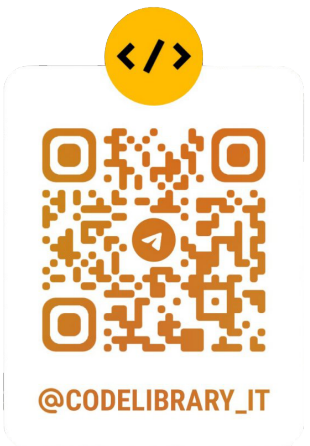
Владимир Дронов

HTML 5, CSS 3 и Web 2.0

Разработка современных Web-сайтов



УДК 681.3.068
ББК 32.973.26-018.1
Д75

Дронов В. А.

Д75 HTML 5, CSS 3 и Web 2.0. Разработка современных Web-сайтов. — СПб.: БХВ-Петербург, 2011. — 416 с.: ил. — (Профессиональное программирование)

ISBN 978-5-9775-0596-3

Практическое руководство по созданию современных Web-сайтов, соответствующих концепции Web 2.0. Описаны языки HTML 5 и CSS 3, применяемые, соответственно, для создания содержимого и представления Web-страниц. Даны принципы Web-программирования на языке JavaScript с использованием библиотеки Ext Core. Рассказано о создании интерактивных Web-страниц, приведены примеры интерактивных элементов, позволяющие сделать Web-страницы удобнее для посетителя. Раскрыты вопросы реализации подгружаемого и генерируемого содержимого, семантической разметки, применения баз данных для формирования Web-страниц. Показаны способы расширения функциональности Web-сайтов с использованием Web-форм, элементов управления, свободно позиционируемых элементов и программного рисования средствами HTML 5.

Для Web-дизайнеров

УДК 681.3.068
ББК 32.973.26-018.1

Группа подготовки издания:

Главный редактор	*Екатерина Кондукова*
Зам. главного редактора	*Евгений Рыбаков*
Зав. редакцией	*Григорий Добин*
Редактор	*Леонид Кочин*
Компьютерная верстка	*Ольги Сергиенко*
Корректор	*Зинаида Дмитриева*
Дизайн серии	*Инны Тачиной*
Оформление обложки	*Елены Беляевой*
Зав. производством	*Николай Тверских*

Лицензия ИД № 02429 от 24.07.00. Подписано в печать 31.08.10.
Формат $70\times100^{1}/_{16}$. Печать офсетная. Усл. печ. л. 33,54.
Тираж 1500 экз. Заказ №
"БХВ-Петербург", 190005, Санкт-Петербург, Измайловский пр., 29.

Санитарно-эпидемиологическое заключение на продукцию
№ 77.99.60.953.Д.005770.05.09 от 26.05.2009 г. выдано Федеральной службой
по надзору в сфере защиты прав потребителей и благополучия человека.

Отпечатано с готовых диапозитивов
в ГУП "Типография "Наука"
199034, Санкт-Петербург, 9 линия, 12

ISBN 978-5-9775-0596-3

Оглавление

Введение

Мир Web-дизайна в очередной раз лихорадит. Шутка ли — новые интернет-технологии на подходе!

Что грядет нового в Web-дизайне

Сейчас, наверно, даже школьники знают, что содержимое Web-страниц создается с помощью языка HTML, а внешний вид их элементов определяется стилями, которые описываются на языке CSS. Существует также возможность написать небольшие программы на языке JavaScript, которые встраиваются в саму Web-страницу и изменяют ее содержимое в ответ на действия посетителя, — Web-сценарии.

Все эти языки и технологии были созданы более десяти лет тому назад, и за последнее время в них мало что изменилось (а в языке HTML не изменилось вообще ничего). Так что за последние лет десять в Web-дизайне не было никаких революций — только небольшие эволюционные изменения.

Но уже готовятся новые стандарты, которые описывают очередные версии этих языков: HTML 5 и CSS 3. Они обещают принести в Web-дизайн много нового.

- Упрощенную вставку на Web-страницу аудио- и видеоматериалов.
- Возможности рисования на Web-страницах.
- Многоколоночную верстку текста.
- Поддержку работы в оффлайновом режиме (при отключении от Интернета).
- Дополнительную поддержку мобильных устройств.
- Поддержку специальных Web-обозревателей для лиц с ограниченными физическими возможностями.
- И, как водится, многое-многое другое.

Звучит заманчиво, но... Сейчас эти стандарты существуют только в виде "черновых" редакций, и когда выйдут "чистовые", окончательные, никто не знает.

Так отчего же разгорелся весь сыр-бор?

Разработчики Web-обозревателей, не дожидаясь, пока их коллеги, "сочиняющие" интернет-стандарты, завершат работу над HTML 5 и CSS 3, уже внедряют под-

держку некоторых их возможностей в свои творения. Так, Mozilla Firefox, Opera, Google Chrome и Apple Safari уже поддерживают интернет-мультимедиа в стиле HTML 5, программное рисование на Web-страницах и работу в оффлайновом режиме. Пусть и не в полной мере, но все-таки поддерживают!

"Не в теме" пока еще Microsoft Windows Internet Explorer. Однако Microsoft уже довольно давно объявила о разработке новой версии своего Web-обозревателя под номером 9. И в плане поддержки всех "горячих" интернет-новинок грозит заткнуть за пояс всех конкурентов. Что ж, у автора есть причины верить: даже предварительная версия Internet Explorer 9, совсем-совсем "сырая", на момент написания этой книги выглядит очень даже неплохо.

Но даже возможности действующих на сегодняшний день версий HTML и CSS на деле используются не в полной мере:

- подгружаемое содержимое — загрузка части содержимого вместо целой Web-страницы — практически не применяется;
- генерируемое содержимое — программное создание части содержимого Web-страницы после ее загрузки — применяется мало;
- разделение содержимого, представления и поведения Web-страниц также почти не реализуется.

А ведь все это способно значительно упростить труд Web-дизайнера и заметно улучшить вид и удобство использования Web-сайтов. Вот такие они Web-дизайнеры, не ведают своей выгоды...

О чем эта книга

В книге описываются:

- Язык HTML и принципы создания содержимого Web-страниц.
- Язык CSS и принципы создания представления Web-страниц.
- Возможности HTML 5 и CSS 3, уже поддерживаемые современными Web-обозревателями.
- Основы Web-программирования, язык JavaScript и принципы создания поведения Web-страниц.
- Библиотека Ext Core — инструмент, призванный упростить труд Web-программиста.
- Создание интерактивных Web-страниц с конкретными примерами.
- Реализация подгружаемого и генерируемого содержимого и семантической разметки данных средствами JavaScript.
- Использование специальных средств — Web-форм, элементов управления и свободных контейнеров — для обеспечения дополнительной функциональности Web-сайтов.
- Реализация программного рисования на Web-страницах средствами HTML 5.

В результате читатель создаст полнофункциональный Web-сайт — справочник по HTML и CSS. Конечно, это только пример — данные технологии можно применить и для разработки любого другого Web-сайта.

Какие программы используются в этой книге

Вопрос далеко не праздный, учитывая то, что за программы сегодня приходится платить... Так что же за программы использовал автор?

Только бесплатные!

- Блокнот — простейший текстовый редактор, стандартно поставляемый в составе Windows.
- Firefox версий 3.6.* — Web-обозреватель. Все примеры тестировались на нем.
- Opera 10.52 и Apple Safari 4.* — Web-обозреватели. На них автор тестировал некоторые примеры.

Другие программы автор в работе практически не применял.

Типографские соглашения

В этой книге часто приводятся форматы написания различных тегов HTML, атрибутов стилей CSS и выражений JavaScript. Нам необходимо запомнить типографские соглашения, используемые для их написания.

> ***Внимание!***
>
> Все эти типографские соглашения автор применяет только в форматах написания тегов HTML, атрибутов стилей CSS и выражений JavaScript. В коде примеров они не имеют смысла.

В угловые скобки (<>) заключены названия значений атрибутов, параметров или фрагментов кода, которые, в свою очередь, набраны курсивом. В код реального Web-сценария, разумеется, нужно подставить реальное значение, конкретный параметр или код.

Пример:

```
<MAP NAME="<имя карты>">
</MAP>
```

Здесь вместо подстроки *<имя карты>* нужно подставить конкретное имя карты.

Пример:

```
<FORM>
  <теги, формирующие элементы управления>
</FORM>
```

Здесь вместо подстроки *<теги, формирующие элементы управления>* следует подставить реальные HTML-теги, формирующие элементы управления.

В квадратные скобки ([]) заключены необязательные фрагменты кода:

```
<LEGEND [ACCESSKEY="<быстрая клавиша>"]><текст заголовка></LEGEND>
```

Здесь атрибут тега ACCESSKEY может присутствовать, а может и отсутствовать.

Символом вертикальной черты (|) разделены фрагменты кода, из которых в данном месте должен присутствовать только один:

```
SHAPE="rect|circle|poly"
```

Здесь в качестве значения атрибута тега SHAPE должна присутствовать только одна из доступных строк: rect, circle или poly.

Слишком длинные, не помещающиеся на одной строке фрагменты кода автор разрывает на несколько строк и в местах разрывов ставит знаки ↳.

Пример:

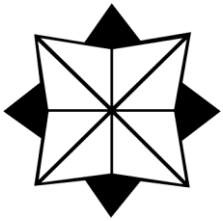
```
var s = "<LI><CODE><A HREF=\"" + aDataBase[i].url + "\">" +
↳aDataBase[i].name + "</A></CODE></LI>";
```

Приведенный код разбит на две строки, но должен быть набран в одну. Знаки ↳ при этом нужно удалить.

Благодарности

Автор приносит благодарности своим родителям, знакомым и коллегам по работе.

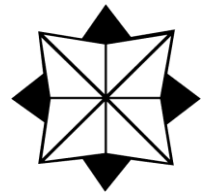
- Губиной Наталье Анатольевне, начальнику отдела АСУ Волжского гуманитарного института (г. Волжский Волгоградской обл.), где работает автор, — за понимание и поддержку.
- Всем работникам отдела АСУ Волжского гуманитарного института — за понимание и поддержку.
- Родителям — за терпение, понимание и поддержку.
- Архангельскому Дмитрию Борисовичу — за дружеское участие.
- Шапошникову Игорю Владимировичу — за содействие.
- Рыбакову Евгению Евгеньевичу, заместителю главного редактора издательства "БХВ-Петербург", — за неоднократные побуждения к работе, без которых автор давно бы обленился.
- Издательству " БХВ-Петербург" — за издание моих книг.
- Разработчикам Web-обозревателей Firefox, Opera, Chrome и Safari и библиотеки Ext Core, если они меня слышат, — за замечательные программные продукты.
- Всем своим читателям и почитателям — за прекрасные отзывы о моих книгах.
- Всем, кого я забыл здесь перечислить, — за все хорошее.



ЧАСТЬ I

Содержимое Web-страниц. Язык HTML 5

ГЛАВА 1



Введение в современный Web-дизайн. Web 2.0. Создание Web-страниц

Всемирная паутина, WWW, Web-дизайн, Web-сайт, Web-страница — все знают, что это такое. Но что такое современная Всемирная паутина, современный Web-дизайн и современная Web-страница?

Именно с ответов на все эти вопросы начнется данная книга. Далее мы немного поговорим о принципах функционирования Интернета, Web-страницах и Web-сайтах, создадим нашу первую Web-страницу, начнем изучать язык HTML 5 и подберем программы, которые будем использовать в работе. Так сказать, с места в карьер...

Современный Web-дизайн. Концепция Web 2.0

Раньше доступ в Интернет можно было получить только с компьютеров. Потом в Интернет стали выходить с мобильных телефонов. Сейчас к Сети подключились мультимедийные плееры, устройства чтения электронных книг и телевизоры. А завтра — кто знает; может быть, мы будем выходить на Web-сайты с утюга или пылесоса...

"Я буду везде", — заявляет Интернет. — "Я стану вездесущим. Все готовьтесь к моему приходу!"

Что требуется от современного Web-сайта

Будем готовиться... Но что нам, как будущим Web-дизайнерам, для этого следует сделать? Соблюсти три несложных правила.

1. Строго соблюдать все интернет-стандарты.
2. Тщательно продумать наполнение Web-страниц.
3. Позаботиться о доступности Web-страниц.

Рассмотрим их подробнее.

Интернет грозится прийти на самые разные устройства, которые могут быть основаны на разных аппаратных и программных платформах, зачастую сильно отли-

чающихся друг от друга. Так, персональные компьютеры построены на аппаратной платформе Intel и программной платформе Microsoft Windows (по крайней мере, большинство). Мобильный телефон автора основан на аппаратно-программной платформе Samsung. А на чем будет работать интернет-пылесос, сейчас не может сказать никто.

Одно объединяет все это аппаратно-программное многообразие — соответствие интернет-стандартам. Иначе устройства в лучшем случае будут отображать Web-страницы неправильно, в худшем — вообще не будут работать.

Из этого следует первое правило из перечисленных ранее — Web-дизайнеры при создании Web-страниц обязаны строго придерживаться современных интернет-стандартов, чтобы их творения одинаково (ну, или почти одинаково) отображались на всех устройствах.

Первое правило также требует отказа от устаревших и закрытых, фирменных интернет-технологий. С устаревшими технологиями все понятно: старье — не помощник новому. Закрытые же технологии неудобны тем, что зачастую контролируются единственной фирмой, которая единолично "заказывает музыку" и далеко не всегда прислушивается к мнению интернет-сообщества. К таким технологиям относятся, в частности, Adobe Flash и Microsoft ActiveX.

Открытыми интернет-стандартами, в том числе и Web-стандартами, занимается организация World Wide Web Consortium (Консорциум Всемирной паутины), или сокращенно W3C. Она разрабатывает стандарты, согласует их с требованиями участников рынка и публикует на своем Web-сайте **http://www.w3.org**. Все опубликованные там стандарты обязательны к применению.

Интернет когда-то начинался как сеть ученых, которым было нужно обмениваться результатами исследований. А что представляли собой эти результаты? В основном, текст, возможно, с иллюстрациями. Ученые — публика в этом смысле невзыскательная, им вполне хватало скромных возможностей тогдашнего WWW.

Теперь же абсолютное большинство пользователей Интернета — обычные обыватели. Им мало простого текста с парой картинок, им подавай хорошо оформленный текст, музыку и видео. Они требовательнее первых обитателей Сети.

Отсюда вытекает второе правило — Web-дизайнеры должны заботиться о полноте и удобстве наполнения Web-страниц.

- Структура Web-страниц должна быть хорошо продумана, чтобы посетитель сразу смог найти на них все, что ему нужно.
- Web-страницы должны легко читаться и не "резать" глаза.
- К важным материалам желательно привлечь внимание посетителя, а маловажные скрыть. В этом могут помочь динамические элементы: раскрывающиеся при щелчке мышью абзацы, гиперссылки, выделяющиеся при наведении курсора мыши, и пр.
- Если Web-сайт посвящен музыке или видео, все это должно быть доступно для воспроизведения прямо на его Web-страницах, без загрузки.

- Одним словом — все для удобства посетителя! (Пожалуй, это правило следовало бы поставить в начале списка...)

Интернет грозится прийти на самые разные устройства с различными характеристиками: быстродействием процессора, объемом памяти, разрешением экрана, скоростью доступа к Сети. Но все они должны обеспечивать единообразный вывод Web-страниц. Как этого достигнуть?

Вот и третье правило — Web-дизайнеры должны заботиться о доступности Web-страниц.

- Web-страницы следует делать как можно более компактными. Чем компактнее файл, тем быстрее он загружается по сети — это аксиома.
- Web-страницы не должны быть чересчур сложными. Чем сложнее Web-страница, тем больше времени и системных ресурсов требует ее обработка и вывод.
- Web-страницы не должны требовать для отображения никакого дополнительного программного обеспечения. В идеале для их вывода достаточно только Web-обозревателя.

Но как эти правила реализуются на практике? Давайте откроем какой-нибудь современный Web-сайт, например, принадлежащий организации W3C (рис. 1.1). Как мы помним, его можно найти по интернет-адресу **http://www.w3.org**.

Что же мы здесь видим?

- Web-сайт создан с учетом всех современных интернет-стандартов. Он отображается во всех Web-обозревателях практически одинаково.
- Web-сайт не использует ни устаревших, ни закрытых интернет-технологий.
- Структура Web-страниц исключительно ясна — мы можем без проблем найти все, что нужно. Слева находится набор гиперссылок, ведущих на другие Web-страницы Web-сайта, посередине — список новостей и гиперссылки на избранные статьи, справа — гиперссылки на дополнительные материалы.
- Web-страница прекрасно читается. Тонкий шрифт без засечек, спокойная серо-голубая цветовая гамма, тонкие рамочки со скругленными углами, минимум графики — ничто не бросается в глаза.
- Есть даже видеоролик!
- Web-страница быстро загружается и мгновенно выводится на экран.
- Web-страница ничего не требует для своего вывода, кроме Web-обозревателя.

Налицо и соблюдение стандартов, и наполнение, и доступность. Три из трех!

Именно такие Web-страницы мы и будем учиться создавать в данной книге.

Концепция Web 2.0

Давайте еще раз обратимся к рассмотренным ранее правилам и немного расширим их.

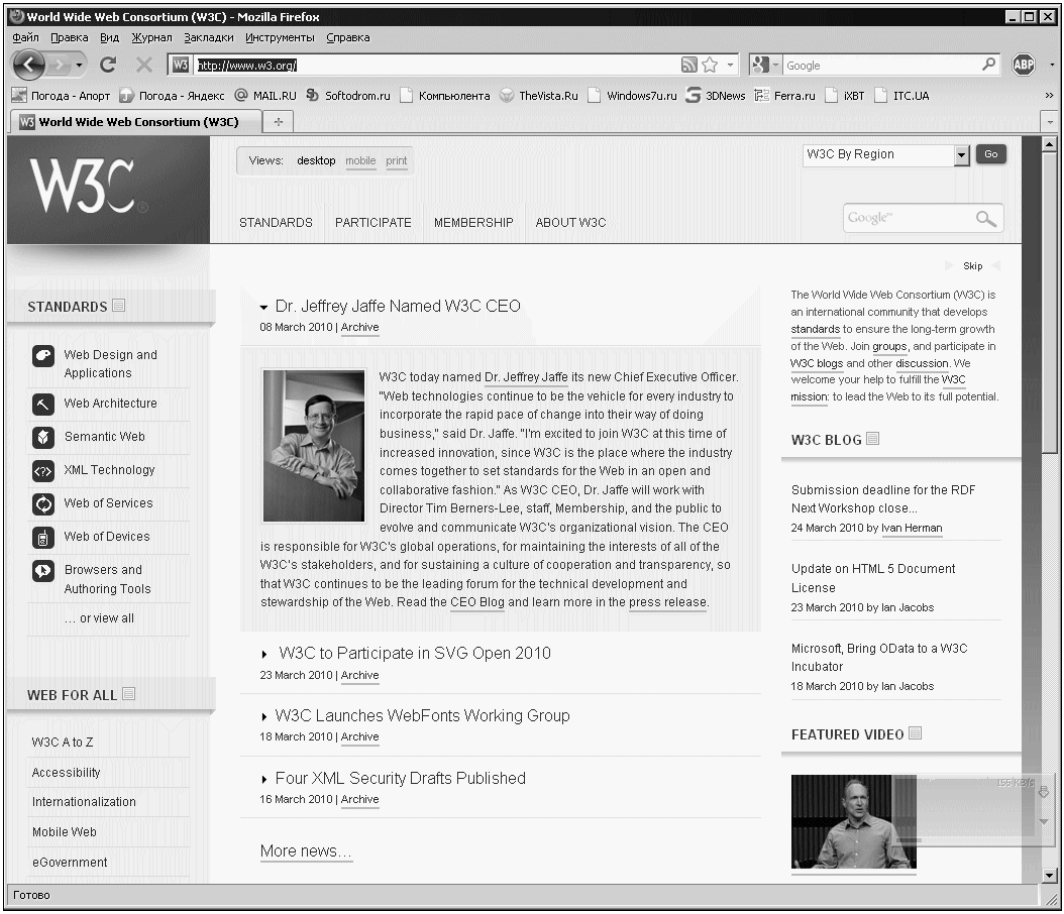


Рис. 1.1. Главная Web-страница Web-сайта организации W3C

- При создании Web-страниц следует придерживаться современных интернет-стандартов. При этом нужно полностью отказаться от устаревших и закрытых интернет-технологий, как не укладывающихся в современную парадигму Web-дизайна и зачастую не поддерживаемых всеми Web-обозревателями.
- Особое внимание нужно обратить на структуру и наполнение Web-страниц. Структура Web-страниц должна быть максимально простой, а наполнение — достаточно богатым, чтобы посетитель быстро нашел нужную ему информацию. Кроме того, необходимо создавать Web-страницы так, чтобы дизайн не мешал восприятию информации.
- Web-страницы обязательно следует делать максимально доступными на любых устройствах. Web-страницы должны быстро загружаться и выводиться на экран. Также Web-страницы не должны требовать для отображения никакого дополнительного программного обеспечения.

Фактически здесь мы привели постулаты так называемой концепции *Web 2.0*. Это список правил, которым должен удовлетворять любой Web-сайт, претендующий на

звание современного. Образно выражаясь, это флаг, который совместно несут труженики Web-индустрии, шагая в ногу со временем.

Также концепция Web 2.0 предусматривает четыре принципа, являющиеся "передним краем" Web-дизайна. Пока еще очень мало Web-сайтов им следует (и "домашний" Web-сайт W3C, увы, не исключение...). Рассмотрим их по порядку.

Принцип первый — разделение содержимого, представления и поведения Web-страницы. Здесь *содержимое* — это информация, которая выводится на Web-странице, *представление* описывает формат вывода этой информации, а *поведение* — реакцию Web-страницы или отдельных ее элементов на действия посетителя. Благодаря их разделению мы сможем править, скажем, содержимое, не затрагивая представление и поведение, или поручать создание содержимого, представления и поведения разным людям.

Принцип второй — *подгружаемое содержимое*. Вместо того чтобы обновлять всю Web-страницу в ответ на щелчок на гиперссылке, мы можем подгружать только ее часть, содержащую необходимую информацию. Это сильно уменьшит объем передаваемой по сети информации (сетевой трафик) и позволит выполнять какие-либо действия с данными после их подгрузки.

Принцип третий — *генерируемое содержимое*. Какая-то часть Web-страницы может не загружаться по сети, а генерироваться прямо на месте, в Web-обозревателе. Так мы еще сильнее сократим сетевой трафик.

Принцип четвертый — *семантическая разметка* данных. Она позволит нам связать выводимые на Web-страницу данные согласно каким-либо правилам. Например, мы можем семантически связать страницы справочника по HTML, и посетитель, загрузив какую-либо страницу, сможет сразу же перейти на связанные с ней страницы, содержащие дополнительные или родственные сведения.

В качестве примера Web-сайта, реализующего эти четыре принципа, можно привести Web-сайт — справочник по библиотеке Ext Core, расположенный по интернет-адресу **http://www.extjs.com/products/core/docs/** и показанный на рис. 1.2.

- Содержимое, представление и поведение составляющих его Web-страниц хранится отдельно, в разных файлах.
- При переходах с одной статьи справочника на другую подгружается только сам текст статьи. Остальные части Web-страницы, в частности иерархический список статей, остаются неизменными.
- После загрузки текста статьи на его основе генерируется окончательное ее представление. Фактически мы имеем генерируемое содержимое.
- Статьи справочника связаны друг с другом семантически. Эти связи используются для генерирования гиперссылок на "родственные" статьи.

Рассмотренные нами два Web-сайта — это концепция Web 2.0 в действии! Хотите создать что-то подобное? Хотите в плане поддержки интернет-стандартов "утереть нос" самому W3C? Тогда читайте эту книгу!

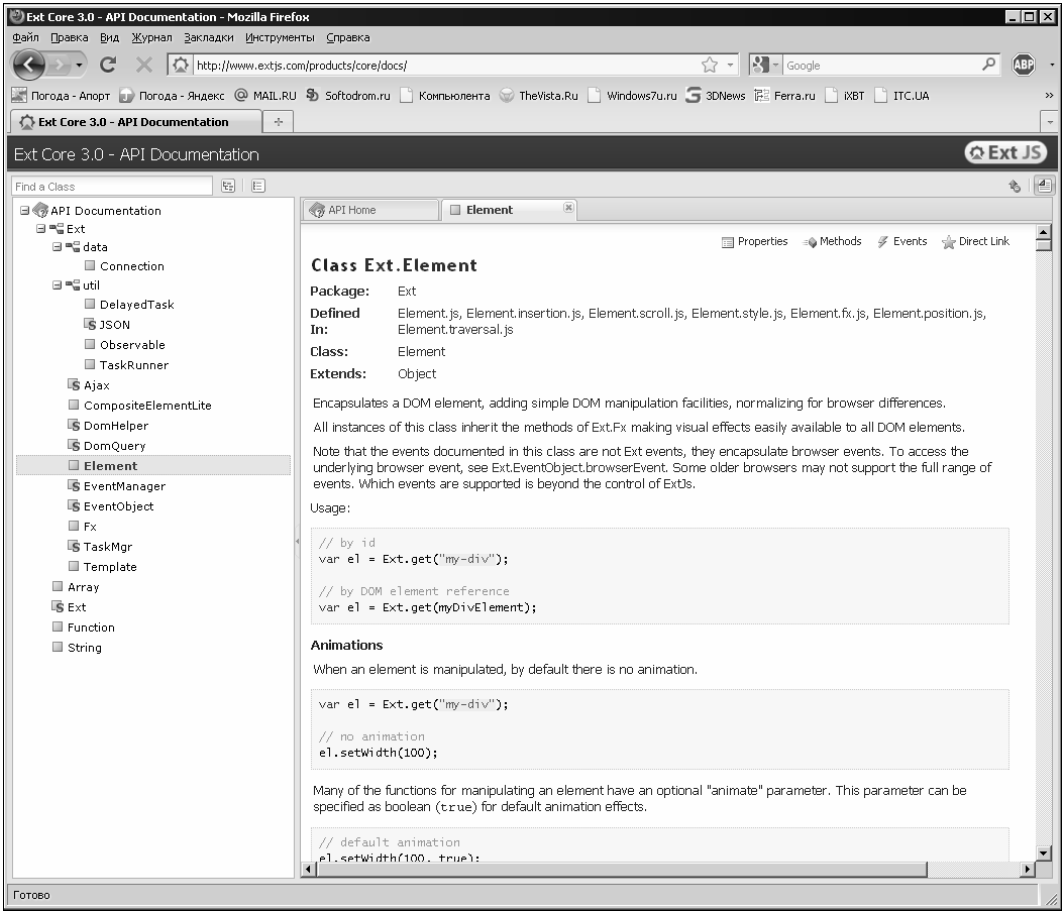


Рис. 1.2. Web-сайт — справочник по библиотеке Ext Core

Интернет: как все это работает

Давайте еще раз посмотрим на Web-сайт — справочник по библиотеке Ext Core. И зададимся вопросом, вынесенным в заголовок данного раздела.

Как все это работает? Откуда Web-обозреватель получает нужную Web-страницу? Кто отвечает за работу сложного механизма под названием Всемирная паутина?

Клиенты и серверы Интернета. Интернет-адреса

Возьмем для примера главную Web-страницу Web-сайта, который мы открыли. Она должна где-то храниться. Но где? На диске другого компьютера, подключенного к сети (в данном случае — к сети Интернет), который может принадлежать как автору Web-сайта, так и сторонней организации, предоставляющей доступ в Интернет *(интернет-провайдеру)*. И хранится она в виде файла или набора файлов, таких же, какие в изобилии "водятся" на нашем собственном компьютере.

Но как мы смогли получить и просмотреть содержимое этого файла? Во-первых, посредством самой сети — она связала компьютер, хранящий файл, с нашим. Во-вторых, с помощью особых программ, которые, собственно, и выполнили передачу файла. Эти программы делятся на две группы.

Программы первой группы взаимодействуют непосредственно с пользователем: принимают от него запросы на информацию, которая хранится где-то в сети, получают ее, выводят на экран и, возможно, позволяют ее править и отправлять обратно. Такие программы называют *клиентами*.

Для просмотра Web-страниц мы пользуемся Web-обозревателем. Это программа-клиент; она принимает от нас интернет-адреса Web-страниц, получает файлы, хранящие их содержимое, и выводит это содержимое на экран. Программа почтового клиента позволяет как извлекать из почтового ящика полученные письма, так и создавать новые. Существуют также клиенты чата, систем мгновенных сообщений и пр.

Но клиенты не имеют прямого доступа к хранящейся на других компьютерах информации. Они не могут просто "залезть" на жесткий диск удаленного компьютера и прочитать оттуда файл. Так сделано из соображений безопасности. Вместо этого они отправляют запросы программам второй группы — серверам.

Серверы работают на компьютерах, хранящих информацию, которая должна быть доступна в сети. Они принимают запросы от клиентов, извлекают требуемую информацию из файлов и отправляют им. Также они могут получать введенную пользователями информацию от клиентов и сохранять их в файлах, при этом, возможно, как-то обработав. Можно сказать, что серверы выступают посредниками между клиентами и запрашиваемой ими информацией.

Для управления Web-сайтами используются *Web-серверы*, которые принимают запросы от клиентов и отправляют им содержимое требуемых файлов. Для управления почтовыми службами применяются серверы электронной почты; они сохраняют пришедшие письма в файлах, выдают их почтовым клиентам по запросу, принимают от клиентов новые сообщения и отправляют их по указанному адресу — в общем, работают как почтовое отделение. Службы чатов и мгновенных сообщений также имеют свои серверы.

Клиенты — лицо Интернета. Серверы — его сердце.

Но как указать, какая информация и с какого сервера нам требуется? С помощью определенным образом составленного интернет-адреса.

Каждая единица информации — файл, ящик электронной почты, канал чата, — доступная в сети, однозначно идентифицируется интернет-адресом, который представляет собой строку из букв, цифр и некоторых других символов.

Интернет-адрес включает в себя две части:

- интернет-адрес программы-сервера, работающей на компьютере;
- указатель на нужную единицу информации, например, путь к файлу, имя ящика электронной почты, имя канала чата и др. (может отсутствовать).

Рассмотрим несколько примеров интернет-адресов.

В интернет-адресе **http://www.somesite.ru/folder1/file1.htm** присутствуют обе части. Здесь **http://www.somesite.ru** — интернет-адрес программы-сервера, в данном случае — Web-сервера, а **/folder1/file1.htm** — путь к запрашиваемому файлу.

В интернет-адресе **http://www.othersite.ru** присутствует только интернет-адрес Web-сервера. Какая информация в этом случае будет отправлена клиенту (Web-обозревателю), мы узнаем потом.

А в адресе **user@mail.someserver.ru** мы видим интернет-адрес сервера электронной почты (**mail.someserver.ru**) и имя почтового ящика (**user**).

Разговор об интернет-адресах еще не закончен. Мы вернемся к нему в *главе 6*, когда будем рассматривать средства навигации по Web-сайту, в частности, гиперссылки. А пока что давайте подробнее поговорим о Web-серверах и их нелегкой "работе".

Web-сайты и Web-серверы

Как мы только что выяснили, все интернет-программы делятся на клиенты и серверы. Клиенты работают на стороне пользователя, получают от них интернет-адреса и выводят им полученную с этих адресов информацию. Серверы принимают запросы от клиентов, находят запрашиваемую ими информацию на дисках серверных компьютеров и отправляют ее клиентам.

Во Всемирной паутине WWW в качестве клиентов используются Web-обозреватели, а в качестве серверов — Web-серверы. Это мы тоже знаем.

Любая информация на дисках компьютера хранится в файлах. Ну, это знает любой более-менее подкованный пользователь...

Web-страницы также хранятся в файлах с расширением htm или html (или, с учетом описанных во введении типографских соглашений, htm[l]). Одна Web-страница занимает один или более файлов.

Web-сайт — это совокупность множества Web-страниц, объединенных общей темой и связанных друг с другом посредством гиперссылок (о них мы поговорим в *главе 6*). Следовательно, Web-сайт — это также набор файлов, возможно, хранящихся в разных папках, — так ими удобнее управлять.

А теперь — внимание! Мы рассмотрим некоторые "интимные" подробности работы Web-серверов, которые знает не каждый интернетчик.

Прежде всего, для хранения всех файлов, составляющих Web-сайт, на диске серверного компьютера выделяется особая папка, называемая *корневой папкой Web-сайта*. Путь к этой папке указывается в настройках Web-сервера, чтобы он смог ее "найти".

Все, повторим — все файлы, составляющие Web-сайт, должны храниться в корневой папке или в папках, вложенных в нее. Файлы, расположенные вне корневой папки, с точки зрения Web-сервера не существуют. Так сделано для безопасности, чтобы злоумышленник не смог получить доступ к дискам серверного компьютера.

Когда в интернет-адресе указывается путь к запрашиваемому файлу, Web-сервер отсчитывает его относительно корневой папки. Это проще всего показать на примерах.

- **http://www.somesite.ru/page1.htm** — в ответ будет отправлен файл page1.htm, хранящийся в корневой папке Web-сайта.
- **http://www.somesite.ru/chapter2/page6.htm** — в ответ будет отправлен файл page6.htm, хранящийся в папке chapter2, которая вложена в корневую папку Web-сайта.
- **http://www.somesite.ru/downloads/others/archive.zip** — в ответ будет отправлен файл archive.zip, хранящийся в папке others, вложенной в папку downloads, которая, в свою очередь, вложена в корневую папку Web-сайта.

Но ведь мы нечасто набираем интернет-адрес, включающий путь к запрашиваемому файлу. Гораздо чаще интернет-адреса включают только адрес программы-сервера, например, **http://www.somesite.ru**. Что в таком случае делает Web-сервер? Какой файл он отправляет в ответ?

Специально для этого предусмотрены так называемые *Web-страницы по умолчанию*. Такая Web-страница выдается клиенту, если он указал в интернет-адресе только путь к файлу, но не его имя. Обычно файл Web-страницы по умолчанию имеет имя default.htm[l] или index.htm[l], хотя его можно изменить в настройках Web-сервера.

Так, если мы наберем интернет-адрес **http://www.somesite.ru**, Web-сервер вернет нам файл Web-страницы по умолчанию, хранящийся в корневой папке Web-сайта. Практически всегда это будет *главная Web-страница* — та, с которой начинается "путешествие" по Web-сайту.

Мы можем набрать и интернет-адрес вида **http://www.somesite.ru/chapter2/**. Тогда Web-сервер отправит нам файл Web-страницы по умолчанию, хранящийся в папке chapter2, вложенной в корневую папку Web-сайта.

С Web-сайтами и Web-серверами пока все. Настала пора заглянуть внутрь Web-страниц и, чего уж тянуть резину, создать нашу первую, совсем простую Web-страничку. И по ходу дела начать изучение языка HTML 5, без которого в Web-дизайне не обойтись.

Основные принципы создания Web-страниц. Язык HTML 5

Web-страницы выглядят зачастую очень пестро: разнокалиберные куски текста, таблицы, картинки, врезки, сноски и даже фильмы. Но описывается все это в виде обычного текста. Да-да, Web-страницы — суть текстовые файлы, которые можно создать с помощью хорошо знакомого нам редактора Блокнот, поставляемого в составе Windows! (Разумеется, подойдет любой аналогичный текстовый редактор.)

Для форматирования содержимого Web-страниц применяется особый язык — *HTML* (HyperText Markup Language, язык гипертекстовой разметки). С помощью

команд — *тегов* — этого языка создают и абзацы текста, и заголовки, и врезки, и даже таблицы.

Первая версия языка HTML появилась очень давно, еще в 1992 году. С тех пор по Сети утекло немало гигабайт... HTML также не стоял на месте. В данный момент готовится к выходу окончательная спецификация новой версии HTML под номером 5, и многие Web-обозреватели уже поддерживают некоторые ее возможности. Ее-то мы и будем изучать.

Язык HTML и его теги

Изучать HTML лучше всего на примере. Так что давайте сразу же создадим нашу первую Web-страничку. Благо Windows уже содержит необходимый для этого инструмент — Блокнот.

> ***НА ЗАМЕТКУ***
>
> Вообще, для создания Web-страниц существует множество специальных программ — Web-редакторов. Они позволяют работать с Web-страницами, даже не зная HTML, — как с документами Microsoft Word, просто набирая текст и форматируя его. Также они следят за правильностью расстановки тегов, помогут быстро создать сложный элемент Web-страницы и даже опубликовать готовый Web-сайт в Сети. К таким программам принадлежит, в частности, известный Web-редактор Adobe Dreamweaver.
>
> Однако мы пока что будем пользоваться простейшим текстовым редактором Блокнот. Это позволит нам лучше познакомиться с HTML.

Откроем Блокнот и наберем в нем текст (или, как говорят бывалые программисты, код), приведенный в листинге 1.1.

Листинг 1.1

```
<!DOCTYPE html>
<HTML>
  <HEAD>
    <META HTTP-EQUIV="Content-Type" CONTENT="text/html; charset=utf-8">
    <TITLE>Пример Web-страницы</TITLE>
  </HEAD>
  <BODY>
    <H1>Справочник по HTML</H1>
    <P>Приветствуем на нашем Web-сайте всех, кто занимается Web-дизайном!
    Здесь вы сможете найти информацию обо всех интернет-технологиях,
    применяемых при создании Web-страниц. В частности, о языке
    <STRONG>HTML</STRONG>.</P>
  </BODY>
</HTML>
```

Проверим набранный код на ошибки и сохраним в файл с именем 1.1.htm. Только сделаем при этом две важные вещи.

1. Сохраним HTML-код в кодировке UTF-8. Для этого в диалоговом окне сохранения файла Блокнота найдем раскрывающийся список **Кодировка** и выберем в нем пункт **UTF-8**.

2. Заключим имя файла в кавычки. Иначе Блокнот добавит к нему расширение txt, и наш файл получит имя 1.1.htm.txt.

Все, наша первая Web-страница готова! Теперь осталось открыть ее в Web-обозревателе и посмотреть на результат.

Мы можем использовать стандартно поставляемый в составе Windows Web-обозреватель Microsoft Internet Explorer. Но Internet Explorer на данный момент не поддерживает HTML 5; его поддержку обещают только в версии 9, которая пока находится в разработке. HTML 5 поддерживают последние версии Mozilla Firefox, Opera, Apple Safari и Google Chrome, поэтому предпочтительнее какая-либо из этих программ.

Откроем же Web-страницу в выбранном Web-обозревателе (автор выбрал Firefox) и посмотрим на нее (рис. 1.3).

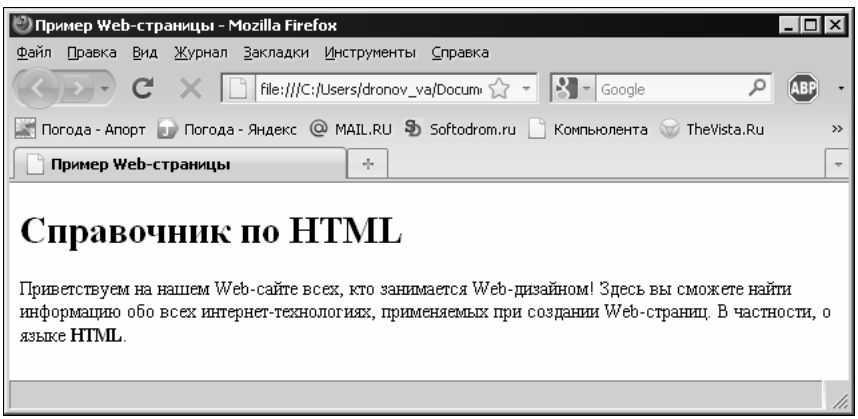


Рис. 1.3. Наша первая Web-страница

Видите? Мы создали Web-страницу, содержащую большой "кричащий" заголовок, абзац текста, который автоматически разбивается на строки и содержит фрагмент текста, выделенный полужирным шрифтом (аббревиатура "HTML"). И все это — в "голом" тексте, набранном в Блокноте!

Теперь посмотрим, что же мы такое написали в файле 1.1.htm. Пока что ограничимся небольшим фрагментом HTML-кода (листинг 1.2).

Листинг 1.2

```
<H1>Справочник по HTML</H1>
<P>Приветствуем на нашем Web-сайте всех, кто занимается Web-дизайном!
Здесь вы сможете найти информацию обо всех интернет-технологиях,
применяемых при создании Web-страниц. В частности, о языке
<STRONG>HTML</STRONG>.</P>
```

Здесь мы видим текст заголовка и абзаца. И еще странные слова, взятые в угловые скобки — символы < и >. Что это такое?

Это и есть теги HTML, о которых упоминалось ранее. Они превращают тот или иной фрагмент HTML-кода в определенный элемент Web-страницы: абзац, заголовок или текст, выделенный полужирным шрифтом.

Начнем с тегов <H1> и </H1>, поскольку они идут первыми. Эти теги превращают фрагмент текста, находящийся между ними, в заголовок. Тег <H1> помечает начало фрагмента, на который распространяется действие тега, и называется *открывающим*. А тег </H1> устанавливает конец "охватываемого" фрагмента и называется *закрывающим*. Что касается самого фрагмента, заключенного между открывающим и закрывающим тегами, то он называется *содержимым тега*. Именно к содержимому применяется действие тега.

Все теги HTML представляют собой символы < и >, внутри которых находится *имя тега*, определяющее назначение тега. Закрывающий тег должен иметь то же имя, что и открывающий; единственное отличие закрывающего тега — символ /, который ставится между символом < и именем тега.

Рассмотренные нами теги <H1> и </H1> в HTML фактически считаются одним тегом <H1>. Такой тег называется *парным*.

Поехали дальше. Парный тег <P> создает на Web-странице абзац; содержимое тега станет текстом этого абзаца. Такой абзац будет отображаться с отступами сверху и снизу. Если он полностью помещается по ширине в окне Web-обозревателя, то отобразится в одну строку; в противном случае сам Web-обозреватель разобьет его на несколько более коротких строк. (То же справедливо и для заголовка.)

Парный тег <STRONG> выводит свое содержимое полужирным шрифтом. Как мы видим, тег <STRONG> вложен внутрь содержимого тега <P>. Это значит, что содержимое тега <STRONG> будет отображаться как часть абзаца (тега <P>).

Давайте ради интереса выделим слова "Web-дизайном" курсивом. Для этого поместим соответствующий фрагмент текста абзаца в парный тег <EM>:

```
<P>Приветствуем на нашем Web-сайте всех, кто занимается
<EM>Web-дизайном</EM>! Здесь вы сможете найти информацию обо всех
. . .
```

Сохраним исправленную Web-страницу и обновим содержимое окна Web-обозревателя, нажав клавишу <F5>. Получилось! Да мы уже стали Web-дизайнерами!

Осталось рассмотреть важнейшие правила, согласно которым пишется HTML-код.

- Имена тегов можно писать как прописными (большими), так и строчными (малыми) буквами. Традиционно в языке HTML имена тегов пишут прописными буквами.
- Между символами <, >, / и именами тегов, а также внутри имен тегов не допускаются пробелы и переносы строк.
- В обычном тексте, не являющемся тегом, не должны присутствовать символы < и >. (Эти символы называют *недопустимыми*.) В противном случае Web-обозреватель сочтет фрагмент текста, где встречается один из этих символов, тегом и отобразит Web-страницу некорректно.

На этом пока закончим. Впоследствии, изучив другие языковые элементы HTML, мы пополним список этих правил.

Вложенность тегов

Если мы снова посмотрим на приведенный в листинге 1.2 фрагмент HTML-кода, то заметим, что одни теги вложены в другие. Так, тег <STRONG> вложен в тег <P>, являясь частью его содержимого. Тег <P>, в свою очередь, вложен в тег <BODY>, а тот — в "глобальный" тег <HTML>. (Теги <BODY> и <HTML> мы рассмотрим чуть позже.) Такая *вложенность тегов* в HTML — обычное явление.

Когда Web-обозреватель встречает тег, вложенный в другой тег, он как бы накладывает действие "внутреннего" тега на эффект "внешнего". Так, действие тега <STRONG> будет наложено на действие тега <P>, и фрагмент абзаца окажется выделенным полужирным шрифтом, при этом оставаясь частью этого абзаца.

Давайте для примера текст "Web-дизайн", который мы недавно поместили в тег <EM>, заключим еще и в тег <STRONG>. Вот так:

```
<P>Приветствуем на нашем Web-сайте всех, кто занимается
<EM><STRONG>Web-дизайном</STRONG></EM>! Здесь вы сможете найти
. . .
```

В этом случае данный текст будет выделен полужирным курсивом. Иными словами, действие тега <STRONG> будет наложено на действие тега <EM>.

Теперь — внимание! Порядок следования закрывающих тегов должен быть обратным тому, в котором следуют теги открывающие. Говоря иначе, теги со всем их содержимым должны полностью вкладываться в другие теги, не оставляя "хвостов" снаружи.

Если же мы нарушим это правило и напишем такой HTML-код (обратите внимание на специально перепутанный порядок следования открывающих тегов):

```
<P>Приветствуем на нашем Web-сайте всех, кто занимается
<EM><STRONG>Web-дизайном</EM></STRONG>! Здесь вы сможете найти
. . .
```

Web-обозреватель может отобразить нашу Web-страницу неправильно.

> ***НА ЗАМЕТКУ***
>
> Нужно сказать, что современные Web-обозреватели "умеют" исправлять мелкие ошибки Web-дизайнера. Но именно мелкие!

Осталось выучить несколько новых терминов. Тег, в который непосредственно вложен данный тег, называется *родительским*, или *родителем*. В свою очередь, тег, вложенный в данный тег, называется *дочерним*, или *потомком*. Так, для тега <EM> в приведенном далее примере тег <P> — родительский, а тег <STRONG> — дочерний. Любой тег может иметь сколько угодно дочерних тегов, но только один родительский (что, впрочем, понятно — не может же он быть непосредственно вложен одновременно в два тега).

Элемент Web-страницы, в который вложен элемент, создаваемый данным тегом, называется *родительским*, или *родителем*. А элемент Web-страницы, который

вложен в данный элемент, — *дочерним*, или *потомком*. То же самое, что и в случае тегов.

Уровень вложенности того или иного тега показывает количество тегов, в которые он последовательно вложен. Если принять за точку отсчета тег <P>, то тег <EM> будет иметь первый уровень вложенности, т. к. он вложен непосредственно в тег <P>. Тег <STRONG> же будет иметь второй уровень вложенности, поскольку он вложен в тег <EM>, а тот, в свою очередь, — в тег <P>. В сложных же Web-страницах уровень вложенности иных тегов может составлять несколько десятков.

Уровень вложенности тегов в HTML-коде обозначают с помощью отступов, которые ставят слева от соответствующего тега и создают с помощью пробелов (листинг 1.3). На отображение Web-страницы они никак не влияют.

Листинг 1.3

```
<BODY>
  <H1>Справочник по HTML</H1>
  <P>Приветствуем на нашем Web-сайте всех, кто занимается Web-дизайном!
  Здесь вы сможете найти информацию обо всех интернет-технологиях,
  применяемых при создании Web-страниц. В частности, о языке
  <STRONG>HTML</STRONG>.</P>
</BODY>
```

Здесь сразу видно, что теги <H1> и <P> вложены в тег <BODY>, — видно по отступам.

Секции Web-страницы

Снова вернемся в полному HTML-коду нашей Web-странички. Мысленно удалим из него уже рассмотренный фрагмент и получим листинг 1.4.

Листинг 1.4

```
<!DOCTYPE html>
<HTML>
  <HEAD>
    <META HTTP-EQUIV="Content-Type" CONTENT="text/html; charset=utf-8">
    <TITLE>Пример Web-страницы</TITLE>
  </HEAD>
  <BODY>
    . . .
  </BODY>
</HTML>
```

Здесь применены несколько тегов, которые нам не знакомы. Это так называемые *невидимые теги* — теги, содержимое которых никак не отображается Web-обозревателем. Они занимаются тем, что хранят сведения о параметрах самой Web-страницы и делят ее на две *секции*, имеющие принципиально разное назначение.

Секция тела Web-страницы находится внутри парного тега <BODY>. Она описывает само содержимое Web-страницы, то, что будет выведено на экран. Именно секцию тела мы рассматривали в предыдущих разделах.

А в парном теге <HEAD> находится *секция заголовка* Web-страницы. (Не путать с заголовком, который создается с помощью тега <H1>!) В эту секцию помещают сведения о параметрах Web-страницы, не отображаемые на экране и предназначенные исключительно для Web-обозревателя.

И заголовок, и тело Web-страницы находятся внутри парного тега <HTML>, который расположен на самом высшем (нулевом) уровне вложенности и не имеет родителя.

Любая Web-страница должна быть правильно отформатирована: иметь секции заголовка и тела и все соответствующие им теги. Только в таком случае она будет считаться корректной с точки зрения стандартов HTML.

Метаданные и тип Web-страницы

Вернемся к сведениям о параметрах Web-страницы, которые находятся в секции ее заголовка. Что это за параметры? И что они задают?

Сначала введем еще пару терминов. Параметры Web-страницы, не отображаемые на экране и предназначенные для Web-обозревателя, назовем *метаданными*. Это своего рода данные, описывающие другие данные, в нашем случае — Web-страницу. А HTML-теги, которые задают метаданные, называются *метатегами*.

Прежде всего, в метаданные входит *название* Web-страницы. Оно отображается в заголовке окна Web-обозревателя, где выводится содержимое данной Web-страницы, и хранится в "истории" (списке посещенных к настоящему времени Web-страниц). Название помещается в парный тег <TITLE> и располагается в секции заголовка Web-страницы:

```
<HEAD>
  . . .
  <TITLE>Пример Web-страницы</TITLE>
</HEAD>
```

Далее, обычно в секции заголовка расположен особый метатег, задающий кодировку, в которой сохранена Web-страница. Этот метатег имеет "говорящее" имя <META>:

```
<HEAD>
  <META HTTP-EQUIV="Content-Type" CONTENT="text/html; charset=utf-8">
  . . .
</HEAD>
```

Приведенный тег задает кодировку UTF-8, в которой мы сохранили нашу Web-страничку. Существуют аналогичные теги, задающие кодировки 1251, КОИ-8, кодировка западноевропейских и восточноазиатских языков и др.

> ***НА ЗАМЕТКУ***
>
> Кодировка *UTF-8* — это разновидность кодировки Unicode, предназначенная для Web-дизайна. Кодировка Unicode (а значит, и UTF-8) может закодировать все символы всех

языков, имеющихся на Земле. Именно она в настоящее время чаще всего применяется для создания Web-страниц.

Кстати, вы не заметили ничего странного в теге <META>? У него нет ни содержимого, ни закрывающей пары! Это так называемый *одинарный* тег, который имеет только открывающую пару. Такой тег действует в той точке HTML-кода, где он сам находится, и либо задает метаданные, либо помещает в соответствующее место Web-страницы какой-либо элемент, не относящийся к тексту. Впоследствии нам будут часто встречаться одинарные теги.

Теперь осталось рассмотреть последний тег, находящийся в самом начале HTML-кода нашей Web-страницы. Этот тег находится даже вне "всеобъемлющего" тега <HTML>. Важная, должно быть, персона... Вот он:

```
<!DOCTYPE html>
```

Метатег <!DOCTYPE> задает, во-первых, версию языка HTML, на которой написана Web-страница, а во-вторых, разновидность данной версии. Так, существуют метатеги <!DOCTYPE>, указывающие на HTML 5, "строгую" и "переходную" разновидности HTML 4.01 (это предыдущая версия языка HTML, еще действующая на данный момент) и язык XHTML (ответвление HTML, имеющее несколько другой синтаксис).

Так вот, метатег <!DOCTYPE html>, который мы поставили в начало нашей Web-странички, указывает на HTML 5. Будем работать только с самыми новыми технологиями! Долой всякое старье!

Атрибуты HTML-тегов

Последний важный вопрос, который мы здесь рассмотрим, — атрибуты HTML-тегов. После этого мы пока что закончим с HTML и обратимся к принципам современного Web-дизайна.

Посмотрим на тег <META>, задающий кодировку Web-страницы:

```
<META HTTP-EQUIV="Content-Type" CONTENT="text/html; charset=utf-8">
```

Здесь мы видим, что между символами < и >, помимо имени тега, присутствуют еще какие-то данные. Это *атрибуты тега*, задающие его параметры. В частности, два атрибута данного тега <META> указывают, что документ представляет собой Web-страницу, и задают ее кодировку.

Каждый атрибут тега имеет *имя*, за которым ставится знак равенства, и *значение* данного атрибута, взятое в двойные кавычки. Так, атрибут с именем HTTP-EQUIV имеет значение "Content-Type", говорящее о том, что данный метатег задает тип документа. А атрибут с именем CONTENT имеет значение "text/html; charset=utf-8", обозначающее, что данный документ представляет собой Web-страницу, и указывающее, что она набрана в кодировке UTF-8.

Атрибуты тегов бывают обязательными и необязательными. *Обязательные* атрибуты должны присутствовать в теге в обязательном порядке. *Необязательные* же ат-

рибуты могут быть опущены; в таком случае тег ведет себя так, будто соответствующему атрибуту присвоено значение по умолчанию.

Атрибуты HTTP-EQUIV и CONTENT тега <META> обязательные — кому нужен метатег без метаданных... А вот атрибут ID, поддерживаемый практически всеми тегами HTML, необязательный, он используется только в особых случаях:

```
<H1 ID="header1">Справочник по HTML</H1>
```

В *частях II* и *III*, работая со стилями CSS и Web-сценариями, мы будем активно пользоваться атрибутом тега ID. А пока что оставим его в покое.

Ранее мы изучили три правила написания HTML-кода. Добавим к ним еще шесть.

- Имена атрибутов тегов могут быть написаны как прописными (большими), так и строчными (малыми) буквами. Традиционно в языке HTML имена атрибутов тегов пишут прописными буквами, а их значения — строчными, если, конечно, значение не чувствительно к регистру букв.
- Имена атрибутов тегов пишут между символами < и > после имени тега и отделяют от него пробелом или разрывом строки. Если в теге присутствуют несколько атрибутов, их отделяют друг от друга также пробелами или разрывами строки.
- Внутри имен атрибутов не должны присутствовать пробелы, в противном случае Web-обозреватель посчитает, что это не один атрибут, а несколько.
- Значение атрибута тега пишут после его имени и заключают в двойные кавычки. Между именем атрибута тега и его значением ставят знак равенства.
- Между именем атрибута тега, знаком равенства и открывающими кавычками могут присутствовать пробелы или разрывы строк.
- Символы двойных кавычек недопустимы и не должны присутствовать в обычном тексте, иначе Web-обозреватель посчитает следующий за ними текст значением атрибута тега.

На этом пока закончим с HTML. В последующих главах *части I* мы продолжим его изучение.

Программы, которыми мы будем пользоваться

Поговорим о программах, с которыми будем работать. Не считая Блокнота (или аналогичного текстового редактора), таких программ две: Web-обозреватель и Web-сервер.

Web-обозреватель

Web-обозревателей на свете довольно много. Но если отбросить старые и специализированные, останется всего пять: Microsoft Internet Explorer, Mozilla Firefox, Opera, Google Chrome и Apple Safari.

Разумеется, все эти программы поддерживают языки HTML, CSS (язык описания каскадных таблиц стилей) и JavaScript (язык, на котором пишутся Web-сценарии). Большинство этих программ поддерживает некоторый набор возможностей HTML 5 и CSS 3 (новая версия CSS):

- Mozilla Firefox — начиная с версии 3.5;
- Opera — начиная с версии 9.5;
- Google Chrome — начиная с версии 2.0.158.0;
- Apple Safari — начиная с версии 4.0.

> ***НА ЗАМЕТКУ***
>
> Отдельные возможности HTML 5 и CSS 3 начали поддерживаться и в более ранних версиях этих программ. Однако, чтобы успешно просматривать Web-страницы, которые мы будем создавать в процессе изучения этой книги, следует использовать указанные версии или более новые.

А что же Microsoft Internet Explorer? Поддерживает ли он HTML 5 и CSS 3 на данный момент? К сожалению, нет. Поддержку HTML 5 и CSS 3 обещают лишь в версии 9.0, которая сейчас только разрабатывается.

Но какой же Web-обозреватель нам выбрать? Какой угодно из перечисленных представителей "передовой четверки" — Firefox, Opera, Chrome или Safari. А лучше — сразу все, чтобы удостовериться, что наши Web-страницы в любом из них отображаются правильно.

Web-сервер

Когда мы тестировали нашу первую Web-страницу, то прекрасно обошлись без Web-сервера, открыв ее прямо в Web-обозревателе. Но в дальнейшем, особенно когда мы начнем реализовывать подгрузку содержимого, Web-сервер все-таки нам понадобится. Многие Web-сценарии нормально работают только в том случае, если Web-страница загружена с Web-сервера. Это сделано ради безопасности.

Так что давайте займемся поиском подходящего Web-сервера.

К счастью, долго искать нам не придется. В составе Windows XP/2003/Vista/Se7en поставляется вполне серьезный Web-сервер Microsoft Internet Information Services. Он исключительно прост в установке и настройке, а его возможностей нам вполне хватит.

Как установить и запустить Internet Information Services, описано в документации, поставляемой в составе Windows. Настраивать его, как правило, не нужно — он сразу после установки "бросается в бой". Более того, Web-сервер Microsoft после установки создает небольшой тестовый Web-сайт, корневая папка которого находится по пути C:\Inetpub\wwwroot. Этой папкой мы и воспользуемся для тестирования наших Web-страничек.

> ***НА ЗАМЕТКУ***
>
> Если вам не нравится Internet Information Services, вы можете использовать любой другой. Например, популярный Web-сервер Apache, хотя его придется настраивать сразу после установки.

Вообще, какой именно Web-сервер установлен, не принципиально. Главное — чтобы он был.

Теперь испытаем свежеустановленный Web-сервер в действии. Для этого нам понадобится любая программа управления файлами (например, поставляемый в составе Windows Проводник). Откроем в ней корневую папку созданного при установке Web-сайта (в случае Internet Information Services это папка C:\Inetpub\wwwroot).

ВНИМАНИЕ!

Если вы пользуетесь Internet Information Services, поставляемым с Windows Vista или Windows Se7en, то имейте в виду, что папка C:\Inetpub\wwwroot защищена *UAC* (User Access Control, управление доступом пользователя) — новой системой защиты ключевых папок и файлов, встроенной в эти операционные системы. Поэтому вам придется сначала запустить Проводник или иную программу управления файлами, которой вы пользуетесь, от имени администратора. Для этого достаточно щелкнуть на ярлыке этой программы правой кнопкой мыши, выбрать в появившемся на экране контекстном меню пункт **Запуск от имени администратора** (Run As Administrator) и положительно ответить на предупреждение UAC.

Удалим из корневой папки все имеющиеся там файлы, чтобы они нам не мешали. И скопируем туда нашу первую Web-страницу 1.1.htm.

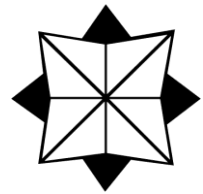
Теперь откроем выбранный ранее Web-обозреватель и наберем в нем интернет-адрес **http://localhost/1.1.htm**. Интернет-адрес **http://localhost** идентифицирует наш собственный компьютер *(локальный хост)*, а /1.1.htm — указание на файл 1.1.htm, хранящийся в корневой папке Web-сайта.

Если мы все сделали правильно, Web-обозреватель отобразит нашу Web-страницу (см. рис. 1.3). Значит, Web-сервер работает.

Что дальше?

В этой главе мы познакомились с современными веяниями в Web-дизайне, узнали о концепции Web 2.0, поняли, как работает Интернет в целом и WWW в частности, начали изучать язык HTML, создали свою первую Web-страницу, и выбрали себе для работы Web-обозреватель и Web-сервер. В общем, неплохо стартовали...

Теперь на какое-то время мы оставим в покое принципы и концепции и сосредоточимся на практике, а именно на HTML. В следующей главе мы рассмотрим средства структурирования текста: абзацы, заголовки, списки и цитаты. Так сказать, научимся делить текст на "куски".



ГЛАВА 2

Структурирование текста

В предыдущей главе мы прошли "ударный" курс современного Web-дизайна, концепции Web 2.0, интернет-технологий и языка HTML. Мы выучили много новых терминов, создали свою первую Web-страничку и даже испытали в действии собственный Web-сервер. Теперь мы готовы к любым трудностям.

В этой главе мы продолжим изучение языка HTML и рассмотрим средства для структурирования текста — разбиения его на отдельные значащие фрагменты, имеющие различное назначение и несущие различный смысл. Такими фрагментами являются абзацы, заголовки, списки и цитаты.

Абзацы

Если оформить текст в виде большого монолитного "куска", его вряд ли кто-то будет читать. Такой "кусок" текста выглядит как высокий черный забор, за которым не видно ни единой мысли автора, забор без единой дверцы, без единой щелочки.

Именно поэтому текст всегда разбивают на абзацы. Небольшие, включающие по несколько связанных по смыслу предложений, они доносят авторский текст постепенно, по частям, от простого к сложному. В общем, превращают непроницаемый "забор" в читабельный текст.

Как мы уже знаем из *главы 1*, язык HTML для создания абзаца предоставляет парный тег `<P>`. Содержимое этого тега становится текстом абзаца:

```
<P>Я — совсем короткий абзац.</P>
<P>А я — уже более длинный абзац. Возможно, Web-обозреватель разобьет
меня на две строки.</P>
```

Абзац HTML отделяется небольшим отступом от предыдущего и последующего элементов страницы. Если абзац полностью помещается по ширине в родительский элемент Web-страницы, он будет выведен в одну строку; в противном случае Web-обозреватель разобьет его текст на несколько более коротких строк.

Абзац — это независимый элемент Web-страницы, который отображается отдельно от других элементов. Такие элементы Web-страницы называются *блочными*, или *блоками*.

Правила отображения текста абзаца Web-обозревателем:

- два и более следующих друг за другом пробела считаются за один пробел;
- перевод строки считается за пробел;
- пробелы и переводы строки между тегами, создающие блочные элементы, никак не отображаются на Web-странице. (Благодаря этому мы можем форматировать HTML-код для более удобного чтения, в том числе ставить отступы для обозначения вложенности тегов.)

Эти же правила справедливы для других блочных элементов, которые мы изучим в этой главе.

Настало время попрактиковаться. Давайте создадим главную Web-страницу нашего первого Web-сайта — справочника по HTML и CSS. Откроем Блокнот и наберем в нем HTML-код, приведенный в листинге 2.1.

Листинг 2.1

```
<!DOCTYPE html>
<HTML>
  <HEAD>
    <META HTTP-EQUIV="Content-Type" CONTENT="text/html; charset=utf-8">
    <TITLE>Справочник по HTML и CSS</TITLE>
  </HEAD>
  <BODY>
    <P>Справочник по HTML и CSS</P>
    <P>Приветствуем на нашем Web-сайте всех, кто занимается Web-дизайном!
    Здесь вы сможете найти информацию обо всех интернет-технологиях,
    применяемых при создании Web-страниц. А именно, о языках HTML и
    CSS.</P>
    <P>Русская Википедия определяет термин HTML так:</P>
    <P>HTML (от англ. HyperText Markup Language — язык разметки
    гипертекста) — стандартный язык разметки документов во Всемирной
    паутине.</P>
    <P>Пожалуй, ни убавить ни прибавить...</P>
    <P>HTML позволяет формировать на Web-страницах следующие
    элементы:</P>
    <P>абзацы;</P>
    <P>заголовки;</P>
    <P>цитаты;</P>
    <P>списки;</P>
    <P>таблицы;</P>
    <P>графические изображения;</P>
    <P>аудио- и видеоролики.</P>
    <P>Основные принципы HTML</P>
    <P>. . .</P>
    <P>Теги HTML</P>
```

```
    <P>!DOCTYPE, BODY, EM, HEAD, HTML, META, P, STRONG, TITLE</P>
  </BODY>
</HTML>
```

Пока это только заготовка для главной Web-страницы. Позднее мы будем дополнять и править ее.

Создадим папку, куда будем "складывать" файлы, составляющие наш Web-сайт. И сохраним в этой папке набранный HTML-код, дав файлу имя index.htm. Как мы помним из *главы 1*, файл главной Web-страницы Web-сайта должен иметь имя index.htm[l] (или default.htm[l], но это встречается реже).

> ***Внимание!***
>
> Впоследствии мы будем сохранять все файлы, составляющие наш Web-сайт, в специально созданной папке. Если какой-либо файл нужно сохранить где-то еще, автор сообщит об этом особо.

Сразу же откроем созданную Web-страницу в Web-обозревателе — так мы сразу сможем определить, нет ли ошибок. Если ошибки все-таки есть, исправим их.

Пока что наша Web-страничка содержит одни абзацы. Первое, что мы должны в нее добавить, — это...

Заголовки

Помимо абзацев, большой текст для удобства чтения и поиска в нем нужного фрагмента обычно делят на более крупные части: параграфы, главы, разделы. HTML не предоставляет средств для такого структурирования текста. Но он позволяет создать заголовки, которые делят текст на части, по крайней мере, визуально. Как в обычной "бумажной" книге.

Прежде всего, уясним, что в HTML есть понятие *уровня заголовка*, указывающее, насколько крупную часть текста открывает данный заголовок. Всего таких уровней шесть, и обозначаются они числами от 1 до 6.

- Заголовок первого уровня (1) открывает самую крупную часть текста. Как правило, это заголовок всей Web-страницы. Web-обозреватель выводит заголовок первого уровня самым большим шрифтом.
- Заголовок второго уровня (2) открывает более мелкую часть текста. Обычно это большой раздел. Web-обозреватель выводит заголовок второго уровня меньшим шрифтом, чем заголовок первого уровня.
- Заголовок третьего уровня (3) открывает еще более мелкую часть текста; обычно главу в разделе. Web-обозреватель выводит такой заголовок еще меньшим шрифтом.
- Заголовки четвертого, пятого и шестого уровней (4–6) открывают отдельные параграфы, крупные, более мелкие и самые мелкие соответственно. Web-обозреватель выводит заголовки четвертого и пятого уровня еще меньшим шрифтом, а заголовок шестого уровня — тем же шрифтом, что и обычные абзацы, только полужирным.

На Web-страницах небольшого и среднего размера обычно применяют заголовки первого, второго и третьего уровня. Меньшие уровни используются только на очень больших Web-страницах, содержащих сложно структурированный текст.

Для создания заголовка HTML предоставляет парный тег <Hn>, где *n* — уровень заголовка. Содержимое этого тега станет текстом заголовка (листинг 2.2).

Листинг 2.2

```
<H1>Я — заголовок Web-страницы, самый главный</H1>
<H2>Я — заголовок раздела</H2>
<H3>Я — заголовок главы</H3>
<H4>Я — заголовок крупного параграфа</H4>
<H5>Я — заголовок параграфа поменьше</H5>
<H6>А я — заголовок маленького параграфа. Ой, какие все вокруг
большие!..</H6>
```

Заголовок также относится к блочным элементам Web-страницы. При его выводе на экран Web-обозреватель следует тем же правилам, что и для абзаца.

Заголовки — это то, чего не хватает нашей Web-страничке index.htm. Давайте их добавим (листинг 2.3).

Листинг 2.3

```
<H1>Справочник по HTML и CSS</H1>
. . .
<H2>Основные принципы HTML</H2>
. . .
<H2>Теги HTML</H2>
```

Мы просто заменили теги <P> в соответствующих фрагментах HTML-кода на теги <H1> и <H2>. Теперь можем открыть Web-страницу в Web-обозревателе и посмотреть на результат.

Списки

Списки используются для того, чтобы представить читателю перечень каких-либо позиций, пронумерованных или непронумерованных, — пунктов списка. Список с пронумерованными пунктами так и называется — *нумерованным*, а с непронумерованными — *маркированным*. В маркированных списках пункты помечаются особым значком — *маркером*, который ставится левее пункта списка.

Маркированные списки обычно служат для простого перечисления каких-либо позиций, порядок следования которых не важен. Если же следует обратить внимание читателя на то, что позиции должны следовать друг за другом именно в том порядке, в котором они перечислены, следует применить нумерованный список.

Web-обозреватель выводит список с отступом слева. Расстояние между пунктами списка он делает меньшими чем в случае абзацев или заголовков. Также он сам расставляет необходимые маркеры или нумерацию.

Любой список в HTML создается в два этапа. Сначала пишут строки, которые станут пунктами списка, и каждую из этих строк помещают внутрь парного тега <LI>. Затем все эти пункты помещают внутрь парного тега <UL> (если создается маркированный список) или <OL> (при создании нумерованного списка) — эти теги определяют сам список (листинг 2.4).

Листинг 2.4

```
<UL>
  <LI>Я — первый пункт маркированного списка.</LI>
  <LI>Я — второй пункт маркированного списка.</LI>
  <LI>Я — третий пункт маркированного списка.</LI>
</UL>
. . .
<OL>
  <LI>Я — первый пункт нумерованного списка.</LI>
  <LI>Я — второй пункт нумерованного списка.</LI>
  <LI>Я — третий пункт нумерованного списка.</LI>
</OL>
```

Списки можно помещать друг в друга, создавая *вложенные списки*. Делается это следующим образом. Сначала во "внешнем" списке (в который должен быть помещен вложенный) отыскивают пункт, после которого должен находиться вложенный список. Затем HTML-код, создающий вложенный список, помещают в разрыв между текстом этого пункта и его закрывающим тегом </LI>. Если же вложенный список должен помещаться в начале "внешнего" списка, его следует вставить между открывающим тегом <LI> первого пункта "внешнего" списка и его текстом. Что, впрочем, логично.

В листинге 2.5 представлен HTML-код, создающий два списка, один из которых вложен внутрь другого. Обратите внимание, где размещается HTML-код, создающий вложенный список.

Листинг 2.5

```
<UL>
  <LI>Я — первый пункт внешнего списка.</LI>
  <LI>Я — второй пункт внешнего списка.
    <UL>
      <LI>Я — первый пункт вложенного списка.</LI>
      <LI>Я — второй пункт вложенного списка.</LI>
      <LI>Я — третий пункт вложенного списка.</LI>
    </UL>
```

```
  </LI>
  <LI>Я — третий пункт внешнего списка.</LI>
</UL>
```

HTML позволяет вкладывать нумерованный список внутрь маркированного и наоборот. Глубина вложения списков не ограничена.

Еще HTML позволяет создать так называемый *список определений*, представляющий собой перечень терминов и их разъяснений. Такой список создают с помощью парного тега <DL>. Внутри него помещают пары "термин — разъяснение", причем термины заключают в парный тег <DT>, а разъяснения — в парный тег <DD> (листинг 2.6).

Листинг 2.6

```
<DL>
  <DT>Содержимое</DT>
  <DD>Информация, отображаемая на Web-странице</DD>
  <DT>Представление</DT>
  <DD>Набор правил, определяющих формат отображения содержимого</DD>
  <DT>Поведение</DT>
  <DD>Набор правил, определяющих реакцию Web-страницы или ее элементов на
  действия посетителя</DD>
</DL>
```

Осталось сказать, что списки и их пункты относятся к блочным элементам Web-страницы, и при их выводе на экран Web-обозреватель руководствуется теми же правилами, что и при выводе абзацев и заголовков.

На нашей Web-странице есть несколько абзацев, перечисляющих основные возможности HTML. Превратим их в маркированный список (листинг 2.7).

Листинг 2.7

```
<UL>
  <LI>абзацы;</LI>
  <LI>заголовки;</LI>
  <LI>цитаты;</LI>
  <LI>списки;</LI>
  <LI>таблицы;</LI>
  <LI>графические изображения;</LI>
  <LI>аудио- и видеоролики.</LI>
</UL>
```

Теперь наша Web-страничка стала выглядеть симпатичнее.

Цитаты

В тексте Web-страницы, которую мы создаем, присутствует большая цитата из Русской Википедии. Давайте ее как-то выделим.

HTML уже приготовил для нас выход из положения — парный тег <BLOCKQUOTE>, внутри которого размещается HTML-код, создающий цитату (листинг 2.8). Web-обозреватель выводит цитату с отступом слева.

Листинг 2.8

```
<BLOCKQUOTE>
  <P>Я — начало большой цитаты.</P>
  <P>Я — продолжение большой цитаты.</P>
</BLOCKQUOTE>
```

Как видим, в тег <BLOCKQUOTE> можно поместить несколько абзацев. Там также могут быть заголовки и списки (если уж возникнет такая потребность).

Большая цитата HTML также относится к блочным элементам.

Осталось только сделать то, что было задумано, — оформить цитату (листинг 2.9).

Листинг 2.9

```
<BLOCKQUOTE>
  <P>HTML (от англ. HyperText Markup Language — язык разметки
    гипертекста) — стандартный язык разметки документов во Всемирной
    паутине.</P>
</BLOCKQUOTE>
```

Текст фиксированного формата

Аппетит приходит во время еды. Мы еще не успели доделать свою первую Web-страницу, а уже хотим сделать еще одну. Давайте же ее сделаем. Дадим аппетиту разгуляться!

Новая Web-страница (листинг 2.10) будет посвящена тегу <TITLE>.

Листинг 2.10

```
<!DOCTYPE html>
<HTML>
  <HEAD>
    <META HTTP-EQUIV="Content-Type" CONTENT="text/html; charset=utf-8">
    <TITLE>Тег TITLE</TITLE>
  </HEAD>
```

```
  <BODY>
    <H1>Тег TITLE</H1>
    <P>Тег TITLE служит для указания названия Web-страницы. Он ставится в
      ее секции заголовка.</P>
    <H6>Пример:</H6>
    <P>!HEAD!
      !TITLE!Я — заголовок Web-страницы!/TITLE!
    !HEAD!</P>
  </BODY>
</HTML>
```

Здесь мы поместили краткое описание тега <TITLE> и код примера, имеющий такой вид:

```
!HEAD!
  !TITLE!Я — заголовок Web-страницы!/TITLE!
!/HEAD!
```

Вместо символов < и >, которые, как мы помним из *главы 1*, недопустимы в обычном тексте, мы поставили восклицательные знаки. В *главе 3* мы узнаем, как все-таки вставить в текст недопустимые символы, и заменим их.

Сохраним набранный код в файле с именем t_title.htm и откроем его в Web-обозревателе. И что мы там увидим?

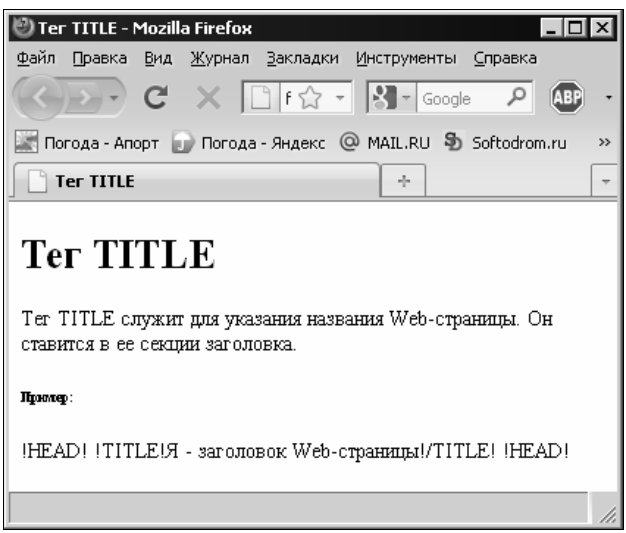


Рис. 2.1. Web-страница t_title.htm в Web-обозревателе. Обратим внимание на код примера

Как видно на рис. 2.1, Web-обозреватель вывел код примера в одну строку и без всяких отступов. Но ведь мы разбили его на три строки и создали отступы с помощью пробелов, чтобы HTML-код лучше читался и сразу была видна вложенность тегов! Что случилось?

Вспомним два правила, которыми руководствуется Web-обозреватель при выводе текста блочного элемента и которые были перечислены в разделе, посвященном

абзацам. Эти правила гласят, что два или более следующих друг за другом пробела или переноса строки преобразуются в один пробел и перенос строки считается за пробел. Так Web-обозреватель и поступил: преобразовал переносы строки в пробелы, а последовательные пробелы — в один пробел. И вывел код примера в виде обычного абзаца, который у него поместился в одну строку.

Web-обозреватель все сделал правильно! Просто мы не указали ему, как следует выводить данный текст.

Специально для случаев, когда текст должен быть выведен именно так, как набран, с сохранением всех пробелов и переносов строк, язык HTML предусматривает парный тег <PRE>. Выводимый текст помещают внутри этого тега (листинг 2.11).

Листинг 2.11

```
<PRE>Я — текст,
который будет выведен
на Web-страницу
со всеми
    отступами
и
переносами
строк.</PRE>
```

Такой текст называется текстом *фиксированного формата*.

Правила отображения текста фиксированного формата:

- для вывода используется моноширинный шрифт (у *моноширинного шрифта* все символы имеют одинаковую ширину, в отличие от *пропорциональных*, у которых ширина символов различна);
- все пробелы и переносы строк сохраняются при выводе (это мы уже знаем);
- если строка текста фиксированного формата не помещается в окне Web-обозревателя по ширине, она ни в коем случае не будет переноситься. Из-за этого может возникнуть потребность в горизонтальной прокрутке Web-страницы. (что, вообще-то, плохой стиль Web-дизайна...);
- возможно наличие HTML-тегов для выделения текста и гиперссылок (подробнее о них будет рассказано в *главах 3* и *5*).

Текст фиксированного формата также является блочным элементом.

Исправим HTML-код Web-страницы t_title.htm так, чтобы пример выводился в виде текста фиксированного формата (листинг 2.12).

Листинг 2.12

```
<!DOCTYPE html>
<HTML>
    . . .
```

```
    <H6>Пример:</H6>
    <PRE>!HEAD!
  !TITLE!Я — заголовок Web-страницы!/TITLE!
!HEAD!</PRE>
  </BODY>
</HTML>
```

Откроем исправленную Web-страницу в Web-обозревателе и убедимся, что он выводится правильно (рис. 2.2).

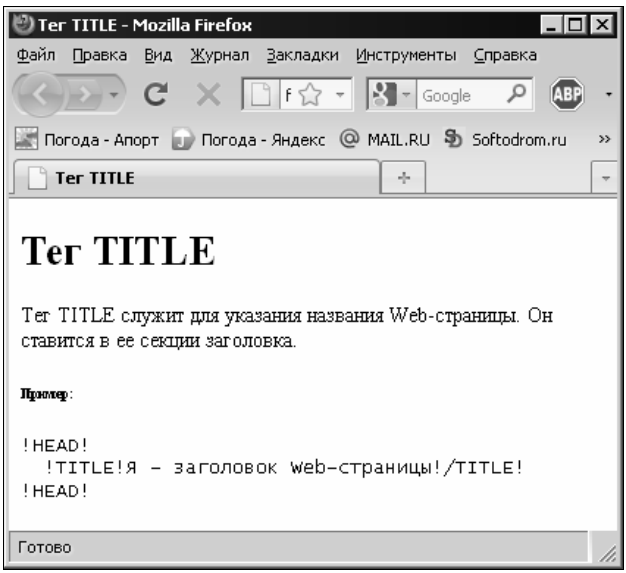


Рис. 2.2. Исправленная Web-страница t_title.htm в Web-обозревателе. Код примера оформлен как текст фиксированного формата

Как правило, текст фиксированного формата используется для вывода исходных текстов программ и быстрой публикации в Сети документов, набранных обычным текстом. В качестве примера можно вспомнить библиотеку Мошкова (**http://www.lib.ru**), в которой все книги опубликованы как раз в виде текста фиксированного формата.

Горизонтальные линии

Что бы нам еще такое сделать с Web-страницами... Давайте дополнительно выделим на главной Web-странице index.htm цитату из Википедии, описывающую HTML. Только как?

Для любителей все выделять HTML припас подарок — горизонтальную линию, создаваемую с помощью одинарного тега <HR>:

```
<P>Я буду отделен от следующего абзаца горизонтальной линией.</P>
<HR>
<P>Я отделен от предыдущего абзаца горизонтальной линией.</P>
```

Горизонтальная линия HTML растягивается по горизонтали на всю ширину Web-страницы, имеет один-два пиксела в толщину и выпуклый или вдавленный вид (конкретные параметры зависят от Web-обозревателя). Она применяется для отделения одной части содержимого Web-страницы от другой и просто "для красоты". Однако нужно помнить, что слишком большое число горизонтальных линий — дурной тон Web-дизайна.

Больше о горизонтальных линиях рассказывать нечего. Так что внесем в HTML-код страницы index.htm необходимые исправления (листинг 2.13).

Листинг 2.13

```
<P>Приветствуем на нашем Web-сайте всех, кто занимается Web-дизайном!
  Здесь вы сможете найти информацию обо всех интернет-технологиях,
  применяемых при создании Web-страниц. А именно, о языках HTML и
  CSS.</P>
<HR>
<P>Русская Википедия определяет термин HTML так:</P>
. . .
<P>Пожалуй, ни убавить ни прибавить...</P>
<HR>
<P>HTML позволяет формировать на Web-страницах следующие элементы:</P>
```

Результат показан на рис. 2.3. Симпатично вышло, не так ли?

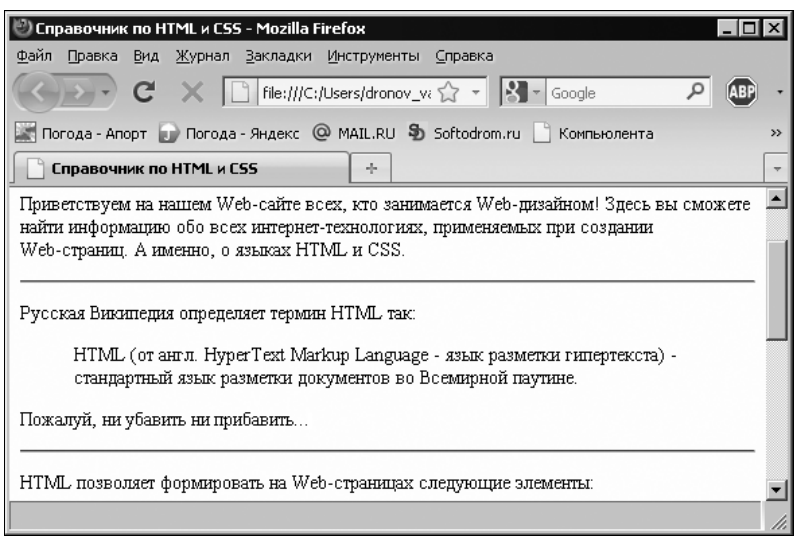


Рис. 2.3. Web-страница index.htm с горизонтальными линиями

Адреса

Часто на Web-страницах указывают контактные данные их создателей (почтовые и электронные адреса, телефоны, факсы и пр.). Для этого HTML предусматривает

особый тег <ADDRESS>. Он ведет себя так же, как тег абзаца <P>, но его содержимое выводится курсивом:

```
<ADDRESS>Я — адрес создателя данной Web-страницы: почтовый, электронный,
телефоны и факсы.</ADDRESS>
```

Комментарии

Напоследок рассмотрим одну очень важную возможность HTML, которая, хоть и не касается напрямую Web-дизайна, но сильно поможет забывчивым Web-дизайнерам.

Комментарий — это фрагмент HTML-кода, который не выводится на Web-страницу и вообще не обрабатывается Web-обозревателем. Он служит для того, чтобы Web-дизайнер смог оставить текстовые заметки для себя или своих коллег.

Текст комментария помещают между открывающим тегом <!-- и закрывающим тегом --> и обязательно отделяют от этих тегов пробелами. Как видим, теги комментария не укладываются в основное правило HTML: закрывающий тег должен иметь то же имя, что и открывающий. Открывающий и закрывающий теги комментария — разные!

Пример:

```
<!-- Я — комментарий. Меня не видно на Web-странице. -->
```

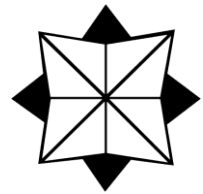
Мы можем создать в HTML-коде наших Web-страниц комментарии, указывающие, что нам следует доработать. Хотя бы просто для практики:

```
<!-- Выделить важные фрагменты текста и доделать код примеров. -->
<!-- Создать Web-страницы, посвященные остальным тегам HTML. -->
```

Что дальше?

В этой главе мы научились разделять текст Web-страницы на логические "куски" — абзацы, создавать заголовки, списки, цитаты, текст фиксированного формата и горизонтальные линии. В общем, структурировать текст для его удобного чтения.

В следующей главе мы научимся выделять фрагменты абзацев и заголовков, чтобы привлечь к ним внимание посетителей. HTML предоставляет для этого много новых тегов, которые мы постараемся запомнить. А еще мы узнаем, как вставить в текст недопустимые символы, в частности, < и >.



ГЛАВА 3

Оформление текста

В предыдущей главе мы научились структурировать текст Web-страницы, разбивая его на логические "куски". Также мы узнали о комментариях HTML, которые позволяют создавать небольшие заметки прямо в коде Web-страницы. И создали две Web-страницы нашего первого Web-сайта.

В этой главе мы продолжим работать с текстом. Мы научимся оформлять его, выделяя отдельные фрагменты текста, чтобы привлечь к ним внимание посетителя. А еще мы научимся вставлять в текст недопустимые символы.

Выделение фрагментов текста

Собственно, средства HTML для оформления текста мы начали изучать еще в *главе 1*. Это парные теги `<STRONG>` и `<EM>`, которые выделяют свое содержимое полужирным и курсивным шрифтом соответственно.

Однако на самом деле теги `<STRONG>` и `<EM>` — это нечто большее, чем просто выделение текста. Они дают фрагменту текста, являющемуся их содержимым, особое значение с точки зрения Web-обозревателя. Они говорят, что данный фрагмент текста является важным, и на него следует обратить внимание посетителя. Тег `<STRONG>` делает фрагмент текста очень важным, а тег `<EM>` — менее важным (листинг 3.1).

Листинг 3.1

```
<P><STRONG>Я — очень важный текст и поэтому набран полужирным
шрифтом!</STRONG> Прочитайте меня в первую очередь!</P>
<P><EM>А я — менее важный текст и набран курсивом. </EM> Не забудьте
прочитать меня, но можете сделать это потом.</P>
```

HTML предусматривает для выделения текста довольно много тегов (табл. 3.1), имеющих две особенности:

- все они парные;
- служат для выделения частей текста блочных элементов (абзацев, заголовков, пунктов списков, текста фиксированного форматирования; подробнее о блочных элементах см. в *главе 2*).

Таблица 3.1. *Теги HTML, предназначенные для выделения фрагментов текста*

Тег	Назначение	Отображение Web-обозревателем
<ABBR>	Аббревиатура	Подчеркнутым
<ACRONYM>	Аббревиатура. Фактически то же самое, что и тег <ABBR>	Подчеркнутым
<CITE>	Небольшая цитата	Курсивом
<CODE>	Фрагмент исходного кода программы	Моноширинным шрифтом
<DEL>	Текст, удаленный из Web-страницы	Зачеркнутым
<DFN>	Новый термин	Курсивом
<EM>	Менее важный текст	Курсивом
<INS>	Текст, вновь помещенный на Web-страницу	Подчеркнутым
<KBD>	Данные, вводимые пользователем в какую-либо программу	Моноширинным шрифтом
<Q>	Небольшая цитата. Фактически то же самое, что и тег <CITE>	Обычным шрифтом
<SAMP>	Данные, выводимые какой-либо программой	Моноширинным шрифтом
<STRONG>	Очень важный текст	Полужирным шрифтом
<TT>	Данные, выводимые какой-либо программой. Фактически то же самое, что и тег <SAMP>	Моноширинным шрифтом
<VAR>	Имя переменной в исходном коде программы	Курсивом

Как уже говорилось ранее, все эти теги служат для выделения фрагментов текста, являющихся частью блочных элементов, скажем, абзацев (листинг 3.2). Элементы Web-страницы, которые они создают, не являются независимыми и не отображаются отдельно от их "соседей", а принадлежат другим элементам — блочным. Такие элементы Web-страницы называются *встроенными*.

Листинг 3.2

```
<P>Теги HTML служат для создания элементов Web-страниц.
<STRONG>Соблюдайте порядок вложенности тегов!</STRONG><P>
<P>Тег <CODE>P</CODE> служит для создания <DFN>абзаца</DFN> HTML.</P>
<P>Язык <ABBR>HTML</ABBR> служит для создания <INS>содержимого</INS>
Web-страниц.</P>
<P>Наберите в Web-обозревателе интернет-адрес
<KBD>http://www.w3.org</KBD>.<P>
```

Из всех рассмотренных нами тегов чаще всего встречаются <STRONG> и <EM>. Остальные теги так и не снискали большой популярности среди Web-дизайнеров.

Для практики давайте откроем Web-страницу index.htm и выделим некоторые фрагменты ее текста с помощью тегов, перечисленных в табл. 3.1 (листинг 3.3).

Листинг 3.3

```
<P>Приветствуем на нашем Web-сайте всех, кто занимается Web-дизайном!
Здесь вы сможете найти информацию обо всех интернет-технологиях,
применяемых при создании Web-страниц. А именно, о языках <DFN>HTML</DFN>
и <DFN>CSS</DFN>.</P>
. . .
<P><CODE>!DOCTYPE</CODE>, <CODE>BODY</CODE>, <CODE>EM</CODE>,
<CODE>HEAD</CODE>, <CODE>HTML</CODE>, <CODE>META</CODE>, <CODE>P</CODE>,
<CODE>STRONG</CODE>, <CODE>TITLE</CODE></P>
```

Все эти фрагменты так и просятся: оформите нас надлежащим образом! Мы ведь особенные!

Разрыв строк

Мы совсем забыли поместить на Web-страницы сведения об авторских правах их разработчика, т. е. о нас. Давайте сделаем это. Поместим их в самый низ Web-страниц посредством изученного в *главе 2* тега <ADDRESS>:

```
<ADDRESS>Все права защищены. Читатели, 2010 год.</ADDRESS>
```

Все хорошо, за одним исключением. Обычно предупреждение о том, что авторские права защищены, и имя или название разработчика разносят на разные строки. Но как нам это сделать?

