранилища и работать посредством него с файлами и каталогами;
- `name`. Возвращает информацию о файловой системе, такую как имя, назначенное ей браузером, и состояние.

Листинг 12.8. Настройка файловой системы

```
function initiate(){
  databox=document.getElementById('databox');
  var button=document.getElementById('fbutton');
  button.addEventListener('click', create, false);

  window.webkitRequestFileSystem(window.PERSISTENT, 5*1024*1024,
         createhd, showerror);
}
function createhd(fs) {
  hd=fs.root;
}
function create(){
```

продолжение ⇩

Листинг 12.8 *(продолжение)*

```
  var name=document.getElementById('myentry').value;
  if(name!=''){
    hd.getFile(name, {create: true, exclusive: false}, show, showerror);
  }
}
function show(entry){
  document.getElementById('myentry').value='';

  databox.innerHTML='Запись создана!<br>';
  databox.innerHTML+='Имя: '+entry.name+'<br>';
  databox.innerHTML+='Путь: '+entry.fullPath+'<br>';
  databox.innerHTML+='Файловая система: '+entry.filesystem.name;
}
function showerror(e){
  alert('Ошибка: '+e.code);
}
window.addEventListener('load', initiate, false);
```

ВНИМАНИЕ

Сейчас единственный вариант реализации этой части API можно найти в браузере Google Chrome. Поскольку реализация экспериментальная, нам понадобилось заменить метод requestFileSystem() специальным методом для Chrome, в названии которого присутствует браузерный префикс, — webkitRequestFileSystem(). Используя данный метод, вы можете тестировать этот и последующие примеры кода в своем браузере.

Объединяя HTML-документ из листинга 12.7 с кодом из листинга 12.8, мы получаем наше первое приложение для работы с новыми файлами на пользовательском компьютере. Сначала код вызывает метод `requestFileSystem()` для создания или получения ссылки на файловую систему; если приложение выполняется впервые, создается новая файловая система. В нашем случае это постоянная файловая система объемом 5 Мбайт ($5 \times 1024 \times 1024$). Если данная операция выполняется успешно, то вызывается функция `createhd()`, в которой процесс инициализации продолжается. В случае ошибок мы используем простую функцию `showerror()`, которую применяли и раньше для других API.

При создании или открытии файловой системы функция `createhd()` получает объект `FileSystem` и сохраняет ссылку на него — то есть значение его свойства `root` — в переменной `hd`.

Создание файлов

Файловая система создана. Остальные функции из листинга 12.8 создают новый файл и выводят на экран данные из этой записи. Когда пользователь нажимает на форме кнопку Создать, происходит вызов функции `create()`. Она связывает текст, введенный пользователем в элементе `<input>`, с переменной `name` и с помощью метода `getFile()` создает файл с этим именем.

Данный метод входит в интерфейс `DirectoryEntry`, являющийся частью рассматриваемого API. Интерфейс предоставляет четыре метода для создания и обработки файлов и каталогов:

- `getFile(path, options, функция для успешного завершения, функция для завершения с ошибкой)`. Создает или открывает файл. Значение атрибута `path` должно включать в себя имя файла и путь к его местоположению (начиная от корня файловой системы). В качестве атрибута `options` можно использовать два флага: `create` и `exclusive`. Оба принимают логические значения. Флаг `create` указывает, нужно ли создавать файл, а флаг `exclusive`, если его значение равно `true`, заставляет метод `getFile()` возвращать ошибку при попытке создать файл, идентичный уже существующему. Также среди параметров метода присутствуют две функции обратного вызова: для случая успешного завершения операции и для завершения с ошибкой;
- `getDirectory(path, options, функция для успешного завершения, функция для завершения с ошибкой)`. Аналогичен предыдущему, но предназначен для работы с каталогами;
- `createReader()`. Возвращает объект `DirectoryReader`, позволяющий считывать записи из указанного каталога;
- `removeRecursively()`. Это специальный метод для удаления каталога со всем его содержимым.

В коде из листинга 12.8 метод `getFile()`, основываясь на значении переменной `name`, создает или извлекает файл. Если такого файла еще не существует, то он создается (благодаря флагу `create: true`), в противном случае извлекается ссылка на него (флаг `exclusive: false`). Функция

create() также проверяет значение переменной name, прежде чем вызывать getFile().

Метод getFile() для обработки успешного или ошибочного завершения операции использует две функции, show() и showerror(). Функция show() получает объект Entry и выводит на экран значения его свойств. С объектами такого типа связаны несколько методов и свойств, о которых поговорим позже. Пока что мы учитываем только свойства name, fullPath и filesystem.

Создание каталогов

Методы getFile() (работает только с файлами) и getDirectory() (работает только с каталогами) аналогичны. Для того чтобы создать каталог, используя для ввода данных все тот же шаблон из листинга 12.7, нужно всего лишь заменить название метода getFile() на getDirectory(), как в следующем коде.

Листинг 12.9. Создание каталога с помощью метода getDirectory()

```
function create(){
  var name=document.getElementById('myentry').value;
  if(name!=''){
    hd.getDirectory(name, {create: true, exclusive: false}, show, showerror);
  }
}
```

Обратите внимание на то, что оба метода принадлежат объекту DirectoryEntry с именем root, для представления которого используется переменная hd. Таким образом, для вызова методов и создания файлов и каталогов в файловой системе нашего приложения приходится всегда использовать данную переменную.

САМОСТОЯТЕЛЬНО

Замените функцию create() в листинге 12.8 функцией из листинга 12.9, чтобы наше приложение создавало не файлы, а каталоги. Загрузите файлы приложения на свой сервер, откройте HTML-документ из листинга 12.7 в браузере и введите в форму на экране нужные данные для создания каталога.

Перечисление файлов

Как уже говорилось, метод `createReader()` позволяет получить список записей (файлов и каталогов) по определенному пути. Этот метод возвращает объект `DirectoryReader`, предлагающий метод `readEntries()` для чтения записей из соответствующего каталога: `readEntries(функция для успешного завершения, функция для завершения с ошибкой)`. Этот метод считывает очередной блок записей из выбранного каталога. При каждом вызове метода функция для успешного завершения возвращает объект со списком записей или значение `null`, если ни одной записи найдено не было.

Метод `readEntries()` считывает список записей блоками. Следовательно, невозможно гарантировать, что единственный вызов метода вернет все существующие записи. Придется вызывать метод до тех пор, пока он не вернет пустой объект.

Прежде чем приступать к написанию кода, нужно учесть и еще одну тонкость. Метод `createReader()` возвращает объект `DirectoryReader` для определенного каталога. Для того чтобы прочитать список файлов, нужно сначала получить объект `Entry` для каталога, который мы собираемся просмотреть.

Листинг 12.10. Приложение для работы с файловой системой

```
function initiate(){
  databox=document.getElementById('databox');
  var button=document.getElementById('fbutton');
  button.addEventListener('click', create, false);

  window.webkitRequestFileSystem(window.PERSISTENT, 5*1024*1024,
         createhd, showerror);
}
function createhd(fs) {
  hd=fs.root;
  path='';
  show();
}
function showerror(e){
  alert('Ошибка: '+e.code);
}
function create(){
```

продолжение ⇩

Листинг 12.10 *(продолжение)*

```
    var name=document.getElementById('myentry').value;
    if(name!=''){
      name=path+name;
      hd.getFile(name, {create: true, exclusive: false}, show,
         showerror);
    }
  }
  function show(){
    document.getElementById('myentry').value='';

    databox.innerHTML='';
    hd.getDirectory(path,null,readdir,showerror);
  }
  function readdir(dir){
    var reader=dir.createReader();
    var read=function(){
      reader.readEntries(function(files){
        if(files.length){
          list(files);
          read();
        }
      }, showerror);
    }
    read();
  }
  function list(files){
    for(var i=0; i<files.length; i++) {
      if(files[i].isFile) {
        databox.innerHTML+=files[i].name+'<br>';
      }else if(files[i].isDirectory){
        databox.innerHTML+='<span onclick="changedir(\''+
                files[i].name+'\')" class="directory">+'+
                files[i].name+'</span><br>';
      }
    }
  }
  function changedir(newpath){
    path=path+newpath+'/';
    show();
  }
  window.addEventListener('load', initiate, false);
```

Конечно, этот код не получится использовать вместо Проводника Windows, но он дает всю информацию, необходимую для построения работающего представления файловой системы в браузере. Давайте проанализируем его шаг за шагом.

Функция `initiate()` делает то же самое, что и в предыдущих примерах кода: открывает или создает файловую систему и в случае успешного завершения этой операции вызывает функцию `createhd()`. Помимо объявления переменной `hd` для хранения ссылки на файловую систему функция `createhd()` инициализирует переменную `path` со значением, равным пустой строке (эта переменная представляет корневой каталог), и вызывает функцию `show()` для вывода на экран списка файлов — это происходит, как только завершается загрузка приложения.

Переменная `path` используется в оставшейся части приложения для хранения текущего пути, с которым работает пользователь. Например, в коде из листинга 12.10 мы немного изменили функцию `create()`, для того чтобы в ней учитывалось это значение. Теперь каждый раз, когда с формы приходит новое имя каталога, к этому имени прибавляется путь и файл создается в текущем каталоге.

Как уже говорилось, для того чтобы вывести список записей, нужно сначала открыть каталог для чтения. Используя метод `getDirectory()` в функции `show()`, мы открываем текущий каталог (соответствующий значению переменной `path`), а затем, если операция завершается успешно, отправляем ссылку на этот каталог функции `readdir()`. Эта функция сохраняет ссылку в переменной `dir`, создает новый объект `DirectoryReader` для текущего каталога и извлекает список записей с помощью метода `readEntries()`.

В функции `readdir()` для правильной организации данных и ограничения контекста применяются анонимные функции. Сначала функция `createReader()` создает объект `DirectoryReader` для каталога, представленного переменной `dir`. Затем динамически создается новая функция под названием `read()`. Она с помощью метода `readEntries()` выполняет считывание записей. Метод `readEntries()` считывает записи блоками, поэтому его необходимо вызывать несколько раз, для того чтобы извлечь все записи, существующие в каталоге. Именно для этого и предназначена функция `read()`. Процесс происходит следующим образом: впервые вызов функции `read()` происходит в конце функции `readdir()`. Внутри функции `read()` мы вызываем метод `readEntries()`. В случае успешного завершения операции этот метод вызывает еще одну анонимную функцию для получения объекта `files` и проверки его содержимого. Если

объект не пуст, то происходит вызов функции `list()`, которая выводит на экран список уже считанных записей, а функция `read()` выполняется еще раз для проверки очередного блока записей. Таким образом, эта функция вызывает саму себя до тех пор, пока не будут возвращены все существующие записи.

Функция `list()` выполняет задачу формирования списка записей (файлов и каталогов) на экране. Она принимает объект `files` и проверяет характеристики каждого объекта, используя два других важных свойства интерфейса `Entry`: `isFile` и `isDirectory`. Как ясно из их названий, эти свойства содержат булевы значения, указывающие, какой объект представляет соответствующая запись, файл или каталог. После проверки записи на экран выводится информация о ней, для этого используется свойство `name`.

Мы по-разному отображаем информацию о файлах и каталогах. Если обнаруживаем, что очередная запись соответствует каталогу, то используем для ее представления на экране элемент `<span>` с обработчиком события `onclick`, который при щелчке на этом элементе вызывает функцию `changedir()`. Назначение этой функции — изменение текущего пути. Она получает имя каталога, добавляет каталог к имеющемуся пути и вызывает функцию `show()` для обновления списка записей на экране. Таким образом, мы можем открывать каталоги, чтобы просматривать их содержимое, всего одним щелчком мыши — так же, как и в любых других файловых менеджерах.

В этом примере мы не учитываем возможности возвращения на предыдущий уровень. Реализовать ее позволяет еще один метод из интерфейса `Entry`: `getParent(функция для успешного завершения, функция для завершения с ошибкой)`. Этот метод возвращает объект `Entry` для каталога, содержащего выбранную запись. Получив объект `Entry`, вы можете считать его свойства и извлечь всю необходимую информацию о родительском каталоге вышеупомянутой записи.

Метод `getParent()` работает очень просто и понятно. Предположим, мы создали дерево каталогов следующего вида: pictures/myvacations. Пользователь в данный момент видит на экране содержимое каталога myvacations. Для того чтобы обеспечить возможность возврата к каталогу pictures, можно в HTML-документе добавить ссылку с обработчиком события `onclick`, вызывающим функцию изменения текущего пути. Подобная функция могла бы выглядеть, как в листинге 12.11.

Листинг 12.11. Возвращение в родительский каталог

```
function goback(){
  hd.getDirectory(path,null,function(dir){
    dir.getParent(function(parent){
      path=parent.fullPath;
      show();
    }, showerror);
  },showerror);
}
```

Функция `goback()` из листинга 12.11 меняет значение переменной `path`, сохраняя в ней название каталога, являющегося родительским по отношению к текущему. Сначала мы извлекаем ссылку на текущий каталог, применяя для этого метод `getDirectory()`. Если операция завершается успешно, этот метод вызывает анонимную функцию. В анонимной функции с помощью метода `getParent()` мы находим родительский объект для каталога, хранящегося в переменной `dir` (то есть текущего каталога). Если операция выполняется успешно, данный метод вызывает другую анонимную функцию, которая принимает родительский объект и присваивает значение его свойства `fullPath` переменной, представляющей текущий путь. В конце также происходит вызов функции `show()`, для того чтобы она обновила представление на экране (она выводит список записей для нового текущего пути).

Разумеется, у этого приложения огромный потенциал для усовершенствования, но это будет вашим домашним заданием.

САМОСТОЯТЕЛЬНО

Добавьте функцию из листинга 12.11 к коду в листинге 12.10 и создайте в HTML-документе ссылку для вызова этой функции (например, вернуться).

Обработка файлов

Ранее уже говорилось о том, что интерфейс `Entry` предоставляет набор свойств и методов для получения информации о файлах и выполнении с ними различных действий. Большинство этих свойств мы уже применяли в предыдущих примерах. Мы использовали свойства `isFile` и `isDirectory` для проверки состояния записи, а значения свойств `name`,

`fullPath` и `filesystem` — для вывода информации на экран. Метод `getParent()`, рассмотренный в предыдущем примере кода, также входит в этот интерфейс. Нам осталось познакомиться еще с несколькими методами, предназначенными для выполнения стандартных операций над файлами и каталогами. Применяя эти методы, мы можем перемещать, копировать и удалять записи точно так же, как это делается в любом настольном приложении:

- `moveTo(parent, new name, функция для успешного завершения, функция для завершения с ошибкой)`. Перемещает запись в другое место в той же файловой системе. Если атрибут `new name` присутствует, то имя записи меняется на значение этого атрибута;
- `copyTo(parent, new name, функция для успешного завершения, функция для завершения с ошибкой)`. Создает копию записи в другом месте в той же файловой системе. Если атрибут `new name` присутствует, то новой записи присваивается имя, соответствующее значению этого атрибута;
- `remove`. Удаляет файл или пустой каталог (для удаления каталога со всем его содержимым необходимо использовать упомянутый ранее метод `removeRecursively()`).

Для тестирования этих методов нам понадобится новый шаблон. Чтобы упростить код, создадим только два поля ввода, в которых нужно будет указывать исходное и целевое местоположение для каждой операции (листинг 12.12).

Листинг 12.12. Новый шаблон для обработки файлов

```
<!DOCTYPE html>
<html lang="ru">
<head>
  <title>Файловый API</title>
  <link rel="stylesheet" href="file.css">
  <script src="file.js"></script>
</head>
<body>
  <section id="formbox">
    <form name="form">
      <p>Источник:<br><input type="text" name="origin"
         id="origin" required></p>
      <p>Цель:<br><input type="text" name="destination"
         id="destination" required></p>
```

```
      <p><input type="button" name="fbutton" id="fbutton"
          value="Выполнить"></p>
    </form>
  </section>
  <section id="databox"></section>
</body>
</html>
```

Перемещение

Метод `moveTo()` требует в качестве входных параметров два объекта `Entry`: первый соответствует файлу, над которым выполняется операция, а второй представляет собой каталог, в который файл будет перемещен. Таким образом, сначала нужно создать ссылку на файл, используя `getFile()`, а затем получить ссылку на целевой каталог с помощью `getDirectory()`. После этого можно вызывать `moveTo()`, передавая ему эту информацию.

Листинг 12.13. Перемещение файлов

```
function initiate(){
  databox=document.getElementById('databox');
  var button=document.getElementById('fbutton');
  button.addEventListener('click', modify, false);

  window.webkitRequestFileSystem(window.PERSISTENT, 5*1024*1024,
         createhd, showerror);
}
function createhd(fs){
  hd=fs.root;
  path='';
  show();
}
function showerror(e){
  alert('Ошибка: '+e.code);
}
function modify(){
  var origin=document.getElementById('origin').value;
  var destination=document.getElementById('destination').value;

  hd.getFile(origin,null,function(file){
    hd.getDirectory(destination,null,function(dir){
```

продолжение ⇗

Листинг 12.13 *(продолжение)*

```
        file.moveTo(dir,null,success,showerror);
      },showerror);
    },showerror);
  }
  function success(){
    document.getElementById('origin').value='';
    document.getElementById('destination').value='';
    show();
  }
  function show(){
    databox.innerHTML='';
    hd.getDirectory(path,null,readdir,showerror);
  }
  function readdir(dir){
    var reader=dir.createReader();
    var read=function(){
      reader.readEntries(function(files){
        if(files.length){
          list(files);
          read();
        }
      }, showerror);
    }
    read();
  }
  function list(files){
    for(var i=0; i<files.length; i++) {
      if(files[i].isFile) {
        databox.innerHTML+=files[i].name+'<br>';
      }else if(files[i].isDirectory){
        databox.innerHTML+='<span onclick="changedir(\''+
                files[i].name+'\')" class="directory">+'+
                files[i].name+'</span><br>';
      }
    }
  }
  function changedir(newpath){
    path=path+newpath+'/';
    show();
  }
  window.addEventListener('load', initiate, false);
```

Для создания или открытия файловой системы и вывода на экран списка записей мы использовали функции из предыдущих примеров.

Единственная новая функция в листинге 12.13 — это `modify()`. Она принимает значения из полей Источник и Цель нашей формы и с учетом этой информации открывает сначала файл-источник, а затем, в случае успеха, — целевой каталог. Если обе операции завершаются успешно, то для объекта `file` вызывается метод `moveTo()` и файл перемещается в каталог, путь к которому хранится в переменной `dir`. Если и эта операция выполняется успешно, то происходит вызов функции `success()` для очистки полей формы, а затем функция `show()` обновляет на экране список записей.

САМОСТОЯТЕЛЬНО

Для тестирования этого примера вам потребуются HTML-файл с шаблоном из листинга 12.12, CSS-файл, который мы использовали с самого начала главы, и файл под названием file.js с кодом из листинга 12.13 (не забудьте перед проверкой загрузить файлы на свой сервер). Создайте в своей файловой системе файлы и каталоги, для того чтобы вам было с чем работать. Для этого можно применить предыдущие примеры кода. Затем воспользуйтесь формой из последнего HTML-документа: укажите имя файла для перемещения (полный путь, начиная с корня) и имя каталога, в который нужно переместить файл (если этот каталог находится в корне файловой системы, то косую черту добавлять не требуется, достаточно имени каталога).

Копирование

Разумеется, единственное различие между методами `moveTo()` и `copyTo()` заключается в том, что второй сохраняет исходный файл. Для того чтобы применить метод `copyTo()`, нужно только лишь поменять название метода в коде из листинга 12.13. Готовая функция `modify()` будет выглядеть, как в листинге 12.14.

Листинг 12.14. Копирование файлов

```
function modify(){
  var origin=document.getElementById('origin').value;
  var destination=document.getElementById('destination').value;

  hd.getFile(origin,null,function(file){
    hd.getDirectory(destination,null,function(dir){
      file.copyTo(dir,null,success,showerror);
    },showerror);
  },showerror);
}
```

САМОСТОЯТЕЛЬНО

Замените функцию modify() в листинге 12.13 аналогичной функцией из листинга 12.14 и откройте шаблон из листинга 12.12, чтобы протестировать пример в браузере. Скопируйте файл, повторив процедуру, которую выполняли для перемещения файла. Вставьте путь к исходному файлу в поле «Источник», а путь к каталогу, куда файл должен быть скопирован, — в поле «Цель».

Удаление

Удалять файлы и каталоги проще, чем перемещать и создавать их. Нам всего лишь нужно получить объект `Entry` для файла или каталога, который мы собираемся удалить, и применить к этой ссылке метод `remove()`.

Листинг 12.15. Удаление файлов и каталогов

```
function modify(){
  var origin=document.getElementById('origin').value;
  var origin=path+origin;
  hd.getFile(origin,null,function(entry){
    entry.remove(success,showerror);
  },showerror);
}
```

Код из листинга 12.15 извлекает значение поля-источника из данных формы. Это значение вместе со значением переменной `path` представляет собой путь к файлу, который мы собираемся удалить. Таким образом, используя метод `getFile()`, мы создаем объект `Entry` для вышеупомянутого файла, а затем применяем к нему метод `remove()`.

САМОСТОЯТЕЛЬНО

Замените функцию modify() в коде из листинга 12.13 новой функцией из листинга 12.15. На этот раз вам нужно будет заполнить только поле Источник, указав путь к удаляемому файлу.

Удаление каталога незначительно отличается от удаления файла: объект `Entry` приходится создавать с использованием `getDirectory()`, однако метод `remove()` менять не требуется, он работает так же, как всегда. При этом все же необходимо учитывать один специфический момент: если каталог не пуст, то метод `remove()` вернет ошибку. Для того чтобы удалить каталог со всем его содержимым, необходимо применить другой метод, который мы упоминали чуть раньше в этой главе. Этот метод называется `removeRecursively()`.

Листинг 12.16. Удаление непустых каталогов

```
function modify(){
  var destination=document.getElementById('destination').value;
  hd.getDirectory(destination,null,function(entry){
    entry.removeRecursively(success,showerror);
  },showerror);
}
```

В функции из листинга 12.16 мы использовали значение поля `destination` для указания каталога, который будет удален. Метод `removeRecursively()` удаляет каталог со всем его содержимым за один проход и в случае успешного завершения операции вызывает функцию `success()`.

САМОСТОЯТЕЛЬНО

Функции modify() в примерах из листингов 12.14–12.16 полностью заменяют собой аналогичную функцию из листинга 12.13. Для тестирования этих примеров откройте код из листинга 12.13, поменяйте функцию modify() на тот ее вариант, который вы желаете протестировать, и откройте в браузере шаблон из листинга 12.12. В зависимости от того, какой метод тестируете, вам понадобится ввести в поля формы одно или два значения.

ВНИМАНИЕ

Если при тестировании этих примеров ваш браузер возвращает ошибку, попробуйте воспользоваться браузером Chromium (*http://www.chromium.org*). Коды, использованные в примерах для этой части API, были проверены в последней из доступных на тот момент версий Google Chrome.

Содержимое файла

Помимо основного файлового и API расширения, рассмотренного ранее, существует еще одно важное расширение под названием API File: Writer (Запись файлов). В этой спецификации объявляются новые интерфейсы для записи и добавления содержимого в файлы. Для успешной работы с этим API расширения необходимо комбинировать его методы с методами из остальных составляющих файлового API и использовать общие объекты.

> **ВНИМАНИЕ**
>
> Текущий способ интеграции спецификаций, составляющих общий файловый API, вызывает определенные вопросы относительно того, какому API должны принадлежать те или иные интерфейсы. Для того чтобы получить более подробную информацию, изучите ссылки для каждого из рассмотренных в книге API на нашем веб-сайте или посетите веб-сайт консорциума W3C по адресу *http://www.w3.org*.

Запись содержимого

Чтобы записать содержимое в файл, нужно создать объект `FileWriter`. Этот объект возвращается методом `createWriter()` интерфейса `FileEntry`, дополняющего интерфейс `Entry` и предоставляющего два метода для работы с файлами:

- `createWriter(функция для успешного завершения, функция для завершения с ошибкой)`. Возвращает объект `FileWriter`, связанный с выбранной записью;
- `file(функция для успешного завершения, функция для завершения с ошибкой)`. Этот метод мы будем применять позже для считывания содержимого файла. Он создает объект `File` (аналогичный тому, который возвращают элемент `<input>` и операция перетаскивания), связанный с выбранной записью.

У объекта `FileWriter`, возвращаемого методом `createWriter()`, есть несколько собственных методов, свойств и событий, упрощающих процесс добавления содержимого в файл:

- `write(data)`. Именно этот метод занимается непосредственной записью данных в файл. Содержимое передается ему в атрибуте `data` в форме бинарного блока;
- `seek(offset)`. Этот метод задает позицию в файле, начиная с которой добавляется содержимое. Значение параметра `offset` необходимо определять в байтах;
- `truncate(size)`. Этот метод устанавливает новую длину файла, равную значению атрибута `size` (в байтах);
- `position`. Это свойство возвращает позицию в файле, в которой будет выполняться очередная операция записи содержимого. Для нового файла значение позиции равно 0, а для файла, в который уже записывалось содержимое или к которому ранее применялся метод `seek()`, значение позиции отличается от нуля;
- `length`. Это свойство возвращает длину файла;
- `writestart`. Это событие срабатывает, когда операция записи начинается;
- `progress`. Это событие срабатывает периодически и информирует о прогрессе операции;
- `write`. Это событие срабатывает после того, как запись данных завершается;
- `abort`. Это событие срабатывает в случае прерывания процесса;
- `error`. Это событие срабатывает, если происходит ошибка;
- `writeend`. Это событие срабатывает по завершении процесса.

Для подготовки содержимого, которое будет добавлено в файл, необходимо создать еще один объект. Конструктор `BlobBuilder()` возвращает объект `BlobBuilder`, предлагающий следующие методы:

- `getBlob(type)`. Возвращает содержимое объекта `BlobBuilder` в форме бинарного блока. С помощью этого метода удобно создавать бинарные блоки для метода `write()`;
- `append(data)`. Присоединяет значение параметра `data` к объекту `BlobBuilder`. Атрибут `data` может представлять собой бинарный блок, объект `ArrayBuffer` или простой текст.

В HTML-документ из листинга 12.17 мы добавили второе поле, в которое нужно будет вводить текст, представляющий собой содержимое файла. Будем использовать код из данного листинга для примеров далее в этой главе.