Можно, конечно, использовать два тега <ADDRESS>: один — для предупреждения, а другой — для имени разработчика. Но тогда строки будут разделены довольно большим расстоянием. (Вспомним — тег <ADDRESS> ведет себя как абзац, т. е. отделяется от соседних абзацев отступом.) А это будет выглядеть некрасиво.

Выход — добавить *разрыв строк* HTML. Он выполняет безусловный перевод строки, в которой он присутствует, в том месте, где проставлен. Разрыв строки определяется одиночным тегом <BR>:

```
<P>Этот абзац будет разорван на две строки в этом<BR>месте.</P>
```

Разрыв строки также относится к встроенным элементам Web-страницы.

Давайте вставим разрыв строки в текст сведения об авторских правах, между точкой в первом предложении и началом второго предложения. Пробел между ними можно убрать — он там совершенно не нужен:

```
<ADDRESS>Все права защищены.<BR>Читатели, 2010 год.</ADDRESS>
```

Откроем исправленную Web-страницу в Web-обозревателе и посмотрим на результат (рис. 3.1). Видно, что текст сведений об авторских правах разделен на строки в том самом месте, куда мы вставили тег <BR>.

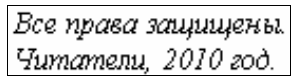


Рис. 3.1. Абзац с разрывом строк

Разрыв строк следует применять экономно, только в тех случаях, когда нужно разделить один абзац на несколько логически связанных частей. Ранее неопытные Web-дизайнеры с его помощью нередко разбивали текст на абзацы, но сейчас это дурной тон Web-дизайна.

Вставка недопустимых символов. Литералы

Давайте откроем Web-страницу t_title.htm и посмотрим на код приведенного там примера использования тега `<TITLE>`. Чего там не хватает? Правильно — символов < и >, с помощью которых и создается тег HTML. Эти символы являются недопустимыми и не должны встречаться в обычном тексте. Мы заменили их восклицательными знаками, но код стал от этого выглядеть просто ужасно.

Так есть ли способ все-таки поместить в обычный текст недопустимые символы? Есть, и весьма изящный.

Создатели HTML решили, что, если уж напрямую эти символы вставить в текст нельзя, значит, их нужно заменить на особую последовательность символов, называемую *литералом*. Встретив литерал, Web-обозреватель "поймет", что здесь должен присутствовать соответствующий недопустимый символ, и выведет его на Web-страницу.

Литералы HTML начинаются с символа `&` и заканчиваются символом `;` (точка с запятой). Между ними помещается определенная последовательность букв. Так, символ < определяется литералом `<`, а символ > — литералом `>`.

Сразу же исправим код примера (листинг 3.4).

Листинг 3.4

```
<!DOCTYPE html>
<HTML>
  . . .
  <BODY>
    . . .
    <H6>Пример:</H6>
    <PRE>&lt;HEAD&gt;
  &lt;TITLE&gt;Я — заголовок Web-страницы&lt;/TITLE&gt;
&lt;HEAD&gt;</PRE>
    . . .
  </BODY>
</HTML>
```

Откроем исправленную Web-страницу в Web-обозревателе. Вот теперь теги в примере отображаются со всеми положенными символами < и >!

Литералов в HTML довольно много. Самые часто применяемые из них перечислены в табл. 3.2.

Таблица 3.2. *Некоторые литералы языка HTML*

Недопустимый символ	Литерал HTML
— (длинное тире)	`&mdash;`
– (короткое тире)	`&ndash;`
"	`&quot;`
&	`&`
<	`<`
>	`>`
©	`&copy;`
®	`&reg;`
Левая двойная кавычка	`&ldquo;`
Левая угловая кавычка	`&laquo;`
Левый апостроф	`&lsquo;`
Многоточие	`&hellip;`
Неразрывный пробел	` `
Правая двойная кавычка	`&rdquo;`
Правая угловая кавычка	`&raquo;`
Правый апостроф	`&rsquo;`
Символ евро	`&euro;`

Среди перечисленных в табл. 3.2 литералов и обозначаемых ими недопустимых символов особенно выделяется один. Это *неразрывный пробел*, обозначаемый литералом ` `. По этому пробелу Web-обозреватель никогда не будет выполнять перенос строк.

Неразрывный пробел необходим, если в каком-то месте предложения перенос строк никогда не должен выполняться. Так, правила правописания русского языка не допускают перенос строк перед длинным тире. Поэтому крайне рекомендуется отделять длинное тире от предыдущего слова неразрывным пробелом:

```
<P>Неразрывный пробел &mdash; очень важный литерал.<P>
```

Здесь литерал ` ` создает неразрывный пробел, а литерал `&mdash;` — длинное тире.

Кстати, если уж на то пошло, мы можем в сведениях об авторских правах вставить символ ©. Вот так:

```
<ADDRESS>Все права защищены.<BR>&copy; читатели, 2010 год.</ADDRESS>
```

HTML также позволяет вставить в текст любой символ, поддерживаемый кодировкой Unicode, просто указав его код. Для этого предусмотрен литерал вида `&#<десятичный код символа>;`.

Но как узнать код нужного символа? Очень просто. В этом нам поможет утилита Таблица символов, поставляемая в составе Windows. Давайте запустим ее и посмотрим на ее окно (рис. 3.2).

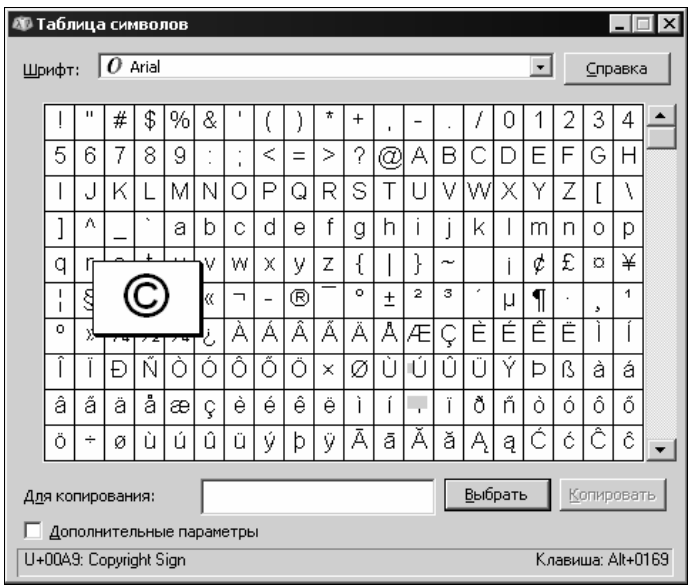


Рис. 3.2. Окно утилиты Таблица символов (выбран символ ©)

В большом списке символов, занимающем почти все окно этой утилиты, выберем нужный нам символ. После этого посмотрим на строку статуса, расположенную вдоль нижнего края окна. В правой ее части находится надпись вида **Клавиша: Alt+*<десятичный код символа>***. Этот-то код нам и нужен!

> ***Внимание!***
>
> Надпись **Клавиша: Alt+*<десятичный код символа>*** появляется в строке статуса окна **Таблица символов** только при выборе символов, которые нельзя ввести непосредственно с клавиатуры.

Так, мы можем вставить в сведения об авторских правах символ ©, использовав литерал `&#0169;`, где `0169` — десятичный код данного символа (см. рис. 3.2):

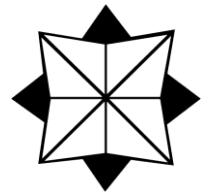
```
<ADDRESS>Все права защищены.<BR>&#0169; читатели, 2010 год.</ADDRESS>
```

Вот и все о литералах HTML. Да и об оформлении текстов тоже.

Что дальше?

В этой главе мы продолжили работу с текстом. Мы научились выделять его фрагменты, придавая им особое значение и привлекая к ним внимание посетителя. Также мы научились разрывать строки абзацев и вставлять в текст недопустимые символы, используя литералы HTML. Все это нам пригодится в дальнейшем.

Фактически мы рассмотрели все средства HTML для работы с текстом. В следующей главе мы займемся графикой и мультимедиа. Мы научимся помещать на Web-страницы изображения, аудио и видеоролики. И помогут нам в этом новые возможности HTML 5.



ГЛАВА 4

Графика и мультимедиа

В предыдущих главах мы работали с текстом. Но не текстом единым жива WWW. Web-страницы могут содержать также графику и мультимедийные данные (аудио- и видеоролики). Умело примененные, они способны значительно оживить Web-сайт. И это не говоря уже о случаях, когда без графики и мультимедиа просто не обойтись. В самом деле, Web-сайт, посвященный музыке, обязан содержать на Web-страницах музыкальные композиции, иначе грош ему цена!

В этой главе мы будем работать с графическими изображениями и мультимедийными данными. И наконец-то начнем изучение новых возможностей HTML 5, которые значительно упростят нам работу.

Но сначала нам нужно рассмотреть один очень важный вопрос и выучить новый термин.

Внедренные элементы Web-страниц

Казалось бы, что может быть проще: описывай прямо в HTML-коде графическое изображение, аудио- или видеоклип — и все в ажуре! Но не тут-то было... Все дело в том, что графика и мультимедийные данные имеют принципиально другую природу, нежели текст. Из-за этого объединить их в одном файле невозможно.

Разработчики HTML нашли оригинальный выход из положения. Прежде всего, графические изображения и мультимедийные данные хранятся в отдельных файлах. А в HTML-коде Web-страниц с помощью особых тегов прописывают ссылки на эти файлы, фактически — их интернет-адреса. Встретив такой тег-ссылку, Web-обозреватель запрашивает у Web-сервера соответствующий файл с изображением, аудио- или видеороликом и выводит его на Web-страницу в то место, где встретился данный тег.

Графические изображения, аудио- и видеоролики и вообще любые данные, хранящиеся в отдельных от Web-страницы файлах, называются *внедренными* элементами Web-страниц.

> ***НА ЗАМЕТКУ***
>
> В *главе 1* говорилось, что Web-страница может храниться в нескольких файлах. Web-страница с внедренными элементами — тому пример.

Графика

Графика на Web-страницах появилась достаточно давно. Предназначенный для этого тег появился еще в версии 3.2 языка HTML, которая вышла в 1997 году. С тех пор Всемирную паутину захлестнула волна интернет-графики (к настоящему времени, надо сказать, поутихшая).

Как уже говорилось, графические изображения — суть внедренные элементы Web-страниц. Это значит, что они сохраняются в отдельных от самой Web-страницы файлах.

Форматы интернет-графики

На данный момент существует несколько десятков форматов хранения графики в файлах. Но Web-обозреватели поддерживают далеко не все. В WWW сейчас используются всего три формата, которые мы далее рассмотрим.

Нужно отметить, что все три формата поддерживают сжатие графической информации. Благодаря сжатию размер графического файла сильно уменьшается, и поэтому он передается по сети быстрее, чем несжатый файл.

Формат *GIF* (Graphics Interchange Format, формат обмена графикой) — старожил среди "сетевых" форматов графики. Он был разработан еще в 1987 году и модернизирован в 1989 году. На данный момент он считается устаревшим, но все еще широко распространен.

Достоинств у него довольно много. Во-первых, GIF позволяет задать для изображения "прозрачный" цвет; закрашенные этим цветом области изображения станут своего рода "дырками", сквозь которые будет "просвечивать" фон родительского элемента. Во-вторых, в одном GIF-файле можно хранить сразу несколько изображений, фактически — настоящий фильм *(анимированный GIF)*. В-третьих, из-за особенностей применяемого в нем сжатия он отлично подходит для хранения штриховых изображений (карт, схем, рисунков карандашом и пр.).

Недостаток у формата GIF всего один — он совершенно не годится для хранения полутоновых изображений (фотографий, картин и т. п.). Это обусловлено тем, что GIF-изображения могут включать всего 256 цветов, и потерями качества при сжатии.

GIF используется для хранения элементов оформления Web-страниц (всяческих линеек, маркеров списков и т. п.) и штриховых изображений.

Формат *JPEG* (Joint Photographic Experts Group, Объединенная группа экспертов по фотографии) был разработан в 1993 году специально для хранения полутоновых изображений. Для чего активно применяется до сих пор.

JPEG, в отличие от GIF, не ограничивает количество цветов у изображения, а реализованное в нем сжатие лучше всего подходит для полутоновых изображений. Однако он плохо справляется с штриховой графикой, не поддерживает "прозрачный" цвет и анимацию.

Формат *PNG* (Portable Network Graphics, перемещаемая сетевая графика) появился на свет в 1996 году. Он разрабатывался как замена устаревшему и не очень удобному GIF, а также, в некоторой степени, JPEG. В настоящее время он последовательно отвоевывает "жизненное пространство" у GIF.

К достоинствам формата PNG можно отнести возможность хранения как штриховых, так и полутоновых изображений и поддержку полупрозрачности. Недостаток всего один и некритичный — невозможность хранения анимации.

Осталось назвать расширения, под которыми сохраняются файлы того или иного формата. Файлы GIF и PNG имеют "говорящие" расширения gif и png, а файлы JPEG — jpg, jpe или jpeg.

Вставка графических изображений

Добавить на Web-страницу графическое изображение позволяет одинарный тег `<IMG>`. Web-обозреватель поместит изображение в том месте Web-страницы, в котором встретился тег `<IMG>`.

В *главе 1* мы говорили об атрибутах тегов HTML, после чего надолго о них забыли. Сейчас настала пора о них вспомнить, т. к. сила тега `<IMG>` — в его атрибутах.

Обязательный атрибут тега `SRC` служит для указания интернет-адреса файла с изображением.

Пример:

```
<IMG SRC="image.gif">
```

Этот тег помещает на Web-страницу изображение, хранящееся в файле image.gif, который находится в той же папке, что и файл самой этой Web-страницы.

Пример:

```
<IMG SRC="/images/picture.jpg">
```

Данный тег помещает на Web-страницу изображение, хранящееся в файле picture.jpg, который находится в папке images, вложенной в корневую папку Web-сайта.

Пример:

```
<IMG SRC="http://www.othersite.ru/book12/author.jpg">
```

А этот тег помещает на Web-страницу изображение, хранящееся в файле с интернет-адресом **http://www.othersite.ru/book12/author.jpg**, т. е. изображение с другого Web-сайта.

> ***НА ЗАМЕТКУ***
>
> Принципы формирования интернет-адресов файлов, применяемые в WWW, мы подробно рассмотрим в *главе 6*.

Мы уже знаем о том, что элементы Web-страницы могут быть блочными и встроенными. Так вот, изображение, помещенное на Web-страницу с помощью тега

<IMG>, — встроенный элемент. Это значит, что он не может "гулять сам по себе", а должен быть частью блочного элемента, например, абзаца.

Из этого следуют два важных вывода.

Во-первых, мы можем вставить графическое изображение прямо в абзац:

```
<P>Посмотрите картинку — <IMG SRC="image.gif"></P>
```

Во-вторых, если нам понадобится отобразить на Web-странице отдельное, не связанное ни с каким абзацем графическое изображение, нам придется поместить его в специально созданный абзац:

```
<P><IMG SRC="/images/picture.jpg"></P>
```

Настала пора попрактиковаться. Найдем подходящий графический файл и поместим его в папку, где хранятся файлы нашего Web-сайта. Автор выбрал изображение значка @, хранящееся в файле image.gif. Разумеется, вы можете выбрать любой другой файл, но в этом случае не забудьте указать его имя в HTML-коде листинга 4.1.

Впишем в раздел тегов Web-страницы index.htm тег <IMG> и создадим описывающую его Web-страницу. Это будет третья Web-страница нашего Web-сайта. Ее HTML-код приведен в листинге 4.1.

Листинг 4.1

```
<!DOCTYPE html>
<HTML>
  <HEAD>
    <META HTTP-EQUIV="Content-Type" CONTENT="text/html; charset=utf-8">
    <TITLE>Тег IMG</TITLE>
  </HEAD>
  <BODY>
    <H1>Тег IMG</H1>
    <P>Тег IMG служит для вставки на Web-страницу графического
    изображения.</P>
    <H6>Пример:</H6>
    <PRE>&lt;P&gt;&lt;IMG SRC=&quot;image.gif&quot;&gt;&lt;/P&gt;</PRE>
    <H6>Результат:</H6>
    <P><IMG SRC="image.gif"></P>
  </BODY>
</HTML>
```

Здесь предполагается, что графический файл, содержимое которого мы будем выводить на Web-страницу, имеет имя image.gif. Если у вас другое имя файла, соответственно исправьте HTML-код.

Сохраним новую Web-страницу в файле t_img.htm и сразу же откроем в Web-обозревателе (рис. 4.1). На этой Web-странице мы увидим код примера вида

```
<P><IMG SRC="image.gif"></P>
```

а чуть ниже — результат его выполнения.

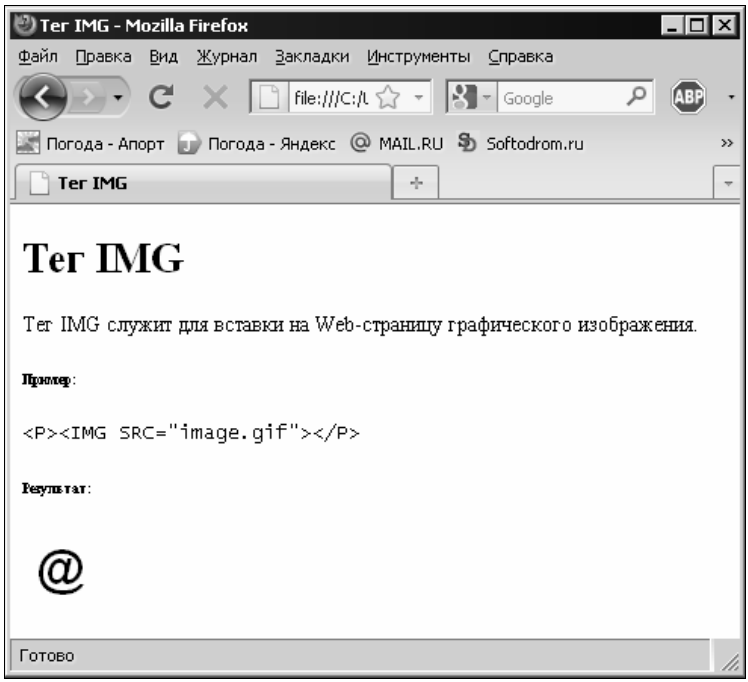


Рис. 4.1. Web-страница t_img.htm в Web-обозревателе

Как видим, ничего сложного в размещении изображения на Web-странице нет. Нужно только приготовить графический файл и вставить в HTML-код один простой тег.

А теперь рассмотрим еще один атрибут тега `<IMG>`, который может нам пригодиться в дальнейшем.

Поскольку изображение хранится в отдельном от Web-страницы файле, Web-обозревателю придется послать Web-серверу еще один запрос на его получение. Web-серверу нужно найти этот файл и отправить его Web-обозревателю. Файл должен загрузиться по сети.

На все это требуется время. Если изображений на Web-странице много, все они велики по размеру, а канал связи медленный, понадобится значительное время. Может случиться так, что сама Web-страница будет успешно загружена и отображена на экране, а изображения — еще нет. И Web-обозреватель вместо не загруженного еще изображения выведет на экран пустой прямоугольник.

Возникают две проблемы. Во-первых, пустые прямоугольники вместо изображений выглядят некрасиво. Во-вторых, посетитель не сможет понять, что за изображение должно находиться вместо того или иного прямоугольника, и стоит ли ждать окончания его загрузки.

И если с первой проблемой справиться практически невозможно, то вторую мы вполне способны решить. Для этого тег `<IMG>` поддерживает необязательный атрибут `ALT`, с помощью которого указывается так называемый *текст замены*. Он будет

отображаться в пустом прямоугольнике, обозначающем незагруженное изображение, пока это изображение не загрузится:

```
<P><IMG SRC="image.gif" ALT="Пример изображения"></P>
```

Здесь мы задали для изображения с Web-страницы t_img.htm текст замены "Пример изображения".

> ***На заметку***
>
> Хорошим тоном Web-дизайна считается указание текста замены только у значащих изображений. У изображений, являющихся элементами оформления Web-страницы, текст замены обычно не указывают.

На этом пока все об интернет-графике. Мы еще вернемся к ней в *главе 6*, когда будем рассматривать изображения-гиперссылки и карты-изображения. А сейчас пора начать разговор о...

Мультимедиа

Мультимедиа — это, в первую очередь, аудио и видео. Мультимедиа в приложении к Web-дизайну — это аудио- и видеоролики, размещенные на Web-страницах.

До недавних пор разместить на Web-странице аудио- или видеоролик можно было только с помощью громадного HTML-кода, дополнительных программ и "шаманских плясок" вокруг всего этого. Но сейчас, с появлением HTML 5 и поддерживающих его (хотя бы частично) Web-обозревателей, потребуется всего один тег. Какой? Очень простой, не сложнее уже знакомого нам тега <IMG>!

> ***Внимание!***
>
> В этой книге будет рассматриваться работа с мультимедиа исключительно средствами HTML 5. Устаревшие способы (в частности, тег <OBJECT>) не описаны.

Форматы файлов и форматы кодирования

Форматов мультимедийных файлов существует не меньше, чем форматов файлов графических. Как и в случае с интернет-графикой, Web-обозреватели поддерживают далеко не все мультимедийные форматы, а только немногие. (Хотелось бы автору посмотреть на Web-обозреватель, который поддерживает все форматы файлов — и на сам Web-обозреватель, и на его размеры...)

Но Web-обозревателю мало поддерживать только сам формат мультимедийных файлов. Он должен быть "знаком" и с форматом кодирования записанной в нем аудио- и (или) видеоинформации. В мире мультимедиа так — разные файлы одного формата могут хранить информацию, закодированную разными форматами. Более того, аудио- и видеодорожки мультимедийного файла практически всегда кодируются разными форматами.

Практически все форматы кодирования мультимедийных данных поддерживают их сжатие. Благодаря этому размер мультимедийных файлов значительно (иногда на

несколько порядков) уменьшается, что благотворно сказывается на скорости их передачи по сети.

Перечислим и кратко опишем все форматы мультимедийных файлов, используемые в Web-дизайне и поддерживаемые Web-обозревателями.

- *WAV* (WAVe, волна) — "старожил" среди мультимедийных форматов. Был разработан Microsoft в самом начале 90-х годов прошлого века для хранения аудиоданных и применяется для этой цели до сих пор. Файлы такого формата имеют расширение wav.
- *OGG* — более новый формат. Был разработан около десяти лет назад некоммерческой организацией Xiph.org для хранения аудио- и видеоинформации. Файлы этого формата имеют расширения ogg (универсальное расширение), oga (аудиофайлы) и ogv (видеофайлы); последние два расширения встречаются редко.
- *MP4* — также "новичок". Был разработан организацией Motion Picture Expert Group (Экспертная группа по вопросам движущегося изображения; также известна как MPEG) в 1998 году для хранения аудио- и видеоданных. Файлы этого формата имеют расширение mp4.
- *QuickTime* — формат очень старый, он старше даже WAV. Был разработан Apple в 1989 году для хранения аудио- и видеоданных. Файлы такого формата имеют расширение mov.

Теперь рассмотрим форматы кодирования аудио и видео, поддерживаемые современными Web-обозревателями.

- *PCM* (Pulse-Coded Modulation, импульсно-кодовая модуляция) — самый простой и самый старый формат кодирования. Он даже не поддерживает сжатие информации. Служит для кодирования аудиоданных.
- *Vorbis* — более современный формат кодирования. Был представлен организацией Xiph.org (разработчиком формата файла OGG) в 2002 году. Используется для кодирования аудиоданных.
- *AAC* (Advanced Audio Coding, развитое кодирование аудио) — не очень новый формат кодирования. Был разработан организацией Motion Picture Expert Group в 1997 году. Применяется для кодирования аудиоданных.
- *Theora* — пожалуй, самый "молодой" формат кодирования. Он также бы разработан организацией Xiph.org несколько лет назад. Используется для кодирования видеоданных.
- *H*.264 — тоже очень "молод". Был представлен организациями Motion Picture Expert Group и Video Coding Experts Group (Группа экспертов по кодированию видео) в 2003 году. Предназначен для кодирования видеоданных.

Почти все эти форматы являются открытыми. Исключения — формат файлов QuickTime, принадлежащий Apple, и формат кодирования H.264, защищенный более чем сотней патентов.

Осталось выяснить, какие сочетания форматов файлов и форматов кодирования используются в Web-дизайне и какие Web-обозреватели их поддерживают. По-

рывшись в Интернете и немного поэкспериментировав, автор свел эти данные в табл. 4.1.

Таблица 4.1. *Сочетания формата мультимедийных файлов и форматов кодирования аудио и видео, используемые в Web-дизайне, и поддержка их современными Web-обозревателями*

Аудио	Видео	Firefox	Opera	Safari	Chrome
Файлы, содержащие только аудио					
WAV-PCM		*	*		*
OGG-Vorbis		*	*		*
MOV-AAC				*	
Файлы, содержащие аудио и видео					
OGG-Vorbis	OGG-Theora	*	*		*
MOV-AAC	MOV-H.264			*	
MP4-AAC	MP4-H.264			*	

Как видим, разные Web-обозреватели поддерживают различные форматы. Из-за этого у нас как у Web-дизайнеров могут быть проблемы...

Типы MIME

По сети передаются самые разные данные: Web-страницы, графические изображения, аудио- и видеофайлы, архивы, исполняемые файлы и пр. Эти данные предназначены разным программам. К тому же, с разными данными программа, принявшая их, может поступить по-разному. Так, Web-обозреватель при получении Web-страницы или графического изображения отобразит их на экране, а при получении архива или исполняемого файла — откроет или сохранит его на диске.

Всем передаваемым по сети данным присваивается особое обозначение, однозначно указывающее на их природу, — тип *MIME* (Multipurpose Internet Mail Extensions, многоцелевые расширения почты Интернета). Тип MIME присваивает данным программа, их отправляющая, например, Web-сервер. А принимающая программа (тот же Web-обозреватель) по типу MIME принятых данных определяет, поддерживает ли она эти данные, и, если поддерживает, что с ними делать.

Web-страница имеет тип MIME `text/html`. Графическое изображение формата GIF имеет тип MIME `image/gif`. Тип MIME исполняемого файла — `application/x-msdownload`, а архива ZIP — `application/x-zip-compressed`. Свои типы MIME имеют и мультимедийные файлы.

Вот о мультимедийных файлах и их типах MIME мы и поговорим.

Ранее было сказано, что современные Web-обозреватели работают с очень ограниченным набором форматов мультимедийных файлов из нескольких десятков существующих. Более того, разные Web-обозреватели поддерживают различные форма-

ты. Поэтому Web-обозреватель должен определить, поддерживает ли он формат полученного файла, т. е. стоит ли его вообще загружать. Как это сделать, мы уже знаем — по типу MIME этого файла.

В табл. 4.2 перечислены типы MIME форматов мультимедийных файлов, поддерживаемых Web-обозревателями на данный момент.

Таблица 4.2. *Типы MIME форматов мультимедийных файлов, поддерживаемых современными Web-обозревателями*

Формат файлов	Тип MIME

WAV	`audio/wave`
	`audio/wav`
	`audio/x-wav`
	`audio/x-pn-wav`

OGG	`audio/ogg` (для аудиофайлов) `video/ogg` (для видеофайлов) `application/ogg`
MP4	`video/mp4`
QuickTime	`video/quicktime`

Как видим, один формат файлов может иметь несколько типов MIME. Обычно выбирается самый первый из списка как самый предпочтительный.

Вооружившись необходимой теорией, приступим к практике. Сейчас мы выясним, как HTML 5 позволит нам поместить аудио или видео на Web-страницу.

Вставка аудиоролика

Для вставки на Web-страницу аудиоролика язык HTML 5 предусматривает парный тег `<AUDIO>`. Интернет-адрес файла, в котором хранится этот аудиоролик, указывают с помощью атрибута `SRC` этого тега:

```
<AUDIO SRC="sound.wav"></AUDIO>
```

Встретив тег `<AUDIO>`, Web-обозреватель может сразу загрузить и воспроизвести аудиофайл, только загрузить его без воспроизведения или вообще ничего не делать. (В последнем случае мы можем запустить воспроизведение из Web-сценария; о Web-сценариях разговор пойдет в *части III*.) Также он может вывести на Web-страницу элементы управления, с помощью которых посетитель запускает воспроизведение аудиофайла, приостанавливает его, прокручивает вперед или назад и регулирует громкость. Все это настраивается с помощью различных атрибутов тега `<AUDIO>`, которые мы скоро рассмотрим.

Тег `<AUDIO>` создает блочный элемент Web-страницы. Так что мы не сможем вставить аудиоролик на Web-страницу в качестве части абзаца. Зато, чтобы поместить

его в отдельный абзац, нам не придется совершать никаких дополнительных действий (в отличие от изображения).

По умолчанию Web-обозреватель не будет воспроизводить аудиоролик. Чтобы он это сделал, в теге <AUDIO> нужно указать особый атрибут AUTOPLAY. Это действительно особый атрибут: он не имеет значения — достаточно одного его присутствия в теге, чтобы он начал действовать *(атрибут тега без значения)*:

```
<P>Сейчас вы услышите звук!</P>
<AUDIO SRC="sound.ogg" AUTOPLAY></AUDIO>
```

По умолчанию аудиоролик никак не отображается на Web-странице (что, впрочем, понятно — аудио нужно не смотреть, а слушать). Но если в теге <AUDIO> поставить атрибут без значения CONTROLS, Web-обозреватель выведет в том месте Web-страницы, где проставлен тег <AUDIO>, элементы управления воспроизведением аудиоролика. Они включают кнопку запуска и приостановки воспроизведения, шкалу воспроизведения и регулятор громкости:

```
<P>Нажмите кнопку воспроизведения, чтобы услышать звук.</P>
<AUDIO SRC="sound.ogg" CONTROLS></AUDIO>
```

Атрибут без значения AUTOBUFFER имеет смысл указывать в теге <AUDIO> только в том случае, если там отсутствует атрибут AUTOPLAY. Если он указан, Web-обозреватель сразу после загрузки Web-страницы начнет загружать файл аудиоролика — это позволит исключить задержку файла перед началом его воспроизведения.

Чтобы проверить полученные знания в действии, нам понадобится аудиоролик поддерживаемого Web-обозревателем формата. Автор нашел у себя небольшой аудиоролик формата WAV-PCM и дал ему имя sound.wav. Вы можете использовать любой другой аудиоролик, но, разумеется, в HTML-коде листинга 4.2 придется указать имя файла, в котором он хранится.

Откроем Web-страницу index.htm и впишем в раздел тегов тег <AUDIO>. Создадим описывающую этот тег Web-страницу, HTML-код которой приведен в листинге 4.2.

Листинг 4.2

```
<!DOCTYPE html>
<HTML>
  <HEAD>
    <META HTTP-EQUIV="Content-Type" CONTENT="text/html; charset=utf-8">
    <TITLE>Тег AUDIO</TITLE>
  </HEAD>
  <BODY>
    <H1>Тег AUDIO</H1>
    <P>Тег AUDIO служит для вставки на Web-страницу аудиоролика.</P>
    <H6>Пример:</H6>
    <PRE>&lt;AUDIO SRC=&quot;sound.wav&quot;
    CONTROLS&gt;&lt;/AUDIO&gt;</PRE>
```

```
    <H6>Результат:</H6>
    <AUDIO SRC="sound.wav" CONTROLS></AUDIO>
  </BODY>
</HTML>
```

Сохраним Web-страницу в файле с именем t_audio.htm. Поместим выбранный аудиофайл (у автора — sound.wav) в папку, где находятся все файлы нашего Web-сайта (и t_audio.htm в том числе). И сразу же откроем только что созданную Web-страницу в Web-обозревателе (рис. 4.2).

Рис. 4.2. Web-страница t_audio.htm в Web-обозревателе

Мы увидим код примера и, ниже, результат его выполнения — симпатичные элементы для управления воспроизведением аудиоролика. Мы можем нажать кнопку воспроизведения и прослушать его.

Вставка видеоролика

Для вставки на Web-страницу видеоролика предназначен парный тег `<VIDEO>`. Интернет-адрес видеофайла указывается с помощью знакомого нам атрибута `SRC` этого тега:

```
<VIDEO SRC="film.ogg"></VIDEO>
```

Встретив этот тег, Web-обозреватель выведет в том месте Web-страницы, где он проставлен, панель просмотра содержимого видеоролика. В зависимости от указанных нами в теге `<VIDEO>` атрибутов, он может сразу загрузить и воспроизвести

аудиофайл, только загрузить его без воспроизведения или вообще ничего не делать. Также он может вывести на Web-страницу элементы управления воспроизведением видеоролика.

Как и тег <AUDIO>, тег <VIDEO> создает блочный элемент Web-страницы и поддерживает атрибуты AUTOPLAY, CONTROLS и AUTOBUFFER:

```
<VIDEO SRC="film.ogg" AUTOPLAY CONTROLS></VIDEO>
```

Если воспроизведение видеоролика еще не запущено, в панели просмотра будет выведен первый его кадр или вообще ничего (конкретное поведение зависит от Web-обозревателя). Но мы можем указать графическое изображение, которое будет туда выведено в качестве заставки. Для этого служит атрибут POSTER тега <VIDEO>; его значение указывает интернет-адрес нужного графического файла:

```
<VIDEO SRC="film.ogg" CONTROLS POSTER="filmposter.jpg"></VIDEO>
```

Здесь мы указали для видеоролика изображение-заставку, которое будет выведено в панели просмотра перед началом его воспроизведения и которое хранится в файле filmposter.jpg.

Ну что, попрактикуемся? Сначала найдем видеофайл подходящего формата. Автор отыскал небольшой видеоролик формата OGG и дал ему имя film.ogg. Вы можете выбрать любой другой видеоролик, но, разумеется, в приводимом далее HTML-коде придется указать имя его файла.

> ***НА ЗАМЕТКУ***
>
> Если вы не найдете видеоролика подходящего формата, то можете сами создать его, перекодировав видеоролик, сохраненный в другом формате. Для этого подойдет утилита SUPER ©, которую можно найти по интернет-адресу **http://www.erightsoft.com/SUPER.html**. Она поддерживает очень много мультимедийных форматов, в том числе и OGG.

Откроем Web-страницу index.htm и впишем в раздел тегов тег <VIDEO>. Создадим описывающую этот тег Web-страницу, HTML-код которой приведен в листинге 4.3.

Листинг 4.3

```
<!DOCTYPE html>
<HTML>
  <HEAD>
    <META HTTP-EQUIV="Content-Type" CONTENT="text/html; charset=utf-8">
    <TITLE>Тег VIDEO</TITLE>
  </HEAD>
  <BODY>
    <H1>Тег VIDEO</H1>
    <P>Тег VIDEO служит для вставки на Web-страницу видеоролика.</P>
    <H6>Пример:</H6>
    <PRE>&lt;VIDEO SRC=&quot;film.ogg&quot;
    CONTROLS&gt;&lt;/VIDEO&gt;</PRE>
    <H6>Результат:</H6>
```

```
    <VIDEO SRC="film.ogg" CONTROLS></VIDEO>
  </BODY>
</HTML>
```

Сохраним Web-страницу в файле с именем t_video.htm. Поместим выбранный видеофайл (у автора — film.ogg) в папку, где находятся все файлы нашего Web-сайта (и t_video.htm в том числе). И откроем готовую Web-страницу t_video.htm в Web-обозревателе (рис. 4.3).

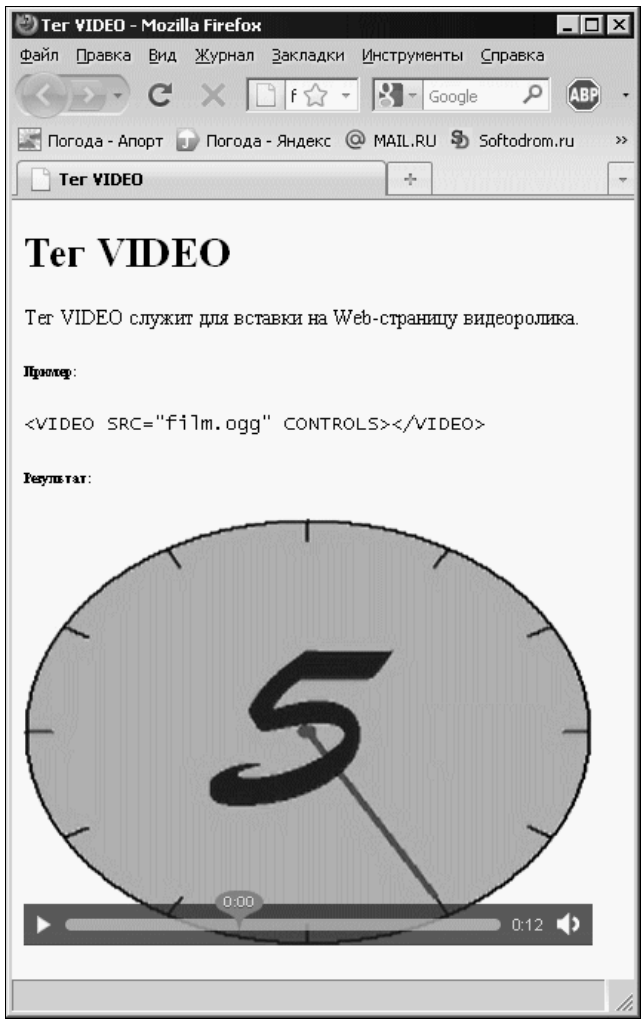


Рис. 4.3. Web-страница t_video.htm в Web-обозревателе

Ниже кода примера мы увидим результат его выполнения — панель просмотра и элементы для управления воспроизведением. Нажмем кнопку воспроизведения и посмотрим "кино". Отметим, что после начала воспроизведения элементы управления пропадут; чтобы получить к ним доступ, следует навести на панель просмотра

курсор мыши. Как только воспроизведение видеоролика закончится, эти элементы управления снова появятся на экране.

Дополнительные возможности тегов <*AUDIO*> и <*VIDEO*>

Но постойте! Раньше мы узнали, что набор поддерживаемых мультимедийных форматов у разных Web-обозревателей различается. И может случиться так, что аудио- или видеоролик, который мы поместили на Web-страницу, окажется какому-то Web-обозревателю не "по зубам". Как быть?

Специально для таких случаев HTML 5 предусматривает возможность указать в одном теге <AUDIO> или <VIDEO> сразу несколько мультимедийных файлов. Web-обозреватель сам выберет из них тот, формат которого он поддерживает.

Если мы собираемся указать сразу несколько мультимедийных файлов в одном теге <AUDIO> или <VIDEO>, то должны сделать две вещи.

1. Убрать из тега <AUDIO> или <VIDEO> указание на мультимедийный файл, т. е. атрибут SRC и его значение.
2. Поместить внутри тега <AUDIO> или <VIDEO> набор тегов <SOURCE>, каждый из которых будет определять интернет-адрес одного мультимедийного файла.

Одинарный тег <SOURCE> указывает как интернет-адрес мультимедийного файла, так и его тип MIME. Для этого предназначены атрибуты SRC и TYPE данного тега соответственно:

```
<VIDEO>
  <SOURCE SRC="film.ogg" TYPE="video/ogg">
  <SOURCE SRC="film.mov" TYPE="video/quicktime">
</VIDEO>
```

Данный тег <VIDEO> определяет сразу два видеофайла, хранящих один и тот же фильм: один — формата OGG, другой — формата QuickTime. Web-обозреватель, поддерживающий формат OGG, загрузит и воспроизведет первый файл, а Web-обозреватель, поддерживающий QuickTime, — второй файл.

Отметим, что тип MIME видеофайла (и, соответственно, атрибут TYPE тега <SOURCE>) можно опустить. Но тогда Web-обозревателю придется загрузить начало файла, чтобы выяснить, поддерживает ли он формат этого файла. А это лишняя и совершенно ненужная нагрузка на сеть. Так что тип MIME лучше указывать всегда.

А если Web-обозреватель вообще не поддерживает теги <AUDIO> и <VIDEO>? В таком случае он их проигнорирует и ничего на Web-страницу не выведет. Однако мы можем указать текст замены, в котором описать возникшую проблему и предложить какой-либо путь ее решения (например, сменить Web-обозреватель). Такой текст замены ставят внутри тега <AUDIO> или <VIDEO> после всех тегов <SOURCE> (если они есть), например, как в листинге 4.4.

Листинг 4.4

```
<VIDEO>
  <SOURCE SRC="film.ogg" TYPE="video/ogg">
  <SOURCE SRC="film.mov" TYPE="video/quicktime">
  Ваш Web-обозреватель не поддерживает вывод мультимедийных данных
  средствами HTML 5. Попробуйте другой.
</VIDEO>
```

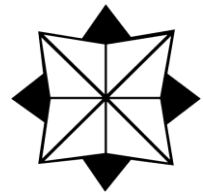
Отметим, что мы не указали в тексте замены теги, создающие абзац. Теги <AUDIO> и <VIDEO> сами по себе — блочные элементы, так что в этом нет нужды.

Вот и все об интернет-мультимедиа и средствах HTML 5 для его поддержки.

Что дальше?

В этой главе мы научились помещать на Web-страницы графические изображения, аудио- и видеоролики. И начали изучать HTML 5.

В следующей главе нас ждут таблицы.



ГЛАВА 5

Таблицы

В предыдущих главах мы узнали, как структурировать и оформлять текст и как помещать на Web-страницу графические изображения, аудио- и видеофайлы. Этих знаний хватит для создания большинства Web-страниц. Но не всех.

Что часто встречается в печатных изданиях, помимо текста и картинок? Таблицы! Таблицами пестрят учебники и справочники, таблицы часто попадаются в газетах и журналах, даже художественные произведения иногда огорошивают нас таблицами. И это понятно. Таблицы — лучший способ уместить множество сведений на ограниченной площади страницы.

Предоставляет ли HTML средства для создания таблиц? Да, и уже довольно давно. Сейчас мы с ними познакомимся.

Создание таблиц

Таблицы HTML создаются в четыре этапа.

На первом этапе в HTML-коде с помощью парного тега `<TABLE>` формируют саму таблицу:

```
<TABLE>
</TABLE>
```

Таблица HTML представляет собой блочный элемент Web-страницы. Это значит, что она размещается отдельно от всех остальных блочных элементов: абзацев, заголовков, больших цитат, аудио- и видеороликов. Так что вставить таблицу в абзац мы не сможем. (Нужно сказать, что таблица в абзаце выглядела бы, по меньшей мере, странно...)

На втором этапе формируют строки таблицы. Для этого предусмотрены парные теги `<TR>`; каждый такой тег создает отдельную строку. Теги `<TR>` помещают внутрь тега `<TABLE>` (листинг 5.1).

Листинг 5.1

```
<TABLE>
  <TR>
```

```
  </TR>
  <TR>
  </TR>
  <TR>
  </TR>
</TABLE>
```

На третьем этапе создают ячейки таблицы, для чего используют парные теги <TD> и <TH>. Тег создает обычную ячейку, а тег — *ячейку заголовка*, в которой будет помещаться "шапка" соответствующего столбца таблицы. Теги <TD> и <TH> помещают в теги <TR>, создающие строки таблицы, в которых должны находиться эти ячейки (листинг 5.2).

Листинг 5.2

```
<TABLE>
  <TR>
    <TH></TH>
    <TH></TH>
    <TH></TH>
  </TR>
  <TR>
    <TD></TD>
    <TD></TD>
    <TD></TD>
  </TR>
  <TR>
    <TD></TD>
    <TD></TD>
    <TD></TD>
  </TR>
</TABLE>
```

На четвертом, последнем, этапе указывают содержимое ячеек, которое помещают в соответствующие теги <TD> и <TH> (листинг 5.3).

Листинг 5.3

```
<TABLE>
  <TR>
    <TH>Столбец 1</TH>
    <TH>Столбец 2</TH>
    <TH>Столбец 3</TH>
  </TR>
  <TR>
    <TD>Ячейка 1.1</TD>
    <TD>Ячейка 1.2</TD>
```

```
    <TD>Ячейка 1.3</TD>
  </TR>
  <TR>
    <TD>Ячейка 2.1</TD>
    <TD>Ячейка 2.2</TD>
    <TD>Ячейка 2.3</TD>
  </TR>
</TABLE>
```

Если нам нужно поместить в ячейку таблицы простой текст, мы можем просто вставить его в соответствующий тег <TD> или <TH> (как показано в листинге 5.3). При этом заключать его в теги, создающие блочные элементы, необязательно.

Если нам потребуется как-то оформить содержимое ячеек, мы применим изученные в *главе 3* теги. Например, мы можем придать номерам ячеек особую важность, воспользовавшись тегом <EM>; в результате они будут выведены курсивом (листинг 5.4).

Листинг 5.4

```
<TABLE>
  . . .
  <TR>
    <TD>Ячейка <EM>1.1</EM></TD>
    <TD>Ячейка <EM>1.2</EM></TD>
    <TD>Ячейка <EM>1.3</EM></TD>
  </TR>
  . . .
</TABLE>
```

Еще мы можем поместить в ячейку графическое изображение:

```
<TD><IMG SRC="picture.jpg" ALT="Картинка в ячейке таблицы"></TD>
```

Но часто бывает необходимо поместить в ячейку таблицы большой текст, иногда состоящий из нескольких абзацев. В таком случае пригодятся знакомые нам по *главе 2* теги, создающие блочные элементы страницы. Теги <TD> и <TH> это позволяют (листинг 5.5).

Листинг 5.5

```
<TD>
  <H4>Это большой текст</H4>
  <P>Это начало большого текста, представляющего собой содержимое ячейки
  таблицы.</P>
  <P>Это продолжение большого текста, представляющего собой содержимое
  ячейки таблицы.</P>
  <P><IMG SRC="picture.jpg" ALT="Иллюстрация к большому тексту"></P>
  <P>А это <STRONG>долгожданное окончание</STRONG> большого текста.</P>
</TD>
```

Данный HTML-код помещает в ячейку таблицы заголовок и четыре абзаца. Причем один из этих абзацев содержит графическое изображение, а часть другого помечена как очень важная (и будет набрана полужирным шрифтом).

HTML-код, создающий таблицы, может показаться несколько громоздким. Но это плата за исключительную гибкость таблиц HTML. Мы можем поместить в таблицу любое содержимое: абзацы, заголовки, изображения, аудио- и видеоролики и даже другие таблицы.

Теперь настала пора рассмотреть правила, которыми руководствуются Web-обозреватели при выводе таблиц на экран.

- Таблица представляет собой блочный элемент Web-страницы (об этом мы уже говорили).
- Размеры таблицы и ее ячеек делаются такими, чтобы полностью вместить их содержимое.
- Между границами отдельных ячеек и между границей каждой ячейки и ее содержимым делается небольшой отступ.
- Текст ячеек заголовка выводится полужирным шрифтом и выравнивается по центру.
- Рамки вокруг всей таблицы и вокруг отдельных ее ячеек не рисуются.

Таблица — всего лишь содержимое Web-страницы, а за ее вывод "отвечает" представление. (Подробнее о содержимом и представлении Web-страницы см. в *главе 1*.) Если нам нужно, например, вывести вокруг таблицы рамку, мы сможем создать соответствующее представление. Этим мы и займемся в *части II*.

И еще несколько правил, согласно которым создается HTML-код таблиц. Если их нарушить, Web-обозреватель отобразит таблицу некорректно или не выведет ее вообще.

- Тег `<TR>` может находиться только внутри тега `<TABLE>`. Любое другое содержимое тега `<TABLE>` (кроме заголовка и секций таблицы, речь о которых пойдет далее) будет проигнорировано.
- Теги `<TD>` и `<TH>` могут находиться только внутри тега `<TR>`. Любое другое содержимое тега `<TR>` будет проигнорировано.
- Содержимое таблицы может находиться только в тегах `<TD>` и `<TH>`.
- Ячейки таблицы должны иметь хоть какие-то содержимое, иначе Web-обозреватель может их вообще не отобразить. Если же какая-то ячейка должна быть пустой, в нее следует поместить неразрывный пробел (HTML-литерал ` `).

Все, с теорией покончено. Настала пора практики. Давайте поместим на Web-страницу index.htm таблицу, перечисляющую все версии языка HTML с указанием года выхода. Вставим ее после цитаты из Википедии и отделяющей ее горизонтальной линии.

Листинг 5.6 содержит фрагмент HTML-кода Web-страницы index.htm, создающий такую таблицу.

Листинг 5.6

```
. . .
<P>Пожалуй, ни убавить ни прибавить...</P>
<HR>
<P>Список версий HTML:</P>
<TABLE>
  <TR>
    <TH>Версия HTML</TH>
    <TH>Год выхода</TH>
    <TH>Особенности</TH>
  </TR>
  <TR>
    <TD>1.0</TD>
    <TD>1992</TD>
    <TD>Официально не была стандартизована</TD>
  </TR>
  <TR>
    <TD>2.0</TD>
    <TD>1995</TD>
    <TD>Первая стандартизованная версия</TD>
  </TR>
  <TR>
    <TD>3.2</TD>
    <TD>1997</TD>
    <TD>Поддержка таблиц и графики</TD>
  </TR>
  <TR>
    <TD>4.0</TD>
    <TD>1997</TD>
    <TD>&quot;Очищен&quot; от устаревших тегов</TD>
  </TR>
  <TR>
    <TD>4.01</TD>
    <TD>1999</TD>
    <TD>В основном, исправление ошибок</TD>
  </TR>
  <TR>
    <TD>5.0</TD>
    <TD>?</TD>
    <TD>В разработке</TD>
  </TR>
</TABLE>
<P>HTML позволяет формировать на Web-страницах следующие элементы:</P>
. . .
```

Сохраним Web-страницу и откроем в Web-обозревателе (рис. 5.1).

Версия HTML	Год выхода	Особенности
1.0	1992	Официально не была стандартизована
2.0	1995	Первая стандартизованная версия
3.2	1997	Поддержка таблиц и графики
4.0	1997	"Очищен" от устаревших тегов
4.01	1999	В основном, исправление ошибок
5.0	?	В разработке

Рис. 5.1. Таблица — список версий HTML на Web-странице index.htm

Как видим, наша первая таблица не очень презентабельна. Web-обозреватель сделал ее сжатой, без рамок, с маленькими отступами между ячейками. Ну да это дело поправимое — прочитав *часть II*, мы сможем оформить таблицу (и другие элементы Web-страницы) как пожелаем.

Заголовок и секции таблицы

Теперь рассмотрим дополнительные возможности HTML по созданию таблиц. На практике они применяются нечасто, но иногда могут пригодиться.

Прежде всего, с помощью парного тега `<CAPTION>` мы можем дать таблице заголовок. Текст заголовка помещают внутрь этого тега, который, в свою очередь, находится внутри тега `<TABLE>` (листинг 5.7).

Листинг 5.7

```
<TABLE>
  <CAPTION>Это таблица</CAPTION>
  <TR>
    <TH>Столбец 1</TH>
    <TH>Столбец 2</TH>
    <TH>Столбец 3</TH>
  </TR>
  . . .
</TABLE>
```

Заголовок таблицы выводится над ней, а текст его выравнивается по центру таблицы. При желании мы можем его как-то оформить, воспользовавшись знакомыми нам по *главе 3* тегами:

```
<CAPTION><STRONG>Это таблица<STRONG></CAPTION>
```

Обычно тег `<CAPTION>` помещается сразу после открывающего тега `<TABLE>` — так логичнее. Но не имеет значения, в каком месте HTML-кода таблицы он присутствует — заголовок все равно будет помещен Web-обозревателем над таблицей.

Кроме того, мы можем логически разбить таблицу HTML на три значащие части — *секции таблицы*:

- *секцию заголовка*, в которой находится строка с ячейками заголовка, формирующая ее "шапку";

- *секцию тела*, где находятся строки таблицы, составляющие основные данные;
- *секцию завершения* со строками, формирующими "поддон" таблицы (обычно в "поддоне" располагают итоговые данные и различные примечания).

Секцию заголовка таблицы формирует тег <THEAD>, секцию тела — <TBODY>, а секцию завершения — <TFOOT>. Все эти теги парные, помещаются непосредственно в тег <TABLE> и содержат теги <TR>, формирующие строки таблицы, которые входят в соответствующую секцию (листинг 5.8).

Листинг 5.8

```
<TABLE>
  <THEAD>
    <TR>
      <TH>Столбец 1</TH>
      <TH>Столбец 2</TH>
      <TH>Столбец 3</TH>
    </TR>
  </THEAD>
  <TBODY>
    <TR>
      <TD>Ячейка 1.1</TD>
      <TD>Ячейка 1.2</TD>
      <TD>Ячейка 1.3</TD>
    </TR>
    <TR>
      <TD>Ячейка 2.1</TD>
      <TD>Ячейка 2.2</TD>
      <TD>Ячейка 2.3</TD>
    </TR>
  </TBODY>
  <TFOOT>
    <TR>
      <TD>Итог по ячейке 2.1</TD>
      <TD>Итог по ячейке 2.2</TD>
      <TD>Итог по ячейке 2.3</TD>
    </TR>
  </TFOOT>
</TABLE>
```

Секции таблицы Web-обозреватель никак не отображает и не выделяет на Web-странице. Они просто делят таблицу на три логические части. Однако мы можем задать для тегов, формирующих секции таблицы, какое-то представление, которое будет управлять их отображением. Подробнее об этом мы узнаем в *части II*.

НА ЗАМЕТКУ

Тег <TABLE> поддерживает необязательный атрибут SUMMARY, с помощью которого мы можем задать примечание к таблице. Это примечание вообще не показывается на эк-

ране, однако может использоваться другими средствами вывода Web-страниц, например, Web-обозревателями для слабовидящих, зачитывающими содержимое Web-страниц. Так или иначе, примечание к таблицам практически никогда не задается.

Чтобы закрепить полученные знания, давайте применим их к таблице — списку версий HTML, который мы недавно создали на Web-странице index.htm. Над этой таблицей мы вставили абзац с текстом "Список версий HTML:", который так и просится в заголовки. Заодно разделим таблицу на секции заголовка и тела. ("Поддона" наша таблица не имеет, так что обойдемся без секции завершения.)

В листинге 5.9 представлен исправленный фрагмент HTML-кода Web-страницы index.htm, который создает эту таблицу.

Листинг 5.9

```
. . .
<P>Пожалуй, ни убавить ни прибавить...</P>
<HR>
<TABLE>
  <CAPTION>Список версий HTML:</CAPTION>
  <THEAD>
    <TR>
      <TH>Версия HTML</TH>
      <TH>Год выхода</TH>
      <TH>Особенности</TH>
    </TR>
  </THEAD>
  <TBODY>
    <TR>
      <TD>1.0</TD>
      <TD>1992</TD>
      <TD>Официально не была стандартизована</TD>
    </TR>
    . . .
    <TR>
      <TD>5.0</TD>
      <TD>?</TD>
      <TD>В разработке</TD>
    </TR>
  </TBODY>
</TABLE>
<P>HTML позволяет формировать на Web-страницах следующие элементы:</P>
. . .
```

Сохраним исправленную Web-страницу и откроем ее в Web-обозревателе. И сразу увидим, что текст "Список версий HTML" теперь выровнен по центру таблицы. Это и неудивительно — ведь мы превратили его в заголовок таблицы. Сама же таблица ничуть не изменилась (что тоже понятно — ведь ее секции Web-обозреватель никак не выделяет).

Объединение ячеек таблиц

Осталось поговорить об одной интересной особенности языка HTML. Это так называемое *объединение ячеек* таблиц. Лучше всего рассмотреть пример — простую таблицу, HTML-код которой приведен в листинге 5.10.

Листинг 5.10

```
<TABLE>
  <TR>
    <TD>1</TD>
    <TD>2</TD>
    <TD>3</TD>
    <TD>4</TD>
    <TD>5</TD>
  </TR>
  <TR>
    <TD>6</TD>
    <TD>7</TD>
    <TD>8</TD>
    <TD>9</TD>
    <TD>10</TD>
  </TR>
  <TR>
    <TD>11</TD>
    <TD>12</TD>
    <TD>13</TD>
    <TD>14</TD>
    <TD>15</TD>
  </TR>
  <TR>
    <TD>16</TD>
    <TD>17</TD>
    <TD>18</TD>
    <TD>19</TD>
    <TD>20</TD>
  </TR>
</TABLE>
```

Это обычная таблица, ячейки которой пронумерованы — так нам будет проще в дальнейшем. На рис. 5.2 показан ее вид в окне Web-обозревателя.

А теперь рассмотрим таблицу на рис. 5.3.

Здесь выполнено объединение некоторых ячеек. Видно, что объединенные ячейки словно слились в одну. Как это сделать?

Специально для этого теги `<TD>` и `<TH>` поддерживают два весьма примечательных необязательных атрибута. Первый — `COLSPAN` — объединяет ячейки по горизонтали, второй — `ROWSPAN` — по вертикали.

1	2	3	4	5
6	7	8	9	10
11	12	13	14	15
16	17	18	19	20

Рис. 5.2. Изначальная таблица, ячейки которой подвергнутся объединению

1+6	2+3		4+5	
	7	8	9	10
11	12+13+14+15			
16	17	18	19	20

Рис. 5.3. Таблица, показанная на рис. 5.2, после объединения некоторых ячеек (объединенные ячейки обозначены сложением их номеров)

Чтобы объединить несколько ячеек по горизонтали в одну, нужно выполнить следующие шаги.

1. Найти в коде HTML тег `<TD>` (`<TH>`), соответствующий первой из объединяемых ячеек (если считать ячейки слева направо).
2. Вписать в него атрибут `COLSPAN` и присвоить ему количество объединяемых ячеек, считая и самую первую из них.
3. Удалить теги `<TD>` (`<TH>`), создающие остальные объединяемые ячейки данной строки.

Давайте объединим ячейки 2 и 3 таблицы (см. листинг 5.10). Исправленный фрагмент кода, создающий первую строку этой таблицы, приведен в листинге 5.11.

Листинг 5.11

```
<TR>
  <TD>1</TD>
  <TD COLSPAN="2">2 + 3</TD>
  <TD>4</TD>
  <TD>5</TD>
</TR>
```

Точно так же создадим объединенные ячейки 4 + 5 и 12 + 13 + 14 + 15.

Объединить ячейки по вертикали чуть сложнее. Вот шаги, которые нужно для этого выполнить.

1. Найти в коде HTML строку (тег <TR>), в которой находится первая из объединяемых ячеек (если считать строки сверху вниз).
2. Найти в коде этой строки тег <TD> (<TH>), соответствующий первой из объединяемых ячеек.
3. Вписать в него атрибут ROWSPAN и присвоить ему количество объединяемых ячеек, считая и самую первую из них.
4. Просмотреть последующие строки и удалить из них теги <TD> (<TH>), создающие остальные объединяемые ячейки.

Нам осталось объединить ячейки 1 и 6 нашей таблицы. Листинг 5.12 содержит исправленный фрагмент ее HTML-кода (исправления затронут первую и вторую строки).

Листинг 5.12

```
<TR>
  <TD ROWSPAN="2">1 + 6</TD>
  <TD COLSPAN="2">2 + 3</TD>
  <TD COLSPAN="2">4 + 5</TD>
</TR>
<TR>
  <TD>7</TD>
  <TD>8</TD>
  <TD>9</TD>
  <TD>10</TD>
</TR>
```

Обратим внимание, что мы удалили из второй строки тег <TD>, создающий шестую ячейку, поскольку она объединилась с первой ячейкой.

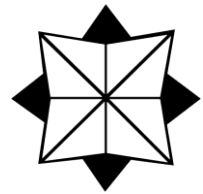
Объединение ячеек сейчас применяется не очень часто. Однако ранее, в период расцвета табличного Web-дизайна (о табличном Web-дизайне см. в *главе 10*), трудно было встретить таблицу без объединенных ячеек. Так или иначе, знать о нем не помешает.

Что дальше?

В этой главе мы познакомились со средствами HTML, предназначенными для формирования таблиц, и создали нашу первую таблицу, перечисляющую версии языка HTML. Должны же посетители нашего Web-сайта знать, с чего все начиналось...

В следующей главе мы познакомимся со средствами навигации по Web-сайту. Это всевозможные гиперссылки, как текстовые, так и графические (изображения-гиперссылки и карты-изображения). И наконец-то объединим разрозненные Web-странички в единый Web-сайт!

ГЛАВА 6



Средства навигации

В предыдущих главах мы наполняли Web-страницы содержимым: текстом, графическими изображениями, аудио- и видеороликами и таблицами. Теперь, по крайней мере, некоторые Web-страницы у нас полностью готовы. (А остальные можно сделать по образу и подобию уже созданных.)

Осталось связать эти разрозненные Web-страницы воедино — в Web-сайт. Как это осуществить? С помощью средств навигации — гиперссылок.

Гиперссылка обычно выглядит как подчеркнутый фрагмент текста; если навести на него курсор мыши, он примет вид "указующего перста". При щелчке на гиперссылке Web-обозреватель загрузит Web-страницу, интернет-адрес которой указан в параметрах данной гиперссылки (*целевую* Web-страницу). Гиперссылка может иметь вид графического изображения или его фрагмента, такие гиперссылки сейчас очень популярны.

Текстовые гиперссылки

Начнем с самых простых — *текстовых* гиперссылок, которые представляют собой фрагмент текста.

Создание гиперссылок

Создать текстовую гиперссылку очень просто. Достаточно найти в блочном элементе (например, абзаце) фрагмент текста, который нужно превратить в гиперссылку, и заключить его в парный тег `<A>`. Интернет-адрес целевой Web-страницы указывают в атрибуте `HREF` этого тега.

Гиперссылка (т. е. тег `<A>`) представляет собой встроенный элемент Web-страницы, т. е. это часть блочного элемента, например, абзаца:

Вот гиперссылка, которая указывает на Web-страницу page125.html, хранящуюся в папке pages, вложенной в корневую папку сайта, на сайте **http://www.somesite.ru**:

```
<A HREF="http://www.somesite.ru/pages/page125.html">Страница №125</A>
```

А эта гиперссылка указывает на архивный файл archive.zip, хранящийся в той же папке, что и Web-страница, которая в данный момент открыта в Web-обозревателе (*текущая* Web-страница):

```
<A HREF="archive.zip">Архив</A>
```

При щелчке на гиперссылке Web-обозреватель предложит загрузить этот архивный файл и либо открыть его, либо сохранить на диске клиентского компьютера.

Пример:

```
<P><A HREF="22.html">Предыдущая страница</A>,
<A HREF="24.html">следующая страница</A>.</P>
```

Этот фрагмент HTML-кода создает абзац, содержащий сразу две гиперссылки, которые указывают на разные целевые Web-страницы.

Текст, являющийся гиперссылкой, можно оформлять, используя любые теги, приведенные в *главе 3*.

Пример:

```
<A HREF="http://www.somesite.ru/pages/page125.html">Страница
<EM>№125</EM></A>
```

Тег <A> поддерживает необязательный атрибут TARGET. Он задает *цель гиперссылки*, указывающую, где будет открыта целевая Web-страница. Так, если атрибуту TARGET присвоить значение "_blank", целевая страница будет открыта в новом окне Web-обозревателя.

Например, если мы изменим код первого примера гиперссылки таким образом (исправления выделены полужирным шрифтом):

```
<A HREF="http://www.somesite.ru/pages/page125.html" TARGET="_blank">
↳Страница №125</A>
```

"Страница №125" будет открыта в новом окне Web-обозревателя.

Чтобы задать обычное поведение гиперссылки (когда целевая Web-страница открывается в том же окне Web-обозревателя), нужно присвоить атрибуту TARGET значение "_self" (это его значение по умолчанию) или вообще убрать данный атрибут из тега <A>.

Имеется также возможность создать гиперссылку, которая никуда не указывает (*"пустая"* гиперссылка). Для этого достаточно задать в качестве значения атрибута HREF значок # ("решетка"):

```
<A HREF="#">А я никуда не веду!</A>
```

При щелчке на такой гиперссылке ничего не произойдет.

> ***НА ЗАМЕТКУ***
>
> "Пустыми" гиперссылками мы будем активно пользоваться в *частях III* и *IV*, когда начнем писать Web-сценарии.

Правила отображения гиперссылок Web-обозревателем:

- обычные гиперссылки выделяются синим цветом;
- гиперссылки, по которым посетитель уже "ходил" (*посещенные* гиперссылки), выводятся темно-красным цветом;
- гиперссылка, по которой посетитель в данный момент щелкает (*активная* гиперссылка), выводится ярко-красным цветом;
- текст любых гиперссылок подчеркивается;
- при помещении курсора мыши на гиперссылку Web-обозреватель меняет его форму на "указующий перст".

Таково поведение по умолчанию, которое мы можем изменить, создав соответствующее представление. О том, как это сделать, будет рассказано в *части II*.

Интернет-адреса в WWW

Об интернет-адресах файлов мы говорили еще в *главе 1*. Однако WWW привносит в них кое-что новое, что нам обязательно нужно знать.

Рассмотрим первый пример гиперссылки из предыдущего раздела. Ее интернет-адрес таков: **http://www.somesite.ru/pages/page125.html**. Он содержит и интернет-адрес Web-сервера, и путь файла, который нужно получить. Поэтому он называется *полным*. Полные интернет-адреса используют, если нужно создать гиперссылку, указывающую на файл, в составе другого Web-сайта.

Однако если гиперссылка указывает на файл, входящий в состав того же Web-сайта, что и файл текущей Web-страницы, предпочтительнее *сокращенный* интернет-адрес, содержащий только имя нужного файла (интернет-адрес Web-сервера и так известен Web-обозревателю).

Существуют два типа сокращенных интернет-адресов. Адреса первого типа задают путь к файлу, который нужно получить (*целевому* файлу), относительно корневой папки Web-сайта. Они содержат в своем начале символ / (слэш), который и говорит Web-серверу, что путь нужно отсчитывать относительно корневой папки.

> ***НА ЗАМЕТКУ***
>
> О корневой папке сайта также было рассказано в *главе 1*. Вкратце: это особая папка, находящаяся на диске компьютера, на котором хранится Web-сайт и работает Web-сервер; в этой папке должны помещаться все файлы Web-сайта.

Например, интернет-адрес

/page3.html

указывает на файл page3.html, хранящийся в корневой папке Web-сайта.

А интернет-адрес

/articles/article1.html

указывает на файл article1.html, хранящийся в папке articles, вложенной в корневую папку Web-сайта.

Такие интернет-адреса называют *абсолютными* и используют, если нужно создать гиперссылку на файл, хранящийся в "глубине" Web-сайта (скажем, в другой папке, нежели файл текущей Web-страницы).

Сокращенные интернет-адреса второго типа задают путь к целевому файлу относительно файла текущей Web-страницы. Они не содержат в начале символа слэша — и в этом их важное отличие от абсолютных интернет-адресов.

Рассмотрим несколько примеров подобных интернет-адресов.

Интернет-адрес

archive.zip

указывает на файл archive.zip, хранящийся в той же папке, что и файл текущей Web-страницы.

Интернет-адрес

chapter3/page1.html

указывает на Web-страницу page1.html, хранящуюся в папке chapter3, вложенной в папку, в которой хранится текущая Web-страница.

Следующий интернет-адрес

../contents.html

указывает на Web-страницу contents.html, хранящуюся в папке, в которую вложена папка, где хранится текущая Web-страница. (Обратим внимание на две точки в начале пути — так обозначается папка предыдущего уровня вложенности.)

Такие интернет-адреса называют *относительными*. Их применяют, если нужно создать гиперссылку на файл, хранящийся в той же папке, что и текущая Web-страница, одной из вложенных в нее папок или папке предыдущего уровня вложенности.

> ***НА ЗАМЕТКУ***
>
> Во многих случаях лучше поэкспериментировать с разными интернет-адресами, чтобы выяснить, какой именно подойдет — абсолютный или относительный.

> ***ВНИМАНИЕ!***
>
> В Web-страницах, которые не должны быть опубликованы на Web-серверах, а будут открываться с диска клиентских компьютеров, следует применять только относительные интернет-адреса. Дело в том, что файловая система компьютера не знает, какую папку считать корневой, поэтому не сможет правильно интерпретировать абсолютные интернет-адреса. (Разумеется, гиперссылки, ссылающиеся на другие Web-сайты, должны содержать полные интернет-адреса.)

Разобравшись с гиперссылками и интернет-адресами, свяжем, наконец, наши Web-странички в единый Web-сайт. Чтобы нам было удобнее, создадим в папке, где хранятся все файлы нашего Web-сайта, папку tags. В эту папку перенесем все Web-страницы, описывающие теги HTML (их у нас пока что четыре), и сопутствующие им файлы (их три: изображение, аудио- и видеоролик). Файл index.htm никуда из корневой папки перемещать не будем — ведь он хранит Web-страницу по умолчанию.

Откроем Web-страницу index.htm и найдем в ней HTML-код, формирующий список тегов. Создадим там гиперссылки, указывающие на соответствующие Web-страницы.

Вот HTML-код, создающий гиперссылку, указывающую на Web-страницу с описанием тега <AUDIO>:

```
<CODE><A HREF="tags/t_audio.htm">AUDIO</A></CODE>
```

Остальные гиперссылки создаются аналогично.

Перейдя на Web-страницу, описывающую какой-либо тег, посетитель должен иметь возможность вернуться назад — на главную Web-страницу. Конечно, это можно сделать, нажав кнопку возврата назад на панели инструментов Web-обозревателя или клавишу <Backspace> на клавиатуре. Но правила хорошего тона Web-дизайна требуют, чтобы на Web-странице присутствовала соответствующая гиперссылка.

Создадим такую гиперссылку на всех Web-страницах, описывающих теги. Поместим ее в самом конце каждой Web-страницы — обычно она находится именно там. Вот так выглядит формирующий ее HTML-код:

```
<P><A HREF="../index.htm">На главную Web-страницу</A></P>
```

Осталось создать на главной Web-странице гиперссылку на Web-страницу Русской Википедии, которая содержит статью, посвященную языку HTML. Вставим ее в конец большой цитаты (листинг 6.1).

Листинг 6.1

```
<BLOCKQUOTE>
  <P>HTML (от англ. HyperText Markup Language — язык разметки
  гипертекста) — стандартный язык разметки документов во Всемирной
  паутине. (<A HREF="http://ru.wikipedia.org/wiki/HTML"
  TARGET="_blank">вся статья</A>)</P>
</BLOCKQUOTE>
```

Здесь мы указали для тега <A> атрибут TARGET со значением "_blank". И Web-страница с текстом статьи об HTML будет открываться в новом окне Web-обозревателя. Так что посетитель сможет "залезть" в Википедию, не покидая нашего Web-сайта.

Почтовые гиперссылки

HTML позволяет нам создать гиперссылку, указывающую на адрес электронной почты *(почтовую гиперссылку)*. Щелчок на ней запустит программу почтового клиента, установленную в системе по умолчанию. Интернет-адрес такой гиперссылки записывается особым образом.

Пусть мы хотим создать гиперссылку, указывающую на почтовый адрес:

user@mailserver.ru

Согласно стандарту HTML, этот почтовый адрес должен быть записан так:

mailto:user@mailserver.ru

причем между двоеточием после **mailto** и собственно адресом не должно быть пробелов.

Тогда наша почтовая гиперссылка будет выглядеть так:

```
<A HREF="mailto:user@mailserver.ru">Отправить письмо</A>
```

В конце главной Web-страницы у нас приведены сведения о правах разработчиков. Давайте превратим слово "читатели" в почтовую гиперссылку. Создающий ее HTML-код будет выглядеть так:

```
<A HREF="mailto:user@mailserver.ru">читатели</A>
```

Адрес электронной почты в этой гиперссылки выдуман. Разумеется, вы можете заменить его на реальный.

Дополнительные возможности гиперссылок

Язык HTML предлагает нам некоторые дополнительные возможности для создания гиперссылок. Их применяют нечасто, но иногда они полезны.

Прежде всего, мы можем указать для гиперссылки *"горячую" клавишу*. Если посетитель нажмет эту клавишу, удерживая нажатой клавишу <Alt>, Web-обозреватель выполнит переход по данной гиперссылке.

Для указания "горячей" клавиши предусмотрен необязательный атрибут `ACCESSKEY` тега `<A>`. Значение этого атрибута — латинская буква, соответствующая нужной клавише:

```
<A HREF="http://www.somesite.ru/pages/page125.html"
ACCESSKEY="d">Страница №125</A>
```

Здесь мы указали для гиперссылки "горячую" клавишу <D>. И, чтобы перейти по ней, посетителю будет достаточно нажать комбинацию клавиш <Alt>+<D>.

На гиперссылках можно щелкать мышью — так поступает большинство пользователей. Но по ним также можно "путешествовать" с помощью клавиатуры. В этом случае говорят о *фокусе ввода* — признаке, какая гиперссылка будет обрабатывать нажатия клавиш. Гиперссылка, имеющая фокус ввода, выделяется тонкой черной штриховой рамкой.

- Если нажать клавишу <Enter>, Web-обозреватель выполнит переход по гиперссылке, имеющей в данный момент фокус ввода.
- Если нажать клавишу <Tab>, Web-обозреватель перенесет фокус ввода на следующую гиперссылку.
- Если нажать комбинацию клавиш <Shift>+<Tab>, Web-обозреватель перенесет фокус ввода на предыдущую гиперссылку.

Порядок, в котором выполняется перенос фокуса ввода с одной гиперссылки на другую при нажатии клавиш <Tab> или <Shift>+<Tab>, так и называется — *поря-*

док обхода. По умолчанию он совпадает с порядком, в котором гиперссылки определены в HTML-коде Web-страницы. Но мы можем указать свой порядок обхода с помощью атрибута TABINDEX тега <A>. Его значение — целое число от –32 767 до 32 767 — *номер в порядке обхода*.

- Если указан положительный номер, именно он будет определять порядок обхода. Иными словами, сначала фокус ввода получит гиперссылка с номером 1, потом — с номером 2, далее — с номером 3 и т. д.
- Если указан номер, равный нулю, обход будет осуществляться в порядке, в котором гиперссылка определена в HTML-коде Web-страницы. Фактически ноль — значение атрибута тега TABINDEX по умолчанию.
- Если указан отрицательный номер, данная гиперссылка вообще исключается из порядка обхода. До нее невозможно будет добраться с помощью клавиатуры — можно будет только щелкать мышью.

Рассмотрим небольшой пример:

```
<P><A HREF="page1.htm" TABINDEX="3">Страница 1</A></P>
<P><A HREF="page2.htm" TABINDEX="2">Страница 2</A></P>
<P><A HREF="page3.htm" TABINDEX="1">Страница 3</A></P>
```

Этот HTML-код создает три гиперссылки с "обратным" порядком обхода. Сначала фокус ввода получит гиперссылка "Страница 3", потом — "Страница 2" и напоследок — "Страница 1".

Графические гиперссылки

В начале этой главы говорилось, что гиперссылка может быть в виде не только фрагмента текста, но и картинки или даже представлять собой фрагмент графического изображения. Вот *графическими гиперссылками* мы сейчас и займемся.

Изображения-гиперссылки

Язык HTML позволяет использовать в качестве содержимого гиперссылки любой фрагмент любого блочного элемента, в том числе и графическое изображение, т. е. создать *изображение-гиперссылку*.

Для создания изображения гиперссылки достаточно поместить внутрь тега <A> тег <IMG>:

```
<A HREF="http://www.w3.org"><IMG SRC="w3logo.gif"></A>
```

Этот HTML-код создает изображение-гиперссылку, указывающую на Web-сайт организации W3C. А в качестве самого изображения выбран логотип этой организации, который мы сохранили в файле в той же папке, где находится файл текущей Web-страницы.

```
<A HREF="mailto:user@mailserver.ru"><IMG SRC="email.gif"></A>
```

А этот HTML-код создает почтовую изображение-гиперссылку.

Правила вывода изображений-гиперссылок Web-обозревателем:

- изображение-гиперссылка окружается рамкой, имеющей соответствующий гиперссылке цвет: синий — для непосещенной, темно-красный — для посещенной и т. д.;
- при помещении курсора мыши на изображение-гиперссылку Web-обозреватель меняет его форму на "указующий перст", как и в случае текстовой гиперссылки.

Рамка вокруг изображения-гиперссылки зачастую выглядит непрезентабельно, поэтому ее обычно убирают, задав соответствующее представление. О представлении Web-страниц мы поговорим в *части II*.

Изображения-карты

А еще HTML позволяет превратить в гиперссылку часть графического изображения. Более того, мы можем разбить изображение на части, каждая из которых будет представлять собой гиперссылку, указывающую на свой интернет-адрес. Такие изображения называют *изображениями-картами*, а ее части-гиперссылки — *"горячими" областями*.

С помощью изображений-карт часто создают заголовки Web-сайтов, фрагменты которого представляют собой гиперссылки, указывающие на определенную Web-страницу. Пик популярности изображений-карт давно прошел, однако их еще можно довольно часто встретить.

Изображение-карту создают в три этапа. Первый этап — создание самого изображения с помощью хорошо нам знакомого тега `<IMG>`:

```
<IMG SRC="map.gif">
```

Второй этап — создание *карты*, особого элемента Web-страницы, который описывает набор "горячих" областей изображения-карты. Сама карта на Web-странице никак не отображается.

Карту создают с помощью парного тега `<MAP>`:

```
<MAP NAME="<имя карты>">
</MAP>
```

> ***Внимание!***
>
> Здесь для описания формата тега `<MAP>` впервые применяются типографские соглашения, перечисленные во *введении* этой книги. Автор настоятельно рекомендует прежде ознакомиться с ними.

С помощью обязательного атрибута `NAME` тега `<MAP>` задается уникальное в пределах Web-страницы имя карты. Оно однозначно идентифицирует данную карту, может содержать только латинские буквы, цифры и знаки подчеркивания и начинаться должно с буквы:

```
<MAP NAME="samplemap">
</MAP>
```

После создания карты следует привязать ее к созданному на первом этапе изображению. Для этого мы применим обязательный в данном случае атрибут USEMAP тега <IMG>. Его значение — имя привязываемой к изображению карты, причем в начале этого имени обязательно следует поставить символ # ("решетка"). (В имени, заданном атрибутом NAME тега <MAP>, символ # присутствовать не должен.)

```
<IMG SRC="map.gif" USEMAP="#samplemap">
```

На третьем этапе создают описания самих "горячих" областей в карте. Их помещают внутрь соответствующего тега <MAP> и формируют с помощью одинарных тегов <AREA>:

```
<AREA [SHAPE="rect|circle|poly"] COORDS="<набор параметров>"
HREF="<интернет-адрес гиперссылки>"|NOHREF
TARGET="<цель гиперссылки>">
```

Необязательный атрибут SHAPE задает форму "горячей" области. Обязательный атрибут COORDS перечисляет координаты, необходимые для построения этой области. Значения атрибута SHAPE:

- "rect" — прямоугольная "горячая" область. Атрибут COORDS в этом случае записывается в виде COORDS="*<X1>*,*<Y1>*,*<X2>*,*<Y2>*", где X1 и Y1 — координаты верхнего левого, а X2 и Y2 — правого нижнего угла прямоугольника. Кстати, если атрибут SHAPE отсутствует, создается именно прямоугольная область;
- "circle" — круглая "горячая" область. В этом случае атрибут COORDS имеет вид COORDS="*<X центра>*,*<Y центра>*,*<радиус>*";
- "poly" — "горячая" область в виде многоугольника. Атрибут COORDS должен иметь вид COORDS="*<X1>*,*<Y1>*,*<X2>*,*<Y3>*,*<X3>*,*<Y3>*...", где X*n* и Y*n* — координаты соответствующей вершины многоугольника.

Атрибут HREF задает интернет-адрес гиперссылки — он, собственно, нам уже знаком. Он может быть заменен атрибутом без значения NOHREF, задающим область, не связанную ни с каким интернет-адресом. Это позволяет создавать оригинальные изображения-карты, например, карту в виде бублика, "дырка" которого никуда не указывает.

Также знакомый нам атрибут TARGET задает цель гиперссылки. (Конечно, указывать его имеет смысл только в том случае, если мы создаем именно "горячую" область, а не "дырку" с атрибутом NOHREF.)

Листинг 6.2 содержит полный HTML-код, создающий изображение-карту.

Листинг 6.2

```
<IMG SRC="map.gif" USEMAP="#samplemap">
. . .
<MAP NAME="samplemap">
  <AREA SHAPE="circle" COORDS="50,50,30" HREF="page1.html">
  <AREA SHAPE="circle" COORDS="50,150,30" HREF="page2.html">
  <AREA SHAPE="poly" COORDS="100,50,100,100,150,50,100,50" NOHREF>
```

```
  <AREA SHAPE="rect" COORDS="0,100,30,100" HREF="appendix.html"
  TARGET="_blank">
</MAP>
```

Здесь мы создали две круглые "горячие" области, указывающие на Web-страницы page1.html и page2.html, многоугольную область, не ссылающуюся никуда, и прямоугольную область, ссылающуюся на Web-страницу appendix.html. Причем последняя "горячая" область при щелчке на ней откроет Web-страницу в новом окне Web-обозревателя.

Вот и все об изображениях-картах и о графических гиперссылках вообще.

Полоса навигации

Гиперссылки не всегда "ходят поодиночке". Довольно часто на Web-страницах присутствуют целые наборы гиперссылок, ссылающихся на разные Web-страницы данного Web-сайта. Такие наборы называются *полосами навигации*.

Полоса навигации может быть расположена по горизонтали, вверху или внизу Web-страницы, или по вертикали, слева или справа. Горизонтальную полосу навигации можно сформировать с помощью обычного абзаца или таблицы с одной строкой и нужным числом ячеек (каждая гиперссылка располагается в своей ячейке таблицы). Вертикальную полосу навигации обычно формируют с помощью таблицы из одного столбца и нужного числа ячеек (опять же, каждая гиперссылка располагается в своей ячейке таблицы), набора абзацев (каждая гиперссылка представляет собой отдельный абзац) или в виде списка (гиперссылки представляют собой пункты этого списка).

Гиперссылки полосы навигации могут быть текстовыми или графическими. В последнем случае практически всегда применяют изображения-гиперссылки. Зачастую для изображений-гиперссылок реализуют особое поведение, заменяющее изображение другим при наведении на соответствующую гиперссылку курсора мыши. (О поведении Web-страницы мы поговорим в *части III*.)

Полосу гиперссылок всегда выделяют, чтобы привлечь к ней внимание посетителя. Ее могут выделить цветом текста и фона, рамкой, увеличенным размером шрифта или всем вместе. Все это реализуется с помощью задания соответствующего представления Web-страницы.

Давайте создадим на главной Web-странице нашего Web-сайта полосу навигации с гиперссылками, указывающими на другие его Web-страницы: посвященную CSS, содержащую список примеров и сведения о разработчиках. Для простоты реализуем ее в виде абзаца, содержащего нужные гиперссылки. Поместим ее в самом верху Web-страницы, перед заголовком (листинг 6.3).

Листинг 6.3

```
. . .
<BODY>
  <P><A HREF="css_index.htm">CSS</A> |
```

```
<A HREF="samples_index.htm">Примеры</A> |
<A HREF="about.htm">О разработчиках</A></P>
<H1>Справочник по HTML и CSS</H1>
. . .
```

Как видим, наша первая полоса навигации очень проста — обычный абзац с набором гиперссылок, разделенных символом | (вертикальная черта). Ну так ведь и наш первый Web-сайт не слишком сложен...

При желании мы можем создать остальные Web-страницы нашего первого Web-сайта. Но не будем сильно в этом усердствовать — все равно мы его потом переделаем.

Якоря

Напоследок рассмотрим еще одну возможность, предлагаемую нам языком HTML и способную сильно упростить посетителям чтение длинных текстов. Хотя она и не относится к гиперссылкам напрямую, но действует совместно с ними.

Это так называемые *якоря* (anchors). Они не указывают на другую Web-страницу (файл, адрес электронной почты), а помечают некоторый фрагмент текущей Web-страницы, чтобы другая гиперссылка могла на него сослаться. Так можно пометить отдельные главы длинного текстового документа и посетитель сможет "перескочить" к нужной ему главе, щелкнув гиперссылку в оглавлении. Очень удобно!

Якоря создают с помощью тега <A>, как и гиперссылки. Только в данном случае атрибут HREF в нем присутствовать не должен. Вместо него в тег <A> помещают обязательный здесь атрибут ID, задающий уникальное в пределах текущей Web-страницы имя создаваемого якоря. К нему предъявляются те же требования, что и к имени карты (см. раздел, посвященный картам).

И еще. Мы уже знаем, что тег <A> парный и в случае гиперссылки в него помещают текст (или изображение), который этой самой гиперссылкой и станет. Когда создают якорь, в этот тег не помещают ничего (так называемый *пустой тег*). По крайней мере, так поступают чаще всего.

Листинг 6.4 иллюстрирует пример HTML-кода, создающего якорь.

Листинг 6.4

```
. . .
<P>Окончание второй главы...</P>
<A ID="chapter3"></A>
<P>Начало третьей главы...<P>
. . .
```

Здесь мы поместили якорь с именем chapter3 перед началом третьей главы нашего воображаемого документа.

Хорошо! Якорь готов. Как теперь на него сослаться с другой Web-страницы? Очень просто. Для этого достаточно создать обычную гиперссылку, добавив в ее интер-

нет-адрес имя нужного нам якоря. Имя якоря ставят в самый конец интернет-адреса и отделяют от него символом # ("решетка").

Предположим, что Web-страница, содержащая якорь `chapter3`, хранится в файле novel.htm. Тогда, чтобы сослаться на этот якорь с другой Web-страницы, мы создадим на последней такую гиперссылку:

```
<A HREF="novel.htm#chapter3">Глава 3</A>
```

При щелчке на такой гиперссылке Web-обозреватель откроет Web-страницу novel.htm и прокрутит ее в окне так, чтобы достичь места, где находится якорь `chapter3`.

Если же нам нужно сослаться на якорь с той же Web-страницы, где он находится, то можно использовать в качестве интернет-адреса только имя данного якоря, предварив его символом "решетки":

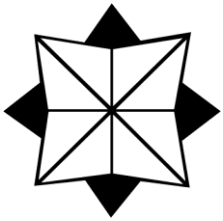
```
<A HREF="#chapter3">Глава 3</A>
```

На этом закончим главу о средствах навигации.

Что дальше?

В этой главе мы рассмотрели средства навигации, предлагаемые языком HTML, а именно — всевозможные гиперссылки. И связали все созданные нами к данному моменту Web-страницы в единый Web-сайт.

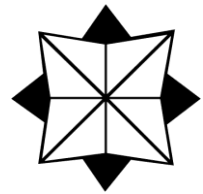
Пожалуй, можно завершить разговор о содержимом Web-страниц и языке HTML, на котором оно создается. Настала пора рассмотреть представление Web-страниц, о котором мы неоднократно упоминали. Именно этому будет посвящена вся следующая часть данной книги.



ЧАСТЬ II

Представление Web-страниц. Каскадные таблицы стилей CSS 3

ГЛАВА 7



Введение в стили CSS

Предыдущая часть книги была посвящена содержимому Web-страниц и языку HTML 5, на котором оно создается. Мы изучили немало новых терминов, использовали много тегов HTML и создали несколько Web-страниц нашего первого Web-сайта. Эти Web-страницы содержат большой объем текста, таблицу, графическое изображение, аудио- и видеоролик. Неплохо для начала.

Только вот выглядят эти Web-страницы как-то невзрачно. Однообразный текст, похожие друг на друга абзацы, таблицы без рамок, тоскливая черно-белая расцветка... Не помешает их как-то оформить. Вы так не считаете?..

За оформление Web-страниц и отдельных их элементов "отвечает" представление. Именно представление поможет нам оформить абзацы, таблицы и гиперссылки так, как хотим мы, а не Web-обозреватель (точнее, его разработчики). Именно представление поможет нам сделать Web-страницы привлекательными.

Мы много раз упоминали о представлении, предвкушая момент знакомства с ним. И момент настал! Вся эта часть книги будет посвящена представлению Web-страниц и технологиям, используемым для его создания.

А еще мы наконец-то займемся собственно Web-дизайном.

Понятие о стилях CSS

Для создания представления Web-страниц предназначена технология *каскадных таблиц стилей* (Cascading Style Sheets, CSS), или просто *таблиц стилей*. Таблица стилей содержит набор правил *(стилей)*, описывающих оформление самой Web-страницы и отдельных ее фрагментов. Эти правила определяют цвет текста и выравнивание абзаца, отступы между графическим изображением и обтекающим его текстом, наличие и параметры рамки у таблицы, цвет фона Web-страницы и многое другое.

Каждый стиль должен быть привязан к соответствующему элементу Web-страницы (или самой Web-странице). После привязки описываемые выбранным стилем параметры начинают применяться к данному элементу. Привязка может быть явная,

когда мы сами указываем, какой стиль к какому элементу Web-страницы привязан, или неявная, когда стиль автоматически привязывается ко всем элементам Web-страницы, созданным с помощью определенного тега.

Таблица стилей может храниться прямо в HTML-коде Web-страницы или в отдельном файле. Последний подход более соответствует концепции Web 2.0; как мы помним из *главы 1*, она требует, чтобы содержимое и представление Web-страницы были разделены. Кроме того, отдельные стили можно поместить прямо в тег HTML, создающий элемент Web-страницы; такой подход используется сейчас довольно редко и, в основном, при экспериментах со стилями.

Таблицы стилей пишут на особом языке, который так и называется — CSS. Стандарт, описывающий первую версию этого языка (CSS 1), появился еще в 1996 году. В настоящее время широко поддерживается и применяется на практике стандарт CSS 2 и ведется разработка стандарта CSS 3, ограниченное подмножество которого уже поддерживают многие Web-обозреватели.

Как раз CSS 3 (точнее, то его подмножество, поддерживаемое современными программами) мы и будем изучать.

Создание стилей CSS

Обычный формат определения стиля CSS иллюстрирует листинг 7.1.

Листинг 7.1

```
<селектор> {
  <атрибут стиля 1>: <значение 1>;
  <атрибут стиля 2>: <значение 2>;
  . . .
  <атрибут стиля n-1>: <значение n-1>;
  <атрибут стиля n>: <значение n>
}
```

Вот основные правила создания стиля.

- Определение стиля включает селектор и список атрибутов стиля с их значениями.
- *Селектор* используется для привязки стиля к элементу Web-страницы, на который он должен распространять свое действие. Фактически селектор однозначно идентифицирует данный стиль.
- За селектором, через пробел, указывают список атрибутов стиля и их значений, заключенный в фигурные скобки.
- *Атрибут стиля* (не путать с атрибутом тега!) представляет один из параметров элемента Web-страницы: цвет шрифта, выравнивание текста, величину отступа, толщину рамки и др. З*начение* атрибута стиля указывают после него через символ : (двоеточие). В некоторых случаях значение атрибута стиля заключают в кавычки.

- Пары *<атрибут стиля>*:*<значение>* отделяют друг от друга символом ; (точка с запятой).
- Между последней парой *<атрибут стиля>*:*<значение>* и закрывающей фигурной скобкой символ ; не ставят, иначе некоторые Web-обозреватели могут неправильно обработать определение стиля.
- Определения различных стилей разделяют пробелами или переводами строк.
- Внутри селекторов и имен стилей не должны присутствовать пробелы и переводы строки. Что касается пробелов и переводов строк, поставленных в других местах определения стиля, то Web-обозреватель их игнорирует. Поэтому мы можем форматировать CSS-код для удобства его чтения, как проделывали это с HTML-кодом.

Но правила — правилами, а главное — практика. Давайте рассмотрим пример небольшого стиля:

```
P { color: #0000FF }
```

Разберем его по частям.

- `P` — это селектор. Он представляет собой имя тега <P>.
- `color` — это атрибут стиля. Он задает цвет текста.
- `#0000FF` — это значение атрибута стиля `color`. Оно представляет код синего цвета, записанный в формате RGB. (Подробнее об RGB-коде и его задании мы поговорим в *главе 8*.)

Когда Web-обозреватель считает описанный стиль, он автоматически применит его ко всем абзацам Web-страницы (тегам <P>). Это, кстати, типичный пример неявной привязки стиля.

Стиль, который мы рассмотрели, называется *стилем переопределения тега*. В качестве селектора здесь указано имя переопределяемого этим стилем HTML-тега без символов < и >. Селектор можно набирать как прописными, так и строчными буквами; автор предпочитает прописные.

Рассмотрим еще пару таких стилей.

Вот стиль переопределения тега <EM>:

```
EM { color: #00FF00;
     font-weight: bold }
```

Любой текст, помещенный в тег <EM>, Web-обозреватель выведет зеленым полужирным шрифтом. Атрибут стиля `font-weight` задает степень "жирности" шрифта, а его значение `bold` — полужирный шрифт.

А это стиль переопределения тега <BODY>:

```
BODY { background-color: #000000;
       color: #FFFFFF }
```

Он задает для всей Web-страницы белый цвет текста (RGB-код `#FFFFFF`) и черный цвет фона (RGB-код `#000000`). Атрибут стиля `background-color`, как мы уже поняли, задает цвет фона.

А теперь рассмотрим совсем другой стиль:

```
.redtext { color: #FF0000 }
```

Он задает красный цвет текста (RGB-код `#FF0000`). Но в качестве селектора используется явно не имя тега — HTML-тега `<REDTEXT>` не существует.

Это другая разновидность стиля CSS — *стилевой класс*. Он может быть привязан к любому тегу. В качестве селектора здесь указывают *имя стилевого класса*, которое его однозначно идентифицирует. Имя стилевого класса должно состоять из букв латинского алфавита, цифр и дефисов, причем начинаться должно с буквы. В определении стилевого класса его имя обязательно предваряется символом точки — это признак того, что определяется именно стилевой класс.

Стилевой класс требует явной привязки к тегу. Для этого служит атрибут `CLASS`, поддерживаемый практически всеми тегами. В качестве значения этого атрибута указывают имя нужного стилевого класса без символа точки:

```
<P CLASS="redtext">Внимание!</P>
```

Здесь мы привязали только что созданный стилевой класс `redtext` к абзацу "Внимание!". В результате этот абзац будет набран красным шрифтом.

Листинг 7.2

```
.attention { color: #FF0000;
             font-style: italic }
. . .
<P><STRONG CLASS="attention">Стилевой класс требует явной привязки
атрибутом тега CLASS!</STRONG></P>
```

В листинге 7.2 мы создали стилевой класс `attention`, который задает красный цвет и курсивное начертание шрифта. (Атрибут стиля `font-style` задает начертание шрифта, а его значение `italic` как раз делает шрифт курсивным.) Затем мы привязали его к тегу `<STRONG>`. В результате содержимое этого тега будет набрано полужирным курсивным шрифтом красного цвета; особую важность и связанную с ним "полужирность" текста задает тег `<STRONG>`, а курсивное начертание и красный цвет — стилевой класс `attention`.

В качестве значения атрибута `CLASS` можно указать сразу несколько имен стилевых классов, разделив их пробелами. В таком случае действие стилевых классов как бы складывается. (Подробнее о действии на элемент Web-страницы нескольких разных стилей мы поговорим потом.)

Листинг 7.3

```
.attention { color: #FF0000;
             font-style: italic }
.bigtext   { font-size: large }
. . .
<P><STRONG CLASS="attention bigtext">Стилевой класс требует явной
привязки атрибутом тега CLASS!</STRONG></P>
```

В примере из листинга 7.3 мы привязали к тегу `<STRONG>` сразу два стилевых класса: `attention` и `bigtext`. В результате содержимое этого тега будет выведено полужирным курсивным шрифтом красного цвета и большого размера. (Атрибут стиля `font-size` указывает размер шрифта, а его значение `large` — большой размер, сравнимый с размером шрифта, которым выводятся заголовки первого уровня.)

Именованный стиль во многом похож на стилевой класс. Селектором этого стиля также является имя, которое его однозначно идентифицирует, и привязывается он к тегу также явно. А дальше начинаются отличия.

- В определении именованного стиля перед его именем ставят символ `#` ("решетка"). Он сообщает Web-обозревателю, что перед ним именованный стиль.
- Привязку именованного стиля к тегу реализуют через атрибут `ID`, также поддерживаемый практически всеми тегами. В качестве значения этого атрибута указывают имя нужного именованного стиля, уже без символа `#`.
- Значение атрибута тега `ID` должно быть уникальным в пределах Web-страницы. Говоря другими словами, в HTML-коде Web-страницы может присутствовать только один тег с заданным значением атрибута `ID`. Поэтому именованные стили используют, если какой-либо стиль следует привязать к одному-единственному элементу Web-страницы.

В примере

```
#bigtext { font-size: large }
. . .
<P ID="bigtext">Это большой текст.</P>
```

абзац "Это большой текст" будет набран крупным шрифтом.

Во всех рассмотренных нами разновидностях стилей был один селектор, с помощью которого и выполнялась привязка. Однако CSS позволяет создавать стили с несколькими селекторами — так называемые *комбинированные стили*. Такие стили служат для привязки к тегам, удовлетворяющим сразу нескольким условиям. Так, мы можем указать, что комбинированный стиль должен быть привязан к тегу, вложенному в другой тег, или к тегу, для которого указан определенный стилевой класс.

Правила, которые установлены стандартом CSS при написании селекторов в комбинированном стиле.

- Селекторами могут выступать имена тегов, имена стилевых классов и имена именованных стилей.
- Селекторы перечисляют слева направо и обозначают уровень вложенности соответствующих тегов, который увеличивается при движении слева направо: теги, указанные правее, должны быть вложены в теги, что стоят левее.
- Если имя тега скомбинировано с именем стилевого класса или именованного стиля, значит, для данного тега должно быть указано это имя стилевого класса или именованного стиля.

- Селекторы разделяют пробелами.
- Стиль привязывают к тегу, обозначенному самым правым селектором.

Мудреные правила, не так ли?.. Чтобы их понять, рассмотрим несколько примеров.

Начнем с самого простого комбинированного стиля:

```
P EM { color: #0000FF }
```

- В качестве селекторов использованы имена тегов <P> и <EM>.
- Сначала идет тег <P>, за ним — тег <EM>. Значит, тег <EM> должен быть вложен в тег <P>.
- Стиль будет привязан к тегу <EM>.

```
<P><EM>Этот текст</EM> станет синим.</P>
<P>А этот не станет.</P>
<P><STRONG>Этот</STRONG> — тоже.</P>
```

Здесь слова "Этот текст" будут набраны синим шрифтом.

Вот еще один комбинированный стиль:

```
P.mini { color: #FF0000;
         font-size: smaller }
```

Имя тега <P> скомбинировано с именем стилевого класса mini. Значит, данный стиль будет применен к любому тегу <P>, для которого указано имя стилевого класса mini. (Значение smaller атрибута стиля font-size задает уменьшенный размер шрифта.)

```
<P CLASS="mini">Маленький красный текстик.</P>
```

И последний пример комбинированного стиля:

```
P.sel <STRONG> { color: #FF0000 }
```

Этот стиль будет применен к тегу <STRONG>, находящемуся внутри тега <P>, к которому привязан стилевой класс sel.

```
<P CLASS="sel"><STRONG>Этот</STRONG> текст станет красным.</P>
```

В данном примере слово "Этот" будет выделено красным цветом.

Стандарт CSS позволяет определить сразу несколько одинаковых стилей, перечислив их селекторы через запятую:

```
H1, .redtext, P EM <STRONG> { color: #FF0000 }
```

Здесь мы создали сразу три одинаковых стиля: стиль переопределения тега <H1>, стилевой класс redtext и комбинированный стиль P EM. Все они задают красный цвет шрифта.

Все четыре рассмотренные нами разновидности стилей CSS могут присутствовать только в таблицах стилей. Если указать их в HTML-коде Web-страницы, они, скорее всего, будут проигнорированы.

Стили последней — пятой — разновидности указывают прямо в HTML-коде Web-страницы, в соответствующем теге. Это *встроенные стили*. В сплоченном семействе стилей они стоят особняком.

- Они не имеют селектора, т. к. ставятся прямо в нужный тег. Селектор в данном случае просто не нужен.
- В них отсутствуют фигурные скобки, поскольку нет нужды отделять список атрибутов стиля от селектора, которого нет.
- Встроенный стиль может быть привязан только к одному тегу — тому, в котором он находится.

Определение встроенного стиля указывают в качестве значения атрибута `STYLE` нужного тега, который поддерживается практически всеми тегами:

```
<P STYLE="font-size: smaller; font-style: italic">Маленький
↳курсивчик.</P>
```

Ранее мы упомянули, что в некоторых случаях значение атрибута стиля нужно заключать в кавычки. Но в определении встроенного стиля вместо кавычек используются апострофы.

Встроенные стили применяются сейчас довольно редко, т. к. нарушают требование концепции Web 2.0 разделять содержимое и представление Web-страниц. В основном их применяют для привязки стилей к одному-единственному элементу Web-страницы (очень редко) и во время экспериментов со стилями.

В *главе 14* мы рассмотрим еще одну разновидность стилей CSS. А пока что закончим с ними и приступим к рассмотрению таблиц стилей.

Таблицы стилей

Мы рассмотрели пять разновидностей стилей CSS. Четыре из них — стилевые классы, стили переопределения тега, именованные и комбинированные стили — могут присутствовать только в таблицах стилей. Это мы уже знаем.

Таблицы стилей, в зависимости от места их хранения, разделяются на два вида.

Внешние таблицы стилей хранятся отдельно от Web-страниц, в файлах с расширением css. Они содержат CSS-код определений стилей.

Листинг 7.4 иллюстрирует пример внешней таблицы стилей.

Листинг 7.4

```
.redtext { color: #FF0000 }
#bigtext { font-size: large }
EM { color: #00FF00;
     font-weight: bold }
P EM { color: #0000FF }
```

Как видим, здесь определены четыре стиля. В принципе, все нам знакомо.

Если внешняя таблица стилей хранится отдельно от Web-страницы, значит, нужно как-то привязать ее к Web-странице. Для этого предназначен одинарный метатег <LINK>, который помещается в секцию заголовка соответствующей Web-страницы. (О метатегах и секциях Web-страниц говорилось в *главе 1*.) Вот формат его написания:

```
<LINK REL="stylesheet" HREF="<интернет-адрес файла таблицы стилей>"
TYPE="text/css">
```

Интернет-адрес файла таблицы стилей записывают в качестве значения атрибута HREF этого тега.

Остальные атрибуты тега <LINK> для нас несущественны. Атрибут REL указывает, чем является файл, на который ссылается тег <LINK>, для текущей Web-страницы; его значение "stylesheet" говорит, что этот файл — внешняя таблица стилей. А атрибут TYPE указывает тип MIME файла, на который ссылается данный тег; внешняя таблица стилей имеет тип MIME text/css.

Листинг 7.5

```
<HEAD>
  . . .
  <LINK REL="stylesheet" HREF="main.css" TYPE="text/css">
  . . .
</HEAD>
```

В примере из листинга 7.5 мы привязали внешнюю таблицу стилей, хранящуюся в файле main.css, к текущей Web-странице.

Преимущество внешних таблиц стилей в том, что их можно привязать сразу к нескольким Web-страницам. Недостаток всего один, да и тот несущественный, — внешняя таблица стилей хранится в отдельном файле, так что есть вероятность его "потерять".

Внутренняя таблица стилей (листинг 7.6) записывается прямо в HTML-код Web-страницы. Ее заключают в парный тег <STYLE> и помещают в секцию заголовка. В остальном она не отличается от ее внешней "коллеги".

Листинг 7.6

```
<HEAD>
  . . .
  <STYLE>
    .redtext { color: #FF0000 }
    #bigtext { font-size: large }
    EM       { color: #00FF00;
               font-weight: bold }
    P EM     { color: #0000FF }
  </STYLE>
  . . .
</HEAD>
```

Преимущество внутренней таблицы стилей в том, что она является неотъемлемой частью Web-страницы и, стало быть, никогда не "потеряется". Недостатков два. Во-первых, стили, определенные во внутренней таблице стилей, применяются только к той Web-странице, в которой эта таблица стилей находится. Во-вторых, внутренняя таблица стилей не соответствует концепции Web 2.0, требующей отделять содержимое Web-страницы от ее представления.

В одной и той же Web-странице могут присутствовать сразу несколько таблиц стилей: несколько внешних и внутренняя (листинг 7.7).

Листинг 7.7

```
<HEAD>
  . . .
  <LINK REL="stylesheet" HREF="styles1.css" TYPE="text/css">
  <LINK REL="stylesheet" HREF="styles2.css" TYPE="text/css">
  . . .
  <STYLE>
    . . .
  </STYLE>
  . . .
</HEAD>
```

В таком случае действие всех этих таблиц стилей складывается. А по каким правилам — мы сейчас выясним.

Правила каскадности и приоритет стилей

Как мы уже выяснили, на один и тот же элемент Web-страницы могут действовать сразу несколько стилей. Это могут быть стили разных видов (стиль переопределения тега, стилевой класс, комбинированный стиль, встроенный стиль) или определенные в разных таблицах стилей (внешних и внутренней). Такое встречается сплошь и рядом, особенно на Web-страницах со сложным оформлением.

Но как Web-обозреватель определяет, какой именно стиль применить к тому или иному элементу Web-страницы? Мы уже знаем, что в таких случаях действие стилей как бы складывается. Но по каким правилам?

Предположим, что мы создали внешнюю таблицу стилей (листинг 7.8).

Листинг 7.8

```
.redtext { color: #FF0000 }
#bigtext { font-size: large }
EM       { color: #00FF00;
           font-weight: bold }
```

После этого мы изготовили Web-страницу, содержащую внутреннюю таблицу стилей (листинг 7.9).

Листинг 7.9

```
<STYLE>
  .redtext { color: #0000FF }
  EM { font-size: smaller }
</STYLE>
```

А в самой Web-странице написали вот такой фрагмент HTML-кода:

```
<P CLASS="redtext">Это красный текст.</P>
<P ID="bigtext" STYLE="color: #FFFF00">Это большой текст.<P>
<P><EM>Это курсив.</EM></P>
```

Хорошо видно, что на элементы этой Web-страницы действуют сразу по нескольку стилей. Так, во второй строке кода к тегу `<P>` привязаны и именованный стиль `bigtext`, и встроенный стиль. Но этого мало — и внешняя, и внутренняя таблицы стилей содержат определение двух одинаковых стилей — стилевого класса `redtext` и стиля переопределения тега `<EM>`!

Так что же мы получим в результате?

Рассмотрим сначала последнюю строку приведенного HTML-кода с тегом `<EM>`. Сначала Web-обозреватель загрузит, обработает и сохранит в памяти внешнюю таблицу стилей. Затем он обработает внутреннюю таблицу стилей и добавит все содержащиеся в ней определения стилей к уже хранящимся во внешней таблице стилей. Это значит, что стили переопределения тега `<EM>`, заданные в разных таблицах стилей, будут сложены, и результирующий стиль, написанный на языке CSS, станет таким:

```
EM { color: #00FF00;
     font-size: smaller;
     font-weight: bold }
```

Именно его и применит Web-обозреватель ко всем тегам `<EM>`, что присутствуют на Web-странице.

Вторая строка HTML-кода, что содержит тег со встроенным стилем, будет обработана так же. Web-обозреватель добавит к считанному из внешней таблицы стилей определению именованного стиля `bigtext` определение встроенного стиля. Результирующий стиль, если записать его на языке CSS, будет таким:

```
#bigtext { color: $FFFF00;
           font-size: large }
```

И, наконец, самая трудная задача — первая строка HTML-кода. Поскольку оба определения стилевого класса `redtext` задают один и тот же параметр — цвет текста (атрибут стиля `color`) — Web-обозреватель поступит так. Он отменит значение этого атрибута, заданное в стиле из внешней таблицы стилей, и заменит его тем, что задано в стиле из внутренней таблицы стилей. Поскольку, с его точки зрения и с точки зрения стандартов CSS, внутренняя таблица стилей — это та рубашка, что "ближе к телу". И тогда результирующий стиль будет таким:

```
.redtext { color: #0000FF }
```

Здесь мы столкнулись с тем, что стили разных видов и заданные в разных таблицах стилей имеют разный *приоритет*. Web-обозреватель руководствуется этим приоритетом, когда формирует в своей памяти окончательные определения стилей.

Теперь познакомимся с правилами, описывающими поведение Web-обозревателя при формировании окончательных стилей. Их еще называют *правилами каскадности*.

- Внешняя таблица стилей, ссылка на которую (тег <LINK>) встречается в HTML-коде страницы позже, имеет приоритет перед той, ссылка на которую встретилась раньше.
- Внутренняя таблица стилей имеет приоритет перед внешними.
- Встроенные стили имеют приоритет перед любыми стилями, заданными в таблицах стилей.
- Более конкретные стили имеют приоритет перед менее конкретными. Это значит, например, что стилевой класс имеет приоритет перед стилем переопределения тега, поскольку стилевой класс привязывается к конкретным тегам. Точно так же комбинированный стиль имеет приоритет перед стилевым классом.
- Если к тегу привязаны несколько стилевых классов, то те, что указаны правее, имеют приоритет перед указанными левее.

Важные атрибуты стилей

А теперь представим себе следующую ситуацию. Предположим, мы создали стили, приведенные в листинге 7.10.

Листинг 7.10

```
.redtext { color: #FF0000;
           font-weight: normal }
EM       { color: #00FF00;
           font-weight: bold }
```

Значение `normal` атрибута стиля `font-weight` задает обычную "жирность" шрифта, т. е. простой, светлый шрифт.

Далее мы поместили на Web-страницу вот такой абзац:

```
<P><EM CLASS="redtext">Это курсив.</EM></P>
```

Правила каскадности мы уже рассмотрели, так что можно сразу сказать, что получится в результате. Текст этого абзаца будет выведен обычным шрифтом красного цвета.

Но предположим, будто нам нужно, чтобы весь текст, выделенный тегом <EM>, в любом случае выводился полужирным шрифтом! Что делать? Создавать другой стилевой класс, специально для такого случая?

Совсем не обязательно. Стандарт CSS предоставляет нам замечательную возможность превратить отдельные атрибуты стиля в определении стиля в *важные*. Параметры, задаваемые важными атрибутами стиля, будут иметь приоритет над всеми аналогичными атрибутами стиля, заданными в других стилях, даже более конкретных. Фактически таким образом мы нарушим правила каскадности стандартными средствами CSS.

Обратим внимание, что важными можно сделать только отдельные атрибуты стиля в определении стиля. Атрибуты стиля, не объявленные важными, все так же будут подчиняться правилам каскадности.

Чтобы сделать атрибут стиля важным, достаточно после его значения через пробел поставить слово `!important` (пишется слитно, без пробелов между восклицательным знаком и словом "important"). Вот так:

```
EM       { color: #00FF00;
           font-weight: bold !important }
```

Теперь текст, выделенный тегом `<EM>`, всегда будет выводиться полужирным шрифтом, даже если данный параметр переопределен в более конкретном стиле.

Важные атрибуты стиля могут очень пригодиться при создании поведения Web-страницы, которое управляет стилями, привязанными к элементам Web-страницы, в ответ на действия посетителя. Мы столкнемся с этим уже в *главе 14*, когда будем создавать свое первое полезное поведение.

На этом рассмотрение принципов создания стилей и таблиц стилей можно закончить. Осталось только поговорить о том...

Какие стили в каких случаях применять

Удачно подобранный набор стилей — результат долгих экспериментов. Нам придется изрядно повозиться, прежде чем мы его получим. Но несколько правил, приведенных далее, помогут нам получить его заметно быстрее.

Прежде всего, следует выделить все правила оформления, которые должны быть применены ко всем Web-страницам и их элементам. К таким правилам относятся параметры шрифта обычных абзацев и заголовков, цвет фона Web-страницы, выравнивание текста, величины отступов и параметры рамки обычных таблиц и пр. В общем, то, что определяет вид всех Web-страниц.

Эти правила преобразуются в общие стили, обычно стили переопределения тегов. Их помещают в одну внешнюю таблицу стилей и привязывают ее ко всем Web-страницам Web-сайта. Такая таблица стилей называется *глобальной*.

Далее выделяют правила оформления, которые должны применяться к некоторым элементам Web-страниц. Это могут быть параметры шрифта каких-то избранных абзацев (например, предупреждений о чем-то), их фрагментов, параметры гиперссылок, входящих в полосу навигации, и др. Данные правила формируют так, чтобы дополнять полученные ранее, но не перекрывать их полностью, — это позволит сократить размер CSS-кода таблиц стилей.

Так мы получим более конкретные стили, дополняющие созданные ранее. Их мы оформим в виде стилевых классов и комбинированных стилей и поместим все в ту же глобальную таблицу стилей. Также можно поместить их в отдельную таблицу стилей *(вторичную)*, которую привязать только к тем Web-страницам, где они используются; этот подход оправдан только если таких стилей получается слишком много, и нет резона "раздувать" глобальную таблицу стилей.

На следующем этапе мы выделим правила, применяемые только к одному элементу Web-страниц. Это могут быть параметры полосы навигации, заголовка Web-сайта, различных контейнеров, определяющих дизайн Web-страниц (о контейнерах речь пойдет в *главе 10*). Они также должны дополнять правила, полученные на предыдущих этапах, но не заменять их полностью.

Эти еще более конкретные стили мы оформим в виде именованных стилей — в данном случае это будет наилучшим выбором. И поместим их в глобальную таблицу стилей. Если их получается слишком много, мы также можем поместить их во вторичную таблицу стилей и привязать ее ко всем Web-страницам.

Последний этап — выделение правил, применяемых к совсем небольшому количеству элементов, которые присутствуют только на отдельных Web-страницах, а то и вообще на одной из них. К таким правилам можно отнести параметры очень специфических абзацев, отдельных иллюстраций и таблиц. Не забываем, что и эти правила, по возможности, не должны полностью перекрывать полученные ранее.

Так мы выделим самые конкретные стили. Их можно оформить в виде стилевых классов или именованных стилей. Помещаться они могут в глобальную таблицу стилей или во вторичную таблицу стилей, привязанную только к тем Web-страницам, где должны использоваться.

Что касается внутренних таблиц стилей и встроенных стилей, то в готовых Web-страницах от них рекомендуется отказаться. Как мы помним еще из *главы 1*, концепция Web 2.0 настоятельно рекомендует разделять содержимое и представление Web-страниц. А внутренние таблицы стилей и встроенные стили нарушают данное правило — о чем мы неоднократно говорили.

Сейчас внутренние таблицы стилей и встроенные стили применяют, в основном, во время экспериментов, чтобы выяснить, как тот или иной стиль влияет на отображение элементов Web-страницы. По завершении экспериментов готовые стили переносят во внешние таблицы стилей и соответствующим образом оформляют.

Так или иначе, конкретные примеры выбора нужной разновидности стиля мы рассмотрим в последующих главах, когда начнем создавать представление для наших Web-страниц. В конце концов, учиться лучше всего на примерах.

Комментарии CSS

В *главе 2* мы узнали о комментариях — особых фрагментах HTML-кода, которые не обрабатываются Web-обозревателем и служат для того, чтобы Web-дизайнер смог оставить какие-то заметки для себя или своих коллег. Для этого язык HTML предоставляет специальный тег.

Создавать комментарии позволяет и язык CSS. Более того, с его помощью мы можем формировать комментарии двух видов.

Комментарий, состоящий из одной строки, создают с помощью символа / (слэш) в самом начале строки, которая станет комментарием:

```
/ Это комментарий
P { color: #0000FF }
```

Однострочный комментарий начинается с символа / и заканчивается концом строки.

Комментарий, состоящий из произвольного числа строк, создают с помощью последовательностей символов /* и */. Между ними помещают строки, которые станут комментарием (листинг 7.11).

Листинг 7.11

```
/*
  Это комментарий,
  состоящий из
  нескольких строк.
*/
P { color: #0000FF }
```

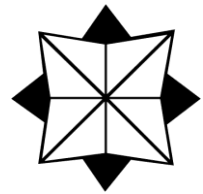
Многострочный комментарий начинается с последовательности символов /* и заканчивается последовательностью */.

Что дальше?

В этой главе мы получили понятие о стилях, таблицах стилей и используемом для их создания языке CSS. Мы разобрались с правилами каскадности и выяснили, какие стили в каких случаях следует применять.

В следующей главе мы приступим к изучению языка CSS. Мы начнем с самых простых атрибутов стилей — тех, что задают правила оформления шрифта. И наконец-то начнем создавать представление наших Web-страниц.

ГЛАВА 8



Параметры шрифта и фона. Контейнеры

В предыдущей главе мы познакомились со стилями и таблицами стилей CSS, с помощью которых создается представление Web-страниц. Мы изучили четыре разновидности стилей и две разновидности таблиц стилей и выяснили, как их правильно применять.

В этой главе мы начнем изучать возможности языка CSS. Сначала рассмотрим атрибуты стилей, задающие параметры шрифта и фона элементов Web-страниц. Затем мы изучим новый элемент Web-страницы — встроенный контейнер — и соответствующий ему HTML-тег, который нам в дальнейшем весьма пригодится.

Заметим сразу, что все атрибуты стиля, описанные в этой главе, применимы к любым элементам Web-страниц — и блочным, и встроенным. Есть, правда, одно исключение, о котором мы предупредим особо.

Параметры шрифта

Начнем с атрибутов стиля, задающих параметры шрифта, которым набран текст. Ведь текст на Web-страницах — всему голова.

Атрибут стиля `font-family` задает имя шрифта, которым будет выведен текст:

`font-family: <`*`список имен шрифтов, разделенных запятыми`*`>|inherit`

Имена шрифтов задаются в виде их названий, например, `Arial` или `Times New Roman`. Если имя шрифта содержит пробелы, его нужно взять в кавычки:

```
P { font-family: Arial }
H1 ( font-family: "Times New Roman" }
```

Если данный атрибут стиля присутствует во встроенном стиле, кавычки заменяют апострофами:

```
<P STYLE="font-family: 'Times New Roman'">
```

Если указанный нами шрифт присутствует на компьютере посетителя, Web-обозреватель его использует. Если же такого шрифта нет, то текст выводится шрифтом, заданным в настройках по умолчанию. И наша Web-страница, возможно,

будет выглядеть не так, как мы задумывали. (Впрочем, шрифты Arial и Times New Roman присутствуют на любом компьютере, работающем под управлением Windows.)

Можно указать несколько наименований шрифтов через запятую:

```
P { font-family: Verdana, Arial }
```

Тогда Web-обозреватель сначала будет искать первый из указанных шрифтов, в случае неудачного поиска — второй, потом третий и т. д.

Вместо имени конкретного шрифта можно задать имя одного из *семейств шрифтов*, представляющих целые наборы аналогичных шрифтов. Таких семейств пять: `serif` (шрифты с засечками), `sans-serif` (шрифты без засечек), `cursive` (шрифты, имитирующие рукописный текст), `fantasy` (декоративные шрифты) и `monospace` (моноширинные шрифты):

```
H2 { font-family: Verdana, Arial, sans-serif }
```

Особое значение `inherit` указывает, что текст данного элемента Web-страницы должен быть набран тем же шрифтом, что и текст родительского элемента. Говорят, что в данном случае элемент Web-страницы "наследует" шрифт от родительского элемента. Это, кстати, значение атрибута стиля `font-family` по умолчанию.

Атрибут стиля `font-size` определяет размер шрифта:

```
font-size: <размер>|xx-small|x-small|small|medium|large|x-large|
↳xx-large|larger|smaller|inherit
```

Размер шрифта можно задать в абсолютных и относительных величинах. Для этого используется одна из *единиц измерения*, поддерживаемая CSS (табл. 8.1).

Таблица 8.1. *Единицы измерения размера, поддерживаемые стандартом CSS*

Название	Обозначение в CSS
Пикселы	px
Пункты	pt
Дюймы	in
Сантиметры	cm
Миллиметры	mm
Пики	pc
Размер буквы "m" текущего шрифта	em
Размер буквы "x" текущего шрифта	ex
Проценты от размера шрифта родительского элемента	%

Обозначение выбранной единицы измерения указывают после самого значения:

```
P { font-size: 10pt }
STRONG { font-size: 1cm }
EM { font-size: 150% }
```

Отметим, что все приведенные в табл. 8.1 единицы измерения подходят для задания значений других атрибутов стилей CSS.

Кроме числовых, атрибут `font-size` может принимать и символьные значения. Так, значения от `xx-small` до `xx-large` задают семь предопределенных размеров шрифта, от самого маленького до самого большого. А значения `larger` и `smaller` представляют следующий размер шрифта, соответственно, по возрастанию и убыванию. Например, если для родительского элемента определен шрифт размера `medium`, то значение `larger` установит для текущего элемента размер шрифта `large`.

Значение `inherit` указывает, что данный элемент Web-страницы должен иметь тот же размер шрифта, что и родительский элемент. Это значение атрибута стиля `font-size` по умолчанию.

Атрибут стиля `color` задает цвет текста:

`color: <`*`цвет`*`>|inherit`

В *главе 7* мы упоминали, что цвет можно задать так называемым RGB-кодом (Red, Green, Blue — красный, зеленый, синий). Он записывается в формате

`#<`*`доля красного цвета`*`><`*`доля зеленого цвета`*`><`*`доля синего цвета`*`>`,

где доли всех цветов указаны в виде шестнадцатеричных чисел от 00 до FF.

Зададим для текста красный цвет:

```
H1 { color: #FF0000 }
```

А теперь серый цвет:

```
ADDRESS { color: #CCCCCC }
```

Кроме того, CSS позволяет задавать цвета по именам. Так, значение `black` соответствует черному цвету, `white` — белому, `red` — красному, `green` — зеленому, а `blue` — синему.

Пример:

```
H1 { color: red }
```

Полный список имен и соответствующих им цветов можно посмотреть на Web-странице **http://msdn.microsoft.com/en-us/library/aa358802%28v=VS.85%29.aspx**.

Значение `inherit` указывает, что данный элемент Web-страницы должен иметь тот же цвет шрифта, что и родительский элемент. Это значение атрибута стиля `font-size` по умолчанию.

> ***Внимание!***
>
> Значение `inherit` поддерживают практически все атрибуты стиля CSS. Оно говорит Web-обозревателю, что элемент Web-страницы, к которому привязан стиль, "наследует" значение соответствующего параметра у родительского элемента. У всех атрибутов это значение по умолчанию. В дальнейшем мы не будем описывать данное значение у каждого атрибута стиля; если же какой-то атрибут стиля не поддерживает его, мы специально об этом упомянем.

С помощью атрибута стиля `color` мы можем, например, задать цвет горизонтальной линии HTML.

Атрибут стиля `opacity` позволяет указать степень полупрозрачности элемента Web-страницы:

```
opacity: <числовое значение>|inherit
```

Значение полупрозрачности представляет собой число от 0 до 1. При этом 0 обозначает полную прозрачность элемента (т. е. элемент фактически не виден), а 1 — полную непрозрачность (это обычное поведение).

Вот пример задания половинной прозрачности (значение 0,5) для текста фиксированного форматирования:

```
PRE { opacity: 0.5 }
```

Отметим, как мы указали дробное число — вместо символа запятой здесь используется точка.

> ***На заметку***
>
> Полупрозрачность обычно целесообразна только для создания специальных эффектов. В обычном тексте применять ее не рекомендуется, т. к. это может обескуражить посетителя.

Атрибут стиля `font-weight` устанавливает "жирность" шрифта:

```
font-weight: normal|bold|bolder|lighter|100|200|300|400|500|600|
↳700|800|900|inherit
```

Здесь доступны семь абсолютных значений от `100` до `900`, представляющих различную "жирность" шрифта, от минимальной до максимальной; при этом обычный шрифт будет иметь "жирность" `400` (или `normal`), а полужирный — `700` (или `bold`). Значение по умолчанию — `400` (`normal`). Значения `bolder` и `lighter` являются относительными и представляют следующие степени "жирности" соответственно в бóльшую и меньшую сторону.

Пример:

```
CODE { font-weight: bold }
```

Атрибут `font-style` задает начертание шрифта:

```
font-style: normal|italic|oblique|inherit
```

Доступны три значения, представляющие обычный шрифт (`normal`), курсив (`italic`) и особое декоративное начертание, похожее на курсив (`oblique`).

Атрибут стиля `text-decoration` задает "украшение" (подчеркивание или зачеркивание), которое будет применено к тексту:

```
text-decoration: none|underline|overline|line-through|blink|inherit
```

Здесь доступны пять значений (не считая `inherit`):

- `none` убирает все "украшения", заданные для шрифта родительского элемента;
- `underline` подчеркивает текст;

- `overline` "надчеркивает" текст, т. е. проводит линию над строками;
- `line-through` зачеркивает текст;
- `blink` делает шрифт мерцающим (на данный момент не поддерживается Safari).

> ***ВНИМАНИЕ!***
>
> Не следует без особой необходимости задавать для текста подчеркивание. Дело в том, что Web-обозреватели по умолчанию выводят гиперссылки подчеркнутыми, и подчеркнутый текст, не являющийся гиперссылкой, может обескуражить посетителя.

Атрибут стиля `font-variant` позволяет указать, как будут выглядеть строчные буквы шрифта:

```
font-variant: normal|small-caps|inherit
```

Значение `small-caps` задает такое поведение шрифта, когда его строчные буквы выглядят точно так же, как прописные, просто имеют меньший размер. Значение `normal` задает для шрифта обычные прописные буквы.

Атрибут стиля `text-transform` позволяет изменить регистр символов текста:

```
text-transform: capitalize|uppercase|lowercase|none|inherit
```

Мы можем преобразовать текст к верхнему (значение `uppercase` этого атрибута) или нижнему (`lowercase`) регистру, преобразовать к верхнему регистру первую букву каждого слова (`capitalize`) или оставить в изначальном виде (`none`).

Атрибут стиля `line-height` задает высоту строки текста:

```
line-height: normal|<расстояние>|inherit
```

Здесь можно задать абсолютную и относительную величину расстояния, указав соответствующую единицу измерения CSS (см. табл. 8.1). При ее отсутствии заданное нами значение сначала умножается на высоту текущего шрифта и затем используется. Таким образом, чтобы увеличить высоту строки вдвое по сравнению с обычной, мы можем написать:

```
P { line-height: 2 }
```

Значение `normal` этого атрибута возвращает управление высотой строки Web-обозревателю.

Атрибут стиля `letter-spacing` позволяет задать дополнительное расстояние между символами текста:

```
letter-spacing: normal|<расстояние>
```

Отметим, что это именно дополнительное расстояние; оно будет добавлено к изначальному, установленному самим Web-обозревателем.

Здесь также можно задать абсолютное и относительное расстояние, указав соответствующую единицу измерения CSS (см. табл. 8.1). Расстояние может быть положительным и отрицательным; в последнем случае символы шрифта будут располагаться друг к другу ближе обычного. Значение `normal` устанавливает дополнительное расстояние по умолчанию, равное нулю.

Атрибут стиля `letter-spacing` не поддерживает значение `inherit`.

Вот пример задания дополнительного расстояния между символами равного пяти пикселам:

```
H1 { letter-spacing: 5px }
```

Текст, набранный такими символами, будет выглядеть разреженным.

А здесь мы задали отрицательное дополнительное расстояние между символами равным двум пикселам:

```
H6 { letter-spacing: -2px }
```

Эти два пиксела будут вычтены из изначального расстояния, в результате символы сблизятся, а текст станет выглядеть сжатым. Возможно, символы даже налезут друг на друга.

Аналогичный атрибут стиля `word-spacing` задает дополнительное расстояние между отдельными словами текста:

```
word-spacing: normal|<расстояние>
```

Пример:

```
H1 { word-spacing: 5mm }
```

Ну, 5 мм, пожалуй, многовато... Хотя это всего лишь пример.

И напоследок рассмотрим атрибут стиля `font`, позволяющий задать одновременно сразу несколько параметров шрифта:

```
font: [<начертание>] [<вид строчных букв>] [<"жирность">]
[<размер>[/<высота строки текста>]] <имя шрифта>
```

Как видим, обязательным является только имя шрифта — остальные параметры могут отсутствовать.

Задаем для текста абзацев шрифт Times New Roman размером 10 пунктов:

```
P { font: 10pt "Times New Roman" }
```

А для заголовков шестого уровня — шрифт Arial размером 12 пунктов и курсивного начертания:

```
H6 { font: italic 12pt Verdana }
```

Параметры, управляющие разрывом строк

По умолчанию Web-обозреватель разбивает текст на строки так, чтобы вместить его в окно и избежать горизонтальной прокрутки. Далеко не всегда при этом он разрывает строки, как нам нужно. Конечно, мы можем установить фиксированное форматирование текста (см. *главу 2*) и принудительно указать, где следует переносить строки, но не всегда это лучший подход.

CSS предлагает два атрибута стиля, позволяющие нам указать, как Web-обозревателю следует разбивать текст на строки. Сейчас мы их рассмотрим.

Атрибут стиля `white-space` задает правила, которыми Web-обозреватель руководствуется при выводе текста. В частности, с его помощью мы можем изменить правила, установленные по умолчанию и рассмотренные нами в *главе 2*. Формат записи этого атрибута стиля:

```
white-space: normal|pre|nowrap|pre-wrap|pre-line|inherit
```

Атрибут стиля `white-space` может иметь пять значений.

- `normal` — последовательности пробелов преобразуются в один пробел, переводы строк также преобразуются в пробелы. Web-обозреватель сам разрывает текст на строки, чтобы вместить его в свое окно по ширине. Фактически это значение предписывает Web-обозревателю применять для вывода текста блочных элементов правила по умолчанию, описанные в *главе 2*.
- `pre` — последовательности пробелов и переводы строк сохраняются; текст выводится точно в таком виде, в каком он записан в HTML-коде. Web-обозреватель сам не разрывает текст на строки. Фактически текст выводится так, словно он помещен в тег `<PRE>` (текст фиксированного форматирования).
- `nowrap` — последовательности пробелов преобразуются в один пробел, переводы строк также преобразуются в пробелы. Однако Web-обозреватель сам не разрывает текст на строки.
- `pre-wrap` — последовательности пробелов и переводы строк сохраняются. Web-обозреватель может разорвать слишком длинные строки, чтобы избежать горизонтальной прокрутки.
- `pre-line` — последовательности пробелов преобразуются в один пробел, переводы строк сохраняются. Web-обозреватель может разорвать слишком длинные строки, чтобы избежать горизонтальной прокрутки.

Чтобы читателям было проще выбрать нужное значение атрибута стиля `white-space`, автор свел все доступные для него значения в табл. 8.2.

Таблица 8.2. *Значения атрибута стиля* `white-space` *и результаты их применения*

Значение	Переводы строк	Последовательности пробелов	Разрыв текста на строки
`normal`	Преобразуются в пробелы	Преобразуются в один пробел	Выполняется
`pre`	Сохраняются	Сохраняются	Не выполняется
`nowrap`	Преобразуются в пробелы	Преобразуются в один пробел	Не выполняется
`pre-wrap`	Сохраняются	Сохраняются	Выполняется
`pre-line`	Преобразуются в пробелы	Сохраняются	Выполняется

Вот стиль, переопределяющий тег `<PRE>` так, чтобы при необходимости его содержимое разрывалось на строки:

```
PRE { white-space: pre-wrap }
```

Атрибут стиля `word-wrap` применяется нечасто, но в некоторых случаях без него не обойтись. Он позволяет указать места, в которых Web-обозреватель может выполнить разрыв текста:

```
word-wrap: normal|break-word|inherit
```

Здесь доступны два значения.

- `normal` — указывает Web-обозревателю, что он может разрывать текст на строки только по пробелам. Это обычное поведение Web-обозревателя.
- `break-word` — разрешает Web-обозревателю выполнять разрыв текста на строки внутри слов. Это может пригодиться, если текст содержит много очень длинных слов, которые по ширине не помещаются в родительский элемент.

Пример:

```
P ( word-wrap: break-word }
```

Здесь мы разрешили Web-обозревателю выполнять разрыв текста на строки внутри слов в содержимом тегов `<P>` (т. е. в абзацах).

Параметры вертикального выравнивания

Иногда возникает ситуация, когда нужно сместить фрагмент по вертикали относительно текста, который его окружает, т. е. задать вертикальное выравнивание текста.

Атрибут стиля `vertical-align` как раз задает вертикальное выравнивание текста:

```
vertical-align: baseline|sub|super|top|text-top|middle|bottom|
text-bottom|<промежуток между базовыми линиями>|inherit
```

Этот атрибут стиля принимает восемь предопределенных значений:

- `baseline` — задает выравнивание базовой линии фрагмента текста по базовой линии текста родительского элемента (это поведение по умолчанию). *Базовой* называется воображаемая линия, на которой располагается текст.
- `sub` — выравнивает базовую линию фрагмента текста по базовой линии нижнего индекса родительского элемента.
- `super` — выравнивает базовую линию фрагмента текста по базовой линии верхнего индекса родительского элемента.
- `top` — выравнивает верхний край фрагмента текста по верхнему краю родительского элемента.
- `text-top` — выравнивает верхний край фрагмента текста по верхнему краю текста родительского элемента.
- `middle` — выравнивает центр фрагмента текста по центру родительского элемента.
- `bottom` — выравнивает нижний край фрагмента текста по нижнему краю родительского элемента.

- `text-bottom` — выравнивает нижний край фрагмента текста по нижнему краю текста родительского элемента.

Кроме того, мы можем указать для данного атрибута стиля абсолютное или относительное значение, задающее, насколько выше или ниже базовой линии текста родительского элемента должна находиться базовая линия фрагмента текста:

```
STRONG { vertical-align: super;
         font-size: smaller }
```

Этот стиль переопределения тега `<STRONG>` задает для текста расположение, совпадающее с базовой линией верхнего индекса, и уменьшенный размер шрифта. Фактически с помощью этого стиля мы превращаем содержимое тега в верхний индекс `<STRONG>`.

Тот же атрибут стиля пригоден для выравнивания графических изображений, являющихся частью абзаца:

```
<P>Это картинка: <IMG STYLE="vertical-align: text-bottom"
SRC="picture.png">.</P>
```

Данный HTML-код создает абзац с графическим изображением. Низ этого изображения будет выровнен по нижнему краю текста абзаца. Иными словами, изображение будет как бы возвышаться над текстом.

Скорее всего, для достижения нужного результата придется поэкспериментировать с различными значениями атрибута стиля `vertical-align`. Очень уж много у него возможных значений, и слишком разный они дают результат в различных случаях. Но ведь Web-дизайнеру не привыкать к экспериментам!..

Параметры тени у текста

Любителям все украшать стандарт CSS 3 предлагает одну очень интересную возможность — создание тени у текста. При умеренном употреблении она может заметно оживить Web-страницу.

Параметры тени задает атрибут стиля `text-shadow`:

```
text-shadow: none | <цвет> <горизонтальное смещение>
<вертикальное смещение> [<радиус размытия>]
```

Значение `none` (установленное по умолчанию) убирает тень у текста.

`Цвет` тени задается в виде RGB-кода или именованного значения.

`Горизонтальное смещение` тени задается в любой единице измерения, поддерживаемой CSS (см. табл. 8.1). Если задано положительное смещение, тень будет расположена правее текста, если отрицательное — левее. Можно также задать и нулевое смещение; тогда тень будет располагаться прямо под текстом. Нулевое смещение имеет смысл только в том случае, если для тени задано размытие.

`Вертикальное смещение` тени также задается в любой единице измерения, поддерживаемой CSS. Если задано положительное смещение, тень будет расположена ниже

текста, если отрицательное — выше. Можно также задать и нулевое смещение; тогда тень будет располагаться прямо под текстом.

Радиус размытия тени также задается в любой единице измерения, поддерживаемой CSS. Если радиус размытия не указан, его значение предполагается равным нулю; в таком случае тень не будет иметь эффекта размытия.

Пример:

```
H1 { text-shadow: black 1mm 1mm 1px }
```

Здесь мы задали для заголовков первого уровня (тега <H1>) тень, расположенную правее и ниже текста на 1 мм и имеющую радиус размытия 1 пиксел.

Параметры фона

Закончив с параметрами текста, займемся фоном. Фон можно указать для фрагмента текста (встроенного элемента), блочного элемента, таблицы, ее ячейки и всей Web-страницы. Хорошо подобранный фон может оживить Web-страницу и выделить отдельные ее элементы.

> ***Внимание!***
>
> Фон у отдельных элементов, отличный от фона самой Web-страницы, следует задавать только в крайних случаях. Иначе Web-страница станет слишком пестрой и неудобной для чтения.

Атрибут стиля background-color служит для задания цвета фона:

```
background-color: transparent|<цвет>|inherit
```

Цвет можно задать в виде RGB-кода или имени. Значение transparent убирает фон совсем; тогда элемент Web-страницы получит "прозрачный" фон. По умолчанию фон у элементов Web-страницы отсутствует, а фон самой Web-страницы задает Web-обозреватель.

Пример:

```
BODY { color: white;
       background-color: black }
```

Здесь мы задали для всей Web-страницы черный фон и белый текст.

Атрибут стиля background-image позволяет назначить в качестве фона графическое изображение *(фоновое изображение)*:

```
background-image: none|url(<интернет-адрес файла изображения>);
```

Обратим внимание, в каком виде задается интернет-адрес файла с фоновым изображением: его заключают в скобки, а перед ними ставят символы url:

```
TABLE.bgr { background-image: url("/table_background.png") }
```

Значение none убирает графический фон.

Графический фон выводится поверх обычного фона, заданного нами с помощью атрибута стиля background-color. И, если фоновое изображение содержит "про-

зрачные" фрагменты (такую возможность поддерживают форматы GIF и PNG), этот фон будет "просвечивать" сквозь них.

Пример:

```
TABLE.yellow { background-color: yellow;
               background-image: url("/yellow_background.png") }
```

Здесь мы задали для таблицы и обычный, и графический фон. Это, кстати, распространенная практика в Web-дизайне.

Если фоновое изображение меньше, чем элемент Web-страницы (или сама Web-страница), для которого оно задано, Web-обозреватель будет повторять это изображение, пока не "замостит" им весь элемент. Параметры этого повторения задает атрибут стиля `background-repeat`:

```
background-repeat: no-repeat|repeat|repeat-x|repeat-y|inherit
```

Здесь доступны четыре значения.

- `no-repeat` — фоновое изображение не будет повторяться никогда; в этом случае часть фона элемента Web-страницы останется не заполненной им.
- `repeat` — фоновое изображение будет повторяться по горизонтали и вертикали (обычное поведение).
- `repeat-x` — фоновое изображение будет повторяться только по горизонтали.
- `repeat-y` — фоновое изображение будет повторяться только по вертикали.

С помощью атрибута стиля `background-position` можно указать позицию фонового изображения относительно элемента Web-страницы, для которого оно назначено:

```
background-position: <горизонтальная позиция> [<вертикальная позиция>] |
inherit;
```

Горизонтальная позиция фонового изображения задается в следующем формате:

```
<числовое значение>|left|center|right
```

Числовое значение указывает местоположение фонового изображения в элементе Web-страницы по горизонтали и может быть задано с применением любой из поддерживаемых CSS единиц измерения (см. табл. 8.1). Также можно указать следующие значения:

- `left` — фоновое изображение прижимается к левому краю элемента Web-страницы (это обычное поведение);
- `center` — располагается по центру;
- `right` — прижимается к правому краю.

Формат задания *вертикальной позиции* фонового изображения таков:

```
<числовое значение>|top|center|bottom
```

Числовое значение указывает местоположение фонового изображения в элементе Web-страницы по вертикали и может быть задано с применением любой из поддерживаемых CSS единиц измерения.

Также возможны следующие значения:

- `top` — фоновое изображение прижимается к верхнему краю элемента Web-страницы (это обычное поведение);
- `center` — располагается по центру;
- `bottom` — прижимается к нижнему краю.

Если для какого-либо элемента Web-страницы указана только позиция фонового изображения по горизонтали, его вертикальная позиция принимается равной `center`.

Пример:

```
TABLE.bgr ( background-position: 1cm top }
```

Этот стиль помещает фоновое изображение на расстоянии 1 см от левого края элемента Web-страницы и прижимает его к нижнему краю данного элемента.

А вот стиль, прижимающий фоновое изображение к правому краю элемента Web-страницы и располагающий его в центре данного элемента по вертикали:

```
TABLE.bgr ( background-position: right }
```

Когда мы прокручиваем содержимое Web-страницы в окне Web-обозревателя, вместе с ней прокручивается и фоновое изображение (если оно есть). Стандарт CSS предлагает забавную возможность — запрет прокрутки графического фона Web-страницы и фиксация его на месте. Фиксацией фона управляет атрибут стиля `background-attachment`:

```
background-attachment: scroll|fixed;
```

Значение `scroll` заставляет Web-обозреватель прокручивать фон вместе с содержимым Web-страницы (это поведение по умолчанию). Значение `fixed` фиксирует фон на месте, и он не будет прокручиваться.

> ***НА ЗАМЕТКУ***
>
> Вероятно, имеет смысл фиксировать только графический фон, заданный для самой Web-страницы. Графический фон у отдельных элементов Web-страницы фиксировать не следует.

На этом мы пока закончим с атрибутами стиля CSS. Рассмотрим новую разновидность элементов Web-страниц, с которой еще не сталкивались. Это...

Контейнеры. Встроенные контейнеры

В самом начале данной главы мы узнали, что все рассмотренные нами атрибуты стилей можно указывать для любых элементов Web-страниц: и блочных, и встроенных. Значит, мы можем задать размер шрифта и для абзаца (блочного тега `<P>`), и для важного текста (встроенных тегов `<STRONG>` и `<EM>`). Это очень полезная возможность.

Но что делать, если нам понадобилось применить какой-либо стиль к произвольному фрагменту текста, не помечая его никаким тегом? Например, нам нужно выде-

лить полужирным шрифтом фрагмент абзаца, но мы не хотим заключать его в тег <STRONG>. Может ли CSS нам в этом помочь?

CSS не может. Зато может HTML. Он специально для таких случаев предоставляет особые элементы Web-страницы — контейнеры — и, конечно, соответствующие теги. О контейнерах сейчас и пойдет разговор.

Контейнер — элемент Web-страницы, предназначенный только для выделения какого-либо ее фрагмента. Таким фрагментом может быть часть блочного элемента (абзаца, заголовка, цитаты, текста фиксированного форматирования и др.), блочный элемент или сразу несколько блочных элементов. Web-обозреватель никак не выделяет контейнер на Web-странице.

Контейнер служит двум целям. Во-первых, с его помощью мы можем привязать к определенному элементу или элементам Web-страницы нужный стиль; для этого достаточно заключить данный элемент или элементы в контейнер и привязать стиль к нему. Во-вторых, он может обеспечивать привязку поведения к элементу или элементам Web-страницы; выполняется это таким же образом, что и в случае стиля. (О поведении Web-страницы разговор пойдет в *части III*.)

Контейнеры бывают блочные и встроенные. Разговор о блочных контейнерах мы отложим до *главы 10*, в которой будем рассматривать контейнерный Web-дизайн. Поговорим о встроенных контейнерах.

Уже по определению ясно, что *встроенный контейнер* является частью блочного элемента Web-страницы. Так, блочным контейнером может стать фрагмент абзаца или цитаты, графическое изображение, помещенное в абзац, и др.

Встроенный контейнер создается с помощью парного тега <SPAN>. Фрагмент блочного элемента, который нужно превратить в содержимое встроенного контейнера, помещают в этот тег:

```
<P><SPAN>Представление</SPAN> создается с помощью стилей CSS.</P>
```

Здесь мы поместили во встроенный контейнер фрагмент абзаца.

Толку от нашего первого встроенного контейнера никакого. Поэтому давайте привяжем к нему какой-нибудь стиль (листинг 8.1).

Листинг 8.1

```
.bolded { font-weight: bold }
. . .
<P><SPAN CLASS="bolded">Представление</SPAN> создается с помощью стилей
CSS.</P>
```

Теперь слово "Представление" будет набрано полужирным шрифтом.

Как правило, потребность во встроенных контейнерах в хорошо структурированных Web-страницах возникает нечасто. Однако если она все же возникнет, без встроенных контейнеров не обойтись.

Представление для нашего Web-сайта, часть 1

Изучив гору теории, приступим к практике. Начнем создавать представление для нашего первого Web-сайта.

Все стили, которые мы применим к Web-страницам, поместим во внешнюю таблицу стилей main.css. Создадим ее и поместим в корневой папке Web-сайта. После чего привяжем ее ко всем Web-страницам, входящим в него. Как мы помним из *главы 7*, это выполняется с помощью тега `<LINK>`.

Для главной Web-страницы index.htm этот тег будет выглядеть так:

```
<LINK REL="stylesheet" HREF="main.css" TYPE="text/css">
```

Для всех Web-страниц, хранящихся в папке tags, он будет таким:

```
<LINK REL="stylesheet" HREF="../main.css" TYPE="text/css">
```

Внесем в HTML-код всех Web-страниц этот тег и сохраним их.

В тегах `<LINK>`, привязывающих таблицу стилей к Web-страницам, мы указали относительные интернет-адреса. Это позволит нам просматривать Web-страницы, открывая их в Web-обозревателе без использования Web-сервера. Так проще.

Теперь наполним нашу первую таблицу стилей нужными стилями. Но сначала определимся, что мы хотим создать. Опишем параметры шрифта и фона наших Web-страниц обычным, "человеческим" языком.

Концепция Web 2.0, о которой говорилось в *главе 1*, определяет специфические требования к шрифту и фону Web-страниц. Это тонкие и достаточно крупные шрифты без засечек, приглушенные цвета, обычный (неграфический) или однотонный графический фон. Остальные характеристики, как говорится, по вкусу.

Исходя из этого, для Web-страницы мы зададим такие параметры.

- Шрифт абзацев — Verdana. Если таковой отсутствует на клиентском компьютере, применим шрифт Arial.
- Шрифт заголовков — Arial. Это позволит нам дополнительно выделить заголовки, сделать их отличными от обычных абзацев.
- Размер шрифта абзацев — 12 пунктов.
- Размер шрифта заголовков первого уровня — 20 пунктов.
- Размер шрифта заголовков второго уровня — 18 пунктов.
- Размер шрифта заголовков шестого уровня — 12 пунктов.
- Шрифт заголовков всех уровней — обычной насыщенности.
- Размер шрифта в таблицах — 10 пунктов. Пусть таблицы также будут отличаться от обычного текста.
- Размер шрифта больших цитат — 10 пунктов. Пусть и цитаты выглядят по-другому.
- Шрифт больших цитат — курсивный.

- Размер шрифта тега адреса — 10 пунктов. Сведения об авторских правах тоже должны отличаться. К тому же их традиционно пишут мелким шрифтом.
- Шрифт тега адреса — курсивный.
- Цвет текста — `#3B4043` (очень-очень темный, почти черный).
- Цвет фона — `#F8F8F8` (очень-очень светлый, почти белый).

> ***НА ЗАМЕТКУ***
>
> Для подбора цветов можно порекомендовать замечательные библиотеки цветовых тем Web-сайтов, доступных по интернет-адресам **http://www.tarusa.ru/~golovan** и **http://avy.ru**. Так, автор выбрал тему "Капли дождя" из рубрики "Сине-голубые"; цвета из этой темы он и будет применять в дальнейшем.

Осталось написать CSS-код в соответствии с изложенным (листинг 8.2).

Листинг 8.2

```
BODY          { color: #3B4043;
                background-color: #F8F8F8;
                font-family: Verdana, Arial, sans-serif }
P             { font-size: 12pt }
H1, H2, H6    { font-weight: normal;
                font-family: Arial, sans-serif }
H1            { font-size: 20pt }
H2            { font-size: 18pt }
H6            { font-size: 12pt }
TABLE         { font-size: 10pt }
BLOCKQUOTE P,
  ADDRESS     { font-size: 10pt;
                font-style: italic }
```

Таблица стилей готова. Рассмотрим все созданные в ней стили один за другим.

Первым идет стиль переопределения тега `<BODY>`. Он задает параметры, общие для всей Web-страницы: шрифт для обычного текста (абзацев, цитат и содержимого таблиц), цвет текста и цвет фона. Все элементы Web-страницы будут использовать данные параметры, если, конечно, мы не переопределим их далее, в более конкретных тегах:

```
BODY          { color: #3B4043;
                background-color: #F8F8F8;
                font-family: Verdana, Arial, sans-serif }
```

Следующим идет стиль переопределения тега `<P>`. Он задает размер шрифта для текста абзацев. Фактически он дополняет стиль переопределения тега `<BODY>`, созданный ранее:

```
P             { font-size: 12pt }
```

А теперь — внимание! Мы создали три одинаковых стиля, переопределяющих теги заголовков `<H1>`, `<H2>` и `<H6>`:

```
H1, H2, H6    { font-weight: normal;
                font-family: Arial, sans-serif }
```

Они задают параметры, общие для всех заголовков: шрифт и его "жирность" (точнее, отсутствие "жирности"). Поскольку эти стили более конкретные, чем созданный ранее стиль переопределения тега <BODY>, заданные в них параметры будут иметь больший приоритет. Следовательно, заголовки будут набраны шрифтом, который мы указали в этих стилях, а не тем, что упомянут в стиле переопределения тега <BODY>.

Далее мы создали три стиля переопределения тегов <H1>, <H2> и <H6>, задающие разные размеры шрифта для заголовков разного уровня. Эти стили дополнят те, что мы создали чуть раньше. В результате заголовки разного уровня будут набраны шрифтом разного размера:

```
H1            { font-size: 20pt }
H2            { font-size: 18pt }
H6            { font-size: 12pt }
```

Следующим идет стиль переопределения тега <TABLE>, задающий размер шрифта и дополняющий созданный ранее стиль переопределения тега <BODY>. Шрифтом данного размера будет набран текст, присутствующий во всех элементах таблицы (обычных ячейках, ячейках заголовка и заголовке таблицы):

```
TABLE         { font-size: 10pt }
```

Последними мы определили два одинаковых стиля: комбинированный стиль BLOCKQUOTE P и стиль переопределения тега <ADDRESS>:

```
BLOCKQUOTE P,
  ADDRESS     { font-size: 10pt;
                font-style: italic }
```

Они задают одинаковые параметры для шрифта большой цитаты и тега адреса. Поскольку для создания большой цитаты мы использовали тег <P>, вложенный в тег <BLOCKQUOTE>, то параметры текста цитаты мы определили посредством комбинированного стиля BLOCKQUOTE P. Оба этих стиля дополняют созданный в самом начале стиль переопределения тега <BODY>.

Как видим, все довольно просто и наглядно. Стили объединяются друг с другом, переопределяя заданные в них параметры, согласно приоритету. А приоритет зависит от конкретности данного стиля, от "близости" его к тегу.

Сохраним таблицу стилей и откроем в Web-обозревателе Web-страницу index.htm. Совсем другой вид! Вот что можно сделать с Web-страницей с помощью стилей CSS! И ведь нам совсем не пришлось править ее HTML-код (если не считать внесение тега <LINK>, выполняющего привязку таблицы стилей). Разделение содержимого и представления, предписываемое концепцией Web 2.0, налицо!

Что бы нам еще такое сделать?.. Давайте немного разредим текст заголовков, чтобы сделать их более заметными. Для этого достаточно добавить к изначальному пространству между его символами дополнительное, равное 1 мм.

Но куда поместить соответствующий атрибут стиля? В CSS-код, создающий три одинаковых стиля переопределения тегов <H1>, <H2> и <H6>. Вот он:

```
H1, H2, H6    { font-weight: normal;
                font-family: Arial, sans-serif }
```

А так он будет выглядеть после соответствующей правки:

```
H1, H2, H6    { font-weight: normal;
                font-family: Arial, sans-serif;
                letter-spacing: 1mm }
```

Вот и все, что нам понадобилось сделать! Сохраним таблицу стилей, выбрав кодировку UTF-8 (см. *главу 1*), и обновим Web-страницу index.htm, открытую в Web-обозревателе, нажав клавишу <F5>. А что, получилось стильно!

А чтобы совсем уж ошарашить будущих посетителей, давайте задействуем возможности CSS 3 и создадим для текста заголовков тень. Добавим соответствующий атрибут стиля, опять же, в CSS-код, создающий три одинаковых стиля переопределения тегов <H1>, <H2> и <H6> (листинг 8.3).

Листинг 8.3

```
H1, H2, H6    { font-weight: normal;
                font-family: Arial, sans-serif;
                letter-spacing: 1mm;
                text-shadow: #CDD9DB 1px 1px }
```

Для тени мы задали цвет #CDD9DB (светло-синий) и совсем небольшие отступы, равные 1 мм. Такая тень будет ненавязчивой, но симпатичной.

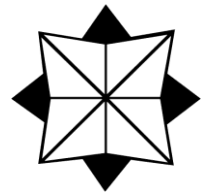
Снова сохраним таблицу стилей и обновим Web-страницу. Посмотрим на результат. Красота...

Мы можем еще поэкспериментировать со стилями — в качестве домашнего задания. Но недолго — нас ждет еще много интересного.

Что дальше?

В этой главе мы научились оформлять текст и фон, используя особые атрибуты стиля. И начали создавать представление для своих Web-страничек.

В следующей главе нас поджидают другие атрибуты стиля — предназначенные для задания параметров абзацев и списков.



ГЛАВА 9

Параметры абзацев, списков и отображения

В предыдущей главе мы рассмотрели атрибуты стиля, предназначенные для задания параметров текста (шрифта, его размера, начертания, тени и пр.) и фона элементов Web-страниц. Изучили новый элемент Web-страницы — встроенный контейнер — и выяснили, зачем он нужен. А также начали создавать представление для Web-страниц нашего первого Web-сайта и неплохо в этом преуспели.

В этой главе мы изучим атрибуты стиля, с помощью которых задают параметры абзацев. Абзацев в широком смысле этого слова — к которым относятся и заголовки, и списки, и теги адреса, и большие цитаты. В общем, блочных элементов, рассмотренных в *главе 2* и предназначенных для структурирования текста.

Далее мы рассмотрим атрибуты стиля, задающие специфические параметры списков и их пунктов (параметры маркеров и нумерации).

Напоследок мы познакомимся с двумя очень специфичными атрибутами стиля, которые позволят нам задать отображение элемента Web-страницы (т. е. будет он блочным или встроенным) и сделать элемент невидимым. Эти атрибуты стиля применяются нечасто и только в совокупности с определенным поведением (см. *часть III*).

Параметры вывода текста

Начнем мы с атрибутов стиля, управляющих выводом текста в структурирующих текст блочных элементах. Их совсем мало. И все они применимы только к блочным элементам.

Атрибут стиля `text-align` задает горизонтальное выравнивание текста:

```
text-align: left|right|center|justify|inherit
```

Здесь доступны значения `left` (выравнивание по левому краю; обычное поведение Web-обозревателя), `right` (по правому краю), `center` (по центру) и `justify` (полное выравнивание).

Примеры:

```
P { text-align: justify }
H1 { text-align: center }
```

Атрибут стиля `text-indent` задает отступ для "красной строки":

```
text-indent: <отступ "красной строки">
```

Здесь допускаются абсолютные и относительные (относительно ширины абзаца) величины отступа. По умолчанию отступ "красной строки" равен нулю. Отметим, что атрибут стиля `text-indent` не поддерживает значение `inherit`.

Пример:

```
P { text-indent: 5mm }
```

Вот теперь абзацы будут иметь "красную строку".

Параметры списков

Списки среди блочных элементов стоят особняком. В основном, из-за того, что, во-первых, содержат в себе другие блочные элементы (отдельные пункты), а во-вторых, включают маркеры и нумерацию, которые расставляет сам Web-обозреватель. Вот о маркерах и нумерации, а точнее, об атрибутах стиля, предназначенных для задания их параметров, мы сейчас и поговорим.

Атрибут стиля `list-style-type` задает вид маркеров или нумерации у пунктов списка:

```
list-style-type: disc|circle|square|decimal|decimal-leading-zero|
lower-roman|upper-roman|lower-greek|lower-alpha|lower-latin|
upper-alpha|upper-latin|armenian|georgian|none|inherit
```

Как видим, доступных значений у этого атрибута стиля очень много. Они обозначают как различные виды маркеров, так и разные способы нумерации.

- `disc` — маркер в виде черного кружка (обычное поведение для маркированных списков).
- `circle` — маркер в виде светлого кружка.
- `square` — маркер в виде квадратика. Он может быть светлым или темным, в зависимости от Web-обозревателя.
- `decimal` — нумерация арабскими цифрами (обычное поведение для нумерованных списков).
- `decimal-leading-zero` — нумерация арабскими цифрами от 01 до 99 с начальным нулем.
- `lower-roman` — нумерация маленькими римскими цифрами.
- `upper-roman` — нумерация большими римскими цифрами.
- `lower-greek` — нумерация маленькими греческими буквами.
- `lower-alpha` и `lower-latin` — нумерация маленькими латинскими буквами.
- `upper-alpha` и `upper-latin` — нумерация большими латинскими буквами.
- `armenian` — нумерация традиционными армянскими цифрами.

- `georgian` — нумерация традиционными грузинскими цифрами.
- `none` — маркер и нумерация отсутствуют (обычное поведение для не-списков).

> ***НА ЗАМЕТКУ***
>
> Мы рассмотрели только значения атрибута стиля `list-style-type`, предлагаемые стандартом CSS 3. Однако стандарт CSS 2 предусматривал еще несколько значений, которые до сих пор поддерживаются Web-обозревателями. Вы можете найти их на Web-странице **https://developer.mozilla.org/en/CSS/list-style-type**.

Атрибут стиля `list-style-type` можно задавать как для самих списков, так и для отдельных пунктов списков. В последнем случае создается список, в котором пункты имеют разные маркеры или нумерацию. Иногда это может пригодиться.

Вот определение маркированного списка с маркером в виде квадратика:

```
UL { list-style-type: square }
```

А в листинге 9.1 мы задали такой же маркер для одного из пунктов маркированного списка.

Листинг 9.1

```
.squared { list-style-type: square }
. . .
<UL>
  <LI>Первый пункт</LI>
  <LI CLASS="squared">Второй пункт (с другим маркером)</LI>
  <LI>Третий пункт</LI>
</UL>
```

Атрибут стиля `list-style-image` позволяет задать в качестве маркера пунктов списка какое-либо графическое изображение (создать *графический маркер*). В этом случае значение атрибута стиля `list-style-type`, если таковой задан, игнорируется:

```
list-style-image: none|<интернет-адрес файла изображения>|inherit
```

Интернет-адрес файла с графическим маркером задается в таком же формате, что и интернет-адрес файла фонового изображения (см. *главу 8*):

```
UL { list-style-image: url(/markers/dot.gif) }
```

Значение `none` убирает графический маркер и возвращает простой, неграфический. Это обычное поведение.

Атрибут стиля `list-style-image` также можно задавать как для самих списков, так и для отдельных их пунктов.

Атрибут стиля `list-style-position` позволяет указать местоположение маркера или нумерации в пункте списка:

```
list-style-position: inside|outside|inherit
```

Доступных значений здесь два:

- `inside` — маркер или нумерация находятся как бы внутри пункта списка, являются его частью;
- `outside` — маркер или нумерация вынесены за пределы пункта списка (это обычное поведение).

Непонятно, зачем нужен данный атрибут стиля, ведь списки, в которых маркер или нумерация вынесены за пределы пунктов, лучше читаются. Ну, раз он присутствует в стандарте CSS, значит, так тому и быть...

Пример:

```
OL { list-style-position: inside }
```

Параметры отображения

Еще одна группа атрибутов стиля управляет тем, как элемент будет отображаться на Web-странице, т. е. будет он блочным или встроенным, и будет ли он отображаться вообще. Эти атрибуты стиля применимы к любым элементам Web-страниц.

Атрибут стиля `visibility` позволяет указать, будет ли элемент отображаться на Web-странице:

```
visibility: visible|hidden|collapse|inherit
```

Он может принимать три значения:

- `visible` — элемент отображается на Web-странице (это обычное поведение);
- `hidden` — элемент не отображается на Web-странице, однако под него все еще выделено на ней место. Другими словами, вместо элемента на Web-странице видна пустая "прореха";
- `collapse` — применим только к строкам, секциям, столбцам и группам столбцов таблицы (о столбцах и группах столбцов пойдет речь в *главе 13*). Элемент не отображается на Web-странице, и под него не выделяется место на ней (т. е. никаких "прорех"). Однако Web-обозреватель считает, что данный элемент Web-страницы все еще на ней присутствует. Данное значение поддерживают не все Web-обозреватели.

Атрибут стиля `visibility` применяется довольно редко и только для создания специальных эффектов, наподобие исчезающего или появляющегося элемента Web-страницы. Используется он всегда совместно с соответствующим поведением (о поведении Web-страниц пойдет речь в *части III*).

Атрибут стиля `display` весьма примечателен. Он позволяет задать вид элемента Web-страницы: будет он блочным, встроенным или вообще пунктом списка.

Пример:

```
display: none|inline|block|inline-block|list-item|run-in|table|
inline-table|table-caption|table-column|table-column-group|
```

```
table-header-group|table-row-group|table-footer-group|table-row|
table-cell|inherit
```

Доступных значений у этого атрибута стиля очень много.

- `none` — элемент вообще не будет отображаться на Web-странице, словно он и не задан в ее HTML-коде.
- `inline` — встроенный элемент.
- `block` — блочный элемент.
- `inline-block` — блочный элемент, который будет "обтекаться" содержимым соседних блочных элементов.
- `list-item` — пункт списка.
- `run-in` — *встраивающийся* элемент. Если за таким элементом следует блочный элемент, он становится частью данного блочного элемента (фактически — встроенным в него элементом), в противном случае он сам становится блочным элементом.
- `table` — таблица.
- `inline-table` — таблица, отформатированная как встроенный элемент. (Оказывается, мы все-таки можем поместить таблицу в абзац! Только кому это нужно...)
- `table-caption` — заголовок таблицы.
- `table-column` — столбец таблицы (подробнее о столбцах таблицы и формирующих их тегах мы поговорим в *главе 13*).
- `table-column-group` — группа столбцов таблицы (подробнее о группах столбцов и формирующих их тегах мы поговорим в *главе 13*).
- `table-header-group` — секция заголовка таблицы.
- `table-row-group` — секция тела таблицы.
- `table-footer-group` — секция завершения таблицы.
- `table-row` — строка таблицы.
- `table-cell` — ячейка таблицы.

> ***Внимание!***
>
> Некоторые значения атрибута стиля `display` определенные Web-обозреватели могут не поддерживать.

Значение по умолчанию атрибута стиля `display` зависит от конкретного элемента Web-страницы. Так, для абзаца значением по умолчанию будет `block`, а для графического изображения (которое, как мы знаем из *главы 4*, является встроенным элементом) — `inline`.

Вот пример комбинированного стиля, позволяющего определенным графическим изображениям отображаться как блочные элементы:

```
IMG.block { display: block }
```

А вот стиль, после применения которого пункты списков станут встроенными элементами и будут выводиться в одну строку:

```
LI { display: inline }
```

Еще один пример:

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

Применение к элементу Web-страницы данного стиля делает элемент невидимым. Более того, на самой Web-странице даже не останется никакого признака того, что данный элемент на ней присутствовал.

В большинстве практических случаев достаточно значений `none`, `block` и `inline` атрибута стиля `display`. Остальные значения слишком специфичны, чтобы часто их применять.

Представление для нашего Web-сайта, часть 2

Что ж, продолжим заниматься представлением для Web-страниц нашего Web-сайта. Зададим для них параметры абзацев и списков.

Снова сформулируем список параметров, которые мы применим к Web-страницам.

- Выравнивание текста в абзацах — полное.
- Выравнивание текста в заголовках — по левому краю. Выровненные по левому краю заголовки читаются лучше выровненных по центру.
- Выравнивание текста в больших цитатах — по левому краю. Так мы дополнительно выделим цитаты.
- Выравнивание текста в теге адреса — по правому краю. Сведения об авторских правах зачастую выравнивают именно так.
- Маркеры у пунктов списков — белый кружок. Придадим стильности нашим спискам.

Исходя из этого, внесем в соответствующие стили, определенные в таблице стилей main.css, следующие изменения:

```
P             { font-size: 12pt;
                text-align: justify }
```

Фактически мы изменили только стиль переопределения тега `<P>`, добавив в него атрибут стиля, задающий полное выравнивание. Остальные стили останутся без изменений, и в соответствующих им элементах Web-страниц (в том числе в заголовках и больших цитатах) выравнивание будет обычным — по левому краю.

После этого добавим в конец таблицы стилей следующий CSS-код:

```
ADDRESS       { text-align: right }
UL            { list-style-type: circle }
```

Здесь мы создали стили переопределения тегов `<ADDRESS>` и `<UL>`. Первый объединится с аналогичным стилем, созданным нами в *главе 8*, и дополнит его параметра-

ми выравнивания (по правому краю). Второй стиль задаст вид маркера для пунктов маркированного списка — белый кружок.

Сохраним таблицу стилей и откроем Web-страницу index.htm в Web-обозревателе. Не бог весть какие изменения, но Web-странице они явно пошли на пользу.

Мы можем еще немного поэкспериментировать с изученными в этой главе атрибутами стилей. Правда, эксперименты долго не продлятся — слишком мало мы изучили атрибутов. Ну ничего, в следующей главе будет где разгуляться!..

Создание полосы навигации

Напоследок создадим для наших Web-страниц нормальную полосу навигации. Сейчас она у нас слишком уж простенькая.

Еще в *главе 6* мы узнали, что полоса навигации может быть горизонтальной или вертикальной, может формироваться в одном абзаце, с помощью набора абзацев, списка или таблицы. Для набора абзацев каждая гиперссылка полосы навигации представляет собой один абзац, для списка — отдельный его пункт, а для таблицы — отдельную ее ячейку. В данный момент наша полоса навигации представляет собой один абзац.

Давайте выберем для формирования полосы навигации список — так сейчас поступают очень часто. И соответственно переделаем HTML-код, формирующий полосу навигации.

В *главе 6*, создавая первую полосу навигации, мы забыли добавить в нее гиперссылку, указывающую на Web-страницу index.htm, которая содержит собственно справочник по HTML. Но правила хорошего тона Web-дизайна требуют, чтобы полоса навигации содержала гиперссылки на все Web-страницы Web-сайта или, по крайней мере, на важнейшие. Поэтому добавим соответствующую гиперссылку в полосу навигации; сделаем это самостоятельно.

Листинг 9.2 содержит фрагмент HTML-кода Web-страницы index.htm, формирующий новую полосу навигации.

Листинг 9.2

```
<UL>
  <LI><A HREF="index.htm">HTML</A></LI>
  <LI><A HREF="css_index.htm">CSS</A></LI>
  <LI><A HREF="samples_index.htm">Примеры</A></LI>
  <LI><A HREF="about.htm">О разработчиках</A></LI>
</UL>
```

Одно из важнейших правил Web-дизайна — полоса навигации должна отличаться от обычного текста. А полоса навигации, сформированная на основе списка, должна отличаться от обычного списка. Как это сделать? С помощью соответствующих стилей.

Сформулируем список параметров для нашей новой полосы навигации.

- Шрифт — Arial. Пусть полоса навигации будет отлична от обычного текста.
- Размер шрифта — 14 пунктов. Полоса навигации должна быть заметна.
- Символы шрифта — только прописные. Так полоса навигации станет еще заметнее.
- Маркер — отсутствует. В полосе навигации маркеры не нужны.

Поскольку полоса навигации будет присутствовать на каждой Web-странице в единственном экземпляре, для ее оформления мы применим именованный стиль (см. *главу 7*). Дадим ему имя `navbar`. CSS-код приведен в листинге 9.3.

Листинг 9.3

```
#navbar       { font-family: Arial, sans-serif;
                font-size: 14pt;
                text-transform: uppercase;
                list-style-type: none }
```

Поместим этот код в самом конце таблицы стилей main.css.

Чтобы привязать именованный стиль к тегу, следует указать его имя в качестве значения атрибута тега `ID` — это мы знаем из *главы 7*. Так и сделаем:

```
<UL ID="navbar">
. . .
```

Осталось сохранить Web-страницу и таблицу стилей и проверить получившийся результат в Web-обозревателе. Что ж, новая полоса навигации заметно лучше старой.

Параметры курсора

CSS предоставляет нам одну очень интересную возможность — указание вида курсора мыши, который он примет при наведении на данный элемент Web-страницы. Это может быть полезно при создании специальных эффектов.

Атрибут стиля `cursor` устанавливает форму курсора мыши при наведении его на данный элемент Web-страницы. Данный атрибут можно применить к любому элементу Web-страницы, как блочному, так и встроенному:

```
cursor: auto|default|none|context-menu|help|pointer|progress|wait|cell|
crosshair|text|vertical-text|alias|copy|move|no-drop|not-allowed|
e-resize|n-resize|ne-resize|nw-resize|s-resize|se-resize|sw-resize|
w-resize|ew-resize|ns-resize|nesw-resize|nwse-resize|col-resize|
row-resize|all-scroll|inherit
```

Как видим, возможных значений у атрибута `cursor` очень много, к тому же многие из них на практике применяются крайне редко. Поэтому мы рассмотрим только самые необходимые.

- `auto` — Web-обозреватель сам управляет формой курсора мыши. Это обычное поведение.
- `default` — курсор по умолчанию, обычно стрелка.
- `none` — курсор мыши вообще не отображается.
- `help` — стрелка с вопросительным знаком.
- `pointer` — "указующий перст". Обычное поведение при наведении курсора мыши на гиперссылку.
- `progress` — стрелка с небольшими песочными часами. Обозначает, что в данный момент работает какой-то фоновый процесс.
- `wait` — песочные часы. Обозначает, что в данный момент Web-обозреватель занят.
- `text` — текстовый курсор. Обычное поведение при наведении курсора мыши на фрагмент текста.

Полный список значений атрибута стиля `cursor` и описание соответствующей им формы курсора мыши вы можете найти на Web-странице **https://developer.mozilla.org/en/CSS/cursor**. Там все просто и наглядно, так что не ошибетесь.

Вот пример задания курсора мыши в виде "указующего перста" для пунктов списка, формирующего только что созданную полосу навигации:

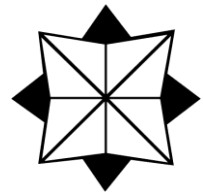
```
#navbar LI { cursor: pointer }
```

Что дальше?

В настоящей главе мы разобрались с атрибутами стиля, задающими параметры абзацев, списков и отображения элементов Web-страницы. Их немного, но при должном применении они могут заметно улучшить наши Web-творения.

В следующей главе мы познакомимся с блочными контейнерами — "коллегами" знакомых нам по *главе 8* встроенных контейнеров. Мы выясним, как можно расположить их на Web-странице в нужном нам порядке и какие атрибуты стиля CSS для этого следует применять. Блочные контейнеры — чрезвычайно мощный инструмент, и мы уделим им много времени.

А еще в следующей главе мы всерьез займемся Web-дизайном.



ГЛАВА 10

Контейнерный Web-дизайн

В предыдущей главе мы рассматривали атрибуты стиля CSS, задающие параметры абзацев и отображения. Их было совсем немного, и глава получилась небольшой.

Теперь мы займемся Web-дизайном. Согласитесь — наши Web-странички, несмотря на созданное нами шикарное представление, выглядят все еще крайне непрезентабельно. Настала пора оформить их как положено.

А для этого нам придется изучить новые элементы Web-страниц — блочные контейнеры. Правда, они не такие уж новые — с их встроенными "коллегами" мы познакомились еще в *главе 8*. Но именно блочные контейнеры дадут нам в руки инструмент современного Web-дизайна.

Разумеется, не обойдется и без дополнительных атрибутов стилей, с помощью которых мы сможем задать необходимые параметры блочных контейнеров (и других блочных элементов). Эти атрибуты стиля позволяют задать размеры и местоположение блочных контейнеров на Web-странице.

А напоследок мы получим от CSS очередной сюрприз — средства управления переполнением. Они пригодятся нам в случаях, если содержимое контейнера не помещается в нем, и не только...

Блочные контейнеры

Как ясно из названия, *блочный* контейнер может хранить только блочные элементы: абзацы, заголовки, большие цитаты, теги адреса, текст фиксированного форматирования, таблицы, аудио- и видеоролики. Блочный контейнер может содержать как один блочный элемент, так и несколько.

Блочный контейнер формируется с помощью парного тега `<DIV>`, внутри которого помещают HTML-код, формирующий содержимое контейнера (листинги 10.1—10.3).

В листинге 10.1 мы поместили в блочный контейнер заголовок и два абзаца.

Листинг 10.1

```
<DIV>
  <H3>Это заголовок</H3>
  <P>Это первый абзац.<P>
  <P>Это второй абзац.<P>
</DIV>
```

Листинг 10.2 иллюстрирует блочный контейнер, содержащий таблицу.

Листинг 10.2

```
<DIV>
  <TABLE>
    <CAPTION>Это таблица</CAPTION>
    <TR>
      <TH>Это первый столбец</TH>
      <TH>Это второй столбец</TH>
    </TR>
    . . .
  </TABLE>
</DIV>
```

А блочный контейнер, приведенный в листинге 10.3, может похвастаться самым "богатым" содержимым.

Листинг 10.3

```
<DIV STYLE="text-align: center">
  <VIDEO SRC="film.ogg" CONTROLS>
  </VIDEO>
  <P>Щелкните кнопку воспроизведения, чтобы просмотреть фильм.</P>
</DIV>
```

Во-первых, мы поместили в него видеоролик и абзац с текстом, предлагающим начать просмотр этого видеоролика. Во-вторых, мы привязали к блочному контейнеру встроенный стиль, задающий выравнивание текста по центру. Отметим при этом, что по центру также будет выровнен и сам видеоролик.

Блочные контейнеры применяют значительно чаще, чем встроенные. Именно на блочных контейнерах основан контейнерный Web-дизайн, о котором сейчас пойдет разговор.

Основы контейнерного Web-дизайна

Контейнерный Web-дизайн известен уже довольно давно. В настоящее время он постепенно вытесняет более старые разновидности Web-дизайна. И тому есть много причин.

Старые разновидности Web-дизайна и их критика

Раньше в Интернете господствовали три разновидности Web-дизайна: текстовый, фреймовый и табличный. Каждый способ имел свои достоинства и недостатки. Но все в той или иной мере проигрывают четвертой разновидности Web-дизайна — контейнерной.

Первым появился, пожалуй, *текстовый* Web-дизайн. Выполненные таким образом Web-страницы представляли собой обычные текстовые документы: набор абзацев, заголовков, больших цитат, текста фиксированного форматирования и таблиц, следующих друг за другом. Классический пример подобного Web-дизайна — созданные нами к данному моменту Web-страницы. Откройте в Web-обозревателе, скажем, Web-страницу index.htm — и вы увидите текстовый Web-дизайн во всей своей, скажем так, красе.

Достоинство текстового Web-дизайна всего одно — исключительная простота HTML-кода. В самом деле, код таких Web-страниц содержит один только текст и, возможно, изображения и таблицы. Никаких специфических элементов, формирующих Web-дизайн как таковой, там нет.

Недостатков же у текстового Web-дизайна много. Во-первых, созданные на его основе Web-страницы выглядят слишком непритязательно. Во-вторых, практически отсутствуют средства произвольного размещения элементов на Web-странице — они могут только следовать друг за другом сверху вниз. В-третьих... а вот об этом следует поговорить подробнее.

Информацию, представленную на Web-странице, можно грубо разделить на четыре фрагмента: заголовок Web-сайта, полосу навигации, *основное содержимое* (информация, уникальная для конкретной Web-страницы, та, ради которой именно эта Web-страница и создавалась) и сведения об авторских правах. Как правило, эти фрагменты визуально отделены друг от друга, так что найти их не составляет труда.

На всех Web-страницах, составляющих Web-сайт, заголовок Web-сайта, полоса навигации и сведения об авторских правах одинаковы. И только основное содержимое у каждой Web-страницы уникально (а оно и должно быть уникально по определению).

Но ведь и заголовок Web-сайта, и полоса навигации, и сведения об авторских правах определяются в HTML-коде каждой Web-страницы. И код этот может быть очень объемным. И что выходит? Значительная часть HTML-кода каждой Web-страницы определяет элементы, которые не меняются от одной Web-страницы к другой!

Чем объемнее HTML-код Web-страницы, тем больше файл, в котором она хранится. Чем значительнее размер файла, тем дольше он загружается. Чем дольше файл загружается, тем больше придется посетителю ждать, пока запрошенная Web-страница появится на экране.

Нет ли способа загрузить не всю Web-страницу целиком, а только ее часть — собственно основное содержимое? К сожалению, текстовый Web-дизайн такого способа не предлагает...

Но выход, тем не менее, был найден в виде "нестандартного" расширения HTML — фреймов. *Фрейм* — это особый элемент Web-страницы, который выводит в указанном ее месте содержимое произвольной Web-страницы, интернет-адрес которой задается в его параметрах. Кроме того, были расширены возможности гиперссылок — теперь они могли выводить целевую Web-страницу в указанном фрейме.

Главная Web-страница Web-сайта в таком варианте представляла собой набор фреймов. Отдельные фрагменты ее содержимого выносились в отдельные Web-страницы, интернет-адреса которых указывались в параметрах соответствующих им фреймов. Остальные Web-страницы включали в себя только основное содержимое. А в параметрах гиперссылок полосы навигации указывалось, в каком фрейме должны загружаться целевые Web-страницы.

Подобный Web-дизайн получил название *фреймового*. Он имел неоспоримое достоинство — резкое увеличение скорости загрузки Web-страниц. И поэтому широко применялся много лет, а кое-где остался и до сих пор.

Однако у фреймов есть существенные недостатки. Во-первых, фреймы так и не были стандартизированы комитетом W3C, поэтому каждый Web-обозреватель обрабатывает их по-своему, не в целом, конечно, а в нюансах, которые, тем не менее, могут быть существенными. Во-вторых, фреймы — очень негибкий элемент Web-страницы; их структуру невозможно поменять.

Конкурентом фреймового Web-дизайна стал появившийся несколько позже *табличный*. Для формирования Web-страницы использовалась большая таблица HTML, в разные ячейки которой помещали заголовок Web-сайта, полосу навигации, различные фрагменты основного содержимого и сведения об авторских правах. Постепенно табличный Web-дизайн вытеснил фреймовый; до сих пор это самый популярный способ создания Web-сайтов.

Достоинства табличного Web-дизайна:

- Таблицы — стандартная часть языка HTML, а значит, можно добиться того, чтобы основанные на них Web-страницы отображались одинаково во всех Web-обозревателях.
- Таблицы HTML можно делать сколь угодно сложными, объединяя их ячейки и вкладывая одни таблицы в другие. Это позволяет делать очень сложные Web-страницы, вмещающие разнородные фрагменты содержимого, имеющие несколько колонок текста и больше похожие на газеты.
- Таблицы и их отдельные ячейки можно легко форматировать с помощью стилей CSS, задавая для них рамки, отступы, фон, выравнивание и другие параметры.

Однако табличный Web-дизайн обладает и множеством недостатков:

- Все та же "монолитность" Web-страниц, что и в случае текстового Web-дизайна. Каждая Web-страница Web-сайта содержит и его заголовок, и полосу навигации, и основное содержимое, и сведения об авторских правах, что не лучшим образом сказывается на ее размерах и на скорости ее загрузки.
- Для формирования сложных таблиц применяется чрезвычайно громоздкий и запутанный HTML-код.

- Старые версии Web-обозревателей не очень удачно реализовывали обработку таблиц: они сначала загружали таблицу целиком, а уже потом выводили ее на экран. Учитывая, что таблицы, с помощью которых формировались Web-страницы, очень большие, загрузка таких Web-страниц отнимала много времени.