Листинг 12.17. Шаблон для определения имени файла и его содержимого

```
<!DOCTYPE html>
<html lang="ru">
<head>
  <title>Файловый API</title>
  <link rel="stylesheet" href="file.css">
  <script src="file.js"></script>
</head>
<body>
  <section id="formbox">
    <form name="form">
      <p>Файл:<br><input type="text" name="myentry"
         id="myentry" required></p>
      <p>Текст:<br><textarea name="mytext" id="mytext" required>
                  </textarea></p> <
      p><input type="button" name="fbutton" id="fbutton"
         value="Выполнить"></p>
    </form>
  </section>
  <section id="databox">
    Информация недоступна
  </section>
</body>
</html>
```

Для выполнения операции записи откроем файловую систему, извлечем ссылку на файл или создадим новый с помощью `getFile()` и вставим в открытый файл предоставленные пользователем значения. Будем пользоваться двумя разными функциями: `writefile()` и `writecontent()`.

> **ВНИМАНИЕ**
>
> Для того чтобы вам было проще учиться, мы постарались не делать файл излишне сложным. Однако вы всегда можете прибегнуть к помощи анонимных функций, если хотите ограничить контекст (поместить все внутрь одной функции), или воспользоваться возможностями объектно-ориентированного программирования для создания более продвинутых и масштабируемых реализаций.

Листинг 12.18. Запись содержимого

```
function initiate(){
  databox=document.getElementById('databox');
  var button=document.getElementById('fbutton');
  button.addEventListener('click', writefile, false);
  window.webkitRequestFileSystem(window.PERSISTENT, 5*1024*1024,
         createhd, showerror);
}
function createhd(fs){
  hd=fs.root;
}
function showerror(e){
  alert('Ошибка: '+e.code);
}
function writefile(){
  var name=document.getElementById('myentry').value;
  hd.getFile(name, {create: true,
     exclusive: false},function(entry){
    entry.createWriter(writecontent, showerror);
  }, showerror);
}
function writecontent(fileWriter) {
  var text=document.getElementById('mytext').value;
  fileWriter.onwriteend=success;
  var blob=new WebKitBlobBuilder();
  blob.append(text);
  fileWriter.write(blob.getBlob());
}
function success(){
  document.getElementById('myentry').value='';
  document.getElementById('mytext').value='';
  databox.innerHTML='Готово!';
}
window.addEventListener('load', initiate, false);
```

ВНИМАНИЕ

Так же, как в случае с методом requestFileSystem(), в текущей реализации Google Chrome с конструктором BlobBuilder() необходимо использовать браузерный префикс. Для тестирования кода в браузере в этом и последующих примерах данный конструктор следует записывать как WebKitBlobBuilder().

Когда пользователь щелкает на кнопке Выполнить, информация из полей отправляется на обработку функциям `writefile()` и `writecontent()`. Функция `writefile()` принимает значение `myentry` и с помощью `getFile()` открывает или создает файл. Возвращенный ею объект `Entry` используется в методе `createWriter()` для создания объекта `FileWriter`. Если операция завершается успешно, происходит вызов функции `writecontent()`.

Функция `writecontent()` принимает объект `FileWriter` и записывает в файл значение из поля `mytext`. Для того чтобы текст можно было использовать в операции записи, сначала его следует преобразовать в бинарный блок. Для этого мы применяем конструктор `BlobBuilder()`. Создаем объект `BlobBuilder`, добавляем текст к этому объекту, используя метод `append()`, и извлекаем содержимое в форме бинарного блока с помощью `getBlob()`. Это дает нам информацию в формате, подходящем для записи в файл посредством метода `write()`.

Весь процесс выполняется асинхронно, то есть о статусе операции нам непрерывно сообщают события. В функции `writecontent()` мы прослушиваем только событие `writeend` (используя обработчик события `onwriteend`). В случае успеха вызываем функцию `success()` и выводим на экране строку «Готово!». Можно также контролировать прогресс выполнения задачи или проверять ошибки, прослушивая остальные события, предоставляемые объектом `FileWriter`.

САМОСТОЯТЕЛЬНО

Скопируйте шаблон из листинга 12.17 в новый HTML-файл (для этого шаблона можно использовать CSS-стили, которые мы создали ранее в листинге 12.2). Создайте JavaScript-файл с именем file.js и поместите в него код из листинга 12.18. Откройте HTML-документ в своем браузере, укажите имя файла и текст, который хотели бы в него вставить. Если на экране появится строка «Готово!», значит, операция была выполнена успешно.

Добавление содержимого

Поскольку мы не указали позицию, с которой должна начинаться запись содержимого, предыдущий код просто записывает бинарный блок в начало файла. Для того чтобы указать конкретную позицию или присоеди-

нить содержимое к концу уже существующего файла, нужно воспользоваться методом `seek()`.

Листинг 12.19. Добавление нового содержимого к имеющемуся

```
function writecontent(fileWriter) {
  var text=document.getElementById('mytext').value;
  fileWriter.seek(fileWriter.length);
  fileWriter.onwriteend=success;
  var blob=new WebKitBlobBuilder();
  blob.append(text);
  fileWriter.write(blob.getBlob());
}
```

САМОСТОЯТЕЛЬНО

Замените функцию writecontent() в листинге 12.18 новым вариантом из листинга 12.19 и откройте HTML-файл в своем браузере. Заполните поля формы: укажите то же имя файла, которое вы использовали в предыдущий раз, и текст, который необходимо дописать в конце файла.

Функция из листинга 12.19 — это усовершенствованный вариант функции `writecontent()` из предыдущего примера. Теперь она включает в себя метод `seek()`, с помощью которого позиция записи переносится в конец файла. Таким образом, метод `write()` добавляет новое содержимое в конец файла, а не записывает его на место уже существующих данных.

Для вычисления позиции конца файла в байтах мы воспользовались свойством `length`. Остальной код ничем не отличается от кода в листинге 12.18.

Считывание содержимого

Настало время попробовать считать то, что мы только что записали в файл, применяя техники из основной спецификации файлового API, которые мы изучили в начале главы. Для чтения и извлечения содержимого файла мы будем пользоваться конструктором `FileReader()` и методами считывания, такими как `readAsText()`.

Листинг 12.20. Считывание файла из файловой системы

```
function initiate(){
  databox=document.getElementById('databox');
  var button=document.getElementById('fbutton');
  button.addEventListener('click', readfile, false);

  window.webkitRequestFileSystem(window.PERSISTENT, 5*1024*1024,
         createhd, showerror);
}
function createhd(fs){
  hd=fs.root;
}
function showerror(e){
  alert('Ошибка: '+e.code);
}
function readfile(){
  var name=document.getElementById('myentry').value;
  hd.getFile(name, {create: false}, function(entry) {
    entry.file(readcontent, showerror);
  }, showerror);
}
function readcontent(file){
  databox.innerHTML='Имя: '+file.name+'<br>';
  databox.innerHTML+='Тип: '+file.type+'<br>';
  databox.innerHTML+='Размер: '+file.size+' bytes<br>';

  var reader=new FileReader();
  reader.onload=success;
  reader.readAsText(file);
}
function success(e){
  var result=e.target.result;
  document.getElementById('myentry').value='';
  databox.innerHTML+='Содержимое: '+result;
}
window.addEventListener('load', initiate, false);
```

Методы, предоставляемые интерфейсом `FileReader` для считывания содержимого файла, такие как `readAsText()`, принимают в качестве атрибута бинарный блок или объект `File`. Объект `File` представляет собой считываемый файл и генерируется элементом `<input>` или операцией перетаскивания. Как уже говорилось, в интерфейсе `FileEntry`

предусмотрена возможность создавать объекты такого типа — для этого предназначен метод `file()`.

Когда пользователь щелкает на кнопке Выполнить, функция `readfile()` принимает значение из поля `myentry` и открывает файл с указанным именем с помощью метода `getFile()`. Если метод выполняется успешно, то возвращается объект `Entry`. Он сохраняется в переменной `entry` и используется для генерирования объекта `File` в методе `file()`.

Поскольку объект `File` — это точно такой же объект, как и генерируемый элементом `<input>` и операцией перетаскивания, мы можем использовать те же свойства, которые применяли раньше. Например, мы вполне можем показать основную информацию о файле еще до того, как начнется его считывание. Мы выводим на экран значения базовых свойств и считываем содержимое файла в функции `readcontent()`.

Процесс считывания аналогичен тому, который закодирован в листинге 12.3. Объект `FileReader` создается с помощью конструктора `FileReader()`, и мы регистрируем обработчик события `onload`, для того чтобы в случае успешного завершения он вызывал функцию `success()`. Непосредственное считывание содержимого файла происходит в методе `readAsText()`.

В функции `success()`, вместо того чтобы показывать на экране строку, как мы делали раньше, отображаем содержимое файла. Для этого считываем значение свойства `result` объекта `FileReader` и выводим его в объекте `databox`.

САМОСТОЯТЕЛЬНО

Код из листинга 12.20 принимает только одно значение — из поля myentry. Откройте HTML-файл с последним шаблоном в своем браузере и введите в поле название файла, который нужно прочитать. Это должен быть файл, который вы создали ранее, в противном случае система вернет сообщение об ошибке (create: false). Если имя файла указано правильно, то вы увидите на экране информацию о файле и его содержимое.

Файловая система в реальной жизни

Для того чтобы хорошо понять потенциал изученных концепций, всегда полезно изучить пример из реальной жизни. В завершение главы мы

создадим приложение, объединяющее несколько техник файлового API с возможностями манипулирования изображениями, предоставляемыми API Canvas (Холст).

В этом примере мы считываем несколько файлов изображений и выводим эти изображения на холсте, выбирая местоположение случайным образом. Все изменения элемента `canvas` сохраняются в файл, для того чтобы их можно было считать и использовать позже. Таким образом, каждый раз, когда вы запускаете приложение, на экране отображается результат предыдущего выполнения.

HTML-документ для примера аналогичен первому шаблону из данной главы. Однако на этот раз он включает в себя элемент `canvas` внутри `databox` (листинг 12.21).

Листинг 12.21. Новый шаблон с элементом <canvas>

```
<!DOCTYPE html>
<html lang="ru">
<head>
  <title>Файловый API</title>
  <link rel="stylesheet" href="file.css">
  <script src="file.js"></script>
</head>
<body>
  <section id="formbox">
    <form name="form">
      <p>Изображения:<br><input type="file" name="myfiles"
         id="myfiles" multiple></p>
    </form>
  </section>
  <section id="databox">
    <canvas id="canvas" width="500" height="350"></canvas>
  </section>
</body>
</html>
```

Код для нашего примера включает в себя уже знакомые вам методы и техники программирования, однако разобраться в сочетании спецификаций поначалу может быть довольно сложно. Давайте сначала познакомимся с кодом, а затем проанализируем его шаг за шагом.

Листинг 12.22. Приложение на основе холста, использующее возможности файлового API

```
function initiate(){
  var elem=document.getElementById('canvas');
  canvas=elem.getContext('2d'); var
  myfiles=document.getElementById('myfiles');
  myfiles.addEventListener('change', process, false);
  window.webkitRequestFileSystem(window.PERSISTENT, 5*1024*1024,
createhd, showerror);
}
function createhd(fs){
  hd=fs.root;
  loadcanvas();
}
function showerror(e){
  alert('Ошибка: '+e.code);
}
function process(e){
  var files=e.target.files;
  for(var f=0;f<files.length;f++){
    var file=files[f];
    if(file.type.match(/image.*/i)){
      var reader=new FileReader();
      reader.onload=show;
      reader.readAsDataURL(file);
    }
  }
}
function show(e){
  var result=e.target.result;
  var image=new Image();
  image.src=result;
  image.addEventListener("load", function(){
    var x=Math.floor(Math.random()*451);
    var y=Math.floor(Math.random()*301);
    canvas.drawImage(image,x,y,100,100);
    savecanvas();
  }, false);
}
function loadcanvas(){
  hd.getFile('canvas.dat', {create: false}, function(entry) {
```

продолжение ⇨

Листинг 12.22 *(продолжение)*

```
    entry.file(function(file){
      var reader=new FileReader();
      reader.onload=function(e){
        var image=new Image();
        image.src=e.target.result;
        image.addEventListener("load", function(){
          canvas.drawImage(image,0,0);
        }, false);
      };
      reader.readAsBinaryString(file);
    }, showerror);
  }, showerror);
}
function savecanvas(){
  var elem=document.getElementById('canvas');
  var info=elem.toDataURL();
  hd.getFile('canvas.dat', {create: true, exclusive: false},
     function(entry) {
    entry.createWriter(function(fileWriter){
      var blob=new WebKitBlobBuilder();
      blob.append(info);
      fileWriter.write(blob.getBlob());
    }, showerror);
  }, showerror);
}
window.addEventListener('load', initiate, false);
```

В этом примере мы работаем с двумя API: API File (Файл) с расширениями и API Canvas (Холст). В функции `initiate()` происходит инициализация обоих API. Сначала с помощью `getContext()` генерируется контекст рисования для холста, а затем посредством `requestFileSystem()` запрашивается файловая система.

Как всегда, как только подготовка файловой системы завершается, мы вызываем функцию `createhd()`, инициализируя переменную `hd` и присваивая ей ссылку на корень файловой системы. На этот раз в конец функции `createhd()` мы добавили вызов новой функции, которая загружает файл с изображением, сгенерированным при прошлом запуске приложения.

Однако давайте сначала все же посмотрим, как это изображение генерируется. Когда пользователь выбирает новые файлы изображений в форме,

содержащейся в HTML-документе, на элементе `<input>` срабатывает событие `change` и происходит вызов функции `process()`. Эта функция принимает файлы, перечисленные в поле ввода, извлекает из массива все объекты `File`, проверяет каждый файл на соответствие формату изображения и считывает содержимое всех записей в методе `readAsDataURL()`. Результирующее значение возвращается в формате data:url.

Как видите, функция `process()` считывает файлы индивидуально, по одному за раз. Если операция завершается успешно, то срабатывает событие `load` и для очередного изображения вызывается функция `show()`. Таким образом, эта функция последовательно обрабатывает каждое изображение, которое выбирает пользователь.

Функция `show()` принимает данные из объекта `reader`, создает с помощью конструктора `Image()` новый объект изображения и в строке `image.src=result` объявляет источником этого изображения данные, считанные из очередного файла.

Работая с изображениями, всегда необходимо принимать во внимание время, необходимое на загрузку файлов. По этой причине после объявления нового источника для объекта изображения добавляем слушатель события `load`. Мы хотим быть абсолютно уверены в том, что изображение полностью загрузилось, прежде чем приступать к обработке его данных. После срабатывания события `load` (что означает завершение загрузки изображения) выполняется анонимная функция, объявленная в методе `addEventListener()`. Эта функция вычисляет для изображения случайную позицию и выводит его на холст посредством метода `drawImage()`. Для каждого изображения определяется фиксированный размер 100×100 пикселов (для того чтобы понять, как проходит процесс обработки, изучите код функции `show()` в листинге 12.22).

После того как изображение нарисовано, происходит вызов функции `savecanvas()`. Эта функция сохраняет состояние холста после каждого изменения, таким образом, при каждом последующем запуске приложения на экране восстанавливается картинка, сохраненная в прошлый раз. Для возвращения содержимого холста в формате data:url используется метод из API Canvas (Холст) под названием `toDataURL()`. Затем в функции `savecanvas()` выполняется последовательность операций по обработке данных. Сначала данные в формате data:url, сгенерированные на основе содержимого холста, сохраняются в переменной `info`. Затем файл `canvas.dat` создается (если он еще не существует) и открывается с помощью метода `getFile()`. Если операция `getFile()` выполняется успешно, то эта запись файловой системы передается анонимной функции, где в методе `createWriter()` создается объект `FileWriter`. Если

объект успешно создается, то этот метод также вызывает анонимную функцию, которая присоединяет значение переменной `info` к объекту `BlobBuilder`, а затем обновленный бинарный блок в методе write() записывается в файл.

ВНИМАНИЕ

В этот раз мы не прослушиваем никакие события объекта FileWriter, так как в случае успеха или ошибки не собираемся предпринимать никаких специальных действий. Однако вы всегда можете воспользоваться функциональностью событий для вывода на экран статуса операции или для обеспечения абсолютного контроля над каждым шагом процесса.

Итак, возвращаемся к функции `loadcanvas()`. Как говорилось ранее, эта функция вызывается внутри функции `createhd()` в момент запуска приложения. Ее назначение — загрузить файл с ранее сохраненными данными изображения и вывести эту картинку на холст. Вы уже знаете, о каком файле идет речь и как он создается, поэтому давайте рассмотрим последовательность действий, выполняемую данной функцией.

Функция `createhd()` вызывает функцию `loadcanvas()`, как только становится понятно, что файловая система готова к использованию. Задача функции `loadcanvas()` (как понятно из ее названия) заключается в том, чтобы загрузить файл canvas.dat, получить данные в формате data:url, сгенерированные после предыдущей модификации элемента холста, и вывести их на холст. Если такого файла не существует, то метод `getFile()` возвращает ошибку, если же файл найден, то метод вызывает анонимную функцию, которая принимает запись файловой системы и с помощью метода `file()` генерирует на ее основе объект `File`. Этот метод в случае успешного выполнения также вызывает анонимную функцию, которая считывает содержимое файла как двоичную строку (посредством метода `readAsBinaryString()`). Данные, которые мы получаем из файла, представляют собой строку в формате data:url, которую необходимо объявить источником изображения, чтобы его можно было вывести на холст. Таким образом, внутри анонимной функции, которую вызывает событие `load`, мы создаем объект изображения, объявляем data:url из файла источником этого изображения и, когда изображение полностью загружено, выводим его на холст в методе `drawImage()`.

На экране все это выглядит очень просто: изображения, выбранные с помощью элемента `<input>`, выводятся на холсте в случайных точках,

и результирующая картинка сохраняется в файле. Если закрыть и заново открыть браузер, то при запуске приложения холст восстанавливается и мы снова видим результат предыдущей работы. Возможно, это не самое полезное приложение в мире, но вы ведь поняли, насколько велик потенциал файлового API?

САМОСТОЯТЕЛЬНО

Применяя возможности API Drag and Drop (Перетаскивание), файлы изображений можно перетаскивать на холст, а не загружать через элемент <input>. Попробуйте объединить код из листинга 12.22 с примерами кода из главы 8, чтобы попрактиковаться во взаимном интегрировании этих API.

Краткий справочник. API File (Файл)

Как и в случае с API IndexedDB (Индексированные базы данных), возможности, предоставляемые файловым API и его расширениями, организованы в несколько интерфейсов. Каждый интерфейс предоставляет методы, свойства и события, которые в сочетании с функциональностью остальных API расширения предлагают множество альтернативных способов создания, считывания и обработки файлов. Далее вы найдете описание всех возможностей, изученных в этой главе, в порядке, соответствующем официальной спецификации.

ВНИМАНИЕ

В этом кратком справочнике описаны только наиболее важные аспекты интерфейсов. Для получения полной спецификации зайдите на наш веб-сайт и проследуйте по ссылкам для этой главы.

Интерфейс Blob (API File (Файл))

Этот интерфейс предоставляет свойства и методы для работы с бинарными блоками (blob). Он наследуется интерфейсом `File`:

- `size`. Это свойство представляет собой размер бинарного блока или файла в байтах;
- `type`. Это свойство представляет собой тип мультимедиа данного бинарного блока или файла;
- `slice(start, length, type)`. Этот метод возвращает части бинарного блока или файла, на которые указывают значения атрибутов `start` (начальная позиция) и `length` (длина фрагмента) (значения определяются в байтах).

Интерфейс File (API File (Файл))

Этот интерфейс представляет собой расширение интерфейса Blob и предназначен для обработки файлов.

- `name`. Это свойство представляет имя файла.

Интерфейс FileReader (API File (Файл))

Этот интерфейс предоставляет специальные методы, свойства и события для считывания файлов и бинарных блоков в память:

- `readAsArrayBuffer(file)`. Этот метод возвращает содержимое файла или бинарного блока в форме объекта `ArrayBuffer`;
- `readAsBinaryString(file)`. Этот метод возвращает содержимое файла или бинарного блока в форме двоичной строки;
- `readAsText(file)`. Этот метод интерпретирует содержимое файла или бинарного блока и возвращает его в форме текста;
- `readAsDataURL(file)`. Этот метод возвращает содержимое файла или бинарного блока в форме URL данных (data:url);
- `abort()`. Этот метод позволяет прервать процесс считывания данных;
- `result`. Это свойство представляет собой данные, возвращенные методами считывания;
- `loadstart`. Это событие срабатывает в начале операции считывания;
- `progress`. Это событие срабатывает периодически и информирует о статусе операции считывания;
- `load`. Это событие срабатывает по завершении операции считывания;
- `abort`. Это событие срабатывает, если операция считывания прерывается;

- `error`. Это событие срабатывает в случае ошибки или сбоя;
- `loadend`. Это событие срабатывает по завершении процесса — как в случае успеха, так и в случае сбоя.

Интерфейс LocalFileSystem (API File: Directories and System (Каталоги и система))

Этот интерфейс предназначен для инициализации файловой системы для конкретного приложения:

- `requestFileSystem(type, size, функция для успешного завершения, функция для завершения с ошибкой)`. Этот метод запрашивает инициализацию файловой системы, настроенной в соответствии со значениями его атрибутов. Атрибут `type` может принимать одно из двух значений: `TEMPORARY` или `PERSISTENT`. Размер (атрибут `size`) следует задавать в байтах.

Интерфейс FileSystem (API File: Directories and System (Каталоги и система))

Этот интерфейс предоставляет информацию о файловой системе:

- `name`. Это свойство представляет собой имя файловой системы;
- `root`. Это свойство представляет собой корневой каталог файловой системы.

Интерфейс Entry (API File: Directories and System (Каталоги и система))

Этот интерфейс предоставляет методы и свойства для обработки записей (файлов и каталогов) в файловой системе:

- `isFile`. Это свойство возвращает логическое (булево) значение, указывающее, является запись файлом или нет;
- `isDirectory`. Это свойство возвращает логическое (булево) значение, указывающее, является запись каталогом или нет;
- `name`. Это свойство представляет собой имя записи;

- `fullPath`. Это свойство представляет собой полный путь от корня файловой системы до данной записи;
- `filesystem`. Это свойство содержит ссылку на файловую систему;
- `moveTo(parent, новое имя, функция для успешного завершения, функция для завершения с ошибкой)`. Этот метод перемещает запись в другое место. Атрибут `parent` представляет собой целевой каталог, в который перемещается в запись. Атрибут `новое имя`, если он указан, содержит новое имя, которое запись получит на новом месте;
- `copyTo(parent, новое имя, функция для успешного завершения, функция для завершения с ошибкой)`. Этот метод создает копию записи. Атрибут `parent` представляет собой каталог, в котором создается копия исходной записи. Атрибут `новое имя`, если он указан, позволяет задать новое имя для копии записи на новом месте;
- `remove(функция для успешного завершения, функция для завершения с ошибкой)`. Этот метод удаляет файл или пустой каталог;
- `getParent(функция для успешного завершения, функция для завершения с ошибкой)`. Этот метод возвращает ссылку на родительский объект `DirectoryEntry` для выбранной записи.

Интерфейс DirectoryEntry (API File: Directories and System (Каталоги и система))

Этот интерфейс предоставляет методы для создания и считывания файлов и каталогов:

- `createReader()`.Создает объект `DirectoryReader` для считывания записей;
- `getFile(path, options, функция для успешного завершения, функция для завершения с ошибкой)`. Создает или считывает файл, указанный в атрибуте `path`. Атрибут `options` настраивается с использованием двух флагов: `create` и `exclusive`. Первый определяет, будет ли файл создаваться, если он еще не существует, а атрибут `exclusive`, когда его значение равно `true`, заставляет метод возвращать ошибку в случае существования такого файла;
- `getDirectory(path, options, функция для успешного завершения, функция для завершения с ошибкой)`. Создает или считывает каталог, указанный в атрибуте `path`. Атрибут `options` настраивается с использованием двух флагов: `create` и `exclusive`. Первый определяет, будет ли каталог создаваться, если он еще не существует, а атрибут `exclusive`,

когда его значение равно `true`, заставляет метод возвращать ошибку в случае существования такого каталога;

- `removeRecursively(функция для успешного завершения, функция для завершения с ошибкой)`. Удаляет каталог со всем его содержимым.

Интерфейс DirectoryReader (API File: Directories and System (Каталоги и система))

Этот интерфейс предоставляет альтернативный способ получения списка записей в указанном каталоге:

- `readEntries(функция для успешного завершения, функция для завершения с ошибкой)`. Считывает блок записей в выбранном каталоге. Если ни одной записи не найдено, он возвращает значение `null`.

Интерфейс FileEntry (API File: Directories and System (Каталоги и система))

Этот интерфейс предоставляет методы для получения объекта `File` для определенного файла, а также объекта `FileWriter`, с помощью которого к файлу можно добавлять содержимое:

- `createWriter(функция для успешного завершения, функция для завершения с ошибкой)`. Создает объект `FileWriter` для записи содержимого в файл;
- `file(функция для успешного завершения, функция для завершения с ошибкой)`. Возвращает объект `File`, представляющий собой выбранный файл.

Интерфейс BlobBuilder (API File: Writer (Запись файлов))

Этот интерфейс предоставляет методы для работы с объектами бинарных блоков:

- `getBlob(type)`. Возвращает содержимое объекта `blob` в виде бинарного блока;
- `append(data)`. Присоединяет данные к объекту `blob`. В интерфейсе предусмотрено три метода `append()`, позволяющих присоединять данные как текст, бинарный блок или объект `ArrayBuffer`.

Интерфейс FileWriter (API File: Writer (Запись файлов))

Интерфейс `FileWriter` представляет собой расширение интерфейса `FileSaver`. Последний мы в этой главе не рассматриваем, но перечисленные далее события входят в него:

- `position`. Это свойство представляет собой текущую позицию, в которой будет выполнена очередная операция записи;
- `length`. Это свойство представляет собой длину файла в байтах;
- `write(blob)`. Этот метод записывает содержимое в файл;
- `seek(offset)`. Этот метод устанавливает новую позицию, в которой будет выполнена очередная операция записи;
- `truncate(size)`. Этот метод меняет длину файла на значение, переданное в атрибуте `size` (в байтах);
- `writestart`. Это событие срабатывает в начале операции записи;
- `progress`. Это событие срабатывает периодически и информирует о статусе операции записи;
- `write`. Это событие срабатывает при завершении операции записи;
- `abort`. Это событие срабатывает, если операция записи прерывается;
- `error`. Это событие срабатывает в случае ошибки;
- `writeend`. Это событие срабатывает по завершении запроса — как в случае успешного исхода, так и при возникновении ошибки.