> ***НА ЗАМЕТКУ***
>
> Современные Web-обозреватели могут выводить таблицу на экран в процессе ее загрузки. Это "умеет" даже Internet Explorer, славящийся своим, мягко говоря, консерватизмом. Так что последняя проблема отпала.

Как видим, все три старых принципа Web-дизайна, наряду с достоинствами, имеют и серьезные недостатки. Поэтому в настоящее время они медленно, но верно отступают под натиском амбициозного новичка, имя которому...

Сущность контейнерного Web-дизайна

Контейнерный Web-дизайн для размещения отдельных фрагментов содержимого Web-страниц использует блочные контейнеры, с которыми мы познакомились в начале этой главы. Отдельные контейнеры создаются для заголовка Web-сайта, полосы навигации, основного содержимого и сведений об авторских правах. Если основное содержимое имеет сложную структуру и состоит из множества независимых частей, для каждой из них создают отдельный контейнер.

Для задания различных параметров блочных контейнеров предусмотрены специальные атрибуты стиля CSS. К таким параметрам относятся размеры (ширина и высота), местоположение контейнеров и их поведение при переполнении. Также мы можем задать для контейнеров цвет фона, создать отступы и рамки, чтобы их выделить (о параметрах отступов и рамок речь пойдет в *главе 11*).

И что, контейнерный Web-дизайн так уж хорош? Давайте рассмотрим недостатки трех старых принципов Web-дизайна и выясним, сможет ли он их решить.

- Непритязательный вид и линейное представление Web-страниц — у текстового Web-дизайна. Мы можем расположить контейнеры на Web-странице практически как угодно и поместить в них произвольное содержимое: текст, таблицы, изображения, аудио- и видеоролики и даже другие контейнеры. А CSS позволит нам задать для них практически любое представление.
- "Монолитность" Web-страниц — у текстового и табличного Web-дизайна. Современные Web-обозреватели позволяют с помощью специально созданного поведения загрузить в контейнер Web-страницу, сохраненную в отдельном файле, т. е. организовать подгружаемое содержимое. Мы займемся этим в *главе 18*.
- "Нестандартность" фреймов — у фреймового Web-дизайна. Контейнеры и соответствующие теги официально стандартизированы комитетом W3C и обрабатываются всеми Web-обозревателями одинаково.
- Громоздкость HTML-кода — у табличного Web-дизайна. HTML-код, формирующий контейнеры, исключительно компактен. Как мы уже знаем, блочный контейнер формируется всего одним парным тегом `<DIV>`.

- Медленная загрузка Web-страниц — у табличного Web-дизайна. Все Web-обозреватели отображают содержимое контейнеров прямо в процессе загрузки, так что Web-страницы визуально загружаются очень быстро.

И что, контейнерный Web-дизайн так хорош? И он совсем не имеет недостатков? Увы, ничего совершенного в мире нет...

Контейнерный Web-дизайн проигрывает табличному в возможности реализации сложного дизайна Web-страниц. Таблица позволяет создать на Web-странице множество колонок разной ширины, содержащих разное содержимое. Чтобы сделать это с помощью контейнеров, скорее всего, придется применять вложенные друг в друга контейнеры, сложные стили и, возможно, поведение, которое уже после окончания загрузки Web-страницы располагает контейнеры в нужных местах. Это, пожалуй, единственный недостаток контейнерного Web-дизайна.

Что ж, с контейнерным Web-дизайном все в основном ясно. Давайте попрактикуемся. Переделаем наши Web-страницы с применением контейнерного Web-дизайна — "легкого", простого, современного.

Представление для нашего Web-сайта, часть 3

Сначала разработаем схему расположения на Web-страницах соответствующих контейнеров. Лучше всего нарисовать ее на бумаге или в программе графического редактора.

Классическая схема Web-дизайна (и не только контейнерного) показана на рис. 10.1. Как видим, в самом верху располагается заголовок Web-сайта, ниже него в одну линию по горизонтали выстроились полоса навигации и основное содержимое, а под ними пристроились сведения об авторских правах. Давайте используем именно эту схему.

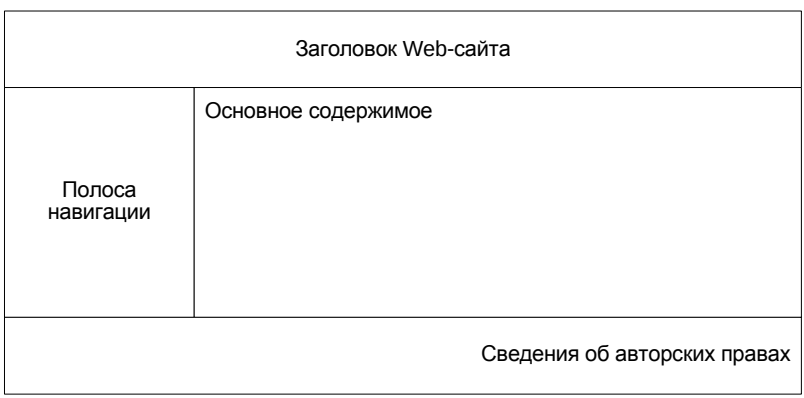


Рис. 10.1. Классическая схема Web-дизайна

Откроем в Блокноте Web-страницу index.htm. Найдем в ее HTML-коде четыре важных фрагмента любой Web-страницы: заголовок Web-сайта, полосу навигации, ос-

новное содержимое и сведения об авторских правах. Поместим их в блочные контейнеры.

На рис. 10.1 показано, что заголовок Web-сайта предшествует полосе навигации, а не наоборот. Поэтому поменяем местами фрагменты HTML-кода, создающие эти контейнеры и их содержимое.

Впоследствии мы привяжем к созданным контейнерам стили, задающие их размеры и местоположение на Web-странице. Поскольку каждый из этих контейнеров присутствует на каждой Web-странице в единственном экземпляре, применим для этого именованные стили. Дадим им такие имена:

- `cheader` — стиль для контейнера с заголовком Web-сайта;
- `cnav` — стиль для контейнера с полосой навигации;
- `cmain` — стиль для контейнера с основным содержимым;
- `ccopyright` — стиль для контейнера со сведениями об авторских правах.

Здесь буква "c" обозначает "container" — контейнер. Так мы сразу поймем, что данные стили применяются именно к контейнерам.

Привязка именованного стиля к тегу выполняется путем указания имени этого стиля в качестве значения атрибута тега `ID`. Сделаем это для всех контейнеров.

В листинге 10.4 приведен фрагмент HTML-кода Web-страницы index.htm со всеми нужными исправлениями.

Листинг 10.4

```
<DIV ID="cheader">
  <H1>Справочник по HTML и CSS</H1>
</DIV>
<DIV ID="cnavbar">
  <UL ID="navbar">
    <LI><A HREF="index.htm">HTML</A></LI>
    <LI><A HREF="css_index.htm">CSS</A></LI>
    <LI><A HREF="samples_index.htm">Примеры</A></LI>
    <LI><A HREF="about.htm">О разработчиках</A></LI>
  </UL>
</DIV>
<DIV ID="cmain">
  <P>Приветствуем на нашем Web-сайте всех, кто занимается Web-дизайном!
  Здесь вы сможете найти информацию обо всех интернет-технологиях,
  применяемых при создании Web-страниц. А именно, о языках
  <DFN>HTML</DFN> и <DFN>CSS</DFN>.</P>
  <HR>
  . . .
  <H2>Теги HTML</H2>
  <P><CODE>!DOCTYPE</CODE>,
  <CODE><A HREF="tags/t_audio.htm">AUDIO</A></CODE>,
```

```
  <CODE>BODY</CODE>, <CODE>EM</CODE>, <CODE>HEAD</CODE>,
  <CODE>HTML</CODE>, <CODE><A HREF="tags/t_img.htm">IMG</A></CODE>,
  <CODE>META</CODE>, <CODE>P</CODE>, <CODE>STRONG</CODE>,
  <CODE><A HREF="tags/t_title.htm">TITLE</A></CODE>,
  <CODE><A HREF="tags/t_video.htm">VIDEO</A></CODE></P>
</DIV>
<DIV ID="ccopyright">
  <ADDRESS>Все права защищены.<BR>&#0169;
  <A HREF="mailto:user@mailserver.ru">читатели</A>, 2010 год.</ADDRESS>
</DIV>
```

Сохраним Web-страницу index.htm и откроем ее в Web-обозревателе. Что-то изменилось по сравнению с тем, что было ранее? Ничего.

Стили, задающие параметры контейнеров

И неудивительно. Вспомним, что мы знаем о контейнерах, и блочных, и встроенных. Они никак не отображаются в Web-обозревателе!

Чтобы ощутить пользу от контейнеров, мы должны применить к ним стили. Именно для этого контейнеры и предназначены. Поэтому сейчас мы займемся атрибутами стиля, задающими параметры контейнеров. Атрибуты стиля и задаваемые ими параметры делятся на две группы.

Параметры размеров

Атрибуты стиля первой группы задают размеры контейнеров. Собственно, их можно применять не только в контейнерах, но и в любых других блочных элементах.

Атрибуты стиля `width` и `height` позволяют задать, соответственно, ширину и высоту элемента Web-страницы:

```
width: auto|<ширина>|inherit
height: auto|<высота>|inherit
```

Мы можем указать абсолютное значение размера, воспользовавшись любой из доступных в CSS единиц измерения. Тогда размер элемента будет неизменным:

```
#cnavbar { width: 100px }
```

Можно задать относительное значение размера в процентах от соответствующего размера родительского элемента. При этом размер элемента будет изменяться при изменении размеров окна Web-обозревателя:

```
#cheader { height: 10% }
```

Значение `auto` отдает управление этим размером на откуп Web-обозревателю (обычное поведение). В последнем случае Web-обозреватель установит такие размеры элемента Web-страницы, чтобы в нем полностью поместилось его содержимое, и не больше.

Если мы назначим для какого-либо элемента Web-страницы относительную ширину или высоту, нам могут пригодиться атрибуты стиля, указывающие минимальные и максимальные возможные размеры для этого элемента. Понятно, что при их задании значение размера не сможет превысить максимальное и стать меньше минимального.

Атрибуты стиля `min-width` и `min-height` позволяют определить минимальную ширину и высоту элемента Web-страницы:

```
min-width: <ширина>
min-height: <высота>
```

Значение ширины или высоты может быть как абсолютным, так и относительным.

Пример:

```
TABLE { min-width: 500px;
        min-height: 300px }
```

Аналогичные атрибуты стиля `max-width` и `max-height` позволяют задать максимальную, соответственно, ширину и высоту элемента Web-страницы:

```
max-width: <ширина>
max-height: <высота>
```

И здесь значение ширины или высоты может быть как абсолютным, так и относительным:

```
TABLE { max-width: 90%;
        max-height: 60% }
```

Параметры размещения. Плавающие контейнеры

Местоположение блочных контейнеров (и любых других блочных элементов) на Web-странице определяют два весьма примечательных атрибута стиля.

Изначально блочные элементы Web-страницы располагаются на ней по вертикали, строго друг за другом, в том порядке, в котором они определены в HTML-коде. Именно так располагаются блочные контейнеры, абзацы и заголовки на всех созданных нами Web-страницах.

Однако мы можем установить для блочного элемента выравнивание по левому или краю родительского элемента (блочного контейнера, в который она вложена, или самой Web-страницы). При этом блочный элемент будет прижиматься к соответствующему краю родителя, а остальное содержимое будет обтекать его с противоположной стороны.

Атрибут стиля `float` как раз и задает, по какому краю родительского элемента будет выравниваться данный элемент Web-страницы:

```
float: left|right|none|inherit
```

Возможны три значения:

- `left` — элемент Web-страницы выравнивается по левому краю родительского элемента, а остальное содержимое обтекает его справа;

- `right` — элемент Web-страницы выравнивается по правому краю родительского элемента, а остальное содержимое обтекает его слева;
- `none` — обычное поведение элемента Web-страницы, когда он следует за своим предшественником и располагается ниже его.

Пример:

```
TABLE.left-aligned { float: left }
```

После применения данного стиля к таблице она будет выровнена по левому краю родительского элемента, а остальное содержимое будет обтекать ее справа.

А что если мы зададим одинаковое значение атрибута стиля `float` для нескольких следующих друг за другом блочных элементов? Они выстроятся по горизонтали друг за другом в том порядке, в котором они определены в HTML-коде, и будут выровнены по указанному краю родительского элемента. Настоящая находка!

Данный атрибут стиля обычно применяют для блочных контейнеров, формирующих дизайн Web-страниц. Такие контейнеры называют *плавающими*.

Возьмем атрибут стиля `float` на заметку. И пойдем дальше.

При создании контейнерного Web-дизайна, основанного на плавающих контейнерах, часто приходится помещать какие-либо контейнеры ниже тех, что были выровнены по левому или правому краю родительского элемента. Если просто убрать у них атрибут стиля `float` или задать для него значение `none`, результат будет непредсказуемым. Поэтому CSS предоставляет возможность однозначно указать, что данный блочный элемент в любом случае должен располагаться ниже плавающих контейнеров, предшествующих ему.

Для этого служит атрибут стиля `clear`:

```
clear: left|right|both|none|inherit
```

Как видим, доступных значений здесь четыре:

- `left` — элемент Web-страницы должен располагаться ниже всех элементов, для которых у атрибута стиля `float` задано значение `left`;
- `right` — элемент Web-страницы должен располагаться ниже всех элементов, для которых у атрибута стиля `float` задано значение `right`;
- `both` — элемент Web-страницы должен располагаться ниже всех элементов, для которых у атрибута стиля `float` задано значение `left` или `right`;
- `none` — обычное поведение. Если контейнеру, для которого указан данный атрибут стиля, предшествуют плавающие контейнеры, задавать это значение не рекомендуется.

Пример:

```
#ccopyright { clear: both }
```

Здесь мы задали для именованного стиля `ccopyright` (он применяется к контейнеру, содержащему сведения об авторских правах) расположение ниже всех контейнеров, выровненных по левому или правому краю родительского элемента.

Представление для нашего Web-сайта, часть 4

Полученных нами знаний уже достаточно для того, чтобы создать контейнерный дизайн для нашего Web-сайта. Давайте займемся этим.

Как обычно, выпишем список параметров для всех созданных ранее контейнеров.

Для контейнера с заголовком Web-сайта (`cheader`):

- ширина — 100% (все окно Web-обозревателя).

Для контейнера с полосой навигации (`cnavbar`):

- ширина — 30% (примерно треть ширины окна Web-обозревателя);
- минимальная ширина — 240 пикселов (это значение получено автором в результате экспериментов; оно примерно равно ширине полосы навигации);
- выравнивание — по левому краю (т. е. это будет плавающий контейнер).

Для контейнера с основным содержимым (`cmain`):

- ширина — 70% (примерно две трети ширины окна Web-обозревателя);
- выравнивание — по левому краю.

Для контейнера со сведениями об авторских правах (`ccopyright`):

- ширина — 100% (все окно Web-обозревателя);
- расположение — ниже всех плавающих контейнеров, выровненных по левому и правому краям.

Как видим, ни у одного контейнера явно не задана высота. Web-обозреватель сам установит ее такой, чтобы контейнер при указанной ширине полностью вместил свое содержимое.

На основе перечисленных требований напишем CSS-код, определяющий нужные стили (листинг 10.5).

Листинг 10.5

```
#cheader      { width: 100% }
#cnavbar      { width: 30%;
                min-width: 240px;
                float: left }
#cmain        { width: 70%;
                float: left }
#ccopyright   { width: 100%;
                clear: both }
```

Поместим этот код в самый конец таблицы стилей main.css, после чего сохраним ее. Откроем Web-страницу index.htm в Web-обозревателе и посмотрим, что получилось (рис. 10.2).

Мы сделали это! Много времени ушло на изучение HTML и CSS, но результат того стоит. Вот она — наша первая Web-страница, выполненная по канонам современного Web-дизайна.

Еще немного полюбуемся на преобразившуюся Web-страницу. И снова за дело.

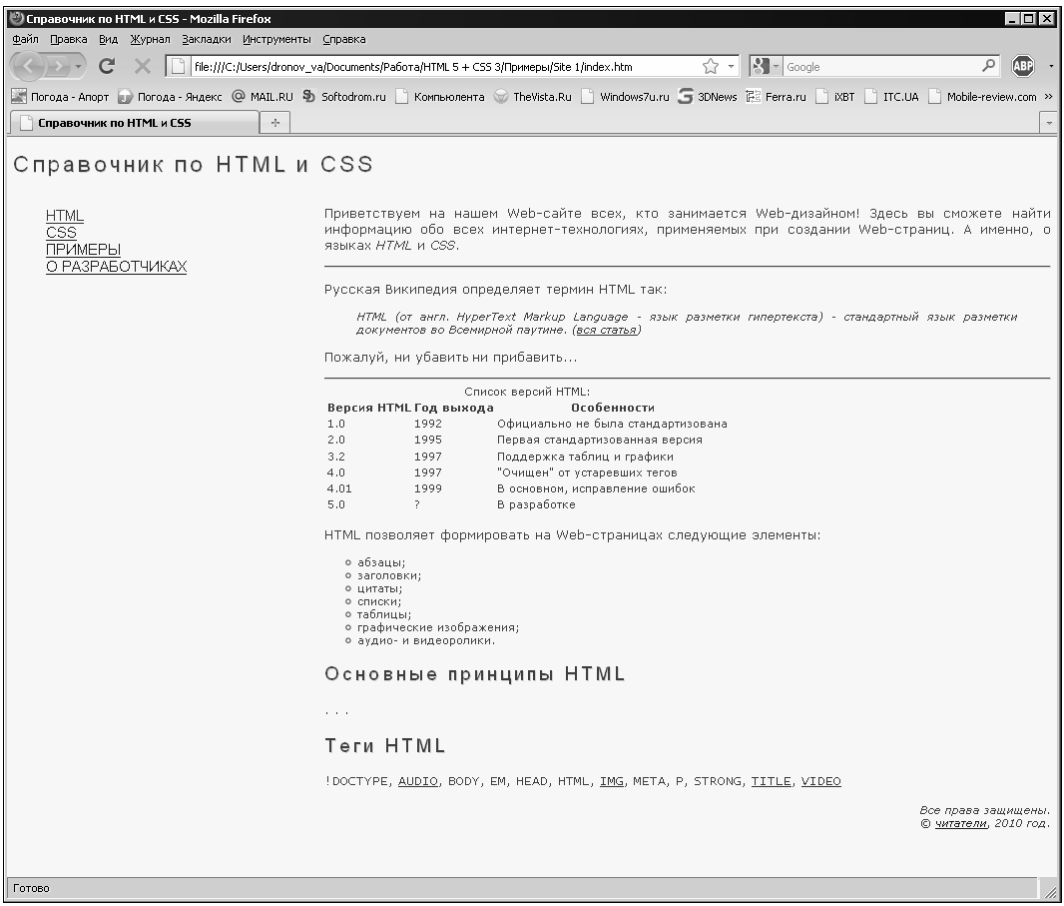


Рис. 10.2. Web-страница index.htm, выполненная на основе контейнерного Web-дизайна, в Web-обозревателе

Параметры переполнения. Контейнеры с прокруткой

Ни у одного из контейнеров, формирующих дизайн нашей Web-страницы, мы не задали явную высоту. Web-обозреватель сам установит для контейнеров такие значения высоты, чтобы они вместили свое содержимое полностью.

Но что случится, если мы зададим для контейнера высоту? Тогда может выйти так, что содержимое контейнера не поместится в нем, и возникнет *переполнение кон-*

тейнера. Поведение контейнера зависит от параметров, которые мы зададим для него.

Атрибут стиля `overflow` как раз и задает поведение контейнера при переполнении:

```
overflow: visible|hidden|scroll|auto|inherit
```

Здесь доступны четыре значения:

- `visible` — высота контейнера увеличится, чтобы полностью вместить все содержимое (обычное поведение);
- `hidden` — не помещающееся в контейнер содержимое будет обрезано. Контейнер сохранит свои размеры;
- `scroll` — в контейнере появятся полосы прокрутки, с помощью которых можно просмотреть не помещающуюся часть содержимого. Эти полосы прокрутки будут присутствовать в контейнере всегда, даже если в них нет нужды;
- `auto` — полосы прокрутки появятся в контейнере, только если в них возникнет необходимость.

Пример:

```
#cmain { overflow: auto }
```

Мы задали для контейнера `cmain` такое поведение, когда при выходе содержимого за границы контейнера в нем появятся полосы прокрутки.

Здесь нужно сказать еще вот что. Атрибут стиля `overflow` имеет смысл только в том случае, если мы зададим для высоты контейнера абсолютное значение. При указании относительного значения высоты контейнера он всегда будет увеличиваться в размерах для того, чтобы вместить все содержимое, как будто для атрибута стиля `overflow` задано значение `visible`:

```
#cmain { height: 500px;
         overflow: auto }
```

Вот теперь контейнер `cmain` при необходимости обзаведется полосами прокрутки.

А в следующем примере атрибут стиля `overflow` можно не указывать — контейнер `cmain` всегда будет вести себя так, будто для упомянутого атрибута стиля задано значение `visible`:

```
#cmain { height: 50%;
         overflow: auto }
```

Атрибуты стиля `overflow-x` и `overflow-y` задают поведение контейнера при выходе содержимого за пределы его границ, соответственно, по горизонтали и вертикали. Доступные значения у них те же, что и у атрибута стиля `overflow`:

```
#cnavbar { width: 270px;
           overflow-x: hidden }
```

Пользуясь только что изученными атрибутами стиля, мы можем создать на Web-странице *контейнеры с прокруткой*! Это обычные контейнеры с большим содер-

жимым, для просмотра которого предусмотрены полосы прокрутки. Их преимущество в том, что посетитель, прокручивая содержимое такого контейнера, не затрагивает все остальные фрагменты Web-страницы (заголовок Web-сайта, полоса навигации и сведения об авторских правах). Весьма удобно.

Контейнеры с прокруткой — высший шик современного Web-дизайна. Нужно будет и нам создать такие.

Представление для нашего Web-сайта, часть 5

Давайте создадим контейнер с прокруткой для вывода основного содержимого Web-страницы. Точнее, переделаем уже созданный контейнер `cmain`.

Что нам для этого потребуется? Во-первых, задать для данного контейнера абсолютное значение высоты, а лучше — и ширины, и высоты. Во-вторых, определить соответствующее поведение при переполнении.

При задании абсолютного значения ширины для контейнера `cmain` нужно установить абсолютное значение ширины и у контейнера `cnavbar`. Смешивать абсолютные и относительные значения ширины у плавающих контейнеров не рекомендуется — результат будет непредсказуемым.

Выпишем список параметров контейнеров `cnavbar` и `cmain`.

Для контейнера `cnavbar`:

- ширина — 240 пикселов;
- высота — 620 пикселов;
- выравнивание — по левому краю.

Для контейнера `cmain`:

- ширина — 780 пикселов;
- высота — 620 пикселов;
- выравнивание — по левому краю;
- поведение при переполнении — появление полос прокрутки.

Размеры контейнеров автор определил в результате экспериментов. Он постарался выбрать такие значения, чтобы пространство Web-страницы было занято максимально полно, а полосы прокрутки у самой Web-страницы отсутствовали. Скорее всего, вам придется подобрать другие значения размеров.

Кроме того, мы задали одинаковую высоту у обоих контейнеров — и `cnavbar`, и `cmain`. Это нужно для того, чтобы исключить некоторые нежелательные эффекты, которые могут возникнуть, если мы создадим у контейнеров рамки или изменим цвет фона (о создании рамок речь пойдет в *главе 11*).

Осталось воплотить наши требования к контейнерам в CSS-код. В листинге 10.6 приведен исправленный фрагмент таблицы стилей main.css, создающий стили, соответствующие контейнерам `cnavbar` и `cmain`.

Листинг 10.6

```
#cnavbar      { width: 240px;
                height: 620px;
                float: left }
#cmain        { width: 780px;
                height: 620px;
                float: left;
                overflow: auto }
```

Внесем эти исправления в таблицу стилей, сохраним ее и проверим, что получилось. А получилось у нас то, что показано на рис. 10.3.

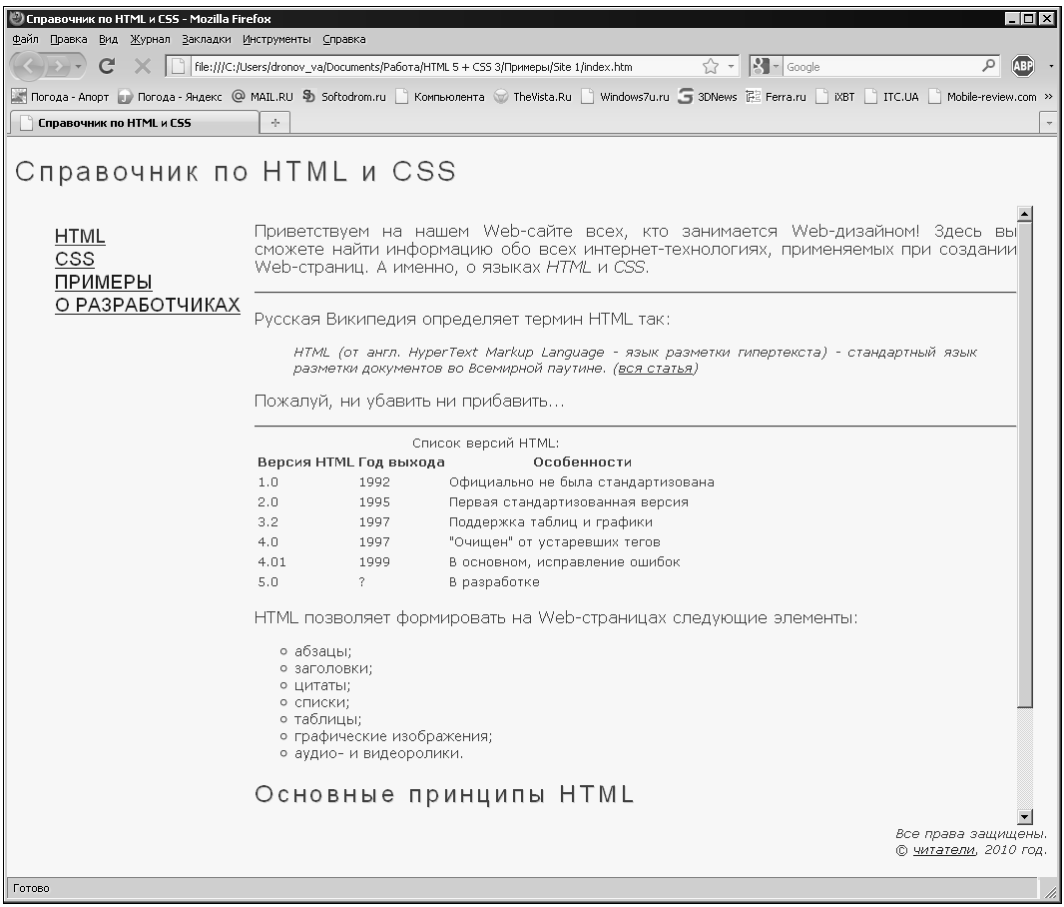


Рис. 10.3. Web-страница index.htm, содержащая контейнер с прокруткой, в Web-обозревателе

У нас получилась на Web-странице этакое "окошко", содержимое которого можно прокручивать независимо от всего остального. Многие ли Web-сайты могут этим похвастаться?..

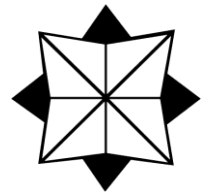
Теперь давайте изменим размеры окна Web-обозревателя. И наш красивый и современный Web-дизайн превратится невесть во что...

В *части III* мы узнаем, как менять размеры контейнеров в ответ на изменение размеров окна Web-обозревателя с помощью специального поведения. А пока что вернем прежние размеры окна. И закончим с контейнерным Web-дизайном.

Что дальше?

В этой главе мы узнали о разных контейнерах: блочных, плавающих и с прокруткой, контейнерном Web-дизайне и атрибутах стиля, задающих различные параметры контейнеров. Контейнеры — вот настоящие "герои" этой главы!

Следующая глава будет посвящена атрибутам стиля, с помощью которых создаются отступы, рамки и выделение элементов Web-страницы. Ну, и контейнерами мы тоже будем заниматься. И сделаем свое Web-творение еще лучше.



ГЛАВА 11

Отступы, рамки и выделение

В предыдущей главе мы занимались контейнерным Web-дизайном: создавали блочные контейнеры, помещали в них содержимое и задавали для них параметры с помощью особых атрибутов стиля CSS. Результат — вполне профессионально созданная Web-страница.

Но чего-то в ней не хватает... Какой-то мелочи недостает, чтобы придать Web-странице лоск. Может быть, отступов и рамок?

Так давайте же разберемся с атрибутами стиля, с помощью которых задают параметры отступов и рамок! И доделаем наконец нашу Web-страницу.

Параметры отступов

Стандарт CSS предлагает средства для создания отступов двух видов.

1. Отступ между воображаемой границей элемента Web-страницы и его содержимым — *внутренний* отступ. Такой отступ принадлежит данному элементу Web-страницы, находится внутри его.
2. Отступ между воображаемой границей данного элемента Web-страницы и воображаемыми границами соседних элементов Web-страницы — *внешний* отступ. Такой отступ не принадлежит данному элементу Web-страницы, находится вне его.

Чтобы лучше понять разницу между внутренним и внешним отступами, давайте рассмотрим ячейку таблицы. Ячейка наполнена содержимым, скажем, текстом, имеет воображаемую границу и окружена другими ячейками.

- Внутренний отступ — это отступ между границей ячейки и содержащимся в ней текстом.
- Внешний отступ — это отступ между границами отдельных ячеек таблицы.

Атрибуты стиля `padding-left`, `padding-top`, `padding-right` и `padding-bottom` позволяют задать величины внутренних отступов, соответственно, слева, сверху, справа и снизу элемента Web-страницы:

```
padding-left|padding-top|padding-right|padding-bottom:
<отступ>|auto|inherit
```

Мы можем указать в качестве величины отступа абсолютное или относительное значение. Значение `auto` задает величину отступа по умолчанию, обычно оно равно нулю.

В листинге 11.1 мы указали внутренний отступ для ячеек таблицы, равный двум пикселам со всех сторон.

Листинг 11.1

```
TD, TH { padding-left: 2px;
         padding-top: 2px;
         padding-right: 2px;
         padding-bottom: 2px }
```

А вот стиль, создающий внутренние отступы, равные двум сантиметрам слева и справа:

```
.indented { padding-left: 2cm;
            padding-right: 2cm }
```

Мы можем привязать такой стиль к абзацу и посмотреть, что получится.

Атрибут стиля `padding` позволяет сразу указать величины внутренних отступов со всех сторон элемента Web-страницы:

```
padding: <отступ 1> [<отступ 2> [<отступ 3> [<отступ 4>]]]
```

- Если указано одно значение, оно задаст величину отступа со всех сторон элемента Web-страницы.
- Если указаны два значения, первое установит величину отступа сверху и снизу, а второе — слева и справа.
- Если указаны три значения, первое определит величину отступа сверху, второе — слева и справа, а третье — снизу.
- Если указаны четыре значения, первое задаст величину отступа сверху, второе — справа, третье — снизу, а четвертое — слева.

Пример:

```
TD, TH { padding: 2px }
.indented { padding: 0cm 2cm 0cm 2cm }
```

Здесь мы просто переписали определения приведенных ранее стилей с использованием атрибута стиля `padding`.

Атрибуты стиля `margin-left`, `margin-top`, `margin-right` и `margin-bottom` позволяют задать величины внешних отступов, соответственно, слева, сверху, справа и снизу:

```
margin-left|margin-top|margin-right|margin-bottom: <отступ>|auto|inherit
```

Здесь также доступны абсолютные и относительные значения. Значение `auto` задает величину отступа по умолчанию, как правило, равное нулю.

Пример:

```
H1 { margin-top: 5mm }
```

Этот стиль создаст у всех заголовков первого уровня отступ сверху 5 мм.

В качестве значений внешних отступов допустимы отрицательные величины:

```
UL { margin-left: -20px }
```

В этом случае Web-обозреватель создаст "отрицательный" отступ. Такой прием позволяет убрать отступы, создаваемые Web-обозревателем по умолчанию, например, отступы слева у больших цитат и списков.

Внешние отступы мы также можем указать с помощью атрибута стиля `margin`. Он задает величины отступа одновременно со всех сторон элемента Web-страницы:

`margin: <отступ 1> [<отступ 2> [<отступ 3> [<отступ 4>]]]`

Этот атрибут стиля ведет себя так же, как его "коллега" `padding`.

Пример:

```
H1 { margin: 5mm 0mm }
```

Однако мы не можем использовать атрибуты стиля `margin-left`, `margin-top`, `margin-right`, `margin-bottom` и `margin` для задания внешних отступов у ячеек таблиц (т. е. расстояния между ячейками) — они просто не будут действовать. Вместо этого следует применить атрибут стиля `border-spacing`:

`border-spacing: <отступ 1> [<отступ 2>]`

Отступы могут быть заданы только в виде абсолютных значений.

- Если указано одно значение, оно задаст величину отступа со всех сторон ячейки таблицы.
- Если указаны два значения, первое задаст величину отступа слева и справа, а второе — сверху и снизу.

Атрибут стиля применяется только к таблицам (тегу `<TABLE>`):

```
TABLE { border-spacing: 1px }
```

Здесь мы задали отступы между ячейками таблицы, равные одному пикселу.

> ***Внимание!***
>
> Задавая отступы, внутренние или внешние, нужно помнить, что они увеличивают размеры элемента Web-страницы. Поэтому, если мы применим отступы к блочным контейнерам, формирующим дизайн Web-страницы, то должны будем соответственно изменить размеры этих контейнеров, иначе они сместятся, и дизайн будет нарушен.
>
> Также нужно знать, что при применении отступов к элементу Web-страницы с размерами, заданными в виде относительных величин, Web-обозреватель сначала вычисляет абсолютный размер элемента, а потом к нему добавляет величины отступов. Так, если мы зададим ширину контейнера в 100%, а потом укажем для него отступы, то Web-обозреватель сначала вычислит его абсолютную ширину, основываясь на размерах окна Web-обозревателя, а потом прибавит к ней величину отступов. В результате ширина контейнера станет больше, чем ширина окна Web-обозревателя, и в окне появятся полосы прокрутки. Весьма неприятный сюрприз...

Параметры рамки

CSS также позволяет нам создать сплошную, пунктирную или точечную рамку по воображаемой границе элемента Web-страницы.

Атрибуты стиля `border-left-width`, `border-top-width`, `border-right-width` и `border-bottom-width` задают толщину, соответственно, левой, верхней, правой и нижней сторон рамки:

```
border-left-width|border-top-width|border-right-width|
border-bottom-width: thin|medium|thick|<толщина>|inherit
```

Мы можем указать либо абсолютное или относительное числовое значение толщины рамки, либо одно из предопределенных значений: `thin` (тонкая), `medium` (средняя) или `thick` (толстая). В последнем случае реальная толщина рамки зависит от Web-обозревателя. Значение толщины по умолчанию также зависит от Web-обозревателя, поэтому ее всегда лучше устанавливать явно.

В листинге 11.2 мы указали толщину рамки у ячеек таблицы, равную одному пикселу.

Листинг 11.2

```
TD, TH { border-left-width: thin;
         border-top-width: thin;
         border-right-width: thin;
         border-bottom-width: thin }
```

А вот стиль, создающий у всех заголовков первого уровня рамку из одной только нижней стороны толщиной 5 пикселов:

```
H1 { border-bottom-width: 5px }
```

Фактически все заголовки первого уровня окажутся подчеркнутыми.

Атрибут стиля `border-width` позволяет указать значения толщины сразу для всех сторон рамки:

```
border-width: <толщина 1> [<толщина 2> [<толщина 3> [<толщина 4>]]]
```

- Если указано одно значение, оно задаст толщину всех сторон рамки.
- Если указаны два значения, первое задаст толщину верхней и нижней, а второе — левой и правой сторон рамки.
- Если указаны три значения, первое задаст толщину верхней, второе — левой и правой, а третье — нижней сторон рамки.
- Если указаны четыре значения, первое задаст толщину верхней, второе — правой, третье — нижней, а четвертое — левой сторон рамки.

Пример:

```
TD, TH { border-width: thin }
```

Атрибуты стиля `border-left-color`, `border-top-color`, `border-right-color` и `border-bottom-color` задают цвет, соответственно, левой, верхней, правой и нижней сторон рамки:

```
border-left-color|border-top-color|border-right-color|
border-bottom-color: transparent|<цвет>|inherit
```

Значение `transparent` задает "прозрачный" цвет, сквозь который будет "просвечивать" фон родительского элемента.

> ***ВНИМАНИЕ!***
> Цвет рамки всегда следует задавать явно — в противном случае она может быть не нарисована.

Пример:

```
H1 { border-bottom-width: 5px
     border-bottom-color: red }
```

Атрибут стиля `border-color` позволяет указать цвет сразу для всех сторон рамки:

```
border-color: <цвет 1> [<цвет 2> [<цвет 3> [<цвет 4>]]]
```

Он ведет себя так же, как и аналогичный атрибут стиля `border-width`:

```
TD, TH { border-width: thin;
         border-color: black }
```

Атрибуты стиля `border-left-style`, `border-top-style`, `border-right-style` и `border-bottom-style` задают стиль линий, которыми будет нарисована, соответственно, левая, верхняя, правая и нижняя сторона рамки:

```
border-left-style|border-top-style|border-right-style|
border-bottom-style: none|hidden|dotted|dashed|solid|double|groove|
ridge|inset|outset|inherit
```

Здесь доступны следующие значения:

- `none` и `hidden` — рамка отсутствует (обычное поведение);
- `dotted` — пунктирная линия;
- `dashed` — штриховая линия;
- `solid` — сплошная линия;
- `double` — двойная линия;
- `groove` — "вдавленная" трехмерная линия;
- `ridge` — "выпуклая" трехмерная линия;
- `inset` — трехмерная "выпуклость";
- `outset` — трехмерное "углубление".

Пример:

```
H1 { border-bottom-width: 5px
     border-bottom-color: red
     border-bottom-style: double }
```

Атрибут стиля `border-style` позволяет указать стиль сразу для всех сторон рамки:

```
border-style: <стиль 1> [<стиль 2> [<стиль 3> [<стиль 4>]]]
```

Он ведет себя так же, как и аналогичные атрибуты стиля `border-width` и `border-color`.

Пример:

```
TD, TH { border-width: thin;
         border-color: black;
         border-style: dotted }
```

Атрибуты стиля `border-left`, `border-top`, `border-right` и `border-bottom` позволяют задать все параметры для, соответственно, левой, верхней, правой и нижней стороны рамки:

```
border-left|border-top|border-right|border-bottom:
<толщина> <стиль> <цвет> | inherit
```

Во многих случаях эти атрибуты стиля оказываются предпочтительнее:

```
H1 { border-bottom: 5px double red }
```

"Глобальный" атрибут стиля `border` позволяет задать все параметры сразу для всех сторон рамки:

```
border: <толщина> <стиль> <цвет> | inherit
TD, TH { border: thin dotted black }
```

> ***ВНИМАНИЕ!***
>
> Рамки также увеличивают размеры элемента Web-страницы. Поэтому, если мы создадим рамки у блочных контейнеров, формирующих дизайн Web-страницы, то должны будем соответственно изменить размеры этих контейнеров, иначе они сместятся, и дизайн будет нарушен.

Представление для нашего Web-сайта, часть 6

Что ж, отступы и рамки мы создавать научились. По крайней мере, теоретически. Настала пора применить полученные знания на практике.

Прежде всего, сделаем отступы между контейнерами, формирующими дизайн наших Web-страниц, и между границами этих контейнеров и их содержимым.

- Внешний отступ между краями окна Web-обозревателя и содержимым Web-страницы равен нулю. Пусть пространство в окне Web-обозревателя используется максимально полно.

  > ***НА ЗАМЕТКУ***
  >
  > По умолчанию каждый Web-обозреватель создает свои отступы между краями своего окна и содержимым Web-страницы.

- Внутренние отступы в контейнере с заголовком Web-сайта (`cheader`) слева и справа — по 20 пикселов. Нам придется отодвинуть текст заголовка от краев окна Web-обозревателя, иначе заголовок будет выглядеть некрасиво.

- Внешний отступ между контейнерами с полосой навигации (`cnavbar`) и с основным содержимым (`cmain`) — 10 пикселов.
- Внутренние отступы у контейнера с основным содержимым (`cmain`) со всех сторон — по 5 пикселов.
- Внутренний отступ у контейнера с основным содержимым (`cmain`) сверху — 10 пикселов. Так мы отделим его от контейнеров `cnavbar` и `cmain`.
- Внутренние отступы в контейнере со сведениями об авторских правах (`ccopyright`) слева и справа — по 20 пикселов. Текст сведений об авторских правах также следует отодвинуть от краев окна Web-обозревателя.

Приведенные значения отступов получены автором в результате экспериментов. Вы можете задать другие.

Теперь разделим все четыре контейнера рамками.

- Контейнер `cheader` получит рамку с одной нижней стороной.
- Контейнер `cmain` — рамку с одной левой стороной.
- Контейнер `ccopyright` — рамку с одной верхней стороной.

Рамки мы сделаем тонкими и точечными. В качестве цвета зададим для них `#B1BEC6` (светло-синий).

В листинге 11.3 приведен исправленный фрагмент CSS-кода таблицы стилей main.css, реализующий выбранные нами параметры отступов и рамок.

Листинг 11.3

```
BODY          { color: #3B4043;
                background-color: #F8F8F8;
                font-family: Verdana, Arial, sans-serif;
                margin: 0px }
. . .
#cheader      { width: 1010px;
                padding: 0 20px;
                border-bottom: thin dotted #B1BEC6 }
#cnavbar      { width: 250px;
                height: 570px;
                float: left;
                margin-right: 10px }
#cmain        { width: 760px;
                height: 570px;
                float: left;
                overflow: auto;
                padding: 5px;
                border-left: thin dotted #B1BEC6 }
#ccopyright   { width: 1010px;
                clear: both;
                padding: 10px 20px 0px 20px;
                border-top: thin dotted #B1BEC6 }
```

Давайте разберем его.

Чтобы убрать отступ между границами окна Web-обозревателя и содержимым Web-страницы, мы использовали атрибут стиля `margin`. Его мы поместили в стиль переопределения тега `<BODY>` и дали ему значение 0 пикселов:

```
BODY          { color: #3B4043;
                background-color: #F8F8F8;
                font-family: Verdana, Arial, sans-serif;
                margin: 0px }
```

В именованном стиле `cheader`, привязанном к одноименному контейнеру, мы задали внутренние отступы слева и справа, равные 20 пикселам, и рамку с одной только нижней стороной. Эта рамка отделит данный контейнер от нижележащих:

```
#cheader      { width: 1010px;
                padding: 0 20px;
                border-bottom: thin dotted #B1BEC6 }
```

Кроме того, мы задали в качестве ширины этого контейнера абсолютное значение. Вспомним: при выводе на экран контейнера с относительной шириной Web-обозреватель сначала вычислит абсолютное значение его ширины, а потом добавит к нему величину отступов. В результате чего контейнер станет шире, чем окно Web-обозревателя, и в окне появятся полосы прокрутки, что нам совсем не нужно. Поэтому для ширины контейнера лучше задать абсолютное значение, подобрав его так, чтобы контейнер гарантированно поместился в окно Web-обозревателя по ширине.

В именованном стиле `cnavbar` мы указали внешний отступ справа 10 пикселов — он визуально отделит контейнер `cnavbar` от контейнера `cmain`:

```
#cnavbar      { width: 250px;
                height: 570px;
                float: left;
                margin-right: 10px }
```

В именованном стиле `cmain` мы задали внутренние отступы и рамку с одной левой стороной — она отделит контейнер `cmain` от контейнера `cnavbar`:

```
#cmain        { width: 760px;
                height: 570px;
                float: left;
                overflow: auto;
                padding: 5px;
                border-left: thin dotted #B1BEC6 }
```

В именованном стиле `ccopyright` мы задаем отступы слева и справа по 20 пикселов, а сверху — 10 пикселов. Кроме того, мы создаем рамку с одной верхней стороной, которая отделит контейнер `ccopyright` от вышерасположенных "соседей":

```
#ccopyright   { width: 1010px;
                clear: both;
```

```
                padding: 10px 20px 0px 20px;
                border-top: thin dotted #B1BEC6 }
```

Вот и все. Сохраним таблицу стилей main.css и откроем Web-страницу index.htm в Web-обозревателе. Автор не будет приводить здесь иллюстрацию, т. к. созданные нами тонкие рамки на ней практически незаметны. Но на экране компьютера они выглядят весьма эффектно.

Посмотрим теперь на полосу навигации. Невыразительные гиперссылки скучились в верхней части контейнера `cnavbar`, просто жалко на них смотреть!.. Давайте их сдвинем влево, немного "разредим", создав отступы, и заодно заключим каждую из них в рамки.

Как мы помним, наша полоса навигации представляет собой список, а отдельные ее гиперссылки — пункты данного списка.

Вот необходимые изменения:

- Список, формирующий полосу гиперссылок, сдвинуть влево на 30 пикселов. Так мы ликвидируем образовавшееся после удаления маркеров свободное пространство слева, которое смотрится некрасиво.
- Внешние отступы у пунктов списка сверху и снизу — 10 пикселов. Так мы отделим гиперссылки друг от друга.
- Рамка пунктов списка — тонкая, сплошная, цвет `#B1BEC6`.
- Внутренние отступы пунктов списка: сверху и снизу — по 5 пикселов, слева и справа — по 10 пикселов. Так мы отделим текст пунктов от рамок.

Осталось только соответственно исправить CSS-код таблицы стилей main.css (листинг 11.4).

Листинг 11.4

```
#navbar       { font-family: Arial, sans-serif;
                font-size: 14pt;
                text-transform: uppercase;
                list-style-type: none;
                margin-left: -30px }
. . .
#navbar LI    { padding: 5px 10px;
                margin: 10px 0px;
                border: thin solid #B1BEC6 }
```

Рассмотрим их подробнее.

В именованный стиль `navbar`, привязанный к тегу списка, который формирует полосу навигации, мы добавили отступ слева, равный –30 пикселов. Благодаря этому список сместится влево, заполняя пустое пространство:

```
#navbar       { font-family: Arial, sans-serif;
                font-size: 14pt;
```

```
                text-transform: uppercase;
                list-style-type: none;
                margin-left: -30px }
```

Вновь созданный комбинированный стиль создаст у пунктов списка, формирующего полосу навигации, рамку и задаст соответствующие отступы:

```
#navbar LI    { padding: 5px 10px;
                margin: 10px 0px;
                border: thin solid #B1BEC6 }
```

Сохраним исправленную таблицу стилей и обновим открытую в Web-обозревателе Web-страницу index.htm, нажав клавишу <F5>. Стало значительно лучше, не так ли (рис. 11.1)?

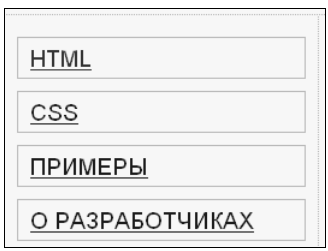


Рис. 11.1. Полоса навигации на Web-странице index.htm после применения к ней стилей

Теперь немного отвлечемся от CSS и займемся HTML. Есть у наших Web-страничек изъян, непростительный для хорошего Web-дизайнера.

Полная полоса навигации

Нет, в целом все хорошо. Только вот полоса навигации у нас какая-то куцая... Правила хорошего тона Web-дизайна предписывают, чтобы в полосе навигации находились гиперссылки, указывающие на все Web-страницы Web-сайта. А что у нас? Часть гиперссылок — да, находится именно там, а остальные скучены в последнем абзаце основного содержимого. Плохо!

Давайте срочно исправим ситуацию. Создадим новую, полную полосу навигации.

Поместим гиперссылки, указывающие на Web-страницы, которые описывают различные теги HTML, ниже гиперссылки, указывающие на Web-страницу index.htm (она посвящена самому HTML). Сделаем у этих гиперссылок небольшой отступ слева — так мы покажем, что они относятся к теме "HTML". То же самое проделаем и с гиперссылками, указывающими на Web-страницы, которые описывают атрибуты стиля CSS и примеры.

Наша полоса навигации — суть список. Ее гиперссылки — пункты этого списка. Поэтому для гиперссылок, указывающих на Web-страницы — описания тегов, логично создать вложенный список. Благо HTML, как мы узнали еще в *главе 2*, позволяет делать вложенные списки.

Откроем в Блокноте Web-страницу index.htm и найдем HTML-код, создающий полосу навигации (листинг 11.5).

Листинг 11.5

```
<UL ID="navbar">
  <LI><A HREF="index.htm">HTML</A></LI>
  <LI><A HREF="css_index.htm">CSS</A></LI>
  <LI><A HREF="samples_index.htm">Примеры</A></LI>
  <LI><A HREF="about.htm">О разработчиках</A></LI>
</UL>
```

Дополним его кодом, создающим вложенный список. Листинг 11.6 содержит исправленный код.

Листинг 11.6

```
<UL ID="navbar">
  <LI><A HREF="index.htm">HTML</A>
    <UL>
    </UL>
  </LI>
  <LI><A HREF="css_index.htm">CSS</A></LI>
  <LI><A HREF="samples_index.htm">Примеры</A></LI>
  <LI><A HREF="about.htm">О разработчиках</A></LI>
</UL>
```

Мы поместили тег `<UL>`, создающий вложенный список, в разрыв между текстом первого пункта "внешнего" списка и его закрывающим тегом `</LI>`. Тогда вложенный список будет располагаться ниже первого пункта "внешнего" списка.

После этого найдем HTML-код, создающий гиперссылки на Web-страницы — описания тегов. Он находится в самом конце кода Web-страницы, в отдельном абзаце контейнера `cmain`. Позаимствуем оттуда фрагменты кода, которые создают отдельные гиперссылки, указывающие на Web-страницы — описания тегов, и создадим на их основе пункты вложенного списка (листинг 11.7).

Листинг 11.7

```
<UL ID="navbar">
  <LI><A HREF="index.htm">HTML</A>
    <UL>
      <LI><CODE>!DOCTYPE</CODE></LI>
      <LI><CODE><A HREF="tags/t_audio.htm">AUDIO</A></CODE></LI>
      <LI><CODE>BODY</CODE></LI>
      . . .
      <LI><CODE><A HREF="tags/t_video.htm">VIDEO</A></CODE></LI>
    </UL>
  </LI>
  <LI><A HREF="css_index.htm">CSS</A></LI>
```

```
  <LI><A HREF="samples_index.htm">Примеры</A></LI>
  <LI><A HREF="about.htm">О разработчиках</A></LI>
</UL>
```

Удалим из контейнера `cmain` код, который создает абзац с гиперссылками, указывающими на Web-страницы — описания тегов, и относящийся к нему заголовок. Надобности в них больше нет.

Сохраним исправленную Web-страницу index.htm и откроем ее в Web-обозревателе. Ниже пункта "HTML" "внешнего" списка, формирующего полосу навигации, мы увидим пункты только что созданного вложенного списка.

Вложенный список в полосе навигации выглядит просто ужасно. Поэтому, не откладывая дела в долгий ящик, приступим к созданию для него представления.

Поскольку мы существенно увеличили размер полосы навигации, может наступить такой момент, когда она станет больше контейнера `cnavbar`. Так что первым делом зададим для этого контейнера подходящее поведение при переполнении, дополнив соответствующий ему именованный стиль (листинг 11.8).

Листинг 11.8

```
#cnavbar      { width: 250px;
                height: 570px;
                float: left;
                overflow: auto;
                margin-right: 10px }
```

Теперь сформулируем требования к вложенному списку и его пунктам.

- Маркер пунктов вложенного списка — отсутствует.
- Вложенный список, формирующий полосу гиперссылок, сдвинуть влево на 30 пикселов. Так мы ликвидируем образовавшееся после удаления маркеров свободное пространство.
- Внешний отступ у вложенного списка сверху — 10 пикселов. Так мы отделим его от пункта "внешнего" списка, в который он вложен.
- Размер шрифта у пунктов вложенного списка — 10 пунктов. Раз уж вложенный список является как бы "подчиненным", то пусть его пункты будут набраны шрифтом меньшего размера.
- Отступы и рамки у пунктов вложенного списка отсутствуют. Так мы сделаем вложенный список компактнее.

Чтобы реализовать эти требования, допишем в конец таблицы стилей main.css CSS-код листинга 11.9.

Листинг 11.9

```
#navbar LI UL    { list-style-type: none;
                   margin-left: -20px;
                   margin-top: 10px }
```

```
#navbar LI UL LI { font-size: 12pt;
                   padding: 0px;
                   margin: 0px;
                   border-style: none }
```

Мы создали два комбинированных стиля. Первый задает параметры вложенного списка. Там комментировать нечего.

Второй комбинированный стиль задает параметры пунктов вложенного списка. Отметим, что в нем мы явно задали величины внешних и внутренних отступов, равные нулю, и отсутствие рамки. Если мы этого не сделаем, к пунктам вложенного списка будут применены параметры, которые мы задали для пунктов "внешнего" списка, и у пунктов вложенного списка также появятся отступы и рамки. Что нам совсем не нужно.

Сохраним таблицу стилей main.css и обновим открытую в Web-обозревателе Web-страницу index.htm, нажав клавишу <F5>. Полоса навигации должна выглядеть так, как показано на рис. 11.2.

А что, получилось неплохо! По-деловому строго и, вместе с тем, симпатично.

На этом закончим с HTML и содержимым Web-страниц. И снова вернемся к CSS и представлению.

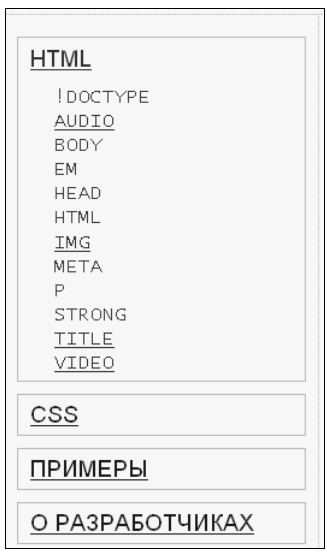


Рис. 11.2. Полная полоса навигации на Web-странице index.htm

Параметры выделения

Мы знаем множество способов привлечь внимание посетителя к определенным элементам Web-страниц, использовав теги HTML или атрибуты стиля CSS. Но CSS 3 предлагает нам еще один способ сделать это — так называемое *выделение*. Именно о нем сейчас и пойдет речь.

- Выделение CSS 3 представляет собой рамку, которой окружается данный элемент Web-страницы.
- Мы можем задавать параметры выделения: толщину, цвет и стиль.
- Выделение, в отличие от знакомой нам рамки CSS, не увеличивает размеры элемента Web-страницы. Так что можно спокойно применять выделение, не опасаясь, что оно нарушит выстраданный нами контейнерный дизайн.

Для задания параметров выделения CSS 3 предназначено четыре специальных атрибута стиля. Сейчас мы их рассмотрим.

Атрибут стиля `outline-width` задает толщину рамки выделения:

```
outline-width: thin|medium|thick|<толщина>|inherit
```

Здесь доступны те же значения, что и для знакомого нам атрибута стиля `border-width`.

Пример:

```
DFN { outline-width: thin }
```

Здесь мы задали для содержимого тега `<DFN>` тонкую рамку выделения.

Атрибут стиля `outline-color` задает цвет рамки выделения:

```
outline-color: <цвет>|inherit
```

> ***Внимание!***
>
> Цвет рамки выделения всегда следует задавать явно — в противном случае оно может быть не нарисовано.

Пример:

```
DFN { outline-width: thin;
      outline-color: black }
```

Теперь выделение содержимого тега `<DFN>` будет выведено черным цветом.

Атрибут стиля `outline-style` задает стиль линий, которыми будет нарисована рамка выделения:

```
outline-style: none|dotted|dashed|solid|double|groove|ridge|inset|
outset|inherit
```

Значения здесь доступны те же, что и для атрибута стиля `border-style`.

Пример:

```
DFN { outline-width: thin;
      outline-color: black;
      outline-style: dotted }
```

Атрибут стиля `outline` позволяет задать сразу все параметры для рамки выделения:

```
outline: <толщина> <стиль> <цвет> | inherit
DFN { outline: thin dotted black }
```

А что, это идея! Давайте добавим в нашу таблицу стилей main.css вот такой стиль:

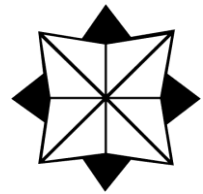
```
DFN           { outline: thin dotted #B1BEC6 }
```

После этого все новые термины (т. е. содержимое тегов `<DFN>`) на наших Web-страницах будут выделены тонкой точечной рамкой светло-синего цвета.

Что дальше?

В этой главе мы научились создавать у элементов Web-страницы отступы и рамки. И доделали представление наших Web-страниц, придав им профессиональный лоск.

В следующей главе мы познакомимся с возможностями CSS, которые задают параметры таблиц. Их будет совсем немного.



ГЛАВА 12

Параметры таблиц

В предыдущей главе мы научились создавать у элементов Web-страниц отступы, рамки и выделение и применили свои умения на практике, сделав наши Web-страницы еще привлекательнее. Заодно мы наконец-то создали полную полосу навигации, включающую гиперссылки на все Web-страницы нашего Web-сайта. Теперь посетителю будет намного удобнее "путешествовать" по нему.

В этой главе мы разберемся с атрибутами стиля CSS для задания различных параметров таблиц. Часть из них мы уже изучили в предыдущих главах, а с некоторыми познакомимся только сейчас.

Параметры выравнивания

Для выравнивания содержимого ячеек таблицы по горизонтали мы применим атрибут стиля `text-align`, описанный в *главе 9*:

```
TD, TH { text-align: center }
```

Этот же атрибут стиля пригоден для выравнивания текста в заголовке таблицы (теге `<CAPTION>`):

```
CAPTION { text-align: left }
```

Содержимое ячеек таблиц по вертикали мы выровняем с помощью атрибута стиля `vertical-align`:

```
vertical-align: baseline|sub|super|top|text-top|middle|bottom|
text-bottom|<промежуток между базовыми линиями>|inherit
```

Применительно к другим элементам Web-страниц он был описан в *главе 8*, но в случае ячеек таблиц ведет себя несколько по-другому.

- `top` — выравнивает содержимое ячейки по ее верхнему краю (обычное поведение).
- `middle` — выравнивает содержимое ячейки по ее центру.
- `bottom` — выравнивает содержимое ячейки по ее нижнему краю.

Остальные значения этого атрибута стиля действуют так же, как и для других элементов Web-страниц (см. *главу 8*):

```
TD, TH { vertical-align: middle }
```

Параметры отступов и рамок

Для задания отступов мы можем пользоваться атрибутами стиля, знакомыми нам по *главе 11*.

- Для задания внутренних отступов между содержимым ячейки и ее границей — атрибутами стиля `padding-left`, `padding-top`, `padding-right`, `padding-bottom` и `padding`.
- Для задания внешних отступов между границами соседних ячеек — атрибутом стиля `border-spacing`.

Параметры рамок зададим через соответствующие атрибуты стиля, которые также знакомы нам по *главе 11* (листинг 12.1).

Листинг 12.1

```
TABLE  { align: center;
         border: medium solid black;
         border-spacing: 1px }
TD, TH { border: thin dotted black;
         padding: 2px }
```

В листинге 12.1 мы назначили для самой таблицы тонкую сплошную черную рамку и отступ между ячейками, равный одному пикселу, а для ячеек этой таблицы — тонкую точечную черную рамку и отступ между границей ячейки и ее содержимым, равный двум пикселам.

Если мы зададим рамки вокруг ячеек таблицы, Web-обозреватель нарисует рамку вокруг каждой ячейки. Такая таблица будет выглядеть как набор прямоугольников-ячеек, заключенный в большой прямоугольник-таблицу (рис. 12.1).

Версия HTML	Год выхода	Особенности
1.0	1992	Официально не была стандартизована
2.0	1995	Первая стандартизованная версия
3.2	1997	Поддержка таблиц и графики
4.0	1997	"Очищен" от устаревших тегов
4.01	1999	В основном, исправление ошибок
5.0	?	В разработке

Рис. 12.1. Обычное поведение Web-обозревателя — рамки рисуются вокруг каждой ячейки таблицы

Однако в печатных изданиях гораздо чаще встречаются таблицы другого вида. В них рамки присутствуют только между ячейками (рис. 12.2).

Версия HTML	Год выхода	Особенности
1.0	1992	Официально не была стандартизована
2.0	1995	Первая стандартизованная версия
3.2	1997	Поддержка таблиц и графики
4.0	1997	"Очищен" от устаревших тегов
4.01	1999	В основном, исправление ошибок
5.0	?	В разработке

Рис. 12.2. Таблица, в которой рисуются только рамки, разделяющие ячейки

Атрибут стиля `border-collapse` указывает Web-обозревателю, как будут рисоваться рамки ячеек в таблице:

```
border-collapse: collapse|separate|inherit
```

- `separate` — каждая ячейка таблицы заключается в отдельную рамку (см. рис. 12.1). Это обычное поведение.
- `collapse` — рисуются рамки, разделяющие ячейки таблицы (см. рис. 12.2).

Данный атрибут стиля применяется только к самим таблицам (тегам `<TABLE>`).

Пример:

```
TABLE { border-collapse: collapse }
```

Параметры размеров

Для задания размеров — ширины и высоты — таблиц и их ячеек подойдут атрибуты стиля `width` и `height`, описанные в *главе 10*.

- Если требуется задать ширину или высоту всей таблицы, нужный атрибут стиля указывают именно для нее:

```
TABLE { width: 100%;
        height: 300px }
```

- Если требуется задать ширину столбца, атрибут стиля `width` указывают для первой ячейки, входящей в этот столбец (листинг 12.2).

Листинг 12.2

```
<TABLE>
  <TR>
    <TH>Первый столбец</TH>
    <TH STYLE="width: 40px">Второй столбец шириной в 40 пикселов</TH>
    <TH>Третий столбец</TH>
  </TR>
  . . .
</TABLE>
```

- Если требуется задать высоту строки, атрибут стиля `height` указывают для первой ячейки этой строки (листинг 12.3).

Листинг 12.3

```
<TABLE>
  . . .
  <TR>
    <TD STYLE="height: 30px">Строка высотой в 30 пикселов</TD>
    . . .
  </TR>
  . . .
</TABLE>
```

Обычно все размеры, которые мы зададим для таблицы и ее ячеек, — не более чем рекомендация для Web-обозревателя. Если содержимое таблицы не будет в ней помещаться, Web-обозреватель увеличит ширину или высоту таблицы. Зачастую это может быть неприемлемо, поэтому стандарт CSS предусматривает средства, позволяющие изменить такое поведение Web-обозревателя.

Атрибут стиля `table-layout` позволяет указать, как Web-обозреватель будет трактовать размеры, заданные нами для таблицы и ее ячеек:

```
table-layout: auto|fixed|inherit
```

- `auto` — Web-обозреватель может изменить размеры таблицы и ее ячеек, если содержимое в них не помещается. Это обычное поведение.
- `fixed` — размеры таблицы и ее ячеек ни в коем случае изменяться не будут. Если содержимое в них не помещается, возникнет переполнение, параметры которого мы можем задавать с помощью атрибутов стиля `overflow`, `overflow-x` и `overflow-y` (см. *главу 10*).

Данный атрибут стиля применяется к самой таблице (тегу `<TABLE>`).

Пример:

```
TABLE { table-layout: fixed;
        overflow: auto }
```

Прочие параметры

И еще несколько полезных атрибутов стиля.

Атрибут стиля `caption-side` указывает местоположение заголовка таблицы относительно самой таблицы:

```
caption-side: top|bottom|inherit
```

- `top` — заголовок располагается над таблицей (обычное поведение).
- `bottom` — заголовок располагается под таблицей.

Данный атрибут стиля применяется к самой таблице (тегу `<TABLE>`).

Пример:

```
TABLE { caption-side: bottom }
```

Атрибут стиля `empty-cells` указывает, как Web-обозреватель должен выводить на экран пустые (не имеющие содержимого) ячейки:

```
empty-cells: show|hide|inherit
```

- `show` — пустые ячейки будут выводиться на экран. Если для них был задан другой фон, на экран будет выведен фон, а если заданы рамки, будут выведены рамки.
- `hide` — пустые ячейки не будут выводиться на экран.

Обычное поведение зависит от Web-обозревателя, так что, если это критично, лучше явно задать нужное значение атрибута стиля `empty-cells`.

Атрибут стиля `empty-cells` также применяется к самой таблице (тегу `<TABLE>`).

Пример:

```
TABLE { empty-cells: hide }
```

Представление для нашего Web-сайта, часть 7

Вот, собственно, и все атрибуты стиля, задающие параметры таблиц. Теоретическая часть данной главы оказалась совсем короткой...

В качестве практики давайте поработаем над нашей единственной таблицей — списком версий HTML. Сделаем ее более удобочитаемой.

Сначала, как обычно, сформулируем перечень ее параметров.

- Внешние отступы сверху и снизу таблицы — 10 пикселов. Пусть таблица будет визуально отделена от "соседей".
- Рамка вокруг самой таблицы — тонкая, сплошная, цвет `#B1BEC6`.
- Будут выводиться только рамки, разделяющие ячейки. Таблицы с такими рамками более привычны.
- Внутренние отступы в ячейках — 2 пиксела.
- Рамки ячеек — тонкие, точечные, цвет `#B1BEC6`.
- Выравнивание текста заголовка таблицы — по левому краю.

Осталось написать CSS-код. Листинг 12.4 содержит исправленный фрагмент таблицы стилей main.css.

Листинг 12.4

```
TABLE         { font-size: 10pt;
                margin: 10px 0px;
                border: thin solid #B1BEC6;
                border-collapse: collapse }
. . .
TD, TH        { padding: 2px;
                border: thin dotted #B1BEC6 }
CAPTION       { text-align: left }
```

Здесь мы дополнили стиль переопределения тега <TABLE> и создали стили переопределения тегов <TD>, <TH> и <CAPTION>. Они столь просты, что не требуют комментариев.

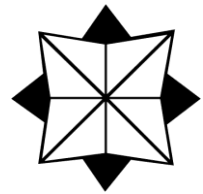
Сохраним таблицу стилей main.css и откроем Web-страницу index.htm в Web-обозревателе. И полюбуемся на таблицу. Рамки и отступы явно пошли ей на пользу.

Что дальше?

В этой главе мы научились задавать параметры таблиц, используя изученные ранее атрибуты стиля, а также познакомились с новыми атрибутами стиля, специфичными именно для таблиц. И привели нашу единственную таблицу в удобный для чтения вид.

В следующей главе мы завершим изучение CSS рассмотрением специальных селекторов. Это невероятно мощное средство, позволяющее привязать стиль к нужному элементу Web-страницы неявно, не пользуясь ни стилевыми классами, ни именованными стилями, предлагает нам стандарт CSS 3. Который все никак не примут...

ГЛАВА 13



Специальные селекторы

В предыдущих главах *части II* мы изучали, в основном, атрибуты стилей CSS. Их очень много; одни задают параметры шрифта, другие — параметры фона, третьи — параметры отступов и пр. Можно сказать, что нет такого параметра, влияющего на представление элементов Web-страницы, который мы не могли бы задать, пользуясь средствами CSS.

В этой главе, последней в данной части, мы поговорим о селекторах стилей. Да-да, тех самых селекторах, с которыми мы познакомились еще в *главе 7* и о которых, казалось, знаем все (ну, или почти все). Стили переопределения тегов, стилевые классы, именованные и комбинированные стили — что там может быть нового?

Может. Специальные селекторы.

Специальный селектор — это селектор, неявно привязывающий стиль к элементу Web-страницы согласно более сложному, чем имя тега, критерию. Таким критерием может быть, например, порядковый номер элемента в родительском элементе, указание на определенную часть содержимого абзаца (первую строку или первую букву), состояние гиперссылки (посещена она или нет) и пр. Обычно специальные селекторы используют в комбинированных стилях, чтобы сделать их более конкретными.

Существует несколько разновидностей специальных селекторов, которые сейчас и рассмотрим.

Комбинаторы

Комбинаторы — разновидность специальных селекторов, привязывающая стиль к элементу Web-страницы на основании его местоположения относительно других элементов.

Комбинатор + позволяет привязать стиль к элементу Web-страницы, непосредственно следующему за другим элементом. При этом оба дочерних элемента должны быть вложенными в один и тот же родительский:

```
<элемент 1> + <элемент 2> { <стиль> }
```

Стиль будет привязан к *элементу 2*, который должен непосредственно следовать за *элементом 1*.

Листинг 13.1

```
H6 + PRE { font-size: smaller }
. . .
<H6>Это заголовок</H6>
<PRE>Этот текст будет набран уменьшенным шрифтом.</PRE>
<P>А этот — обычным шрифтом.</P>
<H6>Это заголовок</H6>
<P>И этот — обычным шрифтом.</P>
<PRE>И этот — обычным шрифтом.</PRE>
```

Стиль, описанный в листинге 13.1, будет применен только к первому тексту фиксированного форматирования, т. к. он непосредственно следует за заголовком шестого уровня.

Комбинатор ~ (тильда) позволяет привязать стиль к элементу Web-страницы, следующему за другим элементом и, возможно, отделенному от него другими элементами. При этом оба дочерних элемента должны быть вложенными в один и тот же родительский:

```
<элемент 1> ~ <элемент 2> { <стиль> }
```

Стиль будет привязан к *элементу 2*, который должен следовать за *элементом 1*. При этом *элемент 2* может быть отделен от *элемента 1* другими элементами.

Листинг 13.2

```
H6 + PRE { font-size: smaller }
. . .
<H6>Это заголовок</H6>
<PRE>Этот текст будет набран уменьшенным шрифтом.</PRE>
<P>А этот — обычным шрифтом.</P>
<H6>Это заголовок</H6>
<P>И этот — обычным шрифтом.</P>
<PRE>А этот — уменьшенным шрифтом.</PRE>
```

Стиль из листинга 13.2 будет применен к обоим текстам фиксированного формата: и тому, что непосредственно следует за заголовком шестого уровня, и тому, который отделен от заголовка абзацем.

Комбинатор > позволяет привязать стиль к элементу Web-страницы, непосредственно вложенному в другой элемент:

```
<элемент 1> > <элемент 2> { <стиль> }
```

Стиль будет привязан к *элементу 2*, который непосредственно вложен в *элемент 1*.

Листинг 13.3

```
BLOCKQUOTE + P { font-style: italic }
. . .
<BLOCKQUOTE>
  <P>Этот абзац будет набран курсивом.</P>
  <DIV>
    <P>А этот абзац — обычным шрифтом.</P>
  </DIV>
</BLOCKQUOTE>
```

Стиль из листинга 13.3 будет применен только к абзацу, непосредственно вложенному в большую цитату. На второй абзац, последовательно вложенный в большую цитату и блочный контейнер, этот стиль действовать не будет.

Комбинатор <пробел> позволяет привязать стиль к элементу Web-страницы, вложенному в другой элемент, причем не обязательно непосредственно:

<элемент 1> <элемент 2> { *<стиль>* }

Стиль будет привязан к *элементу 2*, который так или иначе вложен в *элемент 1*. При этом *элемент 2* может быть вложен в другой элемент, вложенный в *элемент 1*, или даже в несколько таких элементов последовательно.

Листинг 13.4

```
BLOCKQUOTE P { font-style: italic }
. . .
<BLOCKQUOTE>
  <P>Этот абзац будет набран курсивом.</P>
  <DIV>
    <P>Этот абзац — тоже.</P>
  </DIV>
</BLOCKQUOTE>
```

Стиль из листинга 13.4 будет применен к обоим абзацам: и вложенному непосредственно в большую цитату, и тому, что последовательно вложен в большую цитату и блочный контейнер.

Стиль, приведенный в листинге 13.4, нам уже знаком. Да это ведь комбинированный стиль, изученный нами еще в *главе 7*! Получается, что он тоже относится к комбинаторам. А мы и не знали...

Селекторы по атрибутам тега

Селекторы по атрибутам тега — это специальные селекторы, привязывающие стиль к тегу на основании, присутствует ли в нем определенный атрибут или имеет ли он определенное значение.

Селекторы по атрибутам тега используются не сами по себе, а только в совокупности со стилями переопределения тега или комбинированными стилями. Их записывают сразу после основного селектора без всяких пробелов и берут в квадратные скобки.

Селектор вида

```
<основной селектор>[<имя атрибута тега>] { <стиль> }
```

привязывает *стиль* к элементам, теги которых имеют атрибут с указанным *именем*.

Пример:

```
TD[ROWSPAN] { background-color: grey }
```

Этот стиль будет привязан к ячейкам таблицы, теги которых имеют атрибут ROWSPAN, т. е. к ячейкам, объединенным по горизонтали.

Селектор вида

```
<основной селектор>[<имя атрибута тега>=<значение>] { <стиль> }
```

привязывает *стиль* к элементам, теги которых имеют атрибут с указанными *именем* и *значением*.

Пример:

```
TD[ROWSPAN=2] { background-color: grey }
```

Данный стиль будет привязан к ячейкам таблицы, теги которых имеют атрибут ROWSPAN со значением 2, т. е. к двойным ячейкам, объединенным по горизонтали.

Селекторы вида

```
<основной селектор>[<имя атрибута тега>~=
<список значений, разделенных пробелами>] { <стиль> }
```

и

```
<основной селектор>[<имя атрибута тега>|=
<список значений, разделенных запятыми>] { <стиль> }
```

привязывает *стиль* к элементам, теги которых имеют атрибут с указанным *именем* и значением, совпадающим с одним из указанных в *списке*.

Пример:

```
TD[ROWSPAN~=2 3] { background-color: grey }
TD[ROWSPAN|=2,3] { border: thin dotted black }
```

Эти стили будут привязаны к ячейкам таблицы, теги которых имеют атрибут ROWSPAN со значениями 2 и 3, т. е. к двойным и тройным ячейкам, объединенным по горизонтали.

Селектор вида

```
<основной селектор>[<имя атрибута тега>^=<подстрока>] { <стиль> }
```

привязывает *стиль* к элементам, теги которых имеют атрибут с указанным *именем* и значением, начинающимся с указанной *подстроки*.

Пример:

```
IMG[SRC^=http://www.pictures.ru] { margin: 5px }
```

Этот стиль будет привязан к графическим изображениям, теги которых имеют атрибут SRC со значением, начинающимся с подстроки "http://www.pictures.ru", т. е. к изображениям, взятым с Web-сайта **http://www.pictures.ru**.

Селектор вида

```
<основной селектор>[<имя атрибута тега>$=<подстрока>] { <стиль> }
```

привязывает *стиль* к элементам, теги которых имеют атрибут с указанным *именем* и значением, заканчивающимся указанной *подстрокой*.

Пример:

```
IMG[SRC$=gif] { margin: 10px }
```

Данный стиль будет привязан к графическим изображениям, теги которых имеют атрибут SRC со значением, заканчивающимся подстрокой "gif", т. е. к изображениям формата GIF.

Селектор вида

```
<основной селектор>[<имя атрибута тега>*=<подстрока>] { <стиль> }
```

привязывает *стиль* к элементам, теги которых имеют атрибут с указанным *именем* и значением, включающим указанную *подстроку*.

Пример:

```
IMG[SRC*=/picts/] { margin: 10px }
```

Этот стиль будет привязан к графическим изображениям, теги которых имеют атрибут SRC со значением, включающим подстроку "/picts/", т. е. к изображениям, взятым из папки picts Web-сайта, откуда они загружены.

Псевдоэлементы

Псевдоэлементы — разновидность специальных селекторов, привязывающих стиль к определенному фрагменту элемента Web-страницы. Таким фрагментом может быть первый символ или первая строка в абзаце.

Псевдоэлементы используются не сами по себе, а только в совокупности с другими стилями. Их записывают сразу после основного селектора без всяких пробелов:

```
<основной селектор><псевдоэлемент> { <стиль> }
```

Псевдоэлемент `::first-letter` привязывает стиль к первой букве текста в элементе Web-страницы, если ей не предшествует встроенный элемент, не являющийся текстом, например, изображение.

Пример:

```
P::first-letter { font-size: larger }
```

Этот стиль будет привязан к первой букве абзаца.

Псевдоэлемент `::first-line` привязывает стиль к первой строке текста в элементе Web-страницы:

```
P::first-line { text-transform: uppercase }
```

Данный стиль будет привязан к первой строке абзаца.

Псевдоклассы

Псевдоклассы — самая мощная разновидность специальных селекторов. Они позволяют привязать стиль к элементам Web-страницы на основе их состояния (наведен на них курсор мыши или нет) и местоположения в родительском элементе.

Псевдоклассы также используются не сами по себе, а только в совокупности с другими стилями. Их записывают сразу после основного селектора без всяких пробелов:

```
<основной селектор><псевдокласс> { <стиль> }
```

Псевдоклассы, в свою очередь, сами делятся на группы. Каждой из них мы посвятим отдельный раздел.

Псевдоклассы гиперссылок

Псевдоклассы гиперссылок служат для привязки стилей к гиперссылкам на основе их состояния: является гиперссылка посещенной или непосещенной, щелкает на ней посетитель мышью или только навел на нее курсор мыши и др.

Все псевдоклассы гиперссылок, доступные в стандарте CSS 3:

- `:link` — непосещенная гиперссылка;
- `:visited` — посещенная гиперссылка;
- `:active` — гиперссылка, на которой посетитель в данный момент щелкает мышью;
- `:focus` — гиперссылка, имеющая фокус ввода (подробнее о фокусе ввода см. в *главе 6*);
- `:hover` — гиперссылка, на которую наведен курсор мыши.

Псевдоклассы гиперссылок применяются совместно со стилями, задающими параметры для гиперссылок. Это могут быть стили переопределения тега `<A>` или комбинированные стили, созданные на основе стиля переопределения тега `<A>` (листинг 13.5).

Листинг 13.5

```
A:link    { text-decoration: none }
A:visited { text-decoration: overline }
A:active  { text-decoration: underline }
A:focus   { text-decoration: underline }
A:hover   { text-decoration: underline }
```

В листинге 13.5 псевдоклассы гиперссылок действуют совместно со стилями переопределения тега <A>. В результате приведенные стили будут применены ко всем гиперссылкам на Web-странице.

Листинг 13.6

```
A.special:link    { color: darkred }
A.special:visited { color: darkviolet }
A.special:active  { color: red }
A.special:focus   { color: red }
A.special:hover   { color: red }
```

В листинге 13.6 псевдоклассы гиперссылок совмещены с комбинированными стилями, объединяющими стиль переопределения тега <A> и стилевой класс special. Они будут применены только к тем гиперссылкам, к которым был привязан указанный стилевой класс.

Псевдоклассы гиперссылок можно комбинировать, записывая их друг за другом:

```
A:visited:hover { text-decoration: underline }
```

Этот стиль будет применен к посещенной гиперссылке, над которой находится курсор мыши.

Псевдоклассы гиперссылок — единственное средство, позволяющее указать параметры для текста гиперссылок. По крайней мере, насколько удалось выяснить автору...

Структурные псевдоклассы

Структурные псевдоклассы позволяют привязать стиль к элементу Web-страницы на основе его местоположения в родительском элементе.

Псевдоклассы :first-child и :last-child привязывают стиль к элементу Web-страницы, который является, соответственно, первым и последним дочерним элементом своего родителя:

```
UL:first-child { font-weight: bold }
```

Этот стиль будет применен к пункту, являющемуся первым дочерним элементом своего родителя-списка, т. е. к первому пункту списка.

Пример:

```
TD:last-child { color: green }
```

Данный стиль будет применен к последнему дочернему элементу каждой строки таблицы — к ее последней ячейке.

Листинг 13.7 иллюстрирует более интересный случай.

Листинг 13.7

```
#cmain P:first-child { font-weight: bold }
. . .
```

```
<DIV ID="cmain">
  <P>Этот абзац будет набран зеленым цветом.</P>
  <BLOCKQUOTE>
    <P>Этот абзац — тоже.</P>
  </BLOCKQUOTE>
  <BLOCKQUOTE>
    <P>И этот — тоже.</P>
    <P>А этот — нет.</P>
  </BLOCKQUOTE>
</DIV>
```

Стиль, приведенный в листинге 13.7, будет применен к самому первому абзацу, единственному абзацу в первой большой цитате и первому абзацу во второй большой цитате. Дело в том, что последние два абзаца, к которым будет применен стиль, отсчитываются относительно своих родителей — больших цитат — и в них являются первыми.

А если мы изменим данный стиль вот так:

```
#cmain > P:first-child { font-weight: bold }
```

он будет применен только к самому первому абзацу, непосредственно вложенному в контейнер `cmain`. Ведь мы указали комбинатор `>`, который предписывает, чтобы элемент, к которому применяется стиль, должен быть непосредственно вложен в своего родителя.

> ***НА ЗАМЕТКУ***
>
> Стандарт CSS также описывает псевдоклассы `:first-of-type` и `:last-of-type`. Насколько удалось выяснить автору, они работают так же, как и псевдоклассы `:first-child` и `:last-child`, которые мы только что изучили. По крайней мере, в текущей реализации поддержки CSS 3...

Псевдокласс `:only-of-type` привязывает стиль к элементу Web-страницы, который является единственным дочерним элементом, сформированном с помощью данного тега, в своем родителе.

Листинг 13.8 иллюстрирует пример.

Листинг 13.8

```
P:only-of-type { color: green }
. . .
<BLOCKQUOTE>
  <P>Этот абзац будет набран зеленым цветом.</P>
</BLOCKQUOTE>
<BLOCKQUOTE>
  <P>И этот абзац будет набран зеленым цветом.</P>
  <ADDRESS>А этот текст — нет.</ADDRESS>
</BLOCKQUOTE>
```

```
<BLOCKQUOTE>
  <P>И этот — нет.</P>
  <P>И этот — нет.</P>
</BLOCKQUOTE>
```

Стиль из листинга 13.8 будет применен к абзацам, вложенным в первую и вторую большие цитаты, т. к. эти абзацы являются там единственными элементами, сформированными с помощью тега `<P>`. К абзацам, вложенным в третью большую цитату, стиль применен не будет.

Псевдокласс `:nth-child` позволяет привязать стиль к элементам Web-страницы, выбрав их по порядковым номерам, под которыми эти элементы определены в своем родителе:

```
<основной селектор>:nth-child(<a>n+<b>) { <стиль> }
```

После имени данного псевдокласса в скобках указывают формулу, по которой вычисляются номера элементов, к которым будет применен стиль. Эта формула имеет два параметра, задаваемых Web-дизайнером: *a* и *b*. Их значения должны представлять собой неотрицательные целые числа.

Рассмотрим, как выполняется привязка стиля, включающего псевдокласс `:nth-child`.

Сначала Web-обозреватель считывает CSS-код стиля и, руководствуясь его селектором, находит элементы Web-страницы, к которым, возможно, будет применен данный стиль. Также Web-обозреватель определяет родителя этих элементов.

После этого Web-обозреватель разбивает все найденные элементы на группы. Количество элементов в каждой группе задается параметром *a* приведенной формулы. После разбиения Web-обозреватель вычисляет количество получившихся групп.

Далее Web-обозреватель последовательно подставляет в формулу вместо *n* номера получившихся групп, начиная с нуля. В результате каждого прохода вычисления получается номер элемента, к которому применяется стиль.

Для примера создадим таблицу из пяти строк и применим к ней такой стиль:

```
TR:nth-child(2n+1) { text-align: center }
```

В стиле мы указали, что группа должна содержать два дочерних элемента. Тогда Web-обозреватель разобьет строки таблицы на две группы, по две строки в каждой.

- Web-обозреватель подставит в формулу `n` равное 0. После вычисления получится значение 1. Web-обозреватель применит данный стиль к первой строке таблицы.
- Web-обозреватель подставит в формулу `n` равное 1. После вычисления получится значение 3. Web-обозреватель применит данный стиль к третьей строке таблицы.
- Web-обозреватель подставит в формулу `n` равное 2. После вычисления получится значение 5. Web-обозреватель применит данный стиль к пятой строке таблицы. Поскольку 2 — общее количество групп, то на этом вычисления закончатся.

В результате данный стиль будет применен к каждой нечетной строке нашей таблицы.

Если мы создадим такой стиль:

```
TR:nth-child(2n+0) { text-align: center }
```

то он будет применен ко второй и четвертой (четным) строкам нашей таблицы.

Кстати, мы можем сделать запись чуть короче:

```
TR:nth-child(2n) { text-align: center }
```

Ранее мы рассматривали примеры с разбиением дочерних элементов на группы. Но мы можем отменить разбиение, указав нулевое количество элементов в группе. В этом случае Web-обозреватель найдет *b*-й дочерний элемент и применит стиль к нему.

Так, если мы создадим такой стиль:

```
TR:nth-child(0n+2) { text-align: center }
```

то Web-обозреватель применит его ко второй строке таблицы.

Мы можем записать данный стиль и так:

```
TR:nth-child(2) { text-align: center }
```

и Web-обозреватель правильно его обработает.

Вместо указания формулы в скобках мы можем поставить там предопределенные значения `odd` и `even`. Первое привязывает стиль к нечетным дочерним элементам, второе — к четным:

```
TR:nth-child(odd) { background-color: grey }
TR:nth-child(even) { background-color: yellow }
```

Первый стиль будет применен к нечетным строкам таблицы, второй — к четным.

Псевдокласс `:nth-last-child` аналогичен рассмотренному нами псевдоклассу `:nth-child` за тем исключением, что отсчет дочерних элементов ведется не с начала, а с конца родительского элемента.

Пример:

```
TR:nth-last-child(1) { text-align: center }
```

Данный стиль будет применен к последней строке таблицы.

Пример:

```
#cmain P:nth-last-child(2) { font-weight: bold }
```

А этот стиль будет применен к предпоследнему (второму с конца) абзацу в контейнере `cmain`.

> ***НА ЗАМЕТКУ***
>
> Еще стандарт CSS описывает псевдоклассы `:nth-of-type` и `:nth-last-of-type`. Насколько удалось выяснить автору, они работают так же, как и псевдоклассы `:nth-child` и `:nth-last-child`. Хотя, возможно, в следующих реализациях они будут действовать по-другому.

Последний структурный псевдокласс, который мы изучим, — `:empty`. Он привязывает стиль к элементам, не имеющим дочерних элементов.

Пример:

```
P:empty { display: none }
```

Этот стиль будет привязан к пустым (не имеющим содержимого) абзацам.

Псевдоклассы *:not* и *

Особый псевдокласс `:not` позволяет привязать стиль к любому элементу Web-страницы, не удовлетворяющему заданным условиям. Таким условием может быть любой селектор:

<основной селектор>:not(*<селектор выбора>*) { *<стиль>* }

Стиль будет привязан к элементу Web-страницы, удовлетворяющему *основному селектору* и не удовлетворяющему *селектору выбора*.

Пример:

```
DIV:not(#cmain) { background-color: yellow }
```

Здесь мы указали в качестве основного селектора стиль переопределения тега `<DIV>`, а в качестве селектора выбора — именованный стиль `cmain`. В результате данный стиль будет применен ко всем блочным контейнерам, за исключением `cmain`.

А вот стиль, который будет применен ко всем строкам таблицы, за исключением первой:

```
TR:not(:nth-child(1)) { background-color: grey }
```

Псевдокласс `*` ("звездочка") обозначает любой элемент Web-страницы.

Пример:

```
#cmain > *:first-child { border-bottom: medium solid black }
```

Этот стиль будет применен к первому элементу любого типа, который непосредственно вложен в контейнер `cmain`.

Представление для нашего Web-сайта, часть 8

Да, специальные селекторы — мудреная штука... Разобраться в них без хорошей практики невозможно. Так давайте попрактикуемся.

Вот список параметров Web-страниц нашего Web-сайта, которые мы зададим с помощью специальных селекторов.

- Текст непосещенных и посещенных гиперссылок — не подчеркнут, цвет `#576C8C` (темно-синий).

- Текст посещенных гиперссылок, расположенных в полосе навигации, — не подчеркнут, цвет `#576C8C`. Мы задали для непосещенных и посещенных гиперссылок в полосе навигации одинаковые параметры — так принято.
- Текст посещенных гиперссылок, не расположенных в полосе навигации, — не подчеркнут, цвет `#A1AFBA` (синий).
- Текст активной гиперссылки — подчеркнут, цвет `#576C8C`.
- Текст гиперссылки, на которую наведен курсор мыши, — подчеркнут, цвет `#576C8C`.
- Текст гиперссылки, имеющей фокус ввода, — подчеркнут, цвет `#576C8C`.
- Шрифт первой буквы первого абзаца в контейнере `cmain` — 18 пунктов и полужирный.
- Выравнивание текста в ячейках первого и второго столбцов таблицы — списка версий HTML — по центру. Эти ячейки содержат исключительно числа, и центральное выравнивание для них подходит больше.

Чтобы воплотить задуманное, нам потребуется добавить в таблицу стилей main.css несколько новых стилей (листинг 13.9).

Листинг 13.9

```
A:link        { color: #576C8C;
                text-decoration: none }
A:visited     { color: #A1AFBA;
                text-decoration: none }
A:focus, A:hover,
  A:active    { color: #576C8C;
                text-decoration: underline }
```

Стили из листинга 13.9 задают параметры гиперссылок, расположенных вне полосы навигации. Здесь мы активно используем псевдоклассы гиперссылок.

Листинг 13.10

```
#navbar A:link,
  #navbar A:visited { color: #576C8C;
                      text-decoration: none }
#navbar A:focus,
  #navbar A:hover,
  #navbar A:active { color: #576C8C;
                     text-decoration: underline }
```

Стили из листинга 13.10 задают параметры гиперссылок полосы навигации. Отметим, что здесь применяются комбинированные стили, содержащие указание на список `navbar`, который формирует полосу навигации, — так проще и нагляднее.

Вот стиль, задающий параметры первой буквы первого абзаца в контейнере `cmain` (т. е. в основном содержимом Web-страницы):

```
#cmain > P:first-child:first-letter { font-size: 18pt;
                                     font-wight: bold }
```

Он представляет собой комбинированный стиль, содержащий целых три специальных селектора, и весьма сложен. Поэтому рассмотрим его по частям.

- `#cmain > P` — абзац должен быть непосредственно вложен в контейнер `cmain`.
- `#cmain > P:first-child` — помимо того, абзац должен быть первым дочерним элементом своего родителя (данного контейнера).
- `#cmain > P:first-child:first-letter` — ну и, собственно, указываем на первую букву данного абзаца. Именно к ней будет привязан этот стиль.

Обратим внимание — мы специально указали, что абзац должен быть непосредственно вложен в контейнер `cmain`. Если мы этого не сделаем, написав так:

```
#cmain P:first-child:first-letter { font-size: 18pt;
                                   font-weight: bold }
```

стиль будет применен и к абзацу, который вложен в тег большой цитаты — ведь этот абзац также будет первым дочерним элементом своего родителя. А нам это не нужно.

Вот два одинаковых стиля, задающих выравнивание текста по центру для первой и второй ячеек каждой строки таблицы:

```
.table-html-versions TD:first-child,
  .table-html-versions TD:nth-child(2) { text-align: center }
```

Как видим, они представляют собой комбинированный стиль, включающий стилевой класс `table-html-versions`.

Чтобы данные стили начали действовать на таблицу, нам придется привязать этот стилевой класс к ее тегу `<TABLE>`:

```
<TABLE CLASS="table-html-versions">
  <CAPTION>Список версий HTML:</CAPTION>
  . . .
</TABLE>
```

Вот и все. Добавим приведенные стили в таблицу стилей main.css, внесем исправления в Web-страницу index.htm, сохраним их и откроем Web-страницу index.htm в Web-обозревателе. Хороша, правда?

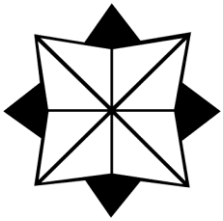
На этом разговор о представлении Web-страниц закончен.

Что дальше?

В этой, последней главе *части II* мы завершили рассмотрение CSS изучением специальных селекторов. И, разумеется, применили новые знания на практике.

Разговор о представлении Web-страниц и стилях CSS, с помощью которых оно создается, был долгим и продуктивным. Мы изучили множество атрибутов стиля, управляющих самыми разными параметрами элементов Web-страниц: шрифтом, цветом, фоном, выравниванием, отступами, рамками и др. А наш Web-сайт стал выглядеть намного лучше.

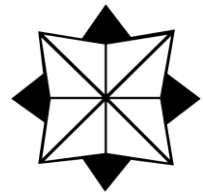
В *части III* мы будем рассматривать поведение Web-страниц. Мы познакомимся с принципами Web-программирования, языком программирования JavaScript и библиотекой Ext Core, сильно упрощающей работу программистов — нас с вами. Так что впереди еще много интересного!



ЧАСТЬ III

Поведение Web-страниц. Web-сценарии

ГЛАВА 14



Введение в Web-программирование. Язык JavaScript

Web-дизайн состоит из трех частей: содержимого, представления и поведения. Это мы узнали еще в *главе 1*. Содержимому была посвящена *часть I*, представлению — *часть II*. Теперь настала очередь обсудить поведение.

Поведение — это набор правил, определяющих, как Web-страница будет реагировать на действия посетителя. Так, мы можем указать, чтобы при наведении курсора мыши на гиперссылку в полосе навигации менялся цвет и стиль рамки, в которую она заключена. Еще мы можем менять содержимое Web-страницы после ее загрузки или даже дополнять его, заставлять контейнер расширяться при щелчке мышью, открывая свое содержимое, анимировать элемент Web-страницы и делать множество других интересных вещей с самой Web-страницей и любым ее фрагментом.

Поведение создается с помощью так называемых *Web-сценариев* — программ, которые записывают прямо в HTML-коде Web-страниц или, что предпочтительнее, в отдельных файлах. Эти программы пишут на языке *JavaScript*. Web-обозреватель считывает JavaScript-код и последовательно выполняет записанные в нем выражения, проводя вычисления и выполняя на основе полученного результата заданные манипуляции над Web-страницей.

Эта глава будет целиком посвящена принципам Web-программирования, языку JavaScript и средствам, предоставляемым Web-обозревателем для написания Web-сценариев. Она будет очень большой, так что приготовимся к долгому чтению и обилию новых терминов.

Примеры Web-сценариев

Язык JavaScript лучше всего изучать, имея перед глазами пару хороших примеров. Поэтому давайте сразу создадим их.

Простейший Web-сценарий

Первый Web-сценарий, который мы напишем, будет совсем простым. Он выведет на Web-страницу текущую дату.

В самом начале этой книги, приступив к изучению HTML, мы создали небольшую Web-страничку 1.1.htm. Найдем ее и откроем в Блокноте. В самом конце ее HTML-кода, перед закрывающим тегом </BODY>, вставим код листинга 14.1.

Листинг 14.1

```
<SCRIPT>
  var dNow = new Date();
  var sNow = dNow.toString();
  document.write(sNow);
</SCRIPT>
```

Это Web-сценарий. Мы поместили его прямо в HTML-код Web-страницы.

Сохраним исправленную Web-страницу и откроем ее в Web-обозревателе. В самом низу мы увидим новый абзац, содержащий сегодняшнюю дату в "международном" формате.

Наш первый Web-сценарий — поведение Web-страницы — работает!

Теперь немного исправим его так, чтобы он выводил дату в привычном для нас формате <*число*>.<*месяц*>.<*год*> (листинг 14.2).

Листинг 14.2

```
<SCRIPT>
  var dNow = new Date();
  var sNow = dNow.getDate() + "." + dNow.getMonth() + "." +
  dNow.getFullYear();
  document.write(sNow);
</SCRIPT>
```

Обновим открытую в Web-обозревателе Web-страницу, нажав клавишу <F5>. Вот теперь дата отображается в привычном формате.

Более сложный Web-сценарий

Теперь сделаем что-нибудь посложнее — заставим пункты списков, формирующих полосу навигации на Web-странице index.htm, менять цвет рамки при наведении на них курсора мыши. Так мы дадим посетителю понять, что данные элементы Web-страницы могут реагировать на его действия.

Сначала откроем таблицу стилей и внесем в ее CSS-код некоторые изменения. Прежде всего, исправим комбинированные стили #navbar LI и #navbar LI UL LI так, как показано в листинге 14.3.

Листинг 14.3

```
#navbar LI    { padding: 5px 10px;
                margin: 10px 0px;
```

```
                border: thin solid #B1BEC6;
                cursor: pointer }
. . .
#navbar LI UL LI { font-size: 12pt;
                   padding: 2px;
                   margin: 0px 0px;
                   border: thin solid #F8F8F8;
                   cursor: pointer }
```

Во-первых, мы задали для пунктов "внешнего" и вложенного списков, формирующих полосу навигации, форму курсора в виде "указующего перста". Так мы дадим посетителю понять, что эти пункты ведут себя как гиперссылки, и на них можно щелкать мышью.

Во-вторых, мы создали у пунктов вложенного списка тонкую сплошную рамку того же цвета, что и фон Web-страницы. Такая рамка будет невидима.

Далее добавим в таблицу стилей вот такой стиль:

```
.hovered      { border-color: #3B4043 !important }
```

Мы создали стилевой класс, задающий цвет рамки. Поскольку стилевой класс является менее конкретным, чем комбинированный стиль, мы сделали атрибут стиля, задающий цвет рамки, важным. (О важных атрибутах стиля см. *главу 7*.)

Если мы привяжем этот стиль к пункту любого списка, формирующего полосу навигации, — "внешнего" или вложенного, — он задаст для пункта новый цвет рамки. В результате рамка пункта "внешнего" списка станет более темной, а невидимая рамка пункта списка вложенного — видимой.

После этого отправимся по интернет-адресу **http://www.extjs.com/products/core/download.php?dl=extcore31** и загрузим оттуда архивный файл с именем вида ext-core-<*номер версии*>.zip — библиотеку Ext Core, упрощающую написание даже очень сложных Web-сценариев. (Подробнее речь о ней пойдет в *главе 15*.) Распакуем этот файл, найдем в распакованном содержимом файл ext-core.js и скопируем в корневую папку нашего Web-сайта, где хранится Web-страница index.htm и таблица стилей main.css. Теперь библиотека Ext Core готова к использованию.

> ***ВНИМАНИЕ!***
>
> Если по указанному интернет-адресу архивный файл библиотеки Ext Core загрузить не удается, следует открыть Web-страницу **http://www.extjs.com/products/core/download.php**, найти на ней гиперссылку загрузки этого файла (на данный момент она имеет текст "Download") и щелкнуть на ней.

Теперь откроем в Блокноте Web-страницу index.htm и поместим в ее секцию заголовка (тег <HEAD>) такой тег:

```
<SCRIPT SRC="ext-core.js"></SCRIPT>
```

А в самом конце HTML-кода этой Web-страницы, перед закрывающим тегом </BODY>, вставим такой тег:

```
<SCRIPT SRC="main.js"></SCRIPT>
```

Напоследок в корневой папке нашего Web-сайта создадим текстовый файл main.js и запишем в него содержимое листинга 14.4.

Листинг 14.4

```
Ext.onReady(function() {
  var ceLinks = Ext.select("UL[id=navbar] LI");

  ceLinks.on("mouseover", function(e, t) {
    Ext.get(this).addClass("hovered");
  });
  ceLinks.on("mouseout", function(e, t) {
    Ext.get(this).removeClass("hovered");
  });
});
```

Это тоже Web-сценарий. Но поместили его мы уже в отдельный файл main.js. Сохраним его, выбрав кодировку UTF-8 (как это сделать, было описано в *главе 1*).

Все, поведение создано. Откроем Web-страницу index.htm в Web-обозревателе. Наведем курсор мыши на любой пункт полосы навигации и увидим, как вокруг него появится темная рамка.

Как Web-сценарии помещаются в Web-страницу

Как мы только что убедились, Web-сценарии, формирующие поведение Web-страницы, можно поместить как в саму Web-страницу, так и в отдельный файл. Рассмотрим подробнее, как это делается.

Для вставки Web-сценария в HTML-код в любом случае применяется парный тег <SCRIPT>. Встретив его, Web-обозреватель поймет, что здесь присутствует Web-сценарий, который следует выполнить, а не выводить на экран.

Мы можем поместить код Web-сценария прямо в тег <SCRIPT>, создав *внутренний* Web-сценарий (листинг 14.5). Собственно, мы уже сделали это, когда создавали наш первый Web-сценарий.

Листинг 14.5

```
<SCRIPT>
  var dNow = new Date();
  var sNow = dNow.getDate() + "." + dNow.getMonth() + "." +
  dNow.getFullYear();
  document.write(sNow);
</SCRIPT>
```

Внутренние Web-сценарии имеют одно неоспоримое преимущество. Поскольку они записаны прямо в коде Web-страницы, то являются ее неотъемлемой частью, и их невозможно "потерять". Однако внутренние Web-сценарии не соответствуют концепции Web 2.0, требующей, чтобы содержимое, представление и поведение Web-страницы были разделены. Поэтому сейчас их применяют довольно редко, практически только при экспериментах (как и внутренние таблицы стилей; подробнее — в *главе 7*).

Мы можем поместить Web-сценарий и в отдельный файл — *файл Web-сценария*, — создав *внешний* Web-сценарий. (Наш второй Web-сценарий именно таков.) Файлы Web-сценария представляют собой обычные текстовые файлы, содержат только код Web-сценария без всяких тегов HTML и имеют расширение js.

Для вставки в Web-страницу Web-сценария, хранящегося в отдельном файле, применяется тег `<SCRIPT>` такого вида:

```
<SCRIPT SRC="<интернет-адрес файла Web-сценария>"></SCRIPT>
```

Тег `<SCRIPT>` оставляют пустым, и в него помещают обязательный в данном случае атрибут `SRC`, в качестве значения которого указывают интернет-адрес нужного файла Web-сценария:

```
<SCRIPT SRC="main.js"></SCRIPT>
```

Внешние Web-сценарии полностью укладываются в концепцию Web 2.0. Кроме того, такие Web-сценарии можно применять сразу на нескольких Web-страницах, задавая для них единообразное поведение. Так что в дальнейшем мы будем создавать поведение именно с помощью внешних Web-сценариев.

В нашем втором Web-сценарии мы использовали библиотеку Ext Core, значительно облегчающую труд Web-программиста. Во всех языках программирования *библиотекой* называется набор готовых языковых конструкций (функций и объектов, о которых речь пойдет потом), написанных сторонними программистами, чтобы облегчить труд их коллег. Так вот, все библиотеки для языка JavaScript, в том числе и Ext Core, реализованы в виде внешних Web-сценариев.

И еще. Web-сценарий всегда выполняется в том месте HTML-кода Web-страницы, где присутствует. При этом не имеет значения, помещен он прямо в HTML-код или находится в отдельном файле.

Из этого следует очень важный вывод. Если Web-сценарий выполняет какие-либо манипуляции над элементами Web-страницы, его нужно поместить после HTML-кода, формирующего эти элементы. Ведь перед тем, как манипулировать этими элементами, Web-обозреватель должен их создать. Что, впрочем, логично.

Язык программирования JavaScript

Настала пора рассмотреть язык программирования JavaScript. Ведь в Web-программировании без него никуда.

Основные понятия JavaScript

Давайте рассмотрим пример еще одного Web-сценария, совсем небольшого:

```
x = 4;
y = 5;
z = x * y;
```

Больше похоже на набор каких-то формул. Но это не формулы, а *выражения* языка JavaScript; каждое выражение представляет собой описание одного законченного действия, выполняемого Web-сценарием.

Разберем приведенный Web-сценарий по выражениям. Вот первое из них:

```
x = 4;
```

Здесь мы видим число `4`. В JavaScript такие числа, а также строки и прочие величины, значения которых никогда не изменяются, называются *константами*. В самом деле, значение числа 4 всегда равно четырем!

Еще мы видим здесь латинскую букву `x`. А она что означает?

О, это весьма примечательная вещь! Это *переменная*, которую можно описать как участок памяти компьютера, имеющий уникальное имя и предназначенный для хранения какой-либо величины — константы или результата вычисления. Наша переменная имеет имя `x`.

Осталось выяснить, что делает символ равенства (`=`), поставленный между переменной и константой. А он здесь стоит не просто так! (Вообще, в коде любой программы, в том числе и Web-сценария, каждый символ что-то да значит.) Это *оператор* — команда, выполняющая определенные действия над данными Web-сценария. А если точнее, то символом `=` обозначается *оператор присваивания*. Он помещает значение, расположенное справа *(операнд)*, в переменную, расположенную слева, в нашем случае — значение `4` в переменную `x`. Если же такой переменной еще нет, она будет создана.

Каждое выражение JavaScript должно оканчиваться символом точки с запятой (`;`), обозначающим конец выражения; его отсутствие вызывает ошибку обработки Web-сценария.

Рассмотрим следующее выражение:

```
y = 5;
```

Оно аналогично первому и присваивает переменной `y` константу `5`. Подобные выражения часто называют *математическими*.

Третье выражение стоит несколько особняком:

```
z = x * y;
```

Здесь мы видим все тот же оператор присваивания, присваивающий что-то переменной `z`. Но что? Результат вычисления произведения значений, хранящихся в переменных `x` и `y`. Вычисление произведения выполняет оператор умножения, который в JavaScript (и во многих других языках программирования) обозначается символом звездочки (`*`). Это *арифметический оператор*.

В результате выполнения приведенного ранее Web-сценария в переменной `z` окажется произведение значений 4 и 5 — 20.

Вот еще один пример математического выражения, на этот раз более сложного:

```
y = y1 * y2 + x1 * x2;
```

Оно вычисляется в следующем порядке:

1. Значение переменной `y1` умножается на значение переменной `y2`.
2. Перемножаются значения переменных `x1` и `x2`.
3. Полученные на шагах 1 и 2 произведения складываются (оператор сложения обозначается привычным нам знаком `+`).
4. Полученная сумма присваивается переменной `y`.

Но почему на шаге 2 выполняется умножение `x1` на `x2`, а не сложение произведения `y1` и `y2` с `x1`. Дело в том, что каждый оператор имеет *приоритет* — своего рода номер в очереди их выполнения. Так вот, оператор умножения имеет более высокий приоритет, чем оператор сложения, поэтому умножение всегда выполняется перед сложением.

А вот еще одно выражение:

```
x = x + 3;
```

Оно абсолютно правильно с точки зрения JavaScript, хоть и выглядит нелепым. В нем сначала выполняется сложение значения переменной `x` и числа `3`, после чего результат сложения снова присваивается переменной `x`. Такие выражения встречаются в Web-сценариях довольно часто.

Типы данных JavaScript

Любая программа при своей работе оперирует некими данными: именем стилевого класса, размерами элемента Web-страницы, цветом шрифта, величиной атмосферного давления и пр. Конечно, не составляют исключения и Web-сценарии — уже первый созданный нами Web-сценарий обрабатывал какие-то данные.

JavaScript может манипулировать данными, относящимися к разным *типам*. Тип данных описывает их возможные значения и набор применимых к ним операций. Давайте перечислим все типы данных, с которыми мы можем столкнуться.

Строковые данные (или *строки*) — это последовательности букв, цифр, пробелов, знаков препинания и прочих символов, заключенные в одинарные или двойные кавычки.

Примеры строк:

```
"JavaScript"
"1234567"
'Строковые данные — это последовательности символов.'
```

Строки могут иметь любую длину (определяемую количеством составляющих их символов), ограниченную лишь объемом свободной памяти компьютера. Теорети-

чески существует предел в 2 Гбайт, но вряд ли в нашей практике встретятся столь длинные строки.

Кроме букв, цифр и знаков препинания, строки могут содержать *специальные символы*, служащие для особых целей (табл. 14.1).

Таблица 14.1. *Специальные символы, поддерживаемые JavaScript, и их коды*

Символ	Описание
`\f`	Прогон листа
`\n`	Перевод строки
`\r`	Возврат каретки
`\t`	Табуляция
`\"`	Двойная кавычка
`\'`	Одинарная кавычка
`\\`	Обратный слэш
`\x<код>`	Любой символ по его *коду* в кодировке Unicode

Таким образом, если нам требуется поместить в строку двойные кавычки, нужно записать ее так:

```
"\"Этот фрагмент текста\" помещен в кавычки"
```

Числовые данные (или *числа*) — это обычные числа, над которыми можно производить арифметические действия, извлекать из них квадратный корень и вычислять тригонометрические функции. Числа могут быть как целыми, так и дробными; в последнем случае целая и дробная части разделяются точкой (не запятой!).

Примеры чисел:

```
13756
454.7873
0.5635
```

Дробные числа могут быть записаны в экспоненциальной форме:

<мантисса>`E`*<порядок>*.

Вот примеры чисел, заданных таким образом (в скобках дано традиционное математическое представление):

`1E-5` (10^{-5})

`8.546E23` ($8{,}546 \cdot 10^{23}$)

Также имеется возможность записи целых чисел в восьмеричном и шестнадцатеричном виде. Восьмеричные числа записываются с нулем в начале (например, `047` или `-012543624`), а шестнадцатеричные — с символами `0x`, также помещенными в начало (например, `0x35F`). Отметим, что в JavaScript так можно записывать только целые числа.

Логическая величина может принимать только два значения: true и false — "истина" и "ложь", — обозначаемые соответственно ключевыми словами `true` и `false`. (*Ключевое слово* — это слово, имеющее в языке программирования особое значение.) Логические величины используются, как правило, в условных выражениях (о них речь пойдет позже).

JavaScript также поддерживает три специальных типа. Тип *null* может принимать единственное значение `null` и применяется в особых случаях. Тип *NaN* также может принимать единственное значение `NaN` и обозначает числовое значение, не являющееся числом (например, математическую бесконечность). Тип *undefined* указывает на то, что переменной не было присвоено никакое значение, и, опять же, принимает единственное значение `undefined`.

> ***Внимание!***
> `undefined` — это не то же самое, что `null`!

Еще два типа данных, поддерживаемые JavaScript и не описанные здесь, мы рассмотрим позже.

Переменные

В начале этой главы мы кое-что узнали о переменных. Сейчас настало время обсудить их детальнее.

Именование переменных

Как мы уже знаем, каждая переменная должна иметь имя, которое однозначно ее идентифицирует. Об именах переменных стоит поговорить подробнее.

Прежде всего, в имени переменной могут присутствовать только латинские буквы, цифры и символы подчеркивания (_), причем первый символ имени должен быть либо буквой, либо символом подчеркивания. Например, `pageAddress`, `_link`, `userName` — правильные имена переменных, а `678vasya` и `Имя пользователя` — неправильные.

Язык JavaScript чувствителен к регистру символов, которыми набраны имена переменных. Это значит, что `pageaddress` и `pageAddress` — разные переменные.

Совпадение имени переменной с ключевым словом языка JavaScript не допускается.

Объявление переменных

Перед использованием переменной в коде Web-сценария рекомендуется выполнить ее *объявление*. Для этого служит *оператор объявления переменной* `var`, после которого указывают имя переменной:

```
var x;
```

Теперь объявленной переменной можно присвоить какое-либо значение:

```
x = 1234;
```

и использовать в Web-сценарии:

```
y = x * 2 + 10;
```

Значение переменной также можно присвоить прямо при ее объявлении:

```
var x = 1234;
```

Также можно объявить сразу несколько переменных:

```
var x, y, textColor = "black";
```

Вообще, объявлять переменные с помощью оператора `var` не обязательно. Мы можем просто присвоить переменной какое-либо значение, и JavaScript сам ее создаст. Просто явное объявление переменных оператором `var` считается хорошим стилем программирования.

Переменная, созданная в каком-либо Web-сценарии, будет доступна во всех остальных Web-сценариях, присутствующих в данной Web-странице. Об исключениях из этого правила мы поговорим потом.

> ***Внимание!***
> Если обратиться к еще не созданной переменной, она вернет значение `undefined`.

Пока закончим с переменными. (Впоследствии, при рассмотрении функций, мы к ним еще вернемся.) И займемся операторами JavaScript.

Операторы

Операторов язык JavaScript поддерживает очень много — на все случаи жизни. Их можно разделить на несколько групп.

Арифметические операторы

Арифметические операторы служат для выполнения арифметических действий над числами. Все арифметические операторы, поддерживаемые JavaScript, перечислены в табл. 14.2.

Таблица 14.2. *Арифметические операторы*

Оператор	Описание
-	Смена знака числа
+	Сложение
-	Вычитание
*	Умножение
/	Деление

Таблица 14.2 *(окончание)*

Оператор	Описание
%	Взятие остатка от деления
++	*Инкремент* (увеличение на единицу)
--	*Декремент* (уменьшение на единицу)

Арифметические операторы делятся на две группы: *унарные* и *бинарные*. Унарные операторы выполняются над одним операндом; к ним относятся операторы смены знака, инкремента и декремента. Унарный оператор извлекает из переменной значение, изменяет его и снова помещает в ту же переменную. Приведем пример выражения с унарным оператором:

```
++r;
```

При выполнении этого выражения в переменной `r` окажется ее значение, увеличенное на единицу. А если записать вот так:

```
s = ++r;
```

то значение `r`, увеличенное на единицу, будет помещено еще и в переменную `s`.

Операторы инкремента и декремента можно ставить как перед операндом, так и после него. Если оператор инкремента стоит перед операндом, то значение операнда сначала увеличивается на единицу, а уже потом используется в дальнейших вычислениях. Если же оператор инкремента стоит после операнда, то его значение сначала вычисляется, а уже потом увеличивается на единицу. Точно так же ведет себя оператор декремента.

Бинарные операторы всегда имеют два операнда и помещают результат в третью переменную. Вот примеры выражений с бинарными операторами:

```
l = r * 3.14;
f = e / 2;
x = x + t / 3;
```

Оператор объединения строк

Оператор объединения строк `+` позволяет соединить две строки в одну. Например, сценарий:

```
s1 = "Java";
s2 = "Script";
s = s1 + s2;
```

поместит в переменную `s` строку `JavaScript`.

Операторы присваивания

Оператор присваивания `=` нам уже знаком. Его еще называют оператором *простого присваивания*, поскольку он просто присваивает переменной новое значение:

```
a = 2;
b = c = 3;
```

Второе выражение в приведенном примере выполняет присвоение значения `3` сразу двум переменным — `b` и `c`.

JavaScript также поддерживает *операторы сложного присваивания*, позволяющие выполнять присваивание одновременно с другой операцией:

```
a = a + b;
a += b;
```

Два этих выражения эквивалентны по результату. Просто во втором указан оператор сложного присваивания `+=`.

Все операторы сложного присваивания, поддерживаемые JavaScript, и их эквиваленты приведены в табл. 14.3.

Таблица 14.3. *Операторы сложного присваивания*

Оператор	Эквивалентное выражение
`a += b;`	`a = a + b;`
`a -= b;`	`a = a - b;`
`a *= b;`	`a = a * b;`
`a /= b;`	`a = a / b;`
`a %= b;`	`a = a % b;`
`a <<= b;`	`a = a << b;`
`a >>= b;`	`a = a >> b;`
`a >>>= b;`	`a = a >>> b;`
`a &= b;`	`a = a & b;`
`a ^= b;`	`a = a ^ b;`
`a |= b;`	`a = a | b;`

Операторы сравнения

Операторы сравнения сравнивают два операнда согласно определенному условию и выдают (или, как говорят программисты, возвращают) логическое значение. Если условие сравнения выполняется, возвращается значение `true`, если не выполняется — `false`.

Все поддерживаемые JavaScript операторы сравнения приведены в табл. 14.4.

Пример:

```
a1 = 2 < 3;
a2 = -4 > 0;
a3 = r < t;
```

Переменная a1 получит значение true (2 меньше 3), переменная a2 — значение false (число –4 по определению не может быть больше нуля), а значение переменной a3 будет зависеть от значений переменных r и t.

Таблица 14.4. *Операторы сравнения*

Оператор	Описание
<	Меньше
>	Больше
==	Равно
<=	Меньше или равно
>=	Больше или равно
!=	Не равно
===	Строго равно
!==	Строго не равно

Можно сравнивать не только числа, но и строки:

```
a = "JavaScript" != "Java";
```

Переменная a получит значение true, т. к. строки "JavaScript" и "Java" не равны.

На двух последних операторах из табл. 14.4 — "строго равно" и "строго не равно" — нужно остановиться подробнее. Это операторы так называемого *строгого сравнения*. Обычные операторы "равно" и "не равно", если встречают операнды разных типов, пытаются преобразовать их к одному типу (о преобразованиях типов см. далее в этой главе). Операторы строгого равенства и строгого неравенства такого преобразования не делают, а в случае несовпадения типов операндов всегда возвращают false.

Пример:

```
a1 = 2 == "2";
a2 = 2 === "2";
```

Переменная a1 получит значение true, т. к. в процессе вычисления строка "2" будет преобразована в число 2, и условие сравнение выполнится. Но переменная a2 получит значение false, т. к. сравниваемые операнды принадлежат разным типам.

Логические операторы

Логические операторы выполняют действия над логическими значениями. Все они приведены в табл. 14.5. А в табл. 14.6 и 14.7 показаны результаты выполнения этих операторов.

Основная область применения логических операторов — выражения сравнения (о них см. далее в этой главе). Приведем примеры таких выражений:

```
a = (b > 0) && (c + 1 != d);
flag = !(status = 0);
```

Таблица 14.5. *Логические операторы*

Оператор	Описание
!	НЕ (логическая инверсия)
&&	И (логическое умножение)
||	ИЛИ (логическое сложение)

Таблица 14.6. *Результаты выполнения операторов И и ИЛИ*

Операнд 1	Операнд 2	&& (И)	|| (ИЛИ)
true	true	true	true
true	false	false	true
false	true	false	true
false	false	false	false

Таблица 14.7. *Результаты выполнения оператора НЕ*

Операнд	! (НЕ)
true	false
false	true

Оператор получения типа *typeof*

Оператор получения типа `typeof` возвращает строку, описывающую тип данных операнда. Операнд, тип которого нужно узнать, помещают после этого оператора и заключают в круглые скобки:

```
s = typeof("str");
```

В результате выполнения этого выражения в переменной `s` окажется строка `"string"`, обозначающая строковый тип.

Все значения, которые может вернуть оператор `typeof`, перечислены в табл. 14.8.

Таблица 14.8. *Значения, возвращаемые оператором* `typeof`

Тип данных	Возвращаемая строка
Строковый	"string"
Числовой	"number"

Таблица 14.8 *(окончание)*

Тип данных	Возвращаемая строка
Логический	`"boolean"`
Объектный (см. далее)	`"object"`
Функциональный (см. далее)	`"function"`
null	`"null"`
undefined	`"undefined"`

Совместимость и преобразование типов данных

Настала пора рассмотреть еще два важных вопроса: *совместимость* типов данных и *преобразование* одного типа к другому.

Что получится, если сложить два числовых значения? Правильно — еще одно числовое значение. А если сложить число и строку? Трудно сказать... Тут JavaScript сталкивается с проблемой несовместимости типов данных и пытается сделать эти типы совместимыми, преобразуя один из них к другому. Сначала он пытается преобразовать строку в число и, если это удается, выполняет сложение. В случае неудачи число будет преобразовано в строку, и две полученные строки будут объединены. Например, в результате выполнения Web-сценария из листинга 14.6 значение переменной `b` при сложении с переменной `a` будет преобразовано в числовой тип; таким образом, переменная `c` будет содержать значение 23.

Листинг 14.6

```
var a, b, c, d, e, f;
a = 11;
b = "12";
c = a + b;
d = "JavaScript";
e = 2;
f = d + e;
```

Но поскольку значение переменной `d` нельзя преобразовать в число, значение `e` будет преобразовано в строку, и результат — значение `f` — станет равным "JavaScript2".

Логические величины преобразуются либо в числовые, либо в строковые, в зависимости от конкретного случая. Значение `true` будет преобразовано в число 1 или строку `"1"`, а значение `false` — в 0 или `"0"`. И наоборот, число 1 будет преобразовано в значение `true`, а число 0 — в значение `false`. Также в `false` будут преобразованы значения `null` и `undefined`.

Видно, что JavaScript изо всех сил пытается правильно выполнить даже некорректно написанные выражения. Иногда это получается, но чаще все работает не так, как планировалось, и, в конце концов, выполнение Web-сценария прерывается в связи с обнаружением ошибки совсем в другом его месте, на абсолютно верном операторе. Поэтому лучше не допускать подобных казусов.

Приоритет операторов

Последний вопрос, который мы здесь разберем, — приоритет операторов. Как мы помним, приоритет влияет на порядок, в котором выполняются операторы в выражении.

Пусть имеется следующее выражение:

```
a = b + c – 10;
```

В этом случае сначала к значению переменной `b` будет прибавлено значение `c`, а потом из суммы будет вычтено 10. Операторы этого выражения имеют одинаковый приоритет и поэтому выполняются строго слева направо.

Теперь рассмотрим такое выражение:

```
a = b + c * 10;
```

Здесь сначала будет выполнено умножение значения `c` на 10, а уже потом к полученному произведению будет прибавлено значение `b`. Оператор умножения имеет больший приоритет, чем оператор сложения, поэтому порядок "строго слева направо" будет нарушен.

Самый низкий приоритет у операторов присваивания. Вот почему сначала вычисляется само выражение, а потом его результат присваивается переменной.

В общем, основной принцип выполнения всех операторов таков: сначала выполняются операторы с более высоким приоритетом, а уже потом — операторы с более низким. Операторы с одинаковым приоритетом выполняются в порядке их следования (слева направо).

В табл. 14.9 перечислены все изученные нами операторы в порядке убывания их приоритетов.

Таблица 14.9. *Приоритет операторов (в порядке убывания)*

Операторы	Описание
`++ -- – ~ ! typeof`	Инкремент, декремент, смена знака, логическое НЕ, определение типа
`* / %`	Умножение, деление, взятие остатка
`+ -`	Сложение и объединение строк, вычитание
`< > <= >=`	Операторы сравнения
`== != === !==`	
`&&`	Логическое И

Таблица 14.9 *(окончание)*

Операторы	Описание
`||`	Логическое ИЛИ
`?`	Условный оператор (см. ниже)
`= <оператор>=`	Присваивание, простое и сложное

> ***Внимание!***
> Запомните эту таблицу. Неправильный порядок выполнения операторов может стать причиной трудно выявляемых ошибок, при которых внешне абсолютно правильное выражение дает неверный результат.

Но что делать, если нам нужно нарушить обычный порядок выполнения операторов? Воспользуемся скобками. При такой записи заключенные в скобки операторы выполняются первыми:

```
a = (b + c) * 10;
```

Здесь сначала будет выполнено сложение значений переменных `b` и `c`, а потом получившаяся сумма будет умножена на 10.

Операторы, заключенные в скобки, также подчиняются приоритету. Поэтому часто используются многократно вложенные скобки:

```
a = ((b + c) * 10 — d) / 2 + 9;
```

Здесь операторы будут выполнены в такой последовательности:

1. Сложение `b` и `c`.
2. Умножение полученной суммы на 10.
3. Вычитание `d` из произведения.
4. Деление разности на 2.
5. Прибавление 9 к частному.

Если удалить скобки:

```
a = b + c * 10 — d / 2 + 9;
```

то порядок выполнения операторов будет таким:

1. Умножение `c` и 10.
2. Деление `d` на 2.
3. Сложение `b` и произведения `c` и 10.
4. Вычитание из полученной суммы частного от деления `d` на 2.
5. Прибавление 9 к полученной разности.

Получается совсем другой результат, не так ли?

Сложные выражения JavaScript

Сложные выражения получили свое название благодаря тому, что все они составлены из нескольких простых выражений. Сложные выражения выполняются специальным образом и служат для особых целей — в основном, для управления процессом выполнения содержащихся в них простых выражений.

Блоки

JavaScript позволяет нам объединить несколько выражений в одно. Такое выражение называется *блочным выражением* или просто *блоком*. Составляющие его выражения заключают в фигурные скобки, например:

```
{
  b = "12";
  c = a – b;
}
```

Как правило, блоки не существуют сами по себе. Чаще всего они входят в состав других сложных выражений.

Условные выражения

Условное выражение позволяет нам выполнить одно из двух входящих в него выражений в зависимости от выполнения или невыполнения какого-либо *условия*. В качестве условия используется значение логической переменной или результат вычисления логического выражения.

Листинг 14.7 иллюстрирует формат условного выражения.

Листинг 14.7

```
if (<условие>)
  <блок "то">
else
  <блок "иначе">
```

Существует также другая, "вырожденная" разновидность условного выражения, содержащая только одно выражение, которое выполняется при выполнении условия и пропускается, если условие не выполнено:

```
if (<условие>)
  <блок "то">
```

Для написания условных выражений предусмотрены особые ключевые слова `if` и `else`. Отметим, что *условие* всегда записывают в круглых скобках.

Если *условие* имеет значение `true`, то выполняется *блок "то"*. Если же *условие* имеет значение `false`, то выполняется *блок "иначе"* (если он присутствует в условном вы-

ражении). А если *блок "иначе"* отсутствует, выполняется следующее выражение Web-сценария.

> ***ВНИМАНИЕ!***
> Значения `null` или `undefined` преобразуются в `false`. Не забываем об этом.

Рассмотрим несколько примеров.

В листинге 14.8 мы сравниваем значение переменной `x` с единицей и в зависимости от результатов сравнения присваиваем переменным `f` и `h` разные значения.

Листинг 14.8

```
if (x == 1) {
  a = "Единица";
  b = 1;
}
 else {
  a = "Не единица";
  b = 22222;
}
```

Условие может быть довольно сложным (листинг 14.9).

Листинг 14.9

```
if ((x == 1) && (y > 10))
  f = 3;
else
  f = 33;
```

Здесь мы использовали сложное условие, возвращающее значение `true` в случае, если значение переменной `x` равно 1 и значение переменной `y` больше 10. Заметим также, что мы подставили одиночные выражения, т. к. фрагменты кода слишком просты, чтобы оформлять их в виде блоков.

Условный оператор ?

Если условное выражение совсем простое, мы можем записать его немного по-другому. А именно, воспользоваться *условным оператором* `?`:

```
<условие> ? <выражение "то"> : <выражение "иначе">;
```

Достоинство этого оператора в том, что он может быть частью выражения. Например:

```
f = (x == 1 && y > 10) ? 3 : 33;
```

Фактически мы записали условное выражение из предыдущего примера, но в виде простого выражения. Компактность кода налицо. Недостаток же оператора `?` в том, что с его помощью можно записывать только самые простые условные выражения.

Приоритет условного оператора один из самых низких. Приоритет ниже него имеют только операторы присваивания.

Выражения выбора

Выражение выбора — это фактически несколько условных выражений, объединенных в одном. Его формат иллюстрирует листинг 14.10.

Листинг 14.10

```
switch (<исходное выражение>) {
  case <значение 1> :
    <блок 1>
    [break;]
  [case <значение 2> :
    <блок 2>
    [break;]]
  <... другие секции case>
  [default :
    <блок, исполняемый для остальных значений>]
}
```

В выражениях выбора присутствуют ключевые слова `switch`, `case` и `default`.

Результат вычисления *исходного выражения* последовательно сравнивается со *значением 1*, *значением 2* и т. д. и, если такое сравнение прошло успешно, выполняется соответствующий блок кода (*блок 1*, *блок 2* и т. д.). Если же ни одно сравнение не увенчалось успехом, выполняется блок кода, находящийся в секции `default` (если, конечно, она присутствует).

Листинг 14.11 иллюстрирует пример выражения выбора.

Листинг 14.11

```
switch (a) {
  case 1 :
    out = "Единица";
    break;
  case 2 :
    out = "Двойка";
    break;
  case 3 :
    out = "Тройка";
    break;
  default :
    out = "Другое число";
}
```

Здесь, если переменная `a` содержит значение 1, переменная `out` получит значение `"Единица"`, если 2 — значение `"Двойка"`, а если 3 — значение `"Тройка"`. Если же переменная `a` содержит какое-то другое значение, переменная `out` получит значение `"Другое число"`.

Циклы

Циклы — это особые выражения, позволяющие выполнить один и тот же блок кода несколько раз. Выполнение кода прерывается по наступлению некоего условия.

JavaScript предлагает программистам несколько разновидностей циклов. Рассмотрим их подробнее.

Цикл со счетчиком

Цикл со счетчиком удобен, если какой-то код нужно выполнить строго определенное число раз. Вероятно, это наиболее распространенный вид цикла.

Цикл со счетчиком записывается так:

```
for (<выражение инициализации>; <условие>; <приращение>)
  <тело цикла>
```

Здесь используется ключевое слово `for`. Поэтому такие циклы часто называют "циклами for".

Выражение инициализации выполняется самым первым и всего один раз. Оно присваивает особой переменной, называемой *счетчиком цикла*, некое начальное значение (обычно 1). Счетчик цикла подсчитывает, сколько раз было выполнено *тело цикла* — собственно код, который нужно выполнить определенное количество раз.

Следующий шаг — проверка *условия*. Оно определяет момент, когда выполнение цикла прервется и начнет выполняться следующий за ним код. Как правило, *условие* сравнивает значение счетчика цикла с его граничным значением. Если *условие* возвращает `true`, выполняется *тело цикла*, в противном случае цикл завершается и начинается выполнение кода, следующего за циклом.

После прохода *тела цикла* выполняется выражение *приращения*, изменяющее значение счетчика. Это выражение обычно инкрементирует счетчик (увеличивает его значение на единицу). Далее снова проверяется *условие*, выполняется *тело цикла*, *приращение* и т. д., пока *условие* не станет равно `false`.

Пример цикла со счетчиком:

```
for (i = 1; i < 11; i++) {
  a += 3;
  b = i * 2 + 1;
}
```

Этот цикл будет выполнен 10 раз. Мы присваиваем счетчику `i` начальное значение 1 и после каждого выполнения тела цикла увеличиваем его на единицу. Цикл перестанет выполняться, когда значение счетчика увеличится до 11, и условие цикла станет ложным.

Счетчик цикла можно записать в одном из выражений тела цикла, как это сделали мы. В нашем случае счетчик `i` будет содержать последовательно возрастающие значения от 1 до 10, которые используются в вычислениях.

Приведем еще два примера цикла со счетчиком:

```
for (i = 10; i > 0; i--) {
  a += 3;
  b = i * 2 + 1;
}
```

Здесь значение счетчика декрементируется. Начальное его значение равно 10. Цикл выполнится 10 раз и завершится, когда счетчик `i` будет содержать 0; при этом значения последнего будут последовательно уменьшаться от 10 до 1.

```
for (i = 2; i < 21; i += 2) b = i * 2 + 1;
```

А в этом примере начальное значение счетчика равно 2, а конечное — 21, но цикл выполнится, опять же, 10 раз. А все потому, что значение счетчика увеличивается на 2 и последовательно принимает значения 2, 4, 6... 20.

Цикл с постусловием

Цикл с постусловием во многом похож на цикл со счетчиком: он выполняется до тех пор, пока остается истинным условие цикла. Причем условие проверяется не до, а после выполнения тела цикла, отчего цикл с постусловием и получил свое название. Такой цикл выполнится хотя бы один раз, даже если его условие с самого начала ложно.

Формат цикла с постусловием:

```
do
  <тело цикла>
while (<условие>);
```

Для задания цикла с постусловием предусмотрены ключевые слова `do` и `while`, поэтому такие циклы часто называют "циклами do-while".

Вот пример цикла с постусловием:

```
do {
  a = a * i + 2;
  ++i;
} while (a < 100);
```

А вот еще один пример:

```
var a = 0, i = 1;
do {
  a = a * i + 2;
  ++i;
} while (i < 20);
```

Хотя здесь удобнее был бы уже знакомый нам и специально предназначенный для таких случаев цикл со счетчиком.

Цикл с предусловием

Цикл с предусловием отличается от цикла с постусловием тем, что условие проверяется перед выполнением тела цикла. Так что, если оно (условие) изначально ложно, цикл не выполнится ни разу:

```
while (<условие>)
  <тело цикла>
```

Для создания цикла с постусловием предусмотрено ключевое слово `while`. Поэтому такие циклы называют еще "циклами while" (не путать с "циклами do-while"!).

Пример цикла с предусловием:

```
while (a < 100) {
  a = a * i + 2;
  ++i;
}
```

Прерывание и перезапуск цикла

Иногда бывает нужно прервать выполнение цикла. Для этого JavaScript предоставляет Web-программистам операторы `break` и `continue`.

Оператор прерывания `break` позволяет *прервать* выполнение цикла и перейти к следующему за ним выражению:

```
while (a < 100) {
  a = a * i + 2;
  if (a > 50) break;
  ++i;
}
```

В этом примере мы прерываем выполнение цикла, если значение переменной `a` превысит 50.

Оператор перезапуска `continue` позволяет *перезапустить* цикл, т. е. оставить невыполненными все последующие выражения, входящие в тело цикла, и запустить выполнение цикла с самого его начала: проверка условия, выполнение приращения и тела и т. д.

Пример:

```
while (a < 100) {
  i = ++i;
  if (i > 9 && i < 11) continue;
  a = a * i + 2;
}
```

Здесь мы пропускаем выражение, вычисляющее `a`, для всех значений `i` от 10 до 20.

Функции

Функция — это особым образом написанный и оформленный фрагмент кода JavaScript, который можно вызвать из любого Web-сценария на данной Web-странице (повторно используемый код, как его часто называют). Так что, если какой-то фрагмент кода встречается в нескольких местах нашего Web-сценария, лучше всего оформить его в виде функции.

Собственно код, ради которого и была создана функция, называется *телом функции*. Каждая функция, кроме того, должна иметь уникальное имя, по которому к ней можно будет обратиться. Функция также может принимать один или несколько параметров и возвращать результат, который можно использовать в выражениях.

Объявление функций

Прежде чем функция будет использована где-то в Web-сценарии, ее нужно объявить. Функцию объявляют с помощью ключевого слова `function`:

```
function <имя функции>([<список параметров, разделенных запятыми>])
  <тело функции>
```

Имя функции, как уже говорилось, должно быть уникально в пределах Web-страницы. Для имен функций действуют те же правила, что и для имен переменных.

Список параметров представляет собой набор переменных, в которые при вызове функции будут помещены значения переданных ей параметров. (О вызове функций будет рассказано далее.) Мы можем придумать для этих переменных любые имена — все равно они будут использованы только внутри тела функции. Это так называемые *формальные параметры* функции.

Список параметров функции помещают в круглые скобки и ставят после ее имени; сами параметры отделяют друг от друга запятыми. Если функция не требует параметров, следует указать пустые скобки — это обязательно.

В пределах *тела функции* над принятыми ею параметрами (если они есть) и другими данными выполняются некоторые действия и, возможно, вырабатывается результат. *Оператор возврата* `return` возвращает результат из функции в выражение, из которого она была вызвана:

```
return <переменная или выражение>;
```

Здесь *переменная* должна содержать возвращаемое значение, а *выражение* должно его вычислять.

Пример объявления функции:

```
function divide(a, b) {
  var c;
  c = a / b;
  return c;
}
```

Данная функция принимает два параметра — `a` и `b`, — после чего делит `a` на `b` и возвращает частное от этого деления.

Эту функцию можно записать компактнее:

```
function divide(a, b) {
  return a / b;
}
```

Или даже так, в одну строку:

```
function divide(a, b) { return a / b; }
```

JavaScript позволяет нам создавать так называемые *необязательные параметры* функций — параметры, которые при вызове можно не указывать, после чего они примут некоторое значение по умолчанию. Вот пример функции с необязательным параметром `b`, имеющим значение по умолчанию 2:

```
function divide(a, b) {
  if (typeof(b) == "undefined") b = 2;
  return a / b;
}
```

Понятно, что мы должны как-то выяснить, был ли при вызове функции указан параметр `b`. Для этого мы используем оператор получения типа `typeof` (он был описан ранее). Если параметр `b` не был указан, данный оператор вернет строку `"undefined"`; тогда мы создадим переменную с именем `b`, как и у необязательного параметра, и присвоим ей число 2 — значение этого параметра по умолчанию, — которое и будет использовано в теле функции. Если возвращенное оператором значение иное, значит, параметр `b` был указан при вызове, и мы используем значение, которое было передано с ним.

Функции и переменные. Локальные переменные

Объявленные ранее функции создают внутри своего тела собственные переменные. Это так называемые *локальные* переменные. Такие переменные доступны только внутри тела функции, в котором они объявлены. При завершении выполнения функции все объявленные в ней локальные переменные уничтожаются.

Разумеется, любая функция может обращаться к любой переменной, объявленной вне ее тела (переменной *уровня Web-страницы*). При этом нужно помнить об одной вещи. Если существуют две переменные с одинаковыми именами, одна — уровня Web-страницы, другая — локальная, то при обращении по этому имени будет получен доступ к локальной переменной. Одноименная переменная уровня Web-страницы будет "замаскирована" своей локальной "тезкой".

Вызов функций

После объявления функции ее можно *вызвать* из любого Web-сценария, присутствующего на этой же Web-странице. Формат вызова функции:

<имя функции>([*<список фактических параметров, разделенных запятыми>*])

Здесь указывается имя нужной функции и в круглых скобках перечисляются *фактические* параметры, над которыми нужно выполнить соответствующие действия. Функция вернет результат, который можно присвоить переменной или использовать в выражении.

> ***Внимание!***
>
> При вызове функции подставляйте именно фактические параметры, а не формальные, указанные в объявлении функции.

Вот пример вызова объявленной нами ранее функции `divide`:

```
d = divide(3, 2);
```

Здесь мы подставили в выражение вызова функции фактические параметры — константы 3 и 2.

А здесь мы выполняем вызов функции с переменными в качестве фактических параметров:

```
s = 4 * divide(x, r) + y;
```

Если функция имеет необязательные параметры и нас удовлетворяют их значения по умолчанию, мы можем при вызове не указывать эти параметры, все или некоторые из них. Например, функцию `divide` со вторым необязательным параметром мы можем вызвать так:

```
s = divide(4);
```

Тогда в переменной `s` окажется число 2 — результат деления 4 (значение первого параметра) на 2 (значение второго, необязательного, параметра по умолчанию).

Если функция не возвращает результат, то ее вызывают так:

```
initVars(1, 2, 3, 6);
```

Более того, так можно вызвать и функцию, возвращающую результат, который в этом случае будет отброшен. Такой способ вызова может быть полезен, если результат, возвращаемый функцией, не нужен для работы Web-сценария.

Если функция не принимает параметров, при ее вызове все равно нужно указать пустые скобки, иначе возникнет ошибка выполнения Web-сценария:

```
s = computeValue();
```

Функции могут вызывать друг друга. Вот пример:

```
function cmp(c, d, e) {
  var f;
  f = divide(c, d) + e;
  return f;
}
```

Здесь мы использовали в функции `cmp` вызов объявленной ранее функции `divide`.

Присваивание функций. Функциональный тип данных

JavaScript позволяет выполнять над функциями один фокус — присваивать функции переменным.

Пример:

```
var someFunc;
someFunc = cmp;
```

Здесь мы присвоили переменной `someFunc` объявленную ранее функцию `cmp`. Заметим, что в этом случае справа от оператора присваивания указывают только имя функции без скобок и параметров.

Впоследствии мы можем вызывать данную функцию, обратившись к переменной, которой она была присвоена:

```
c = someFunc(1, 2, 3);
```

Здесь мы вызвали функцию `cmp` через переменную `someFunc`.

Переменная, которой была присвоена функция, хранит данные, относящиеся к *функциональному типу*. Это еще один тип данных, поддерживаемый JavaScript.

А можно сделать и так:

```
var someFunc = function(c, d, e) {
  var f;
  f = divide(c, d) + e;
  return f;
}
```

Здесь мы объявили функцию и сразу же присвоили ее переменной. Как видим, имени объявляемой функции мы не указали — в данном случае оно не нужно.

А еще JavaScript позволяет нам указать функцию в качестве параметра другой функции. Для примера давайте рассмотрим фрагмент JavaScript-кода нашего второго Web-сценария:

```
ceLinks.on("mouseover", function(e, t) {
  Ext.get(t).addClass("hovered");
});
```

Здесь второй параметр функции `on` (вообще-то, это не функция, а метод объекта, но об этом потом) — другая функция, объявленная там же. Эта функция принимает два параметра и содержит одно выражение.

В следующих главах, изучая библиотеку Ext Core, мы будем часто сталкиваться с функциями, которые присваиваются переменным и передаются в качестве параметра другим функциям. Настолько часто, что скоро привыкнем к этому.

Массивы

Массив — это пронумерованный набор переменных *(элементов)*, фактически хранящийся в одной переменной. Доступ к отдельному элементу массива выполняется

по его порядковому номеру, называемому *индексом*. А общее число элементов массива называется его *размером*.

> ***Внимание!***
> Нумерация элементов массива начинается с нуля.

Чтобы создать массив, нужно просто присвоить любой переменной список его элементов, разделенных запятыми и заключенных в квадратные скобки:

```
var someArray = [1, 2, 3, 4];
```

Здесь мы создали массив, содержащий четыре элемента, и присвоили его переменной `someArray`. После этого мы можем получить доступ к любому из элементов по его индексу, указав его после имени переменной массива в квадратных скобках:

```
a = massive[2];
```

В данном примере мы получили доступ к третьему элементу массива. (Нумерация элементов массива начинается с нуля — помните об этом!)

Определять сразу все элементы массива необязательно:

```
someArray2 = [1, 2, , 4];
```

Здесь мы пропустили третий элемент массива, и он остался неопределенным (т. е. будет содержать значение `undefined`).

При необходимости мы легко сможем добавить к массиву еще один элемент, просто присвоив ему требуемое значение:

```
someArray[4] = 9;
```

При этом будет создан новый, пятый по счету, элемент массива с индексом 4 и значением 9.

Можно даже сделать так:

```
someArray[7] = 9;
```

В этом случае будут созданы четыре новых элемента, и восьмой элемент получит значение 9. Пятый, шестой и седьмой останутся неопределенными (`undefined`).

Мы можем присвоить любому элементу массива другой массив (или, как говорят опытные программисты, создать *вложенный* массив):

```
someArray[2] = ["n1", "n2", "n3"];
```

После этого можно получить доступ к любому элементу вложенного массива, указав последовательно оба индекса: сначала — индекс во "внешнем" массиве, потом — индекс во вложенном:

```
str = someArray[2][1];
```

Переменная `str` получит в качестве значения строку, содержащуюся во втором элементе вложенного массива, — `n2`.

Ранее говорилось, что доступ к элементам массива выполняется по числовому индексу. Но JavaScript позволяет создавать и массивы, элементы которых имеют строковые индексы (*ассоциативные массивы*, или *хэши*).

Пример:

```
var hash;
hash["AUDIO"] = "t_audio.htm";
hash["IMG"] = "t_img.htm";
hash["TITLE"] = "t_title.htm";
```

JavaScript также позволяет нам создать массив, вообще не содержащий элементов (пустой массив). Для этого достаточно присвоить любой переменной "пустые" квадратные скобки:

```
var someArray;
someArray = [];
```

Разумеется, впоследствии мы можем и даже должны наполнить этот массив элементами:

```
someArray[0] = 1;
someArray[1] = 2;
someArray[2] = 3;
```

Массивы идеально подходят в тех случаях, когда нужно хранить в одном месте упорядоченный набор данных. Ведь массив фактически представляет собой одну переменную.

Ссылки

Осталось рассмотреть еще один момент, связанный с организацией доступа к данным. Это так называемые *ссылки* — своего рода указатели на массивы и экземпляры объектов (о них будет рассказано далее), хранящиеся в соответствующих им переменных.

Когда мы создаем массив, JavaScript выделяет под него область памяти и помещает в нее значения элементов этого массива. Но в переменную, которой мы присвоили вновь созданный массив, помещается не сама эта область памяти, а ссылка на нее. Если теперь обратиться к какому-либо элементу этого массива, JavaScript извлечет из переменной ссылку, по ней найдет нужную область памяти, вычислит местонахождение нужного элемента и вернет его значение.

Далее, если мы присвоим значение переменной массива другой переменной, будет выполнено присваивание именно ссылки. В результате получатся две переменные, ссылающиеся на одну область памяти, хранящую сам этот массив.

Рассмотрим такой пример:

```
var myArray = ["AUDIO", "IMG", "TITLE"];
var newArray = myArray;
```

Здесь создается массив `myArray` с тремя элементами и далее он присваивается переменной `newArray` (при этом данная переменная получает ссылку на массив). Если потом мы присвоим новое значение первому элементу массива `myArray`:

```
myArray[0] = "VIDEO";
```

и обратимся к нему через переменную `newArray`:

```
s = newArray[0];
```

то в переменной `s` окажется строка `"VIDEO"` — новое значение первого элемента этого массива. Фактически переменные `myArray` и `newArray` указывают на один и тот же массив.

> ***Внимание!***
>
> В дальнейшем для простоты мы будем считать, что в переменной хранится не ссылка на массив (экземпляр объекта), а сам массив (экземпляр объекта). Если нужно будет специально указать, что переменная хранит ссылку, мы так и укажем.

Переменная, хранящая ссылку на массив (и экземпляр объекта), содержит данные *объектного типа*. Это последний тип данных, поддерживаемый JavaScript, который мы здесь рассмотрим.

> ***На заметку***
>
> Ранее мы узнали, что можем присвоить функцию переменной. Так вот, фактически переменная, которой была присвоена функция, также хранит ссылку на нее.

Объекты

Итак, мы познакомились с типами данных, переменными, константами, операторами, простыми и сложными выражениями, функциями и массивами. Но это была, так сказать, присказка, а сказка будет впереди. Настала пора узнать о самых сложных структурах данных JavaScript — объектах.

Понятия объекта и экземпляра объекта

В начале этой главы мы познакомились с типами данных, определяющими саму природу данных и набор действий, которые можно выполнять с этими данными. Так, строковый тип определяет, что данные этого типа представляют собой строки — наборы символов, которые можно объединять и сравнивать с другими строками.

Это были так называемые *простые типы данных*. Сущность, относящаяся к такому типу, может хранить только одно значение.

Однако JavaScript предоставляет нам и *сложные типы данных*. Сущность такого типа может хранить сразу несколько значений. Один из примеров сложного типа данных — уже знакомые нам массивы.

Другой пример сложного типа данных — объекты. *Объект* — это сложная сущность, способная хранить сразу несколько значений разных типов. Для этого объект определяет набор своего рода внутренних переменных, называемых *свойствами*; такое свойство может хранить одну сущность, относящуюся к простому или сложному типу данных. Так, одно свойство объекта может хранить строки, другое — числа, третье — массивы.

А еще объект может содержать набор внутренних функций, называемых *методами*. Методы могут обрабатывать данные, хранящиеся в свойствах или полученные "извне", менять значения свойств и возвращать результат, как обычные функции.

Объект — это всего лишь тип данных. Сущности же, хранящие реальные данные и созданные на основе этого объекта, называются его *экземплярами*. Точно так же, как строка `"JavaScript"` — экземпляр строкового типа данных, хранящий реальную строку.

Экземпляры объектов имеют такую же природу, как и массивы. Сам экземпляр объекта находится где-то в памяти компьютера, а в переменной хранится ссылка на него. При присваивании ее значения другой переменной выполняется присваивание именно ссылки. Не забываем об этом.

Каждый объект должен иметь уникальное имя, по которому к нему можно обратиться, чтобы создать его экземпляр. К именам объектов предъявляют те же требования, что и к именам переменных и функций.

Объекты — невероятно мощное средство объединить данные (свойства) и средства их обработки (методы) воедино. Так, объект, представляющий абзац Web-страницы, объединяет параметры этого абзаца, хранящиеся в различных свойствах, и инструменты для манипуляции абзацем, предоставляемые соответствующими методами. Нам не придется "раскидывать" параметры абзаца по десяткам переменных и пользоваться для работы с ним массой отдельных функций — все это сведено в один объект.

Экземпляры одного объекта — отдельные сущности, не влияющие друг на друга. Мы можем работать с одним экземпляром объекта, а другие экземпляры того же самого объекта останутся неизменными. Так, мы можем изменить параметры одного абзаца, присвоив новые значения свойствам соответствующего экземпляра объекта, не затрагивая другие абзацы на этой же Web-странице.

Все объекты, доступные в Web-сценариях, можно разделить на три разновидности:

- предоставляемые самим языком JavaScript *(встроенные объекты)*;
- предоставляемые Web-обозревателем *(объекты Web-обозревателя)*;
- созданные нами или сторонним разработчиком на самом JavaScript *(пользовательские объекты)*. В частности, популярные JavaScript-библиотеки, такие как Ext Core, создают множество пользовательских объектов.

С точки зрения Web-программиста все эти объекты одинаковы. И работа ними и их экземплярами выполняется сходным образом.

Получение экземпляра объекта

Но как нам получить экземпляр нужного объекта?

- Экземпляры многих объектов создаются самим языком JavaScript, Web-обозревателем или библиотеками сторонних разработчиков.
- Экземпляры некоторых объектов возвращаются функциями или методами других объектов.

- Мы можем создать экземпляр объекта сами, написав соответствующее выражение.

Давайте рассмотрим все эти случаи подробно.

Прежде всего, сам язык JavaScript предоставляет нам несколько экземпляров различных объектов. Они хранятся в особых переменных, также создаваемых самим языком.

> ***Внимание!***
>
> Вообще-то, в переменных хранятся ссылки на экземпляры объектов. Просто раньше мы договорились, что будем считать, будто в переменных хранятся сами экземпляры объектов, — так проще.

Так, переменная `Math` хранит экземпляр одноименного объекта, поддерживающего множество методов для выполнения математических и тригонометрических вычислений над числами:

```
var a = Math.sqrt(2);
```

Это выражение поместит в переменную `a` квадратный корень из 2. Метод `sqrt` объекта `Math` как раз вычисляет квадратный корень из числа, переданного ему в качестве единственного параметра.

А метод `sin` объекта `Math` вычисляет синус угла, заданного в радианах и переданного данному методу единственным параметром:

```
var b = Math.sin(0.1);
```

Как видим, мы просто используем переменную `Math`, созданную языком JavaScript, чтобы получить доступ к экземпляру объекта `Math` и его методам.

Web-обозреватель также предоставляет множество экземпляров объектов, представляющих текущую Web-страницу, различные ее элементы и сам Web-обозреватель. Так, Web-страницу представляет экземпляр объекта `HTMLDocument`, который хранится в переменной `document`, также созданной Web-обозревателем.

В частности, объект `HTMLDocument` поддерживает метод `write`, выводящий переданную ему в качестве единственного параметра строку в то место Web-страницы, где встретился вызов этого метода:

```
document.write("Привет, посетитель!");
```

И в этом случае мы просто используем переменную `document`, созданную Web-обозревателем, чтобы получить доступ к экземпляру объекта `HTMLDocument` и его методу `write`.

Сторонние библиотеки также создают множество экземпляров объектов сами. Так, библиотека Ext Core создает экземпляр объекта `Ext`, хранящийся в одноименной переменной:

```
var ceLinks = Ext.select("UL[id=navbar] > LI");
```

Здесь мы обратились к методу `select` объекта `Ext`. Данный метод возвращает массив экземпляров объектов `Element`, каждый из которых представляет элемент Web-

страницы, удовлетворяющий переданному ему в качестве единственного параметра специальному селектору CSS. (Кстати, объект `Element` также определен библиотекой Ext Core.)

Идем далее. Ранее было сказано, что экземпляр объекта можно получить в качестве результата выполнения функции или метода. Вот типичный пример:

```
var elNavbar = Ext.get("navbar");
```

Метод `get` объекта `Ext`, созданного библиотекой Ext Core, возвращает экземпляр объекта `Element`, представляющий определенный элемент Web-страницы. В данном случае это будет элемент, значение атрибута тега `ID` которого равно `"navbar"` — именно такую строку мы передали методу `get` в качестве параметра.

Здесь мы также получили готовый экземпляр объекта, с которым сразу можем начать работу.

И, наконец, с помощью *оператора создания экземпляра* `new` мы можем создать экземпляр нужного нам объекта сами, явно:

```
new <имя объекта>([<список параметров, разделенных запятыми>])
```

Оператор `new` возвращает созданный экземпляр объекта. Его можно присвоить какой-либо переменной или свойству или передать в качестве параметра функции или методу.

`Список параметров` может как присутствовать, так и отсутствовать. Обычно он содержит значения, которые присваиваются свойствам экземпляра объекта при его создании.

Язык JavaScript предоставляет нам объект `Date`, хранящий значение даты и времени. Его экземпляры можно создать только явно, оператором `new`:

```
var dNow = new Date();
```

После выполнения этого выражения в переменной `dNow` окажется экземпляр объекта `Date`, хранящий сегодняшнюю дату и текущее время.

А вот выражение, которое поместит в переменную `dNewYear` экземпляр объекта `Date`, хранящий дату 31 декабря 2009 года и текущее время:

```
var dNewYear = new Date(2009, 12, 31);
```

Но какой способ получения экземпляра объекта в каком случае применять? Выбор зависит от объекта, экземпляр которого мы хотим получить.

- Экземпляры объектов Web-обозревателя, представляющие Web-страницу и сам Web-обозреватель, доступны через соответствующие переменные.
- Экземпляры объектов Web-обозревателя, представляющие различные элементы Web-страницы, получаются в результате выполнения особой функции или метода.
- Экземпляры всех объектов языка JavaScript, кроме `Math`, получаются явным созданием с помощью оператора `new`.

- Экземпляр объекта `Math` языка JavaScript доступен через одноименную переменную.

Это самые общие правила. А конкретные приемы получения нужного экземпляра объекта мы рассмотрим потом, когда начнем заниматься Web-программированием вплотную.

Работа с экземпляром объекта

Получив тем или иным способом экземпляр объекта, мы можем с ним работать: вызывать его методы и получать доступ к его свойствам.

Ранее мы уже познакомились с методами различных объектов и выяснили, как они вызываются. Для этого к переменной, хранящей экземпляр объекта, добавляется справа точка, а после нее записывается вызов метода. Метод — суть "собственная" функция объекта, и вызывается она так же (см. раздел, посвященный вызовам функций).

Пример:

```
var elNavbar = Ext.get("navbar");
```

Здесь мы вызвали метод `get` объекта `Ext`, экземпляр которого хранится в переменной `Ext`, передав данному методу параметр — строку `"navbar"`. Как мы уже знаем, метод `get` ищет на Web-странице элемент, имеющий значение атрибута тега `ID`, которое совпадает с переданной ему в качестве параметра строкой. Возвращенный этим методом результат — экземпляр объекта `Element`, представляющего список `navbar`, — будет присвоен переменной `elNavbar`.

Пример:

```
var elParent = elNavbar.parent();
```

А здесь мы вызвали метод `parent` у экземпляра объекта `Element`, хранящегося в переменной `elNavbar` и полученного в предыдущем выражении. Возвращенный этим методом результат — экземпляр объекта `Element`, представляющего элемент Web-страницы, который является родителем списка `navbar`, — будет присвоен переменной `elParent`.

Если результат, возвращаемый каким-либо методом (назовем его методом 1), представляет собой объект, мы можем сразу вызвать метод у него (это будет метод 2). Для этого мы добавим справа от вызова метода 1 точку, после которой поставим вызов метода 2.

Пример:

```
elNavbar.parent().addClass("hovered");
```

Здесь мы сначала вызвали метод `parent` у экземпляра объекта `Element`, хранящегося в переменной `elNavbar` (метод 1). У полученного в результате вызова метода 1 результата — экземпляра объекта `Element`, представляющего элемент-родитель, — мы вызвали метод `addClass` (метод 2).

Такие цепочки последовательных вызовов методов, когда стоящий справа метод вызывается у экземпляра, возвращенного методом, стоящим слева, встречаются в JavaScript-коде очень часто.

Доступ к свойствам объекта выполняется аналогично. К переменной, хранящей экземпляр объекта, добавляется справа точка, а после нее записывается имя свойства.

Пример:

```
var sID = elParent.id;
```

В данном примере мы обратились к свойству `id` объекта `Element`, экземпляр которого хранится в переменной `elParent`. Это свойство хранит значение атрибута тега `ID` у соответствующего элемента Web-страницы.

Пример:

```
Ext.enableFx = false;
```

Здесь мы присвоили новое значение свойству `enableFx` объекта `Ext`, экземпляр которого хранится в переменной `Ext`.

Пример:

```
var sID = elNavbar.parent().id;
```

А здесь мы сначала получили родитель элемента Web-страницы, хранящегося в переменной `elNavbar`, после чего извлекли значение атрибута тега `ID` уже у полученного родителя.

Что ж, с объектами и их экземплярами, свойствами и методами мы познакомились. Теперь давайте кратко "пробежимся" по объектам, которые будем использовать при написании Web-сценариев. Рассмотрим только встроенные объекты JavaScript и объекты Web-обозревателя; пользовательским объектам, создаваемым библиотекой Ext Core, будет полностью посвящена *глава 15*.

Встроенные объекты языка JavaScript

Ранее мы познакомились со встроенным объектом `Date`, который предоставляется самим языком JavaScript и служит для хранения значений даты и времени:

```
var dNow = new Date();
```

Объект `Date` поддерживает ряд методов, позволяющих получать отдельные составляющие даты и времени и манипулировать ими. Так, метод `getDate` возвращает число, `getMonth` — номер месяца, а `getFullYear` — год. Все эти методы не принимают параметров, а возвращаемые ими результаты представляют собой числа.

Пример:

```
var sNow = dNow.getDate() + "." + dNow.getMonth() + "." +
dNow.getFullYear();
```

Здесь мы объединяем в одну строку число, номер месяца и год, разделяя их точками. Таким образом мы получим значение даты в формате *<число>.<месяц>.<год>*.

При этом JavaScript сам выполняет неявное преобразование числовых величин в строки.

Объект `String` служит для хранения строк.

```
var s = "JavaScript";
```

Мы только что создали экземпляр объекта `String`, хранящий строку `JavaScript`.

Здесь мы столкнулись с одной особенностью JavaScript, отличающей его от других языков программирования. Все значения простых типов данных в нем на самом деле являются экземплярами соответствующих объектов. Так, строка — это фактически экземпляр объекта `String`.

Свойство `length` объекта `String` хранит длину строки в символах:

```
var l = s.length;
var l = "JavaScript".length;
```

Эти выражения помещают в переменную `l` длину строки `JavaScript`.

Метод `substr` возвращает фрагмент строки заданной длины, начинающийся с указанного символа:

`substr(`*<номер первого символа>*`[, `*<длина фрагмента>*`]);`

Первым параметром передается номер первого символа, включаемого в возвращаемый фрагмент строки.

> ***Внимание!***
> В JavaScript символы в строках нумеруются, начиная с нуля.

Второй, необязательный, параметр задает длину возвращаемого фрагмента в символах. Если он опущен, возвращаемый фрагмент будет содержать все оставшиеся символы строки.

После выполнения Web-сценария

```
var s1 = s.substr(4);
var s2 = s.substr(4, 2);
```

в переменной `s1` окажется строка `"Script"`, а в переменной `s2` — строка `"Sc"`.

Объект `Number` служит для хранения чисел, а объект `Boolean` — логических величин:

```
var n = 123;
var b = false;
```

Числа и логические величины с точки зрения JavaScript также представляют собой экземпляры соответствующих объектов.

Объект `Array` служит для хранения массивов:

```
var a = [1, 2, 3, 4];
```

Он поддерживает единственное свойство `length`, возвращающее размер массива, т. е. число элементов в нем:

```
var l = a.length;
var l = [1, 2, 3, 4].length;
```

Уже знакомый нам объект `Math`, единственный экземпляр которого создается самим JavaScript и хранится в переменной `Math`, представляет набор методов для выполнения математических и тригонометрических вычислений.

Мы не будем рассматривать здесь все встроенные объекты JavaScript и поддерживаемые ими свойства и методы. Это можно найти в любой книге по JavaScript-программированию.

Объект *Object* и использование его экземпляров

Но об одном встроенном объекте следует поговорить особо. Это объект `Object`, весьма специфический.

Экземпляры этого объекта обычно используются для хранения сложных структур данных, включающих произвольный набор свойств и методов. Один экземпляр объекта `Object` может иметь один набор свойств и методов, а другой экземпляр — совсем другой.

Экземпляры объекта `Object` создают с помощью особых выражений, называемых *инициализаторами*. Инициализатор чем-то похож на определение стиля (листинг 14.12).

Листинг 14.12

```
{
  <имя свойства 1>: <значение свойства 1>,
  <имя свойства 2>: <значение свойства 2>,
  . . .
  <имя свойства n-1>: <значение свойства n-1>;
  <имя свойства n>: <значение свойства n>
  <имя метода 1>: <функция, реализующая метод 1>,
  <имя метода 2>: <функция, реализующая метод 2>,
  . . .
  <имя метода n-1>: <функция, реализующая метод n-1>,
  <имя метода n>: <функция, реализующая метод n>
}
```

После выполнения инициализатора JavaScript вернет нам готовый экземпляр объекта `Object`, который мы можем присвоить какой-либо переменной или использовать в качестве параметра функции или метода.

Пример:

```
var oConfig = { tag: "DIV",
                id: "cother",
                html: "Это прочие сведения." };
```

Здесь мы получили экземпляр объекта `Object` со свойствами `tag`, `id` и `html`, задали для этих свойств значения и сохранили получившийся экземпляр в переменной `oConfig`.

Пример:

```
var oConfig2 = { url: "pages/t_img.htm",
                 success: function (response, opts){
                   var obj = Ext.decode(response.responseText);
                 }
               };
```

А здесь мы создали экземпляр объекта `Object` со свойством `url` и методом `success` и сохранили получившийся экземпляр в переменной `oConfig2`. (Код последнего примера взят из документации по библиотеке Ext Core.)

Обратим внимание на два момента. Во-первых, функцию, реализующую метод `success`, мы объявили прямо в инициализаторе. Во-вторых, создание метода в данном случае — суть присваивание функции, которая реализует этот метод, свойству, имя которого станет именем метода. Следовательно, здесь тот же самый случай, что и с присваиванием функции переменной (см. раздел, посвященный функциям).

Экземпляры объекта `Object` в библиотеке Ext Core обычно служат для задания различных необязательных параметров и создаются как раз с помощью инициализаторов. Так что мы часто будем иметь с ними дело.

Объекты Web-обозревателя. Объектная модель документа DOM

Объекты, предоставляемые Web-обозревателем, делятся на две группы:

- объекты, представляющие Web-страницу и элементы, созданные с помощью разных тегов (абзац, заголовок, таблица, изображение и др.);
- объекты, представляющие сам Web-обозреватель.

Начнем с объектов первой группы.

Как мы уже знаем, саму Web-страницу представляет объект `HTMLDocument`. Единственный экземпляр данного объекта хранится в переменной `document` и представляет Web-страницу, открытую в текущем окне Web-обозревателя.

Отдельный элемент Web-страницы, независимо от тега, с помощью которого он создан, представляется объектом `HTMLElement`. На этом объекте основаны другие объекты, представляющие элементы Web-страницы, которые созданы на основе определенных тегов. Так, абзац представляется объектом `HTMLParagraphElement`, изображение — объектом `HTMLImageElement`, гиперссылка — объектом `HTMLLinkElement`, а таблица — объектом `HTMLTableElement`.

Для каждого элемента загруженной Web-страницы Web-обозреватель создает экземпляр соответствующего объекта. Например, для каждого абзаца создается экземпляр объекта `HTMLParagraphElement`, для каждого изображения — экземпляр объекта `HTMLImageElement`, для каждой гиперссылки — экземпляр объекта `HTMLLinkElement`, а для каждой таблицы — экземпляр объекта `HTMLTableElement`.

В результате в памяти компьютера создается структура взаимосвязанных экземпляров объектов, соответствующая структуре элементов Web-страницы. Она называет-

ся *объектной моделью документа*, или *DOM* (сокращение от Document Object Model — объектная модель документа).

Объект `HTMLDocument` поддерживает ряд методов для доступа к нужному элементу Web-страницы, в смысле — к представляющему его экземпляру соответствующего объекта. Обычно для уникальной идентификации элемента Web-страницы используется значение атрибута тега `ID`. Мы поговорим об этом подробнее в *главе 15*.

Объект `HTMLElement` поддерживает свойства и методы, общие для всех типов элементов Web-страницы. Они позволяют получить или задать значение какого-либо атрибута тега, привязать к нему стилевой класс, вставить в элемент Web-страницы другой элемент в качестве дочернего и пр. Объекты, созданные на основе `HTMLElement`, расширяют этот набор свойств и методов.

DOM является одним из основополагающих стандартов Интернета, разрабатывается и утверждается организацией W3C. Все Web-обозреватели обязаны ее поддерживать.

Что касается объектов второй группы, то их немного. Это, прежде всего, объект `Window`, представляющий окно Web-обозревателя и поддерживающий ряд свойств и методов, с помощью которых мы можем им управлять. Экземпляр этого объекта, представляющий текущее окно Web-обозревателя, хранится в переменной `window`. Кроме того, существует еще несколько объектов, представляющих Web-обозреватель, но они встречаются значительно реже.

К рассмотрению объектов Web-обозревателя мы вернемся в *главе 15*. А пока что закончим с ними.

Свойства и методы экземпляра объекта

Как мы уже знаем, объект определяет набор свойств и методов, которые затем станут "собственностью" всех его экземпляров. Иными словами, экземпляр объекта получит все свойства и методы, что объявлены в объекте, на основе которого он создан. Это так называемые *свойства* и *методы объекта*.

Однако язык JavaScript предоставляет возможность создать у экземпляра объекта сколько угодно свойств и методов, которые будут принадлежать только ему. Другие экземпляры того же объекта эти свойства и методы не получат.

Пример:

```
var dNow = new Date();
dNow.someProperty = 3;
```

Здесь мы создали у экземпляра объекта `Date`, хранящегося в переменной `dNow`, свойство `someProperty` и присвоили ему значение `3`. Данное свойство будет принадлежать только этому экземпляру объекта `Date`.

Пример:

```
elNavbar.someMethod = function() { . . . };
```

Здесь мы добавили к экземпляру объекта `Element` библиотеки Ext Core, хранящемуся в переменной `elNavbar`, метод `someMethod`. Опять же, данный метод будет принадлежать только этому экземпляру объекта.

А в следующем примере мы добавили к созданному ранее конфигуратору `oConfig` свойство `style` и присвоили этому свойству строку с определением встроенного стиля:

```
oConfig.style = "color: red;";
```

Такие свойства и методы, принадлежащие только одному экземпляру объекта, называются *свойствами* и *методами экземпляра*.

В Web-программировании свойства и методы экземпляра позволяют связать какой-либо элемент Web-страницы с некоторыми данными, которые будут использованы далее в Web-сценарии применительно к этому элементу. В последующих главах мы рассмотрим такие случаи.

Правила написания выражений

В процессе чтения этой главы мы изучили множество выражений JavaScript. Но так и не узнали, по каким правилам они пишутся. Настала пора восполнить пробел в наших знаниях.

- Между операндами, операторами, вызовами функций и методов и ключевыми словами допускается сколько угодно пробелов и разрывов строк.
- Запрещаются переносы строк внутри константы, ключевого слова, имени переменной, свойства, функции, метода или объекта. В противном случае мы получим сообщение об ошибке.
- Признаком конца выражения служит символ точки с запятой (`;`).

В качестве примера давайте возьмем одно из выражений из начала главы:

```
y = y1 * y2 + x1 * x2;
```

Мы можем записать его так, разнеся на две строки:

```
y = y1 * y2 +
x1 * x2;
```

Или даже так:

```
y =
y1 *
y2 +
x1 *
x2;
```

И в любом случае оно будет выполнено, т. к. при написании выражения мы не нарушили ни одного из перечисленных правил.

Но если мы нарушим их, выполнив перенос строк внутри имени переменной `y2`:

```
y = y1 * y
2 + x1 * x2;
```

получим сообщение об ошибке.

JavaScript весьма либерально относится к тому, как мы пишем его выражения. Благодаря этому мы можем форматировать JavaScript-код с помощью разрывов строк и пробелов для удобства его чтения.

Комментарии JavaScript

Из *глав 2* и *7* мы знаем о существовании комментариев — особых фрагментов кода HTML и CSS, которые не обрабатываются Web-обозревателем и служат для того, чтобы Web-дизайнер смог оставить какие-либо заметки для себя или своих коллег. Было бы странно, если бы JavaScript не предоставлял аналогичной возможности.

Комментарии JavaScript бывают двух видов.

Комментарий, состоящий из одной строки, создают с помощью символа `/` (слэш), который помещают в самом начале строки комментария:

```
/ Это комментарий
var dNow = new Date();
```

Однострочный комментарий начинается с символа `/` и заканчивается концом строки.

Комментарий, состоящий из произвольного числа строк, создают с помощью последовательностей символов `/*` и `*/`. Между ними помещают строки, которые станут комментарием:

```
/*
  Это комментарий,
  состоящий из
  нескольких строк.
*/
var dNow = new Date();
```

Многострочный комментарий начинается с последовательности символов `/*` и заканчивается последовательностью `*/`.

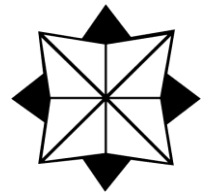
Вот и все о языке JavaScript.

Что дальше?

В этой главе мы изучили язык JavaScript, познакомились с его операторами, выражениями, функциями, массивами и объектами. Также мы узнали об объектной модели документов, представляющей Web-страницу и все ее элементы в виде экземпляров соответствующих объектов. И даже создали поведения для своих Web-страниц.

В следующей главе мы изучим библиотеку Ext Core, которой будем пользоваться в дальнейшем. Также мы познакомимся с некоторыми объектами Web-обозревателя, с которыми будем работать напрямую. А еще узнаем о событиях и их обработке.

ГЛАВА 15



Библиотека Ext Core и объекты Web-обозревателя

В предыдущей главе мы узнали, как создается поведение Web-страниц, и познакомились с Web-сценариями и языком программирования JavaScript, на котором они пишутся. Еще мы написали два простых Web-сценария, один из которых выводил на Web-страницу текущую дату, а другой менял цвет рамки у элементов полосы навигации при наведении на них курсора мыши. И незаметно для себя стали Web-программистами...

Также мы получили в свое распоряжение библиотеку Ext Core, призванную облегчить труд Web-программистов — нас с вами. О, это замечательная библиотека!..

Замечательная настолько, что ей будет посвящена почти вся эта глава. Еще мы рассмотрим пару объектов Web-обозревателя, с которыми мы будем работать напрямую, но разговор о них будет совсем коротким.

Итак, просим любить и жаловать — библиотека Ext Core!

Библиотека Ext Core

В этом разделе мы будем изучать самые полезные для нас на данный момент возможности библиотеки Ext Core. Полностью она описана в справочнике, доступном на ее "домашнем" Web-сайте.

Зачем нужна библиотека Ext Core

Но зачем нужна эта библиотека? Почему бы нам не работать напрямую с объектами Web-обозревателя, обращаясь к их свойствам и вызывая их методы? Почему, чтобы управлять содержимым Web-страницы, нужны дополнительные инструменты и дополнительные сложности?

В том-то и дело, что библиотека Ext Core призвана устранять сложности, а не создавать их.

Вспомним, что мы узнали в *главе 14*. Web-обозреватель представляет саму Web-страницу и отдельные ее элементы в виде экземпляров особых объектов — объек-

тов Web-обозревателя. В результате в памяти компьютера формируется структура экземпляров объектов, представляющая содержимое Web-страницы. Она называется объектной моделью документа, или DOM.

Организация W3C требует, чтобы DOM соответствовала разработанному стандарту, тесно связанному со стандартами HTML и CSS. Разумеется, все Web-обозреватели его поддерживают.

Однако использовать объекты Web-обозревателя для программного управления Web-страницей очень нелегко. В основном, из-за того, что эти объекты предусматривают только самые базовые возможности для доступа к экземплярам объектов, представляющим нужные элементы Web-страницы, и управления ими. И зачастую этих возможностей слишком мало для комфортного программирования.

Нет, элементарные вещи, наподобие изменения цвета рамки элемента при наведении на него курсора мыши, действительно реализовать просто. Но если нам понадобится нечто более сложное, например, подгружать фрагмент содержимого Web-страницы из другого файла или скрывать и раскрывать часть содержимого элемента при щелчке на нем, мы будем вынуждены решать многочисленные проблемы, связанные с тем, что инструментов, предлагаемых стандартом DOM, окажется недостаточно.

Поэтому опытные Web-программисты и создают дополнительные библиотеки, добавляющие к инструментам, предусмотренным стандартом DOM, свои собственные, как правило, более мощные. К таким библиотекам относится и Ext Core.

Далее. Стандарт — стандартом, но разные Web-обозреватели зачастую по-разному его поддерживают. Сплошь и рядом возникают ситуации, когда один и тот же Web-сценарий прекрасно работает, скажем, в Firefox, но никак не хочет правильно исполняться в Internet Explorer. Ладно, Internet Explorer, как говорится, — отрезанный ломоть, до сих пор не получивший поддержку HTML 5 и CSS 3, но ведь и Opera, и Chrome, и Safari тоже поддерживают некоторые аспекты стандарта DOM по-своему. Налицо несовместимость Web-обозревателей, которую приходится как-то обходить.

К тому же, некоторые объекты Web-обозревателя до сих пор не стандартизированы, и, насколько известно автору, даже не было предпринято попытки их стандартизировать. К таким объектам относятся те, что представляют сам Web-обозреватель, в частности, объект `Window`.

Поэтому сторонние библиотеки, расширяющие возможности DOM, еще и устраняют несовместимость Web-обозревателей. И Ext Core — не исключение.

Сейчас такие библиотеки — Prototype, jQuery и "героиня" этой главы Ext Core — становятся все более и более популярными. Они применяются при программировании поведения для всех сложных Web-сайтов. В России даже выходят посвященные им книги; по крайней мере, автору встречалась книга, посвященная jQuery.

Ну а мы вернемся к Ext Core.

Использование библиотеки Ext Core

Библиотека Ext Core распространяется с Web-страницы **http://www.extjs.com/products/core/?ref=family** своего "домашнего" Web-сайта. На ней мы найдем гиперссылки на Web-страницы загрузки библиотеки, краткое руководство программиста с примерами и полный справочник по ней.

Библиотека Ext Core распространяется в виде архива ZIP, хранящегося в файле с именем вида ext-core-<*номер версии*>.zip и содержащего саму библиотеку, ее исходные коды в "читабельном" виде и несколько примеров. На момент написания книги была доступна версия 3.1.0, ее-то и использовал автор.

Сама библиотека Ext Core хранится в файле Web-сценария ext-core.js, который находится в описанном архиве. Она написана на языке JavaScript и представляет собой объявления многочисленных объектов, их свойств и методов и переменных, хранящих экземпляры объектов, — всем этим Web-программист будет активно пользоваться.

Код Ext Core нужно выполнить еще до загрузки Web-страницы, тогда библиотека сможет успешно создать все свои объекты, экземпляры объектов и переменные. Для этого в секцию тела Web-страницы (тег `<HEAD>`) помещают тег `<SCRIPT>` такого вида:

```
<SCRIPT SRC="ext-core.js"></SCRIPT>
```

Он указывает Web-обозревателю загрузить Web-сценарии, хранящиеся в файле ext-core.js и составляющие эту библиотеку, и выполнить их еще до того, как будет загружена секция тела Web-страницы. Таким образом, когда Web-страница будет выведена на экран, библиотека Ext Core окажется "во всеоружии".

Что касается Web-сценариев, составляющих поведение Web-страницы, то они хранятся в отдельном файле — так требует концепция Web 2.0. Тег `<SCRIPT>`, загружающий и выполняющий эти Web-сценарии, помещают в самом конце HTML-кода Web-страницы, перед закрывающим тегом `</BODY>`:

```
<SCRIPT SRC="<интернет-адрес файла Web-сценария, хранящего код
↳поведения для Web-страницы>.js"></SCRIPT>
```

Любые Web-сценарии, выполняющие манипуляции с элементами Web-страницы, должны быть выполнены только после того, как Web-страница будет полностью загружена, а соответствующая ей DOM — сформирована. Но как этого добиться? Даже если мы поместим тег `<SCRIPT>`, загружающий и выполняющий их, в самом конце HTML-кода Web-страницы, проблемы это не решит. Ведь, даже если Web-страница уже загружена, формирование DOM Web-страницы будет продолжаться, и когда оно закончится — неизвестно.

Ext Core предоставляет нам элегантное решение: все Web-сценарии, которые должны быть выполнены после завершения загрузки Web-страницы, оформляются в виде тела функции, не принимающей параметров. Эта функция передается в качестве параметра методу `onReady` объекта `Ext`:

```
Ext.onReady(function() {
  <код, выполняемый после загрузки Web-страницы>
});
```

Метод `onReady` сохраняет переданную ему функцию в особом свойстве единственного экземпляра объекта `Ext` и впоследствии, когда Web-страница будет полностью загружена, а ее DOM — сформирована, вызывает ее.

Собственно, так мы и поступили в *главе 14*, когда создавали второй Web-сценарий.

Ключевые объекты библиотеки Ext Core

Как уже неоднократно говорилось, библиотека Ext Core объявляет несколько объектов, которые представляют различные элементы Web-страницы, специальные структуры данных, применяемые при программировании, и саму эту библиотеку. Настала пора рассмотреть некоторые из этих объектов хотя бы вкратце.

> ***НА ЗАМЕТКУ***
>
> Язык JavaScript предоставляет средства для создания новых объектов либо "с нуля", либо на основе уже существующих. Однако эти средства довольно сложны, и начинающим Web-программистам они ни к чему. Поэтому мы не будем их рассматривать.

Прежде всего, это объект `Ext`, о котором мы уже наслышаны. Он предоставляет набор методов для доступа к нужному элементу Web-страницы, знакомый нам метод `onReady` и несколько служебных методов. Кроме того, он служит "вместилищем" для остальных объектов библиотеки `Ext`.

Единственный экземпляр объекта `Ext` создается самой библиотекой Ext Core и хранится в переменной `Ext`.

С объектом `Element` (точнее, `Ext.Element`, поскольку он хранится в объекте `Ext`) мы также знакомы по *главе 14*. Он представляет элемент Web-страницы, независимо от формирующего его тега.

Объект `Element` фактически заменяет "универсальный" объект Web-обозревателя `HTMLElement` и все созданные на его основе объекты, представляющие отдельные теги. Он поддерживает множество методов, позволяющих манипулировать элементом Web-страницы, привязывать к нему стилевые классы, задавать отдельные параметры стиля и вставлять в него другие элементы в качестве дочерних. Работать с экземплярами объекта `Element` несравнимо проще, чем с экземплярами объектов Web-обозревателя.

Экземпляры объекта `Element` можно получить только в результате вызова определенных методов этого же или других объектов.

Весьма примечательный объект `CompositeElementLite` (точнее, `Ext.CompositeElementLite`) представляет массив экземпляров объекта `Element` (о массивах было рассказано в *главе 14*). Такие объекты-массивы называются *коллекциями* и, в отличие от обычных массивов, поддерживают дополнительный набор методов, позволяющих манипулировать отдельными экземплярами объектов, составляющими коллекцию (ее *элементами*).

Экземпляры объекта `CompositeElementLite` получают в результате вызова определенных методов других объектов. Они содержат экземпляры объекта `Element`, представляющие набор элементов Web-страницы, которые удовлетворяют некому критерию, например, селектору CSS.

Объект `DomHelper` (полное имя — `Ext.DomHelper`) служит для создания новых элементов Web-страницы на основе указанных тега, значений его атрибутов и содержимого. В результате получается экземпляр объекта `Element`, представляющий созданный элемент.