Интерфейс FileError (API File и расширения)

Если определенный процесс завершится ошибкой, несколько методов из рассматриваемого API вернут посредством функции обратного вызова значение, соответствующее этой ошибке. Сравните это значение со следующим списком, для того чтобы узнать причину сбоя:

- `NOT_FOUND_ERR` — значение 1;
- `SECURITY_ERR` — значение 2;
- `ABORT_ERR` — значение 3;
- `NOT_READABLE_ERR` — значение 4;
- `ENCODING_ERR` — значение 5;
- `NO_MODIFICATION_ALLOWED_ERR` — значение 6;
- `INVALID_STATE_ERR` — значение 7;

- `SYNTAX_ERR` — значение 8;
- `INVALID_MODIFICATION_ERR` — значение 9;
- `QUOTA_EXCEEDED_ERR` — значение 10;
- `TYPE_MISMATCH_ERR` — значение 11;
- `PATH_EXISTS_ERR` — значение 12.

13 Коммуникационный API

Ajax уровня 2

Это первая часть API Communication (Коммуникация). Спецификация, носящая неофициальное название коммуникационного API, состоит из трех частей: XMLHttpRequest уровня 2, пересылка сообщений между документами (API Web Messaging (Веб-сообщения)) и веб-сокеты (API WebSocket). Первая из перечисленных коммуникационных технологий представляет собой усовершенствованную версию давно существующего объекта XMLHttpRequest, который до сих пор активно используется для обеспечения взаимодействия с серверами и построения Ajax-приложений.

Второй уровень XMLHttpRequest (XMLHttpRequest Level 2) включает в себя новые возможности, такие как обмен данными между разными источниками и события для управления прохождением запроса. Благодаря этим улучшениям мы теперь можем создавать более простые сценарии и предлагать пользователям новую функциональность, такую как взаимодействие с несколькими серверами из одного приложения или работа с небольшими фрагментами данных вместо целых файлов, — и это далеко не полный список новинок.

Самый важный элемент данного API — это, конечно, объект XMLHttpRequest. Для создания этого объекта используется конструктор `XMLHttpRequest()`. Он возвращает объект XMLHttpRequest, на базе ко-

торого можно создать запрос и затем прослушивать события для управления процессом коммуникации.

Объект, создаваемый конструктором `XMLHttpRequest()`, предлагает важные методы для инициирования и управления запросом:

- `open(method, url, async)`. Настраивает ждущий запрос. В атрибуте `method` нужно указать HTTP-метод, с помощью которого будет открыто соединение, например `GET` или `POST`. Атрибут `url` объявляет местоположение сценария, который будет обрабатывать запрос. Атрибут `async` содержит булево значение, определяющее, по какому типу будет происходить коммуникация: синхронно (`false`) или асинхронно (`true`). Если необходимо, в параметрах метода также можно указать имя пользователя и пароль;
- `send(data)`. Фактически, инициирует запрос. У объекта XMLHttpRequest есть несколько версий данного метода, предназначенных для обработки данных разных типов. Атрибут `data` необязательный, но если он используется, то в качестве его значения можно указать объект `ArrayBuffer`, бинарный блок, документ, строку или объект `FormData`;
- `abort()`. Отменяет запрос.

Извлечение данных

Давайте создадим небольшое приложение, в котором будем извлекать информацию из текстового файла на сервере, применяя для этого метод `GET`. Нам понадобится HTML-документ с кнопкой, которая будет инициировать запрос.

Листинг 13.1. Шаблон для Ajax-запросов

```
<!DOCTYPE html>
<html lang="ru">
<head>
  <title>Ajax Level 2</title>
  <link rel="stylesheet" href="ajax.css">
  <script src="ajax.js"></script>
</head>
<body>
  <section id="formbox">
```

продолжение ⇩

Листинг 13.1 *(продолжение)*

```
    <form name="form">
      <p><input type="button" name="button" id="button"
value="Выполнить"></p>
    </form>
  </section>
  <section id="databox"></section>
</body>
</html>
```

Для того чтобы максимально упростить код, сохраняем уже знакомую вам структуру HTML-документа и применяем только базовые стили.

Листинг 13.2. Стили для оформления полей на экране

```
#formbox{
  float: left;
  padding: 20px;
  border: 1px solid #999999;
}
#databox{
  float: left;
  width: 500px;
  margin-left: 20px;
  padding: 20px;
  border: 1px solid #999999;
}
```

САМОСТОЯТЕЛЬНО

Создайте HTML-файл с шаблоном из листинга 13.1 и CSS-файл с именем ajax.css, в который нужно поместить правила из листинга 13.2. Для того чтобы тестировать следующие примеры в браузере, вам понадобится загрузить все файлы, включая файл с JavaScript-кодом и файл, к которому вы будете обращаться, на сервер. В последующих примерах вы найдете дополнительные инструкции по тестированию.

Код в этом примере будет считывать содержимое из файла на сервере и выводить его на экран. Никакие данные на сервер не отправляются — мы всего лишь выполняем запрос `GET` и показываем полученную информацию.

Листинг 13.3. Считывание файла

```
function initiate(){
  databox=document.getElementById('databox');

  var button=document.getElementById('button');
  button.addEventListener('click', read, false);
}
function read(){
  var url="textfile.txt";
  var request=new XMLHttpRequest();
  request.addEventListener('load',show,false);
  request.open("GET", url, true);
  request.send(null);
}
function show(e){
  databox.innerHTML=e.target.responseText;
}
window.addEventListener('load', initiate, false);
```

В коде из листинга 13.3 представлена типичная функция `initiate()`. Она вызывается после полной загрузки документа. Функция создает ссылку на `databox` и добавляет прослушиватель события `click` к кнопке.

Когда пользователь нажимает кнопку Выполнить, происходит вызов функции `read()`. Здесь мы можем наблюдать все изученные ранее методы в действии. В первую очередь объявляется URL-адрес файла, который мы собираемся прочитать. Мы пока что не рассматривали запросы между разными источниками, поэтому файл должен находиться в том же домене (а в нашем примере и в том же каталоге), что и файл с кодом JavaScript. На следующем шаге с помощью конструктора `XMLHttpRequest()` создается объект, который сразу же присваивается переменной `request`. С использованием этой переменной мы затем создаем прослушиватель события `load` и запускаем процесс, вызывая методы `open()` и `send()`. Поскольку в этом запросе никакие данные не отправляются, метод `send()` пуст (передается значение `null`), однако методу `open()` требуются все атрибуты, иначе он не сможет настроить запрос. Используя этот метод, мы объявляем запрос типа `GET`, указывая URL-адрес файла для считывания и тип операции (`true`, так как нам требуется асинхронная операция).

Асинхронный тип операции подразумевает, что браузер продолжает обрабатывать код, пока происходит считывание файла. О завершении операции нас информирует событие `load`. После того как файл будет загружен, сработает это событие и произойдет вызов функции `show()`. Она

заменяет содержимое поля `databox` значением свойства `responseText`, и на этом процесс завершается.

САМОСТОЯТЕЛЬНО

Для тестирования этого примера создайте текстовый файл с именем textfile.txt и добавьте в него какое-нибудь содержимое. Загрузите этот файл и остальные файлы, содержащие коды из листингов 13.1–13.3, на свой сервер и откройте HTML-документ в браузере. После того как вы щелкнете на кнопке Выполнить, на экране будет показано содержимое текстового файла.

ВНИМАНИЕ

Когда ответ обрабатывается с использованием innerHTML, интерпретируются коды HTML и JavaScript. По соображениям безопасности рекомендуется вместо этого использовать innerText. Выберите тот или иной вариант в зависимости от требований вашего приложения.

Свойства ответа

Есть три типа свойств ответа, с помощью которых мы обрабатываем возвращенную запросом информацию:

- `response`. Это свойство общего назначения. Оно возвращает ответ на запрос согласно значению атрибута `responseType`;
- `responseText`. Возвращает ответ на запрос в текстовом формате;
- `responseXML`. Возвращает ответ на запрос в форме XML-документа.

События

Помимо `load` в спецификации описаны еще несколько событий для объекта `XMLHttpRequest`:

- `loadstart`. Срабатывает в начале запроса;
- `progress`. Срабатывает периодически во время отправки и загрузки данных;

- `abort`. Срабатывает в случае прерывания запроса;
- `error`. Срабатывает, если во время запроса происходит ошибка;
- `load`. Срабатывает по завершении запроса;
- `timeout`. Если указано значение `timeout`, то срабатывает в случае, если запрос не удалось завершить в течение указанного времени;
- `loadend`. Срабатывает по завершении запроса (как в случае успеха, так и в случае ошибки).

Вероятно, самое интересное событие из перечисленных — это `progress`. Оно срабатывает приблизительно каждые 50 мс, информируя о состоянии запроса. Проверяя событие `progress`, мы можем сообщать пользователю о каждом шаге процесса и создавать профессиональные коммуникационные приложения.

Листинг 13.4. Информирование о ходе выполнения запроса

```
function initiate(){
  databox=document.getElementById('databox');

  var button=document.getElementById('button');
  button.addEventListener('click', read, false);
}
function read(){
  var url="trailer.ogg";
  var request=new XMLHttpRequest();
  request.addEventListener('loadstart',start,false);
  request.addEventListener('progress',status,false);
  request.addEventListener('load',show,false);
  request.open("GET", url, true);
  request.send(null);
}
function start(){
  databox.innerHTML='<progress value="0" max="100">0%</progress>';
}
function status(e){
  if(e.lengthComputable){
    var per=parseInt(e.loaded/e.total*100);
    var progressbar=databox.querySelector("progress");
    progressbar.value=per;
    progressbar.innerHTML=per+'%';
```

продолжение ⇙

Листинг 13.4 *(продолжение)*

```
  }
}
function show(e){
  databox.innerHTML='Готово';
}
window.addEventListener('load', initiate, false);
```

В коде из листинга 13.4 для мониторинга запроса мы используем три события: `loadstart`, `progress` и `load`. Событие `loadstart` вызывает функцию `start()`, для того чтобы впервые показать на экране индикатор прогресса. Пока выполняется загрузка файла, событие `progress` успевает несколько раз сработать и вызвать функцию `status()`. Эта функция сообщает о прогрессе запроса посредством элемента `<progress>` и значений свойств из интерфейса `ProgressEvent`.

Наконец, когда загрузка файла завершается, срабатывает событие `load` и функция `show()` выводит на экран строку «Готово».

> **ВНИМАНИЕ**
>
> Для добавления нового элемента <progress> в документ мы использовали innerHTML. Хотя это удобно для нашего конкретного примера, в реальных приложениях так делать не рекомендуется. Элементы обычно добавляются в DOM с использованием JavaScript-метода createElement() и appendChild().

Событие `progress` объявляется в спецификации как часть интерфейса `ProgressEvent`. Это общий интерфейс для всех API, включающий в себя три ценных свойства. Они возвращают информацию о процессе, наблюдение за которым осуществляется с помощью данного события:

- `lengthComputable`. Это всего лишь булево значение, возвращающее `true` в случае, когда показатели прогресса поддаются вычислению, и `false` — в противном случае. В наших примерах мы используем его для проверки того, что остальные свойства имеют допустимые значения;
- `loaded`. Возвращает размер части данных (в байтах), уже загруженной с сервера или на сервер;
- `total`. Возвращает общий размер данных (в байтах), которые должны быть загружены с сервера или на сервер.

ВНИМАНИЕ

В зависимости от того, какое подключение к Интернету вы используете, чтобы увидеть работающий индикатор прогресса, вам может потребоваться запросить загрузку очень большого файла. В коде из листинга 13.4 мы использовали URL-адрес видеофайла, с которым работали в главе 5 во время знакомства с API мультимедиа. Вы можете выбрать какой-либо другой файл или загрузить этот по адресу *http://www.minkbooks.com/content/trailer.ogg*.

Отправка данных

Пока что мы только принимали информацию с сервера, но ничего не отправляли и даже не использовали никакие другие HTTP-методы, кроме `GET`. В следующем примере будем работать с методом `POST` и новым объектом, позволяющим отправлять информацию с использованием элементов виртуальной формы.

Мы не рассказывали о том, как отправлять данные с использованием метода `GET`, потому что это чрезвычайно просто — нужно всего лишь добавить соответствующие значения в URL-адрес. Вы создаете для переменной `url` путь вида `textfile.txt?val1=1&val2=2`, и указанные значения отправляются в запросе. Атрибуты `val1` и `val2` из этого примера будут прочитаны на сервере как переменные `GET`. Разумеется, текстовый файл не в состоянии обработать эту информацию, поэтому чаще всего на сервере добавляется PHP-файл или серверный сценарий другого типа, который получает и интерпретирует эти значения. Однако с запросами типа `POST` все не так просто.

Как вы, вероятно, знаете, запрос `POST` включает в себя всю информацию, отправляемую методом `GET`, и в дополнение этого тело сообщения. В теле сообщения мы можем отправлять информацию любого типа и любой длины. Очень часто наиболее удобным способом предоставления такой информации оказывается HTML-форма, однако для динамических приложений это не лучший вариант. Для решения этой проблемы рассматриваемый API предоставляет интерфейс `FormData`. Этот простой интерфейс включает в себя только конструктор и метод для получения и работы с объектами `FormData`:

- `FormData()`. Этот конструктор возвращает объект `FormData`, который затем в методе `send()` используется для отправки информации;

- `append(name, value)`. Этот метод добавляет данные к объекту `FormData`. Он принимает в качестве атрибутов пару из ключевого слова и значения. В атрибуте `value` можно передавать как строку, так и бинарный блок. Возвращаемые данные представляют собой поле формы.

Листинг 13.5. Отправка виртуальной формы

```
function initiate(){
  databox=document.getElementById('databox');

  var button=document.getElementById('button');
  button.addEventListener('click', send, false);
}
function send(){
  var data=new FormData();
  data.append('name','John');
  data.append('lastname','Doe');

  var url="process.php";
  var request=new XMLHttpRequest();
  request.addEventListener('load',show,false);
  request.open("POST", url, true);
  request.send(data);
}
function show(e){
  databox.innerHTML=e.target.responseText;
}
window.addEventListener('load', initiate, false);
```

Когда информация отправляется на сервер, это означает, что там она должна быть обработана с получением определенного результата. Обычно этот результат сохраняется на сервере, а нам отправляется некая информация, содержащая ответ на запрос. В коде из листинга 13.5 мы отправляем данные на сервер для обработки файлом process.php, а затем выводим на экран информацию, возвращенную этим сценарием.

Для проверки данного примера можно вывести значения, полученные сценарием process.php, при помощи кода из листинга 13.6.

Листинг 13.6. Простой ответ на запрос типа POST (process.php)

```
<?PHP
  print('Ваше имя: '.$_POST['name'].'<br>');
  print('Ваша фамилия: '.$_POST['lastname']);
?>
```

Давайте сначала посмотрим, как информацию готовят к отправке. В функции `send()` в листинге 13.5 мы обратились к конструктору `FormData()`, а возвращенный им объект `FormData` сохранили в переменной `data`. Затем, используя метод `append()`, добавили к этому объекту две пары из ключевого слова и значения с именами `name` и `lastname`. Эти значения представляют собой поля формы ввода данных.

Инициализация запроса происходит точно так же, как в предыдущих примерах кода, но на этот раз в первом атрибуте метода `open()` передается тип `POST`, а не `GET`, а методу `send()` передается объект `data`, а не значение `null`.

Когда пользователь щелкает на кнопке Выполнить, происходит вызов функции `send()` и форма, созданная в объекте `FormData`, отправляется на сервер. Файл process.php получает данные (`name` и `lastname`) и возвращает браузеру текст, содержащий эту информацию. Функция `show()` вызывается после завершения процесса и выводит на экран полученную информацию посредством свойства `responseText`.

САМОСТОЯТЕЛЬНО

Для тестирования этого примера необходимо загрузить на сервер несколько файлов. Мы будем использовать те же версии HTML-документа и стилей CSS, которые определены в листингах 13.1 и 13.2. Вместо предыдущего примера JavaScript-кода необходимо взять код из листинга 13.5, также потребуется создать новый файл с именем process.php, содержащий код из листинга 13.6. Загрузите все эти файлы на сервер и откройте HTML-документ в своем браузере. После того как вы щелкнете на кнопке Выполнить, увидите на экране текст, возвращенный файлом process.php.

Запросы между разными источниками

До сих пор мы работали со сценариями и файлами данных из одного и того же каталога в одном и том же домене. Но XMLHttpRequest уровня 2 позволяет генерировать запросы между разными источниками, то есть мы можем в пределах одного приложения взаимодействовать с разными серверами.

Для того чтобы обеспечить возможность обращения из одного источника к другому, необходимо настроить авторизацию на сервере. Это делается

путем объявления источников, которым разрешен доступ к приложению. Они объявляются в заголовке, отправляемом сервером, на котором находится файл для обработки запроса.

Например, наше приложение расположено в домене www.domain1.com, и мы обращаемся к файлу process.php из нашего примера в домене www.domain2.com. На втором сервере должна быть выполнена необходимая настройка, а именно: домен www.domain1.com должен быть объявлен как допустимый источник для вызова XMLHttpRequest.

Эту информацию можно добавить в конфигурационные файлы сервера или объявить в заголовке сценария. Если мы для примера выберем второй вариант, то понадобится всего лишь добавить заголовок `Access-Control-Allow-Origin` к сценарию process.php.

Листинг 13.7. Настройка для разрешения запросов из разных источников

```
<?PHP
  header('Access-Control-Allow-Origin: *');

  print('Ваше имя: '.$_POST['name'].'<br>');
  print('Ваша фамилия: '.$_POST['lastname']);
?>
```

ВНИМАНИЕ

В PHP-коде из листинга 13.7 мы добавляем новое значение только в заголовок, возвращаемый сценарием process.php. Для того чтобы включить этот параметр в каждый заголовок, возвращаемый сервером, нужно внести изменения в конфигурационные файлы HTTP-сервера. Подробнее об этом вы можете узнать, изучив ссылки для этой главы на нашем вебсайте или обратившись к документации по вашему серверу HTTP.

Значение * в заголовке `Access-Control-Allow-Origin` представляет собой любые источники. Таким образом, к коду из листинга 13.7 можно обращаться из приложений в любых источниках. Однако если мы заменим звездочку (*) конкретным источником, например http://www.domain1.com, то к сценарию смогут обращаться только приложения из домена www.domain1.com.

Загрузка файлов на сервер

Задача загрузки файлов на сервер неизменно вызывает головную боль у всех веб-разработчиков. Эта возможность востребована почти в каждом современном приложении, но не учтена в браузерах. Рассматриваемый API решает данную проблему. Он предлагает новый атрибут, возвращающий объект XMLHttpRequestUpload, который позволяет использовать все методы, свойства и события объектов XMLHttpRequest, а также контролировать процессы загрузки. Атрибут `upload` возвращает объект XMLHttpRequestUpload. Его необходимо вызывать из существующего объекта XMLHttpRequest.

Для работы с этими атрибутом и объектами нам понадобится новый шаблон. Он будет включать в себя поле `<input>` для выбора файла, предназначенного для загрузки на сервер.

Листинг 13.8. Шаблон для загрузки файлов на сервер

```
<!DOCTYPE html>
<html lang="ru">
<head>
  <title>Ajax Level 2</title>
  <link rel="stylesheet" href="ajax.css">
  <script src="ajax.js"></script>
</head>
<body>
  <section id="formbox">
    <form name="form">
      <p>Файл для загрузки:<br><input type="file" name="myfiles"
              id="myfiles"></p>
    </form>
  </section>
  <section id="databox"></section>
</body>
</html>
```

Для того чтобы загрузить файл на сервер, нужно вставить в поле на форме ссылку на файл. Объект `FormData`, который мы изучили в предыдущем примере, отлично справляется с обработкой такого типа данных. Система автоматически определяет, какая информация добавляется в объект `FormData`, и создает соответствующие заголовки для запроса. Далее процесс протекает точно так же, как в других примерах ранее в этой главе.

Листинг 13.9. Загрузка файла на сервер с помощью объекта FormData

```
function initiate(){
  databox=document.getElementById('databox');

  var myfiles=document.getElementById('myfiles');
  myfiles.addEventListener('change', upload, false);
}
function upload(e){
  var files=e.target.files;
  var file=files[0];

  var data=new FormData();
  data.append('file',file);

  var url="process.php";
  var request=new XMLHttpRequest();
  var xmlupload=request.upload;
  xmlupload.addEventListener('loadstart',start,false);
  xmlupload.addEventListener('progress',status,false);
  xmlupload.addEventListener('load',show,false);
  request.open("POST", url, true);
  request.send(data);
}
function start(){
  databox.innerHTML='<progress value="0" max="100">0%</progress>';
}
function status(e){
  if(e.lengthComputable){
    var per=parseInt(e.loaded/e.total*100);
    var progressbar=databox.querySelector("progress");
    progressbar.value=per;
    progressbar.innerHTML=per+'%';
  }
}
function show(e){
  databox.innerHTML='Готово';
}
window.addEventListener('load', initiate, false);
```

Главная функция в листинге 13.9 — это `upload()`. Она вызывается, когда пользователь выбирает новый файл в элементе `<input>` нашего шаблона (то есть когда срабатывает событие `change`). Выбранный файл мы

извлекаем и сохраняем в переменной `file` точно так же, как делали это раньше при изучении файлового API в главе 12 и API перетаскивания в главе 8. Все эти методы возвращают один и тот же объект `File`.

После того как ссылка на файл получена, создается объект `FormData` и мы присоединяем к нему наш файл при помощи метода `append()`. Для отправки формы инициируем запрос типа `POST`. Сначала с переменной `request` связывается обычный объект XMLHttpRequest. Затем с помощью атрибута `upload` создается объект XMLHttpRequestUpload, который в нашем коде представляет переменная `xmlupload`. Используя эту новую переменную, мы добавляем прослушиватели событий для всех событий процесса загрузки и наконец отправляем запрос.

Оставшаяся часть кода ничем не отличается от того, что вы уже видели в листинге 13.4. Другими словами, когда процесс загрузки начинается, на экране отобразится индикатор процесса. Вид индикатора обновляется в соответствии с тем, на каком этапе в данный момент находится процесс отправки файла на сервер.

Приложение из реальной жизни

Вряд ли веб-разработчики считают удобной загрузку файлов на сервер по одному, и редко кому нравится использовать поле `<input>` для выбора предназначенных для отправки файлов. Программисты стараются делать свои приложения максимально интуитивно понятными, и лучший способ создать такое приложение — объединить техники и методы, с которыми пользователи давно и близко знакомы. Благодаря возможностям API перетаскивания мы создадим практическое приложение, позволяющее загружать на сервер несколько файлов одновременно. Пользователю для выбора файлов нужно будет перетащить их на специальную область на экране.

Для начала создадим HTML-документ с зоной приема.

Листинг 13.10. Зона приема для загрузки файлов на сервер

```
<!DOCTYPE html>
<html lang="ru">
<head>
  <title>Ajax Level 2</title>
  <link rel="stylesheet" href="ajax.css">
  <script src="ajax.js"></script>
</head>
```

продолжение ⇩

Листинг 13.10 *(продолжение)*

```
<body>
  <section id="databox">
    <p>Перетащите файлы сюда</p>
  </section>
</body>
</html>
```

Код JavaScript для этого примера будет самым сложным из рассмотренных нами до сих пор в этой книге. В нем не только объединяются возможности двух API, но и постоянно используются анонимные функции, позволяющие аккуратно организовать код и ограничить его единым контекстом (все находится внутри одной и той же функции). Нам необходимо собрать файлы, которые пользователь перетащит на элемент `databox`, и перечислить их на экране, затем подготовить формы с файлами, предназначенными для отправки, и создать для каждого из них запрос на загрузку. Кроме того, мы создадим индикаторы прогресса для всех файлов и будем обновлять их в процессе загрузки.

Листинг 13.11. Последовательная загрузка файлов на сервер

```
function initiate(){
  databox=document.getElementById('databox');

  databox.addEventListener('dragenter', function(e){
e.preventDefault(); }, false);
  databox.addEventListener('dragover', function(e){
e.preventDefault(); }, false);
  databox.addEventListener('drop', dropped, false);
}
function dropped(e){
  e.preventDefault();
  var files=e.dataTransfer.files;
  if(files.length){
    var list='';
    for(var f=0;f<files.length;f++){
      var file=files[f];
      list+='<blockquote>Файл: '+file.name;
      list+='<br><span><progress value="0" max="100">0%
          </progress></span>';
      list+='</blockquote>';
    }
```

```
    databox.innerHTML=list;

    var count=0;

    var upload=function(){
      var file=files[count];
      var data=new FormData();
      data.append('file',file);
      var url="process.php";
      var request=new XMLHttpRequest();
      var xmlupload=request.upload;

      xmlupload.addEventListener('progress',function(e){
        if(e.lengthComputable){
          var child=count+1;
          var per=parseInt(e.loaded/e.total*100);
          var progressbar=databox.querySelector(
             "blockquote:nth-child("+child+") > span > progress");
          progressbar.value=per;
          progressbar.innerHTML=per+'%';
        }
      },false);
      xmlupload.addEventListener('load',function(){
        var child=count+1;
        var elem=databox.querySelector(
           "blockquote:nth-child("+child+") > span");
        elem.innerHTML='Готово!';

        count++;
        if(count<files.length){
          upload();
        }
      },false);
      request.open("POST", url, true);
      request.send(data);
      }
    upload();
  }
}
window.addEventListener('load', initiate, false);
```

Понимаю, в этом коде нелегко разобраться с первого взгляда, поэтому предлагаю изучить происходящее шаг за шагом. Как всегда, все начинается с вызова функции `initiate()` — это происходит сразу же, как только завершается загрузка документа. Эта функция создает ссылку на элемент `databox`, который в нашем примере будет служить зоной приема файлов, и добавляет прослушиватели трех событий, позволяющих контролировать операцию перетаскивания. Для того чтобы вспомнить подробности работы с API перетаскивания, обратитесь к главе 8. Если коротко: событие `dragenter` срабатывает, когда при перетаскивании файлов указатель мыши оказывается над зоной приема, событие `dragover` срабатывает периодически, пока указатель мыши с файлами находится над зоной приема, а событие `drop` срабатывает, когда пользователь отпускает файлы на зоне приема. В этом примере мы никак не обрабатываем события `dragenter` и `dragover`, просто отменяем их, для того чтобы запретить действия браузера по умолчанию. Единственное событие, на которое мы реагируем, — это `drop`. Прослушиватель этого события вызывает функцию `dropped()` каждый раз, когда пользователь отпускает какой-либо объект над элементом `databox`.

В первой строке кода функции `dropped()` также используется метод `preventDefault()`. Это необходимо для того, чтобы мы могли самостоятельно обработать файлы, которые пользователь перетащил на зону приема, запретив браузеру выполнять над ними действия по умолчанию. Теперь, когда мы обеспечили полный контроль над ситуацией, настало время обработать файлы, попавшие в зону приема. Сначала мы извлекаем их из объекта `dataTransfer`. Возвращаемое данным объектом значение представляет собой массив файлов, и мы сохраняем его в переменной `files`. Для того чтобы убедиться, что в зону приема попали именно файлы (а не элементы другого типа), проверяем значение свойства `length` в условном операторе `if(files.length)`. Если данное значение не равно 0 или `null`, значит, пользователь перетащил на зону приема один или несколько файлов и мы можем продолжать работу.