Единственный экземпляр объекта `DomHelper` создается самой библиотекой Ext Core и хранится в переменной `Ext.DomHelper`.

Объект `EventObject` (`Ext.EventObject`) служит для хранения сведений о возникшем в элементе Web-страницы событии. (О событиях будет рассказано далее, в соответствующем разделе.)

Экземпляр объекта `EventObject` создается самой библиотекой Ext Core и передается в функцию — обработчик события первым параметром.

Доступ к нужному элементу Web-страницы

В самом деле, перед тем как начать манипулировать элементом Web-страницы, точнее, представляющим его экземпляром объекта `Element`, нужно как-то получить к нему доступ. Как?

Здесь нам помогут методы объекта `Ext`, которые мы сейчас рассмотрим.

Метод `get` возвращает экземпляр объекта `Element`, представляющий определенный элемент Web-страницы:

```
Ext.get(<значение атрибута тега ID>|<экземпляр объекта HTMLElement>)
```

Как видим, этот метод принимает один параметр. Им может быть строка, содержащая значение атрибута тега `ID`, по которому будет выполняться поиск элемента Web-страницы.

Пример:

```
var elNavbar = Ext.get("navbar");
```

Здесь мы получили экземпляр объекта `Element`, представляющий "внешний" список `navbar`, что формирует полосу навигации.

В *главе 7* мы узнали, что атрибут тега `ID` обеспечивает привязку к элементу Web-страницы именованного стиля. Но чаще всего он используется, чтобы дать элементу уникальное в пределах Web-страницы имя. В таком случае говорят, что такой-то элемент Web-страницы имеет такое-то имя, например, "внешний" список, формирующий нашу полосу навигации, имеет имя `navbar`.

Еще мы можем передать методу `get` экземпляр объекта `HTMLElement`, представляющий нужный нам элемент Web-страницы. Такой вызов данного метода применяют, если хотят создать на основе экземпляра объекта `HTMLElement` экземпляр объекта

`Element` и получить в свои руки всю мощь Ext Core. Кстати, мы с этим потом столкнемся.

Метод `get` имеет важную особенность, о которой мы обязательно должны знать. Дело в том, что библиотека Ext Core при инициализации объявляет внутренний массив, или, если точнее, хэш (см. *главу 14*). При первом доступе к какому-либо элементу Web-страницы метод `get` создает представляющий данный элемент экземпляр объекта `Element` и помещает его в этот массив. При повторном доступе к тому же самому элементу Web-страницы соответствующий ему экземпляр объекта `Element` просто извлекается из данного массива. Такой подход позволяет значительно увеличить быстродействие, ведь извлечение элемента массива выполняется много быстрее, чем создание экземпляра объекта.

Сохранение каких-либо структур данных во внутреннем хранилище с целью ускорения к ним доступа называется *кэшированием*. А само это внутреннее хранилище (в случае библиотеки Ext Core — массив) называется *кэшем*.

Однако кэширование имеет и недостатки. Предположим, что мы создали Web-страницу с множеством элементов, которыми планируем управлять программно, из Web-сценариев. Причем доступ к каждому из этих элементов мы получаем всего один раз за все время, пока Web-страница открыта в Web-обозревателе, после чего больше их не трогаем. В результате кэш быстро "засорится" экземплярами объекта `Element`, что представляют не нужные нам более элементы Web-страницы. Что приведет к лишней трате памяти.

Поэтому создатели Ext Core предусмотрели метод `fly`, применяемый именно в таких случаях:

```
Ext.fly(<значение атрибута тега ID>|<экземпляр объекта HTMLElement>)
```

Он полностью аналогичен только что рассмотренному нами методу `get` с одним исключением — он кэширует элементы Web-страницы (в смысле — представляющие их экземпляры объекта `Element`) не в массиве, а в обычной переменной. Это значит, что в кэше хранится только элемент Web-страницы, к которому обращались при последнем вызове этого метода:

```
var elNavbar = Ext.fly("navbar");
```

Также метод `fly` работает быстрее, чем метод `get`, за счет того, что ему не нужно искать экземпляр объекта `Element`, соответствующего запрашиваемого элементу Web-страницы, в массиве, который может быть очень большим. Ему достаточно проверить всего одну переменную.

- Метод `get` следует использовать, если данный элемент Web-страницы понадобится нам в дальнейшем.
- Метод `fly` полезен, если нам требуется получить доступ к элементу Web-страницы всего один раз.

Не принимающий параметров метод `getBody` возвращает экземпляр объекта `Element`, представляющий секцию тела Web-страницы (тег `<BODY>`):

```
var elBody = Ext.getBody();
```

Метод `getDom` возвращает экземпляр объекта Web-обозревателя `HTMLElement`, представляющий определенный элемент `Web-страницы`:

```
Ext.getDom(<значение атрибута тега ID>|<экземпляр объекта Element>)
```

Этот метод принимает один параметр, которым может быть строка с именем элемента Web-страницы (значением атрибута `ID` его тега) или экземпляр объекта `Element`, представляющий этот элемент.

Пример:

```
var htelNavbar = Ext.getDom("navbar");
```

Здесь мы получили экземпляр объекта `HTMLElement`, представляющий "внешний" список `navbar`.

Пример:

```
var elCMain = Ext.get("cmain");
var htelCMain = Ext.getDom(elCMain);
```

Здесь мы в два этапа получили экземпляр объекта `HTMLElement`, представляющий контейнер `cmain`. На первом этапе мы с помощью метода `get` получили представляющий его экземпляр объекта `Element` библиотеки Ext Core, а на втором — вызовом метода `getDom` — экземпляр объекта `HTMLElement` Web-обозревателя.

Свойство `dom` объекта `Element` возвращает экземпляр объекта Web-обозревателя `HTMLElement`, представляющий элемент Web-страницы:

```
var elCMain = Ext.get("cmain");
var htelCMain = elCMain.dom;
```

Некоторые методы объектов библиотеки Ext Core требуют в качестве параметров экземпляр объекта `HTMLElement`. Так что свойство `dom` и метод `getDom` нам пригодятся.

Доступ сразу к нескольким элементам Web-страницы

Зачастую приходится выполнять одинаковые манипуляции не с одним, а сразу с несколькими элементами Web-страницы, соответствующие одному критерию (обычно это селектор CSS).

Метод `select` объекта `Ext` возвращает экземпляр объекта `CompositeElementLite`, содержащий экземпляры объекта `Element`, которые представляют все элементы Web-страницы, что удовлетворяют заданному селектору CSS:

```
Ext.select(<селектор CSS>)
```

Единственным параметром этому методу передается строка с одним или несколькими селекторами. Если строка содержит несколько селекторов, их отделяют друг от друга запятыми.

Библиотека Ext Core существенно расширяет набор селекторов по сравнению с поддерживаемыми стандартом CSS. Давайте их рассмотрим.

- *<имя тега>* — элемент, созданный с помощью *тега*.
- *<имя тега 1> <имя тега 2>* — элемент, созданный с помощью *тега 2* и вложенный в *тег 1*, не обязательно непосредственно (может быть вложен в другой тег, вложенный в *тег 1*, или даже в несколько таких тегов последовательно).
- *<имя тега 1> > <имя тега 2>* или *<имя тега 1>/<имя тега 2>* — элемент, созданный с помощью *тега 2* и непосредственно вложенный в *тег 1*.
- *<имя тега 1> + <имя тега 2>* — элемент, созданный с помощью *тега 2*, которому непосредственно предшествует *тег 1* того же уровня вложенности.
- *<имя тега 1> ~ <имя тега 2>* — элемент, созданный с помощью *тега 2*, которому предшествует *тег 1* того же уровня вложенности, не обязательно непосредственно.
- `*` — элемент, созданный с помощью любого тега.
- `.`*<имя стилевого класса>* — элемент с привязанным *стилевым классом*.
- `[`*<имя атрибута тега>*`]` — элемент, тег которого включает *атрибут*.
- `[`*<имя атрибута тега>*`=`*<значение>*`]` — элемент, тег которого включает *атрибут* с заданным *значением*.
- `[`*<имя атрибута тега>*`!=`*<подстрока>*`]` — элемент, тег которого включает *атрибут* со значением, не равным *подстроке*.
- `[`*<имя атрибута тега>*`^=`*<подстрока>*`]` — элемент, тег которого включает *атрибут* со значением, начинающимся с заданной *подстроки*.
- `[`*<имя атрибута тега>*`$=`*<подстрока>*`]` — элемент, тег которого включает *атрибут* со значением, заканчивающимся заданной *подстрокой*.
- `[`*<имя атрибута тега>*`*=`*<подстрока>*`]` — элемент, тег которого включает *атрибут* со значением, включающим заданную *подстроку*.
- `[`*<имя атрибута тега>*`%=2]` — элемент, тег которого включает *атрибут* со значением, которое без остатка делится на 2.
- `:first-child` — первый потомок данного элемента.
- `:last-child` — последний потомок данного элемента.
- `:only-child` — единственный потомок данного элемента.
- `:nth-child(`*<номер>*`)` — потомок данного элемента с заданным *номером*.
- `:nth-child(even)` или `:even` — четные потомки данного элемента.
- `:nth-child(odd)` или `:odd` — нечетные потомки данного элемента.
- `:first` — первый элемент из соответствующих селектору.
- `:last` — последний элемент из соответствующих селектору.
- `:nth(`*<номер>*`)` — элемент с заданным *номером* из соответствующих селектору.
- `:contains(`*<подстрока>*`)` — элемент, содержимое которого включает заданную *подстроку*.

- :nodeValue(*<подстрока>*) — элемент, содержимое которого равно заданной *подстроке*.
- :not(*<селектор>*) — элемент, не удовлетворяющий *селектору*.
- :has(*<селектор>*) — элемент, который имеет хотя бы один потомок, удовлетворяющий *селектору*.
- :next(*<селектор>*) — элемент, следующий за которым элемент того же уровня вложенности удовлетворяет *селектору*.
- :prev(*<селектор>*) — элемент, предшествующий которому элемент того же уровня вложенности удовлетворяет *селектору*.
- {*<имя атрибута стиля>*=*<значение>*} — элемент, стиль которого имеет *атрибут* с заданным *значением*.
- {*<имя атрибута стиля>*!=*<подстрока>*} — элемент, стиль которого имеет *атрибут* со значением, не равным заданной *подстроке*.
- {*<имя атрибута стиля>*^=*<подстрока>*} — элемент, стиль которого имеет *атрибут* со значением, начинающимся с заданной *подстроки*.
- {*<имя атрибута стиля>*$=*<подстрока>*} — элемент, стиль которого имеет *атрибут* со значением, заканчивающимся заданной *подстрокой*.
- {*<имя атрибута стиля>**=*<подстрока>*} — элемент, стиль которого имеет *атрибут* со значением, включающим заданную *подстроку*.
- {*<имя атрибута стиля>*%=2} — элемент, стиль которого имеет *атрибут* со значением, без остатка делящимся на 2.

Если ни один подходящий элемент Web-страницы не был найден, метод select возвращает экземпляр объекта CompositeElementLite, не содержащий ни одного экземпляра объекта Element ("пустую" коллекцию).

Здесь мы получаем экземпляр объекта CompositeElementLite, содержащий экземпляры объекта Element, которые представляют все блочные контейнеры:

```
var clContainers = Ext.select("DIV");
```

А здесь мы получаем блочный контейнер cmain:

```
var clContainers = Ext.select("DIV[id=cmain]");
```

Здесь мы получаем все пункты "внешнего" списка navbar, формирующего полосу навигации:

```
var clOuterItems = Ext.select("UL[id=navbar] > LI");
```

А здесь мы получаем все первые абзацы, непосредственно вложенные в контейнеры:

```
var clP = Ext.select("DIV > P:first");
```

Здесь мы получаем все горизонтальные линии, которым непосредственно предшествуют абзацы того же уровня вложенности:

```
var clHR = Ext.select("P + HR");
```

А здесь мы получаем все абзацы и теги адреса на Web-странице:

```
var clPA = Ext.select("P, ADDRESS");
```

Доступ к родительскому, дочерним и соседним элементам Web-страницы

Теперь предположим, что мы наконец-то получили нужный нам элемент Web-страницы и хотим найти его родителя, потомка или "соседей" по уровню вложенности. Для этого Ext Core предоставляет нам множество методов объекта `Element`, которые будут описаны далее.

Метод `parent` возвращает родитель данного элемента Web-страницы в виде экземпляра объекта `Element`:

```
<экземпляр объекта Element>.parent([<селектор CSS>[, true]])
```

Первый, необязательный, параметр задает селектор CSS, которому должен удовлетворять родитель, в виде строки; можно также указать несколько селекторов через запятую. Если непосредственный родитель не удовлетворяет этому селектору, метод проверит родитель родителя и т. д., пока не будет найден подходящий элемент или достигнут тег с нулевым уровнем вложенности (тег `<HTML>`).

Если первый параметр не задан или с ним передана пустая строка, будет возвращен непосредственный родитель этого элемента.

Если вторым, также необязательным, параметром передано значение `true`, метод `parent` вернет экземпляр объекта Web-обозревателя `HTMLElement`.

Если подходящий родитель найден не будет, метод вернет значение `null`.

Здесь мы сначала получаем в переменной `elNavbar` "внешний" список `navbar`, формирующий полосу навигации, а потом в переменной `elCNavbar` — его непосредственного родителя:

```
var elNavbar = Ext.get("navbar");
var elCNavbar = elNavbar.parent();
```

Им окажется контейнер `cnavbar`.

А здесь мы пытаемся получить родителя списка `navbar`, который создан с помощью тега `<SPAN>`:

```
var elSpan = elNavbar.parent("SPAN");
```

Поскольку такого родителя у списка не существует, в переменной `elSpan` окажется значение `null`.

Метод `select` позволяет получить коллекцию дочерних элементов для данного элемента, удовлетворяющих заданному селектору, в виде экземпляра объекта `CompositeElementLite`:

```
<экземпляр объекта Element>.select(<селектор CSS>)
```

Единственным параметром этому методу передается строка с селектором или селекторами CSS.

Пример:

```
var clUL = elNavbar.select("LI > UL");
```

В переменной clUL окажется коллекция пунктов списка navbar, которые содержат вложенные списки.

Метод child возвращает первый встретившийся потомок данного элемента Web-страницы в виде экземпляра объекта Element:

```
<экземпляр объекта Element>.child([<селектор CSS>[, true]])
```

Первый, необязательный, параметр задает селектор CSS, которому должен удовлетворять потомок, в виде строки; можно также указать несколько селекторов через запятую. Если непосредственный потомок не удовлетворяет этому селектору, метод проверит потомки всех потомков данного элемента.

Если первый параметр не задан или с ним передана пустая строка, будут просматриваться все потомки данного элемента.

Если вторым, также необязательным, параметром передано значение true, метод child вернет экземпляр объекта Web-обозревателя HTMLElement.

Если подходящий потомок найден не будет, метод вернет значение null.

Пример:

```
var elUL = elNavbar.child();
```

В переменной elUL окажется первый пункт списка navbar.

Пример:

```
var elUL = elNavbar.child("LI:nodeValue=CSS");
```

В переменной elUL окажется пункт списка navbar, который содержит текст "CSS".

Метод down отличается от метода child тем, что ищет только среди непосредственных потомков текущего элемента Web-страницы:

```
<экземпляр объекта Element>.down([<селектор CSS>[, true]])
```

Параметры метода down те же, что у методов parent и child.

Пример:

```
var elUL = elNavbar.down();
```

В переменной elUL окажется первый пункт списка navbar.

Методы next и prev возвращают, соответственно, следующий и предыдущий элемент Web-страницы того же уровня вложенности, что и данный элемент:

```
<экземпляр объекта Element>.next|prev([<селектор CSS>[, true]])
```

Параметры этих методов те же, что у методов parent и child.

Пример:

```
var elDiv = Ext.get("cmain").next();
```

В переменной `elDiv` окажется контейнер `ccopyright` — следующий за контейнером `cmain`.

Пример:

```
var elP = elNavbar.prev();
```

В переменной `elP` окажется значение `null`, т. к. список `navbar` не имеет предыдущих элементов того же уровня вложенности и вообще является единственным потомком своего родителя.

Методы `first` и `last` возвращают, соответственно, первый и последний элемент Web-страницы того же уровня вложенности, что и данный элемент:

```
<экземпляр объекта Element>.first|last([<селектор CSS>[, true]])
```

Параметры этих методов те же, что у методов `parent` и `child`.

Пример:

```
var elCHeader = Ext.get("cmain").first();
var elCCopyright = Ext.get("cmain").last();
```

В переменной `elCHeader` окажется контейнер `cheader`, а в переменной `elCCopyright` — контейнер `ccopyright`. Это, соответственно, первый и последний из блочных контейнеров — "соседей" контейнера `cmain`.

Метод `is` возвращает `true`, если данный элемент Web-страницы совпадает с заданными селектором, и `false` в противном случае.

В примере из листинга 15.1 мы проверяем, создан ли контейнер `cmain` с помощью тега `<P>`. Разумеется, это не так.

Листинг 15.1

```
<экземпляр объекта Element>.is(<селектор CSS>)
var elCMain = Ext.get("cmain");
if (elCMain.is("P"))
  var s = "Это абзац."
else
  var s = "Это не абзац. Тьфу на него!";
```

Получение и задание размеров и местоположения элемента Web-страницы

Добравшись до нужного элемента Web-страницы, мы можем начать работать с ним, например, получить и задать его размеры и местоположение с помощью описанных в этом разделе методов объекта `Element`.

Методы `getWidth` и `getHeight` возвращают, соответственно, ширину и высоту данного элемента Web-страницы в виде числа в пикселах:

```
<экземпляр объекта Element>.getWidth|getHeight([true])
```

Если этим методам не передавать никаких параметров, они вернут полную ширину и высоту элемента Web-страницы, с учетом рамки и внутренних отступов. Если же им передать значение `true`, они вернут ширину и высоту только содержимого элемента, без учета рамки и внутренних отступов.

Пример:

```
var iWidth = Ext.get("cmain").getWidth();
```

В переменной `iWidth` окажется полная ширина контейнера `cmain`.

Методы `setWidth` и `setHeight` задают, соответственно, ширину и высоту данного элемента Web-страницы:

<экземпляр объекта Element>.setWidth|setHeight(*<значение>*)

Единственный параметр, передаваемый данным методам, — числовое значение ширины или высоты в пикселах:

```
Ext.get("cmain").setWidth(700);
```

Методы `getX` и `getY` возвращают, соответственно, горизонтальную и вертикальную координаты верхнего левого угла данного элемента Web-страницы в виде числа в пикселах. Координаты, возвращенные этими методами, отсчитываются относительно верхнего левого угла Web-страницы. Параметров эти методы не принимают.

Пример:

```
var elCMain = Ext.get("cmain");
var x = elCMain.getX();
var y = elCMain.getY();
```

Метод `getOffsetTo` возвращает смещение по горизонтали и вертикали данного элемента Web-страницы относительно другого элемента:

<экземпляр объекта Element>.getOffsetTo(*<экземпляр объекта Element>*)

В качестве параметра этому методу передается экземпляр объекта `Element`, представляющий элемент Web-страницы, относительно которого нужно узнать смещение данного элемента.

Метод `getOffsetTo` возвращает массив из двух элементов (чисел в пикселах): первый представляет смещение по горизонтали, второй — по вертикали.

Пример

```
var m = Ext.get("cmain").getOffsetTo(Ext.get("cnavbar"));
var x = m[0];
var y = m[1];
```

Здесь мы получим в переменных `x` и `y`, соответственно, горизонтальное и вертикальное смещения контейнера `cmain` относительно контейнера `cnavbar`.

Получение размеров Web-страницы и клиентской области окна Web-обозревателя

Также часто бывает нужно узнать размеры всей Web-страницы и внутренней части окна Web-обозревателя, в которой выводится содержимое Web-страницы (*клиентской области* окна). Для этого предназначены методы особого объекта `Ext.lib.Dom`.

Методы `getDocumentWidth` и `getDocumentHeight` возвращают полную, соответственно, ширину и высоту Web-страницы в числовом виде в пикселах. Параметров они не принимают:

```
var pageWidth = Ext.lib.Dom.getDocumentWidth();
var pageHeight = Ext.lib.Dom.getDocumentHeight();
```

Методы `getViewportWidth` и `getViewportHeight` возвращают полную, соответственно, ширину и высоту клиентской области окна Web-обозревателя также в числовом виде и в пикселах. Параметров они не принимают:

```
var clientWidth = Ext.lib.Dom.getViewportWidth();
var clientHeight = Ext.lib.Dom.getViewportHeight();
```

> ***Внимание!***
>
> Описанные здесь методы почему-то не документированы в справочнике по Ext Core. Автор обнаружил их в JavaScript-коде этой библиотеки.

Получение и задание значений атрибутов тега

Часто приходится получать и задавать значения атрибутов тега, с помощью которого создан элемент Web-страницы. Для этого Ext Core предоставляет два удобных метода и одно свойство объекта `Element`.

Метод `getAttribute` возвращает значение атрибута тега с указанным именем:

<экземпляр объекта Element>.getAttribute(*<имя атрибута тега>*)

В качестве параметра методу передается строка с именем атрибута тега. Метод возвращает строку с его значением.

Пример:

```
var s = Ext.get("cmain").child("A:first").getAttribute("href");
```

Здесь мы получаем значение атрибута тега `HREF` (интернет-адрес) первой гиперссылки в контейнере `cmain`.

Метод `set` задает новые значения для атрибутов тега:

<экземпляр объекта Element>.set(*<конфигуратор>*)

В *главе 14* мы узнали о встроенном объекте JavaScript `Object` и выражениях-инициализаторах, с помощью которых создаются его экземпляры. Также мы узнали, что в Ext Core экземпляры этого объекта применяются для задания параметров многих методов. Так вот, метод `set` — первый из изученных нами, где используется такой подход.

В качестве параметра этому методу передается экземпляр объекта `Object`. Имена его свойств должны совпадать с именами атрибутов тега, которым следует дать новые значения, а значения этих свойств собственно задают значения соответствующих атрибутов тега.

В терминологии Ext Core экземпляры объекта `Object`, задающие набор параметров для метода, называются *конфигураторами*.

Пример:

```
var oConf = { target: "_blank" };
var s = Ext.get("cmain").select("A").set(oConf);
```

Здесь мы задаем для всех гиперссылок в контейнере `cmain` значение атрибута стиля `TARGET`, равное `"_blank"`. Для этого мы используем конфигуратор, содержащий свойство `target` со значением `"_blank"`.

Свойство `id` возвращает строку со значением атрибута тега `ID`, т. е. имя элемента Web-страницы:

```
var sID = Ext.getBody().child("DIV:last").id;
```

Здесь мы получаем имя последнего контейнера на Web-странице — `"ccopyright"`.

К сожалению, задать новое имя для элемента Web-страницы с помощью свойства `id` мы не сможем. Конечно, можно присвоить этому свойству новое значение, но оно не будет перенесено в атрибут тега `ID` данного элемента. Так что нам придется воспользоваться методом `set`:

```
Ext.getBody().child("DIV:last").set({ id: "lastdiv" });
```

> ***Внимание!***
> Вообще, менять имя элемента Web-страницы в Web-сценарии — дурной тон программирования. Имя элемента должно задаваться всего один раз — при его создании.

Управление привязкой стилевых классов

Привязка и "отвязка" стилевых классов — одна из самых часто выполняемых в Web-сценариях операций. Было бы странно, если библиотека Ext Core не предоставляла средств для ее выполнения.

Методы объекта `Element`, которые мы сейчас рассмотрим, выполняют привязку стилевых классов к элементу Web-страницы и удаление их из привязки ("отвязку").

Метод `addClass` выполняет привязку указанного стилевого класса к данному элементу Web-страницы. Если такой стилевой класс уже есть в привязке, повторная его привязка не выполняется:

<экземпляр объекта Element>.addClass(*<имя стилевого класса>*)

В качестве параметра данному методу передается строка с именем привязываемого стилевого класса:

```
Ext.select("P").addClass("someclass");
```

Здесь мы привязываем ко всем абзацам на Web-странице стилевой класс `someclass`.

Метод `removeClass` удаляет указанный стилевой класс из привязки к данному элементу Web-страницы. Если такого стилевого класса в привязке нет, никаких действий не выполняется:

<экземпляр объекта Element>.removeClass(*<имя стилевого класса>*)

Параметр данного метода — строка с именем привязываемого стилевого класса:

```
Ext.select("P").removeClass("someclass");
```

Здесь мы удаляем привязанный ранее ко всем абзацам на Web-странице стилевой класс `someclass`.

Метод `toggleClass` привязывает заданный стилевой класс к элементу Web-страницы, если он еще не был привязан, и удаляет его из привязки в противном случае:

<экземпляр объекта Element>.toggleClass(*<имя стилевого класса>*)

Параметр данного метода — строка с именем привязываемого стилевого класса:

```
Ext.select("P").toggleClass("someclass");
```

Метод `replaceClass` удаляет из привязки к данному элементу Web-страницы указанный стилевой класс и привязывает другой:

<экземпляр объекта Element>.replaceClass
↳(*<имя стилевого класса, удаляемого из привязки>*,
↳*<имя стилевого класса, добавляемого в привязку>*)

В качестве параметров этому методу передаются две строки с именами "отвязываемого" и привязываемого стилевых классов:

```
Ext.select("P").replaceClass("someclass", "otherclass");
```

Метод `radioClass` привязывает указанный стилевой класс к данному элементу Web-страницы и удаляет его из привязки у всех элементов того же уровня вложенности:

<экземпляр объекта Element>.radioClass(*<имя стилевого класса>*)

Параметр данного метода — строка с именем привязываемого стилевого класса:

```
elNavbar.child("UL LI:nodeValue=IMG").radioClass("hovered");
```

Здесь мы привязываем стилевой класс `hovered` к тому пункту вложенного списка, формирующего полосу навигации, который содержит текст "IMG", и удаляем его из привязки у всех остальных пунктов этого же списка.

Метод `hasClass` возвращает `true`, если указанный стилевой класс присутствует в привязке к данному элементу Web-страницы, и `false` в противном случае:

<экземпляр объекта Element>.hasClass(*<имя стилевого класса>*)

Параметр этого метода — имя стилевого класса:

```
if (Ext.get("cnavbar").hasClass("hovered")) {
  var s = "К полосе навигации такой стилевой класс не привязан"
else
  var s = "И что он там делает?..";
```

Получение и задание значений атрибутов стиля

Получение или задание значений атрибута стиля, примененного к какому-либо элементу Web-страницы, выполняется тоже весьма часто. В этом случае нам пригодятся несколько методов объекта `Element`, описанных далее.

Метод `getStyle` возвращает значение указанного атрибута стиля для данного элемента Web-страницы:

<экземпляр объекта Element>.getStyle(*<имя атрибута стиля>*)

Параметр данного метода — строка с именем нужного атрибута стиля. Значение этого атрибута стиля возвращается также в виде строки.

Значение атрибута стиля, возвращаемое методом `getStyle`, получается в результате сложения действия всех стилей, привязанных к элементу Web-страницы явно или неявно. Если указанный атрибут стиля не был задан ни в одном из привязанных к этому элементу стилей, метод `getStyle` вернет значение по умолчанию, устанавливаемое самим Web-обозревателем.

Пример:

```
var marginLeft = elNavbar.getStyle("margin-left");
```

Метод `getColor` служит для получения значений атрибутов стиля, задающих цвет:

<экземпляр объекта Element>.getColor(*<имя атрибута стиля>*,
<значение цвета по умолчанию>[, *<префикс>*])

Первый параметр — имя нужного атрибута стиля в виде строки. Допускается задание только тех атрибутов стилей, что принимают в качестве значения цвет; это могут быть атрибуты стиля, задающие цвет текста, фона, рамки или выделения.

Второй параметр — значение цвета по умолчанию. Оно будет возвращено, если метод не смог получить значение указанного атрибута стиля.

Третий, необязательный, параметр — префикс, который добавляется в начале возвращаемого значения цвета. Если данный параметр не указан, в качестве префикса используется символ решетки `#`.

Метод `getColor` возвращает RGB-код цвета в знакомом нам формате

<префикс>RRGGBB,

где *RR*, *GG* и *BB* — доля, соответственно, красной, зеленой и синей составляющей.

Пример:

```
var color = Ext.getBody().getColor("color", "#FFFFFF");
```

В переменной `color` окажется строка `"#3B4043"` — цвет текста наших Web-страниц. Мы не указали третий параметр метода `getColor`, поэтому возвращенное им значение имеет префикс по умолчанию — `#`.

Пример:

```
var color2 = Ext.getBody().getColor("color", "#FFFFFF", "");
```

В переменной `color2` окажется строка `"3B4043"`. Третьим параметром мы передали пустую строку, поэтому возвращенное методом `getColor` значение не имеет префикса (точнее, имеет — пустую строку).

Метод `setStyle` задает новое значение для указанного атрибута стиля. Он имеет два формата вызова:

```
<экземпляр объекта Element>.setStyle(<имя атрибута стиля>,
<новое значение атрибута стиля>)
```

Первым параметром в этом случае передается имя нужного атрибута стиля, а вторым — его новое значение. Оба эти значения передаются в виде строк.

Пример:

```
Ext.getBody().setStyle("color", "black");
```

Здесь мы задаем для цвета текста всей Web-страницы новое значение — `black`.

Также мы можем передать методу `setStyle` единственный параметр — конфигуратор. Его свойства должны соответствовать атрибутам стиля, а значения свойств станут новыми значениями этих атрибутов стиля:

```
<экземпляр объекта Element>.setStyle(<конфигуратор>)
```

Пример:

```
Ext.getBody().setStyle({ color: "black", background-color: "white" });
```

Здесь мы задаем новые значения цвета сразу для текста и фона Web-страницы.

Метод `setOpacity` задает новое значение полупрозрачности для данного элемента Web-страницы:

```
<экземпляр объекта Element>.setOpacity(<новое значение>)
```

Новое значение полупрозрачности передается в виде числа от 0 (полная прозрачность) до 1 (полная непрозрачность):

```
Ext.get("cheader").setOpacity(0.5);
```

Метод `clearOpacity` делает данный элемент Web-страницы полностью непрозрачным. Он не принимает параметров:

```
Ext.get("cheader").clearOpacity();
```

Метод `clearOpacity` рекомендуется применять, чтобы сделать элемент Web-страницы непрозрачным. Вызов метода `setOpacity` с параметром, равным нулю, может не дать ожидаемых результатов в некоторых Web-обозревателях.

Метод `setDisplayed` задает новое значение атрибута стиля `display`:

```
<экземпляр объекта Element>.setDisplayed(<новое значение>)
```

Этому методу можно передать строку с новым значением атрибута стиля `display`. Также можно передать значение `true` или `false`; первое значение выведет данный элемент Web-страницы на экран, второе скроет его.

Пример:

```
Ext.get("cheader").setDisplayed(false);
Ext.get("cheader").setDisplayed("none");
```

Оба выражения выполняют одно и то же действие — скрывают контейнер `cnavbar`.

Управление видимостью элементов Web-страницы

Еще библиотека Ext Core предлагает нам несколько методов объекта `Element`, позволяющих скрывать элементы Web-страницы и снова их показывать.

Метод `setVisibilityMode` позволяет указать, с помощью какого атрибута стиля будет выполняться скрытие и показ данного элемента Web-страницы: `display` или `visibility`. (Эти атрибуты стиля были описаны в *главе 9*.)

<экземпляр объекта Element>.setVisibilityMode(*<имя атрибута стиля>*)

Единственным параметром этому методу передается строка с именем нужного атрибута стиля. Мы можем использовать значения свойств `Ext.Element.DISPLAY` и `Ext.Element.VISIBILITY`; первое свойство хранит имя атрибута стиля `display`, второе — `visibility`.

Пример:

```
Ext.get("navbar").setVisibilityMode(Ext.Element.DISPLAY);
```

Если метод `setVisibilityMode` для данного элемента Web-страницы ни разу не был вызван, для управления видимостью элемента Web-страницы будет использован атрибут стиля `visibility`.

Метод `setVisible` скрывает или снова выводит данный элемент Web-страницы на экран:

<экземпляр объекта Element>.setVisible(true|false)

Если методу передано значение `true`, данный элемент Web-страницы будет выведен на экран, если же передано значение `false` — он будет скрыт:

```
Ext.get("navbar").setVisible(false);
```

Методы `show` и `hide`, соответственно, показывают и скрывают данный элемент Web-страницы. Они не принимают параметров:

```
Ext.get("navbar").show();
```

Метод `toggle` скрывает данный элемент Web-страницы, если он присутствует на экране, и выводит на экран, если он скрыт. Он не принимает параметров:

```
Ext.get("navbar").toggle();
```

Метод `isVisible` возвращает `true`, если данный элемент Web-страницы видим, и `false`, если невидим. Он не принимает параметров.

Пример:

```
var elNavbar = Ext.get("navbar");
if (elNavbar.isVisible())
  elNavbar.show();
```

Кроме того, управлять видимостью элемента Web-страницы можно методом `setDisplayed`, рассмотренным в предыдущем разделе. Этот метод для скрытия и открытия элемента всегда использует атрибут стиля `display`.

Добавление и удаление элементов Web-страницы

А теперь — высший пилотаж Web-программирования! Программное добавление на Web-страницу новых элементов и программное же их удаление. Для этого применяют методы объекта `DomHelper`.

Метод `append` добавляет новый элемент Web-страницы в качестве потомка в конец указанного элемента:

```
Ext.DomHelper.append(<элемент — будущий родитель>, <конфигуратор>
[, true])
```

Первый параметр — элемент Web-страницы, который станет родителем для вновь создаваемого элемента. Это может быть либо строка с именем элемента, либо представляющий его экземпляр объекта `Element`.

Второй параметр — конфигуратор, задающий тег, содержимое и значения атрибутов тега создаваемого элемента Web-страницы.

Все эти параметры задают в следующих свойствах конфигуратора:

- `tag` — имя тега в виде строки;
- `html` — HTML-код, представляющий содержимое элемента;
- `cls` — стилевой класс, который будет привязан к элементу;
- `children` или `cn` — массив конфигураторов, представляющих потомки данного элемента;
- *<имя атрибута тега>* — значение соответствующего *атрибута тега*.

Метод `append` возвращает экземпляр объекта `HTMLElement`, представляющий созданный элемент Web-страницы. Но если мы передадим в качестве третьего, необязательного, параметра значение `true`, он вернет более удобный в работе и уже привычный для нас экземпляр объекта `Element`.

Листинг 15.2

```
var oConf = { tag: "P",
              html: "Привет от Web-сценария!",
              cls: "someclass",
              id: "newparagraph"
            }
Ext.DomHelper.append("cmain", oConf);
```

В листинге 15.2 мы добавили в конец контейнера `cmain` новый абзац, имеющий следующие параметры:

- тег — <P> (задан свойством tag конфигуратора);
- содержимое — текст "Привет от Web-сценария!" (задано свойством html конфигуратора);
- стилевой класс — someclass (свойство cls);
- имя — newparagraph (свойство id, соответствующее атрибуту тега ID).

Чтобы свежедобавленному абзацу не было скучно, мы добавили сразу же после него маркированный список из трех пунктов (листинг 15.3).

Листинг 15.3

```
var oConf2 = { tag: "UL",
               children: [
                 { tag: "LI", html: "Первый пункт" },
                 { tag: "LI", html: "Второй пункт" },
                 { tag: "LI", html: "Третий пункт" }
               ]
             };
Ext.DomHelper.append("cmain", oConf2);
```

Пункты списка мы задали с помощью свойства children конфигуратора. Этому свойству мы присвоили массив, каждый из элементов которого представляет собой конфигуратор, описывающий параметры одного из пунктов списка.

Метод insertFirst аналогичен только что рассмотренному нами методу append за тем исключением, что вставляет созданный элемент Web-страницы в самое начало указанного элемента:

```
Ext.DomHelper.insertFirst(<элемент — будущий родитель>, <конфигуратор>
[, true])
```

Как видим, этот метод принимает те же параметры, что и метод append.

Пример:

```
Ext.DomHelper.insertFirst("cmain", oConf);
```

Это выражение вставляет абзац, описываемый конфигуратором oConf, в самое начало контейнера cmain.

Методы insertBefore и insertAfter вставляют созданный элемент Web-страницы, соответственно, перед и после данного элемента на том же уровне вложенности:

```
Ext.DomHelper.insertBefore|insertAfter(<элемент — будущий "сосед">,
<конфигуратор>[, true])
```

Первым параметром передается либо строка с именем элемента Web-страницы, который станет "соседом" вновь созданного элемента, либо представляющий его экземпляр объекта Element. Остальные параметры аналогичны соответствующим параметрам метода append.

Пример:

```
var oConf3 = { tag: "HR" }
Ext.DomHelper.insertBefore("navbar", oConf3);
Ext.DomHelper.insertAfter("navbar", oConf3);
```

Мы только что поместили до и после списка `navbar` горизонтальные линии.

Метод `insertHtml` позволяет создать новый элемент Web-страницы на основе строки с его HTML-кодом и поместить его возле указанного элемента или в него в качестве потомка:

```
Ext.DomHelper.insertHtml(<местоположение>,
<элемент — будущий "сосед" или родитель>, <HTML-код>)
```

Первый параметр — строка, указывающая, куда будет помещен созданный методом элемент Web-страницы:

- `"beforeBegin"` — созданный элемент будет помещен перед открывающим тегом указанного элемента и станет его предыдущим "соседом" по уровню вложенности;
- `"afterBegin"` — созданный элемент будет помещен после открывающего тега указанного элемента и станет его первым потомком;
- `"beforeEnd"` — созданный элемент будет помещен перед закрывающим тегом указанного элемента и станет его последним потомком;
- `"afterEnd"` — созданный элемент будет помещен после закрывающего тега указанного элемента и станет его следующим "соседом" по уровню вложенности.

Второй параметр — элемент Web-страницы, который станет "соседом" или родителем для вновь создаваемого элемента. Это должен быть представляющий его экземпляр объекта `HTMLElement` (не `Element`!).

Третий параметр — строка с HTML-кодом, с помощью которого будет создан новый элемент.

Метод `insertHtml` возвращает экземпляр объекта `HTMLElement`, представляющий созданный элемент Web-страницы. К сожалению, указать ему вернуть экземпляр объекта `Element` мы не можем.

Пример:

```
var htelCMain = Ext.getDom("cmain");
Ext.DomHelper.insertHtml("afterBegin", htelCMain,
"<P ID=\"newparagraph\" CLASS=\"someclass\"></P>");
```

Здесь мы добавили в начало контейнера `cmain` новый абзац с именем `newparagraph` и привязанным к нему стилевым классом `someclass`.

Пример:

```
var htelNavbar = Ext.getDom("navbar");
Ext.DomHelper.insertHtml("beforeBegin", htelNavbar, "<HR>");
Ext.DomHelper.insertHtml("afterEnd", htelNavbar, "<HR>");
```

А здесь мы поместили до и после списка, формирующего полосу навигации, горизонтальные линии HTML.

Метод `overwrite` создает новый элемент Web-страницы и помещает его внутрь указанного элемента, заменяя все его предыдущее содержимое:

```
Ext.DomHelper.overwrite(<элемент — будущий родитель>,
<конфигуратор>|<HTML-код>[, true])
```

Первый параметр — элемент Web-страницы, который станет родителем для вновь создаваемого элемента. Это может быть либо строка с именем элемента, либо представляющий его экземпляр объекта `Element`.

Второй параметр — либо конфигуратор, описывающий параметры создаваемого элемента, либо строка с HTML-кодом, на основе которого он будет создан.

Метод `overwrite` возвращает экземпляр объекта `HTMLElement`, представляющий созданный элемент Web-страницы. Но если мы передадим в качестве третьего, необязательного, параметра значение `true`, он вернет экземпляр объекта `Element`.

Пример:

```
var oConf4 = { tag: "P",
               html: "Новое содержимое контейнера."
             }
Ext.DomHelper.overwrite("cmain", oConf4);
```

Здесь мы создаем новый абзац и помещаем его в контейнер `cmain`, полностью заменяя его предыдущее содержимое.

Метод `markup` принимает в качестве единственного параметра конфигуратор и возвращает строку с созданным на его основе HTML-кодом.

Пример:

```
Ext.DomHelper.markup(<HTML-код>)
var s = Ext.DomHelper.markup(oConf4);
```

В переменной `s` окажется строка "`<P>Новое содержимое контейнера.</P>`".

Создавать новые элементы Web-страницы мы можем также с помощью рассмотренных далее методов объекта `Element`. Вероятно, во многих случаях они будут удобнее.

Метод `createChild` создает новый элемент Web-страницы и делает его потомком данного элемента:

```
<экземпляр объекта Element>.createChild(<конфигуратор>
[, <элемент, перед которым будет вставлен созданный элемент>])
```

Первым параметром данному методу передается конфигуратор, описывающий параметры создаваемого элемента Web-страницы.

Если второй параметр опущен, созданный элемент Web-страницы будет помещен в самом конце данного элемента и станет его последним потомком. Если же в качестве его передать какой-либо элемент-потомок в виде экземпляра объекта `Element`, создаваемый элемент будет вставлен перед ним.

Метод `createChild` возвращает экземпляр объекта `Element`, представляющий созданный элемент.

Пример:

```
var elCMain = Ext.get("cmain");
elCMain.createChild(oConf, elCMain.first());
```

Здесь мы вставляем абзац, описываемый конфигуратором `oConf`, в самое начало контейнера `cmain` — перед первым его потомком.

Метод `insertFirst` принимает в качестве параметра конфигуратор, создает на его основе элемент Web-страницы и помещает его в начало данного элемента в качестве его первого потомка:

```
<экземпляр объекта Element>.insertFirst(<конфигуратор>)
Ext.get("cmain").createChild(oConf);
```

Метод `replaceWith` принимает в качестве параметра конфигуратор, создает на его основе элемент Web-страницы и полностью заменяет им данный элемент.

В примере из листинга 15.4 мы удаляем полностью контейнер `cmain` и помещаем на его место другой контейнер, описываемый конфигуратором `oCont5`, с новым содержимым и тем же именем.

Листинг 15.4

```
<экземпляр объекта Element>.replaceWith(<конфигуратор>)
var oConf5 = { tag: "DIV",
               html: "<P>Новый контейнер с новым содержимым.</P>",
               id: "cmain"
             }
Ext.get("cmain").replaceWith(oConf5);
```

Метод `wrap` принимает в качестве параметра конфигуратор, создает на его основе элемент Web-страницы и помещает в него данный элемент, делая его потомком созданного элемента:

```
<экземпляр объекта Element>.wrap([<конфигуратор>])
```

Как видим, при вызове этого метода мы можем не указывать конфигуратор. В таком случае метод `wrap` создаст блочный контейнер на основе тега `<DIV>`.

Пример:

```
Ext.select("UL[id=navbar]").wrap();
```

Здесь мы заключаем список `navbar`, формирующий полосу навигации, в блочный контейнер. Обратим внимание, что мы не передали методу `wrap` никаких параметров — он сам "поймет", что именно блочный контейнер мы хотим создать.

А в следующем примере мы заключаем все абзацы, непосредственно вложенные в контейнер `cmain`, в большие цитаты:

```
Ext.select("DIV[id=cmain] > P").wrap({ tag: "BLOCKQUOTE" });
```

Да, библиотека Ext Core представляет весьма мощные средства для создания новых элементов Web-страницы. К сожалению, удалить ненужные элементы можно только методом `remove` объекта `Element`. Он немедленно удаляет данный элемент Web-страницы со всем его содержимым, не принимает параметров и не возвращает значения.

Пример:

```
Ext.get("cmain").remove();
```

Здесь мы удаляем контейнер `cmain` со всем его содержимым.

Обработка событий

Теперь самое время рассмотреть один ключевой вопрос Web-программирования: события, их возникновение и обработка.

Понятие события и его обработки

Рассматривая примеры Web-сценариев, мы исходили из предположения, что они выполняются при загрузке Web-страницы. Как мы уже знаем из *главы 14*, Web-сценарий исполняется в том месте HTML-кода Web-страницы, в котором присутствует создающий его тег `<SCRIPT>`. При этом неважно, является Web-сценарий внутренним (помещенном прямо в HTML-код) или внешним (хранящимся в отдельном файле Web-сценария).

Однако существует большая группа Web-сценариев, которые выполняются при возникновении определенного события, к которому эти Web-сценарии были привязаны. О них-то и пойдет сейчас разговор.

Событием в терминологии Web-программирования называется некое условие, которое возникает в Web-обозревателе в ответ на действия посетителя или в процессе работы самого Web-обозревателя. Так, щелчок левой кнопкой мыши на элементе Web-страницы приводит к возникновению события "щелчок левой кнопкой мыши", а перемещение курсора мыши над элементом Web-страницы — "перемещение курсора мыши". Нажатие клавиши на клавиатуре приводит к возникновению события "нажатие клавиши", а ошибка в загрузке изображения — "ошибка загрузки".

Существует много разнообразных событий, как говорится, на все случаи жизни. Ежесекундно их возникает десятки, если не сотни.

Так вот, мы можем заставить Web-сценарий выполняться в ответ на возникновение определенного события в определенном элементе Web-страницы. Для этого нужный Web-сценарий особым образом привязывается к данному элементу Web-страницы и событию. Такие Web-сценарии называются *обработчиками событий*.

Что может делать обработчик события? Да что угодно! При наведении курсора мыши он может привязывать к элементу Web-страницы другой стилевой класс, меняя его представление. (Именно такой обработчик события мы создали в *главе 14*.) При щелчке левой кнопкой мыши на элементе Web-страницы — разворачивать или сворачивать блочный контейнер, открывая или скрывая его содержимое. А при из-

менении размеров окна Web-обозревателя — менять размеры блочных контейнеров, чтобы полностью занять ими клиентскую область окна.

Теперь уясним следующие моменты, связанные с обработчиками событий.

- Обработчик события оформляется в виде функции, которая принимает два параметра. Подробнее об этом мы поговорим потом.
- Обработчик события привязывается к конкретному элементу Web-страницы, в котором возникают события, требующие обработки. Так, если нужно обработать событие "щелчок левой кнопкой мыши" в каком-либо абзаце, обработчик привязывается к данному абзацу.
- Обработчик события привязывается к конкретному событию. Так, если мы привязали обработчик к событию "щелчок левой кнопкой мыши", он будет выполняться только при возникновении именно этого события. Другие события, скажем, "двойной щелчок левой кнопкой мыши", он обрабатывать не будет.
- Обработчик события выполняется только при возникновении заданного события в элементе Web-страницы, к которому он привязан. Во время загрузки Web-страницы он не выполняется.
- Мы можем привязать один и тот же обработчик сразу к нескольким элементам Web-страницы и нескольким событиям. Так, один и тот же обработчик может обрабатывать событие "щелчок левой кнопкой мыши" в абзаце и в гиперссылке. Кстати, так часто и делают.

События объекта *Element*

Самые полезные для нас на данный момент события, поддерживаемые объектом `Element` библиотеки Ext Core, представлены в табл. 15.1. Их довольно много, и некоторые из них поддерживаются только определенными элементами Web-страницы.

Таблица 15.1. *События объекта* `Element`

Событие	Описание
`abort`	Возникает при прерывании загрузки изображения, аудио- или видеофайла посетителем. Не всплывает. Действие по умолчанию — отмена загрузки, отменить его невозможно
`blur`	Возникает, когда гиперссылка теряет фокус ввода. Не всплывает. Действие по умолчанию — потеря гиперссылкой фокуса ввода, отменить его невозможно
`click`	Возникает при щелчке левой кнопкой мыши на элементе Web-страницы после событий `mousedown` и `mouseup`. Всплывает. Действие по умолчанию зависит от конкретного элемента, может быть отменено
`dblclick`	Возникает при двойном щелчке левой кнопкой мыши на элементе Web-страницы. Всплывает. Действие по умолчанию зависит от конкретного элемента, может быть отменено
`error`	Возникает при ошибке при загрузке изображения, аудио- или видеофайла. Не всплывает. Действие по умолчанию — вывод сообщения об ошибке, отменить его невозможно

Таблица 15.1 *(окончание)*

Событие	Описание
`focus`	Возникает, когда гиперссылка получает фокус ввода. Не всплывает. Действие по умолчанию — получение гиперссылкой фокуса ввода, отменить его невозможно
`keydown`	Возникает при нажатии любой клавиши. Всплывает. Действие по умолчанию — передача нажатой клавиши элементу, имеющему фокус ввода, может быть отменено
`keypress`	Возникает при нажатии любой алфавитно-цифровой клавиши между событиями `keydown` и `keyup`. Если клавиша удерживается нажатой, возникает постоянно, пока клавиша не будет отпущена. Всплывает. Действие по умолчанию — передача нажатой клавиши элементу, имеющему фокус ввода, может быть отменено
`keyup`	Возникает при отпускании нажатой ранее клавиши. Всплывает. Действие по умолчанию — передача нажатой клавиши элементу, имеющему фокус ввода, может быть отменено
`load`	Возникает после успешной загрузки изображения, аудио- или видеофайла. Не всплывает. Действие по умолчанию отсутствует
`mousedown`	Возникает при нажатии левой кнопки мыши. Всплывает. Действие по умолчанию отсутствует
`mousemove`	Возникает при перемещении курсора мыши над элементом Web-страницы. Всплывает. Действие по умолчанию отсутствует
`mouseout`	Возникает, когда курсор мыши уводится с элемента Web-страницы. Всплывает. Действие по умолчанию отсутствует
`mouseover`	Возникает, когда курсор мыши наводится на элемент Web-страницы. Всплывает. Действие по умолчанию отсутствует
`mouseup`	Возникает при отпускании нажатой ранее левой кнопки мыши. Всплывает. Действие по умолчанию отсутствует

Не будем пока вдаваться в детали, мы к ним еще вернемся. Сейчас рассмотрим более насущный вопрос.

Привязка и удаление обработчиков событий

Метод `on` объекта `Element` выполняет привязку указанной функции к указанному событию данного элемента Web-страницы в качестве обработчика:

<экземпляр объекта Element>.on(*<событие>*, *<функция-обработчик>*)

Первым параметром методу передается строка с названием события, к которому выполняется привязка обработчика. Названия событий приведены в первом столбце табл. 15.1.

Второй параметр — функция, которая станет обработчиком события. Эта функция должна принимать следующие параметры:

- первый — экземпляр объекта `EventObject`, представляющий сведения о событии и позволяющий им управлять (мы рассмотрим этот объект потом);

- второй — экземпляр объекта `HTMLElement`, представляющий элемент Web-страницы, в котором изначально возникло данное событие.

Кроме того, в функцию-обработчик неявно передается еще один параметр — экземпляр объекта `HTMLElement`, представляющий элемент Web-страницы, в котором в данный момент обрабатывается данное событие, — тот самый элемент, к которому привязан этот обработчик. Событие могло возникнуть в нем изначально, а могло всплыть из дочернего элемента; подробнее об этом будет рассказано в следующем разделе. Данный параметр доступен в теле функции-обработчика через переменную `this`.

Пример:

```
Ext.get("navbar").on("mouseover",
  function(e, t) {
    Ext.get(this).addClass("hovered");
  }
);
```

Здесь мы привязываем к списку `navbar` обработчик события `mouseover`. Первый параметр метода `on` определяет название события, которое мы хотим обрабатывать. Второй параметр этого метода содержит объявление функции-обработчика.

В теле функции-обработчика мы обращаемся к переменной `this`, чтобы получить экземпляр объекта `HTMLElement`, представляющий элемент Web-страницы, чье событие мы обрабатываем. Чтобы получить из него соответствующий экземпляр объекта `Element`, мы используем метод `get`. После чего привязываем к полученному экземпляру объекта `Element` стилевой класс `hovered` вызовом метода `addClass`.

Отметим, что наша функция-обработчик принимает два параметра, которые, впрочем, нигде в ее теле не используются. Так что мы можем вообще не указывать их в объявлении функции-обработчика:

```
Ext.get("navbar").on("mouseover",
  function() {
    Ext.get(this).addClass("hovered");
  }
);
```

Мы можем оформить обработчик события в виде функции, имеющей имя, а потом указать это имя в качестве второго параметра метода `on`:

```
function navbarMouseOver() {
  Ext.get(this).addClass("hovered");
}
Ext.get("navbar").on("mouseover", navbarMouseOver);
```

Это полезно, если мы хотим привязать один обработчик сразу к нескольким событиям одного или нескольких элементов Web-страницы.

Метод `removeAllListeners` объекта `Element` удаляет все привязанные к данному элементу Web-страницы обработчики событий. Он не принимает параметров.

Пример:

```
Ext.get("navbar").removeAllListeners();
```

Всплытие и действие по умолчанию

И еще два важных вопроса, без понимания которых невозможно писать эффективные обработчики событий, — всплытие событий и их действия по умолчанию.

Давайте рассмотрим пункты списков, формирующих полосу навигации, и "внешнего", и вложенного. Мы привязали к ним обработчики событий `mouseOver` и `mouseOut` — это выполняет второй Web-сценарий, написанный нами в *главе 14*. Его фрагмент приведен в листинге 15.5.

Листинг 15.5

```
var ceLinks = Ext.select("UL[id=navbar] LI");
ceLinks.on("mouseover", function(e, t) {
  Ext.get(this).addClass("hovered");
});
ceLinks.on("mouseout", function(e, t) {
  Ext.get(this).removeClass("hovered");
});
```

Откроем Web-страницу index.htm в Web-обозревателе и наведем курсор мыши на пункт "HTML" "внешнего" списка. Рамка вокруг него тотчас станет более темной. Уберем курсор — и рамка примет прежний цвет. Обработчики событий `mouseOver` и `mouseOut` работают.

Теперь наведем курсор мыши на один из пунктов вложенного списка, например, "AUDIO". Вокруг этого пункта также появится рамка. Что и неудивительно, ведь к пунктам вложенного списка также привязаны обработчики соответствующих событий.

Но при этом изменит свой цвет и рамка вокруг пункта "HTML" "внешнего" списка! Выходит, данные события возникают не только в пункте "AUDIO" вложенного списка, но и в пункте "HTML" "внешнего" списка, в который он вложен. Почему?

Вот тут мы столкнулись со *всплытием* событий. Событие, возникшее в каком-либо элементе Web-страницы, после этого возникает в его родителе, далее в родителе родителя и т. д., пока оно не достигнет секции тела Web-страницы (тега `<BODY>`). Можно сказать, что событие "всплывает" из элемента в элемент "вверх" по уровню их вложенности друг в друга.

Давайте рассмотрим всплытие событий на примере нашей полосы навигации. Мы наводим курсор мыши на пункт "AUDIO" вложенного списка. В нем (в теге `<LI>`) возникает событие `mouseOver`. Поскольку к данному пункту списка привязан обработчик этого события, Web-обозреватель его выполняет. Далее событие "всплывает" в сам вложенный список (тег `<UL>`). Никаких обработчиков событий мы к нему не привязали, поэтому событие сразу же следует в пункт "HTML" "внешнего" спи-

ска, в котором находится вложенный список. К нему, опять же, привязан обработчик события `mouseOver`, который выполняется Web-обозревателем. Далее событие "всплывает" во "внешний" список, контейнер `cnavbar` и секцию тела Web-страницы, в которой и прекращает свое существование.

Теперь посмотрим на пункт "AUDIO" вложенного списка. В него вложен тег `<CODE>`, в который, в свою очередь, вложена гиперссылка (тег `<A>`), а в нее — уже текст "AUDIO". Когда мы наводим курсор мыши на этот текст, событие `mouseOver` возникает-то в теге `<A>`! Потом оно "всплывает" в тег `<CODE>`, а уже после этого — в тег `<LI>`, формирующий пункт списка "AUDIO", где и обрабатывается.

А сейчас представим, что событие `mouseOver` не всплывает. Тогда, чтобы обработать его в пункте вложенного списка, нам пришлось бы привязывать разные обработчики к тегам `<LI>`, `<CODE>` и `<A>`. Представляете, сколько потребуется кода! Который в этом случае будет еще и заметно сложнее...

Так что всплытие событий — благо для Web-программиста. Благодаря ему мы можем писать обработчики, реагирующие на события сразу в нескольких элементах Web-страницы, — для этого достаточно привязать обработчик к родителю этих элементов. Так, кстати, часто и делают.

Теперь об обработке всплывающих событий. Мы уже знаем, что функция — обработчик события принимает три параметра: два — явно, третий — неявно.

- Первый явный параметр — экземпляр объекта `EventObject`, хранящий сведения о событии. Разговор о нем пока отложим.
- Второй явный параметр — экземпляр объекта `HTMLElement`, представляющий элемент Web-страницы, в котором изначально возникло это событие. Отметим — возникло изначально, т. е. событие гарантированно не всплыло в него из какого-либо дочернего элемента.
- Неявный параметр, доступный в теле функции-обработчика через переменную `this`, — экземпляр объекта `HTMLElement`, представляющий элемент Web-страницы, в котором это событие обрабатывается, т. е. тот элемент, к которому привязан обработчик. Причем событие могло как возникнуть в этом элементе изначально, так и всплыть из дочернего элемента.

Если событие всплыло в данный элемент Web-страницы из дочернего элемента, то оба эти параметра указывают на разные элементы Web-страницы. Помните об этом!

Ради эксперимента давайте изменим код второго Web-сценария, написанного в *главе 14*, так, как показано в листинге 15.6.

Листинг 15.6

```
var ceLinks = Ext.select("UL[id=navbar] LI");
ceLinks.on("mouseover", function(e, t) {
  Ext.get(t).addClass("hovered");
});
```

```
ceLinks.on("mouseout", function(e, t) {
  Ext.get(t).removeClass("hovered");
});
```

Здесь мы в теле функций-обработчиков получаем не элемент, в котором обрабатывается событие (переменная `this`), а элемент, в котором это событие изначально возникло (второй параметр этих функций `t`). Что получится в результате?

Наведем курсор мыши на текст пункта "AUDIO" списка. В теге `<A>` возникнет событие `mouseover`, которое потом всплывет в тег `<CODE>` и далее — в тег `<LI>`. В теге `<LI>` оно и будет обработано. Только стилевой класс `hovered` в результате будет привязан не к тегу `<LI>`, а к тегу `<A>`! А поскольку для этого тега больше никаких параметров рамки не задано (стилевой класс `hovered` задает только цвет рамки — см. *главу 14*), рамка вокруг него не появится. Совсем другой результат!

В табл. 15.1, перечисляющей события объекта `Element`, прямо указано, какие события всплывают, а какие — нет. Да-да, не все события относятся к всплывающим... Но такие "невсплывающие" события обычно очень специфичны и требуют обработки, что называется, на месте.

Многие из всплывающих событий поддерживают возможность прерывания их всплытия на любом уровне. Это реализуется особым методом объекта `EventObject`, представляющем сведения о событии. Мы рассмотрим его чуть позже.

Некоторые события, возникшие в определенных элементах Web-страницы, Web-обозреватель обрабатывает сам, реализуя так называемое *действие по умолчанию*. Так, действие по умолчанию для события `click` в гиперссылке — переход на целевую Web-страницу, а для события `focus` — получение гиперссылкой фокуса ввода.

Обработчик, привязанный к событию, для которого Web-обозреватель выполняет действие по умолчанию, выполняется перед тем, как действие по умолчанию будет выполнено. Это предоставляет возможность отмены действия по умолчанию (не для всех событий) — вызовом особого метода все того же объекта `EventObject`. Мы рассмотрим данный метод очень скоро.

Получение сведений о событии. Объект *EventObject*

Мы уже знаем, что первый параметр функции-обработчика события хранит экземпляр объекта `EventObject`, содержащий сведения о событии и позволяющий им управлять. Давайте рассмотрим некоторые методы этого объекта, которые будут для нас наиболее полезны.

Метод `getCharCode` возвращает код алфавитно-цифрового символа, введенного с клавиатуры, в кодировке Unicode в виде числа. Он не принимает параметров.

Коды алфавитно-цифровых символов можно узнать с помощью утилиты Таблица символов, поставляемой в составе Windows.

Метод `getKey` возвращает код нажатой на клавиатуре клавиши в кодировке Unicode в виде числа. Он не принимает параметров.

Коды клавиш клавиатуры можно найти на Web-странице **http://msdn.microsoft.com/en-us/library/ms927178.aspx**.

Методы `getPageX` и `getPageY` возвращают, соответственно, горизонтальную и вертикальную координаты курсора мыши относительно Web-страницы в виде чисел в пикселах. Они не принимают параметров.

Метод `preventDefault` отменяет действия по умолчанию для события. Он не принимает параметров и не возвращает значения.

Метод `stopPropagation` отменяет дальнейшее всплытие события. Он не принимает параметров и не возвращает значения.

Метод `stopEvent` отменяет действия по умолчанию для события и отменяет его дальнейшее всплытие. Фактически он объединяет действие методов `preventDefault` и `stopPropagation`. Этот метод также не принимает параметров и не возвращает значения.

Мы можем исправить код нашего второго Web-сценария, написанного в *главе 14*, как показано в листинге 15.7, и посмотреть, что получится.

Листинг 15.7

```
var ceLinks = Ext.select("UL[id=navbar] LI");
ceLinks.on("mouseover", function(e, t) {
  Ext.get(this).addClass("hovered");
  e.stopEvent();
});
ceLinks.on("mouseout", function(e, t) {
  Ext.get(this).removeClass("hovered");
  e.stopEvent();
});
```

Объект *CompositeElementLite*

Вернемся в начало этой главы и вспомним, как мы получали доступ к нужному нам элементу Web-страницы.

Мы можем получить доступ к одному элементу Web-страницы:

```
var elCMain = Ext.get("cmain");
```

Или сразу к нескольким:

```
var clContainers = Ext.select("DIV");
```

Мы помним, что метод `select` объекта `Ext` возвращает экземпляр объекта `CompositeElementLite` — коллекцию экземпляров объекта `Element`, представляющих все подходящие под указанный селектор CSS элементы Web-страницы. Настала пора рассмотреть объект `CompositeElementLite` подробнее.

Прежде всего, объект `CompositeElementLite` поддерживает все методы объекта `Element`, предназначенные для управления привязкой стилевых классов, атрибутами

тега и стиля, привязкой обработчиков событий и т. п. Так что мы можем привязывать стилевые классы и обработчики событий сразу к нескольким элементам Web-страницы. (Собственно, мы это уже делали.)

Метод `getCount` возвращает число элементов данной коллекции. Он не принимает параметров:

```
var i = clContainers.getCount();
```

В переменной `i` окажется число элементов в полученной ранее коллекции `clContainers` — 5.

Метод `item` возвращает элемент данной коллекции с указанным индексом в виде экземпляра объекта `Element`:

```
<экземпляр объекта CompositeElementLite>.item(<индекс>)
```

Как видим, этот метод принимает единственный параметр — индекс требуемого элемента коллекции в виде числа.

Пример:

```
var elDiv = clContainers.item(i – 1);
```

В переменной `elDiv` окажется последний элемент коллекции `clContainers`. Поскольку элементы коллекции, как и элементы обычного массива, нумеруются, начиная с нуля, мы передали методу `item` значение, на единицу меньшее, чем число элементов в коллекции.

А в следующем примере мы последовательно извлекаем все элементы коллекции `clContainers` и выполняем над ними какие-то действия:

```
for(var k = 0; k < i; k++) {
  var elDiv = clContainers.item(k);
  // Что-то делаем
}
```

Метод `indexOf` возвращает индекс указанного элемента в данной коллекции в виде числа:

```
<экземпляр объекта CompositeElementLite>.indexOf(<элемент>)
```

Единственным параметром этому методу передается искомый элемент. Им может быть строка с именем элемента, экземпляр объекта `Element` или `HTMLElement`.

Если переданный элемент в коллекции отсутствует, метод `indexOf` возвращает –1.

Пример:

```
var ind = clContainers.indexOf("cnavbar");
```

В переменной `ind` окажется индекс контейнера `cnavbar` в коллекции `clContainers` — 1.

Метод `each` вызывает для каждого элемента данной коллекции указанную функцию:

```
<экземпляр объекта CompositeElementLite>.each(<функция>)
```

Единственным параметром этому методу передается функция, которая будет вызвана для каждого элемента данной коллекции. Она должна принимать следующие параметры:

- элемент коллекции в виде экземпляра объекта `Element`;
- сама эта коллекция в виде экземпляра объекта `CompositeElementLite`;
- индекс элемента коллекции в виде числа.

Кроме того, элемент коллекции доступен в теле этой функции через переменную `this`.

В примере из листинга 15.8 мы привязываем к каждому контейнеру, входящему в коллекцию `clContainers`, стилевой класс `hovered`.

Листинг 15.8

```
clContainers.each(function(el, cl, ind)
  {
    el.addClass("hovered");
  }
);
```

Другой вариант того же Web-сценария иллюстрирует листинг 15.9.

Листинг 15.9

```
clContainers.each(function(el, cl, ind)
  {
    this.addClass("hovered");
  }
);
```

Еще проще написать так:

```
clContainers.addClass("hovered");
```

На этом мы пока закончим с библиотекой Ext Core. В следующих главах мы к ней еще вернемся и рассмотрим другие ее возможности.

Объекты Web-обозревателя

Как видим, библиотека Ext Core позволяет сделать очень многое, написав несколько строчек JavaScript-кода. Если бы мы пользовались для этого исключительно объектами Web-обозревателя, объем кода вырос бы на порядок — не меньше.

Но с помощью Ext Core мы можем сделать не все. Некоторые вещи доступны только через объекты Web-обозревателя.

Один из таких объектов — `HTMLDocument`, представляющий Web-страницу. Единственный его экземпляр, представляющий текущую Web-страницу, доступен через переменную `document`. Это мы уже знаем.

Из всех свойств объекта `HTMLDocument` интерес для нас представляют немногие. Его методы и события нам вряд ли пригодятся.

Свойство `title` хранит текст заголовка Web-страницы (содержимое тега `<TITLE>`) в виде строки. Заголовок, как мы помним из *главы 1*, выводится в строке заголовка окна Web-обозревателя, в котором открыта данная Web-страница.

Пример:

```
var sTitle = document.title;
```

В переменной `sTitle` окажется строка с текстом заголовка Web-страницы.

А в следующем примере мы задаем для Web-страницы новый заголовок:

```
document.title = "Заголовок";
```

Свойство `location` хранит экземпляр объекта `Location`, представляющий интернет-адрес Web-страницы. Нам будет полезно только свойство `href`, хранящее интернет-адрес Web-страницы в виде строки:

```
var sHREF = document.location.href;
```

В переменной `sHREF` окажется строка с интернет-адресом Web-страницы.

Пример:

```
document.location.href = "http://www.w3.org";
```

Здесь мы переходим на Web-страницу **http://www.w3.org**. Да-да, с помощью свойства `href` объекта `Location` мы можем заставить Web-обозреватель открыть другую Web-страницу, присвоив этому свойству ее интернет-адрес!

Объект `Window` представляет окно Web-обозревателя. Единственный экземпляр этого объекта, представляющий текущее окно Web-обозревателя, хранится в переменной `window`. Это мы тоже знаем.

Рассмотрим полезные для нас методы и события объекта `Window`.

Метод `alert` выводит на экран окно-предупреждение с указанным текстом и кнопкой **OK**. Такие окна-предупреждения выводят посетителю сообщение, которое он обязательно должен прочитать:

```
window.alert(<текст, выводимый в окне-предупреждении>)
```

Единственный передаваемый параметр — строка с текстом, который будет выведен в окне-предупреждении:

```
window.alert("Привет от объекта Window!");
```

Метод `confirm` выводит на экран окно-предупреждение с указанным текстом и кнопками **OK** и **Отмена**. Такие окна-предупреждения обычно используются, чтобы запросить у посетителя подтверждение или отмену какого-либо действия:

```
window.confirm(<текст, выводимый в окне-предупреждении>)
```

Единственный передаваемый параметр — строка с текстом, который будет выведен в окне-предупреждении.

Метод `confirm` возвращает `true`, если посетитель нажал кнопку **OK**, и `false`, если он нажал кнопку **Отмена**.

Событие `resize` возникает, когда посетитель изменяет размеры окна Web-обозревателя.

Пример:

```
Ext.fly(window).on(function()
  {
    // Что-то делаем
  }
);
```

Здесь показано, как рекомендуется привязывать обработчик к событию `resize` окна Web-обозревателя (это справедливо и для других событий окна).

Объект `HTMLElement`, как мы уже знаем, представляет элемент Web-страницы. Рассмотрим некоторые его свойства.

Свойство `textContent` хранит текстовое содержимое элемента Web-страницы в виде строки. Если элемент не имеет текстового содержимого, оно хранит значение `null`.

Пример:

```
var htelCHeader = Ext.getDom("cheader");
var s = htelCHeader.textContent;
htelCHeader.textContent = "!" + s + "!";
```

Здесь мы получаем текстовое содержимое контейнера `cheader`, добавляем к нему слева и справа восклицательные знаки и снова помещаем его в контейнер `cheader`.

Свойство `innerHTML` хранит HTML-код содержимого данного элемента Web-страницы в виде строки.

Пример:

```
var htelCHeader = Ext.getDom("cheader");
var s = htelCHeader.textContent;
htelCHeader.innerHTML = "<EM>" + s + "</EM>";
```

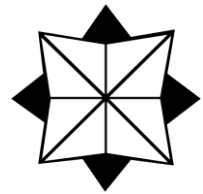
Объектам Web-обозревателя впору посвящать отдельную книгу — настолько это объемная тема. Мы рассмотрели только несколько инструментов, которые предоставляют они и которые не найти в библиотеке Ext Core. Впоследствии мы еще вернемся к объектам Web-обозревателя, но это будет нескоро.

Что дальше?

Эта глава почти полностью была посвящена библиотеке Ext Core — инструменту, значительно упрощающему работу Web-программиста. Мы рассмотрели несколько объектов этой библиотеки и такое количество их методов, что просто голова кругом.

В следующей главе мы применим полученные знания на практике. О, это будет интересно и поучительно!

ГЛАВА 16



Создание интерактивных Web-страниц

В предыдущей главе мы изучили библиотеку Ext Core. От обилия объектов, свойств, методов и событий голова идет кругом... Как же применить все это богатство на практике?

Этому будет целиком посвящена данная глава. Мы научимся создавать Web-страницы, реагирующие (причем с пользой для дела) на действия посетителя, — *интерактивные* Web-страницы.

Давайте сначала составим список того, чему хотим "научить" наши Web-страницы.

- Изменять размеры блочных контейнеров, формирующих дизайн Web-страниц, чтобы они занимали всю клиентскую область окна Web-обозревателя.
- Менять цвет рамки у пунктов полосы навигации при наведении на них курсора мыши. (Собственно, мы уже это сделали в *главе 14*; здесь мы просто разберем, как все работает.)
- Выполнять переход на Web-страницу при щелчке на любом месте соответствующего пункта полосы навигации, не обязательно точно на гиперссылке.
- Скрывать и открывать вложенные списки, формирующие полосу навигации, при щелчке на пункте "внешнего" списка, в который они вложены.
- Выделять пункт полосы навигации, соответствующий открытой в данный момент Web-странице.
- Реализовать скрытие и открытие текста примеров на Web-страницах, посвященных отдельным тегам и атрибутам стиля, при щелчке мышью на заголовке "Пример:".

Довольно много, не так ли? Но библиотека Ext Core поможет нам сделать все с минимальным объемом кода.

Управление размерами блочных контейнеров

И первое, что мы сделаем, — заставим блочные контейнеры на наших Web-страницах изменять свои размеры так, чтобы занимать всю клиентскую область окна Web-обозревателя и при этом не выходить за ее пределы.

Сначала откроем таблицу стилей main.css и найдем в ней стиль переопределения тега <BODY>. Изменим его так, как показано в листинге 16.1.

Листинг 16.1

```
BODY          { color: #3B4043;
                background-color: #F8F8F8;
                font-family: Verdana, Arial, sans-serif;
                margin: 0px;
                overflow: hidden }
```

Мы добавили в этот стиль атрибут `overflow` со значением `hidden`, тем самым убрав у всей Web-страницы полосы прокрутки. Они нам не нужны, более того, появляясь в самый неподходящий момент, они могут нарушить местоположение блочных контейнеров.

Сохраним таблицу стилей. И сразу же откроем файл Web-сценариев main.js. В самом его начале, еще до вызова метода `onReady` объекта `Ext`, вставим код листинга 16.2.

Листинг 16.2

```
function adjustContainers() {
  var clientWidth = Ext.lib.Dom.getViewportWidth();
  var clientHeight = Ext.lib.Dom.getViewportHeight();

  var cNavbarWidth = Ext.get("cnavbar").getWidth();
  var cHeaderHeight = Ext.get("cheader").getHeight();
  var cCopyrightHeight = Ext.get("ccopyright").getHeight();

  Ext.get("cheader").setWidth(clientWidth);
  var cNavbarHeight = clientHeight - cHeaderHeight - cCopyrightHeight -
  30;
  Ext.get("cnavbar").setHeight(cNavbarHeight);
  Ext.get("cmain").setHeight(cNavbarHeight);
  Ext.get("cmain").setWidth(clientWidth - cNavbarWidth - 10);
  Ext.get("ccopyright").setWidth(clientWidth);
}
```

В конце тела функции, которую мы передаем в качестве параметра методу `onReady` объекта `Ext`, вставим два выражения:

```
Ext.fly(window).on("resize", adjustContainers);
adjustContainers();
```

Сохраним файл main.js. Откроем Web-страницу index.htm в Web-обозревателе — и сразу отметим, что блочные контейнеры приняли такие размеры, чтобы занять всю клиентскую область окна Web-обозревателя. Попробуем изменить размеры окна и понаблюдаем, как меняются размеры контейнеров.

Но как все это работает? Сейчас разберемся.

Код, добавленный нами в файл main.js, объявляет функцию `adjustContainers`, которая станет обработчиком события `resize` окна Web-обозревателя. Именно эта функция будет задавать размеры контейнеров. Давайте рассмотрим ее код построчно.

Сначала мы получаем ширину и высоту клиентской области окна Web-обозревателя:

```
var clientWidth = Ext.lib.Dom.getViewportWidth();
var clientHeight = Ext.lib.Dom.getViewportHeight();
```

К данным значениям мы будем обращаться не один раз, так что лучше сохранить их в переменных. Ведь доступ к переменной выполняется значительно быстрее, чем вызов метода.

Затем получаем ширину контейнера `cnavbar`, высоту контейнеров `cheader` и `ccopyright`:

```
var cNavbarWidth = Ext.get("cnavbar").getWidth();
var cHeaderHeight = Ext.get("cheader").getHeight();
var cCopyrightHeight = Ext.get("ccopyright").getHeight();
```

Эти значения также понадобятся нам в дальнейшем.

Далее задаем ширину контейнера `cheader` равной ширине клиентской области окна Web-обозревателя:

```
Ext.get("cheader").setWidth(clientWidth);
```

Вычисляем высоту контейнеров `cnavbar` и `cmain`:

```
var cNavbarHeight = clientHeight — cHeaderHeight — cCopyrightHeight -
30;
```

Для этого вычитаем из высоты клиентской области высоту контейнеров `cheader` и `ccopyright` и дополнительно 30 пикселов — чтобы создать отступ между нижним краем контейнера `ccopyright` и нижним краем клиентской области.

Задаем полученное ранее значение в качестве высоты контейнеров `cnavbar` и `cmain`:

```
Ext.get("cnavbar").setHeight(cNavbarHeight);
Ext.get("cmain").setHeight(cNavbarHeight);
```

Задаем ширину контейнера `cmain`:

```
Ext.get("cmain").setWidth(clientWidth — cNavbarWidth — 10);
```

Ее значение получаем, вычтя из ширины клиентской области ширину контейнера `cnavbar` и еще 10 пикселов — величину внешнего отступа между ними (он задан в именованном стиле `navbar` атрибутом стиля `margin-right`).

Напоследок задаем ширину контейнера `ccopyright` равной ширине клиентской области окна Web-обозревателя:

```
Ext.get("ccopyright").setWidth(clientWidth);
```

На этом выполнение функции `adjustContainers` закончилось.

Теперь рассмотрим два выражения, помещенные в тело функции, являющейся параметром метода `onReady` объекта `Ext`.

Привязываем функцию `adjustContainers` в качестве обработчика к событию `resize` окна Web-обозревателя:

```
Ext.fly(window).on("resize", adjustContainers);
```

И сразу же ее вызываем, чтобы контейнеры приняли правильные размеры сразу после загрузки Web-страницы:

```
adjustContainers();
```

Вот и все. Согласитесь — ничего сложного.

Выделение пункта полосы навигации при наведении на него курсора мыши

Ну, это мы уже сделали. В листинге 16.3 приведен написанный нами в *главе 14* JavaScript-код.

Листинг 16.3

```
var ceLinks = Ext.select("UL[id=navbar] LI");

ceLinks.on("mouseover", function(e, t) {
  Ext.get(this).addClass("hovered");
});
ceLinks.on("mouseout", function(e, t) {
  Ext.get(this).removeClass("hovered");
});
```

Разберем его построчно.

Сначала получаем все пункты списков, формирующих полосу навигации, — и "внешнего", и всех вложенных:

```
var ceLinks = Ext.select("UL[id=navbar] LI");
```

Затем привязываем к событию `mouseover` полученных пунктов функцию-обработчик, которую там же и объявляем:

```
ceLinks.on("mouseover", function(e, t) {
  Ext.get(this).addClass("hovered");
});
```

Этот обработчик сначала получит из переменной `this` экземпляр объекта `HTMLElement`, представляющий пункт списка, к которому, собственно, он и привязан. Вызовом метода `get` он преобразует его в экземпляр объекта `Element` и вызовом метода `addClass` привяжет к нему стилевой класс `hovered` (его определение см. в *главе 14*), который изменит цвет рамки этого пункта.

Также привязываем обработчик к событию `mouseout` полученных пунктов:

```
ceLinks.on("mouseout", function(e, t) {
  Ext.get(this).removeClass("hovered");
});
```

Он уберет стилевой класс `hovered` из привязки к данному пункту списка, и его рамка примет прежний цвет.

Переход на целевую Web-страницу при щелчке на пункте полосы навигации

На очереди — реализация перехода на целевую Web-страницу при щелчке на любом месте в пункте полосы навигации, не обязательно точно на гиперссылке.

Откроем таблицу стилей main.css и найдем в ней стили гиперссылок с селекторами вида `#navbar A<`*`псевдокласс`*`>`. Удалим их и вместо них напишем стиль, приведенный в листинге 16.4.

Листинг 16.4

```
#navbar A:focus,
  #navbar A:hover,
  #navbar A:active,
  #navbar A:visited { color: #576C8C;
                      text-decoration: none }
```

Он задает для гиперссылок в списке `navbar` независимо от их состояния одинаковый цвет текста и отсутствие подчеркивания. Негласный стандарт для гиперссылок полосы навигации требует, чтобы они всегда имели один и тот же вид.

Далее откроем файл Web-сценария main.js и вставим в функцию, которая передается методу `onReady` объекта `Ext` в качестве параметра, код листинга 16.5.

Листинг 16.5

```
ceLinks.on("click", function(e, t) {
  var elA = Ext.get(this).child("A");
  if (elA) {
    var href = elA.getAttribute("HREF");
    e.stopEvent();
    window.location.href = href;
  }
});
```

В листинге 16.5 мы привязываем к событию `click` всех пунктов всех списков, формирующих полосу навигации, функцию-обработчик, которую там же и объявляем. Рассмотрим тело этой функции построчно.

Сначала получаем из переменной `this` экземпляр объекта `HTMLElement`, представляющий элемент Web-страницы, в котором обрабатывается событие, преобразуем его в экземпляр объекта `Element` и сразу же получаем его потомок, созданный тегом `<A>`:

```
var elA = Ext.get(this).child("A");
```

Фактически мы получаем пункт списка, на который щелкнули мышью, и добираемся до находящейся в нем гиперссылки.

Поскольку у нас пока еще не все пункты списков имеют гиперссылки, обязательно проверяем, существует ли вообще эта гиперссылка:

```
if (elA) {
```

Если она существует, метод `child` в предыдущем выражении вернет ссылку на экземпляр объекта `Element`, которое языком JavaScript будет преобразовано в логическое значение `true`. Условие будет выполнено, следовательно, выполнится блок "то" условного выражения.

Если же такой гиперссылки нет, метод `child` вернет значение `null`. JavaScript преобразует его в значение `false`, условие не будет выполнено, и код блока "то" условного выражения будет пропущен.

Первое выражение блока "то" извлекает значение атрибута `HREF` тега `<A>` — интернет-адрес гиперссылки:

```
  var href = elA.getAttribute("HREF");
```

Останавливаем всплытие события и отменяем его действие по умолчанию:

```
  e.stopEvent();
```

И вот почему...

Мы выполняем программный переход на другую Web-страницу, так что "услуги" гиперссылки нам в этом случае не нужны. Поэтому мы отменяем действие по умолчанию события `click` для гиперссылки — переход на целевую Web-страницу.

Событие `click`, возникнув в пункте вложенного списка, продолжит всплытие, пока не окажется в пункте "внешнего" списка, к которому тоже привязан обработчик этого события. Этот обработчик также выполнится и произведет переход на Web-страницу, на которую указывает гиперссылка уже пункта "внешнего" списка, что нам совсем не нужно. Поэтому мы останавливаем дальнейшее всплытие события.

Далее выполняем переход на полученный интернет-адрес:

```
  window.location.href = href;
}
```

На этом блок "то" условного выражения закончен.

Вот и все. Теперь посетители нашего Web-сайта, чтобы перейти на другую Web-страницу, могут щелкать на любом месте пункта полосы навигации, не обязательно точно на находящейся в нем гиперссылке.

Что ж, настало время создать остальные Web-страницы нашего Web-сайта. Точнее, переделать их под новый дизайн.

Откроем папку tags, где хранятся файлы Web-страниц, описывающих различные теги HTML. Переименуем их все, добавив расширение bak. После этого откроем в Блокноте Web-страницу index.htm.

Дальнейшие действия рассмотрим на примере Web-страницы t_audio.htm, описывающей тег <AUDIO>.

1. Копируем открытую Web-страницу index.htm под именем t_audio.htm в папку tags.
2. Открываем старую Web-страницу, которая сейчас хранится в файле t_audio.htm.bak.
3. Копируем HTML-код из секции тела Web-страницы t_audio.htm.bak в контейнер cmain только что созданной Web-страницы t_audio.htm. Код, формирующий гиперссылку на Web-страницу index.htm, можно опустить — эта гиперссылка нам больше не нужна.
4. Исправляем в коде Web-страницы t_audio.htm интернет-адреса, указывающие на файлы других Web-страниц, таблицы стилей и Web-сценария. Здесь все просто: смотрим, в какой папке хранится целевой файл, и указываем в интернет-адресе относительный путь к нему.

   Вот теги <LINK> и <SCRIPT>, указывающие на внешнюю таблицу стилей и файлы Web-сценариев:

   ```
   <LINK REL="stylesheet" HREF="../main.css" TYPE="text/css">
   <SCRIPT SRC="../ext-core.js"></SCRIPT>
   <SCRIPT SRC="../main.js"></SCRIPT>
   ```

   А вот HTML-код, формирующий гиперссылки на Web-страницы index.htm и t_img.htm:

   ```
   <LI><A HREF="../index.htm">HTML</A>
   <LI><CODE><A HREF="t_img.htm">IMG</A></CODE></LI>
   ```

   Интернет-адреса остальных гиперссылок формируются аналогично.
5. Сохраняем готовую Web-страницу t_audio.htm.

Остается проделать все это с остальными Web-страницами, описывающими теги, что мы к этому времени создали.

Создадим также Web-страницы с описанием технологии CSS, примеров и сведениями о разработчиках. Они будут храниться в файлах css_index.htm, samples_index.htm и about.htm.

Во втором и третьем пункте "внешнего" списка, формирующего полосу навигации, создадим вложенные списки по аналогии с тем, что мы уже создали в первом его пункте. Пусть они будут содержать по два-три пункта, потом мы добавим остальные.

Также создадим хотя бы по одной Web-странице, описывающей какой-нибудь атрибут стиля CSS и пример. Сформируем гиперссылки на эти Web-страницы на основе соответствующих пунктов только что созданных вложенных списков.

Конечно, наш Web-сайт все еще неполон, но для его отладки созданных Web-страниц вполне хватит. Потом, закончив с программированием, мы доделаем его до конца.

Скрытие и открытие вложенных списков

Вложенные списки в полосе навигации чрезвычайно громоздки. Как бы сделать так, чтобы все они были скрыты и появлялись только при щелчке на пункте "внешнего" списка, в который они вложены?

Легко!

- Изначально все вложенные списки у нас будут скрыты.
- При открытии Web-страницы, содержащей один из разделов Web-сайта ("HTML", "CSS" или "Примеры"), будет открываться тот вложенный список, что вложен в соответствующий пункт "внешнего" списка.
- При открытии Web-страницы, описывающей тег, атрибут стиля или пример, будет открываться вложенный список, содержащий указывающий на эту Web-страницу пункт.
- В данный момент может быть открыт только один вложенный список — остальные будут скрыты.
- Для скрытия и раскрытия вложенного списка мы будем менять у него значение атрибута стиля `display` (см. *главу 9*) с помощью методов объекта `Element`, управляющих видимостью элемента Web-страницы.

Откроем файл Web-сценария main.js и запишем где-либо в его начале, еще до вызова метода `onReady` объекта `Ext`, объявление функции, приведенное в листинге 16.6.

Листинг 16.6

```
function showInnerList(iIndex) {
  var elNavbar = Ext.get("navbar");
  var ceInnerLists = elNavbar.select("UL");
  ceInnerLists.setDisplayed(false);
  if (iIndex) {
    var sSelector = "UL:nth(" + iIndex + ")";
    elNavbar.child(sSelector).setDisplayed(true);
  }
}
```

Данная функция с именем `showInnerList` будет скрывать все вложенные списки и открывать только тот из них, порядковый номер которого передан ей в качестве единственного параметра. Если параметр опущен, но никакой вложенный список открыт не будет.

Рассмотрим код этой функции построчно.

Сначала получаем "внешний" список, формирующий полосу навигации:

```
var elNavbar = Ext.get("navbar");
```

Затем получаем все вложенные в него списки:

```
var ceInnerLists = elNavbar.select("UL");
```

Далее скрываем все вложенные списки, для чего используем метод `setDisplayed` — так проще:

```
ceInnerLists.setDisplayed(false);
```

Проверяем, был ли функции `showInnerList` передан параметр:

```
if (iIndex) {
```

Если он был передан, переменная `iIndex` будет содержать число, которое преобразуется в значение `true`, и условие выполнится. В противном случае переменная `iIndex` получит значение `null`, которое будет преобразовано в `false`, и условие не выполнится.

Если параметр функции `showInnerList` был передан, выполняется следующий код.

Формируем строку с селектором CSS, который будет выбирать вложенный список, чей порядковый номер был передан с параметром:

```
  var sSelector = "UL:nth(" + iIndex + ")";
```

Выбираем вложенный список с заданным номером и открываем его:

```
  elNavbar.child(sSelector).setDisplayed(true);
}
```

На этом выполнение функции `showInnerList` завершится.

Теперь вставим в конец тела функции, которая передается в качестве параметра методу `onReady` объекта `Ext`, такое выражение:

```
showInnerList(outerIndex);
```

Здесь мы вызываем функцию `showInnerList`, передавая ей в качестве параметра значение переменной `outerIndex`. Эта переменная будет хранить номер вложенного списка, который требуется открыть.

Теперь откроем Web-страницу index.htm и в секцию ее заголовка (в теге `<HEAD>`) вставим такой код:

```
<SCRIPT>
  outerIndex = 1;
</SCRIPT>
```

Мы присваиваем переменной `outerIndex` число 1 — номер вложенного списка, который должен быть открыт при открытии Web-страницы index.htm (это список раздела "HTML"). Когда будут выполняться Web-сценарии, хранящиеся в файле

main.js, в том числе и вызов функции `showInnerList`, значение этой переменной будет передано данной функции в качестве параметра.

Здесь мы немного нарушили требования концепции Web 2.0, предписывающие хранить поведение Web-страницы отдельно от ее содержимого. Но в данном случае это оправдано, т. к. этот Web-сценарий у разных Web-страниц нашего Web-сайта будет различаться, и создавать ради него для каждой Web-страницы "персональный" файл Web-сценария слишком расточительно.

Такой же Web-сценарий мы вставим в секцию заголовка Web-страниц, описывающих теги HTML. И не забываем сохранять исправленные Web-страницы.

В секцию заголовка Web-страницы css_index.htm и Web-страниц, описывающих атрибуты стиля CSS, мы вставим аналогичный код:

```
<SCRIPT>
  outerIndex = 2;
</SCRIPT>
```

Он укажет, что при открытии данных Web-страниц должен быть открыт второй по счету вложенный список — раздела "CSS".

В секцию заголовка Web-страницы samples_index.htm и Web-страниц, содержащих примеры, мы вставим код... сами догадайтесь, какой. (Подсказка: он должен раскрыть третий вложенный список.)

А вот в секцию заголовка Web-страницы about.htm вставим такой код:

```
<SCRIPT>
  outerIndex = null;
</SCRIPT>
```

Поскольку четвертый пункт "внешнего" списка не содержит вложенного списка, открывать там ничего не нужно — следует только скрыть уже открытые вложенные списки.

Сохраним все исправленные Web-страницы и опробуем их в деле. Неплохо получилось, правда?

Выделение пункта полосы навигации, соответствующего открытой в данный момент Web-странице

Что ж, скрытие и открытие вложенных списков, формирующих полосу навигации, в "содружестве" с библиотекой Ext Core реализуется весьма просто. Но работа над полосой навигации еще не закончена.

Хорошим тоном сейчас считается как-то выделять пункт полосы навигации, соответствующий открытой в данный момент Web-страницы. Выделяют его обычно рамкой или "инверсными" цветами (в качестве цвета текста используется цвет фона, а в качестве цвета фона — цвет текста). Давайте и мы сделаем нечто подобное.

- Пункт вложенного списка мы будем выделять "инверсными" цветами.
- Пункт "внешнего" списка, не имеющий вложенного списка, мы также будем выделять "инверсными" цветами.
- Пункт "внешнего" списка, имеющий вложенный список, мы никак выделять не будем — его выделит открытый вложенный список.

Заодно давайте объединим установку выделения на пункт полосы навигации со скрытием и открытием вложенных списков.

Откроем таблицу стилей main.css и добавим в нее стили из листинга 16.7.

Листинг 16.7

```
.selected,
  #navbar .selected A:link,
  #navbar .selected A:focus,
  #navbar .selected A:hover,
  #navbar .selected A:active,
  #navbar .selected A:visited { color: #F8F8F8;
                                background-color: #576C8C }
```

Здесь мы определяем стилевой класс `selected`, который будем привязывать к выделяемому пункту списка, и четыре стиля, задающие "инверсные" цвета для текста, которые будут применяться к гиперссылке, находящейся в выделенном пункте списка, в зависимости от ее состояния.

Сохраним таблицу стилей. И откроем файл Web-сценария main.js. Удалим из него объявление функции `showInnerList`, созданное ранее, и вставим на его место объявление функции, приведенной в листинге 16.8.

Листинг 16.8

```
function selectItem(iIndex, sText) {
  var elNavbar = Ext.get("navbar");
  elNavbar.select("LI").removeClass("selected");
  var ceInnerLists = elNavbar.select("UL");
  ceInnerLists.setDisplayed(false);
  var sSelector = "> LI:nth(" + iIndex + ")";
  var elOuterItem = elNavbar.child(sSelector);
  var elInnerList = elOuterItem.child("UL");
  if (elInnerList) {
    elInnerList.setDisplayed(true);
    if (sText) {
      sSelector = "LI:has(:nodeValue(" + sText + "))";
      elOuterItem.child(sSelector).addClass("selected");
    }
  } else
    elOuterItem.addClass("selected");
}
```

Эта функция с именем `selectItem` будет выполнять несколько действий:

- Если первым параметром передан порядковый номер пункта "внешнего" списка, который содержит вложенный список, она откроет данный список и скроет все остальные вложенные списки.
- Если первым параметром передан порядковый номер пункта "внешнего" списка, который не содержит вложенный список, она выделит данный пункт "внешнего" списка и, опять же, скроет все остальные вложенные списки.
- Если вторым параметром передан текст пункта вложенного списка, она дополнительно выделит этот пункт, сняв выделение со всех остальных пунктов вложенных списков. Если второй параметр опущен, она ничего выделять не будет.

Рассмотрим ее код построчно.

Получаем "внешний" список, формирующий полосу навигации:

```
var elNavbar = Ext.get("navbar");
```

Убираем из привязки ко всем пунктам всех списков, формирующих полосу навигации, — и "внешнего", и вложенных — стилевой класс `selected`:

```
elNavbar.select("LI").removeClass("selected");
```

Так мы снимем выделение со всех пунктов всех списков.

Сворачиваем все вложенные списки:

```
var ceInnerLists = elNavbar.select("UL");
ceInnerLists.setDisplayed(false);
```

Эти три выражения перекочевали из функции `showInnerList` без изменений.

Формируем строку, содержащую селектор CSS, который выбирает пункт "внешнего" списка с переданным функции `selectItem` в качестве первого параметра порядковым номером, и получаем этот пункт:

```
var sSelector = "> LI:nth(" + iIndex + ")";
var elOuterItem = elNavbar.child(sSelector);
```

Получаем вложенный список, присутствующий в этом пункте, разумеется, если он там есть:

```
var elInnerList = elOuterItem.child("UL");
```

Если вложенный список есть, метод `child` вернет экземпляр объекта `Element`, в противном случае — значение `null`.

Проверяем, что мы получили в результате вызова метода `child` в предыдущем выражении:

```
if (elInnerList) {
```

Если этот метод вернул экземпляр объекта `Element`, последний будет преобразован в значение `true`, и условие выполнится. Если же он вернул значение `null`, оно будет преобразовано в `false`, и условие не выполнится.

Если вложенный список существует, открываем его:

```
  elInnerList.setDisplayed(true);
```

Проверяем, был ли функции `selectItem` передан второй параметр — текст пункта вложенного списка, который следует выделить:

```
  if (sText) {
```

Если он был передан, формируем строку с селектором CSS, выбирающим пункт вложенного списка, потомок которого содержит текст, переданный этим самым вторым параметром:

```
    sSelector = "LI:has(:nodeValue(" + sText + "))";
```

Получаем пункт вложенного списка, удовлетворяющий сформированному ранее селектору, и сразу же привязываем к нему стилевой класс `selected`:

```
    elOuterItem.child(sSelector).addClass("selected");
  }
```

Если полученный ранее пункт "внешнего" списка с указанным порядковым номером не содержит вложенного списка, привязываем к этому пункту стилевой класс `selected`:

```
} else
  elOuterItem.addClass("selected");
```

На этом выполнение функции завершается.

В теле функции, передаваемой в качестве параметра методу `onReady` объекта `Ext`, ранее мы добавили вызов функции `showInnerList`. Удалим его и вместо него вставим такой код:

```
selectItem(outerIndex, innerText);
```

Здесь мы вызываем функцию `selectItem`, передавая ей в качестве параметров значения переменных `outerIndex` и `innerText`. Первая переменная будет хранить номер вложенного списка, который требуется открыть, а вторая — текст пункта вложенного списка, который нужно выделить.

Откроем Web-страницу index.htm, найдем код, вставленный ранее в ее секцию заголовка, и дополним его таким образом:

```
<SCRIPT>
  outerIndex = 1;
  innerText = null;
</SCRIPT>
```

Мы добавили выражение, присваивающее переменной `innerText` значение `null`. При выполнении Web-сценариев, хранящихся в файле main.js, в том числе и вызове функции `selectItem`, значения обеих этих переменных будут переданы данной функции в качестве параметров. В результате откроется первый вложенный список, и ни один его пункт не будет выделен.

Откроем Web-страницу t_audio.htm и дополним вставленный ранее код в ее секцию заголовка так:

```
<SCRIPT>
  outerIndex = 1;
  innerText = "AUDIO";
</SCRIPT>
```

В результате будет открыт первый вложенный список и в нем выделен пункт "AUDIO".

Аналогично заменим вставленный ранее код во всех остальных Web-страницах нашего Web-сайта и проверим их в действии.

Скрытие и открытие текста примеров

На Web-страницах, описывающих теги HTML и атрибуты стиля CSS, мы поместили текст примеров применения того или иного тега или атрибута стиля. Часто его делают скрывающимся и открывающимся в ответ на щелчок мышью — так можно сделать Web-страницы визуально менее громоздкими.

Давайте и мы реализуем нечто подобное на своих Web-страничках. Хотя бы в самом простом варианте.

- Текст примера мы поместим в блочный контейнер, к которому привяжем стилевой класс `sample`. Этот стилевой класс будет служить двум целям: во-первых, он обозначит данный контейнер как "вместилище" для примера, во-вторых, он задаст для контейнера особое представление, чтобы его выделить среди остального содержимого Web-страницы.
- Особое представление для контейнера с текстом примера будет включать внутренние отступы и рамку.
- Все содержимое контейнера изначально будет скрыто, за исключением первого его потомка (у нас — заголовок "Пример:" шестого уровня).
- При щелчке мышью на первом потомке контейнер (т. е. остальное его содержимое) будет открываться или снова скрываться.
- При наведении курсора мыши на первый потомок контейнера будет меняться цвет его рамки. Кроме того, мы зададим для него особое представление — форму курсора в виде "указующего перста". Так мы дадим посетителю понять, что данный элемент Web-страницы реагирует на щелчки мышью.
- На Web-странице могут быть несколько контейнеров с текстом примеров. Это следует учесть при написании Web-сценария.

Откроем таблицу стилей main.css и добавим в нее такие стили:

```
.sample       { padding: 5px;
                border: thin dotted #B1BEC6 }
.sample > :first-child { margin: 0px 0px;
                         cursor: pointer }
```

Первый стиль — это стилевой класс `sample`, помечающий контейнер как "вместилище" для примера. Он задает для контейнера параметры внутренних отступов и рамки.

Второй стиль задает для первого непосредственного потомка данного контейнера нулевые внешние отступы (чтобы убрать ненужное свободное пространство выше и ниже его) и форму курсора в виде "указующего перста".

Теперь откроем файл Web-сценария main.js и поместим где-либо в его начале, перед вызовом метода `onReady` объекта `Ext`, код листинга 16.9.

Листинг 16.9

```
function showHideSample(e, t) {
  var elDiv = Ext.fly(t).parent(".sample");
  var ceSampleText = elDiv.select("*:not(:first-child)");
  ceSampleText.setVisibilityMode(Ext.Element.DISPLAY);
  ceSampleText.toggle();
}

function prepareSamples() {
  var ceSamples = Ext.select(".sample");
  ceSamples.each(function(el, cl, ind){
    var elH6 = el.child(":first");
    elH6.on("click", showHideSample);
    elH6.on("mouseover", function(e, t) {
      Ext.get(this).parent("DIV").addClass("hovered");
    });
    elH6.on("mouseout", function(e, t) {
      Ext.get(this).parent("DIV").removeClass("hovered");
    });
    var ceSampleText = el.select("*:not(:first-child)");
    ceSampleText.setDisplayed(false);
  });
}
```

Мы объявили функции `showHideSample` и `prepareSamples`. Первая будет обрабатывать событие `click` в первом потомке контейнера с текстом примера и в ответ на это событие скрывать или открывать данный контейнер. Вторая будет выполнять подготовительные действия: при открытии Web-страницы скрывать контейнеры с текстом примеров и привязывать к ним обработчики событий.

Рассмотрим код этих функций построчно. И начнем с функции `prepareSamples`.

Сначала получаем все контейнеры с привязанным стилевым классом `sample`, т. е. содержащие текст примеров:

```
var ceSamples = Ext.select(".sample");
```

Затем для каждого полученного контейнера выполняем описанные далее манипуляции:

```
ceSamples.each(function(el, cl, ind){
```

Получаем первый потомок контейнера:

```
var elH6 = el.child(":first");
```

Привязываем к нему в качестве обработчика события `click` функцию `showHideSample`:

```
elH6.on("click", showHideSample);
```

Привязываем к нему функцию — обработчик события `mouseover`, которую там же и объявляем:

```
elH6.on("mouseover", function(e, t) {
  Ext.fly(this).parent("DIV").addClass("hovered");
});
```

Эта функция получит родитель элемента Web-страницы, в котором возникло данное событие, т. е. контейнер с текстом примера, и привяжет к нему стилевой класс `hovered`.

Не забываем привязать функцию-обработчик события `mouseout`, которая будет убирать из привязки к контейнеру стилевой класс `hovered`:

```
elH6.on("mouseout", function(e, t) {
  Ext.fly(this).parent("DIV").removeClass("hovered");
});
```

Получаем все остальные элементы-потомки контейнера (не являющиеся первым потомком):

```
var ceSampleText = el.select("*:not(:first-child)");
```

И скрываем их:

```
  ceSampleText.setDisplayed(false);
});
```

На очереди — функция `showHideSample`.

Получаем родитель элемента Web-страницы, в котором возникло событие:

```
var elDiv = Ext.fly(t).parent(".sample");
```

Поскольку этот обработчик привязан к первому потомку контейнера с текстом примера, здесь мы получим сам этот контейнер.

Получаем все элементы-потомки контейнера, не являющиеся первым потомком:

```
var ceSampleText = elDiv.select("*:not(:first-child)");
```

Указываем, что для управления их видимостью будет использован атрибут стиля `display`, и изменяем их видимость (открываем, если они скрыты, и скрываем, если они открыты):

```
ceSampleText.setVisibilityMode(Ext.Element.DISPLAY);
ceSampleText.toggle();
```

Использовать здесь метод `toggle` проще, чем `setDisplayed` — нам не придется проверять, открыты данные элементы Web-страницы или нет. Правда, перед этим по-

требуется указать, что видимостью элемента будет управлять атрибут стиля `display`, но это мелочи.

Еще нам нужно поставить вызов функции `prepareSamples` в самый конец функции, передаваемой методу `onReady` объекта `Ext`:

```
prepareSamples();
```

Теперь откроем любую Web-страницу, содержащую описание тега HTML или атрибута стиля CSS, и поместим текст примера в блочный контейнер с привязанным стилевым классом `sample`. Листинг 16.10 иллюстрирует HTML-код Web-страницы t_audio.htm.

Листинг 16.10

```
<DIV CLASS="sample">
  <H6>Пример:</H6>
  <PRE>&lt;AUDIO SRC=&quot;sound.wav&quot;
  ↳CONTROLS&gt;&lt;/AUDIO&gt;</PRE>
  <H6>Результат:</H6>
  <AUDIO SRC="sound.wav" CONTROLS></AUDIO>
</DIV>
```

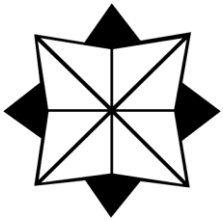
Внесем исправления во все аналогичные Web-страницы и проверим их в действии.

Что дальше?

В этой главе мы вплотную занимались практическим Web-программированием. Мы создали на наших Web-страницах настоящую полосу навигации с подсветкой пункта, на который наведен курсор мыши, с выделением пункта, соответствующего открытой в данный момент Web-странице, с обработкой щелчков на всем пространстве пунктов. Мы заставили блочные контейнеры изменять свои размеры так, чтобы занимать все свободное пространство в клиентской области окна Web-обозревателя. И, наконец, мы создали скрывающиеся и открывающиеся контейнеры с текстом примеров.

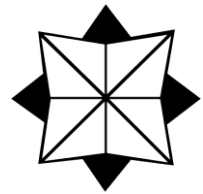
Часть IV будет посвящена самым современным подходам в Web-дизайне и Web-программировании: подгружаемому и генерируемому содержимому и семантической разметке. Уже в следующей главе мы научимся подгружать часть содержимого Web-страницы из сторонних файлов программно.

А библиотека Ext Core нам в этом поможет.



ЧАСТЬ IV

Подгружаемое и генерируемое содержимое. Семантическая разметка



ГЛАВА 17

Подгружаемое содержимое

В предыдущей части мы начали заниматься Web-программированием: изучили язык JavaScript, правила написания Web-сценариев и библиотеку Ext Core и создали поведение для наших Web-страниц, причем весьма развитое. Теперь наш Web-сайт выглядит вполне профессионально.

Скрывающиеся и раскрывающиеся элементы Web-страницы, блочные контейнеры, автоматически выстраивающиеся на Web-странице, полоса навигации с "горячими" пунктами — все это, конечно, впечатляет. Но уже никого особенно не удивляет. В Сети существует множество Web-сайтов, щеголяющих подобными "штуковинами". Все это — уже давно пройденный этап в развитии Web-дизайна.

Нужно что-то новое. Еще более впечатляющее. Еще более интерактивное. И обязательно несущее какие-то выгоды и пользователю, и разработчику — иначе грош ему цена.

Теперь мы займемся технологиями, находящимися на переднем крае Web-дизайна. Мы реализуем на наших Web-страницах, во-первых, подгружаемое содержимое, во-вторых, генерируемое содержимое, в-третьих, семантическую разметку данных. И от всего получим практическую пользу.

Данная глава целиком и полностью будет посвящена подгружаемому содержимому — самой простой из перечисленных технологий.

Монолитные и блочные Web-страницы

Об этом уже упоминалось в *главе 10*, посвященной блочным контейнерам и контейнерному Web-дизайну. Настала пора рассмотреть вопрос подробнее.

Давайте откроем любую нашу Web-страницу, скажем, index.htm, в Web-обозревателе. Что мы увидим? Набор блочных контейнеров, содержащих заголовок Web-сайта, полосу навигации, основное содержимое и сведения об авторских правах. Все эти контейнеры составляют неотъемлемую часть содержимого Web-страницы и определяются в ее HTML-коде.

Теперь посмотрим на HTML-код этой Web-страницы, для чего откроем ее в Блокноте. Да, ее HTML-код велик... Немало места отведено коду, создающему основное содержимое Web-страницы: там и большая цитата, и список, и даже таблица.

Не меньше (если не больше) места занимает код, создающий полосу навигации, со всеми ее списками, вложенными друг в друга, пунктами этих списков и гиперссылками. Код заголовка Web-сайта, сведений об авторских правах и служебный код (создающий сами контейнеры, секции Web-страницы и пр.) хоть и относительно невелик по объему, но тоже там присутствует.

Отсюда следует первый вывод: HTML-код, хранящийся в файле index.htm, определяет все содержимое данной Web-страницы без исключения. Такие Web-страницы на жаргоне Web-дизайнеров называются *монолитными*. Все Web-страницы, созданные нами на данный момент, — монолитные.

Теперь откроем другую Web-страницу, например, css_index.htm. Здесь почти то же самое: заголовок Web-сайта, полоса навигации и сведения об авторских правах. Только основное содержимое другое.

Если мы начнем поочередно открывать другие Web-страницы, то увидим на них аналогичную картину. Заголовок Web-сайта, полоса навигации и сведения об авторских правах остаются без изменений, отличается только основное содержимое.

Выходит, бóльшая часть содержимого наших монолитных Web-страниц неизменна. Настолько неизменна, что мы можем копировать соответствующий HTML-код из одной Web-страницы в другую без всякой правки или пересохранять Web-страницу под другим именем, чтобы создать новую. (В *главе 16*, кстати, мы так и поступили.)

Второй вывод: монолитные Web-страницы, принадлежащие одному Web-сайту, отличаются друг от друга только небольшим фрагментом своего содержимого. Остальной код на всех этих Web-страницах один и тот же.

Когда мы открываем монолитную Web-страницу в Web-обозревателе, она загружается по сети целиком, что занимает некоторое время. Если мы щелкнем на гиперссылке, чтобы перейти на другую Web-страницу, она также будет загружена целиком, на что также потребуется время. При этом львиная доля времени потратится на то, чтобы загрузить те же самые заголовок Web-сайта, полосу навигации и сведения об авторских правах — элементы, которые остаются неизменными на всех Web-страницах. Это первый недостаток монолитных Web-страниц.

Второй недостаток затрагивает уже Web-дизайнеров. Если нам потребуется внести изменения в какой-либо из повторяющихся на всех Web-страницах элементов, например, добавить новый пункт в полосу навигации, то придется вносить соответствующие изменения в HTML-код всех Web-страниц данного Web-сайта. Работа долгая, тяжелая, вдобавок, выполняя ее, нетрудно сделать ошибку...

Выход их этого положения — задействовать подгружаемое содержимое. О нем уже упоминалось в *главе 1*. Вкратце: отдельные части содержимого Web-страниц средствами Web-обозревателя загружаются из других файлов и выводятся в указанное место Web-страницы, а именно в указанный блочный контейнер.

Исходя из этого, набросаем план действий.

- Создаем Web-страницу, содержащую только элементы, остающиеся неизменными на всех Web-страницах нашего Web-сайта, и пустой контейнер, в который

будет выводиться основное содержимое. Назовем эту Web-страницу *базовой*. Базовую Web-страницу сделаем главной Web-страницей Web-сайта.

- Сохраняем основное содержимое всех Web-страниц в отдельных файлах, которые будут хранить только HTML-код, создающий основное содержимое.
- При открытии базовой Web-страницы (мы использовали ее в качестве главной) загружаем основное содержимое, которое должно присутствовать в ней изначально, из файла, где оно хранится, и выводим его в предназначенный для этого контейнер.
- При щелчке на гиперссылке загружаем основное содержимое целевой Web-страницы из соответствующего файла и также выводим его в контейнер.

Достоинств у такого подхода два. Во-первых, файлы с основным содержимым Web-страниц получатся существенно меньше за счет того, что они хранят только основное содержимое и не включают HTML-код, создающий элементы, повторяющиеся от Web-страницы к Web-странице. Во-вторых, если нужно внести исправления в какую-либо Web-страницу, править понадобится всего один файл: либо файл с базовой Web-страницей, либо файл с основным содержимым.

Осталось только сказать, что Web-страницы, "собирающие" свое содержимое из фрагментов, хранящихся в отдельных файлах, неофициально называются *блочными*.

Но как нам загрузить фрагмент содержимого Web-страницы, хранящийся в отдельном файле? О, современные Web-обозреватели предоставляют все средства для этого. А библиотека Ext Core позволяет задействовать данные средства, написав минимум JavaScript-кода.

Подгрузка содержимого Web-страниц

Для подгрузки фрагмента содержимого Web-страницы из стороннего файла и вывода его в указанный элемент Web-страницы библиотека Ext Core предлагает метод `load` объекта `Element`:

```
<экземпляр объекта Element>.load(<интернет-адрес>|<конфигуратор>)
```

Единственный параметр метода `load` задает либо интернет-адрес файла, в котором хранится предназначенный для загрузки фрагмент содержимого Web-страницы, либо конфигуратор, задающий параметры для загрузки данного файла.

Вот свойства этого конфигуратора и соответствующие им параметры:

- `url` — интернет-адрес загружаемого файла в виде строки. Это единственный обязательный параметр.
- `success` — функция, которая будет вызвана, если загрузка файла завершится успешно.
- `failure` — функция, которая будет вызвана, если при загрузке файла возникнет ошибка.
- `callback` — функция, которая будет вызвана после окончания загрузки файла, независимо от того, успешно загрузится файл или нет.

- `timeout` — промежуток времени *(тайм-аут)*, в течение которого Web-обозреватель будет ожидать окончания загрузки файла. Если файл за этот промежуток времени не загрузится, библиотека Ext Core будет считать, что возникла ошибка загрузки. Тайм-аут задается в виде числа в миллисекундах; значение по умолчанию — 30 000 мс (30 с).

Функции, указанные в параметрах `success` и `failure`, не принимают параметров. Функция, указанная в параметре `callback`, должна принимать два параметра: конфигуратор, переданный методу `load` в качестве параметра, и логическое значение, равное `true` в случае удачной загрузки файла и `false` при возникновении ошибки.

Метод `load` полностью заменяет все содержимое элемента Web-страницы, у которого он был вызван. Добавить загруженное содержимое к уже имеющемуся с его помощью мы не сможем.

А теперь важный момент! Метод `load` при вызове отправляет Web-серверу запрос на получение указанного файла и сразу же завершает свою работу (начинает выполняться код, следующий за вызовом этого метода). Он не ждет, когда файл будет загружен и выведен на экран. Программисты называют подобные методы *асинхронными*.

Но что делать, если нам понадобится выполнять какие-либо манипуляции с содержимым загруженного файла? Посмотрим на список свойств конфигуратора, передаваемый методу `load` в качестве параметра. Там присутствует свойство `success`, которому присваивается функция, вызываемая после успешной загрузки файла. Мы можем поместить код, манипулирующий загруженным содержимым, в тело этой функции. (Еще можно использовать функцию, присваиваемую свойству `callback` конфигуратора, только в этом случае придется проверять, успешно ли завершилась загрузка.)

> ***Внимание!***
>
> Средства Web-обозревателя для подгрузки содержимого из стороннего файла, которые задействует библиотека Ext Core, работают только в том случае, если все эти файлы, а также файл базовой Web-страницы загружаются с Web-сервера. При загрузке файлов без участия Web-сервера, прямо с файловой системы локального компьютера, они не работают — это ограничения самого Web-обозревателя.

Еще в *главе 1* мы установили и опробовали в работе Web-сервер Microsoft Internet Information Services. Долгое время мы им не пользовались, но теперь настал момент, когда его "услуги" нам потребуются. Так что проверим его еще раз — потом будет не до того.

Реализация подгрузки содержимого

Теоретическая часть главы вышла очень короткой. Значит, больше времени останется на практику!

Давайте создадим новый Web-сайт, в котором и реализуем подгрузку содержимого. Его Web-страницы мы изготовим на основе соответствующих Web-страниц старого Web-сайта.

> ***Внимание!***
>
> Далее будем называть фрагменты содержимого Web-страниц, хранящиеся в отдельных файлах, также Web-страницами. Ведь это Web-страницы и есть! Если же потребуется отметить различие между "полноразмерной" Web-страницей и фрагментом, мы это особо отметим.

Прежде всего, создадим папку, которая станет корневой для нового Web-сайта. Назовем ее Site 2. Скопируем туда из корневой папки старого Web-сайта (Site 1) файлы main.css, main.js и ext-core.js. И сразу же создадим в ней папки chapters (для Web-страниц разделов), tags (для Web-страниц с описаниями тегов HTML), attrs (для Web-страниц, описывающих атрибуты стиля CSS) и samples (для Web-страниц с примерами).

Откроем в Блокноте Web-страницу index.htm старого Web-сайта и пересохраним под тем же именем в папке Site 2. Удалим из ее контейнера `cmain` все содержимое и сохраним.

Снова откроем Web-страницу index.htm старого Web-сайта и пересохраним ее в папке chapters, вложенной в папку Site 2, под именем html.htm. Удалим из нее все, кроме содержимого контейнера `cmain`, и сохраним снова.

Аналогично создадим остальные Web-страницы нового Web-сайта. И не забудем скопировать файлы графического изображения, аудио- и видеоклипа из папки старого Web-сайта в соответствующую папку нового.

Откроем Web-страницу index.htm уже нового Web-сайта и исправим интернет-адреса в гиперссылках, указывающих на другие Web-страницы. При этом будем иметь в виду, что интернет-адреса всех файлов с внедренными объектами (изображениями, аудио- и видеороликами) следует указывать относительно Web-страницы index.htm — ведь именно она фактически будет их загружать.

Web-страницы готовы. Настала пора создавать соответствующее поведение.

Откроем Web-страницу index.htm нового Web-сайта в Блокноте и удалим из секции ее заголовка код, объявляющий переменные `outerIndex` и `innerText`. Они нам больше не потребуются, так что незачем засорять память компьютера ненужными данными.

Потом откроем файл main.js нового Web-сайта, найдем в нем объявление функции `selectItem` и удалим его. Также удалим выражение, вызывающее эту функцию, из тела функции, которая передается методу `onReady` объекта `Ext`. Весь этот код нам больше не понадобится — скрытие и открытие вложенных списков и выделение пунктов полосы навигации мы реализуем по-другому, более "интеллектуально".

Но как?

Прежде всего, мы будем хранить ссылки на выделенный в данный момент пункт полосы навигации и открытый вложенный список. Для этого мы объявим две переменные.

Далее мы напишем функцию, которая станет обработчиком события `click` пунктов полосы навигации. Эта функция будет выполнять три задачи. Во-первых, она реализует подгрузку содержимого из файла, чей интернет-адрес указан в гиперссылке

пункта меню, на котором щелкнули мышью, и вывод его в контейнере `cmain`. Во-вторых, она будет делать тексты примеров скрывающимися и открывающимися при щелчке мышью на их первых потомках (для этого мы используем код, написанный в *главе 16*). В-третьих, она будет управлять скрытием и открытием вложенных списков и выделением пунктов полосы навигации при щелчках на них.

- Если посетитель щелкнул на пункте "внешнего" списка, формирующего полосу навигации, который имеет вложенный список, эта функция:
  - удалит выделение с выделенного ранее пункта полосы навигации (если он был выделен). Выделенный пункт полосы навигации будет храниться в особой переменной, которую мы собираемся объявить;
  - скроет открытый ранее вложенный список (если он был открыт и не вложен в пункт, на котором щелкнули мышью, — в этом случае нам нет нужды скрывать его, чтобы тотчас открыть). Открытый вложенный список также будет храниться в особой переменной, которую мы собираемся объявить;
  - развернет список, вложенный в пункт, на котором щелкнули мышью.
- Если посетитель щелкнул на пункте "внешнего" списка, не имеющего вложенного списка, эта функция:
  - скроет открытый ранее вложенный список (если он был открыт);
  - удалит выделение с выделенного ранее пункта полосы навигации (если он был выделен и если это не тот же самый пункт, на котором щелкнули мышью, — в этом случае незачем снимать с него выделение, чтобы сразу же выделить);
  - выделит пункт, на котором щелкнули мышью.
- Если посетитель щелкнул на пункте вложенного списка, эта функция:
  - удалит выделение с выделенного ранее пункта полосы навигации (если он был выделен и если это не тот же самый пункт, на котором щелкнули мышью);
  - выделит пункт, на котором щелкнули мышью.

Еще нам понадобятся две вспомогательные функции. Первая, совсем простая, будет вызвана после загрузки базовой Web-страницы index.htm и сделает все вложенные списки в полосе навигации скрытыми.

Вторая функция удалит все обработчики событий, привязанные к первым потомкам контейнеров со стилевым классом `sample` ("вместилища" для текста примеров). Перед тем как выводить в контейнере `cmain` новое содержимое, нужно удалить обработчики событий, привязанные к старому его содержимому. Так мы очистим память компьютера от не нужных более обработчиков событий и устраним потенциальную возможность появления ошибок. Данная функция будет вызываться в теле функции, которую мы описали чуть раньше, перед подгрузкой и выводом содержимого другого файла.

Что ж, задачи ясны. За работу!

В самом начале файла main.js поместим два выражения:

```
var elLastInnerList = null;
var elLastItem = null;
```

Они объявляют переменные `elLastInnerList` и `elLastItem` и присваивают им значение `null`. Первая переменная будет хранить открытый в данный момент вложенный список, вторая — выделенный в данный момент пункт полосы навигации.

Далее объявим функцию `loadFragment`, которая будет обрабатывать событие `click` пунктов полосы навигации (листинг 17.1). Она будет принимать два параметра: пункт, на котором щелкнули мышью, в виде экземпляра объекта `Element` и экземпляр объекта `EventObject`, хранящий сведения о событии. Второй параметр необязательный.

Листинг 17.1

```
function loadFragment(elLI, e) {
  if (e)
    e.stopEvent();
  var elA = elLI.child("A");
  if (elA) {
    cleanupSamples();
    var href = elA.getAttribute("HREF");
    Ext.get("cmain").load({ url: href, success: prepareSamples });
    if (elLI.parent("UL").id == "navbar") {
      var elInnerList = elLI.child("UL");
      if (elInnerList) {
        if (elLastItem) {
          elLastItem.removeClass("selected");
          elLastItem = null;
        }
        if ((elLastInnerList) && (elLastInnerList.dom !=
        elInnerList.dom))
          elLastInnerList.setDisplayed(false);
        elInnerList.setDisplayed(true);
        elLastInnerList = elInnerList;
      } else {
        if (elLastInnerList) {
          elLastInnerList.setDisplayed(false);
          elLastInnerList = null;
        }
        if ((elLastItem) && (elLastItem.dom != elLI.dom))
          elLastItem.removeClass("selected");
        elLI.addClass("selected");
        elLastItem = elLI;
      }
```

```
    } else {
      if ((elLastItem) && (elLastItem.dom != elLI.dom))
        elLastItem.removeClass("selected");
      elLI.addClass("selected");
      elLastItem = elLI;
    }
  }
}
```

Рассмотрим листинг 17.1 построчно.

Если функции `loadFragment` был передан второй параметр — экземпляр объекта `EventObject`, хранящий сведения о событии, — отменяем действие события по умолчанию и останавливаем его всплытие:

```
if (e)
  e.stopEvent();
```

Этот код перекочевал из функции-обработчика события `click` пунктов полосы навигации нашего предыдущего Web-сайта (подробности — в *главе 16*).

Получаем гиперссылку, вложенную в пункт полосы навигации, на котором щелкнули мышью:

```
var elA = elLI.child("A");
```

Если эта гиперссылка существует, выполняем следующий код:

```
if (elA) {
```

Вызываем функцию `cleanupSamples`, которая удалит обработчики событий у первых потомков контейнеров с текстами примеров:

```
  cleanupSamples();
```

Мы объявим эту функцию потом.

Получаем интернет-адрес гиперссылки:

```
  var href = elA.getAttribute("HREF");
```

Загружаем в контейнер `cmain` содержимое файла, на который указывает полученный интернет-адрес:

```
  Ext.get("cmain").load({ url: href, success: prepareSamples });
```

После его загрузки вызываем функцию `prepareSamples`, которая добавит контейнерам с текстом примеров возможность скрытия и раскрытия в ответ на щелчки мышью. Эту функцию мы объявили в *главе 16*.

Получаем список — родитель пункта полосы навигации, на котором щелкнули мышью, и проверяем, `navbar` ли его имя:

```
  if (elLI.parent("UL").id == "navbar") {
```

Если это так, значит, данный пункт является пунктом "внешнего" списка `navbar`, в противном случае — вложенного.

Если это пункт "внешнего" списка, получаем вложенный в него список, если, конечно, он там присутствует:

```
var elInnerList = elLI.child("UL");
```

Проверяем, присутствует ли в пункте вложенный список:

```
if (elInnerList) {
```

Если вложенный список присутствует, проверяем, выделен ли в данный момент какой-либо пункт полосы навигации, т. е. содержит ли переменная `elLastItem` какое-либо значение, отличное от `null`:

```
if (elLastItem) {
  elLastItem.removeClass("selected");
  elLastItem = null;
}
```

Если это так, удаляем из привязки к выделенному пункту стилевой класс `selected`, делая его невыделенным, и присваиваем переменной `elLastItem` значение `null`, указывая тем самым, что ни один пункт полосы навигации в данный момент не выделен.

Также проверяем, открыт ли в данный момент какой-либо вложенный список (содержит ли переменная `elLastInnerList` какое-либо значение, отличное от `null`), и не тот ли это список, что вложен в пункт, на котором щелкнули мышью:

```
if ((elLastInnerList) && (elLastInnerList.dom !=
elInnerList.dom))
  elLastInnerList.setDisplayed(false);
```

Если это так, скрываем этот список.

Обратим внимание, как мы проверяем тождественность открытого в данный момент вложенного списка тому, что содержится в пункте, на котором щелкнули мышью. Мы сравниваем не экземпляры объекта `Element`, возвращенные методами библиотеки Ext Core, а экземпляры объекта `HTMLElement` Web-обозревателя. Дело в том, что методы библиотеки Ext Core в разные моменты времени могут вернуть совершенно разные экземпляры объекта `Element`, тем не менее, представляющие один и тот же элемент Web-страницы. В то время как экземпляр объекта `HTMLElement`, представляющий данный элемент Web-страницы, всегда один и тот же.

Чтобы получить доступ к экземпляру объекта `HTMLElement`, мы используем свойство `dom` объекта `Element`. Оно было описано в *главе 15*.

Далее нам остается только открыть вложенный список, присутствующий в пункте, на котором щелкнули мышью:

```
elInnerList.setDisplayed(true);
```

...и сохранить его в переменной `elLastInnerList`:

```
elLastInnerList = elInnerList;
```

Если вложенный список в пункте "внешнего" списка, на котором щелкнули мышью, отсутствует, выполняется следующий код:

```
    } else {
```

Проверяем, открыт ли в данный момент какой-либо вложенный список (содержит ли переменная `elLastInnerList` какое-либо значение, отличное от `null`):

```
      if (elLastInnerList) {
        elLastInnerList.setDisplayed(false);
        elLastInnerList = null;
      }
```

Если это так, скрываем открытый вложенный список и присваиваем переменной `elLastInnerList` значение `null`, указывая, что ни один вложенный список в данный момент не открыт.

Затем проверяем, выделен ли в данный момент какой-либо пункт полосы навигации (содержит ли переменная `elLastItem` какое-либо значение, отличное от `null`), и не тот ли это пункт, на котором щелкнули мышью:

```
      if ((elLastItem) && (elLastItem.dom != elLI.dom))
        elLastItem.removeClass("selected");
```

Здесь мы также сравниваем экземпляры объекта `HTMLElement` Web-обозревателя. Если какой-то пункт выделен и это не тот же самый пункт, на котором щелкнули мышью, снимаем с него выделение, убрав стилевой класс `selected` из привязки к нему.

Выделяем пункт, на котором щелкнули мышью, привязав к нему стилевой класс `selected`:

```
      elLI.addClass("selected");
```

И сохраняем этот пункт в переменной `elLastItem`:

```
      elLastItem = elLI;
    }
```

Если пункт, на котором щелкнули мышью, находится во вложенном списке, выполняется следующий код:

```
  } else {
```

Далее все нам уже знакомо:

```
    if ((elLastItem) && (elLastItem.dom != elLI.dom))
      elLastItem.removeClass("selected");
    elLI.addClass("selected");
    elLastItem = elLI;
  }
}
```

Проверяем, выделен ли в данный момент какой-либо пункт полосы навигации, и не тот ли это пункт, на котором щелкнули мышью. Если это так, снимаем с него выде-

ление, убрав стилевой класс `selected` из привязки к нему. Далее выделяем пункт, на котором щелкнули мышью, привязав к нему стилевой класс `selected`, и сохраняем этот пункт в переменной `elLastItem`. На этом выполнение функции `loadFragment` закончено.

Теперь объявим функцию `hideInnerLists`, которая скроет все вложенные списки в полосе навигации после загрузки базовой Web-страницы index.htm. Вот соответствующий код:

```
function hideInnerLists() {
  var ceInnerLists = Ext.get("navbar").select("UL");
  ceInnerLists.setDisplayed(false);
}
```

Комментировать здесь особо нечего. Мы получаем все списки, вложенные во "внешний" список `navbar`, и скрываем их.

На очереди — функция `cleanupSamples` (листинг 17.2). Она удалит обработчики событий, привязанные к первым потомкам контейнеров с текстом примеров (тех, к которым привязан стилевой класс `sample`).

Листинг 17.2

```
function cleanupSamples() {
  var ceSamples = Ext.select(".sample");
  ceSamples.each(function(el, cl, ind){
    var elH6 = el.child(":first");
    elH6.removeAllListeners();
  });
}
```

Получаем все элементы Web-страницы с привязанным стилевым классом `sample`, для каждого получаем первый потомок и убираем все привязанные к нему ранее обработчики событий.

Осталось только внести исправления в код тела функции, которая передается методу `onReady` объекта `Ext`. Прежде всего, найдем выражение, привязывающее обработчик события `click` к пунктам всех списков в полосе навигации. Исправим его так, чтобы оно выглядело следующим образом:

```
ceLinks.on("click", function(e, t){ loadFragment(Ext.get(this), e) });
```

Тело нового обработчика этого события представляет собой вызов объявленной нами ранее функции `loadFragment`. В качестве параметров мы передаем ей экземпляр объекта `Element`, представляющий пункт полосы навигации, на котором щелкнули мышью, и полученный из экземпляра объекта `HTMLElement`, доступный из переменной `this`, а также экземпляр объекта `EventObject`, хранящий сведения о событии и переданный функции-обработчику первым параметром.

А в конце тела функции, которая передается методу `onReady` объекта `Ext`, вставим два выражения:

```
hideInnerLists();
loadFragment(Ext.get("navbar").child("> LI:first"));
```

Первое вызывает функцию `hideInnerLists`, которую мы объявили ранее. Эта функция скроет все вложенные списки в полосе навигации после загрузки базовой Web-страницы index.htm.

Второе выражение вызовет функцию `loadFragment` и передаст ей в качестве параметра первый пункт "внешнего" списка `navbar` полосы навигации ("HTML"). В результате изначально в контейнер `cmain` будет загружено описание языка HTML (содержимое файла html.htm, хранящегося в папке chapters).

Сохраним файл main.js. И проверим наш второй Web-сайт в действии. Как мы помним, для этого понадобится Web-сервер, который у нас уже есть — Microsoft Internet Information Services. (Мы установили его еще в *главе 1*.)

С помощью любой программы управления файлами (того же Проводника) откроем корневую папку созданного при установке этого Web-сервера тестового Web-сайта (папка c:\Inetpub\wwwroot). Очистим корневую папку от имеющихся там файлов и скопируем в нее все содержимое папки Site 2 (если кто забыл — в ней мы храним все файлы нового Web-сайта).

> ***ВНИМАНИЕ!***
>
> Описанную операцию мы будем выполнять всякий раз после любой правки файлов Web-сайта. Автор больше не будет об этом напоминать.

> ***НА ЗАМЕТКУ***
>
> Впрочем, есть способ избежать подобной операции: поместить рабочие файлы Web-сайта в корневую папку Web-сервера и работать с ними прямо там.

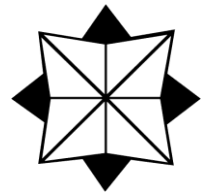
Откроем новый Web-сайт в Web-обозревателе, для чего запустим его и наберем в поле ввода интернет-адреса **http://localhost**. Этот интернет-адрес, как мы помним из *главы 1*, идентифицирует наш собственный компьютер — локальный хост.

Когда главная Web-страница нового Web-сайта откроется, проверим, присутствует ли в контейнере `cmain` изначально описание языка HTML и открыт ли первый вложенный список (находящийся в пункте "HTML" полосы навигации). Пощелкаем на пунктах полосы навигации и посмотрим, как меняется содержимое контейнера `cmain`. И отметим, насколько быстро оно меняется. Вот что значит современные технологии!

Что дальше?

В этой главе мы создали новый Web-сайт, в котором реализовали самую современную на данный момент технологию — подгружаемое содержимое. Теперь нам есть чем похвастаться перед коллегами!

В следующей главе мы совладаем со второй технологией — "последним писком" Web-моды — генерируемым содержимым. Мы будем создавать полосу навигации нашего Web-сайта программно, в Web-сценарии, на основе сведений, хранящихся в базе данных. Ну и попутно узнаем, что такое база данных.



ГЛАВА 18

Генерируемое содержимое

В предыдущей главе мы познакомились с одной из самых передовых интернет-технологий, применяемых в Web-дизайне, — подгружаемым содержимым. И, как оказалось, подгружать только часть содержимого Web-страницы значительно выгоднее, чем загружать Web-страницу целиком, — сокращается объем информации, пересылаемой по сети, а значит, уменьшается время ее загрузки.

В этой главе мы узнаем еще об одной интернет-технологии, постепенно завоевывающей популярность, — генерируемом содержимом. Мы научимся создавать часть содержимого Web-страницы программно, с помощью Web-сценариев, на основе информации, хранящейся в базе данных. А также узнаем, что такое база данных и как ее создать средствами JavaScript.

Введение в генерируемое содержимое. Базы данных

Генерируемое содержимое Web-страницы, как следует из определения, — это фрагменты ее содержимого, которые не описываются в HTML-коде, а создаются программно, особым Web-сценарием. Содержимое может генерироваться как при открытии Web-страницы, так и в процессе ее просмотра, в ответ на действия посетителя.

Какие фрагменты содержимого Web-страницы можно генерировать программно? Да какие угодно! Программно можно создавать, скажем, текст сведений об авторских правах, добавив соответствующий код в файл Web-сценариев, выполняемый при загрузке каждой Web-страницы Web-сайта (у нас это файл main.js). Можно создавать гиперссылки, указывающие на Web-страницы с какими-либо дополнительными пояснениями или связанными сведениями (раздел "См. также"). Можно создавать оглавления, списки использованных при написании статьи материалов, предметные указатели. Да и вообще всю Web-страницу можно создать программно (но это, конечно, совсем уж экстремальный случай).

У генерируемого содержимого есть два неоспоримых достоинства.

Достоинство первое — сокращение размера HTML-кода Web-страниц. В самом деле, если какая-то часть содержимого Web-страницы создается программно, значит,

нет нужды "забивать" этот фрагмент в HTML-код. Код станет компактнее, файл загрузится по сети быстрее, и посетитель будет доволен.

Достоинство второе — унификация данных. Здесь придется начать издалека...

Создать программно текст сведений об авторских правах легко — нужно только найти подходящий метод библиотеки Ext Core, написать вызывающее его выражение и передать данному методу строку, содержащую нужный текст или HTML-код, создающий требуемый текст. Соответствующий Web-сценарий займет всего несколько строк.

Совсем другое дело — создание, скажем, списка материалов, использованных при написании статьи. Помимо самого Web-сценария, выполняющего генерацию этого списка, нам нужны данные, на основе которых он будет создаваться: названия источников, авторы, год выхода и пр. Откуда их взять?

Подготовить самим, разумеется. Никто, кроме нас, этого не сделает. Но в каком виде их создавать?

Давайте посмотрим на список использованных материалов с точки зрения программиста. Что он собой представляет? Правильно, набор однотипных позиций, каждая из которых описывает один материал: статью, книгу, Web-страницу и т. д. Таких позиций может быть сколько угодно.

Смотрим дальше. Каждая позиция этого списка содержит однотипные фрагменты данных: автор, название, год выпуска, номер страницы и пр. Каждый фрагмент можно представить в виде константы JavaScript, имеющей определенный тип: строковый, числовой, логический и пр. Так, название и имя автора материала представляют собой константы строкового типа, а год выхода — константу числового типа.

И еще. Каждая позиция списка содержит строго определенное количество фрагментов данных, имеющих строго определенное назначение и строго определенный тип. Так, позиция, описывающая материал, включает его название, имя его автора и год выхода. Возможно, какие-либо фрагменты будут необязательными; например, номер страницы имеет смысл указывать только у печатного материала, а у Web-страницы его можно опустить, поскольку он не имеет смысла.

Теперь переведем все это на терминологию JavaScript.

- Сам список материалов — это массив.
- Отдельная позиция списка материалов — это либо также массив, либо, что удобнее, экземпляр объекта `Object`, т. е. конфигуратор. (Хотя конфигуратором его называть некорректно — он хранит не набор параметров для метода, а данные, предназначенные для обработки.)

Значит, мы можем описать список использованных материалов в виде массива JavaScript, каждый элемент которого представляет собой конфигуратор, свойства которого описывают различные фрагменты данных. Сделать это довольно просто.

Мы можем пойти дальше, создав "глобальный" массив, содержащий сведения обо всех материалах, использованных при написании всех статей, которые мы опубликовали на своем Web-сайте. А для отдельных статей мы создадим "локальный"

массив — список материалов, элементы которого хранят индексы соответствующих элементов "глобального" массива. Ведь зачастую при написании разных статей используются одни и те же материалы, и дублировать их описания в разных "локальных" массивах нерационально.

JavaScript-код, создающий такой массив, помещают в отдельный файл Web-сценария. Он будет одинаковым для всех Web-страниц, содержащих статьи.

Подобные хранилища данных, предназначенные для генерирования содержимого Web-страниц и прочих целей и хранящиеся отдельно от самих Web-страниц, называются *базами данных*.

> ***НА ЗАМЕТКУ***
>
> Конечно, это базы данных в широком смысле этого слова. С "настоящими" базами данных, управляемыми программами Microsoft Access, Microsoft SQL Server, Oracle, MySQL, Embarcadero Interbase и пр., они имеют не очень много сходства.

Но мы совсем забыли о втором достоинстве генерируемого содержимого. В чем же оно заключается?

А в том, что хранящиеся в базе данные можно задействовать по-разному.

- Мы можем сгенерировать на их основе "локальные" списки материалов для отдельных статей, которые поместим на Web-страницы с текстом этих статей.
- Мы можем сгенерировать на основе базы данных "глобальный" список материалов для всех статей, который поместим на отдельную Web-страницу.
- Мы можем выполнять в базе данных поиск использованного материала и выводить на Web-страницу его результаты.

Иначе говоря, мы можем на основе одной и той же информации, хранящейся в базе данных, генерировать разное содержимое Web-страницы. Вот она — унификация данных!

Реализация генерируемого содержимого

Давайте реализуем генерируемое содержимое на нашем Web-сайте. Мы создадим базу данных, содержащую список всех Web-страниц (файлов с подгружаемым содержимым) с названиями и гиперссылками.

- На основе этой базы данных мы будем генерировать вложенные списки полосы навигации.
- В *главе 19* мы осуществим вывод на каждой Web-странице раздела "См. также", в котором поместим гиперссылки на Web-страницы с "родственными" данными.
- В последующих главах мы организуем поиск нужной Web-страницы, опять же, на основе данных, хранящихся в этой базе.

Для генерирования содержимого Web-страницы мы применим соответствующие методы объекта `Element` библиотеки Ext Core (см. *главу 15*). Этих методов довольно много, и не составит труда выбрать подходящий.

Создание базы данных

Чтобы генерировать содержимое Web-страницы на основе каких-то данных, нужно сначала подготовить сами данные. Поэтому начнем работу с создания базы данных.

Наша база данных будет представлять собой три массива, хранящие списки Web-страниц, которые описывают, соответственно, теги HTML, атрибуты стиля CSS и примеры. Элементы массивов будут хранить конфигураторы, описывающие эти Web-страницы и хранящие их названия и интернет-адреса в виде строк.

Что ж, цель поставлена. За работу!

Создадим текстовый файл с именем data.js и поместим его в папке Site 2. Откроем его и наберем код, приведенный в листинге 18.1.

Листинг 18.1

```
var aHTML = [];
aHTML[0] =  { name: "!DOCTYPE", url: "tags/t_doctype.htm" };
aHTML[1] =  { name: "AUDIO", url: "tags/t_audio.htm" };
aHTML[2] =  { name: "BODY", url: "tags/t_body.htm" };
aHTML[3] =  { name: "EM", url: "tags/t_em.htm" };
aHTML[4] =  { name: "HEAD", url: "tags/t_head.htm" };
aHTML[5] =  { name: "HTML", url: "tags/t_html.htm" };
aHTML[6] =  { name: "IMG", url: "tags/t_img.htm" };
aHTML[7] =  { name: "META", url: "tags/t_meta.htm" };
aHTML[8] =  { name: "P", url: "tags/t_p.htm" };
aHTML[9] =  { name: "STRONG", url: "tags/t_strong.htm" };
aHTML[10] = { name: "TITLE", url: "tags/t_title.htm" };
aHTML[11] = { name: "VIDEO", url: "tags/t_video.htm" };

var aCSS = [];
aCSS[0] =  { name: "border", url: "attrs/a_border.htm" };
aCSS[1] =  { name: "color", url: "attrs/a_color.htm" };
aCSS[2] =  { name: "margin", url: "attrs/a_margin.htm" };

var aSamples = [];
aSamples[0] =  { name: "Гиперссылки", url: "samples/a_hyperlinks.htm" };
aSamples[1] =  { name: "Контейнеры", url: "samples/a_containers.htm" };
aSamples[2] =  { name: "Таблицы", url: "samples/a_tables.htm" };
```

Здесь мы объявили массивы `aHTML`, `aCSS` и `aSamples`, которые будут хранить списки Web-страниц, описывающих, соответственно, теги HTML, атрибуты стиля CSS и примеры.

Элементы каждого из этих массивов хранят конфигураторы с двумя свойствами:

- `name` — название соответствующего пункта вложенного списка в виде строки;
- `url` — интернет-адрес файла с фрагментом содержимого также в виде строки.

Сохраним набранный код в кодировке UTF-8. Вообще, не забываем, что после любых правок кода его нужно сохранять.

Затем откроем в Блокноте Web-страницу index.htm и вставим в ее секцию заголовка такой код:

```
<SCRIPT SRC="data.js"></SCRIPT>
```

Он загрузит и выполнит только что созданный нами файл Web-сценария data.js. В результате в памяти компьютера будут созданы три массива — наша база данных.

Отметим, что файл Web-сценария загружается и выполняется в самом начале загрузки Web-страницы index.htm. Поэтому, когда дело дойдет до исполнения Web-сценариев, хранящихся в файле main.js (а они выполняются в конце загрузки Web-страницы), наша база данных уже будет сформирована и готова к работе.

Генерирование полосы навигации

Теперь можно заняться кодом, генерирующим вложенные списки в полосе навигации.

Снова откроем Web-страницу index.htm в Блокноте, если уже ее закрыли. Удалим весь HTML-код, формирующий пункты вложенных списков в полосе навигации, но оставим фрагмент, создающий сами вложенные списки. Результат приведен в листинге 18.2.

Листинг 18.2

```
<UL ID="navbar">
  <LI><A HREF="chapters/html.htm">HTML</A>
    <UL>
    </UL>
  </LI>
  <LI><A HREF="chapters/css.htm">CSS</A>
    <UL>
    </UL>
  </LI>
  <LI><A HREF="chapters/samples.htm">Примеры</A>
    <UL>
    </UL>
  </LI>
  <LI><A HREF="chapters/about.htm">О разработчиках</A></LI>
</UL>
```

После этого откроем файл Web-сценария main.js и поместим перед вызовом метода `onReady` объекта `Ext` код из листинга 18.3.

Листинг 18.3

```
function generateInnerList(aDataBase, elInnerList) {
  for (var i = 0; i < aDataBase.length; i++) {
```

```
    var s = "<LI><CODE><A HREF=\"" + aDataBase[i].url + "\">" +
    aDataBase[i].name + "</A></CODE></LI>";
    elInnerList.insertHtml("beforeEnd", s);
  }
}
```

Он объявляет функцию `generateInnerList`, которая будет создавать пункты одного вложенного списка. Эта функция принимает два обязательных параметра:

- один из формирующих нашу базу данных массивов; на основе этого массива будут созданы пункты указанного вложенного списка;
- вложенный список, в котором будут создаваться эти пункты, в виде экземпляра объекта `Element`.

Ее код очень прост. Рассмотрим его построчно.

Запускаем цикл со счетчиком, в теле которого будут создаваться пункты списка:

```
for (var i = 0; i < aDataBase.length; i++) {
```

Счетчик цикла — переменная `i`, начальное значение счетчика — 0, конечное значение — размер массива, переданного первым параметром (он берется из свойства `length` объекта `Array`; подробнее — в *главе 14*), приращение — инкремент счетчика. В результате цикл выполнится столько раз, сколько элементов содержит массив, переданный первым параметром.

Формируем строку с HTML-кодом, создающим пункт списка:

```
  var s = "<LI><CODE><A HREF=\"" + aDataBase[i].url + "\">" +
  aDataBase[i].name + "</A></CODE></LI>";
```

Название пункта и интернет-адрес файла с фрагментом содержимого берем из соответствующих свойств конфигуратора, являющегося элементом переданного первым параметром массива.

Создаем пункт списка на основе строки, сформированной в предыдущем выражении:

```
  elInnerList.insertHtml("beforeEnd", s);
}
```

В качестве места, куда будет помещен новый пункт, мы указываем `"beforeEnd"` — перед закрывающим тегом. В результате новые пункты будут добавляться в конец списка.

На этом выполнение тела цикла завершается. А после того, как цикл закончит работу, завершится выполнение самой функции `generateInnerList`.

Теперь вставим в самое начало тела функции, передаваемой методу `onReady` объекта `Ext`, три выражения:

```
generateInnerList(aHTML, Ext.get("navbar").child("> LI:nth(1) UL"));
generateInnerList(aCSS, Ext.get("navbar").child("> LI:nth(2) UL"));
generateInnerList(aSamples, Ext.get("navbar").child("> LI:nth(3) UL"));
```

Мы трижды вызываем функцию `generateInnerList`, поочередно передавая ей три массива, составляющих базу данных, и три вложенных списка, формирующих полосу навигации.

Три приведенных выражения создадут пункты вложенных списков, формирующих полосу навигации. Следующие выражения привяжут к ним обработчики событий. В результате наша полоса навигации будет работать как прежде, будто она не создается Web-сценарием, а полностью формируется в HTML-коде.

Откроем готовую Web-страницу index.htm, набрав интернет-адрес **http://localhost** в Web-обозревателе, и убедимся в этом.

Сортировка базы данных

Итак, Web-сценарий, генерирующий полосу навигации, работает. Самое время дополнить полосу навигации еще парой пунктов.

Откроем файл Web-сценария data.js и добавим в массив `aCSS` два элемента:

```
aCSS[3] =  { name: "font-family", url: "attrs/a_font-family.htm" };
aCSS[4] =  { name: "font-size", url: "attrs/a_font-size.htm" };
```

Откроем Web-страницу index.htm в Web-обозревателе. И обнаружим, что во втором вложенном списке (перечисляющем атрибуты стиля CSS) появятся два новых пункта. Причем они окажутся в самом его конце, нарушив принятый нами алфавитный порядок.

Дело в том, что эти два новых элемента мы добавили в самый конец массива `aCSS`. Функция `generateInnerList` "пройдет" по этому массиву и создаст новые пункты в том порядке, в котором соответствующие элементы в нем присутствуют.

Но мы-то хотим, чтобы они выводились в алфавитном порядке! Значит, нужно как-то отсортировать массив `aCSS`.

Специально для таких случаев объект JavaScript `Array` поддерживает метод `sort`. Он как раз и выполняет сортировку массива, у которого вызван:

<массив>`.sort([`*<функция сравнения>*`])`

Если этот метод был вызван без параметров, он отсортирует массив по строковому представлению его элементов. Каждый элемент массива он преобразует в строку и отсортирует элементы по алфавитному порядку этих строк (а если точнее, то по кодам символов, составляющих эти строки).

Такая сортировка подойдет, если элементы массива хранят строки. Но если там окажутся числа или экземпляры объектов, результаты сортировки могут оказаться совсем не теми, что мы ожидаем. В самом деле, как будут отсортированы в этом случае экземпляры объектов — непонятно.

Но метод `sort` поддерживает необязательный параметр — функцию сравнения, которая поможет нам отсортировать элементы массива как нам нужно.

Функция сравнения должна принимать два параметра — сравниваемые элементы массива — и возвращать число, указывающее, какой из этих элементов с точки зрения программиста "меньше".

- Если первый элемент "меньше" второго, функция сравнения должна вернуть отрицательное число (обычно –1).
- Если оба элемента "равны", функция сравнения должна вернуть 0.
- Если первый элемент "больше" второго, функция сравнения должна вернуть положительное число (обычно 1).

Давайте напишем функцию (листинг 18.4), которая будет сравнивать элементы массивов `aHTML`, `aCSS` и `aSamples` по значению свойства `name` конфигуратора. Сравнивать строки мы будем с помощью знакомых нам по *главе 14* операторов сравнения — они прекрасно работают и со строками.

Листинг 18.4

```
function sortArray(c1, c2) {
  if (c1.name < c2.name)
    return -1
  else
    if (c1.name > c2.name)
      return 1
    else
      return 0;
}
```

Поместим код листинга 18.4 в самом конце файла Web-сценария data.js, после объявлений всех трех массивов.

Осталось только, собственно, выполнить сортировку массивов. Это сделают три следующих выражения, которые мы поместим после объявления функции сравнения:

```
aHTML.sort(sortArray);
aCSS.sort(sortArray);
aSamples.sort(sortArray);
```

Вот и все. Проверим Web-страницу index.htm в действии и убедимся, что пункты во вложенных списках расположены в алфавитном порядке.

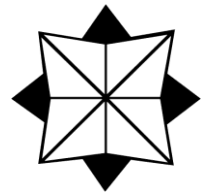
На этом пока закончим с генерируемым содержимым.

Что дальше?

В этой главе мы научились генерировать содержимое Web-страницы программно, с помощью особых Web-сценариев. Кроме того, мы познакомились с базами данных, научились создавать их средствами JavaScript и даже сортировать. Теперь вложенные списки в нашей полосе навигации создаются программно, на основе информации, хранящейся в базе данных.

В следующей главе мы продолжим заниматься генерируемым содержимым и рассмотрим более сложные случаи. А еще мы узнаем, что такое семантическая разметка данных и рассмотрим способы организации ее все теми же средствами JavaScript. JavaScript вообще многогранный язык...

ГЛАВА 19



Семантическая разметка данных

В предыдущей главе мы имели дело с генерируемым содержимым Web-страницы — содержимым, создаваемым с помощью Web-сценариев. Мы также узнали, что такое базы данных и как они могут нам помочь. В качестве практики мы создали базу данных, описывающую пункты вложенных списков нашей полосы навигации, и Web-сценарий, генерирующий эти пункты на основе хранящихся в базе данных сведений.

В этой главе мы продолжим заниматься генерируемым содержимым и создадим раздел "См. также" для каждой Web-страницы нашего Web-сайта. В новый раздел мы поместим гиперссылки на Web-страницы со связанными материалами: описаниями тегов HTML и атрибутов стиля CSS, имеющих сходное назначение, и примерами использования того или иного тега или атрибута стиля. Раздел "См. также" будет генерироваться программно.

А еще мы познакомимся с принципом семантической разметки данных и научимся создавать ее средствами JavaScript. С нее-то мы и начнем.

Введение в семантическую разметку данных

Мы собираемся создать на всех Web-страницах раздел "См. также". Этот раздел будет включать гиперссылки на Web-страницы, хранящие описания тегов HTML и атрибутов стиля CSS, имеющих аналогичное назначение, и тексты примеров использования данного тега или атрибута стиля, т. е. гиперссылки на Web-страницы со сведениями, которые так или иначе связаны с текущей Web-страницей.

Такие данные, связанные друг с другом, в Web-дизайне и Web-программировании встречаются очень часто. Это не обязательно Web-страницы — связанными могут быть и данные, обрабатываемые Web-сценариями.

В таком случае говорят, что между данными установлены *семантические связи*. Такие связи указывают, что эти данные имеют отношение "главный — подчиненный", "метка — собственно данные", являются составными частями большой структуры данных или цепочки связанных данных. А процесс установки семантических связей между данными называется *семантической разметкой*.

Примером данных, связанных отношениями "главный — подчиненный", можно назвать набор Web-страниц, одна из которых содержит оглавление большого текста, а остальные — отдельные главы этого текста. Оглавление представляет собой набор гиперссылок, указывающих на Web-страницы с отдельными главами, причем гиперссылки и выступают в качестве семантических связей.

Web-дизайнер может создать на каждой Web-странице, хранящей отдельную главу, гиперссылку на Web-страницу, где представлена следующая глава. Это пример семантической связи данных в цепочку, где в качестве связи выступают также гиперссылки.

Что касается связи данных в единую структуру, то в качестве примера можно привести базу данных, созданную нами в *главе 18*. Отдельные фрагменты данных — конфигураторы, — хранящие названия и интернет-адреса отдельных Web-страниц, с помощью семантической связи объединены в большую структуру — список Web-страниц. В качестве связи выступает механизм массивов, предлагаемый языком JavaScript.

Правда, все эти случаи — суть простейшие формы семантической разметки данных, реализуемой, так сказать, "подручными" средствами. Для создания полноценной семантической разметки существуют особые языки описания как связываемых данных, так и, собственно, самих связей. Эти языки разметки, как и прочие интернет-стандарты, разрабатывает организация W3C. Обработкой семантически связанных данных занимаются особые программы-серверы, работающие совместно с Web-серверами. А для извлечения таких данных нужны особые программы-клиенты.

Поскольку в нашей книге описываются исключительно клиентские технологии, мы не будем рассматривать языки семантической разметки данных и программы для их обработки, а сосредоточимся именно на "подручных" средствах: гиперссылках и механизмах языка JavaScript, таких как массивы.

Конечно, средства эти не такие гибкие, как предоставляемые специализированными языками и программами, но для нужд обычного Web-дизайнера их вполне достаточно. К тому же, они имеют неоспоримое достоинство — прекрасно работают в обычном Web-обозревателе без привлечения каких бы то ни было дополнительных программ.

Реализация семантической разметки средствами JavaScript

Давайте еще раз перечислим, что попадет в раздел "См. также" каждой Web-страницы.

- Гиперссылки на Web-страницы, описывающие теги HTML и атрибуты стиля CSS со сходным назначением.
- Гиперссылки на Web-страницы с примерами применения данного тега или атрибута стиля.

- Гиперссылки на Web-страницы с описаниями тегов и атрибутов стиля, присутствующих в данном примере.

Раздел "См. также" будет генерироваться программно, специальным Web-сценарием. Генерироваться он будет на основе каких-то данных — это очевидно. Но каких?

В *главе 18* мы создали базу данных, хранящую список всех Web-страниц нашего Web-сайта с их интернет-адресами и краткими названиями. На ее основе создаются вложенные списки в полосе навигации; краткие названия Web-страниц превращаются в тексты пунктов, а их интернет-адреса становятся интернет-адресами гиперссылок, помещаемых в эти пункты.

Напрашивается решение: для каждой Web-страницы создать подобную, но "локальную" базу данных, которая будет хранить список связанных Web-страниц. Тогда мы сможем написать Web-сценарий, который "просмотрит" эту базу данных и в итоге создаст раздел "См. также". Мы уже проделали такое в той же *главе 18*, значит, сделаем еще раз.

Мы не будем хранить в "локальной" базе данных полное описание этих Web-страниц, с краткими названиями и интернет-адресами. Как уже говорилось в *главе 18*, это нерационально. Лучше хранить в ней индексы соответствующих элементов "глобальной" базы данных, хранящейся в файле data.js, и какой-либо признак, указывающий, в каких именно массивах, составляющих "глобальную" базу данных, находятся эти элементы. Тогда мы избежим многократного повторения одних и тех же сведений в разных базах данных.

А если (как в нашем случае) элементы "глобальной" базы данных хранят конфигураторы, или вообще экземпляры объектов, задача значительно упрощается. Вспомним, что говорилось в *главе 14*: переменная с экземпляром объекта фактически хранит ссылку на него, а при присваивании ее значения другой переменной выполняется присваивание именно ссылки — экземпляр объекта при этом не дублируется. Значит, мы можем присвоить элементам "локальной" базы данных экземпляры объектов, которые извлечем из соответствующих элементов "глобальной" базы данных.

Фактически мы реализуем семантическую связь между "локальными" и "глобальной" базами данных средствами JavaScript. Это будет связь типа "метка — собственно данные".

"Локальную" базу данных мы также оформим в виде Web-сценария, который объявляет соответствующий массив (или массивы), и сохраним его в том же файле, в котором хранится HTML-код Web-страницы (точнее, подгружаемого фрагмента содержимого). Так будет проще — на одну Web-страницу будет приходиться один файл, а не два.

Но тут возникает большая проблема. Дело в том, что средства подгрузки содержимого, предоставляемые Web-обозревателем, не выполняют ни хранящиеся в подгружаемом фрагменте внутренние Web-сценарии, ни привязанные к нему внешние. Это сделано ради безопасности.

Что же нам делать?

Поместить связанные данные, на основе которых будет формироваться раздел "См. также", прямо в "глобальную" базу данных. Для этого мы создадим в конфигураторах, описывающих отдельные Web-страницы, еще одно свойство — `related`. Оно будет хранить массив элементов массивов `aHTML`, `aCSS` и `aSamples`, описывающих связанные Web-страницы. Это свойство мы сделаем необязательным, т. е. в каких-то случаях его можно не указывать, и Web-сценарий, создающий раздел "См. также", сработает правильно.

Так мы создадим семантические связи типа "главный — подчиненный". В качестве связи будет выступать механизм массивов JavaScript.

Сказано — сделано! Откроем файл Web-сценариев data.js, найдем в нем место, где заканчивается код, который объявляет массивы `aHTML`, `aCSS` и `aSamples`, и вставим туда такое выражение:

```
aHTML[0].related =  [aHTML[4], aHTML[7], aHTML[10]];
```

Мы взяли первый элемент массива `aHTML` (с индексом 0), добавили к хранящемуся в нем конфигуратору свойство `related` и присвоили этому свойству массив из трех конфигураторов, хранящихся в пятом, восьмом и одиннадцатом элементах (индексы 4, 7 и 10) того же массива. Таким образом мы указали, что для Web-страницы с описанием тега `<!DOCTYPE>` связанными являются Web-страницы с описаниями тегов `<HEAD>`, `<META>` и `<TITLE>`.

Но почему мы написали это выражение после кода, объявляющего все элементы массивов `aHTML`, `aCSS` и `aSamples`? Да потому, что перед тем, как присваивать какой-либо переменной или элементу массива экземпляр объекта, его нужно создать. Иначе данная переменная или элемент массива получит значение `undefined`, что нам совсем не нужно.

После этого выражения добавим такое:

```
aHTML[8].related =  [aHTML[3], aHTML[9], aCSS[0]];
```

Здесь мы указали, что для Web-страницы с описанием тега `<P>` (именно эту Web-страницу описывает элемент массива `aHTML` с индексом 8) связанными являются Web-страницы с описаниями тегов `<EM>` и `<STRONG>` и атрибута стиля `border`.

Аналогично укажем связанные данные для остальных Web-страниц нашего Web-сайта. Необязательно для всех — хотя бы для нескольких, чтобы только проверить, как все это работает.

Создание раздела "См. также"

При создании раздела "См. также" нам потребуется решить четыре задачи.

- Соотнести каждый пункт полосы навигации со списком связанных материалов соответствующей Web-страницы.
- Собственно создать раздел "См. также" после загрузки Web-страницы.
- Обеспечить загрузку Web-страницы при щелчке на гиперссылке этого раздела.

- Реализовать скрытие и раскрытие вложенных списков и выделение пункта полосы навигации при щелчке на гиперссылке раздела "См. также".

Начнем по порядку.

Откроем файл Web-сценария main.js и найдем в нем объявление функции `generateInnerList`, которая, как мы помним, создает за один раз один вложенный список в полосе навигации. Немного исправим ее код, чтобы он выглядел так, как показано в листинге 19.1.

Листинг 19.1

```
function generateInnerList(aDataBase, elInnerList) {
  for (var i = 0; i < aDataBase.length; i++) {
    var s = "<LI><CODE><A HREF=\"" + aDataBase[i].url + "\">" +
    aDataBase[i].name +
    "</A></CODE></LI>";
    var htelItem = elInnerList.insertHtml("beforeEnd", s);
    htelItem.related = aDataBase[i].related;
  }
}
```

В *главе 14* мы узнали о свойствах и методах, которые мы можем создать у любого экземпляра любого объекта и которые относятся только к данному экземпляру, не затрагивая другие, даже созданные на основе того же объекта. Так вот, для хранения списка материалов, связанных с какой-либо Web-страницей, мы создадим свойство экземпляра с именем `related`. Это свойство будет создано у экземпляра объекта `HTMLElement`, представляющего пункт вложенного списка, который соответствует данной Web-странице. Метод `insertHtml` (см. листинг 19.1) как раз возвращает экземпляр объекта `HTMLElement`, что нам на руку.

Первую задачу — соотношение пункта полосы навигации со списком связанных материалов — мы решили.

Найдем объявление функции `prepareSamples`, которая подготавливает текст примеров и, что важно для нас, выполняется после загрузки фрагмента содержимого. Добавим в конец тела функции, передаваемой в качестве параметра методу `each`, такое выражение:

```
generateRelated();
```

Это вызов функции `generateRelated`, которая и создаст раздел "См. также" и которую мы сейчас объявим.

Не будем откладывать дело в долгий ящик. Листинг 19.2 содержит объявление функции `generateRelated`.

Листинг 19.2

```
function generateRelated() {
  var s = "";
```

```
  var oRelated = elLastItem.dom.related;
  if (oRelated) {
    for (var i = 0; i < oRelated.length; i++) {
      if (s != "")
        s += ", ";
      s += "<CODE><A HREF=\"" + oRelated[i].url + "\">" +
      oRelated[i].name + "</A></CODE>";
    }
    var htelRelated = Ext.get("cmain").insertHtml("beforeEnd",
    "<P>См. также: " + s + "</P>");
    Ext.fly(htelRelated).select("A").on("click", function(e, t) {
      var href = Ext.fly(this).getAttribute("href");
      var elA = Ext.get("navbar").child("A[href=" + href + "]");
      var elItem = elA.parent("LI");
      loadFragment(elItem, e);
    });
  }
}
```

Рассмотрим приведенный код построчно.

Объявляем переменную `s`, в которой будет храниться HTML-код, формирующий раздел "См. также" в виде строки:

```
var s = "";
```

Первое, что нам нужно, — получить список материалов, связанных с загруженной в данный момент Web-страницей (точнее, фрагментом содержимого). Как его получить? Вспомним, что еще в *главе 17* мы объявили переменную `elLastItem`. Она хранит пункт полосы навигации, на котором посетитель щелкнул мышью и который как раз и соответствует загруженному в данный момент фрагменту содержимого. Вот и решение:

```
var oRelated = elLastItem.dom.related;
```

Значение переменной `elLastItem` — это экземпляр объекта `Element`. Соответствующий ему экземпляр объекта `HTMLElement` мы можем получить через свойство `dom`. А полученный экземпляр объекта `HTMLElement`, в свою очередь, содержит свойство `related`, которое и хранит массив конфигураторов, представляющих Web-страницы со связанными материалами; мы сами создали это свойство при формировании пунктов вложенных списков (исправленная функция `generateInnerList`).

Проверяем, присутствует ли это свойство в экземпляре объекта `HTMLElement`, т. е. существует ли у данной Web-страницы список связанных материалов:

```
if (oRelated) {
```

Если существует, запускаем цикл со счетчиком, который обработает все элементы этого массива:

```
  for (var i = 0; i < oRelated.length; i++) {
```

Фрагменты HTML-кода, формирующие отдельные гиперссылки раздела "См. также", мы будем добавлять в строку, хранящуюся в переменной `s`:

```
    if (s != "")
      s += ", ";
```

Эти гиперссылки следует разделить запятыми, что и выполняет приведенное выражение. Оно проверяет, есть ли в переменной `s` непустая строка, т. е. сформирована ли в ней хотя бы одна гиперссылка, и, если это так, добавляет к ней запятую — разделитель гиперссылок.

Формируем HTML-код, создающий гиперссылку на очередную Web-страницу из списка связанных материалов, и добавляем его к значению переменной `s`:

```
    s += "<CODE><A HREF=\"" + oRelated[i].url + "\">" +
    oRelated[i].name + "</A></CODE>";
  }
```

На этом тело цикла завершается.

Далее создаем на основе сформированного HTML-кода абзац, который и станет разделом "См. также". Добавляем его в самый конец контейнера `cmain` (перед его закрывающим тегом):

```
  var htelRelated = Ext.get("cmain").insertHtml("beforeEnd",
  "<P>См. также: " + s + "</P>");
```

Выбираем из сформированного в предыдущем выражении абзаца все гиперссылки и привязываем к ним функцию — обработчик события `click`, которую там же и объявляем:

```
  Ext.fly(htelRelated).select("A").on("click", function(e, t) {
```

Первым делом эта функция получит значение атрибута `HREF` — интернет-адрес — гиперссылки, на которой щелкнули мышью:

```
    var href = Ext.fly(this).getAttribute("href");
```

Далее она ищет в списке `navbar` гиперссылку с тем же интернет-адресом:

```
    var elA = Ext.get("navbar").child("A[href=" + href + "]");
```

Пункты нашей полосы навигации не содержат повторяющихся интернет-адресов, поэтому такой прием будет работать.

Найдя такую гиперссылку, она получает ее родителя — сам пункт:

```
    var elItem = elA.parent("LI");
```

который, вместе с экземпляром объекта `EventObject`, представляющим возникшее событие, "скармливает" объявленной нами в *главе 17* функции `loadFragment`. Последняя выполнит загрузку соответствующего фрагмента содержимого и выделит соответствующий пункт полосы навигации:

```
    loadFragment(elItem, e);
  });
}
```

На этом выполнение функции `generateRelated` заканчивается.

Теперь отыщем код, объявляющий функцию `cleanupSamples`. Она, как мы помним, удаляет обработчики событий у текста примеров перед загрузкой другого фрагмента содержимого. Это нужно, чтобы не засорять внутренние структуры данных библиотеки Ext Core сведениями о привязке обработчиков событий к уже не существующим элементам Web-страницы.

Нам нужно дополнить этот код, чтобы он также удалял обработчики событий у гиперссылок раздела "См. также". Сделаем проще — будем выбирать все гиперссылки в контейнере `cmain` и удалять у них обработчики событий.

Листинг 19.3 содержит дополненное объявление функции `cleanupSamples`.

Листинг 19.3

```
function cleanupSamples() {
  var ceSamples = Ext.select(".sample");
  ceSamples.each(function(el, cl, ind){
    var elH6 = el.child(":first");
    elH6.removeAllListeners();
  });
  var ceA = Ext.get("cmain").select("A");
  ceA.removeAllListeners();
}
```

Так, вторая и третья задачи решены. Мы сформировали раздел "См. также" и реализовали загрузку Web-страницы при щелчке на гиперссылке этого раздела.

Теперь представим себе такую ситуацию. Посетитель щелкает на каком-либо пункте полосы навигации, Web-обозреватель выводит соответствующую Web-страницу и формирует на ней раздел "См. также". Этот раздел содержит несколько гиперссылок, указывающих на Web-страницы с описаниями тегов HTML (первый раздел — "HTML"), и несколько гиперссылок, указывающих на Web-страницы с описаниями атрибутов стиля CSS (второй раздел — "CSS"). Предположим, что загруженная в данный момент Web-страница также содержит описание тега HTML (принадлежит к первому разделу); значит, первый, представляющий Web-страницы из первого раздела, вложенный список в полосе навигации открыт, остальные — скрыты.

В этот момент посетитель щелкает на гиперссылке раздела "См. также", ведущей на Web-страницу с описанием атрибута стиля CSS (принадлежащей второму разделу), и Web-страница успешно загружается. По идее, второй вложенный список (который представляет все Web-страницы второго раздела) в полосе навигации должен быть открыт, а первый, открытый ранее, — скрыт. Ничего подобного! Первый вложенный список так и останется открытым.

Дело в том, что, создавая в *главе 17* Web-сценарий, управляющий скрытием и раскрытием вложенных списков в полосе навигации и выделением ее пунктов, мы полагали, что щелкать на пунктах полосы навигации будет человек. А он в принципе

не может щелкнуть на пункте, находящемся в скрытом списке, — ведь скрытый список вообще не присутствует на Web-странице. Поэтому мы не реализовали в функции `loadFragment` возможность открытия вложенного списка, на пункте которого щелкнул посетитель, — это было просто незачем.

Сейчас же мы написали функцию `generateRelated`, которая создает раздел "См. также" с набором гиперссылок и привязывает к ним обработчик события `click`. Этот обработчик находит в полосе навигации пункт, который имеет гиперссылку с тем же интернет-адресом, и, если можно так сказать, программно "щелкает" на нем. А он ведь может "щелкнуть" и на пункте скрытого списка!

Вывод: нам потребуется дополнить объявление функции `loadFragment` так, чтобы она открывала вложенный список, на пункте которого "щелкнули", если, конечно, он еще не открыт. Найдем это объявление и отыщем в нем фрагмент кода, приведенный в листинге 19.4.

Листинг 19.4

```
. . .
} else {
  if ((elLastItem) && (elLastItem.dom != elLI.dom))
    elLastItem.removeClass("selected");
  elLI.addClass("selected");
  elLastItem = elLI;
}
```

Этот код управляет выделением пункта вложенного списка, на котором щелкнули мышью или "щелкнули" программно. Дополним его так, как показано в листинге 19.5.

Листинг 19.5

```
} else {
  var elInnerList = elLI.parent("UL");
  if ((elLastInnerList) && (elLastInnerList.dom != elInnerList.dom))
    elLastInnerList.setDisplayed(false);
  elInnerList.setDisplayed(true);
  elLastInnerList = elInnerList;
  if ((elLastItem) && (elLastItem.dom != elLI.dom))
    elLastItem.removeClass("selected");
  elLI.addClass("selected");
  elLastItem = elLI;
}
```

Добавленные нами выражения находят вложенный список, в котором присутствует данный пункт, проверяют, открыт ли он в текущий момент, и, если не открыт, открывают, скрывая при этом ранее открытый список.

Вот решена и четвертая задача... Уф-ф-ф!

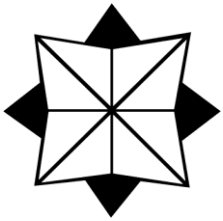
Сохраним все исправленные файлы и проверим готовую Web-страницу index.htm в действии. Откроем Web-обозреватель, наберем в поле ввода интернет-адрес **http://localhost**, пощелкаем на пунктах полосы навигации и гиперссылках раздела "См. также" и посмотрим на сменяющие друг друга Web-страницы и скрывающиеся и открывающиеся вложенные списки. Красота!

Здесь мы реализовали раздел "См. также" в виде абзаца с гиперссылками. Также его можно выполнить в виде списка или таблицы. Можно даже сделать раздел скрывающимся и раскрывающимся в ответ на щелчок мышью. Так, кстати, часто и поступают. Вы тоже можете так сделать; пусть это будет вашим домашним заданием.

Что дальше?

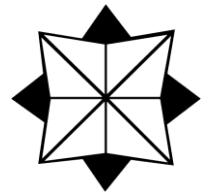
В этой главе мы познакомились с принципом семантической разметки данных и применили ее к нашей базе данных средствами JavaScript. В результате мы смогли создать на Web-страницах нашего Web-сайта раздел "См. также", содержащий гиперссылки на Web-страницы со связанными данными.

Помнится, мы планировали реализовать на Web-сайте еще и поиск материалов по введенному посетителем слову или фрагменту слова. Далее мы им займемся. И начнем с элементов управления, с помощью которых реализуется ввод данных посетителем.



ЧАСТЬ V

Последние штрихи



ГЛАВА 20

Web-формы и элементы управления

В предыдущих главах мы проделали большую работу. Во-первых, реализовали подгрузку фрагментов содержимого Web-страницы вместо загрузки целых Web-страниц. Во-вторых, сделали часть содержимого Web-страниц, а именно полосу навигации, генерируемой Web-сценарием. В-третьих, создали у всех Web-страниц, описывающих теги HTML, атрибуты стиля CSS и примеры, раздел "См. также", содержащий гиперссылки на Web-страницы со связанными материалами. Попутно мы узнали о базах данных и семантической разметке, без которых создать все это было бы крайне проблематично.

Но многое еще предстоит выполнить. В частности, реализовать поиск по Web-сайту. Идея была такой: посетитель вводит искомое слово — название тега или атрибута стиля или часть его названия, особый Web-сценарий ищет это слово в базе данных и выводит на Web-страницу гиперссылки на найденные Web-страницы.

Чтобы реализовать поиск, нам понадобятся:

- какие-либо средства, которые примут у посетителя искомое слово;
- Web-сценарий, который будет, собственно, выполнять поиск и формировать его результаты;
- элемент Web-страницы, куда будут выводиться результаты поиска.

С последним пунктом все просто. Мы создадим либо абзац, либо список, либо контейнер, где будет формироваться набор гиперссылок на искомые Web-страницы. С Web-сценарием, который будет выполнять поиск, тоже не должно возникнуть сложностей — достаточно просмотреть массивы, формирующие базу данных, и отобрать из них те элементы, что описывают подходящие Web-страницы.

Но как нам принять от посетителя искомое слово? Ясно, что для этого понадобится создать на Web-странице набор элементов управления как в Windows-приложениях: полей ввода, списков, флажков, переключателей и кнопок. Но как это сделать?

Web-формы и элементы управления HTML

Очень просто. Язык HTML предоставляет набор тегов для создания разнообразных элементов управления. Эти элементы управления уже "умеют" откликаться на дей-

ствия посетителя: поля ввода — принимать введенные символы, флажки — устанавливаться и сбрасываться, переключатели — переключаться, списки — прокручиваться, выделять пункты, разворачиваться и сворачиваться, а кнопки — нажиматься. Всем этим будет заниматься Web-обозреватель; нам самим ничего делать не придется.

Набор элементов управления, поддерживаемый HTML, невелик. Он включает поля ввода различного назначения, область редактирования, обычный и раскрывающийся список, флажок, переключатель, обычную и графическую кнопку. Более сложные элементы управления (таблицы, "блокноты" с вкладками, панели инструментов и пр.) так просто создать не получится. Хотя, как правило, для создания простых Web-приложений перечисленного ограниченного набора вполне достаточно.

> ***НА ЗАМЕТКУ***
>
> Существуют JavaScript-библиотеки для создания сложных элементов управления: индикаторов прогресса, регуляторов, "блокнотов", таблиц, панелей инструментов, меню и даже "лент" в стиле Microsoft Office 2007 и "окон". К таким библиотекам можно отнести, в частности, Ext, основанную на знакомой нам Ext Core.

Стандарт HTML требует, чтобы все элементы управления находились внутри Web-формы. *Web-форма* — это особый элемент Web-страницы, служащий "вместилищем" для элементов управления. На Web-странице она никак не отображается (если, конечно, мы не зададим для нее какого-либо представления); в этом Web-форма схожа с блочным контейнером. Для создания Web-формы HTML предусматривает особый тег.

Стандарт HTML требует, чтобы каждый элемент управления имел уникальное в пределах Web-формы имя. Это нужно для успешного формирования пар данных перед отправкой их серверному приложению.

Назначение Web-форм и элементов управления. Серверные приложения

Стандарт HTML поддерживал Web-формы и элементы управления еще до появления Web-сценариев и языка JavaScript. Но зачем?

Существует множество Web-сайтов, которые позволяют посетителю ввести какие-либо данные и получить результат их обработки: поисковые машины, почтовые Web-сервисы, интернет-магазины, интернет-опросники, социальные сети и пр.

Функциональность таких Web-сайтов реализуется с помощью особых программ, которые работают на серверном компьютере совместно с Web-сервером, — *серверных приложений*. Именно они обрабатывают полученные от посетителя Web-сайта данные и выдают результат в виде обычной Web-страницы. Именно для них в HTML предусмотрена возможность создания Web-форм и элементов управления — чтобы посетитель мог ввести данные, которые потом обработает серверное приложение.

Вот основная схема работы серверного приложения.

- Посетитель вводит в элементы управления, расположенные в Web-форме на Web-странице, нужные данные.
- Введя данные, посетитель нажимает расположенную в той же Web-форме особую кнопку — *кнопку отправки данных*.
- Web-форма кодирует введенные в нее данные и отправляет их серверному приложению, расположенному по указанному интернет-адресу.
- Web-сервер перехватывает отправленные данные, запускает серверное приложение и передает данные ему.
- Серверное приложение обрабатывает полученные данные.
- Серверное приложение формирует Web-страницу с результатами обработки данных посетителя и передает ее Web-серверу.
- Web-сервер получает сформированную серверным приложением Web-страницу и отправляет ее посетителю.

Для того чтобы успешно подготовить введенные посетителем данные и отправить их серверному приложению, Web-форма должна "знать" значения трех параметров.

- Интернет-адрес серверного приложения. Это обычный интернет-адрес, указывающий на файл серверного приложения, вида **http://www.somesite.ru/apps/app.exe**.
- Метод отправки данных, указывающий вид, в котором данные будут отправлены. Таких методов HTML поддерживает два.
  - Метод GET формирует из введенных посетителем данных набор пар вида `<имя элемента управления>=<введенные в него данные>`. (Ранее уже говорилось, что каждый элемент управления обязательно должен иметь уникальное в пределах Web-формы имя.) Эти пары добавляются справа к интернет-адресу серверного приложения, отделяясь от него символом `?` (вопросительный знак); сами пары разделяются символами `&` (амперсанд). Полученный таким образом интернет-адрес отправляется Web-серверу, который извлекает из него интернет-адрес серверного приложения и сами данные.
  - Метод POST также формирует из введенных данных пары вида `<имя элемента управления>=<введенные в него данные>`. Но отправляет он их не в составе интернет-адреса, а вслед за ним, в качестве дополнительных данных.
- Метод кодирования данных. Он актуален только при отправке данных методом POST; для метода GET его можно не указывать.

Все это имеет смысл только в том случае, если мы создаем Web-форму для отправки данных серверному приложению. Поскольку книга посвящена исключительно клиентским интернет-технологиям, мы не будем подробно рассматривать кодирование и пересылку данных. Эти сведения можно найти в любой книге по HTML, благо от версии к версии этого языка они практически не меняются.

Внимание!

Далее мы будем рассматривать только те возможности Web-форм и элементов управления, которые полезны исключительно при клиентском Web-программировании. Возможности, необходимые для взаимодействия с серверными приложениями, мы опустим.

Создание Web-форм и элементов управления

Настала пора рассмотреть средства языков HTML и CSS, предназначенные для создания Web-форм и элементов управления, и возможности объектов Web-обозревателя и библиотеки Ext Core для работы с ними. Их довольно много.

Создание Web-форм

Для создания Web-формы применяется парный тег `<FORM>`, внутри которого помещают теги, формирующие элементы управления, входящие в эту Web-форму:

```
<FORM>
  <теги, формирующие элементы управления>
</FORM>
```

Web-форма ведет себя как блочный элемент Web-страницы. (О блочных элементах см. *главу 2*.)

Тег `<FORM>` поддерживает обязательный атрибут `ACTION`, который указывает интернет-адрес серверного приложения. Если Web-форма служит для ввода данных, предназначенных для обработки Web-сценарием, в качестве значения этого атрибута тега указывают "пустой" интернет-адрес `#`:

```
<FORM ACTION="#">
  . . .
</FORM>
```

Создание элементов управления

Большинство элементов управления HTML создают посредством одинарного тега `<INPUT>`. Какой именно элемент управления следует создать, указывают с помощью необязательного атрибута `TYPE` этого тега. Некоторые элементы управления, такие как область редактирования и списки, создают с помощью других тегов. Мы обязательно их рассмотрим.

Все эти теги поддерживают уже знакомые нам атрибуты `ID`, `CLASS` и `STYLE`. Следовательно, мы можем дать элементу управления имя, по которому сможем получить к нему доступ из Web-сценария, привязать к нему именованный стиль или стилевой класс и задать для него встроенный стиль.

Ранее было сказано, что на основе данных, введенных в элементы управления, Web-форма, в которой эти элементы управления находятся, сформирует пары вида *<имя элемента управления>=<введенные в него данные>*, которые отправит серверно-

му приложению. Так вот, *имя элемента управления*, которое будет фигурировать в этих парах, задается атрибутом тега NAME — не ID! Это обязательный атрибут тега — если его не указать, при работе с элементом управления возможны проблемы.

Что касается атрибута тега ID, то он задает имя, под которым элемент управления будет доступен в Web-сценариях, а также имя именованного стиля. Собственно, об этом мы уже знаем.

Обычно для каждого элемента управления атрибутами тега ID и NAME указывают одно и то же имя — просто чтобы не ломать голову и устранить разнобой в именах. Хотя это и не обязательно.

Все элементы управления HTML представляют собой встроенные элементы Web-страницы (см. *главу 3*).