Теперь переходим к непосредственной обработке полученных файлов. В цикле `for` мы проходим по массиву файлов и создаем список элементов `<blockquote>`, содержащих заключенные в теги `<span>` имена файлов и соответствующие индикаторы прогресса. После того как создание списка завершено, результат выводится на экран внутри элемента `databox`.

Создается впечатление, что вся работа происходит в функции `dropped()`, но на самом деле внутри этой функции мы создали еще одну и назвали ее `upload()`. Она занимается процессом загрузки каждого из файлов на

сервер. Таким образом, после вывода списка файлов на экран мы создаем эту функцию и вызываем ее для каждого файла в списке.

Функция `upload()` создается через анонимную функцию. Внутри нее мы выбираем файл из массива, используя в качестве индекса переменную `count`. При инициализации переменной ей присваивается значение 0, поэтому во время первого вызова функции `upload()` выбирается и загружается на сервер первый файл из списка.

Для загрузки каждого файла из списка мы прибегаем к тому же способу, что и во всех предыдущих примерах. Ссылка на файл сохраняется в переменной `file`, с помощью конструктора `FormData()` создается объект `FormData`, и файл добавляется к этому объекту посредством метода `append()`.

На этот раз для управления процессом мы прослушиваем только два события: `progress` и `load`. При каждом срабатывании события `progress` вызывается анонимная функция, обновляющая индикатор прогресса для соответствующего загружаемого файла. Для идентификации элемента `<progress>`, связанного с этим файлом, применяется метод `querySelector()` с псевдоклассом `:nth-child()`. При этом индекс псевдокласса вычисляется на основе значения переменной `count`. Эта переменная содержит значение индекса для массива файлов, но диапазон индексов начинается с 0, а номера дочерних элементов, на которые ссылается `:nth-child()`, — с 1. Для того чтобы получить правильный номер и найти нужный элемент `<progress>`, мы добавляем к значению переменной `count` единицу, сохраняем результат в переменной `child` и используем это значение.

Каждый раз, когда описанный процесс завершается, нам необходимо информировать пользователя о ситуации и переходить к следующему файлу в массиве `files`. Для этой цели в анонимной функции, которая выполняется при срабатывании события `load`, мы увеличиваем на 1 значение переменной `count`, заменяем элемент `<progress>` строкой «Готово!» и снова вызываем функцию `upload()` на случай, если еще остались необработанные файлы.

Последовательно прочитав листинг 13.11, вы увидите, что после своего объявления функция `upload()` впервые вызывается в конце функции `dropped()`. Поскольку мы ранее инициализировали переменную `count` со значением 0, первым обрабатывается первый файл в массиве `files`. Затем, когда загрузка этого файла завершается, срабатывает событие `load` и происходит вызов анонимной функции для обработки данного события. Она, в свою очередь, увеличивает значение `count` на 1 и снова вызывает функцию `upload()` для обработки следующего файла в массиве

array. Таким образом, все файлы, которые пользователь перетащил на зону приема, один за другим загружаются на сервер.

Пересылка сообщений между разными документами

Эта часть так называемого коммуникационного API носит официальное название API Web Messaging (Веб-сообщения). Пересылка сообщений между разными документами (Cross Document Messaging) — это техника, позволяющая приложениям из разных источников обмениваться друг с другом данными. Приложения, выполняющиеся в разных фреймах, вкладках и окнах (и даже другие API), теперь могут общаться между собой, пользуясь возможностями данной технологии. Процедура проста: мы публикуем сообщение от одного документа и обрабатываем это сообщение в целевом документе.

Конструктор

Для публикации сообщений в данном API предусмотрен метод postMessage(): postMessage(message, target). Он применяется к свойству contentWindow того объекта Window, который получает сообщение. Атрибут message — это строка, содержащая передаваемое сообщение, атрибут target представляет собой домен целевого документа (имя хоста или порт — подробнее об этом далее). Целью может служить как определенный домен, так и все возможные документы (если используется символ *). Также можно указать, что цель совпадает с источником, передав параметру значение / (косая черта). Кроме того, метод может в качестве третьего значения принимать массив портов.

События и свойства сообщений

Коммуникационный метод работает в асинхронном режиме. Для прослушивания событий, публикуемых для определенного документа, API предоставляет событие message, которое возвращает информацию через следующие свойства:

- data. Возвращает содержимое сообщения;

- `origin`. Возвращает источник документа, который отправил сообщение, чаще всего имя хоста. Данное значение можно использовать для отправки ответного сообщения;
- `source`. Возвращает объект, идентифицирующий источник сообщения. С помощью данного значения можно сослаться на отправителя и отправить ответное сообщение, как мы увидим далее.

Публикация сообщения

При построении примера для данного API необходимо учесть следующее: процесс обмена данными происходит между разными окнами (фреймами, вкладками или другими API), таким образом, мы должны создать документы и коды для каждой стороны. В примере будем использовать шаблон с фреймом iframe и соответствующими кодами JavaScript для каждого участника. Начнем с основного HTML-документа (листинг 13.12).

Листинг 13.12. Основной документ с фреймом iframe

```
<!DOCTYPE html>
<html lang="ru">
<head>
  <title>Обмен сообщениями между документами</title>
  <link rel="stylesheet" href="messaging.css">
  <script src="messaging.js"></script>
</head>
<body>
  <section id="formbox">
    <form name="form">
      <p>Ваше имя: <input type="text" name="name" id="name"
                    required></p>
      <p><input type="button" name="button" id="button"
          value="Отправить"></p>
    </form>
  </section>
  <section id="databox">
    <iframe id="iframe" src="iframe.html" width="500"
            height="350"></iframe>
  </section>
</body>
</html>
```

Как видите, здесь присутствуют два элемента <section>, как и в предыдущих шаблонах, однако в элемент databox теперь входит элемент <iframe>, в котором загружается файл iframe.html. К этому мы вернемся позже. А пока что добавим к структуре несколько стилей (листинг 13.13).

Листинг 13.13. Стилизация полей (messaging.css)

```
#formbox{
  float: left;
  padding: 20px;
  border: 1px solid #999999;
}
#databox{
  float: left;
  width: 500px;
  margin-left: 20px;
  padding: 20px;
border: 1px solid #999999;
}
```

Код JavaScript для основного документа должен принимать значение, введенное пользователем в поле name формы, и отправлять его документу внутри iframe, применяя для этого метод postMessage().

Листинг 13.14. Публикация сообщения (messaging.js)

```
function initiate(){
  var button=document.getElementById('button');
  button.addEventListener('click', send, false);
}
function send(){
  var name=document.getElementById('name').value;
  var iframe=document.getElementById('iframe');

  iframe.contentWindow.postMessage(name, '*');
}
window.addEventListener('load', initiate, false);
```

В коде из листинга 13.14 сообщение формируется на основе значения в поле ввода name. В качестве цели указан символ *, для того чтобы сообщение отправлялось всем документам, выполняющимся внутри фрейма iframe (независимо от их источника).

Когда пользователь щелкает на кнопке Отправить, происходит вызов функции `send()` и значение из поля ввода отправляется содержимому iframe. Теперь iframe должен принять сообщение и обработать его. Мы создадим для этого фрейма небольшой HTML-документ, в котором полученная информация будет выводиться на экран (листинг 13.15).

Листинг 13.15. Шаблон для фрейма (iframe.html)

```
<!DOCTYPE html>
<html lang="ru">
<head>
  <title>Окно iframe</title>
  <script src="iframe.js"></script>
</head>
<body>
  <section>
    <div><b>Сообщение из главного окна:</b></div>
    <div id="databox"></div>
  </section>
</body>
</html>
```

В этом шаблоне есть собственный элемент `databox`, в котором мы можем отображать сообщение, а для обработки сообщения используется отдельный файл с кодом JavaScript.

Листинг 13.16. Обработка сообщений в целевом документе (iframe.js)

```
function initiate(){
  window.addEventListener('message', receiver, false);
}
function receiver(e){
  var databox=document.getElementById('databox');
  databox.innerHTML='сообщение от: '+e.origin+'<br>';
  databox.innerHTML+='сообщение: '+e.data;
}
window.addEventListener('load', initiate, false);
```

Как говорилось ранее, в данном API для настройки прослушивания сообщений предусмотрены событие `message` и несколько свойств. В коде из листинга 13.16 добавляется прослушиватель события `message`, который при срабатывании события вызывает функцию `receiver()`. Функция

выводит содержимое сообщения и сведения о документе, отправившем сообщение, проверяя для этого значения свойств `data` и `origin`.

Не забывайте, что этот код принадлежит HTML-документу для фрейма iframe, а не главному документу из листинга 13.12. Это два разных документа, у каждого свои среда, контекст и сценарии: один из них представляет собой главное окно браузера, а второй находится внутри iframe.

САМОСТОЯТЕЛЬНО

Для тестирования описанного примера вам понадобится создать и загрузить на сервер пять файлов. Создайте HTML-файл с кодом из листинга 13.12 — это главный документ. Для этого документа необходимо также добавить файл messaging.css со стилями из листинга 13.13 и файл messaging.js с JavaScript-кодом из листинга 13.14. Шаблон из листинга 13.12 включает в себя элемент <iframe>, источником которого является файл iframe.html. Создайте этот файл и поместите в него код из листинга 13.15, также создайте соответствующий файл iframe.js с кодом из листинга 13.16. Загрузите все файлы на свой сервер, откройте первый HTML-документ в браузере и с помощью формы отправьте свое имя во фрейм iframe.

Фильтрация при обмене сообщениями между разными источниками

То, как мы реализовали предыдущий пример, нельзя назвать передовой практикой, особенно с учетом необходимости обеспечения безопасности данных. Код в главном документе отправляет сообщение в определенный фрейм, но никак не контролирует то, каким документам позволено считывать это сообщение (данные сообщения смогут прочитать любые документы внутри iframe). Помимо этого код во фрейме iframe не проверяет источник и обрабатывает все полученные сообщения. Для того чтобы предотвратить злоупотребления такой реализацией, необходимо улучшить обе стороны коммуникационного процесса.

В следующем примере мы исправим ситуацию и продемонстрируем, как отвечать на сообщение из целевого документа, используя еще одно свойство события `message` — свойство `source`.

Листинг 13.17. Обмен сообщениями с конкретными источниками и целевыми документами

```
<!DOCTYPE html>
<html lang="ru">
<head>
  <title>Обмен сообщениями между документами</title>
  <link rel="stylesheet" href="messaging.css">
  <script src="messaging.js"></script>
</head>
<body>
  <section id="formbox">
    <form name="form">
      <p>Ваше имя: <input type="text" name="name" id="name"
         required></p>
      <p><input type="button" name="button" id="button"
          value="Отправить"></p>
    </form>
  </section>
  <section id="databox">
    <iframe id="iframe" src="http://www.domain2.com/iframe.html"
            width="500" height="350"></iframe>
  </section>
</body>
</html>
```

Предполагается, что новый HTML-документ с кодом из листинга 13.17 расположен в домене www.domain1.com. Однако если вы внимательно просмотрите код, то обнаружите, что фрейм iframe загружает файл из другого местоположения в домене www.domain2.com. Для предотвращения злоупотреблений эти два местоположения необходимо объявить в коде JavaScript, точно указав, каким документам разрешено читать сообщения и откуда эти сообщения могут быть отправлены.

В коде из листинга 13.17 мы не только указываем HTML-файл, являющийся источником фрейма, как уже делали раньше, — теперь мы объявляем полный путь к другому местоположению (www.domain2.com). Таким образом, главный документ находится в домене www.domain1.com, а содержимое iframe — в домене www.domain2.com. В следующих примерах кода это условие принимается во внимание.

Листинг 13.18. Обмен сообщениями с конкретными источниками

```
function initiate(){
  var button=document.getElementById('button');
  button.addEventListener('click', send, false);

  window.addEventListener('message', receiver, false);
}
function send(){
  var name=document.getElementById('name').value;
  var iframe=document.getElementById('iframe');
  iframe.contentWindow.postMessage(name, 'http://www.domain2.com');
}
function receiver(e){
  if(e.origin=='http://www.domain2.com'){
    document.getElementById('name').value=e.data;
  }
}
window.addEventListener('load', initiate, false);
```

Обратите внимание на функцию `send()` в коде из листинга 13.18. В методе `postMessage()` теперь объявляется конкретный целевой домен для отправки сообщения (www.domain2.com). Прочитать это сообщение смогут только документы внутри iframe и принадлежащие указанному источнику.

В функции `initiate()` в листинге 13.18 мы также добавили прослушиватель события `message`. Задача этого прослушивателя, а также функции `receiver()` — получить ответ, отправленный документом из iframe. Подробнее об этом поговорим чуть позже.

Давайте пока что изучим код для фрейма и посмотрим, каким образом обрабатывается сообщение из конкретного источника, а также как отправить ответ на это сообщение (для описания структуры фрейма будем использовать старый HTML-документ из листинга 13.15).

Листинг 13.19. Отправка ответа главному документу (iframe.js)

```
function initiate(){
  window.addEventListener('message', receiver, false);
}
function receiver(e){
  var databox=document.getElementById('databox');
  if(e.origin=='http://www.domain1.com'){
```

```
    databox.innerHTML='допустимое сообщение: '+e.data;
    e.source.postMessage('сообщение получено', e.origin);
  }else{
    databox.innerHTML='недопустимый источник';
  }
}
window.addEventListener('load', initiate, false);
```

Фильтрация источника происходит очень просто: мы сравниваем значение свойства `origin` с доменом, который считаем допустимым источником сообщений. Если выясняется, что это правильный источник, то сообщение выводится на экран, а обратно с использованием значения свойства `source` отправляется ответ. Мы используем свойство `origin`, чтобы объявить, что данный ответ будет доступен только тому окну, которое отправило исходное сообщение. Теперь вернитесь к листингу 13.18 и посмотрите, как функция `receiver()` обрабатывает этот ответ.

САМОСТОЯТЕЛЬНО

Этот пример немного сложнее предыдущих. Мы работаем с двумя разными источниками, поэтому для тестирования примеров кода потребуются два разных домена (или субдомена). Замените названия доменов в коде фактическими именами ваших доменов и загрузите файлы для главного документа в один домен, а файлы для фрейма — в другой. Главный документ будет загружать во фрейм код из второго домена, и вы сможете наглядно посмотреть, как осуществляется коммуникационный процесс между этими двумя источниками.

Веб-сокеты

В этой части главы мы поговорим о последней составляющей так называемого коммуникационного API. API WebSocket (Веб-сокеты) предоставляет функциональность для создания более скоростных и производительных каналов коммуникации между браузерами и серверами. Подключение выполняется через TCP-сокет без отправки заголовков HTTP; таким образом, объем данных, пересылаемых при каждом вызове, сокращается. Кроме того, это подключение относится к постоянному

типу: серверы могут отправлять клиентским приложениям обновленные ответы на предыдущие запросы, и нам даже не придется повторно обращаться к серверу, чтобы получить более свежие данные. Вместо этого сервер автоматически отправляет нам сведения о текущем состоянии.

Веб-сокеты иногда считают усовершенствованной версией Ajax, но это совершенно иной механизм коммуникации, позволяющий создавать приложения для обработки данных в реальном времени на масштабируемых платформах, такие как многопользовательские видеоигры или чаты.

Этот основанный на JavaScript API довольно прост: всего несколько методов и событий для открытия и закрытия соединения и отправки и прослушивания сообщений. Однако никакие серверы по умолчанию не поддерживают этот новый протокол, поэтому понадобится установить собственный WS-сервер (сервер WebSocket). После этого мы сможем создавать коммуникационные каналы между браузером и сервером, на котором развернем наше приложение.

Конфигурация WS-сервера

Если вы опытный программист, то, вероятно, сможете самостоятельно создать серверный сценарий для поддержки веб-сокетов, однако если вы желаете тратить свое свободное время на более приятные дела, то в вашем распоряжении уже есть несколько сценариев настройки WS-сервера и подготовки к обработке WS-подключений. В зависимости от собственных предпочтений вы можете выбрать один из множества сценариев, написанных на PHP, Java, Ruby и других языках программирования. Чтобы ознакомиться с полным списком, зайдите на наш веб-сайт и изучите ссылки для этой главы.

ВНИМАНИЕ

На момент написания данной главы ситуация такова: спецификация проходит обновление в части безопасности, а библиотек, в которых эти усовершенствования были бы реализованы, пока нет. По этой причине предлагаемая нами библиотека phpwebsocket работает только в Chrome и к моменту, когда вы возьмете в руки эту книгу, вероятно, уже устареет. Рекомендуем заглянуть на наш веб-сайт, где поддерживается актуальный список полезных ссылок, или поискать в Сети новые доступные WS-серверы.

Поскольку выбор и настройка WS-сервера зависят от конфигурации вашего сервера, мы будем тестировать примеры из этой части главы с использованием XAMPP и сценария PHP. XAMPP — это простой в установке сервер Apache, включающий в себя все необходимые приложения для выполнения PHP-сценариев на вашем компьютере. Для загрузки и установки XAMPP перейдите по ссылке http://www.apachefriends.org/en/xampp.html.

После того как вы установите XAMPP, вам понадобится PHP-сценарий, в котором будет выполняться WS-сервер. Есть несколько версий, однако для тестирования наших примеров воспользуемся библиотекой phpwebsocket, которую можно загрузить по адресу http://code.google.com/p/phpwebsocket/.

САМОСТОЯТЕЛЬНО

Зайдите на веб-сайт *http://www.apachefriends.org/en/xampp.html* и загрузите и установите версию приложения, подходящую для вашей операционной системы. Также вам понадобится файл server.php, который можно загрузить с веб-сайта *http://code.google.com/p/phpwebsocket/*. Скопируйте этот файл в каталог htdocs, созданный сервером XAMPP (этот каталог будет служить локальным хостом (localhost), и там будут находиться все исполняемые файлы). Мы не будем редактировать этот файл в целях упрощения примеров, но вы всегда можете попробовать адаптировать его под свои нужды или вообще избежать возни со сторонними приложениями, обратившись посредством API к собственному WS-серверу.

Веб-сокеты работают с постоянным соединением, поэтому сценарий WS-сервера должен выполняться все время, отлавливая запросы и отправляя пользователям обновленные данные. Для того чтобы выполнить PHP-файл, включив таким образом WS-сервер, откройте панель управления XAMPP и вызовите приложение Shell (скорее всего, вы увидите вверху окна кнопку Shell). В открывшейся консоли найдите каталог htdocs, в котором вы сохранили файл server.php, и выполните следующую команду: `php -q server.php` (в Windows для управления консолью используются команды из DOS: для выбора каталога используйте команду `CD`, а для перечисления его содержимого — команду `DIR`.).

Теперь WS-сервер запущен и готов к обработке запросов. Поскольку он работает на вашем компьютере, обращаться к файлам нужно будет посредством ссылки на локальный хост (localhost). Для выполнения

примеров кода копируйте все файлы в каталог htdocs и открывайте HTML-шаблоны в браузере, указывая в качестве хоста http://localhost/ (например, http://localhost/example.html).

Конструктор

Прежде чем приступать к созданию кода для взаимодействия с WS-сервером, давайте узнаем, какие средства предоставляются для этой цели API-интерфейсом. В спецификации объявляется только один интерфейс с парой методов, свойств и событий, а также конструктор `WebSocket(url)` для создания подключения. Этот конструктор определяет соединение между приложением и WS-сервером, адрес которого передается в атрибуте `url`. Он возвращает объект WebSocket со ссылкой на соединение. Также метод может принимать второй атрибут, содержащий массив коммуникационных подпротоколов.

Методы

Соединение инициализируется конструктором, поэтому для управления соединением используются только два метода:

- `send(data)`. Предназначен для отправки сообщения WS-серверу. Атрибут `data` представляет собой строку с передаваемой на сервер информацией;
- `close()`. Закрывает соединение.

Свойства

Следующие свойства позволяют проверять конфигурацию и статус соединения:

- `url`. Возвращает URL-адрес, к которому подключено приложение;
- `protocol`. Возвращает используемый подпротокол, если он был задан;
- `readyState`. Возвращает число, представляющее собой состояние подключения: 0 означает, что соединение еще не установлено, 1 — соединение открыто, 2 — соединение закрывается и 3 — соединение закрыто;
- `bufferedAmount`. Это очень полезное свойство, при помощи которого можно проверить, сколько данных было запрошено, но еще не отправ-

лено на сервер. Возвращаемое значение помогает корректировать объемы данных и частоту запросов, для того чтобы не перегружать сервер.

События

Чтобы проверять состояние подключения и прослушивать отправляемые сервером сообщения, нам нужны события. События, которые предоставляет этот API, перечислены далее:

- `open`. Срабатывает, когда соединение открывается;
- `message`. Срабатывает, когда сервер присылает сообщение;
- `error`. Срабатывает в случае ошибки;
- `close`. Срабатывает при закрытии соединения.

Шаблон

В файле server.php, который вы загрузили с веб-сайта Google Codes, есть функция `process()`. Она обрабатывает небольшой предопределенный список команд и отправляет обратно подходящий ответ. Для тестирования следующих примеров мы будем использовать форму с одним полем. В поле можно будет ввести одну из поддерживаемых команд и отправить ее на сервер (листинг 13.20).

Листинг 13.20. Отправка команд

```
<!DOCTYPE html>
<html lang="ru">
<head>
  <title>WebSocket</title>
  <link rel="stylesheet" href="websocket.css">
  <script src="websocket.js"></script>
</head>
<body>
  <section id="formbox">
    <form name="form">
      <p>Команда:<br><input type="text" name="command"
         id="command"></p>
```

продолжение ⇩

Листинг 13.20 *(продолжение)*

```
      <p><input type="button" name="button" id="button"
          value="Отправить"></p>
    </form>
  </section>
  <section id="databox"></section>
</body>
</html>
```

Также нам потребуется CSS-файл со следующими стилями (листинг 13.21). Сохраните его с именем websocket.css.

Листинг 13.21. Обычные стили для полей

```
#formbox{
  float: left;
  padding: 20px;
  border: 1px solid #999999;
}
#databox{
  float: left;
  width: 500px;
  height: 350px;
  overflow: auto;
  margin-left: 20px;
  padding: 20px;
  border: 1px solid #999999;
}
```

Начало обмена данными

Как всегда, за поддержку всего процесса отвечает JavaScript-код. Давайте создадим наше первое коммуникационное приложение и проверим, как работает этот API (листинг 13.22).

Листинг 13.22. Отправка сообщений на сервер

```
function initiate(){
  databox=document.getElementById('databox');
  var button=document.getElementById('button');
```

```
  button.addEventListener('click', send, false);

    socket=new WebSocket("ws://localhost:12345/server.php");
    socket.addEventListener('message', received, false);
}
function received(e){
  var list=databox.innerHTML;
  databox.innerHTML='Получено: '+e.data+'<br>'+list;
}
function send(){
  var command=document.getElementById('command').value;
  socket.send(command);
}
window.addEventListener('load', initiate, false);
```

В функции `initiate()` мы создаем объект WebSocket и сохраняем его в переменной `socket`. Атрибут `url` указывает местоположение файла server.php на нашем компьютере (localhost) и содержит номер порта для создаваемого соединения. Чаще всего мы указываем хост по IP-адресу сервера, добавляя номер порта 8000 или 8080, однако это зависит от требований конкретного приложения, конфигурации сервера, доступных портов, местоположения файла на сервере и т. д. Использование IP-адреса вместо доменного имени позволяет избежать этапа DNS-трансляции (всегда прибегайте к этой технике доступа к приложению, для того чтобы не тратить время на трансляцию имени домена в соответствующий IP-адрес.).

ВНИМАНИЕ

Файл server.php включает в себя функцию process(), назначение которой — обрабатывать все вызовы и отправлять обратно ответ. Вы можете отредактировать эту функцию с учетом требований вашего приложения, но в наших примерах мы используем ее в том виде, в котором она предлагается в библиотеке Google Codes. Эта функция проверяет полученное сообщение и сравнивает его значение с предустановленным списком команд. В той версии, с помощью которой мы тестировали наши примеры, доступны следующие команды: hello, hi, name, age, date, time, thanks и bye. Например, если вы отправите команду hello, то сервер вернет сообщение «hello human».

Получив объект WebSocket, мы добавляем к нему прослушиватель события `message`. Событие `message` срабатывает каждый раз, когда WS-сервер отправляет браузеру сообщение, и в этом случае в нашем приложении происходит вызов функции `received()`. Как и в других коммуникационных API, у этого события есть свойство `data`, внутри которого находится содержимое сообщения. Мы используем его в функции `received()` для вывода сообщения на экран.

Для отправки сообщений на сервер мы добавили функцию `send()`. Она принимает значение элемента `<input>` с именем `command` и отправляет его WS-серверу в методе `send()`.

Полное приложение

Наш первый пример хорошо иллюстрирует работу коммуникационного процесса в рассматриваемом API. Конструктор WebSocket открывает соединение, метод `send()` отправляет сообщения, которые должны обрабатываться на сервере, а событие `message` информирует приложение о прибытии новых сообщений с сервера. Однако мы в этом примере не закрывали соединение, не проверяли ошибки и даже не удостоверились, что соединение готово к работе. Давайте рассмотрим пример, включающий в себя все предоставляемые API-интерфейсом события. Код из этого примера информирует пользователя о статусе подключения на каждом шаге процесса.

Листинг 13.23. Информирование о состоянии подключения

```
function initiate(){
  databox=document.getElementById('databox');
  var button=document.getElementById('button');
  button.addEventListener('click', send, false);

  socket=new WebSocket("ws://localhost:12345/server.php");
  socket.addEventListener('open', opened, false);
  socket.addEventListener('message', received, false);
  socket.addEventListener('close', closed, false);
  socket.addEventListener('error', error, false);
}
function opened(){
  databox.innerHTML='СОЕДИНЕНИЕ ОТКРЫТО<br>';
  databox.innerHTML+='Статус: '+socket.readyState;
}
```

```
function received(e){
  var list=databox.innerHTML;
  databox.innerHTML='Получено: '+e.data+'<br>'+list;
}
function closed(){
  var list=databox.innerHTML;
  databox.innerHTML='СОЕДИНЕНИЕ ЗАКРЫТО<br>'+list;

  var button=document.getElementById('button');
  button.disabled=true;
}
function error(){
  var list=databox.innerHTML;
  databox.innerHTML='ОШИБКА<br>'+list;
}
function send(){
  var command=document.getElementById('command').value;
  if(command=='close'){
    socket.close();
  }else{
    socket.send(command);
  }
}
window.addEventListener('load', initiate, false);
```

САМОСТОЯТЕЛЬНО

Для тестирования последнего примера требуются HTML-документ и CSS-стили из листингов 13.20 и 13.21 соответственно. Скопируйте эти файлы в каталог htdocs, созданный приложением XAMPP, и откройте консоль Shell, щелкнув на кнопке Shell на панели управления XAMPP. В открывшейся консоли вам нужно будет перейти к каталогу htdocs и запустить PHP-сервер командой php –q server.php. Откройте браузер и перейдите на страницу *http://localhost/client.html* (где client.html — это имя вашего HTML-файла с документом из листинга 13.20). Вводите команды в поле формы и нажимайте кнопку Отправить. В зависимости от того, какая команда отправлена (hello, hi, name, age, date, time, thanks или bye), вы будете получать от сервера разные ответы. Для того чтобы закрыть соединение, отправьте команду close.

По сравнению с предыдущим примером в код из листинга 13.23 мы добавили несколько усовершенствований. Создали прослушиватели всех событий, предлагаемых объектом WebSocket, и соответствующие функции для обработки каждого события. Помимо этого, когда подключение создается, мы, используя значение свойства `readyState`, выводим на экран информацию о его статусе. Закрываем соединение методом `close()`, когда с формы передается команда `close`, и деактивируем кнопку `Отправить` после закрытия соединения (`button.disabled=true`).

ВНИМАНИЕ

Сейчас файл server.php из библиотеки Google Codes работает не совсем правильно. В нем не реализованы новые меры безопасности, и, скорее всего, вам понадобится несколько раз обновить HTML-страницу, чтобы увидеть сообщение «Соединение открыто». Только после этого можно будет приступать к отправке сообщений. Чтобы проверить, какие новые WS-серверы появились, перейдите на наш веб-сайт и изучите ссылки для этой главы или же просто выполните поиск в Сети.

Краткий справочник. API Communication (Коммуникация)

В HTML5 входят три разных API, обслуживающие различные коммуникационные задачи. XMLHttpRequest уровня 2 — это улучшенная версия давно знакомого вам объекта XMLHttpRequest для приложений Ajax. API Web Messaging (Веб-сообщения) обеспечивает возможность обмена сообщениями между окнами, вкладками, фреймами и даже другими API. API WebSocket (Веб-сокеты) предоставляет новые альтернативные способы установления быстрых и эффективных соединений между сервером и клиентом.

XMLHttpRequest уровня 2

Этот API включает в себя конструктор для объектов XMLHttpRequest и несколько методов, свойств и событий, помогающих обрабатывать соединение:

- `XMLHttpRequest()`. Этот конструктор возвращает объект XMLHttpRequest, необходимый для открытия и обработки соединения с сервером;
- `open(method, url, async)`. Этот метод открывает соединение между приложением и сервером. Атрибут `method` определяет HTTP-метод, с использованием которого будет пересылаться информация (например, `GET`, `POST`). Атрибут `url` содержит путь к сценарию, который должен получить информацию. Атрибут `async` — это булево (логическое) значение, определяющее тип соединения: синхронное или асинхронное (значение `true` соответствует асинхронному типу);
- `send(data)`. Этот метод отправляет значение атрибута `data` на сервер. В атрибут можно поместить объект `ArrayBuffer`, бинарный блок, документ, строку или объект `FormData`;
- `abort()`. Этот метод отменяет запрос;
- `timeout`. Это свойство устанавливает максимальное время обработки запроса (в миллисекундах);
- `readyState`. Это свойство возвращает значение, представляющее статус соединения: 0 означает, что объект создан, 1 — соединение открыто, 2 — получен заголовок ответа, 3 — получен ответ, 4 — пересылка данных завершена;
- `responseType`. Это свойство возвращает тип ответа. Также его можно использовать для изменения типа ответа. Допустимые значения `arraybuffer`, `blob`, `document` и `text`;
- `response`. Это свойство возвращает ответ на запрос в формате, объявленном в свойстве `responseType`;
- `responseText`. Это свойство возвращает ответ на запрос в текстовом формате;
- `responseXML`. Это свойство возвращает ответ на запрос в форме XML-документа;
- `loadstart`. Это событие срабатывает в начале запроса;
- `progress`. Это событие срабатывает периодически во время обработки запроса;
- `abort`. Это событие срабатывает, если запрос прерывается;

- `error`. Это событие срабатывает, когда происходит ошибка;
- `load`. Это событие срабатывает при успешном завершении запроса;
- `timeout`. Это событие срабатывает, когда истекает указанное с помощью свойства `timeout` время, выделенное на обработку запроса;
- `loadend`. Это событие срабатывает по завершении запроса (как в случае успеха, так и в случае ошибки).

Также добавлен специальный атрибут, позволяющий для отправки данных на сервер создавать объект XMLHttpRequestUpload вместо обычного объекта XMLHttpRequest:

- `upload`. Возвращает объект XMLHttpRequestUpload. Объект использует те же методы, свойства и события, что и объект XMLHttpRequest, но работает только в процессе загрузки данных на сервер.

В состав API также входит интерфейс для создания объектов `FormData`, представляющих собой HTML-формы:

- `FormData()`. Этот конструктор возвращает объект `FormData`, представляющий HTML-форму;
- `append(name, value)`. Этот метод добавляет данные в объект `FormData`. Каждый фрагмент данных, добавляемый в объект, представляет собой поле формы, имя и значение которого объявляются в атрибутах. В качестве атрибута `value` можно передавать как строку, так и бинарный блок.

В данном API используется обычный интерфейс ProgressEvent (который применяется и в других API для управления прогрессом выполнения операции), включающий в себя следующие свойства:

- `lengthComputable`. Возвращает булево значение, позволяющее определить, допустимы ли значения остальных свойств;
- `loaded`. Возвращает общий размер уже загруженных с сервера или на сервер данных (в байтах);
- `total`. Возвращает общий размер данных, которые должны быть загружены с сервера или на сервер (в байтах).

API Web Messaging (Веб-сообщения)

Этот API включает в себя только один интерфейс, предоставляющий несколько методов, свойств и событий для обмена данными с приложениями в других окнах, вкладках, фреймах и даже других API:

- `postMessage(message, target)`. Этот метод отправляет сообщение через определенное свойство `contentWindow` в документ, объявленный в качестве цели пересылки в атрибуте `target`. Атрибут `message` содержит пересылаемое сообщение;
- `message`. Это событие срабатывает при получении сообщения;
- `data`. Это свойство события `message` возвращает содержимое полученного сообщения;
- `origin`. Это свойство события `message` возвращает источник документа, отправившего сообщение;
- `source`. Это свойство события `message` возвращает ссылку на свойство `contentWindow`, из которого было отправлено сообщение.

API WebSocket (Веб-сокеты)

Этот API включает в себя конструктор, который возвращает объект WebSocket и открывает соединение. Кроме того, он предоставляет несколько методов, свойств и событий для контролирования процесса обмена данными между клиентом и сервером:

- `WebSocket(url)`. Этот конструктор возвращает объект WebSocket и открывает соединение с сервером. В атрибуте `url` объявляются путь к сценарию, в котором выполняется WS-сервер, а также порт для обмена данными. В качестве второго атрибута можно передать массив подпротоколов;
- `send(data)`. Этот метод отправляет сообщение WS-серверу. В атрибуте `data` должна содержаться строка с информацией, которую необходимо передать на сервер;
- `close()`. Этот метод закрывает соединение с WS-сервером;
- `url`. Это свойство возвращает URL-адрес, который в приложении используется для подключения к серверу;
- `protocol`. Это свойство возвращает подпротокол, используемый для обработки соединения, если он был указан;
- `readyState`. Это свойство возвращает значение, представляющее собой состояние соединения: 0 означает, что соединение еще не установлено, 1 — соединение открыто, 2 — соединение закрывается, 3 — соединение закрыто;
- `bufferedAmount`. Это свойство возвращает общий объем данных, ожидающих отправки на сервер;

- `open`. Это событие срабатывает, когда соединение открывается;
- `message`. Это событие срабатывает, когда сервер отправляет приложению сообщение;
- `error`. Это событие срабатывает в случае ошибки;
- `close`. Это событие срабатывает при закрытии соединения.

14 API рабочих процессов

Самая тяжелая работа

Язык программирования JavaScript превратился в основной инструмент построения успешных приложений в Сети. Как мы объясняли в главе 4, его уже нельзя назвать всего лишь альтернативой для создания симпатичных (а иногда раздражающих) эффектов на веб-страницах. Этот язык стал неотъемлемой частью Сети и по праву считается технологией первостепенной значимости, которую каждый разработчик должен понимать и применять.

JavaScript уже достиг статуса языка программирования общего назначения, поэтому от него требуют предоставления элементарных возможностей, которые, однако, попросту не предусмотрены его природой. Этот язык первоначально создавался как язык сценариев, который должен обрабатывать одновременно только один фрагмент кода. Отсутствие в JavaScript многопоточности (то есть возможности обрабатывать несколько разных фрагментов кода одновременно) снижает эффективность и масштабируемость языка, к тому же делает невозможной эмуляцию в Сети некоторых настольных приложений.

Web Workers (Рабочие процессы) — это API, разработанный исключительно для того, чтобы наконец решить эту проблему, превратив JavaScript в многопоточный язык. Теперь благодаря HTML5 мы можем выполнять длительные сценарии в фоновом режиме, в то время как на странице продолжает работать главный сценарий, получающий данные от пользователя и обеспечивающий ответную реакцию.

Создание рабочего процесса

API Web Workers работает очень просто: код рабочего процесса сохраняется в отдельном JavaScript-файле, и разные сценарии общаются между собой посредством сообщений. Обычно сообщения, отправляемые рабочим процессам из главного кода, содержат информацию, предназначенную для обработки. В свою очередь, рабочие процессы возвращают сообщения с результатами обработки. Для отправки и получения сообщений в этом API применяются техники, реализованные в других API, с которыми мы познакомились раньше. Например, уже известные вам события и методы позволяют отправлять и получать сообщения между сценариями:

- `Worker(scriptURL)`. Прежде чем начинать обмен данными с рабочим процессом, необходимо получить объект, указывающий на файл, в котором содержится код рабочего процесса. Этот метод возвращает объект Worker. Атрибут `scriptURL` содержит URL-адрес файла с кодом (рабочим процессом), который будет выполняться в фоновом режиме;
- `postMessage(message)`. Это тот же метод, который мы изучили в главе 13, когда обсуждали API Web Messaging (Веб-сообщения), только реализованный для объекта Worker. Он отправляет сообщение в код рабочего процесса или обратно. Атрибут `message` должен содержать строку или объект JSON, представляющий передаваемое сообщение;
- `message`. Это событие вам тоже знакомо — оно прослушивает сообщения, отправляемые нашему коду. Как и метод `postMessage()`, оно применимо как к рабочему процессу, так и к главному коду. Отправленное сообщение извлекается из свойства `data`.

Отправка и получение сообщений

Для того чтобы посмотреть, как рабочие процессы и главный код обмениваются друг с другом сообщениями, создадим простой шаблон. Мы будем отправлять рабочему процессу сообщение, содержащее имя из поля ввода, и выводить на экран ответ.

Даже для самого простого примера использования рабочих процессов требуются как минимум три файла: главный документ, файл с главным JavaScript-кодом и файл с кодом рабочего процесса. В листинге 14.1 представлен наш HTML-документ.

Листинг 14.1. Шаблон для тестирования рабочих процессов

```
<!DOCTYPE html>
<html lang="ru">
<head>
  <title>Рабочие процессы</title>
  <link rel="stylesheet" href="webworkers.css">
  <script src="webworkers.js"></script>
</head>
<body>
  <section id="formbox">
    <form name="form">
      <p>Имя:<br><input type="text" name="name" id="name"></p>
      <p><input type="button" name="button" id="button"
value="Отправить"></p>
    </form>
  </section>
  <section id="databox"></section>
</body>
</html>
```

К этому шаблону мы добавим CSS-файл под названием webworkers.css со следующими правилами (листинг 14.2).

Листинг 14.2. Стилизация полей

```
#formbox{
  float: left;
  padding: 20px;
  border: 1px solid #999999;
}
#databox{
  float: left;
  width: 500px;
  margin-left: 20px;
  padding: 20px;
  border: 1px solid #999999;
}
```

Код JavaScript для главного документа должен уметь отправлять рабочему процессу информацию, которую тот будет обрабатывать. Кроме того, требуется прослушивание ответа.

Листинг 14.3. Простой пример использования API рабочих процессов

```
function initiate(){
  databox=document.getElementById('databox');
  var button=document.getElementById('button');
  button.addEventListener('click', send, false);

  worker=new Worker('worker.js');
  worker.addEventListener('message', received, false);
}
function send(){
  var name=document.getElementById('name').value;
  worker.postMessage(name);
}
function received(e){
  databox.innerHTML=e.data;
}
window.addEventListener('load', initiate, false);
```

В листинге 14.3 содержится код нашего документа (который должен быть сохранен в файле webworkers.js). В функции `initiate()` мы сначала создаем необходимые ссылки для поля `databox` и кнопки отправки сообщения, а затем объект Worker. Конструктор `Worker()` принимает файл с кодом рабочего процесса worker.js и возвращает объект Worker, содержащий ссылку на этот файл. Любое взаимодействие с данным объектом, по сути, означает взаимодействие с кодом в файле, на который он указывает.

После того как мы получим нужный объект, необходимо добавить прослушиватель события `message`, для того чтобы отлавливать поступающие от рабочего процесса сообщения. При получении очередного сообщения происходит вызов функции `received()` и на экране отображается значение свойства `data` (само сообщение).

Вторая часть коммуникационного процесса описана в функции `send()`. Когда пользователь щелкает на кнопке Отправить, функция извлекает значение поля ввода `name` и, используя `postMessage()`, отправляет содержащее его сообщение рабочему процессу.

Создав функции `received()` и `send()` для обработки коммуникационного процесса, мы теперь готовы отправлять сообщения рабочему процессу и обрабатывать его ответы. Давайте подготовим сам рабочий процесс.

Листинг 14.4. Код рабочего процесса (worker.js)

```
addEventListener('message', received, false);

function received(e){
  var answer='Ваше имя – '+e.data;
  postMessage(answer); }
```

Так же, как код из листинга 14.3, код рабочего процесса должен непрерывно слушать эфир, дожидаясь сообщений от главного кода, — для этого используется обработчик события `message`. В первой строке листинга 14.4 мы добавляем в рабочий процесс прослушиватель данного события. При каждом срабатывании события (получении сообщения) он будет вызывать функцию `received()`. В этой функции значение свойства `data` присоединяется к неизменяемой строке и отправляется обратно главному коду в методе `postMessage()`.

САМОСТОЯТЕЛЬНО

Сравните фрагменты кода в листингах 14.3 и 14.4 (главный код и код рабочего процесса). Обратите внимание на то, как происходит процесс коммуникации: в обоих примерах для этого используются одни и те же метод и событие. Создайте файлы с фрагментами кода из листингов 14.1–14.4, загрузите их на свой сервер и откройте HTML-документ в браузере.

ВНИМАНИЕ

Для создания ссылки на рабочий процесс можно использовать self или this (например, self.postMessage()) или просто объявить методы так, как сделано в листинге 14.4.

Разумеется, это чрезвычайно простой рабочий процесс. В действительности он даже ничего не обрабатывает — всего лишь создает строку на основе полученного сообщения и сразу же отправляет обратно ответ. Однако этот пример помогает понять, как между собой общаются разные сценарии и как использовать данный API в целом.

Несмотря на простоту рабочих процессов, прежде чем приступать к их созданию, необходимо задуматься о нескольких важных моментах. Единственный способ прямого обмена данными с рабочими процессами — это пересылка сообщений. Сообщения должны создаваться на базе строк или JSON-объектов, так как рабочие процессы не поддерживают получение

данных других типов. Кроме того, они не могут обращаться к документу и манипулировать объектами HTML, функциями JavaScript и переменными в главном коде. Рабочие процессы — это «вещи в себе», которые умеют только обрабатывать информацию, полученную в сообщениях, и отправлять результат обратно с использованием того же механизма.

Распознавание ошибок

Несмотря на упомянутые ограничения, рабочие процессы все же представляют собой весьма гибкий и мощный механизм. Внутри рабочих процессов можно использовать функции, методы и целые API. Учитывая то, насколько сложными рабочие процессы могут становиться при добавлении в них всех этих возможностей, без функциональности обработки ошибок не обойтись. API Web Workers (Рабочие процессы) включает в себя специальное событие `error`, позволяющее распознавать ошибки и возвращать все доступные сведения о текущей ситуации. Это событие срабатывает на объекте Worker в главном коде при возникновении любой ошибки. Оно предоставляет информацию посредством трех свойств: `message`, `filename` и `lineno`. Свойство `message` выдает сообщение об ошибке. Это строка, благодаря которой мы узнаем, что пошло не так. Свойство `filename` содержит имя файла, код в котором вызвал ошибку. Его полезно применять, когда внутри рабочего процесса загружаются внешние файлы (пример этого мы рассмотрим чуть позже). Наконец, свойство `lineno` возвращает номер строки, в которой произошла ошибка.

Давайте создадим код, который будет показывать возвращенную рабочим процессом ошибку.

Листинг 14.5. Использование события error

```
function initiate(){
  databox=document.getElementById('databox');
  var button=document.getElementById('button');
  button.addEventListener('click', send, false);

  worker=new Worker('worker.js');
  worker.addEventListener('error', error, false);
}
function send(){
  var name=document.getElementById('name').value;
  worker.postMessage(name);
}
```

```
function error(e){
  databox.innerHTML='Ошибка: '+e.message+'<br>';
  databox.innerHTML+='Имя файла: '+e.filename+'<br>';
  databox.innerHTML+='Номер строки: '+e.lineno;
}
window.addEventListener('load', initiate, false);
```

Код в этом примере очень похож на код из листинга 14.3. Здесь мы создаем рабочий процесс, но используем только событие `error`, поскольку на этот раз не собираемся слушать ответы рабочего процесса — нас интересуют только ошибки. Конечно, пользы от этого кода никакой, однако он все же демонстрирует, каким образом возвращаются ошибки и какая информация в подобных ситуациях доступна.

Для того чтобы намеренно вызвать ошибку, мы можем обратиться внутри рабочего процесса к несуществующей функции.

Листинг 14.6. Неработающий рабочий процесс

```
addEventListener('message', received, false);

function received(e){
  test();
}
```

В этом рабочем процессе все так же нужно прослушивать событие `message`, для того чтобы не пропустить сообщение от главного кода, с которого начинается работа. Когда сообщение приходит, происходит вызов функции `received()`, и она вызывает несуществующую функцию `test()`, генерируя тем самым простейшую ошибку.

Когда возникает ошибка, в главном коде срабатывает событие `error` и вызывается функция `error()`, которая выводит на экран значения трех свойств объекта события. Изучите код в листинге 14.5, для того чтобы понять, как функция принимает и обрабатывает эту информацию.

САМОСТОЯТЕЛЬНО

В этом примере мы используем HTML-документ и правила CSS из листингов 14.1 и 14.2. Скопируйте код из листинга 14.5 в файл webworkers.js, а код из листинга 14.6 — в файл worker.js. Откройте шаблон из листинга 14.1 в своем браузере и отправьте с помощью формы любую строку. На экране появится информация об ошибке, возвращенной рабочим процессом.

Остановка рабочих процессов

Рабочие процессы — это специальные блоки кода, постоянно выполняющиеся в фоновом режиме в ожидании поступления информации на обработку. Обычно необходимость в фоновой обработке возникает в специфических обстоятельствах, поэтому рабочие процессы создаются с учетом конкретных задач. Таким образом, их услуги требуются не все время, и считается хорошей практикой программирования останавливать или прерывать рабочие процессы, когда становится ясно, что они больше не нужны.

Для этой цели в API предусмотрено два метода:

- `terminate()`. Останавливает рабочий процесс из главного кода;
- `close()`. Позволяет остановить рабочий процесс изнутри него самого.

Когда мы прерываем рабочий процесс, он полностью останавливается и все задания в цикле событий отбрасываются. Для тестирования обоих методов создадим небольшое приложение, работающее по тому же принципу, что и в первом примере, но также реагирующее на две специальные команды: `close1` и `close2`. Если отправить из формы строку «close1» или «close2», то рабочий процесс будет остановлен главным кодом или кодом внутри самого рабочего процесса посредством `terminate()` или `close()` соответственно.

Листинг 14.7. Остановка рабочего процесса из главного кода

```
function initiate(){
  databox=document.getElementById('databox');
  var button=document.getElementById('button');
  button.addEventListener('click', send, false);

  worker=new Worker('worker.js');
  worker.addEventListener('message', received, false);
}
function send(){
  var name=document.getElementById('name').value;
  if(name=='close1'){
    worker.terminate();
    databox.innerHTML='Рабочий процесс прерван';
  }else{
    worker.postMessage(name);
  }
}
```

```
function received(e){
  databox.innerHTML=e.data;
}
window.addEventListener('load', initiate, false);
```

Единственное отличие кода в листинге 14.7 от кода в листинге 14.3 заключается в новом условном операторе `if`, проверяющем, не была ли передана команда `close1`. Если вместо произвольного имени через форму передается эта команда, то происходит вызов метода `terminate()` и на экран выводится сообщение о том, что рабочий процесс был остановлен. В то же время любая строка, отличающаяся от этой команды, отправляется рабочему процессу как обычное сообщение.

Код рабочего процесса будет выполнять похожую работу. Если окажется, что в полученном сообщении содержится строка «close2», то рабочий процесс остановит сам себя посредством метода `close()`, в противном случае он отправит обратно сообщение.

Листинг 14.8. Рабочий процесс, останавливающий сам себя

```
addEventListener('message', received, false);

function received(e){
  if(e.data=='close2'){
    postMessage('Рабочий процесс прерван');
    close();
  }else{
    var answer='Ваше имя – '+e.data;
    postMessage(answer);
  }
}
```

САМОСТОЯТЕЛЬНО

Используйте тот же HTML-документ и правила CSS из листингов 14.1 и 14.2. Скопируйте код из листинга 14.7 в файл webworkers.js, а код из листинга 14.8 — в файл worker.js. Откройте шаблон в браузере и отправьте через форму команду close1 или close2. После этого рабочий процесс больше не будет присылать сообщения.

Синхронные API

У рабочих процессов есть определенные ограничения на взаимодействие с главным документом и доступ к его содержимому, однако, как мы уже говорили, если речь идет об обработке данных и функциональности, ситуация выглядит гораздо лучше. Например, в рабочих процессах можно использовать обычные методы, такие как `setTimeout()` и `setInterval()`, загружать дополнительную информацию с серверов посредством `XMLHttpRequest` и применять возможности некоторых API, создавая таким образом весьма мощный код. Последнее звучит наиболее многообещающе, однако и здесь есть одна загвоздка: нам придется познакомиться с другими реализациями этих API, доступными для рабочих процессов.

С некоторыми API мы уже познакомились, однако рассматривали только асинхронные реализации. Для большинства API доступны две версии: асинхронная и синхронная. Эти две версии одного API выполняют одинаковые задачи, однако используют особые методы в зависимости от того, каким способом выполняется обработка. Асинхронные API удобно применять для длительных операций, требующих ресурсов, которые главный документ предоставить не в состоянии. Асинхронные операции выполняются в фоновом режиме, пока главный код продолжает работать обычным образом без всяких замираний. Поскольку рабочие процессы представляют собой новые потоки, выполняющиеся одновременно с главным кодом, они по своей природе являются асинхронными, что исключает необходимость в других асинхронных операциях.

ВНИМАНИЕ

Синхронные версии предлагаются для нескольких API, например файлового API и API индексированных баз данных, однако большинство из них в настоящее время находятся на этапе разработки или работают нестабильно. Более подробную информацию и примеры вы найдете, изучив ссылки для этой главы на нашем веб-сайте.

Импорт сценариев

Отдельно стоит упомянуть такую возможность, как импортирование внутри рабочих процессов внешних JavaScript-файлов. Рабочий процесс может содержать весь код, необходимый для выполнения поставленной задачи. Однако для одного и того же документа может быть создано

несколько рабочих процессов, и тогда появляется вероятность, что какая-то часть кода будет дублироваться. Мы можем выделить эти фрагменты, поместить их в отдельный файл и загружать его внутри любого рабочего процесса, применяя новый метод `importScripts()`: `importScripts(file)`. Он загружает внешний JavaScript-файл для добавления в рабочий процесс нового кода. Атрибут `file` содержит путь к нужному файлу.

Если вам когда-либо приходилось использовать методы в других языках программирования, то вы наверняка заметите сходство между `importScripts()` и такими функциями, как, например, `include()` в PHP. Код из файла встраивается в рабочий процесс так, как будто он всегда был его частью.

Для того чтобы применить новый метод `importScripts()`, необходимо объявить его в начале рабочего процесса. Код рабочего процесса не готов к запуску до тех пор, пока все объявленные файлы не будут загружены.

Листинг 14.9. Загрузка внешнего JavaScript-кода для рабочего процесса

```
importScripts('morecodes.js');

addEventListener('message', received, false);

function received(e){
  test();
}
```

Код в листинге 14.9 совершенно нефункциональный — это всего лишь пример использования метода `importScripts()`. В нашей гипотетической ситуации файл morecodes.js, содержащий функцию `test()`, загружается сразу же после загрузки файла рабочего процесса. После этого функцию `test()` (и любые другие функции из файла morecodes.js) можно применять в любом месте кода рабочего процесса.

Общие рабочие процессы

Тот тип рабочих процессов, с которым мы познакомились ранее, называется выделенным рабочим процессом (Dedicated Worker). Выделенный рабочий процесс отправляет ответы только главному коду, который его создал. Существуют также общие рабочие процессы, умеющие взаимодействовать с разными документами из одного источника. Работа с несколькими подключениями означает, что к одному и тому же рабочему

процессу можно обращаться из разных окон, вкладок и фреймов и одновременно обновлять и синхронизировать содержимое во всех них, создавая таким образом очень сложные приложения.

Соединение происходит через порты, которые можно сохранять в рабочем процессе для последующих обращений. Для использования общих рабочих процессов и портов в этой части API созданы новые свойства, события и методы:

- `SharedWorker(scriptURL)`. Этот конструктор заменяет собой уже знакомый вам конструктор `Worker()`, предназначенный для выделенных рабочих процессов. Как всегда, атрибут `scriptURL` объявляет путь к JavaScript-файлу с кодом рабочего процесса. Можно также добавить необязательный второй атрибут, чтобы указать имя рабочего процесса;
- `port`. При создании нового объекта SharedWorker для этого документа одновременно создается новый порт, который сохраняется в свойстве `port`. В дальнейшем посредством этого свойства происходят обращение к порту и обмен данными с рабочим процессом;
- `connect`. Это специальное событие, предназначенное для проверки наличия новых подключений изнутри рабочего процесса. Оно срабатывает каждый раз, когда из документа открывается новое соединение с рабочим процессом. Это событие удобно применять для отслеживания всех подключений, доступных для рабочего процесса (и обращения ко всем документам, использующим этот рабочий процесс);
- `start()`. Этот метод доступен для объектов MessagePort (один из объектов, возвращаемых при создании общего рабочего процесса). Он начинает отправку сообщений, приходящих на порт. Этот метод необходимо вызвать для того, чтобы открыть соединения после создания объекта SharedWorker.

Конструктор `SharedWorker()` возвращает объекты SharedWorker и MessagePort, а также значение порта, через который будет осуществляться соединение с рабочим процессом. Обмен данными с общим рабочим процессом может происходить только через порт, значение которого содержится в свойстве `port`.

Для тестирования общих рабочих процессов нам потребуются минимум два разных документа в одном источнике, два файла с JavaScript-кодом (по одному для каждого документа) и файл с кодом рабочего процесса.

Документ HTML для нашего примера включает в себя фрейм iframe, позволяющий загрузить второй документ в том же окне. И главный документ, и документ из iframe будут общаться с одним и тем же рабочим процессом (листинг 14.10).

Листинг 14.10. Шаблон для тестирования общих рабочих процессов

```
<!DOCTYPE html>
<html lang="ru">
<head>
  <title>Рабочие процессы</title>
  <link rel="stylesheet" href="webworkers.css">
  <script src="webworkers.js"></script>
</head>
<body>
  <section id="formbox">
    <form name="form">
      <p>Имя:<br><input type="text" name="name" id="name"></p>
      <p><input type="button" name="button" id="button"
          value="Отправить"></p>
    </form>
  </section>
  <section id="databox">
    <iframe id="iframe" src="iframe.html" width="500"
height="350"></iframe>
  </section>
</body>
</html>
```

Документ, определяющий iframe, — это простой код HTML с разделом `<section>` для уже знакомого нам блока `databox`, включающий в себя ссылку на файл iframe.js, в котором находится код для соединения с рабочим процессом (листинг 14.11).

Листинг 14.11. Шаблон для фрейма iframe (iframe.html)

```
<!DOCTYPE html>
<html lang="ru">
<head>
  <title>окно iframe</title>
  <script src="iframe.js"></script>
</head>
<body>
  <section id="databox"></section>
</body>
</html>
```

С каждым из документов HTML связан собственный JavaScript-код, открывающий соединение с рабочим процессом и обрабатывающий приходящие от него ответы. В обоих фрагментах необходимо создать объект SharedWorker и использовать для отправки и получения сообщений порт из свойства `port`. Давайте сначала рассмотрим код для главного документа.

Листинг 14.12. Подключение к рабочему процессу из главного документа (webworkers.js)

```
function initiate(){
  var button=document.getElementById('button');
  button.addEventListener('click', send, false);

  worker=new SharedWorker('worker.js');
  worker.port.addEventListener('message', received, false);
  worker.port.start();
}
function received(e){
  alert(e.data);
}
function send(){
  var name=document.getElementById('name').value;
  worker.port.postMessage(name);
}
window.addEventListener('load', initiate, false);
```

В каждом документе, который должен обращаться к общему рабочему процессу, необходимо создать объект SharedWorker и установить соответствующее соединение. В коде из листинга 14.12 мы создаем объект рабочего процесса, ссылаясь на файл worker.js, а затем выполняем операции по обмену данными, используя соответствующий порт, сохраненный в свойстве `port`.

После того как добавлен прослушиватель события `message`, обеспечивающий возможность отлавливания ответов рабочего процесса, мы вызываем метод `start()`, чтобы начать отправку сообщений. Подключение к общему рабочему процессу не создается до тех пор, пока не будет выполнен этот метод (если только мы не используем вместо метода `addEventListener()` обработчики событий, такие как `onmessage`).

Обратите внимание на то, что функция `send()` почти такая же, как в предыдущих примерах, только на этот раз обмен данными осуществляется с учетом значения свойства `port`.

Код фрейма также почти не меняется.

Листинг 14.13. Подключение из фрейма iframe (iframe.js)

```
function initiate(){
  worker=new SharedWorker('worker.js');
  worker.port.addEventListener('message', received, false);
  worker.port.start();
}
function received(e){
  var databox=document.getElementById('databox');
  databox.innerHTML=e.data;
}
window.addEventListener('load', initiate, false);
```

В обоих примерах кода объект SharedWorker создается с использованием ссылки на один и тот же файл (worker.js), а подключение устанавливается посредством свойства `port` (хотя значения портов и различаются). Единственное существенное отличие кода главного документа от кода iframe заключается в том, каким образом обрабатывается ответ от рабочего процесса. В главном документе функция `received()` создает всплывающее сообщение (см. листинг 14.12), а в iframe ответ выводится в форме простого текста внутри блока `databox` (см. листинг 14.13).

Теперь настало время посмотреть, как же общий рабочий процесс справляется со всеми этими подключениями, отправляя правильные ответы в соответствующие документы. Как вы помните, у нас есть только один рабочий процесс для обоих документов — именно поэтому он называется общим. Таким образом, рабочему процессу необходимо как-то различать запросы и сохранять подключения для использования в дальнейшем. Мы будем записывать ссылки на порты всех документов в массив под названием `myports`.

Листинг 14.14. Код общего рабочего процесса (worker.js)

```
myports=new Array();

addEventListener('connect', connect, false);

function connect(e){
  myports.push(e.ports[0]);
  e.ports[0].onmessage=send;
}
```

продолжение ⇩

Листинг 14.14 *(продолжение)*

```
function send(e){
  for(f=0; f < myports.length; f++){
    myports[f].postMessage('Ваше имя: '+e.data);
  }
}
```

Процедура аналогична той, которую мы описывали для выделенных рабочих процессов. На этот раз необходимо учитывать только то, какому документу отправляется ответ, так как к одному рабочему процессу одновременно могут подключаться несколько документов. Для этого мы используем массив `ports` события `connect`, в котором содержится значение последнего созданного порта (то есть в массиве есть только одно значение в позиции 0).

Каждый раз, когда код запрашивает подключение к рабочему процессу, срабатывает событие `connect`. В коде из листинга 14.14 это событие вызывает функцию `connect()`. В этой функции мы выполняем две операции. Во-первых, извлекаем значение порта из свойства `ports` (в позиции 0) и сохраняем в массиве `myports` (который инициализировали в начале рабочего процесса). Во-вторых, регистрируем обработчик события `onmessage` для этого конкретного порта и указываем, что при получении сообщения будет вызываться функция `send()`.

САМОСТОЯТЕЛЬНО

Для тестирования этого примера вам понадобится создать и загрузить на свой сервер несколько файлов. Создайте HTML-файл с шаблоном из листинга 14.10, присвоив ему любое название на выбор. Этот шаблон будет загружать тот же файл webworkers.css, который мы используем на протяжении всей главы, файл webworkers.js с кодом из листинга 14.12 и файл iframe.html в качестве источника для создания фрейма (этот файл должен содержать код из листинга 14.11). Кроме того, нужно создать файл с именем worker.js и поместить в него код рабочего процесса из листинга 14.14. После того как все файлы будут сохранены на сервере, откройте первый документ в своем браузере. Отправьте сообщение в рабочий процесс с помощью формы и посмотрите, как оба документа (главный документ и документ во фрейме) обработают ответ.

В результате, когда из главного кода в рабочий процесс отправляется сообщение — причем неважно, из какого документа, — происходит вызов функции `send()`. Внутри функции в цикле `for` мы извлекаем из массива `myports` все открытые порты для данного рабочего процесса и отправляем сообщения во все подключенные документы. Принцип работы ничем не отличается от описанного ранее для выделенных рабочих процессов, только на этот раз ответ отправляется в несколько документов, а не в один.

ВНИМАНИЕ

На момент написания этой главы общие рабочие процессы реализованы только в браузерах на базе механизма WebKit, таких как Google Chrome и Safari.

Краткий справочник. API Web Workers (Рабочие процессы)

API Web Workers (Рабочие процессы) реализует многозадачность в языке программирования JavaScript. Этот API позволяет обрабатывать код в фоновом режиме, не нарушая обычный ход выполнения главного документа.

Рабочие процессы

Существует два типа рабочих процессов: выделенные и общие. Тем не менее следующие методы и события работают одинаково для обоих типов:

- `postMessage(message)`. Этот метод отправляет сообщение рабочему процессу, главному коду или на соответствующий порт. В атрибуте `message` содержится строка или JSON-объект с нужным сообщением;
- `terminate()`. Этот метод останавливает рабочий процесс из главного кода;
- `close()`. Этот метод останавливает рабочий процесс изнутри него самого;
- `importScripts(file)`. Этот метод загружает внешний JavaScript-файл для добавления к рабочему процессу нового кода. Атрибут `file` содержит путь к нужному файлу;

- `message`. Это событие срабатывает, когда в код отправляется сообщение. Его можно использовать в рабочем процессе для прослушивания сообщений от главного кода и наоборот;
- `error`. Это событие срабатывает, когда в рабочем процессе происходит ошибка. Оно используется в главном коде для отслеживания ошибок рабочего процесса. Событие возвращает три свойства: `message`, `filename` и `lineno`. Свойство `message` представляет собой сообщение об ошибке, свойство `filename` содержит имя файла с кодом, вызвавшим ошибку, а свойство `lineno` возвращает номер строки, где произошла ошибка.

Выделенные рабочие процессы

Для создания выделенных рабочих процессов предназначен отдельный конструктор:

- `Worker(scriptURL)`. Возвращает объект Worker. Атрибут `scriptURL` содержит путь к файлу с кодом рабочего процесса.

Общие рабочие процессы

Из-за особенностей самой природы общих рабочих процессов в данном API потребовалось создать несколько методов, событий и свойств, предназначенных только для такого типа рабочих процессов:

- `SharedWorker(scriptURL)`. Этот конструктор возвращает объект SharedWorker. Атрибут `scriptURL` содержит путь к файлу с кодом общего рабочего процесса. Также можно добавить необязательный второй атрибут `name`, если требуется указать имя рабочего процесса;
- `port`. Это свойство возвращает значение порта, через который установлено соединение с рабочим процессом;
- `connect`. Это событие срабатывает в рабочем процессе, когда документ запрашивает новое подключение;
- `start()`. Этот метод запускает отправку сообщений. Он используется для открытия подключения к рабочему процессу.

15 API истории

Интерфейс History (История)

То, что в HTML5 обычно называют API истории (History API), в действительности представляет собой улучшенную версию старого API, который никогда официально реализован не был, но в течение многих лет поддерживался браузерами. Этот старый API состоял из набора методов и свойств, а также объекта History. В новом API истории этот объект усовершенствован, и все вместе теперь описано в официальной спецификации HTML5 как интерфейс History (История). В данном интерфейсе сохранены все старые методы и свойства, а также добавлены некоторые новые, позволяющие проверять и модифицировать историю браузера в соответствии с требованиями наших приложений.

Навигация по Сети

История браузера содержит список всех веб-страниц (их URL-адреса), на которые пользователь заходил в течение одного сеанса. Именно этот список лежит в основе всей функциональности навигации. Используя навигационные кнопки, которые в любом браузере можно найти у левого края навигационной полосы, мы можем переходить по списку вперед и назад, возвращаясь к ранее открытым документам. Список строится с учетом реальных URL-адресов, генерируемых веб-сайтами и указанных во всех гиперссылках содержащихся на них документов. Нажимая

кнопки со стрелками на панели браузера, мы можем загрузить веб-сайт, который посетили раньше, или вернуться на предыдущую страницу.

Несмотря на практичность кнопок браузера, иногда бывает удобнее перемещаться по истории внутри самого документа. Для эмуляции навигации из кода JavaScript всегда были доступны следующие методы и свойства:

- `back()`. Этот метод возвращает браузер на один шаг назад в истории сеанса (эмуляция стрелки влево);
- `forward()`. Этот метод переводит браузер на один шаг вперед в истории сеанса (эмуляция стрелки вправо);
- `go(steps)`. Этот метод переводит браузер назад или вперед на указанное число шагов в истории сеанса. Атрибут `steps` может принимать отрицательные или положительные значения в зависимости от того, в каком направлении нужно переместиться;
- `length`. Это свойство возвращает число записей в истории сеанса (общее число URL-адресов в списке).

Перечисленные методы и свойства объявляются как часть объекта History с помощью таких выражений, как, например, `history.back()`. Также можно через объект Window ссылаться на окно, но это не обязательно. Например, вернуться на предыдущую веб-страницу можно с помощью кода `window.history.back()` или `window.history.go(-1)`.

ВНИМАНИЕ

Эта часть API уже известна и используется большинством современных веб-дизайнеров и программистов. Мы не будем приводить в этой главе примеры использования перечисленных методов, однако вы всегда можете посетить наш веб-сайт и изучить ссылки для данной главы, чтобы получить дополнительную информацию.

Новые методы

Когда использование объекта XMLHttpRequest стало привычным делом, а Ajax-приложения с успехом завоевали Сеть, принципы навигации и доступа к документам навсегда изменились. Стало распространенной практикой добавлять на страницы небольшие сценарии, умеющие извлекать информацию с серверов и отображать ее в текущем документе без необходимости перезагружать страницу или загружать новую. Пользователи

взаимодействуют с современными веб-сайтами и приложениями, не выходя за пределы одного URL. Они получают информацию, вводят данные и собирают результаты обработки, не покидая одну-единственную страницу. Сеть перешла на новый уровень и теперь эмулирует настольные приложения.

Однако браузеры все так же отслеживают активность пользователя через URL-адреса. По сути, URL-адреса — это данные из списка навигации, указывающие, где пользователь находится. Но поскольку в новых веб-приложениях разработчики стараются не использовать URL-адреса, очевидно, что теряются важные шаги процесса. Пользователи могут обновить данные на веб-странице десяток раз, однако в списке истории не останется никаких записей о том, какие действия были выполнены.

В существующий API истории были добавлены новые методы и свойства, позволяющие вручную из JavaScript-кода модифицировать URL-адреса на панели Location (Местоположение), а также в списке истории. Теперь мы можем с легкостью добавлять фальшивые URL-адреса в список истории, тщательно отслеживая активность пользователя.

- `pushState(state, title, url)`. Этот метод создает новую запись в истории сеанса. Атрибут `state` объявляет значение состояния записи. Он используется для идентификации записи впоследствии и может содержать строку или JSON-объект. Атрибут `title` представляет собой заголовок записи, а атрибут `url` — это URL-адрес создаваемой записи (он заменит текущий URL-адрес на панели Location (Местоположение));
- `replaceState(state, title, url)`. Этот метод работает аналогично `pushState()`, но не создает новую запись. Вместо этого он заменяет информацию в текущей записи;
- `state`. Это свойство возвращает значение состояния текущей записи. Всегда равно `null`, если только не было объявлено в одном из перечисленных ранее методов посредством атрибута `state`.

Фальшивые URL-адреса

URL-адреса, которые генерируются с использованием методов, подобных `pushState()`, — фальшивые в том смысле, что браузер никогда не проверяет их достоверность и существование документов, на которые они указывают. Наша задача — сделать так, чтобы фальшивые URL-адреса содержали правильную и полезную информацию.

Для создания новой записи в истории браузера и изменения URL-адреса на панели Location (Местоположение) необходимо применить метод `pushState()`. Давайте посмотрим, как это работает.

Листинг 15.1. Базовый шаблон для тестирования API истории

```
<!DOCTYPE html>
<html lang="ru">
<head>
  <title>API истории браузера</title>
  <link rel="stylesheet" href="history.css">
  <script src="history.js"></script>
</head>
<body>
  <section id="maincontent">
    Это содержимое никогда не обновляется<br>
    <span id="url">страница 2</span>
  </section>
  <aside id="databox"></aside>
</body>
</html>
```

В листинге 15.1 представлен HTML-код с базовыми элементами, необходимыми для тестирования API истории. Внутри элемента `<section>` находятся постоянное содержимое, идентифицируемое значением `maincontent`, текст, который мы превратим в ссылку для генерирования виртуальной второй страницы веб-сайта, и обычный блок `databox` для альтернативного содержимого.

Давайте добавим к документу стили.

Листинг 15.2. Стили для полей и элементов (history.css)

```
#maincontent{
  float: left;
  padding: 20px;
  border: 1px solid #999999;
}
#databox{
  float: left;
  width: 500px;
  margin-left: 20px;
```

```
  padding: 20px;
  border: 1px solid #999999;
}
#maincontent span{
  color: #0000FF;
  cursor: pointer;
}
```

В этом примере мы добавим в историю новую запись посредством метода `pushState()` и обновим содержимое окна, не перезагружая страницу и не загружая другой документ.

Листинг 15.3. Генерация нового URL-адреса и содержимого (history.js)

```
function initiate(){
  databox=document.getElementById('databox');
  url=document.getElementById('url');
  url.addEventListener('click', changepage, false);
}
function changepage(){
  databox.innerHTML='url-адрес – page2';
  window.history.pushState(null, null, 'page2.html');
}
window.addEventListener('load', initiate, false);
```

В функции `initiate()` из листинга 15.3 мы создали необходимую ссылку на элемент `databox` и добавили к элементу `<span>` прослушиватель события `click`. Когда пользователь щелкает на тексте внутри `<span>`, происходит вызов функции `changepage()`.

Функция `changepage()` выполняет две задачи: обновляет содержимое страницы, добавляя новую информацию, и вставляет в список истории новый URL-адрес. После выполнения функции в поле `databox` отображается текст «url-адрес — страница 2», а URL-адрес главного документа на панели Location (Местоположение) заменяется фальшивым URL page2.html.

Атрибуты `state` и `title` для метода `pushState()` на этот раз объявлялись со значением `null`. В настоящее время атрибут `title` ни в одном из браузеров не используется, поэтому ему всегда присваивается значение `null`, однако атрибут `state` мы настроим и применим в следующих примерах.

САМОСТОЯТЕЛЬНО

Скопируйте шаблон из листинга 15.1 в HTML-файл. Создайте CSS-файл с названием history.css и поместите в него стили из листинга 15.2. Также создайте JavaScript-файл history.js с кодом из листинга 15.3. Загрузите все файлы на свой сервер и откройте в браузере HTML-файл. Щелкните на тексте «страница 2» и посмотрите на URL-адрес на панели Location (Местоположение) — он примет новое значение, определенное в коде.

Возможности отслеживания

Пока что мы всего лишь немного изменили историю сеанса. Мы заставили браузер поверить, что пользователь посетил страницу, URL-адреса которой даже не существует. После того как вы щелкнули на ссылке «страница 2», на панели Location (Местоположение) отобразился фальшивый URL-адрес page2.html, а внутрь `databox` добавилось новое содержимое и ни для одной из этих операций не потребовалось обновлять страницу или загружать другой документ. Это симпатичный трюк, но он еще не завершен. Браузер пока что не считает новый URL-адрес реальным документом. Если вы с помощью кнопок навигации перейдете на шаг назад, а затем на шаг вперед в истории сеанса, то URL-адрес изменится. Новый фальшивый адрес сменится старым адресом главного документа, однако содержимое документа останется неизменным. Нам необходимо отслеживать ситуацию повторного посещения фальшивых URL-адресов и выполнять соответствующие модификации в документе, показывая пользователю правильное состояние.

Раньше мы упоминали свойство `state`. Значение этого свойства можно устанавливать во время создания нового URL-адреса, и именно с помощью него позднее мы определяем, каким должен быть текущий веб-адрес. Для работы с этим свойством в API предусмотрено новое событие `popstate`. Оно срабатывает в определенных ситуациях: когда пользователь повторно заходит на URL-адрес или при повторной загрузке документа. У него есть свойство `state`, содержащее значение состояния, объявленное при генерации URL-адреса методом `pushState()` или `replaceState()`. Это значение равно `null`, если URL-адрес реальный, но при условии, что до этого мы не поменяли его с помощью `replaceState()`. Соответствующий пример рассмотрим в дальнейшем.

В следующем коде усовершенствуем предыдущий пример, добавив событие `popstate` и метод `replaceState()` для распознавания того, какой именно URL-адрес пользователь запрашивает в данный момент.

Листинг 15.4. Отслеживание позиции пользователя (history.js)

```
function initiate(){
  databox=document.getElementById('databox');
  url=document.getElementById('url');
  url.addEventListener('click', changepage, false);
  window.addEventListener('popstate', newurl, false);
  window.history.replaceState(1, null);
}
function changepage(){
  showpage(2);
  window.history.pushState(2, null, 'page2.html');
}
function newurl(e){
  showpage(e.state);
}
function showpage(current){
  databox.innerHTML='url-адрес — page '+current;
}
window.addEventListener('load', initiate, false);
```

Мы в своем приложении должны проделать две операции, для того чтобы получить полный контроль над ситуацией. Во-первых, необходимо объявить значение состояния для всех URL-адресов, которые мы собираемся использовать, — как для фальшивых, так и для реальных. А во-вторых, нужно настроить обновление содержимого документа в соответствии с текущим URL.

В функции `initiate()` из листинга 15.4 мы добавили прослушиватель события `popstate`. Он будет вызывать функцию `newurl()` каждый раз при повторном посещении URL-адреса. Данная функция всего лишь обновляет содержимое элемента `databox`, указывая адрес текущей страницы. Она принимает значение свойства `state` и отправляет его функции `showpage()`, которая выводит информацию на экран.

Это работает для любых фальшивых URL-адресов, но, как уже говорилось, у реальных URL-адресов состояние по умолчанию не задано. Используя метод `replaceState()` в конце функции `initiate()`, мы меняем информацию о текущей записи (реальном URL главного документа),

присваивая ей значение состояния 1. Теперь, когда пользователь будет возвращаться к главному документу, мы будем узнавать об этом по значению `state`.

Функция `changepage()` почти не изменилась — она всего лишь вызывает функцию `showpage()` для обновления содержимого документа. Кроме того, здесь мы присваиваем фальшивому URL-адресу значение состояния 2.

Приложение работает следующим образом: когда пользователь щелкает на ссылке «страница 2», на экране отображается сообщение «url-адрес — страница 2», а URL-адрес на панели Location (Местоположение) изменяется на page2.html (но все так же включает в себя полный путь, разумеется). Это мы делали и раньше, а самое интересное начинается сейчас. Если пользователь нажимает на стрелку влево на панели навигации браузера, то на панели Location (Местоположение) появляется предыдущий URL-адрес из списка истории (то есть реальный URL-адрес нашего документа) и срабатывает событие `popstate`. Это событие вызывает функцию `newurl()`, которая считывает значение свойства `state` и отправляет его функции `showpage()`. Теперь значение состояния равно 1 (мы объявили его для данного URL-адреса в методе `replaceState()`), кроме того, сообщение на экране меняется на «url-адрес — страница 1». Если пользователь возвращается к фальшивому URL-адресу, щелкнув на стрелке вправо на панели навигации, то значение состояния меняется на 2 и на экране снова отображается сообщение «url-адрес — страница 2».

Как видите, значение свойства `state` может быть любым, но оно необходимо для проверки того, какой URL-адрес является текущим. Основываясь на значении данного свойства, вы можете адаптировать содержимое документа к актуальной ситуации.

САМОСТОЯТЕЛЬНО

Используйте фрагменты кода из листингов 15.1 и 15.2 для создания HTML-документа и документа со стилями CSS. Скопируйте код из листинга 15.4 в файл history.js и загрузите все файлы на свой сервер. Откройте шаблон HTML в браузере и щелкните на тексте «страница 2». Значение в адресной строке и содержимое поля databox изменятся в соответствии с новым URL-адресом. Несколько раз нажмите на кнопки Назад и Вперед на панели навигации и посмотрите, как URL-адрес и содержимое, связанное с этим URL-адресом, будут обновляться на экране (содержимое документа соответствует текущему состоянию).

ВНИМАНИЕ

URL-адрес page2.html, который мы генерируем в предыдущих примерах с помощью метода pushState(), считается фальшивым, но он должен быть реальным. Назначение этого API — не создавать фальшивые адреса, а предоставлять программистам альтернативный способ сохранения пользовательской активности в истории браузера, для того чтобы пользователь в любой момент (даже после закрытия браузера) мог вернуться к предыдущему состоянию документа. Вы должны удостовериться в том, что код на серверной стороне возвращает это состояние и обеспечивает правильное содержимое для каждого запрошенного URL-адреса (как реального, так и фальшивого).

Реальный пример

В следующем примере рассмотрим более практичное приложение. Мы воспользуемся возможностями API истории и изученными ранее методами для загрузки набора из четырех изображений одного документа. Каждое изображение свяжем с фальшивым URL-адресом, который можно будет в дальнейшем использовать для запроса определенного изображения с сервера.

Главный документ загружается с изображением по умолчанию. Это изображение связывается с первой из четырех ссылок, являющихся частью постоянного содержимого. Все эти ссылки указывают на фальшивые URL-адреса и ссылаются на состояния, а не на реальные документы. То же самое относится к ссылке на главный документ, которая заменяется на page1.html. Вам станет понятнее, что здесь происходит, когда мы чуть позже подробно разберем пример.

Листинг 15.5. Шаблон для «реального» приложения

```
<!DOCTYPE html>
<html lang="ru">
<head>
  <title>API истории</title>
  <link rel="stylesheet" href="history.css">
  <script src="history.js"></script>
</head>
```

продолжение ⇩

Листинг 15.5 *(продолжение)*

```
<body>
  <section id="maincontent">
    Это содержимое никогда не обновляется<br>
    <span id="url1">изображение 1</span> -
    <span id="url2">изображение 2</span> -
    <span id="url3">изображение 3</span> -
    <span id="url4">изображение 4</span> -
  </section>
  <aside id="databox">
    <img id="image"
     src="http://www.minkbooks.com/content/monster1.gif">
  </aside>
</body>
</html>
```

Единственное различие между новым и предыдущим приложениями заключается в количестве ссылок и создаваемых URL-адресов. В коде из листинга 15.4 было два состояния: 1, соответствующее главному документу, и 2 для фальшивого URL-адреса page2.html, который мы создавали с помощью метода `pushState()`. В данном примере необходимо автоматизировать процесс, чтобы создать четыре фальшивых URL-адреса, соответствующих каждому из доступных изображений.

Листинг 15.6. Манипулирование историей (history.js)

```
function initiate(){
  for(var f=1;f<5;f++){
    url=document.getElementById('url'+f);
    url.addEventListener('click', function(x){
      return function(){ changepage(x);}
    }(f), false);
  }

  window.addEventListener('popstate', newurl, false);

  window.history.replaceState(1, null, 'page1.html');
}
function changepage(page){
  showpage(page);
  window.history.pushState(page, null, 'page'+page+'.html');
}
```

```
function newurl(e){
  showpage(e.state);
}
function showpage(current){
  if(current!=null){
    image=document.getElementById('image');
    image.src='http://www.minkbooks.com/content/monster' +
    current + '.gif';
  }
}
window.addEventListener('load', initiate, false);
```

Как видите, мы используем те же функции, но с заметными изменениями. Во-первых, внутри функции `initiate()` передаем методу `replaceState()` атрибут `url` со значением `page1.html`. Мы решили запрограммировать наше приложение таким способом: объявить для главного документа состояние 1 и связать с ним URL-адрес page1.html (независимо от реального URL-адреса документа). Благодаря этому становится проще переходить от одного URL к другому, так как для построения адресов мы всегда используем один и тот же формат, меняется только значение свойства `state`. В этом можно убедиться, изучив код функции `changepage()`. Каждый раз, когда пользователь щелкает на одной из ссылок в шаблоне, происходит вызов этой функции, фальшивый URL-адрес собирается с учетом значения переменной `page` и добавляется в список истории. Значение, получаемое этой функцией, заранее устанавливается в цикле `for` в начале функции `initiate()`. Мы задаем значение 1 для ссылки url1, 2 — для url2 и т. д.

Когда пользователь посещает любой URL-адрес, выполняется функция `showpage()`. Она обновляет содержимое страницы (изображение) в соответствии с выбранным URL. Поскольку иногда, когда срабатывает событие `popstate`, значение свойства `state` равно `null` (например, после самой первой загрузки главного документа), прежде чем делать что-либо еще, мы проверяем значение, полученное функцией `showpage()`. Если оно отличается от `null`, значит, для данного URL-адреса определено свойство `state`, и на экран выводится изображение, соответствующее этому состоянию.

В этом примере используются изображения monster1.gif, monster2.gif, monster3.gif и monster4.gif, и их нумерация соответствует значениям свойства `state`. Таким образом, мы можем ориентироваться на значение этого свойства для выбора изображения, которое будет выведено на экран.

Однако всегда необходимо помнить, что это значение может быть абсолютно произвольным и процесс создания фальшивых URL-адресов и соответствующего содержимого необходимо настроить в зависимости от требований конкретного приложения.

Также помните, что у пользователей должна быть возможность вернуться к любому из сгенерированных приложением URL-адресов и во всех случаях увидеть на экране правильное содержимое. Вы должны подготовить серверную сторону для обработки этих URL, гарантировав, что любое состояние всегда будет доступно и готово к использованию. Например, пользователь открывает новое окно и вводит URL-адрес page2.html. В этом случае сервер должен возвратить главный документ с изображением monster2.gif, а не просто шаблон из листинга 15.5. Основной смысл этого API в том и заключается, чтобы обеспечить для пользователей возможность в любой момент вернуться к любому предыдущему состоянию, а сделать это можно только одним способом — превратив фальшивые URL в допустимые.

ВНИМАНИЕ

В листинге 15.6 мы использовали цикл for для добавления прослушивателя события click к каждому элементу в документе. Здесь мы воспользовались техникой JavaScript, позволяющей отправлять функциям реальные значения. Для того чтобы отправить функции обратного вызова в методе addEventListener() конкретное значение, необходимо задать само это значение, а не содержащую его переменную. Если использовать переменную, то в действительности функция получит не значение, а ссылку на него. Таким образом, для отправки текущего значения переменной f цикла for нам пришлось воспользоваться двумя анонимными функциями. Первая функция выполняется в момент вызова метода addEventListener(). Она получает текущее значение переменной f (обратите внимание на круглые скобки в конце) и помещает его в переменную x. Затем функция возвращает вторую анонимную функцию со значением переменной x. Именно вторая функция выполняется с правильным значением при срабатывании события. Чтобы побольше узнать об этой технике, зайдите на наш веб-сайт и изучите ссылки для этой главы.

САМОСТОЯТЕЛЬНО

Для тестирования последнего примера используйте HTML-документ из листинга 15.5 и файл со стилями CSS из листинга 15.2. Скопируйте код из листинга 15.6 в файл history.js и загрузите все файлы на свой сервер. Откройте шаблон в браузере и пощелкайте на ссылках. Затем испытайте перемещение по уже посещенным URL-адресам с помощью навигационных кнопок браузера. Изображения на экране должны меняться в соответствии с тем, какой URL-адрес отображается на панели Location (Местоположение).

Краткий справочник. API History (История)

API History (История) позволяет манипулировать хранящейся в браузере историей сеанса, отслеживая таким образом активность пользователя в пределах одного документа. Этот API добавлен в официальную спецификацию и носит название интерфейса History (История). В данном интерфейсе старые и давно существующие методы и свойства объединяются с новыми:

- `length`. Это свойство возвращает общее число записей в списке истории;
- `state`. Это свойство возвращает состояние для текущего URL-адреса;
- `go(step)`. Этот метод переводит браузер назад или вперед на несколько шагов в списке истории в зависимости от значения атрибута `step`. Значение атрибута может быть отрицательным и положительным, определяя разные направления перемещения;
- `back()`. Этот метод загружает предыдущий URL-адрес в списке истории;
- `forward()`. Этот метод загружает следующий URL-адрес в списке истории;
- `pushState(state, title, url)`. Этот метод вставляет новые данные в список истории. Атрибут `state` содержит значение состояния, которое мы хотим связать с новой записью. Атрибут `title` представляет собой заголовок записи. Атрибут `url` — это новый URL-адрес, который мы добавляем в список истории;

- `replaceState(state, title, url)`. Этот метод модифицирует текущую запись в списке истории. Атрибут `state` содержит значение состояния, которое мы хотим поместить в текущую запись. Атрибут `title` представляет собой заголовок записи. А атрибут `url` — это новый URL-адрес, который мы добавляем в текущую запись списка истории;
- `popstate`. Это событие срабатывает в определенных обстоятельствах и сообщает о текущем состоянии.

16 API автономной работы

Манифест кэша

Дни, когда нам приходилось работать без подключения к Сети, остались далеко позади. Но эта глава называется «API автономной работы» — не противоречим ли мы сами себе? Давайте подумаем. Мы почти всю жизнь работали автономно. Настольные приложения были основными средствами производства. А затем Сеть внезапно превратилась в новую рабочую платформу. Интерактивные приложения становятся все сложнее и сложнее, а HTML5 еще больше затрудняет выбор победителя в противостоянии интерактивных и автономных приложений. Базы данных, доступ к файлам и файловые хранилища, графические инструменты, редактирование изображений и видео и пр. составляют основополагающий набор возможностей, доступных сегодня в Сети. Наша ежедневная деятельность вращается вокруг Сети, а промышленные окружения становятся интерактивными. Время автономной работы прошло.

Однако по мере перехода к полной интерактивности веб-приложения становятся все более замысловатыми. Они работают с объемными файлами и требуют много времени для загрузки. К тому моменту, когда сетевые приложения полностью заменят собой настольные, работать в Сети станет попросту невозможно. Пользователи будут не в состоянии загружать несколько мегабайт файлов каждый раз, когда у них возникнет необходимость обратиться к другому приложению. Кроме того, невозможно

рассчитывать на то, что доступ к Интернету будет работать всегда, без единого исключения. Автономные приложения скоро прекратят существование, но в текущих обстоятельствах интерактивные приложения также обречены на провал.

На помощь приходит API Offline (Автономная работа). Фактически, этот API предоставляет альтернативное решение для сохранения приложений и веб-файлов на компьютере пользователя для дальнейшей работы. Одного обращения к Сети достаточно для загрузки всех файлов, необходимых для запуска приложения и его использования как в сетевом, так и в автономном окружении. После того как загрузка файлов завершается, приложение выполняется в браузере, используя эти файлы, как и любое настольное приложение. Этот процесс не зависит от того, что происходит на сервере, и даже от наличия сетевого подключения.

Файл манифеста

Веб-приложение или сложный веб-сайт состоят из нескольких файлов, но не все они нужны для выполнения приложения и не все должны храниться на компьютере пользователя. В этом API определяется специальный файл, внутри которого объявляется список файлов, необходимых для автономной работы. Это всего лишь текстовый файл, называемый манифестом. Внутри него находится список URL-адресов, указывающих местоположение запрашиваемых файлов. Манифест можно создать в любом текстовом редакторе. Файл нужно сохранить с расширением .manifest, а его содержимое должно начинаться со строки `CACHE MANIFEST`, как в следующем примере (листинг 16.1).

Листинг 16.1. Файл манифеста кэша

```
CACHE MANIFEST
cache.html
cache.css
cache.js
```

То, что перечислено под строкой `CACHE MANIFEST`, — это полный список файлов, которые нужны приложению для работы на пользовательском компьютере без обращения к внешним ресурсам. В нашем примере файл cache.html используется как главный документ приложения, файл cache.css содержит стили CSS, а в файле cache.js находятся коды JavaScript.

Категории

Мы можем объявить файлы, необходимые приложению для работы в автономном режиме, но точно так же можно объявить, какие файлы должны быть доступны только через сетевое подключение. Они могут относиться к той части приложения, которая имеет смысл только при интерактивной работе, — например, чату.

Для идентификации типов файлов, перечисленных в файле манифеста, в API предусмотрены три категории:

- `CACHE`. Это категория по умолчанию. Все файлы из этой категории сохраняются на компьютер пользователя для дальнейшего использования;
- `NETWORK`. Считается белым списком; файлы в ней доступны только через сетевое подключение;
- `FALLBACK`. Предназначена для файлов, которые хорошо было бы получать с сервера при наличии подключения, но которые все же можно заменять автономной версией. Если браузер обнаружит, что Сеть доступна, он попытается загрузить исходный файл. В противном случае будет загружен подменный файл с компьютера пользователя.

Наш файл манифеста с использованием категорий мог бы выглядеть примерно так, как на листинге 16.2.

Листинг 16.2. Объявление файлов по категориям

```
CACHE MANIFEST

CACHE:
cache.html
cache.css
cache.js

NETWORK:
chat.html

FALLBACK:
newslist.html nonews.html
```

В новом файле манифеста из листинга 16.2 файлы перечислены в соответствии с категориями. Три файла из категории `CACHE` будут загружаться, сохраняться на пользовательском компьютере и затем постоянно

использоваться при выполнении приложения (до тех пор, пока мы не объявим какие-либо другие условия кэширования файлов). Файл chat.html из категории NETWORK будет доступен только при наличии сетевого подключения. Что же касается файла newslist.html из категории FALLBACK, то, когда возможно, будет использоваться его серверная версия, в случае невозможности доступа к нему он будет заменяться файлом nonews.html. Так же, как и файлы из категории CACHE, файл nonews.html кэшируется и сохраняется на компьютере пользователя, для того чтобы к нему можно было обратиться в любой момент.

Категорию FALLBACK удобно использовать не только для замены отдельных файлов, но и для определения ссылок на целые каталоги. Например, строка вида `/ noconnection.html` означает, что любой файл, недоступный в кэше, заменяется файлом noconnection.html. С помощью такой техники можно выводить на экран документ с просьбой проверить сетевое подключение в случае, если пользователь попытается обратиться к той части приложения, которая в автономном режиме недоступна.

Комментарии

Для добавления комментариев в файл манифеста используется символ # (по одному символу на строку). Поскольку файлы упорядочиваются по категориям, может показаться, что в комментариях нет никакого смысла. Однако они важны для настройки обновлений. Файл манифеста определяет не только то, какие файлы должны кэшироваться, но также когда это должно происходить. У браузера нет другого способа узнать об обновлении файлов приложения, кроме как через файл манифеста. Если обновленные файлы абсолютно такие же и никаких новых файлов в список не добавлено, то файл манифеста не меняется и браузер, считая, что никаких изменений не было, продолжает использовать уже кэшированные старые файлы. Для того чтобы заставить браузер снова загрузить файлы приложения, необходимо сообщить об обновлении, используя комментарии. Чаще всего достаточно бывает одного комментария с датой последнего обновления, как в следующем примере.

Листинг 16.3. Комментарии для информирования об обновлении

```
CACHE MANIFEST

CACHE:
cache.html
```

```
cache.css
cache.js

NETWORK:
chat.html

FALLBACK:
newslist.html nonews.html
# date 2011/08/10
```

Предположим, вы доработали код функций в файле cache.js. На компьютерах пользователей этот файл уже кэширован, и браузеры продолжают использовать старую версию вместо новой. Если поменять дату в конце файла манифеста или добавить новые комментарии, то браузеры узнают об обновлении и повторно загрузят все файлы, включая улучшенную версию cache.js. После обновления кэша приложение будет выполняться в браузере уже с новыми копиями файлов.

Использование файла манифеста

После того как все файлы, необходимые для выполнения приложения, выбраны и подготовлен полный список указывающих на эти файлы URL-адресов, необходимо добавить файл манифеста в наши документы. В API предусмотрен новый атрибут элемента `<html>`, позволяющий задать местоположение манифеста.

Листинг 16.4. Загрузка файла манифеста

```
<!DOCTYPE html>
<html lang="ru" manifest="mycache.manifest">
<head>
  <title>API Автономной работы</title>
  <link rel="stylesheet" href="cache.css">
  <script src="cache.js"></script>
</head>
<body>
  <section id="databox">
    Автономное приложение
  </section>
</body>
</html>
```

В листинге 16.4 показан небольшой HTML-документ, в котором внутри элемента `<html>` присутствует атрибут `manifest`. Атрибут `manifest` указывает местоположение файла манифеста, который используется для создания кэша приложения. В остальном документ не меняется: как обычно, мы добавляем файлы со стилями CSS и кодами JavaScript, и это происходит независимо от содержимого файла манифеста.

Файл манифеста необходимо сохранить с расширением .manifest, а имя вы можете выбрать по своему усмотрению (в нашем примере это mycache). Когда браузер обнаруживает в документе атрибут `manifest`, он пытается сначала загрузить файл манифеста, а затем все ресурсы, перечисленные внутри него. Атрибут `manifest` нужно добавить во все документы HTML, которые должны входить в кэш приложения. Процесс абсолютно прозрачен для пользователей, и его можно контролировать программно посредством API, как мы вскоре увидим.

Помимо расширения файла манифеста и его внутренней структуры необходимо принять во внимание еще одну важную особенность. Файл манифеста серверы должны обслуживать с применением правильного MIME-типа. С каждым файлом связан MIME-тип, указывающий на формат его содержимого. Например, MIME-тип для HTML-файла — это `text/html`. Файл манифеста необходимо обслуживать с использованием MIME-типа `text/cache-manifest`, в противном случае браузер вернет ошибку.

ВНИМАНИЕ

В настоящее время MIME-тип text/cache-manifest не входит в конфигурацию браузеров по умолчанию. Его необходимо добавить на сервер вручную. Процесс добавления зависит от того, какой тип сервера у вас установлен. Для некоторых версий Apache, например, достаточно добавить в файл httpd.conf строку AddType text/cache-manifest .manifest. После этого сервер начинает обслуживать файлы манифеста с использованием правильного типа.

API автономной работы

Для генерации кэша небольших веб-сайтов или простых приложений достаточно всего одного файла манифеста, однако сложные приложения

требуют более серьезного контроля. В файле манифеста перечисляются файлы, которые должны быть кэшированы, но он не в состоянии сообщить, сколько файлов уже было загружено, какие ошибки возникли в процессе, не появилось ли свежих обновлений, а также предоставить информацию о других критических ситуациях. Для того чтобы учесть эти возможные сценарии, в API появился новый объект ApplicationCache, предлагающий методы, свойства и события для управления процессом от начала и до конца.

Ошибки

Вероятно, самым важным событием объекта ApplicationCache можно считать событие `error`. Если в процессе считывания файлов с сервера происходит ошибка, то приложение не кэшируется либо кэш не обновляется. Чрезвычайно важно получать оперативную информацию о таких состояниях, для того чтобы предпринимать соответствующие действия.

Используя HTML-документ из листинга 16.4, мы построим небольшое приложение и посмотрим, как работает событие `error`.

Листинг 16.5. Проверка ошибок

```
function initiate(){
  var cache=window.applicationCache;
  cache.addEventListener('error', showerror, false);
}
function showerror(){
  alert('error');
}
window.addEventListener('load', initiate, false);
```

Атрибут `applicationCache` объекта `window`, который мы использовали в коде из листинга 16.5, возвращает объект ApplicationCache для данного документа. После сохранения ссылки на этот объект в переменной `cache` мы добавляем к объекту прослушиватель события `error`. При срабатывании события прослушиватель будет вызывать функцию `showerror()`, а эта функция, в свою очередь, сообщит нам об ошибке.

Файл CSS должен включать в себя стили для элемента `<section>` нашего документа. Создайте собственные стили или используйте следующие (листинг 16.6).

Листинг 16.6. Правило CSS для элемента databox

```
#databox{
  width: 500px;
  height: 300px;
  margin: 10px;
  padding: 10px;
  border: 1px solid #999999;
}
```

САМОСТОЯТЕЛЬНО

Создайте HTML-файл с кодом из листинга 16.4, JavaScript-файл под именем cache.js с кодом из листинга 16.5 и файл манифеста с названием mycache.manifest. Как уже говорилось, вам понадобится добавить в файл манифеста полный список файлов, подлежащих кэшированию, поместив их в категорию CACHE. В нашем примере это файл HTML, файл cache.js, а также файл cache.css, содержащий стили для документа. Загрузите эти файлы на свой сервер и откройте HTML-файл в браузере. Если вы сотрете файл манифеста или забудете добавить соответствующий MIME-тип для этого файла в настройки сервера, то сработает событие error. Также для того, чтобы пронаблюдать автономную работу приложения с использованием нового кэша, вы можете разорвать подключение к Интернету или включить настройку Work Offline (Работать автономно) в Firefox.

ВНИМАНИЕ

Сейчас реализация API Offline (Автономная работа) в большинстве браузеров находится на экспериментальной стадии. Мы рекомендуем тестировать примеры из этой главы в браузерах Firefox и Google Chrome. В Firefox есть настройка для деактивации подключения и перехода к автономной работе (выберите вариант Work Offline (Работать автономно) в меню Developer (Разработчик)). Кроме того, Firefox — это единственный браузер, позволяющий стирать кэш, что значительно упрощает его изучение (перейдите в меню Options (Параметры)▸Advanced (Расширенные)▸Network (Сеть) и выберите кэш, который должен быть очищен). В то же время в Google Chrome реализованы почти все доступные события, и он позволяет экспериментировать с любыми возможностями.

Online и Offline

У объекта Navigator появилось новое свойство под названием `onLine`, указывающее на текущее состояние подключения. С этим свойством связаны два события, которые срабатывают при изменении значения свойства. Свойство и события не принадлежат объекту ApplicationCache, но весьма полезны для работы с этим API:

- `online`. Срабатывает, когда значение свойства `onLine` меняется на `true`;
- `offline`. Срабатывает, когда значение свойства `onLine` меняется на `false`.

Далее представлен пример использования этих событий.

Листинг 16.7. Проверка состояния подключения

```
function initiate(){
  databox=document.getElementById('databox');

  window.addEventListener('online', function(){ state(1); },
false);
  window.addEventListener('offline', function(){ state(2); },
false);
}
function state(value){
  switch(value){
    case 1:
      databox.innerHTML+='<br>Подключение к Сети есть';
      break;
    case 2:
      databox.innerHTML+='<br>Подключения к Сети нет';
      break;
  }
}
window.addEventListener('load', initiate, false);
```

В коде из листинга 16.7 мы использовали анонимные функции для обработки событий и отправки значения функции `state()`, которая выводит соответствующее сообщение внутри элемента `databox`. События будут срабатывать при каждом изменении значения свойства `onLine`.

ВНИМАНИЕ

Невозможно гарантировать, что значение свойства всегда будет правильным. Прослушивание этих событий на настольном компьютере, вероятно, не даст никакого результата, даже если отключить компьютер от Интернета. Для тестирования данного примера на ПК мы рекомендуем использовать настройку Work Offline (Работать автономно), доступную в Firefox.

САМОСТОЯТЕЛЬНО

Используйте те же файлы HTML и CSS, что и в предыдущем примере. Скопируйте код из листинга 16.7 в файл cache.js. Используя возможности Firefox, сотрите кэш вашего веб-сайта, а затем загрузите это приложение. Для тестирования событий можно использовать настройку Work Offline (Работать автономно) в меню Firefox. Каждый раз, когда вы щелкаете на этом пункте меню, рабочие условия меняются и внутри элемента databox выводится новое сообщение.

Обработка кэша

Создание или обновление кэша может занять от нескольких секунд до нескольких минут в зависимости от размера загружаемых файлов. Процесс проходит в несколько этапов в соответствии с тем, что браузер в состоянии сделать в каждый момент. При обычном обновлении, например, браузер сначала пытается прочитать файл манифеста, для того чтобы узнать о возможных обновлениях, затем, если обновление было, загрузить все перечисленные в манифесте файлы и в конце проинформировать о завершении процесса. Для информирования о текущем этапе процесса в API предусмотрено свойство `status`. Оно может принимать следующие значения:

- `UNCACHED` (значение 0). Указывает, что для данного приложения кэш еще не создан;
- `IDLE` (значение 1). Указывает, что кэш для приложения актуален и используется;
- `CHECKING` (значение 2). Указывает, что браузер проверяет обновления;

- DOWNLOADING (значение 3). Указывает, что происходит загрузка файлов кэша;
- UPDATEREADY (значение 4). Указывает, что кэш приложения доступен и используется, но файлы не актуальны; есть готовое обновление для замены устаревших файлов;
- OBSOLETE (значение 5). Указывает, что текущий кэш больше не используется.

Значение свойства status можно проверять в любой момент, но лучше для проверки этапа процесса и состояния кэша использовать события объекта ApplicationCache. Следующие события обычно срабатывают последовательно, и часть из них связана с определенными состояниями кэша приложения:

- checking. Срабатывает, когда браузер проверяет наличие обновлений;
- noupdate. Срабатывает, если изменения в файле манифеста не обнаружены;
- downloading. Срабатывает, когда браузер обнаруживает новое обновление и начинает загрузку файлов;
- cached. Срабатывает, когда кэш готов;
- updateready. Срабатывает, когда процесс загрузки обновления завершается;
- obsolete. Срабатывает, когда выясняется, что файл манифеста недоступен и кэш можно удалить.

Следующий пример поможет вам лучше разобраться в процессе. В этом коде при каждом срабатывании события в элемент databox добавляется сообщение со значением события и значением свойства status.

Листинг 16.8. Проверка подключения

```
function initiate(){
  databox=document.getElementById('databox');

  cache=window.applicationCache;
  cache.addEventListener('checking', function(){ show(1); }, false);
  cache.addEventListener('downloading', function(){ show(2); }, false);
  cache.addEventListener('cached', function(){ show(3); }, false);
  cache.addEventListener('updateready', function(){ show(4); }, false);
  cache.addEventListener('obsolete', function(){ show(5); }, false);
}
```

продолжение ⇩

Листинг 16.8 *(продолжение)*

```
function show(value){
  databox.innerHTML+='<br>Состояние: '+cache.status;
  databox.innerHTML+=' | Событие: '+value;
}
window.addEventListener('load', initiate, false);
```

Для каждого события мы использовали анонимную функцию. Эти функции отправляют функции `show()` разные значения, идентифицирующие соответствующее событие. Значение для события и значение свойства `status` выводятся на экран каждый раз, когда браузер переходит к новому этапу генерации кэша.

САМОСТОЯТЕЛЬНО

Используйте файлы HTML и CSS из предыдущих примеров. Скопируйте код из листинга 16.8 в файл cache.js. Загрузите приложение на свой сервер и посмотрите, как при загрузке документа данные для разных этапов процесса будут отображаться на экране в соответствии со статусом кэша.

ВНИМАНИЕ

Если кэш уже был создан, очень важно выполнить несколько разных действий для очистки кэша и загрузки новой версии. Изменение файла манифеста — это один из этапов, но не единственный. Браузеры сохраняют копии файлов на несколько часов, прежде чем проверять наличие обновлений, поэтому независимо от того, сколько новых комментариев или файлов вы добавите в файл манифеста, в течение какого-то времени браузер все равно будет использовать старую версию кэша. Мы рекомендуем для тестирования примера изменить имена всех файлов. Например, если вы добавите в конце названия номер (скажем, cache2.js), то браузер посчитает ваши файлы новым приложением и создаст новый кэш. Разумеется, так делать можно только в целях тестирования.

Прогресс

Приложения, включающие в себя изображения, несколько файлов с кодом, информацию для баз данных, видео и любые другие объемные файлы,

обычно загружаются довольно долго. Для того чтобы отслеживать этот процесс, API предоставляет уже известное вам событие `progress`. Это такое же событие, как мы уже использовали раньше при знакомстве с другими API.

Событие `progress` срабатывает только в процессе загрузки файлов. В следующем примере мы используем событие `noupdate` совместно с описанными ранее событиями `cached` и `updateready`, для того чтобы сообщить пользователю, что процесс загрузки завершился.

Листинг 16.9. Прогресс загрузки

```
function initiate(){
  databox=document.getElementById('databox');
  databox.innerHTML='<progress value="0" max="100">0%</progress>';

  cache=window.applicationCache;
  cache.addEventListener('progress', progress, false);
  cache.addEventListener('cached', show, false);
  cache.addEventListener('updateready', show, false);
  cache.addEventListener('noupdate', show, false);
}
function progress(e){
  if(e.lengthComputable){
    var per=parseInt(e.loaded/e.total*100);
    var progressbar=databox.querySelector("progress");
    progressbar.value=per;
    progressbar.innerHTML=per+'%';
  }
}
function show(){
  databox.innerHTML='Готово';
}
window.addEventListener('load', initiate, false);
```

Как всегда, событие `progress` срабатывает периодически, информируя нас о состоянии процесса. В коде из листинга 16.9 при каждом срабатывании события `progress` происходит вызов функции `progress()` и информация на экране в элементе `<progress>` обновляется.

Процесс может завершаться по-разному. Возможно, приложение кэшировалось впервые, и тогда сработает событие `cached`. Если кэш уже существовал и мы загрузили обновление, то по завершении загрузки файлов сработает событие `updateready`. Третья возможность такова, что кэш уже

использовался и обновлений не было, поэтому все завершилось событием `noupdate`. Мы прослушиваем события для всех перечисленных ситуаций и во всех случаях вызываем функцию `show()`, которая выводит на экран сообщение «Готово», извещая пользователя о завершении процесса.

Подробнее о функции `progress()` рассказывается в главе 13.

САМОСТОЯТЕЛЬНО

Используйте файлы HTML и CSS из предыдущих примеров. Скопируйте код из листинга 16.8 в файл cache.js. Загрузите приложение на свой сервер и откройте в браузере главный документ. Для того чтобы увидеть работающий индикатор прогресса, в манифесте необходимо описать объемный файл (сейчас браузеры накладывают определенные ограничения на размер кэша; рекомендуем тестировать пример с файлами объемом несколько мегабайт, но не больше 5 Мбайт). Например, если вы используете видео trailer.ogg, с которым мы работали в главе 5, то файл манифеста будет выглядеть так:

```
CACHE MANIFEST
cache.html
cache.css
cache.js
trailer.ogg
# date 2011/06/27
```

ВНИМАНИЕ

Для добавления в документ нового элемента <progress> мы использовали innerHTML. Это нельзя назвать рекомендуемой практикой, однако в нашем маленьком примере так сделать можно. Чаще всего элементы добавляются в DOM с помощью метода JavaScript createElement() совместно с appendChild().

Обновление кэша

Пока что мы узнали только, как создать кэш для нашего приложения, как сообщить браузеру о том, что доступно обновление, и как контролировать

процесс обновления, когда пользователь запускает приложение. Это полезно, однако непонятно для пользователя. Кэш и его обновления загружаются сразу же после того, как пользователь открывает приложение, что может вызвать задержки и прочие неприятности при старте. Рассматриваемый API решает эту проблему, предоставляя новые методы для обновления кэша в процессе работы приложения:

- `update()`. Инициирует обновление кэша. Приказывает браузеру сначала загрузить файл манифеста, а затем, если в манифесте обнаружатся изменения (что означает модификацию подлежащих кэшированию файлов), продолжить загрузку остальных файлов;
- `swapCache()`. Переключает браузер на работу с самой свежей версией кэша после обновления. Не запускает новые сценарии и не заменяет ресурсы, а просто сообщает браузеру, что доступен новый кэш, из которого теперь можно считывать данные.

Для обновления кэша достаточно вызывать метод `update()`. События `updateready` и `noupdate` позволяют проверить результат процесса. В следующем примере мы будем использовать новый HTML-документ с двумя кнопками. Одна позволяет запросить обновление, а вторая — проверить код в кэше.

Листинг 16.10. HTML-документ для тестирования метода update()

```
<!DOCTYPE html>
<html lang="ru" manifest="mycache.manifest">
<head>
  <title>API автономной работы</title>
  <link rel="stylesheet" href="cache.css">
  <script src="cache.js"></script>
</head>
<body>
  <section id="databox">
    Автономное приложение
  </section>
  <button id="update">Обновить кэш</button>
  <button id="test">Тест</button>
</body>
</html>
```

В коде JavaScript реализованы техники, с которыми вы уже знакомы; мы всего лишь добавили две новые функции для поддержки кнопок.

Листинг 16.11. Обновление кэша и проверка текущей версии

```
function initiate(){
  databox=document.getElementById('databox');
  var update=document.getElementById('update');
  update.addEventListener('click', updatecache, false);
  var test=document.getElementById('test');
  test.addEventListener('click', testcache, false);

  cache=window.applicationCache;
  cache.addEventListener('updateready', function(){ show(1); },
        false);
  cache.addEventListener('noupdate', function(){ show(2); },
        false);
}
function updatecache(){
  cache.update();
}
function testcache(){
  databox.innerHTML+='<br>поменяйте это сообщение';
}
function show(value){
  switch(value){
    case 1:
      databox.innerHTML+='<br>Обновление выполнено;
      break;
    case 2:
      databox.innerHTML+='<br>Обновления отсутствуют';
      break;
  }
}
window.addEventListener('load', initiate, false);
```

В функции `initiate()` мы добавляем для обеих кнопок прослушиватель события `click`. При щелчке на кнопке `update` происходит вызов функции `updatecache()`, в которой выполняется метод `update()`. Если пользователь щелкает на кнопке `test`, то вызывается функция `testcache()`, которая выводит в элементе `databox` новый текст. Вы можете затем изменить этот текст, чтобы создать новую версию кода и проверить, обновился он или нет.

САМОСТОЯТЕЛЬНО

Создайте новый HTML-документ с кодом из листинга 16.10. Файл манифеста и CSS-файл те же, которые вы использовали в предыдущих примерах (если только вы не меняли имена файлов — в этом случае нужно обновить список файлов в манифесте). Скопируйте код из листинга 16.11 в файл с именем cache.js и загрузите все эти файлы на свой сервер. Откройте главный документ в браузере и протестируйте приложение.

После завершения загрузки HTML-документа вы видите в окне привычный уже элемент `databox` и две кнопки под ним. Как уже говорилось, событие `click` для кнопки Обновить кэш вызывает функцию `updatecache()`. Когда пользователь щелкает на этой кнопке, внутри функции выполняется метод `update()` и запускается процесс обновления кэша. Браузер загружает файл манифеста и сравнивает его с аналогичным файлом в кэше. Если файл изменился, то происходит повторная загрузка всех перечисленных в манифесте файлов. По завершении процесса срабатывает событие `updateready`. Оно вызывает функцию `show()` со значением 1, которое соответствует сообщению «Обновление выполнено». Но если файл манифеста не менялся, то обновления не происходит и срабатывает событие `noupdate`. Это событие вызывает функцию `show()` со значением 2, и внутри элемента `databox` выводится сообщение «Обновления отсутствуют».

Для того чтобы проверить, как этот код работает, отредактируйте файл манифеста или добавьте в него комментарии. После изменения манифеста запросите обновление кэша, щелкнув на соответствующей кнопке. В поле `databox` появится сообщение «Обновление выполнено». Кроме того, попробуйте поиграть с текстом в функции `testcache()`, чтобы определить момент, когда обновление выполняется.

ВНИМАНИЕ

На этот раз вам не нужно стирать кэш в браузере, чтобы загрузить новую версию. Метод update() заставляет браузер загружать файл манифеста и остальные файлы в случае, когда обнаруживается обновление. Однако новый кэш недоступен до тех пор, пока пользователь не обновит страницу.

Краткий справочник. API Offline (Автономная работа)

API автономной работы представляет собой набор техник, в том числе специальный файл под названием «манифест» и несколько методов, событий и свойств. Все они предназначены для того, чтобы работающие в браузере приложения могли быть кэшированы на компьютере пользователя. Основное назначение данного API — обеспечивать постоянный доступ к приложениям, в том числе возможность работать с ними даже в условиях отсутствия подключения к Интернету.

Файл манифеста

Файл манифеста — это текстовый файл с расширением .manifest, содержащий список файлов, которые должны быть помещены в кэш. Он всегда начинается со строки `CACHE MANIFEST`, а его содержимое можно разбить на следующие категории:

- `CACHE`. В эту категорию помещаются кэшируемые файлы;
- `NETWORK`. К этой категории относятся файлы, обращение к которым возможно только через сетевое подключение;
- `FALLBACK`. В этой категории перечисляются альтернативные ресурсы для файлов с сервера, недоступных в данный момент.

Свойства

У объекта Navigator появилось новое свойство, позволяющее проверять состояние подключения:

- `onLine`. Возвращает булево значение, указывающее на состояние подключения. Равно `false`, если браузер работает автономно, и `true` — в противном случае.

В данном API предусмотрено свойство `status`, предназначенное для проверки статуса кэша приложения. Это свойство принадлежит объекту ApplicationCache и может принимать следующие значения:

- `UNCACHED` (значение 0). Указывает, что кэш для приложения еще не создавался;

- `IDLE` (значение 1). Указывает, что кэш для приложения актуален и используется;
- `CHECKING` (значение 2). Указывает, что браузер проверяет обновления;
- `DOWNLOADING` (значение 3). Указывает, что браузер находится в процессе загрузки файлов кэша;
- `UPDATEREADY` (значение 4). Указывает, что кэш для приложения доступен и используется, но неактуален; обновление готово для загрузки и замены старых файлов;
- `OBSOLETE` (значение 5). Указывает, что текущий кэш больше не используется.

События

У объекта окна есть два события, позволяющие проверять статус подключения:

- `online`. Срабатывает, когда значение свойства `onLine` меняется на `true`;
- `offline`. Срабатывает, когда значение свойства `onLine` меняется на `false`.

В API также предусмотрено несколько принадлежащих объекту ApplicationCache событий, которые информируют о различных условиях, связанных с кэшем:

- `checking`. Срабатывает, когда браузер проверяет обновления;
- `noupdate`. Срабатывает, если обновления не обнаружены;
- `downloading`. Срабатывает, если браузер обнаружил обновление и начал загрузку файлов;
- `cached`. Срабатывает, когда кэш готов;
- `updateready`. Срабатывает, когда процесс загрузки обновления завершается;
- `obsolete`. Срабатывает, когда обнаруживается, что файл манифеста недоступен, и кэш удаляется;
- `progress`. Срабатывает во время процесса загрузки файлов кэша;
- `error`. Срабатывает, если во время создания или обновления кэша происходит ошибка.

Методы

Два метода в рассматриваемом API позволяют запросить обновление кэша:

- `update()`. Инициирует обновление кэша. Заставляет браузер загрузить файл манифеста и в случае, если обнаруживаются изменения, обновить и остальные файлы в кэше;
- `swapCache()`. Переключает браузер на самую свежую версию кэша после обновления. Не запускает сценарии и не заменяет ресурсы, просто сообщает браузеру, что в дальнейшем информацию можно считывать из нового кэша.

Заключение

Работаем для реального мира

Эта книга посвящена HTML5. Мы создавали ее как руководство для разработчиков, дизайнеров и программистов, желающих сооружать революционные веб-сайты и приложения. Она предназначена для гения, находящегося внутри каждого из нас. Для вдохновителя. Мы находимся сейчас в процессе перехода, нам повезло наблюдать удивительный период, когда старые технологии сливаются с новыми, а рынки едва поспевают за изменениями. Каждую минуту миллионы и миллионы копий новых браузеров загружаются из Сети, но в то же время миллионы и миллионы пользователей даже не догадываются о существовании новинок. На рынке все еще полно старых компьютеров под управлением Windows 98, на которых работает Internet Explorer 6 или что-нибудь похуже.

Создание продуктов для Сети всегда было непростой задачей, и со временем все только усложняется. Несмотря на длительные упорные попытки внедрить стандарты для Сети, даже новые браузеры все еще не могут похвастаться единообразной поддержкой новых спецификаций. При этом по всему миру по сей день работает огромное количество старых браузеров, не поддерживающих никаких стандартов и только отравляющих нам жизнь.

Таким образом, настало время посмотреть, что мы в состоянии сделать, чтобы познакомить человечество с HTML5. Как нам внедрять новинки и творить, быть вдохновителями в мире, для которого развитие

технологий значит очень мало? Как успешно работать с новыми технологиями и делать их доступными для всех?

Альтернативы

Когда речь заходит об альтернативах, нам в первую очередь необходимо определиться с собственной позицией. Мы можем быть грубыми, вежливыми, находчивыми или просто прилежными работниками. Грубый разработчик скажет: «Эй, это и должно работать только в новых браузерах. Новые браузеры раздают бесплатно, не ленитесь, просто возьмите и загрузите себе один из них». Вежливый: «Мы разрабатывали это с учетом преимуществ новых технологий; если вы желаете насладиться всеми возможностями и потенциалом нашего творения, то обновите свой браузер. А пока что я могу предложить вам старую версию». Находчивый: «Мы делаем передовые технологии доступными для всех. Вам ничего не придется предпринимать самостоятельно, мы обо всем позаботились за вас». И наконец, прилежный работник скажет: «Вот версия нашего веб-сайта, адаптированная под ваш браузер, а вот другая, для новых браузеров, предлагающая больше возможностей, и, кстати, у нас еще есть экспериментальная передовая версия нашего супер-навороченного приложения».

Если же вам требуется наиболее полезный и практичный подход, то для случая, когда браузер пользователя не поддерживает HTML5, вы можете выбрать один из следующих вариантов:

- **информируйте.** Попросите пользователя обновить браузер, если в вашем приложении есть функции, без которых не обойтись, но они не работают в устаревших браузерах;
- **адаптируйте.** Выбирайте для документа другие стили и коды, ориентируясь на возможности, доступные в браузере пользователя;
- **перенаправляйте.** Перенаправляйте пользователей на совершенно другие документы, разработанные специально для старых браузеров;
- **эмулируйте.** Используйте библиотеки, позволяющие предоставлять возможности HTML5 пользователям старых браузеров.

Modernizr

Независимо от того, какой вариант вы выберете, в первую очередь необходимо определить, доступны ли в браузере пользователя возможности HTML5, реализованные в вашем приложении. Новые возможности не зависят друг от друга, и проверить их наличие не составляет труда, однако

техники распознавания настолько же разнообразны, как и сама функциональности HTML5. Разработчикам приходится учитывать разные версии множества браузеров и зависеть от кодов, которые зачастую оказываются не слишком надежными.

Для решения этой проблемы была разработана небольшая библиотека под названием Modernizr. Эта библиотека создает объект с именем Modernizr, включающий в себя свойства для каждой из возможностей HTML5. Эти свойства возвращают булево значение `true` или `false` в зависимости от того, доступна ли соответствующая возможность.

Данная библиотека поставляется с открытым кодом, она написана на языке программирования JavaScript, и вы можете бесплатно загрузить ее с веб-сайта http://www.modernizr.com. Всего лишь скопируйте на свой компьютер файл JavaScript и добавьте его в заголовок документа, как в следующем примере.

Листинг 3.1. Добавление библиотеки Modernizr в документ

```
<!DOCTYPE html>
<html lang="ru">
<head>
  <title>Modernizr</title>
  <script src="modernizr.min.js"></script>
  <script src="modernizr.js"></script>
</head>
<body>
  <section id="databox">
    содержимое
  </section>
</body>
</html>
```

Файл под названием modernizr.min.js — это копия файла с библиотекой Modernizr, который мы загрузили с сайта создателей Modernizr. Второй файл, указанный в «голове» HTML-документа в листинге 3.1, — это наш собственный JavaScript-код, проверяющий значения свойств, которые предоставляет библиотека.

Листинг 3.2. Проверка доступности CSS-стилей для создания тени поля

```
function initiate(){
  var databox=document.getElementById('databox');
```

продолжение ⇩

Листинг 3.2 *(продолжение)*

```
  if(Modernizr.boxshadow){
    databox.innerHTML='Тени для полей доступны;
  }else{
    databox.innerHTML='Тени для полей недоступны;
  }
}
window.addEventListener('load', initiate, false);
```

Как видно из кода листинга 3.2, определить наличие любой возможности HTML5 можно с помощью всего лишь условного оператора `if` и соответствующего свойства объекта Modernizr. Для каждой возможности предусмотрено отдельное свойство.

> **ВНИМАНИЕ**
>
> Это всего лишь краткое знакомство с потрясающе полезной библиотекой. Используя Modernizr, вы также можете выбирать наборы CSS-стилей из файлов CSS, не прибегая к помощи JavaScript. Modernizr предоставляет специальные классы, которые можно добавлять в таблицы стилей для выбора нужных свойств CSS в соответствии с перечнем доступных возможностей. Чтобы подробнее узнать об этом, зайдите на веб-сайт создателей библиотеки *http://www.modernizr.com*.

Библиотеки

После того как вы выяснили, какие именно возможности в браузере пользователя доступны, можете либо продолжать работать только с тем, что удалось обнаружить, либо порекомендовать пользователю обновить программное обеспечение. Однако представим на минутку, что вы упрямый разработчик или сумасшедший программист, который (так же, как ваши пользователи и клиенты) не задумывается о принадлежности браузера, его версии или бета-версии, нереализованных функциях и вообще ни о чем на свете. Вы хотите использовать только новейшие технологии, и неважно, что для этого потребуется!

Что ж, в таком случае на помощь приходят независимые библиотеки. Десятки программистов в мире еще более непрошибаемо упрямы, чем вы и ваши клиенты, и они разрабатывают и постоянно совершенствуют библиотеки, эмулирующие возможности HTML5 в старых браузерах,

в частности, создают API-интерфейсы JavaScript. Благодаря их усилиям новые HTML-элементы, CSS3-селекторы и стили и даже такие сложные API, как Canvas (Холст) и Web Storage (Веб-хранилище), уже доступны для использования в любом браузере, имеющемся на рынке.

Актуальный список библиотек вы найдете на сайте http://www.github.com/Modernizr/Modernizr/wiki/HTML5-Cross-browser-Polyfills.

ВНИМАНИЕ

Для получения дополнительной информации зайдите на наш веб-сайт и изучите ссылки для заключения.

Google Chrome Frame

Последняя надежда — Google Chrome Frame. Лично я считаю, что вначале это было хорошей идеей, но сейчас гораздо эффективнее рекомендовать пользователям обновлять браузеры, чем просить их загружать встраиваемые модули подобные Google Chrome Frame.

Модуль Google Chrome Frame разрабатывался специально для старых версий браузера Internet Explorer. Целью его создания было привнесение всей мощи и возможностей Google Chrome в браузеры, не готовые к внедрению новых технологий, но которые все так же часто встречаются на компьютерах пользователей и занимают существенную долю рынка.

Как я уже сказал, это было неплохой идеей. Вы добавляли в свои документы всего один простой тег HTML, и на экране пользователя появлялось сообщение с просьбой загрузить Google Chrome Frame, прежде чем начинать работу с вашим веб-сайтом или приложением. После завершения этого простого шага все возможности, поддерживаемые Google Chrome, автоматически становились доступными в браузере пользователя. Однако пользователям все равно приходилось загружать из Сети какое-то программное обеспечение. Я не вижу разницы между загрузкой встраиваемого модуля и новой версии браузера, учитывая, что версия Internet Explorer с поддержкой HTML5 также распространяется бесплатно. Сегодня, когда так много браузеров с готовностью обрабатывают код HTML5, гораздо лучше подсказывать пользователям дорогу к новому программному обеспечению, чем заставлять их пользоваться странными и непонятными встраиваемыми модулями.

Для того чтобы больше узнать об использовании Google Chrome Frame, посетите веб-сайт http://code.google.com/chrome/chromeframe/.

Работаем для облака

В новом мире мобильных устройств и облачных вычислений становится не так уж важно, насколько свежая версия браузера установлена у пользователя, — нам все равно приходится беспокоиться о массе других вещей. Считается, что инновацией, с которой началось все это безумие, стало устройство iPhone. С момента его появления Сеть кардинально изменилась в нескольких аспектах. Вслед за iPhone появился iPad, и бесчисленное множество имитаций этих устройств заполонило новый рынок. Благодаря этому радикальному изменению в электронном мире мобильный доступ к Интернету превратился в повсеместную практику. Внезапно владельцы новых устройств стали важной целевой аудиторией веб-сайтов и веб-приложений, а разнообразие платформ, экранов и интерфейсов заставило разработчиков учиться адаптировать свои продукты к каждому возможному случаю.

Сегодня независимо от того, какой тип технологий мы применяем, наши веб-сайты и приложения обязаны адаптироваться ко всем существующим платформам, сохраняя при этом единообразие и обеспечивая каждому возможность пользоваться нашими продуктами. К счастью, в HTML эти ситуации всегда принимали во внимание, и элемент `<link>` предоставляет атрибут `media`, позволяющий выбирать внешние ресурсы в соответствии с предопределенными параметрами.

Листинг 3.3. Разные CSS-файлы для разных устройств

```
<!DOCTYPE html>
<html lang="ru">
<head>
  <title>Главный документ</title>
  <link rel="stylesheet" href="webstyles.css" media="all and
   (min-width: 769px)">
  <link rel="stylesheet" href="tablet.css" media="all and
   (min-width: 321px) and (max-width: 768px)">
  <link rel="stylesheet" href="phone.css" media="all and
   (min-width: 0px) and (max-width: 320px)">
</head>
<body>
…
</body>
</html>
```

Выбор CSS-стилей — несложный способ справиться с задачей. В соответствии с используемым устройством и размером его экрана мы загружаем разные файлы CSS и применяем соответствующие стили. Элементы HTML можно растрировать, адаптируя целые документы к конкретным размерам и обстоятельствам.

В листинге 3.3 мы добавили три разных CSS-файла, предназначенных для разных ситуаций. Для распознавания ситуации используются значения атрибута `media`, который присутствует в каждом теге `<link>`. Проверяя свойства `min-width` и `max-width`, мы выбираем тот CSS-файл, который подойдет к разрешению конкретного экрана, используемого для просмотра документа. Если размер экрана по горизонтали находится в диапазоне от 0 до 320 пикселов, то загружается файл phone.css. Для разрешения от 321 до 768 пикселов предназначен файл tablet.css. И наконец, для визуализации документа на экранах с шириной более 768 пикселов используется файл webstyles.css.

В этом примере мы учитываем три возможных сценария: документ отображается на экране небольшого смартфона, на планшете и на полноразмерном компьютере. Экраны этих устройств обычно характеризуются перечисленными размерами.

Разумеется, адаптация документов подразумевает изменение не только стилей CSS. Интерфейсы, предоставляемые этими устройствами, несколько отличаются от интерфейсов настольного компьютера из-за отсутствия таких физических компонентов, как клавиатура и мышь. Привычные события, например `click` и `mouseover`, в некоторых случаях заменяются событиями прикосновений. К тому же на мобильных устройствах чаще всего доступна еще одна важная новая функция — изменение ориентации экрана. Вследствие этого меняется размер пространства для отображения документа. Все эти отличия новых устройств от старых компьютерных интерфейсов делают почти невозможной адаптацию дизайна и функциональности путем простого добавления или модификации правил CSS. Приходится адаптировать коды с помощью JavaScript или даже распознавать ситуацию и перенаправлять пользователя на другую версию веб-сайта, предназначенную специально для доступа к приложению с конкретного устройства.

ВНИМАНИЕ

Это обсуждение лежит за пределами списка тем, рассматриваемых в данной книге. Более подробную информацию вы сможете получить, посетив наш веб-сайт.

Заключительные рекомендации

Кто-то из разработчиков всегда будет говорить: «Если вы используете технологии, недоступные для 5 % браузеров на рынке, то вы теряете 5 % клиентов». Мой ответ таков: «Если ваша задача — удовлетворить клиентов, то адаптируйте, перенаправляйте и эмулируйте, но, если вы работаете на себя, информируйте».

Вы всегда должны находить пути к успеху. Если вы работаете на других, то для достижения успеха вам необходимо предоставлять полностью рабочее решение, продукт, который клиенты вашего клиента смогут использовать независимо от того, какие компьютер, браузер или систему они выберут для себя. Однако если вы работаете на себя, то для того, чтобы стать успешными, вам необходимо создавать лучшее из лучшего, вы должны производить перемены и опережать конкурентов независимо от того, что за программы те самые 5 % пользователей установили на свои компьютеры. Вы должны работать для 20 % пользователей, которые уже загрузили новейшую версию Firefox, 15 % пользователей, на компьютерах которых работает Google Chrome, и 10 % пользователей, который заходят в Интернет через Safari на мобильных устройствах. Миллионы людей уже готовы стать вашими клиентами. Пусть тот разработчик спрашивает, почему вы готовы пойти на потерю 5 % рынка, — я спрошу, почему он готов потерять свой шанс на успех.

Вы никогда не завоюете 100 % рынка, и с этим фактом необходимо смириться. Вы не создаете веб-сайты на китайском или португальском языках. Вы работаете не для 100 % рынка, вы уже вкладываетесь в то, чтобы удовлетворить лишь небольшую его часть. Так зачем же продолжать ограничивать себя? Разрабатывайте для той части рынка, которая непрерывно увеличивается и помогает вам выпустить на свободу внутреннего гения. Так же, как вы не беспокоитесь о рынках, говорящих на иностранных языках, не забивайте себе голову мыслями о доле рынка, все так же использующей устаревшие технологии. Информируйте людей. Рассказывайте им о том, что они теряют. Пользуйтесь преимуществами новейших технологий и будьте вдохновителями. Разрабатывайте для будущего, и вас ждет успех.

Хуан Диего Гоше

HTML5. Для профессионалов

Перевела с английского Е. Шикарева

Заведующий редакцией	*А. Кривцов*
Руководитель проекта	*А. Юрченко*
Ведущий редактор	*Ю. Сергиенко*
Литературный редактор	*Н. Рощина*
Художественный редактор	*К. Радзевич*
Корректор	*И. Тимофеева*
Верстка	*Е. Волошина*

ООО «Питер Пресс», 192102, Санкт-Петербург, ул. Андреевская (д. Волкова), д. 3, литер А, пом. 7Н.
Налоговая льгота — общероссийский классификатор продукции ОК 005-93, том 2;
95 3005 — литература учебная.
Подписано в печать 05.10.12. Формат 70x100/16. Усл. п. л. 39,990. Тираж 2000. Заказ
Отпечатано с готовых диапозитивов в ИПК ООО «Ленинградское издательство».
194044, Санкт-Петербург, ул. Менделеевская, 9.

ВАМ НРАВЯТСЯ НАШИ КНИГИ? ЗАРАБАТЫВАЙТЕ ВМЕСТЕ С НАМИ!

У Вас есть свой сайт?
Вы ведете блог?
Регулярно общаетесь на форумах? Интересуетесь литературой, любите рекомендовать хорошие книги и хотели бы стать нашим партнером?
ЭТО ВПОЛНЕ РЕАЛЬНО!

СТАНЬТЕ УЧАСТНИКОМ ПАРТНЕРСКОЙ ПРОГРАММЫ ИЗДАТЕЛЬСТВА «ПИТЕР»!

- *Зарегистрируйтесь на нашем сайте в качестве партнера по адресу* ***www.piter.com/ePartners***
- *Получите свой персональный уникальный номер партнера*
- *Выбирайте книги на сайте* ***www.piter.com,*** *размещайте информацию о них на своем сайте, в блоге или на форуме и добавляйте в текст ссылки на эти книги (на сайт* ***www.piter.com****)*

ВНИМАНИЕ! *В каждую ссылку необходимо добавить свой персональный уникальный номер партнера.*

С этого момента получайте 10% от стоимости каждой покупки, которую совершит клиент, придя в интернет-магазин «Питер» по ссылке с Вашим партнерским номером. А если покупатель приобрел не только эту книгу, но и другие издания, Вы получаете дополнительно по 5% от стоимости каждой книги.

Деньги с виртуального счета Вы можете потратить на покупку книг в интернет-магазине издательства «Питер», а также, если сумма будет больше 500 рублей, перевести их на кошелек в системе Яндекс.Деньги или Web.Money.

Пример партнерской ссылки:
http://www.piter.com/book.phtml?978538800282 – обычная ссылка
http://www.piter.com/book.phtml?978538800282**&refer=0000** – партнерская ссылка, где 0000 – это ваш уникальный партнерский номер

Подробно о Партнерской программе ИД «Питер» читайте на сайте WWW.PITER.COM