Теперь рассмотрим все элементы управления HTML и особенности их создания.

Поле ввода

Поле ввода — наиболее распространенный элемент управления в Web-формах — создается с помощью одинарного тега <INPUT>:

```
<INPUT [TYPE="text"] [VALUE="<изначальное значение>"] [SIZE="<размер>"]
[MAXLENGTH="<максимальное количество символов>"] [DISABLED]
[TABINDEX="<номер в порядке обхода>"] [ACCESSKEY="<быстрая клавиша>"]
[READONLY] [AUTOFOCUS]>
```

Атрибут тега TYPE, как уже говорилось, задает тип элемента управления. Значение "text" указывает Web-обозревателю создать именно поле ввода. Поле ввода также создается, если атрибут тега TYPE не указан (как уже говорилось, он необязательный).

Необязательный атрибут тега VALUE задает значение, которое должно присутствовать в поле ввода изначально. Если этот атрибут не указан, поле ввода не будет содержать ничего.

Необязательный атрибут тега SIZE задает длину поля ввода в символах. Если он не указан, длина поля ввода будет зависеть от Web-обозревателя.

Необязательный атрибут тега MAXLENGTH задает максимальный размер строки, которую можно ввести в это поле ввода, в символах. Если этот атрибут тега не указан, в поле ввода можно будет ввести строку неограниченного размера.

Необязательные атрибуты тега TABINDEX и ACCESSKEY задают, соответственно, номер в порядке обхода и "горячую" клавишу для доступа к элементу управления. Они знакомы нам по гиперссылкам (см. *главу 6*).

Атрибут тега без значения DISABLED позволяет сделать поле ввода недоступным для посетителя; оно будет отображаться серым цветом, и посетитель не сможет даже его активизировать. Если этот атрибут присутствует в теге, поле ввода недоступно, если отсутствует — доступно.

Атрибут тега без значения READONLY позволяет сделать поле ввода доступным только для чтения; при этом посетитель все-таки сможет активизировать это поле, вы-

делить содержащийся в нем текст и скопировать его в Буфер обмена. Если этот атрибут тега присутствует, поле ввода будет доступно только для чтения, если отсутствует — доступно и для чтения, и для ввода.

Если атрибут тега без значения AUTOFOCUS присутствует, данное поле ввода будет автоматически активизировано при открытии Web-страницы. Если же он отсутствует, поле ввода активизировано не будет и посетителю придется его активизировать щелчком мышью или клавишами <Tab> или <Shift>+<Tab>.

> ***Внимание!***
>
> Атрибут тега AUTOFOCUS можно указывать только для одного элемента управления на всей Web-странице.

Листинг 20.1

```
<FORM ACTION="#">
  <P>Имя: <INPUT TYPE="text" ID="name1" NAME="name1" SIZE="20"
  AUTOFOCUS></P>
  <P>Фамилия: <INPUT TYPE="text" ID="name2" NAME="name2" SIZE="30"></P>
</FORM>
```

В листинге 20.1 мы создаем Web-форму с двумя полями ввода: name1 длиной 20 символов, автоматически активизирующееся при открытии Web-страницы, и name2 длиной 30 символов. Оба поля ввода имеют надписи, представляющие собой обычный текст и расположенные перед ними.

Обратим внимание, что для размещения элементов управления в Web-форме мы использовали абзацы. Вообще, для этого можно применять любые элементы Web-страниц из уже знакомых нам: списки, таблицы, контейнеры и пр.

Поле ввода пароля

Поле ввода пароля ничем не отличается от обычного поля ввода за тем исключением, что вместо вводимых символов в нем отображаются точки. Такие поля ввода широко применяют для запроса паролей и других конфиденциальных данных.

Поле ввода пароля также создается с помощью одинарного тега <INPUT>:

```
<INPUT TYPE="password" [VALUE="<изначальное значение>"]
[SIZE="<размер>"] [MAXLENGTH="<максимальное количество символов>"]
[TABINDEX="<номер в порядке обхода>"] [ACCESSKEY="<быстрая клавиша>"]
[DISABLED] [READONLY] [AUTOFOCUS]>
```

Значение "password" атрибута тега TYPE указывает Web-обозревателю создать поле ввода пароля. Остальные атрибуты нам уже знакомы по обычному полю ввода.

Листинг 20.2

```
<FORM ACTION="#">
  <P>Имя: <INPUT TYPE="text" ID="login" NAME="login" SIZE="20"
```

```
  AUTOFOCUS></P>
  <P>Пароль: <INPUT TYPE="password" ID="password" NAME="password"
  SIZE="20"></P>
</FORM>
```

В листинге 20.2 мы создаем Web-форму с обычным полем ввода и полем ввода пароля. Первое — `login`, длиной 20 символов, будет автоматически активизироваться при открытии Web-страницы. Второе — `password`, длиной также 20 символов.

Поле ввода значения для поиска

Поле ввода значения для поиска появилось в HTML 5. Оно ничем не отличается от обычного поля ввода за тем исключением, что из введенного в него значения автоматически удаляются переводы строк.

Поле ввода значения для поиска также создается с помощью одинарного тега `<INPUT>`:

```
<INPUT TYPE="search" [VALUE="<изначальное значение>"]
[SIZE="<размер>"] [MAXLENGTH="<максимальное количество символов>"]
[TABINDEX="<номер в порядке обхода>"] [ACCESSKEY="<быстрая клавиша>"]
[DISABLED] [READONLY] [AUTOFOCUS]>
```

Значение `"search"` атрибута тега `TYPE` указывает Web-обозревателю создать поле ввода значения для поиска. Остальные атрибуты нам уже знакомы по обычному полю ввода (листинг 20.3).

Листинг 20.3

```
<FORM ACTION="#">
  <P>Найти: <INPUT TYPE="search" ID="keyword" NAME="keyword"
  SIZE="40"></P>
</FORM>
```

Область редактирования

Область редактирования создается парным тегом `<TEXTAREA>`:

```
<TEXTAREA [ROWS="<высота>"] [COLS="<ширина>"] [WRAP="off|soft|hard"]
[TABINDEX="<номер в порядке обхода>"] [ACCESSKEY="<быстрая клавиша>"]
[DISABLED] [READONLY] [AUTOFOCUS]>
  <изначальное значение>
</TEXTAREA>
```

Значение, которое должно изначально присутствовать в области редактирования, помещается внутрь тега `<TEXTAREA>`. Это должен быть текст без всяких HTML-тегов.

Необязательный атрибут тега `ROWS` задает высоту области редактирования в строках. Если он не указан, высота области редактирования будет зависеть от Web-обозревателя.

Необязательный атрибут тега COLS задает ширину области редактирования в символах. Если он не указан, высота области редактирования будет зависеть от Web-обозревателя.

Необязательный атрибут тега WRAP позволяет управлять переносом строк в области редактирования. Атрибут WRAP может принимать два значения:

- "soft" — область редактирования будет автоматически выполнять перенос слишком длинных строк. При этом в само значение, введенное в область редактирования, символы перевода строк вставляться не будут.
- "hard" — область редактирования будет автоматически выполнять перенос слишком длинных строк. При этом в соответствующие места значения, введенного в область редактирования, будут вставлены символы перевода строк.

Если атрибут тега WRAP не указан, область редактирования будет вести себя так, словно задано значение "soft".

Остальные атрибуты, поддерживаемые тегом <TEXTAREA>, нам уже знакомы (листинг 20.4).

Листинг 20.4

```
<FORM ACTION="#">
  <P>
    Введите сюда ваш отзыв о Web-сайте:<BR>
    <TEXTAREA ID="opinion" NAME="opinion" COLS="60" ROWS="10">
      Отличный Web-сайт!
    </TEXTAREA>
  </P>
</FORM>
```

Кнопка

Кнопка при нажатии запускает на выполнение какое-либо действие. Она создается с помощью тега <INPUT>:

```
<INPUT TYPE="button" VALUE="<надпись>"
[TABINDEX="<номер в порядке обхода>"] [ACCESSKEY="<быстрая клавиша>"]
[DISABLED] [AUTOFOCUS]>
```

Значение "button" атрибута тега TYPE указывает Web-обозревателю создать обычную кнопку. Атрибут тега VALUE, задающий надпись для кнопки, в этом случае является обязательным. Остальные атрибуты тега нам уже знакомы (листинг 20.5).

Листинг 20.5

```
<FORM ACTION="#">
  <P>
    Найти:
    <INPUT TYPE="search" ID="keyword" NAME="keyword" SIZE="40">
```

```
    <INPUT TYPE="button" ID="find" NAME="find" VALUE="Искать!">
  </P>
</FORM>
```

Флажок

Флажки встречаются в Web-формах нечасто, в случаях, когда нужно дать посетителю возможность выбрать или не выбрать какую-то опцию. Для создания флажков применяется тег <INPUT>:

```
<INPUT TYPE="checkbox" [CHECKED]
[TABINDEX="<номер в порядке обхода>"] [ACCESSKEY="<быстрая клавиша>"]
[DISABLED] [AUTOFOCUS]>
```

Значение "checkbox" атрибута тега TYPE указывает Web-обозревателю создать именно флажок.

Атрибут тега без значения CHECKED позволяет сделать флажок изначально установленным. Если он присутствует, флажок будет установлен изначально, если отсутствует — сброшен.

Остальные атрибуты тега нам уже знакомы (листинг 20.6).

Листинг 20.6

```
<FORM ACTION="#">
  <P>
    <INPUT TYPE="checkbox" ID="updates" NAME="updates" CHECKED>
    Я хочу получать письма со списком обновлений Web-сайта
  </P>
</FORM>
```

Переключатель

Переключатели в Web-формах, как и в окнах Windows-приложений, применяются только группами. Группа переключателей предоставляет посетителю возможность выбрать одну из нескольких доступных альтернатив. В одиночку же переключатели абсолютно бесполезны — флажки в таких случаях гораздо удобнее.

А теперь — очень важная вещь! Ранее мы говорили, что каждый элемент управления должен иметь уникальное в пределах Web-формы имя, задаваемое атрибутом тега NAME. Это имя необходимо для формирования данных, отсылаемых серверному приложению.

Но из этого правила есть исключение — переключатели. В их случае атрибут тега NAME задает имя группы переключателей. Иными словами, переключатели, входящие в одну группу, должны иметь одинаковое имя, заданное атрибутом тега NAME, и данное имя должно быть уникально в пределах формы.

Тем не менее, имена переключателей, задаваемые атрибутом тега ID, могут быть разными. Это позволит нам получать доступ из Web-сценария к отдельным переключателям группы и проверять, установлены они или сброшены.

Создается переключатель с помощью все того же тега <INPUT>:

```
<INPUT TYPE="radio" [CHECKED]
[TABINDEX="<номер в порядке обхода>"] [ACCESSKEY="<быстрая клавиша>"]
[DISABLED] [AUTOFOCUS]>
```

Значение "radio" атрибута тега TYPE указывает Web-обозревателю создать именно переключатель. Остальные атрибуты тега нам уже знакомы.

В группе только один переключатель может быть установлен. Это значит, что атрибут тега без значения CHECKED можно указывать только для одного переключателя в группе. Листинг 20.7 содержит пример переключателя.

Листинг 20.7

```
<FORM ACTION="#">
  <P>
    <INPUT TYPE="radio" ID="updates_yes" NAME="updates" CHECKED>
    Я хочу получать письма со списком обновлений Web-сайта
  </P>
  <P>
    <INPUT TYPE="radio" ID="updates_no" NAME="updates">
    Я не хочу получать письма со списком обновлений Web-сайта
  </P>
</FORM>
```

Список, обычный или раскрывающийся

Списки, как обычные, так и раскрывающиеся, реализуют с помощью парного тега <SELECT>, внутри которого помещают парные теги <OPTION>, создающие пункты списка.

Начнем с парного тега <SELECT>, который создает сам список:

```
<SELECT [SIZE="<высота в пунктах (позициях)>"] [MULTIPLE]
[TABINDEX="<номер в порядке обхода>"] [ACCESSKEY="<быстрая клавиша>"]
[DISABLED] [AUTOFOCUS]>
  <теги <OPTION>, создающие пункты списка>
</SELECT>
```

Необязательный атрибут тега SIZE задает высоту списка в пунктах (имеются в виду пункты списка). Если этот атрибут тега имеет значение, отличное от единицы, создается обычный список, имеющий высоту, равную указанному значению пунктов. Если же значение этого атрибута тега равно единице или вообще отсутствует, создается раскрывающийся список (имеющий в высоту один пункт).

Атрибут тега без значения MULTIPLE позволяет создать список, в котором можно выбрать сразу несколько пунктов. Если этот атрибут тега присутствует, в списке можно будет выбрать сразу несколько пунктов, если отсутствует — можно будет выбрать только один пункт. Понятно, что данный атрибут тега имеет смысл указы-

вать только для обычных списков (если атрибут тега SIZE имеет значение, отличное от 1); в раскрывающемся списке в любом случае можно выбрать только один пункт.

Остальные атрибуты этого тега нам уже знакомы.

Парный тег <OPTION> создает отдельный пункт списка. Он может присутствовать только в теге <SELECT>:

```
<OPTION [LABEL="<текст пункта>"] [SELECTED] [DISABLED]>
  [<текст пункта>]
<OPTION>
```

Текст пункта списка либо помещают внутрь тега <OPTION>, либо указывают с помощью атрибута тега LABEL.

Атрибут тега без значения SELECTED позволяет сделать данный пункт списка изначально выбранным. Если этот атрибут тега указан, пункт списка будет изначально выбранным, если не указан — не будет выбранным.

Уже знакомый нам атрибут тега без значения DISABLED позволяет сделать данный пункт недоступным для выбора. Листинг 20.8 иллюстрирует пример.

Листинг 20.8

```
<FORM ACTION="#">
  <P>
    Выполнять поиск по
    <SELECT ID="search_in" NAME="search_in">
      <OPTION>названиям</OPTION>
      <OPTION>ключевым словам</OPTION>
      <OPTION SELECTED>названиям и ключевым словам</OPTION>
    </SELECT>
  </P>
</FORM>
```

HTML также позволяет объединять пункты списка в группы по какому-либо родственному признаку. Такую группу создают с помощью парного тега <OPTGROUP>; в него помещают теги, создающие пункты списка, которые входят в эту группу. Тег <OPTGROUP> может присутствовать только внутри тега <SELECT>:

```
<OPTGROUP LABEL="<заголовок группы>" [DISABLED]>
  <теги <OPTION>, создающие пункты, которые входят в группу>
<OPTGROUP>
```

Обязательный в этом случае атрибут тега LABEL задает заголовок группы. А атрибут тега без значения DISABLED позволяет сделать все пункты данной группы недоступными для выбора (листинг 20.9).

Листинг 20.9

```
<FORM ACTION="#">
  <P>
    Выполнять поиск по
    <SELECT ID="search_in" NAME="search_in">
      <OPTGROUP LABEL="Быстрый поиск">
        <OPTION>названиям</OPTION>
        <OPTION>ключевым словам</OPTION>
      </OPTGROUP>
      <OPTION SELECTED>названиям и ключевым словам</OPTION>
    </SELECT>
  </P>
</FORM>
```

Надпись

Строго говоря, *надпись* — это не элемент управления. Она просто задает для элемента управления текстовую надпись, которая описывает его назначение. Если посетитель щелкнет мышью на надписи, элемент управления будет активизирован.

Надпись создают с помощью парного тега <LABEL>:

```
<LABEL [FOR="<имя элемента управления, к которому относится надпись>"]
[TABINDEX="<номер в порядке обхода>"] [ACCESSKEY="<быстрая клавиша>"]>
  <текст надписи>[<элемент управления>]
</LABEL>
```

Есть два способа привязать надпись к элементу управления, который она должна описывать. Сейчас мы их рассмотрим.

При первом способе (листинг 20.10) элементу управления, к которому привязывается надпись, дают имя с помощью атрибута тега ID. (Впрочем, любой элемент управления должен иметь имя.) Это имя указывают в качестве значения обязательного в таком случае атрибута FOR тега <LABEL>, создающего надпись.

Листинг 20.10

```
<FORM ACTION="#">
  <P><LABEL FOR="keyword">Найти:</LABEL>
  <INPUT TYPE="search" ID="keyword" NAME="keyword"
  SIZE="40"></P>
</FORM>
```

При втором способе (листинг 20.11) элемент управления, к которому привязывается надпись, помещают в сам тег <LABEL>, создающий ее, сразу после текста надписи.

Листинг 20.11

```
<FORM ACTION="#">
  <P><LABEL>Найти: <INPUT TYPE="search" ID="keyword" NAME="keyword"
  SIZE="40"></LABEL></P>
</FORM>
```

Надписи в Web-формах встречаются довольно редко. Обычно Web-дизайнеры ограничиваются простым текстом, который ставят до или после элемента управления.

Группа

Группу также нельзя отнести к "настоящим" элементам управления. Она объединяет несколько элементов управления, имеющих сходное назначение. Визуально группа представляет собой рамку, окружающую элементы управления и, возможно, имеющую заголовок, расположенный прямо на ее верхней или нижней границе.

Группу создают с помощью парного тега <FIELDSET>:

```
<FIELDSET>
  <элементы управления, объединяемые в группу>
</FIELDSET>
```

Видно, что теги, создающие элементы управления, которые должны быть объединены в группу, помещают прямо в тег <FIELDSET>.

Кроме того, в теге <FIELDSET> может присутствовать парный тег <LEGEND>, создающий заголовок группы:

```
<LEGEND [ACCESSKEY="<быстрая клавиша>"]><текст заголовка></LEGEND>
```

Текст заголовка помещают прямо внутри этого тега.

Тег <LEGEND> должен помещаться либо сразу же после открывающего тега <FIELDSET>, либо перед закрывающим тегом </FIELDSET>. В первом случае заголовок будет присутствовать на верхней границе группы, во втором случае — на нижней границе. В листинге 20.12 приведен пример группы.

Листинг 20.12

```
<FORM ACTION="#">
  <FIELDSET>
    <LEGEND>Найти:</LEGEND>
    <P>
      <INPUT TYPE="search" ID="keyword" NAME="keyword" SIZE="40">
      <INPUT TYPE="button" ID="find" NAME="find" VALUE="Искать!">
    </P>
  </FIELDSET>
</FORM>
```

Прочие элементы управления

HTML позволяет создать еще несколько элементов управления, которые необходимы только для взаимодействия с серверными приложениями. Если же Web-форма служит для ввода данных, предназначенных для обработки Web-сценарием, эти элементы управления не имеют смысла.

Прежде всего, это кнопка отправки данных, о которой мы уже говорили в начале главы. Она отличается от обычной кнопки только значением атрибута TYPE тега <INPUT> — "submit".

Далее, в Web-форме может присутствовать *кнопка очистки*. При нажатии на такую кнопку все элементы управления в Web-форме получают изначальные значения, заданные в HTML-коде. Значение атрибута TYPE тега <INPUT>, создающего подобную кнопку, должно быть "reset".

Поле ввода имени файла служит для указания имени файла, который будет отправлен серверному приложению (сам файл, а не его имя). Оно состоит из собственно поля ввода и расположенной правее его кнопки **Обзор**, при нажатии которой на экране появится стандартное диалоговое окно открытия файла Windows, в котором можно выбрать отправляемый файл.

Поле ввода имени файла отличается от обычного поля ввода значением атрибута TYPE тега <INPUT> — "file". В теге <INPUT> в этом случае поддерживаются атрибуты ACCESSKEY, AUTOFOCUS, DISABLED, SIZE и TABINDEX.

Графическая кнопка отправки данных — это графическое изображение, при щелчке на котором Web-форма запускает процесс отправки введенных данных серверному приложению. Фактически это кнопка отправки данных, в качестве которой выступает изображение.

Графическую кнопку отправки данных создают с помощью тега <INPUT>. Значение атрибута TYPE этого тега должно быть "image". Атрибут тега SRC задает интернет-адрес файла с графическим изображением, а атрибут тега ALT — текст замены (подробнее см. в *главе 4*). Также поддерживаются атрибуты ACCESSKEY, AUTOFOCUS, DISABLED и TABINDEX тега <INPUT>.

Скрытое поле — это фактически вообще не элемент управления, поскольку никак не отображается на Web-странице. Оно служит для хранения каких-либо служебных данных, необходимых для серверного приложения, показ которых посетителю нежелателен.

Скрытое поле создают с помощью тега <INPUT>. Значение атрибута TYPE этого тега должно быть "hidden". Атрибут VALUE тега <INPUT> задает хранимое в скрытом поле значение.

Специальные селекторы CSS, предназначенные для работы с элементами управления

Язык CSS предоставляет несколько специальных селекторов, с помощью которых можно неявно привязать стиль к элементам управления на основе их состояния. Все они относятся к структурным псевдоклассам.

- `:enabled` — привязывает стиль к элементам управления, доступным для посетителя.
- `:disabled` — привязывает стиль к элементам управления, недоступным для посетителя.
- `:checked` — привязывает стиль к установленным флажкам и переключателям.

Листинг 20.13 иллюстрирует пример.

Листинг 20.13

```
:disabled { color: #B1BEC6 }
:checked { font-weight: bold }
. . .
<FORM ACTION="#">
  <P>
    <INPUT TYPE="radio" ID="updates_yes" NAME="updates" CHECKED>
    Я хочу получать письма со списком обновлений Web-сайта
  </P>
  <P>
    <INPUT TYPE="radio" ID="updates_no" NAME="updates" CHECKED >
    Я не хочу получать письма со списком обновлений Web-сайта
  </P>
  <P>Почтовый адрес: <INPUT TYPE="text" ID="email" NAME="email"
  DISABLED></P>
</FORM>
```

Работа с элементами управления

Толку от Web-формы немного, если вводимые в ней данные никак не обрабатываются. Поскольку мы занимаемся исключительно клиентскими интернет-технологиями, обрабатывать данные мы будем в Web-сценариях.

А чтобы обработать в Web-сценариях данные, введенные в элементы управления, мы должны их как-то получить оттуда. Кроме того, нам пригодится возможность манипулировать элементами управления из Web-сценариев: делать их доступными и недоступными, устанавливать и сбрасывать флажки, включать переключатели, выбирать пункты списков и пр. И, поскольку львиная доля Web-сценариев — это обработчики событий, мы должны знать, какие события поддерживают элементы управления и когда они возникают.

Вот об этом и пойдет сейчас разговор.

Свойства и методы объекта *HTMLElement*, применяемые для работы с элементами управления

Сначала мы рассмотрим самые полезные для нас свойства и методы объектов Web-обозревателя, представляющих различные элементы управления. Запомним: это именно объекты Web-обозревателя, производные от объекта `HTMLElement`.

Свойство disabled позволяет сделать элемент управления доступным или недоступным для посетителя. Значение true этого свойства делает элемент управления доступным, значение false — недоступным. Листинг 20.14 иллюстрирует пример.

Листинг 20.14

```
<FORM ACTION="#">
  <P>
    <INPUT TYPE="checkbox" ID="updates" NAME="updates">
    Я хочу получать письма со списком обновлений Web-сайта
  </P>
  <P>Почтовый адрес: <INPUT TYPE="text" ID="email" NAME="email"></P>
</FORM>
. . .
Ext.getDom("email").disabled = false;
```

Здесь мы с помощью метода getDom получаем экземпляр объекта HTMLElement, представляющий поле ввода почтового адреса email, и делаем его недоступным для ввода, присвоив свойству disabled значение false.

Свойство readOnly позволяет сделать элемент управления доступным или недоступным для ввода. Значение true этого свойства делает элемент управления недоступным для ввода, значение false — доступным:

```
Ext.getDom("email").readOnly = false;
```

Свойство value задает или возвращает значение, введенное в поле ввода или область редактирования, в виде строки:

```
var sEmail = Ext.getDom("email").value;
```

Свойство checked позволяет получить или задать состояние флажка или переключателя — установлен он или нет. Значение true обозначает, что флажок или переключатель установлен, значение false — сброшен:

```
Ext.get("updates").on("click", function() {
  var htelEmail = Ext.getDom("email");
  htelEmail.disabled = this.checked;
});
```

Здесь мы привязываем к флажку updates функцию — обработчик события click, которую тут же и объявляем. Эта функция делает доступным для посетителя поле ввода email, если флажок установлен, и недоступным — если он сброшен. Наша задача упрощается тем, что переменная this, доступная в теле функции-обработчика события и хранящая элемент Web-страницы, в котором обрабатывается событие, хранит этот элемент в виде экземпляра объекта HTMLElement. Спасибо разработчикам Ext Core!

Еще один пример приведен в листинге 20.15.

Листинг 20.15

```
<FORM ACTION="#">
  <P>
    <INPUT TYPE="radio" ID="updates_yes" NAME="updates" CHECKED>
    Я хочу получать письма со списком обновлений Web-сайта
  </P>
  <P>
    <INPUT TYPE="radio" ID="updates_no" NAME="updates">
    Я не хочу получать письма со списком обновлений Web-сайта
  </P>
  <P>Почтовый адрес: <INPUT TYPE="text" ID="email" NAME="email"></P>
</FORM>
. . .
Ext.get("updates_yes").on("click", function() {
  var htelEmail = Ext.getDom("email");
  htelEmail.disabled = this.checked;
});
```

В листинге 20.15 мы выполняем аналогичные действия, но уже с группой из двух переключателей `updates2`. Обратим внимание, что мы проверяем состояние только первого переключателя этой группы — `updates_yes`. В группе может быть включен только один переключатель, и если посетитель включит второй переключатель этой группы, первый переключатель отключится. Фактически группа из двух переключателей ведет себя как флажок.

Свойство `selectedIndex` задает или возвращает номер выбранного в списке пункта в виде числа. При этом:

- если список позволяет выбирать одновременно только один пункт, возвращается номер именно этого пункта;
- если список позволяет выбирать сразу несколько пунктов, возвращается номер первого выбранного пункта;
- если ни один пункт в списке не выбран, возвращается значение –1.

Понятно, что пользы от свойства `selectedIndex` будет больше в том случае, если список позволяет выбирать только один пункт одновременно. Хотя в любом случае его можно применять для проверки, выбран ли в списке хоть один пункт. Листинг 20.16 иллюстрирует пример.

Листинг 20.16

```
<FORM ACTION="#">
  <P>
    Выполнять поиск по
    <SELECT ID="search_in" NAME="search_in">
      <OPTION>названиям</OPTION>
      <OPTION>ключевым словам</OPTION>
```

```
      <OPTION SELECTED>названиям и ключевым словам</OPTION>
    </SELECT>
  </P>
</FORM>
. . .
var iIndex = Ext.getDom("search_in").selectedIndex;
if (iIndex == -1) {
  //если в списке не выбран ни один пункт, делаем одно
} else {
  //если в списке выбран какой-либо пункт, делаем другое
}
```

Свойство `options` возвращает коллекцию пунктов списка. Эта коллекция является экземпляром объекта `HTMLOptionsCollection`:

```
var clItems = Ext.getDom("search_in").options;
```

Свойство `length` объекта `HTMLOptionsCollection` возвращает число элементов в коллекции, т. е. количество пунктов в списке:

```
var iItemsCount = clItems.length;
```

Для доступа к отдельным пунктам в этой коллекции мы можем использовать числовые индексы, как и в случае массива:

```
var htelSecondItem = clItems[1];
```

Здесь мы получаем второй пункт списка.

Отдельный пункт списка представляется экземпляром объекта `HTMLOptionElement`. Он поддерживает уже знакомое нам свойство `disabled`, позволяющее разрешить или запретить доступ к данному пункту списка.

А еще он поддерживает свойство `selected`, указывающее, выбран ли данный пункт списка. Значение `true` обозначает, что пункт списка выбран, а значение `false` — не выбран. Это свойство удобно применять, чтобы выяснить, какие пункты выбраны в списке, позволяющем выбирать сразу несколько пунктов (листинг 20.17).

Листинг 20.17

```
<FORM ACTION="#">
  <P>
    С помощью каких тегов HTML формируется таблица?
    <SELECT ID="answer" NAME="answer" SIZE="5" MULTIPLE>
      <OPTION>TR</OPTION>
      <OPTION>DIV</OPTION>
      <OPTION>TABLE</OPTION>
      <OPTION>TH</OPTION>
      <OPTION>TT</OPTION>
      <OPTION>HEAD</OPTION>
      <OPTION>TD</OPTION>
    </SELECT>
```

```
  </P>
</FORM>
. . .
var clItems = Ext.getDom("answer").options;
if ((clItems[0].selected) && (clItems[2].selected)
&& (clItems[3].selected) && (clItems[6].selected)) {
  var s = "Вы ответили правильно!";
} else {
  var s = "Неправильно! Будьте внимательнее.";
}
```

В листинге 20.17 мы создали что-то наподобие онлайнового экзамена. Посетителю требуется выбрать в списке `answer` пункты, представляющие теги HTML, с помощью которых создаются таблицы. Если все эти пункты выбраны, ответ считается правильным.

Свойство `form` возвращает экземпляр объекта `HTMLElement`, представляющий Web-форму, в которой находится данный элемент управления:

```
var htelForm = Ext.getDom("answer").form;
```

Метод `focus` делает данный элемент управления активным. Он не принимает параметров и не возвращает результата:

```
Ext.getDom("email").focus();
```

Метод `blur` делает данный элемент управления, наоборот, неактивным; при этом фокус ввода переносится на следующий в порядке обхода элемент управления. Данный метод также не принимает параметров и не возвращает результата:

```
Ext.getDom("email").blur();
```

Метод `select` выделяет все содержимое поля ввода или области редактирования. Он не принимает параметров и не возвращает результата:

```
Ext.getDom("email").select();
```

Метод `click` позволяет имитировать щелчок на кнопке. Он не принимает параметров и не возвращает результата (листинг 20.18).

Листинг 20.18

```
<FORM ACTION="#">
  <P>
    Найти:
    <INPUT TYPE="search" ID="keyword" NAME="keyword" SIZE="40">
    <INPUT TYPE="button" ID="find" NAME="find" VALUE="Искать!">
  </P>
</FORM>
. . .
Ext.getDom("find").click();
```

Свойства и методы объекта *Element*, применяемые для работы с элементами управления

А теперь обратимся к объекту `Element` библиотеки Ext Core и посмотрим, что он может предложить нам для работы с элементами управления.

Метод `getValue` возвращает значение, введенное в поле ввода или область редактирования, в виде строки или числа:

```
<экземпляр объекта Element>.getValue(<преобразовать в число>)
```

Если этому методу передать в качестве параметра значение `false`, он вернет значение поля ввода или области редактирования в виде строки. Если же ему передать значение `true`, он попытается преобразовать это значение в число и в случае успеха вернет его; в противном случае он вернет это значение в виде строки:

```
var sEmail = Ext.get("email").getValue(false);
```

Метод `focus` делает данный элемент управления активным. Он не принимает параметров и не возвращает результата. Если вызвать этот метод у элемента Web-страницы, не являющимся элементом управления, ничего не произойдет:

```
Ext.get("email").focus();
```

Метод `blur` делает данный элемент управления неактивным; при этом фокус ввода переносится на следующий в порядке обхода элемент управления:

```
Ext.get("email").blur();
```

Данный метод также не принимает параметров и не возвращает результата. Если вызвать его у элемента Web-страницы, не являющимся элементом управления, ничего не произойдет.

Метод `select` поддерживает еще один селектор — `:checked`. Он соответствует всем установленным флажкам и переключателям:

```
var clChecked = Ext.get("cmain").select(":checked");
```

События элементов управления

Специфические события, поддерживаемые элементами управления, перечислены в табл. 20.1. Их немного.

Таблица 20.1. *Специфические события, поддерживаемые элементами управления*

Событие	Описание
`blur`	Возникает, когда элемент управления теряет фокус ввода. Не всплывает. Действие по умолчанию — потеря элементом управления фокуса ввода, отменить его невозможно
`change`	Возникает при изменении значения в поле ввода или области редактирования. Не всплывает. Действие по умолчанию — изменение значения, может быть отменено

Таблица 20.1 *(окончание)*

Событие	Описание
`click`	Возникает при щелчке левой кнопкой мыши на поле ввода и области редактирования, нажатии кнопки, установке или сбросе флажка и переключателя и выборе пункта списка. Всплывает. Действие по умолчанию зависит от конкретного элемента управления, может быть отменено
`focus`	Возникает, когда элемент управления получает фокус ввода. Не всплывает. Действие по умолчанию — получение элементом управления фокуса ввода, отменить его невозможно
`select`	Возникает при выделении посетителем значения или его части в поле ввода или области редактирования. Всплывает. Поведение по умолчанию — выделение значения, может быть отменено

Элементы управления также поддерживают события `dblclick`, `keydown`, `keypress`, `keyup`, `mousedown`, `mousemove`, `mouseout`, `mouseover` и `mouseup`, описанные в табл. 15.1.

Реализация поиска на Web-сайте

Теоретическая часть, посвященная Web-формам и элементам управления, закончена. Давайте попрактикуемся.

Для практики мы реализуем давно задуманное — поиск на нашем Web-сайте. Поиск будет осуществляться на основе информации, хранящейся в базе данных, которую мы создали еще в *главе 18*. База данных — вещь универсальная и может пригодиться для многих дел. Мы уже убедились в этом, когда в *главе 19* создавали раздел "См. также" у Web-страниц, куда поместили связанные с ними материалы.

Чтобы усложнить себе задачу и упростить жизнь посетителям, мы реализуем поиск, во-первых, по названиям Web-страниц, во-вторых, по ключевым словам, связанным с каждой Web-страницей. *Ключевым словом* в данном случае называется специальным образом подобранное кодовое слово, характеризующее конкретный материал. Скажем, для материала, рассказывающего о теге `<AUDIO>`, ключевыми словами будут "мультимедиа" и "аудио", поскольку он описывает способ размещения на Web-страницах **аудио**роликов, относящихся к **мультимедийным** материалам.

Далее, мы предоставим посетителю возможность выбирать критерии поиска: только по названиям, только по ключевым словам или и по названиям, и по ключевым словам. Для этого мы используем раскрывающийся список, т. к. он занимает немного места на Web-странице и вполне информативен.

Что касается самого поиска, то реализовать его несложно. Достаточно просмотреть все три массива, составляющие нашу базу данных, найти элементы, содержащие название или ключевое слово, совпадающее с введенным посетителем значением, и скопировать их в другой массив, который будет хранить результаты поиска. Потом на основе полученного массива мы сформируем, скажем, список, пункты которого будут представлять собой гиперссылки на соответствующие Web-страницы, и вставим его в контейнер `cmain`. Почти как в случае раздела "См. также".

Еще мы предусмотрим ситуацию, когда посетитель введет не все искомое название или ключевое слово, а только его начало. Соответственно, Web-сценарий, который мы напишем, будет искать элементы базы данных, начало названия или одного из ключевых слов которого совпадает с тем, что ввел посетитель.

Что ж, основной план работ мы набросали, а детали будем прояснять по ходу дела. Начнем с базы данных.

Подготовка базы данных

Нам потребуется указать для каждого элемента массива, составляющего базу данных и представляющего одну из Web-страниц, список ключевых слов. Для этого мы создадим у конфигураторов — элементов этих массивов новое свойство `keyword`, которому и присвоим список соответствующих ключевых слов. Он будет представлять собой обычную строку с ключевыми словами, разделенными запятыми, — так его проще обрабатывать.

Откроем файл Web-сценариев data.js и поместим после кода, создающего свойство `related` со связанными данными, но перед кодом, выполняющим сортировку базы, такое выражение:

```
aHTML[0].keyword =  "тип,версия";
```

Мы взяли первый элемент массива `aHTML` (с индексом 0), добавили к хранящемуся в нем конфигуратору свойство `keyword` и присвоили этому свойству строку с ключевыми словами "тип" и "версия". Следовательно, мы указали, что Web-страницу с описанием тега `<!DOCTYPE>` будут характеризовать эти два ключевых слова.

Аналогично укажем ключевые слова для остальных Web-страниц нашего Web-сайта. Необязательно для всех — хотя бы для нескольких, чтобы только проверить поиск в работе.

Создание Web-формы

На очереди — Web-форма, в которую посетитель будет вводить искомое слово или его часть. Вот только куда ее поместить? Давайте пока что вставим ее в контейнер `cnavbar`, ниже полосы навигации, непосредственно перед закрывающим тегом `</DIV>`, формирующим этот контейнер. В *главе 21* мы найдем Web-форме поиска местоположение получше.

Наша первая "рабочая" Web-форма будет содержать следующие элементы:

- надпись "Поиск", чтобы посетитель сразу понял, зачем нужна эта Web-форма;
- поле ввода значения для поиска, где указывается искомое слово или начало слова;
- кнопку, запускающую поиск;
- раскрывающийся список для выбора режима поиска (только по названиям, только по ключевым словам или одновременно по названиям и по ключевым словам).

Содержимое Web-формы мы поместим в один абзац, который разобьем на три строки с помощью тегов разрыва строк (см. *главу 3*). Первая строка будет содержать надпись, вторая — поле ввода и кнопку, третья — раскрывающийся список. Можно, конечно, разместить эти элементы в трех отдельных абзацах, но так Web-форма займет на Web-странице слишком много места.

Поле ввода искомого слова мы назовем `keyword`, кнопку — `find`, а раскрывающийся список — `search_in`.

Листинг 20.19 содержит HTML-код, создающий Web-форму.

Листинг 20.19

```
<FORM ACTION="#">
  <P>
    Поиск:<BR>
    <INPUT TYPE="search" ID="keyword" NAME="keyword" SIZE="20">
    <INPUT TYPE="button" ID="find" NAME="find" VALUE="Искать!"><BR>
    <SELECT ID="search_in" NAME="search_in">
      <OPTION>В названиях</OPTION>
      <OPTION>В ключевых словах</OPTION>
      <OPTION SELECTED>В названиях и ключевых словах</OPTION>
    </SELECT>
  </P>
</FORM>
```

Вставим его в соответствующее место файла фрагмента html.htm.

Обязательно проверим, правильно ли наша Web-форма отображается на Web-странице. Если мы допустили в HTML-коде ошибку, лучше исправить ее прямо сейчас.

Написание Web-сценария, выполняющего поиск

Осталось написать Web-сценарий, который будет искать Web-страницы, удовлетворяющие заданным посетителем условиям.

Откроем файл Web-сценария main.js и поместим где-либо в теле функции, передаваемой в качестве параметра методу `onReady` объекта `Ext`, такое выражение:

```
Ext.get("find").on("click", searchData);
```

Оно привязывает к событию `click` кнопки `find` функцию-обработчик `searchData`, которая будет выполнять поиск и выводить его результаты и которую мы объявим чуть позже. Кнопка `find` созданной нами Web-формы запускает процесс поиска, а событие `click`, как мы уже знаем, возникает при щелчке на кнопке.

Теперь объявим функцию `searchData`. Она не будет ни принимать параметры, ни возвращать результат. Объявляющий ее код (листинг 20.20) поместим где-либо перед вызовом методу `onReady` объекта `Ext`.

Листинг 20.20

```
function searchData() {
  var sKeyword = Ext.get("keyword").getValue(false);
  if (sKeyword != "") {
    var iSearchMode = Ext.getDom("search_in").selectedIndex;
    var aResult = [];
    searchInArray(sKeyword, aHTML, aResult, iSearchMode);
    searchInArray(sKeyword, aCSS, aResult, iSearchMode);
    searchInArray(sKeyword, aSamples, aResult, iSearchMode);
    if (aResult.length > 0) {
      var s = "";
      for (var i = 0; i < aResult.length; i++) {
        s += "<LI><A HREF=\"" + aResult[i].url + "\">" +
        aResult[i].name + "</A></LI>";
      }
      var htelResult = Ext.get("cmain").insertHtml("beforeEnd",
      "<P>Результаты поиска:</P><UL>" + s + "</UL>");
      Ext.fly(htelResult).select("A").on("click", function(e, t) {
        var href = Ext.fly(this).getAttribute("href");
        var elA = Ext.get("navbar").child("A[href=" + href + "]");
        var elItem = elA.parent("LI");
        loadFragment(elItem, e);
      });
    }
  }
}
```

Рассмотрим его построчно.

Получаем искомое слово, введенное посетителем в поле ввода `keyword`:

```
var sKeyword = Ext.get("keyword").getValue(false);
```

Проверяем, ввел ли посетитель вообще что-либо в это поле ввода:

```
if (sKeyword != "") {
```

Если ввел, получаем номер пункта, выбранного в раскрывающемся списке `search_in`:

```
  var iSearchMode = Ext.getDom("search_in").selectedIndex;
```

Объявляем массив, который будет хранить набор элементов массивов `aHTML`, `aCSS` и `aSamples`, имеющих название или одно из ключевых слов, начало которого совпадает с введенным посетителем словом:

```
  var aResult = [];
```

Фактически этот массив будет хранить результаты поиска.

Для каждого из массивов `aHTML`, `aCSS` и `aSamples` вызываем функцию `searchInArray`, которую объявим потом:

```
  searchInArray(sKeyword, aHTML, aResult, iSearchMode);
  searchInArray(sKeyword, aCSS, aResult, iSearchMode);
  searchInArray(sKeyword, aSamples, aResult, iSearchMode);
```

Эта функция будет искать элементы в массиве, переданном ей вторым параметром, имеющие название или одно из ключевых слов, начало которого совпадает со словом, переданным первым параметром, и помещать их в массив с результатами поиска, переданный третьим параметром. Четвертым параметром этой функции передается номер пункта, выбранного в раскрывающемся списке `search_in`, — режим поиска.

Проверяем, есть ли в массиве с результатами поиска хоть один элемент, т. е. увенчался ли поиск успехом:

```
  if (aResult.length > 0) {
```

Если так, объявляем переменную, которая будет хранить строку с HTML-кодом, формирующим пункты списка с результатами поиска:

```
    var s = "";
```

(Ранее мы договорились, что будем выводить результаты поиска на Web-страницу в виде списка HTML; каждый пункт этого списка будет содержать гиперссылку на соответствующую Web-страницу.)

Запускаем цикл со счетчиком, который будет просматривать все элементы массива с результатами поиска:

```
    for (var i = 0; i < aResult.length; i++) {
```

Тело этого цикла на основе каждого элемента массива с результатами поиска сформирует HTML-код, создающий пункт списка с гиперссылкой:

```
      s += "<LI><A HREF=\"" + aResult[i].url + "\">" +
      aResult[i].name + "</A></LI>";
    }
```

На основе полученного таким образом HTML-кода создаем список с результатами поиска и помещаем его в самом конце контейнера `cmain`:

```
    var htelResult = Ext.get("cmain").insertHtml("beforeEnd",
    "<P>Результаты поиска:</P><UL>" + s + "</UL>");
```

Выбираем из этого списка все гиперссылки и привязываем к ним обработчик события `click`, который будет обрабатывать щелчки на этих гиперссылках и выполнять загрузку соответствующей Web-страницы:

```
    Ext.fly(htelResult).select("A").on("click", function(e, t) {
      var href = Ext.fly(this).getAttribute("href");
      var elA = Ext.get("navbar").child("A[href=" + href + "]");
      var elItem = elA.parent("LI");
      loadFragment(elItem, e);
    });
  }
}
```

Этот фрагмент кода без изменений перекочевал сюда из функции `generateRelated`, объявленной в *главе 19*. (В принципе, будет лучше оформить его в виде отдельной функции, но это вы можете сделать сами, в качестве домашнего задания.)

На этом выполнение функции `searchData` заканчивается.

Осталось объявить функцию `searchInArray`, которая, собственно, будет выполнять поиск в массивах, составляющих базу данных. Объявляющий код (листинг 20.21) мы поместим где-либо перед объявлением функции `searchData`.

Листинг 20.21

```
function searchInArray(sKeyword, aDataArray, aResultArray, iSearchMode) {
  var sN, sK;
  var sKw = "," + sKeyword.toLowerCase();
  for(var i = 0; i < aDataArray.length; i++) {
    sN = "," + aDataArray[i].name.toLowerCase();
    if (aDataArray[i].keyword)
      sK = "," + aDataArray[i].keyword.toLowerCase()
    else
      sK = "";
    if (((iSearchMode == 0) || (iSearchMode == 2)) &&
    (sN.indexOf(sKw) != -1))
      aResultArray[aResultArray.length] = aDataArray[i]
    else
      if (((iSearchMode == 1) || (iSearchMode == 2)) &&
      (sK.indexOf(sKw) != -1))
        aResultArray[aResultArray.length] = aDataArray[i];
  }
}
```

Как уже говорилось, эта функция принимает четыре параметра:

- искомое слово в виде строки;
- массив, составляющий базу данных, в котором будет выполняться поиск;
- массив, в который будут помещаться результаты поиска;
- число, обозначающее режим поиска. Фактически это номер пункта, выбранного посетителем в раскрывающемся списке `search_in`.

Результат эта функция возвращать не будет.

Давайте рассмотрим объявляющий ее код построчно, т. к. он довольно сложен, хоть и невелик по размеру.

Объявляем служебные переменные:

```
var sN, sK;
```

Преобразуем полученное первым параметром искомое слово к нижнему регистру, добавляем к ней спереди запятую и сохраняем в служебной переменной для дальнейшего использования:

```
var sKw = "," + sKeyword.toLowerCase();
```

Посетитель — человек непредсказуемый. Кто знает, в каком регистре он наберет искомое слово — в верхнем или нижнем, прописными буквами или строчными. А названия Web-страниц нашего Web-сайта указаны как в верхнем, так и в нижнем регистре. И строка, набранная в верхнем регистре, не равна строке, содержащей те же символы, но набранные в нижнем регистре; так, строки "`title`" и "`TITLE`", хоть и содержат одни и те же символы, не равны, поскольку эти символы набраны в разных регистрах.

Выход — перед сравнением строк принудительно преобразовать их к какому-либо одному регистру, скажем, к нижнему. В этом нам поможет метод `toLowerCase` объекта JavaScript `String`. Он как раз возвращает строку, равную той, для которой он вызван, но набранную в нижнем регистре. Параметров он не принимает.

Но зачем добавлять к искомой строке спереди запятую? Давайте разберемся.

Предположим, посетитель захотел найти материалы по тегу `<IMG>`. Причем посетитель попался на редкость ленивый, и, вместо того чтобы набрать имя тега полностью, ввел только букву "I".

Средства JavaScript позволяют узнать, присутствует ли в какой-либо строке указанная подстрока. (Как мы потом узнаем, за это "отвечает" особый метод объекта `String`.) Другими словами, приняв за подстроку введенное посетителем искомое слово, мы с помощью этих средств можем легко узнать, присутствует ли оно в названии или списке ключевых слов какого-либо элемента базы данных. Так, мы выясним, что в строке "IMG" присутствует подстрока "I", а в строке "!DOCTYPE" — нет.

Но ведь подстрока "I" присутствует и в строках "AUDIO", "VIDEO" и "TITLE"! А мы решили, что будем выбирать только те материалы, начало названий или ключевых слов содержит указанное слово. Начало, а не середина или конец! К сожалению, средства JavaScript не позволяют указать, в какой именно части слова должна присутствовать искомая подстрока...

Чтобы решить возникшую проблему, мы пойдем на небольшую хитрость. Мы добавим в начало искомого слова, названия и списка ключевых слов каждой Web-страницы какой-нибудь символ, например, запятую. А уже после этого будем выполнять сам поиск.

Например, если мы добавим к строкам "I", "IMG" и "AUDIO" спереди запятую, то получим ",I", ",IMG" и ",AUDIO". Посмотрим, что получится: строка ",IMG" содержит подстроку ",I", а ",AUDIO" — не содержит. Принятое нами правило поиска — указанное слово должно содержаться в начале названия — теперь выполняется. Как говорится, не мытьем, так катаньем. Ладно, поехали дальше...

Запускаем цикл со счетчиком, который будет просматривать все элементы массива базы данных, переданного вторым параметром:

```
for(var i = 0; i < aDataArray.length; i++) {
```

В теле этого цикла мы получаем название очередного элемента этого массива, преобразуем его к нижнему регистру, добавляем к нему спереди запятую и присваиваем объявленной ранее служебной переменной:

```
  sN = "," + aDataArray[i].name.toLowerCase();
```

Проверяем, есть ли у данного элемента свойство `keyword`:

```
if (aDataArray[i].keyword)
  sK = "," + aDataArray[i].keyword.toLowerCase()
else
  sK = "";
```

(Ранее мы решили, что оно будет необязательным.) Если оно есть, преобразуем его значение — список ключевых слов — к нижнему регистру, добавляем к нему спереди запятую и присваиваем объявленной ранее служебной переменной. Если его нет, присваиваем той же служебной переменной пустую строку — "пустой" список ключевых слов.

Если посетитель выбрал первый или третий пункты раскрывающегося списка `search_in` (т. е. если указан режим поиска по названиям или по названиям и ключевым словам; номер выбранного пункта списка `search_in` передается четвертым параметром) и если в названии элемента массива присутствует указанное посетителем слово:

```
if (((iSearchMode == 0) || (iSearchMode == 2)) &&
(sN.indexOf(sKw) != -1))
```

мы присваиваем этот элемент новому элементу массива результатов, переданного третьим параметром:

```
  aResultArray[aResultArray.length] = aDataArray[i]
```

Чтобы добавить к массиву новый элемент, нужно дать ему индекс, на единицу больший индекса его последнего уже существующего элемента. В качестве этого индекса удобно использовать размер массива — ведь он всегда на единицу больше индекса последнего элемента массива (конечно, при условии, что индексы всех элементов данного массива представляют собой непрерывающуюся последовательность чисел, причем каждое следующее число больше предыдущего на единицу).

Если посетитель выбрал второй или третий пункты раскрывающегося списка `search_in` (т. е. если указан режим поиска по ключевым словам или по названиям и ключевым словам) и если в списке ключевых слов элемента массива присутствует указанное посетителем слово

```
else
  if (((iSearchMode == 1) || (iSearchMode == 2)) &&
  (sK.indexOf(sKw) != -1))
```

мы присваиваем этот элемент новому элементу массива результатов, переданного третьим параметром:

```
    aResultArray[aResultArray.length] = aDataArray[i];
}
```

На этом выполнение тела цикла и тела функции `searchInArray` заканчивается.

Что ж, поиск готов. Откроем наш Web-сайт, наберем в поле ввода какое-либо слово и нажмем кнопку **Искать!**. Если поиск увенчается успехом, в самом конце контей-

нера `cmain` мы увидим список, пункты которого будут содержать гиперссылки на найденные Web-страницы.

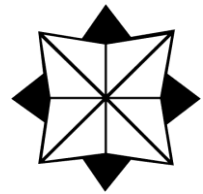
Поиск работает!

Что дальше?

В этой главе мы познакомились с Web-формами и элементами управления, тегами HTML для их создания и средствами объектов Web-обозревателя и библиотеки Ext Core для работы с ними. А еще мы наконец-то реализовали поиск на своем Web-сайте!

Только вот выглядит наш поиск на редкость непрезентабельно... Ну ничего, в следующей главе мы существенно улучшим его внешний вид! И помогут нам свободно позиционированные контейнеры — особым образом созданные блочные контейнеры, размеры и местоположение которых на Web-странице мы можем задавать совершенно произвольно.

ГЛАВА 21



Свободно позиционируемые элементы Web-страницы

В предыдущей главе мы познакомились с Web-формами и элементами управления, HTML-тегами для их создания и средствами объектов Web-обозревателя и библиотеки Ext Core для работы с ними. На основе этих элементов управления и базы данных мы создали систему поиска для своего Web-сайта. Наш небольшой Web-сайтик теперь выглядит просто шикарно!

Только вот поиск у нас вышел какой-то корявый, больше похожий не на готовое решение, а на экспериментальный "прибамбас", который со временем переделают во что-либо более приемлемое (или уберут совсем). Давайте сами посмотрим на него критическим взглядом.

- Контейнер с полосой навигации — не лучшее место для Web-формы поиска. То, что она нарушает дизайн Web-страницы, не страшно — задать для нее подходящее представление не составит для нас труда. Хуже другое — полоса навигации в какой-то момент станет слишком большой, не поместится в контейнер, Web-форма "уедет" вниз, и посетителю, чтобы до нее добраться, придется пользоваться полосами прокрутки. А ведь он должен догадаться, что Web-форма поиска еще присутствует на Web-странице, а не пропала бесследно и невесть куда!
- Низ контейнера с основным содержимым тоже плохо подходит для размещения результатов поиска: основное содержимое может оказаться слишком большим, чтобы поместиться в контейнер полностью, и посетителю придется пользоваться полосами прокрутки, чтобы добраться до результатов поиска.
- Теперь предположим, что посетитель выполнил поиск, который оказался удачным, и в низу контейнера с основным содержимым появится список с результатами. После этого посетитель снова выполнил успешный поиск, и в низу контейнера с основным содержимым появится еще один список — с результатами нового поиска. Если повторять поиск снова и снова, будут появляться все новые списки с результатами, и так без конца.

В принципе, некоторые из перечисленных проблем можно решить известными нам средствами. Как уже говорилось, для Web-формы мы можем создать представление, "облагораживающее" ее. Мы способны выявлять, присутствует ли уже в основном содержимом список с результатами поиска, и удалять его; для этого мы

можем привязать к списку стилевой класс или дать ему имя, по которому сможем его найти.

Но куда выводить результаты поиска, чтобы посетитель сразу их увидел?

Напрашивается вывод: создать на Web-странице еще один блочный контейнер, поместить в него Web-форму и туда же выводить результаты поиска. Но дополнительный контейнер займет определенное место на Web-странице, может быть, слишком много места. А значит, останется меньше пространства для остальных контейнеров.

Мы можем сделать этот контейнер изначально небольшим, ровно таким, чтобы вместить только Web-форму. Если же поиск увенчается успехом, мы увеличим размер контейнера, чтобы он вместил список с результатами. Но тогда остальные контейнеры будут "ездить" по Web-странице туда-сюда, чем немало развеселят посетителей. Хорошее настроение — оно, конечно, замечательно, но все равно это не выход.

Свободно позиционируемые контейнеры

Давайте вернемся назад, к языкам HTML и CSS, и посмотрим, не предложат ли они нам что-либо, радикально решающее эту проблему. Так и есть!

Понятие свободно позиционируемого элемента Web-страницы

Откроем любую из созданных нами ранее Web-страниц и посмотрим на нее. Что мы видим?

Прежде всего, расположением элементов этих Web-страниц управляет сам Web-обозреватель. При этом он руководствуется следующими правилами.

- Элемент выводится на экран в том месте, в котором находится определяющий его HTML-код. Так, контейнер `cheader` мы определили в самом начале HTML-кода Web-страницы index.htm, поэтому он будет выведен в самом ее начале, т. е. в верхней части.
- Если для элементов задано значение `none` атрибута стиля `float` (см. *главу 10*) или этот атрибут стиля вообще отсутствует, то элементы выстроятся друг за другом по вертикали. Пример: контейнеры `cheader` и `cnavbar`, для которых мы не указали этот атрибут стиля.
- При других значениях атрибута стиля `float` элементы выстроятся по горизонтали. Пример: контейнеры `cnavbar` и `cmain`, для которых мы задали значение `left` атрибута стиля `float`.

Произвольно управлять местоположением элементов Web-страницы в этом случае мы не можем. Поэтому такие элементы называются *непозиционируемыми*.

Web-дизайнерам и особенно Web-программистам такое положение дел не нравилось. Именно поэтому уже довольно давно в языке CSS появилась возможность

создавать *свободно позиционируемые*, или *свободные*, элементы Web-страницы. Подобный элемент может располагаться где угодно на Web-странице, независимо от места в HTML-коде, где стоит определяющий его тег.

Рассмотрим особенности свободно позиционируемых элементов Web-страницы.

- Местоположение свободно позиционируемого элемента задается произвольно в виде горизонтальной и вертикальной координат его верхнего левого угла. Координаты задают относительно верхнего левого угла родителя данного элемента.
- Под свободно позиционируемый элемент на Web-странице место не выделяется.
- Свободно позиционируемые элементы находятся "выше" обычного содержимого Web-страницы, как бы "плавают" над ним и перекрывают его.
- Свободно позиционируемые элементы могут перекрывать друг друга. Обычное содержимое Web-страницы свободные элементы перекрывают в любом случае.
- Слово "перекрывают" в предыдущих двух пунктах обозначает, что содержимое Web-страницы, находящееся под свободным элементом, не будет видно — его скроет свободный элемент.
- Свободно позиционируемые элементы могут иметь любое содержимое, в том числе и другие свободно позиционируемые элементы.

Существующая реализация CSS позволяет сделать свободно позиционируемыми только блочные контейнеры. В этом случае говорят о *свободно позиционируемых*, или *свободных*, контейнерах.

Создание свободно позиционируемых элементов

Свободные элементы Web-страницы создают с помощью особых атрибутов стиля CSS, которые мы сейчас рассмотрим.

Самый важный атрибут стиля — `position`. Он задает способ позиционирования элемента Web-страницы:

```
position: static|absolute|relative|fixed|inherit
```

Этот атрибут стиля может принимать четыре значения:

- `static` — контейнер непозиционируемый (поведение по умолчанию);
- `absolute` — элемент Web-страницы свободно позиционируемый. Его координаты задаются относительно верхнего левого угла родителя. Место на Web-странице под такой элемент не выделяется. Если содержимое родителя прокручивается, свободно позиционируемый элемент будет перемещаться вместе с ним;
- `relative` — элемент Web-страницы *относительно позиционируемый*. Его координаты отсчитываются относительно точки, в которой он находился бы, если был непозиционируемым. На Web-странице выделяется место под такой элемент;

- `fixed` — элемент Web-страницы *фиксированно позиционируемый*. Он ведет себя как свободный элемент, с двумя исключениями. Во-первых, его координаты задаются относительно верхнего левого угла Web-страницы. Во-вторых, если содержимое родителя прокручивается, фиксированно позиционируемый элемент не будет перемещаться вместе с ним.

Пример:

```
#search { position: absolute }
```

Здесь мы превратили контейнер `search` в свободно позиционируемый.

Атрибуты стиля `left` и `top` задают, соответственно, горизонтальную и вертикальную координаты верхнего левого угла свободно, относительно или фиксированно позиционируемого элемента Web-страницы:

```
left|top: <значение>|auto|inherit
```

Значения координат можно указывать в любых единицах измерения, поддерживаемых стандартом CSS (см. табл. 8.1). Значение `auto` возвращает управление соответствующей координатой Web-обозревателю.

В примере из листинга 21.1 мы задали координаты и размеры контейнера `search`.

Листинг 21.1

```
#search { position: absolute;
          left: 200px;
          top: 100px;
          width: 300px;
          height: 200px }
```

Мы уже знаем, что свободные элементы могут перекрывать друг друга. При этом элемент, определенный в HTML-коде позже, перекрывает элемент, определенный раньше. Однако мы можем сами задать порядок их перекрытия друг другом, указав так называемый *z-индекс*. Он представляет собой целое число, указывающее номер в порядке перекрытия; при этом элементы с бóльшим z-индексом перекрывают элементы с меньшим z-индексом. Z-индекс задается атрибутом стиля с "говорящим" именем `z-index`:

```
z-index: <номер>|auto|inherit
```

Как уже говорилось, z-индекс указывается в виде целого числа. Значение `auto` возвращает управление порядком перекрытия Web-обозревателю. Листинг 21.2 иллюстрирует пример.

Листинг 21.2

```
#search { position: absolute;
          left: 200px;
          top: 100px;
          width: 300px;
```

```
          height: 200px;
          z-index: 2 }
#main   { position: absolute;
          left: 100px;
          top: 0px;
          width: 600px;
          height: 500px;
          z-index: 0 }
```

Контейнер `search` перекроет контейнер `main`, поскольку для него задан больший z-индекс.

Еще один атрибут стиля, который иногда может быть полезен, — `clip`. Он определяет координаты прямоугольной области, задающей видимую область свободного элемента. Фрагмент содержимого элемента, попадающий в эту область (ее, кстати, называют *маской*), будет видим на Web-странице, остальная часть содержимого будет скрыта.

Вот синтаксис записи атрибута `clip`:

```
clip: rect(<верхняя граница>, <правая граница>, <нижняя граница>,
↳<левая граница>)|auto|inherit
```

Здесь:

- *верхняя граница* — расстояние от верхней границы свободного элемента до верхней границы маски по вертикали;
- *правая граница* — расстояние от левой границы свободного элемента до правой границы маски по горизонтали;
- *нижняя граница* — расстояние от верхней границы свободного элемента до нижней границы маски по вертикали;
- *левая граница* — расстояние от левой границы свободного элемента до левой границы маски по горизонтали.

Значение `auto` атрибута стиля `clip` убирает маску и тем самым делает все содержимое свободного элемента видимым. Это поведение по умолчанию. Листинг 21.3 иллюстрирует пример.

Листинг 21.3

```
#search { position: absolute;
          left: 200px;
          top: 100px;
          width: 300px;
          height: 200px;
          z-index: 2;
          clip: rect(100px, 200px, 200px, 0px) }
```

Средства библиотеки Ext Core для управления свободно позиционируемыми элементами

Настала пора рассмотреть методы объекта `Element` библиотеки Ext Core, с помощью которых мы можем управлять свободно позиционируемыми элементами Web-страницы. Их немного.

Метод `position` задает способ позиционирования, z-индекс и координаты данного элемента:

```
<экземпляр объекта Element>.position(<способ позиционирования>
↳[, <z-индекс>[, <горизонтальная координата>
↳[, <вертикальная координата>]]])
```

Первым — обязательным — параметром передается соответствующее значение атрибута стиля `position` в виде строки: `"absolute"`, `"relative"` или `"fixed"`. Остальные — необязательные — параметры определяют, соответственно, z-индекс, горизонтальную и вертикальную координаты в пикселах; все эти значения задаются в виде чисел.

Пример:

```
var elSearch = Ext.get("search");
elSearch.position("absolute", 2);
```

Методы `setX` и `setY` задают, соответственно, горизонтальную и вертикальную координаты данного элемента относительно верхнего левого угла Web-страницы:

```
<экземпляр объекта Element>.setX|setY(<значение координаты>)
```

Значение координат указывают в пикселах в виде числа:

```
elSearch.setX(400);
elSearch.setY(200);
```

Метод `setLocation` задает сразу обе координаты данного элемента также относительно верхнего левого угла Web-страницы:

```
<экземпляр объекта Element>.setLocation(<горизонтальная координата>,
↳<вертикальная координата>)
```

Оба значения координат задают в виде чисел в пикселах:

```
elSearch.setLocation(400, 200);
```

Метод `clearPositioning` делает данный элемент непозиционируемым и удаляет заданные для него координаты. Этот метод не принимает параметров и не возвращает результата:

```
elSearch.clearPositioning();
```

Реализация усовершенствованного поиска

Что ж, все, что нам нужно знать о свободно позиционируемых элементах, мы рассмотрели. Настала пора практических занятий.

В *главе 20* мы реализовали на нашем Web-сайте систему поиска. Получилось, мягко говоря, не очень профессионально, о чем уже говорилось в начале этой главы. Давайте улучшим ситуацию.

Прежде всего, мы создадим новый контейнер, дадим ему имя `csearch` и поместим в него Web-форму поиска. В этот же контейнер, ниже Web-формы, мы вставим список, в котором будут выводиться результаты поиска. (Результаты мы будем формировать в виде пунктов списка, содержащих гиперссылки на найденные Web-страницы, — как и в *главе 20*.) Дадим этому списку имя `search_result` и сделаем его изначально скрытым.

Когда посетитель выполнит поиск, мы проверим, присутствуют ли в списке `search_result` какие-либо пункты (т. е. выполнялся ли поиск ранее и был ли он удачным), и, если присутствуют, удалим их и скроем этот список. Если поиск увенчался успехом, мы сформируем в списке `search_result` пункты, содержащие гиперссылки на найденные Web-страницы, и откроем его. Таким образом, список `search_result` будет присутствовать на экране только в случае успешного поиска.

Когда посетитель щелкнет на любом месте Web-страницы (неважно — на гиперссылке, в том числе и гиперссылке в списке результатов поиска, на абзаце, на изображении или вообще на пустом месте), мы должны скрыть список `search_result`. Это нужно для того, чтобы этот список не присутствовал на экране постоянно и не мешал посетителю.

В остальном новый поиск будет работать так же, как и старый.

Создание контейнера с Web-формой поиска

Откроем Web-страницу index.htm в Блокноте, найдем созданный в *главе 20* фрагмент кода, создающий Web-форму поиска, и удалим его. Вместо него мы вставим сразу после открывающего тега `<BODY>` код, приведенный в листинге 21.4.

Листинг 21.4

```
<DIV ID="csearch">
  <FORM ACTION="#">
    <P>
      <INPUT TYPE="search" ID="keyword" NAME="keyword" SIZE="20">
      <INPUT TYPE="button" ID="find" NAME="find" VALUE="Искать!"><BR>
      <SELECT ID="search_in" NAME="search_in">
        <OPTION>В названиях</OPTION>
        <OPTION>В ключевых словах</OPTION>
        <OPTION SELECTED>В названиях и ключевых словах</OPTION>
      </SELECT>
    </P>
    <UL ID="search_result">
    </UL>
  </FORM>
</DIV>
```

Он создает контейнер `csearch`, а в нем — Web-форму поиска. В Web-форме присутствуют те же элементы управления: поле ввода искомого слова, кнопка запуска поиска и раскрывающийся список для выбора режима поиска. Ниже Web-формы мы поместили список `search_result`, в котором будут выводиться результаты поиска.

Далее нам нужно задать стиль для только что созданного контейнера `csearch`, который сделает его свободно позиционируемым. Откроем таблицу стилей main.css в Блокноте и добавим в нее CSS-код, приведенный в листинге 21.5.

Листинг 21.5

```
#csearch      { background-color: #F8F8F8;
                position: absolute;
                left: 600px;
                top: 0px;
                padding: 2px;
                border: thin solid #B1BEC6 }
```

Здесь мы, собственно, делаем контейнер `csearch` свободно позиционируемым, задаем для него начальные координаты, внутренние отступы и рамку. Внутренние отступы будут совсем небольшими, чтобы контейнер сохранял компактность, а рамка — сплошной — пусть Web-форма поиска будет сразу заметна.

А еще мы указываем для контейнера `csearch` цвет фона — такой же, как у Web-страницы. Если мы этого не сделаем, фон контейнера будет прозрачным, и сквозь него станет просвечивать содержимое Web-страницы, расположенное "ниже" контейнера. А это будет выглядеть очень некрасиво.

Раз уж мы правим представление Web-страницы, давайте сразу зададим стили для элементов управления и списка `search_result`, чтобы сделать их привлекательнее. Рассмотрим эти стили отдельно друг от друга.

Мы задаем для абзаца, в котором поместили элементы управления, и списка `search_result` нулевые внешние отступы, чтобы сделать контейнер `csearch` компактнее:

```
#csearch P,
  #search_result  { margin: 0px }
```

Для элементов управления назначаем размер шрифта 10 пунктов:

```
INPUT, SELECT { font-size: 10pt }
```

Дело в том, что размер шрифта по умолчанию, принятый для элементов управления, слишком мал, что понравится далеко не всем посетителям.

Убираем у пунктов списка `search_result` маркеры и слишком большой отступ слева, где, собственно, выводятся эти маркеры:

```
#search_result LI { list-style-type: none;
                    margin-left: -40px; }
```

Так мы сделаем контейнер csearch еще компактнее.

На этом с Web-формой и элементами управления покончено.

Написание Web-сценария, выполняющего поиск

Осталось создать (точнее, переделать уже созданный в *главе 20*) Web-сценарий, который, собственно, будет выполнять поиск.

Откроем файл Web-сценария main.js в Блокноте и добавим в его начало такое выражение:

```
var cSearchHeight = 0;
```

Оно объявляет служебную переменную, в которой будет сохранена изначальная высота контейнера csearch. Это значение мы будем использовать для позиционирования данного контейнера. Его высота в процессе работы будет постоянно меняться, поэтому для его позиционирования нам понадобится изначальное значение высоты.

Далее найдем тело функции, передаваемой методу onReady объекта Ext. В самом его начале поместим два выражения:

```
Ext.get("search_result").setDisplayed(false);
cSearchHeight = Ext.get("csearch").getHeight();
```

Первое выражение сразу скроет список search_result, а второе присвоит изначальную высоту контейнера csearch объявленной ранее служебной переменной.

Функцию adjustContainers, задающую размеры контейнеров, мы объявили еще в *главе 16* и с тех пор ни разу к ней не возвращались. Настала пора внести в объявление этой функции некоторые правки.

Вот выражения, которые мы добавим в самый конец adjustContainers:

```
var elCSearch = Ext.get("csearch");
elCSearch.setLocation(clientWidth − elCSearch.getWidth(),
Ext.get("cmain").getY() − cSearchHeight);
```

Они позиционируют контейнер csearch так, чтобы он в любом случае находился над верхним левым углом контейнера cmain. Кстати, здесь используется значение изначальной высоты контейнера csearch, которое мы сохранили ранее.

Теперь найдем объявление функции searchData, написанной нами в *главе 20*. Переделаем его так, как показано в листинге 21.6.

Листинг 21.6

```
function searchData() {
  var elSearchResult = Ext.get("search_result");
  elSearchResult.select("A").removeAllListeners();
  elSearchResult.dom.innerHTML = "";
  elSearchResult.setDisplayed(false);
  var sKeyword = Ext.get("keyword").getValue(false);
```

```
  if (sKeyword != "") {
    var iSearchMode = Ext.getDom("search_in").selectedIndex;
    var aResult = [];
    searchInArray(sKeyword, aHTML, aResult, iSearchMode);
    searchInArray(sKeyword, aCSS, aResult, iSearchMode);
    searchInArray(sKeyword, aSamples, aResult, iSearchMode);
    if (aResult.length > 0) {
      var s = "";
      for (var i = 0; i < aResult.length; i++) {
        s += "<LI><A HREF=\"" + aResult[i].url + "\">" +
        aResult[i].name + "</A></LI>";
      }
      var htelResult = elSearchResult.insertHtml("beforeEnd", s);
      Ext.fly(htelResult).select("A").on("click", function(e, t) {
        var href = Ext.fly(this).getAttribute("href");
        var elA = Ext.get("navbar").child("A[href=" + href + "]");
        var elItem = elA.parent("LI");
        loadFragment(elItem, e);
      });
      elSearchResult.setDisplayed(true);
    }
  }
}
```

Рассмотрим листинг 21.6 построчно.

Перед поиском нам нужно удалить все пункты, уже присутствующие в списке `search_result`. Для этого мы сначала удаляем все обработчики событий, привязанные к гиперссылкам, находящимся в пунктах этого списка, а потом удаляем сами пункты:

```
var elSearchResult = Ext.get("search_result");
elSearchResult.select("A").removeAllListeners();
elSearchResult.dom.innerHTML = "";
elSearchResult.setDisplayed(false);
```

Напоследок скрываем список `search_result`.

Обратим внимание, как выполняется удаление пунктов списка `search_result`. Из *главы 15* мы знаем, что объект Web-обозревателя `HTMLElement` поддерживает свойство `innerHTML`, хранящее HTML-код, создающий содержимое данного элемента Web-страницы, в виде строки. Значит, чтобы удалить все содержимое данного элемента, мы можем получить соответствующий ему экземпляр объекта `HTMLElement` (через свойство `dom` объекта Ext Core `Element`) и присвоить его свойству `innerHTML` пустую строку. Что мы и делаем.

Листинг 21.7

```
var sKeyword = Ext.get("keyword").getValue(false);
if (sKeyword != "") {
  var iSearchMode = Ext.getDom("search_in").selectedIndex;
```

```
  var aResult = [];
  searchInArray(sKeyword, aHTML, aResult, iSearchMode);
  searchInArray(sKeyword, aCSS, aResult, iSearchMode);
  searchInArray(sKeyword, aSamples, aResult, iSearchMode);
  if (aResult.length > 0) {
    var s = "";
    for (var i = 0; i < aResult.length; i++) {
      s += "<LI><A HREF=\"" + aResult[i].url + "\">" +
      aResult[i].name + "</A></LI>";
    }
    var htelResult = elSearchResult.insertHtml("beforeEnd", s);
    Ext.fly(htelResult).select("A").on("click", function(e, t) {
      var href = Ext.fly(this).getAttribute("href");
      var elA = Ext.get("navbar").child("A[href=" + href + "]");
      var elItem = elA.parent("LI");
      loadFragment(elItem, e);
    });
```

Фрагмент кода, приведенный в листинге 21.7, перекочевал из предыдущей реализации функции `searchData` практически без изменений. Мы уже знаем, что он делает.

Сформировав пункты списка `search_result`, открываем его:

```
    elSearchResult.setDisplayed(true);
  }
}
```

На этом выполнение функции `searchData` заканчивается.

Функция `cleanupSamples`, которую мы объявили в *главе 16*, удаляет обработчики событий, привязанные к гиперссылкам раздела "См. также" и результатам поиска. Найдем объявляющий ее код и удалим выражения, которые убирают обработчики событий у гиперссылок результатов поиска, — ведь ранее мы поместили выполняющий это действие код в функцию `searchData`. После этого объявление функции `cleanupSamples` будет выглядеть так, как в листинге 21.8.

Листинг 21.8

```
function cleanupSamples() {
  var ceSamples = Ext.select(".sample");
  ceSamples.each(function(el, cl, ind){
    var elH6 = el.child(":first");
    elH6.removeAllListeners();
  });
}
```

Так, бóльшую часть работы мы сделали. Осталось реализовать скрытие списка `search_result` при щелчке на содержимом Web-страницы.

Вернемся к телу функции, передаваемой параметром методу `onReady` объекта `Ext`, и добавим в его конец такое выражение:

```
Ext.getBody().on("click",
function(){ Ext.get("search_result").setDisplayed(false); });
```

Оно привязывает к событию `click` секции тела Web-страницы обработчик, который скрывает список `search_result`.

В *главе 15* мы узнали, что некоторые события, в том числе и `click`, имеют обыкновение всплывать из элементов, в которых они изначально возникли, в их родители, затем — в родители их родителей и, наконец, в секцию тела Web-страницы. Обработчик события `click`, который мы только что привязали к секции тела Web-страницы, сработает независимо от того, в каком элементе Web-страницы возникло это событие, и список `search_result` в любом случае будет скрыт.

Но тут возникает очень неприятный момент: событие `click` кнопки запуска поиска также рано или поздно всплывет в секцию тела Web-страницы. Давайте посмотрим, что получится в результате. Посетитель нажмет кнопку запуска поиска, функция `searchData` сформирует пункты списка результатов и откроет этот список, после чего выполнится обработчик события `click`, привязанный нами к секции тела Web-страницы, который скроет список результатов. Непорядок!

Найдем в теле функции, передаваемой параметром методу `onReady` объекта `Ext`, вот это выражение:

```
Ext.get("find").on("click", searchData);
```

Оно привязывает обработчик к событию `click` кнопки, запускающей поиск. Изменим его следующим образом:

```
Ext.get("find").on("click", function(e){
  searchData();
  e.stopPropagation();
});
```

Новый обработчик события `click` сначала вызовет функцию `searchData`, собственно выполняющую поиск, а потом подавит всплытие возникшего события. Как видим, для этого используется метод `stopPropagation` объекта Ext Core `EventObject` (см. *главу 15*).

И еще. В обработчике события `click` пунктов полосы навигации (функция `loadFragment`) у нас подавляется всплытие этого события. Следовательно, если посетитель щелкнет на пункте полосы навигации (или гиперссылке раздела "См. также", или гиперссылке пункта в списке результатов поиска), событие `click` не всплывет в секцию тела Web-страницы, привязанный к нему обработчик не выполнится, и список `search_result` скрыт не будет. Нам нужно это исправить.

Найдем код, объявляющий функцию `loadFragment`, и добавим в самый его конец такое выражение:

```
Ext.get("search_result").setDisplayed(false);
```

Что оно делает, мы уже знаем.

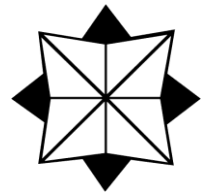
Сохраним все исправленные файлы и проверим поиск в действии. Вот теперь он выглядит вполне профессионально!..

Что дальше?

В этой главе мы познакомились со свободно позиционируемыми элементами Web-страницы и даже применили их на практике в новой версии системы поиска. Мелочь, конечно, — иные Web-дизайнеры с помощью свободных элементов творят на Web-страницах чудеса, — но для начала получилось неплохо.

В следующей, последней, главе мы познакомимся с еще одной возможностью HTML 5 — программируемой графикой. Мы научимся рисовать произвольные фигуры на Web-странице. А еще мы наделим наш Web-сайт графическим логотипом.

ГЛАВА 22



Программируемая графика

В предыдущей главе мы познакомились со свободно позиционируемыми элементами Web-страницы и использовали их, чтобы улучшить систему поиска на нашем Web-сайте. Получилось неплохо, правда?

На этом мы закончим с поиском и обратим внимание на заголовок нашего Web-сайта. Как-то неказисто он выглядит — обычный текст без всяких изысков. И это в то время, когда большинство Web-сайтов щеголяют шикарными графическими логотипами. Почему у нас такого нет?

Потому что мы этим еще не занимались. Вообще, сделать графический логотип Web-сайта проще простого — достаточно нарисовать его в каком-либо графическом редакторе и сохранить в любом формате, применяемом в Интернете. А как поместить на Web-страницу графическое изображение, мы уже знаем — изучали еще в *главе 4*.

Существует и другой путь — задействовать возможности программируемой графики, предлагаемые HTML 5. Они позволяют нарисовать любую, даже весьма сложную фигуру программно, из Web-сценария. Причем пользоваться этими возможностями не так и сложно — достаточно уяснить пару концепций и выучить несколько десятков методов.

Давайте так и сделаем. Ведь если мы взялись изучать HTML 5, так уж будем идти до конца.

Канва

Начнем сразу с первой концепции, на основе которой работает программируемая графика HTML 5 и которую нам нужно уяснить. Все рисование выполняется в особом элементе Web-страницы, еще нам не встречавшемся, — в *канве* (ее еще называют *холстом*). В других элементах (абзацах, заголовках, таблицах, графических изображениях и пр.) программное рисование не работает.

Канву создают с помощью парного тега `<CANVAS>`:

```
<CANVAS ID="<имя>" [WIDTH="<ширина>"] [HEIGHT="<высота>"]></CANVAS>
```

Мы уже знаем, что рисование в канве выполняется программно, в Web-сценарии. А перед тем как что-то нарисовать, нам придется получить доступ к канве. Сделать это проще всего через имя, заданное атрибутом тега ID. Именно поэтому данный атрибут тега помечен здесь как обязательный.

Необязательные атрибуты тега WIDTH и HEIGHT задают, соответственно, ширину и высоту канвы в пикселах (по умолчанию 300×150 пикселов).

> ***Внимание!***
> Задавать размеры канвы с помощью стилей CSS не рекомендуется.

Вот HTML-код, создающий на странице канву cnv размером 400×300 пикселов:

```
<CANVAS ID="cnv" WIDTH="400" HEIGHT="300"></CANVAS>
```

Канва представляется как экземпляр объекта Web-обозревателя HTMLCanvasElement, производный от объекта HTMLElement. Для нас будет полезен только единственный метод этого объекта, который мы скоро рассмотрим.

Контекст рисования

Рисование на канве выполняется с помощью особых свойств и методов объекта... нет, не HTMLCanvasElement, а CanvasRenderingContext2D. Этот объект представляет так называемый *контекст рисования*, который можно рассматривать как набор инструментов, используемый для рисования на данной канве.

Значит, перед тем как начать рисование, нам придется как-то получить экземпляр объекта Web-обозревателя CanvasRenderingContext2D для данной канвы. Это выполняется вызовом единственного метода getContext объекта HTMLCanvasElement:

```
<канва>.getContext("2d")
```

Мы видим, что метод getContext принимает единственный параметр — строку "2d". Возвращает он то, что нам нужно, — экземпляр объекта CanvasRenderingContext2D, представляющий контекст рисования данной канвы.

Напишем небольшой Web-сценарий, который помещает в переменную ctxCanvas контекст рисования для ранее созданной канвы cnv:

```
var htelCanvas = Ext.getDom("cnv");
var ctxCanvas = htelCanvas.getContext("2d");
```

Впоследствии мы будем пользоваться этим контекстом рисования для наших примеров.

Вот теперь, вооружившись контекстом рисования канвы, мы можем начать рисовать на ней с помощью весьма многочисленных свойств и методов объекта CanvasRenderingContext2D, которые мы обязательно рассмотрим.

> ***Внимание!***
> Все свойства и методы, рассматриваемые далее, принадлежат объекту CanvasRenderingContext2D, если об этом не сказано специально.

При выполнении операций рисования нам потребуется задавать координаты точек, в которых будет начинаться и заканчиваться рисование фигур и пр. Координаты измеряются в пикселах и отсчитываются от верхнего левого угла канвы; другими словами — в верхнем левом углу канвы находится начало ее координат. Запомним это.

Рисование простейших фигур

Начнем мы с самых простых операций — рисования различных прямоугольников, с заливкой и без нее.

Для рисования прямоугольника без заливки (т. е. одного лишь контура прямоугольника) предназначен метод `strokeRect` объекта `CanvasRenderingContext2D`:

```
<контекст рисования>.strokeRect(<горизонтальная координата>,
↳<вертикальная координата>, <ширина>, <высота>)
```

Первые два параметра задают горизонтальную и вертикальную координаты верхнего левого угла рисуемого прямоугольника в пикселах в виде чисел. Третий и четвертый параметры задают, соответственно, ширину и высоту прямоугольника, также в пикселах и также в виде чисел. Метод `strokeRect` не возвращает результата.

Пример:

```
ctxCanvas.strokeRect(20, 20, 360, 260);
```

Метод `fillRect` рисует прямоугольник с заливкой:

```
<контекст рисования>.fillRect(<горизонтальная координата>,
↳<вертикальная координата>, <ширина>, <высота>)
```

Как видим, формат его вызова такой же, как у метода `strokeRect`:

```
ctxCanvas.fillRect(40, 40, 320, 220);
```

Весьма полезный для создания сложных фигур метод `clearRect` очищает заданную прямоугольную область от любой присутствовавшей там графики:

```
<контекст рисования>.clearRect(<горизонтальная координата>,
↳<вертикальная координата>, <ширина>, <высота>)
```

И его формат вызова схож с форматом вызова метода `strokeRect`.

Вот выражения, которые рисуют большой прямоугольник с заливкой, занимающий всю канву `cnv`, после чего создают в его середине прямоугольную "прореху":

```
ctxCanvas.fillRect(0, 0, 400, 300);
ctxCanvas.clearRect(100, 100, 200, 100);
```

А это выражение очищает канву от всей присутствующей на ней графики:

```
ctxCanvas.clearRect(0, 0, 400, 300);
```

Задание цвета, уровня прозрачности и толщины линий

Во время работы с канвой нам придется задавать цвета линий и заливок, уровень их прозрачности и толщину линий. Это выполняется с помощью особых свойств объекта `CanvasRenderingContext2D`.

Свойство `strokeStyle` задает цвет линий контура. Все фигуры, которые мы впоследствии нарисуем, будут иметь контур данного цвета. Цвет задают в виде строки либо с именем цвета, либо в привычном нам формате `#RRGGBB`, либо в двух других форматах, которые мы сейчас рассмотрим.

Вот первый формат:

```
rgb(<красная составляющая>, <зеленая составляющая>, <синяя составляющая>)
```

Здесь все три составляющие цвета имеют вид десятичных чисел от 0 до 255.

Второй формат позволяет дополнительно задать уровень прозрачности рисуемых линий:

```
rgba(<красная составляющая>, <зеленая составляющая>,
<синяя составляющая>, <уровень прозрачности>)
```

Три составляющие цвета также представляют собой десятичные числа от 0 до 255. Уровень прозрачности задают в виде числа от 0.0 (полностью прозрачный) до 1.0 (полностью непрозрачный).

Все четыре выражения задают непрозрачный красный цвет линий контура:

```
ctxCanvas.strokeStyle = "red";
ctxCanvas.strokeStyle = "#FF0000";
ctxCanvas.strokeStyle = "rgb(255, 0, 0)";
ctxCanvas.strokeStyle = "rgb(255, 0, 0, 1)";
```

А вот выражение, задающее для линий контура полупрозрачный черный цвет:

```
ctxCanvas.strokeStyle = "rgb(0, 0, 0, 0.5)";
```

Изначально, сразу после загрузки и вывода канвы на Web-страницу, линии контура будут иметь черный цвет.

Свойство `fillStyle` определяет цвет заливки, также в строковом виде и с использованием тех же форматов, что описаны ранее. Для цвета заливок действуют те же правила, что и для цвета линий. По умолчанию цвет заливок также черный.

Вот выражение, задающее тускло-зеленый непрозрачный цвет заливки:

```
ctxCanvas.fillStyle = "rgb(0, 127, 0)";
```

Еще один пример иллюстрирует листинг 22.1.

Листинг 22.1

```
ctxCanvas.strokeStyle = "rgba(255, 0, 0, 1)";
ctxCanvas.fillStyle = "rgba(255, 0, 0, 1)";
```

```
ctxCanvas.fillRect(0, 100, 400, 100);
ctxCanvas.strokeStyle = "rgba(0, 255, 0, 0.5)";
ctxCanvas.fillStyle = "rgba(0, 255, 0, 0.5)";
ctxCanvas.fillRect(100, 0, 200, 300);
```

Web-сценарий из листинга 22.1 рисует прямоугольник с заливкой, используя и для контура, и для заливки непрозрачный красный цвет, после чего поверх него рисует прямоугольник с заливкой, но уже полупрозрачным зеленым цветом. При этом сквозь полупрозрачный второй прямоугольник будет просвечивать непрозрачный первый. Интересный эффект, кстати!

> ***Внимание!***
>
> Нельзя присваивать значение свойства `strokeStyle` свойству `fillStyle` и наоборот. Это вызовет ошибку в Web-сценарии.

Свойство `lineWidth` задает толщину линий в пикселах в виде числа.

Пример:

```
ctxCanvas.lineWidth = 20;
ctxCanvas.strokeRect(20, 20, 360, 260);
```

Этот Web-сценарий рисует прямоугольник без заливки линиями толщиной 20 пикселов.

Свойство `globalAlpha`, возможно, также нам пригодится. Оно позволяет задать уровень прозрачности для любой графики, которую мы впоследствии нарисуем. Уровень прозрачности также задается в виде числа от 0.0 (полностью прозрачный) до 1.0 (полностью непрозрачный).

Вот выражение, задающее для всей графики, которую мы потом нарисуем на канве, уровень прозрачности 10%:

```
ctxCanvas.globalAlpha = 0.1;
```

Рисование сложных фигур

Канва также поддерживает рисование более сложных, чем прямоугольники, фигур с контурами из множества прямых и кривых линий. Сейчас мы выясним, как это делается, и рассмотрим соответствующие методы объекта `CanvasRenderingContext2D`.

Как рисуются сложные контуры

Контуры сложных фигур рисуются в три этапа.

1. Web-обозреватель ставится в известность, что сейчас начнется рисование контура сложной фигуры.
2. Рисуются отдельные линии, прямые и кривые, составляющие сложный контур.
3. Web-обозреватель ставится в известность, что рисование контура закончено, и теперь фигура должна быть выведена на канву, возможно, с созданием заливки.

Также можно указать Web-обозревателю, что следует замкнуть нарисованный контур.

Рисование сложного контура начинается с вызова метода `beginPath`, который не принимает параметров и не возвращает результата.

Собственно рисование линий, составляющих сложный контур, выполняют особые методы, которые мы далее рассмотрим.

После окончания рисования сложного контура мы можем захотеть, чтобы Web-обозреватель его замкнул. Это реализует метод `closePath`, который не принимает параметров и не возвращает результата. После его вызова последняя точка контура будет соединена с самой первой, в которой началось его рисование.

Завершает рисование контура вызов одного из двух методов: `stroke` или `fill`. Первый метод просто завершает рисование контура, второй, помимо этого, замыкает контур, если он не замкнут, и рисует заливку получившейся фигуры. Оба метода не принимают параметров и не возвращают результата.

А теперь рассмотрим методы, которые используются для рисования разнообразных линий, составляющих сложный контур.

Перо. Перемещение пера

Для рисования сложного контура используется концепция *пера* — воображаемого инструмента рисования. Перо можно перемещать в любую точку на канве. Рисование каждой линии контура начинается в точке, где в данный момент находится перо. После рисования каждой линии перо перемещается в ее конечную точку, из которой тут же можно начать рисование следующей линии контура.

Изначально, сразу после загрузки Web-страницы и вывода канвы, перо находится в точке с координатами [0,0], т. е. в верхнем левом углу канвы. Переместить перо в другую точку канвы, где мы собираемся начать рисование контура, позволяет метод `moveTo`:

```
<контекст рисования>.moveTo(<горизонтальная координата>,
↳<вертикальная координата>)
```

Параметры этого метода задают горизонтальную и вертикальную координаты точки, в которую должно переместиться перо, в пикселах в виде чисел. Метод `moveTo` не возвращает результата.

Пример:

```
ctxCanvas.moveTo(200, 150);
```

Это выражение перемещает перо в центр канвы `cnv` — в точку с координатами [200,150].

Прямые линии

Прямые линии рисовать проще всего. Для этого используется метод `lineTo`:

```
<контекст рисования>.lineTo(<горизонтальная координата>,
↳<вертикальная координата>)
```

Начальная точка рисуемой прямой будет находиться в том месте, где в данный момент установлено перо (об этом уже говорилось ранее). Координаты конечной точки в пикселах задают параметры метода `lineTo`. Метод не возвращает результата.

После рисования прямой линии перо будет установлено в ее конечной точке. Мы можем прямо из этой точки начать рисование следующей линии контура.

Листинг 22.2

```
ctxCanvas.beginPath();
ctxCanvas.moveTo(20, 20);
ctxCanvas.lineTo(380, 20);
ctxCanvas.lineTo(200, 280);
ctxCanvas.closePath();
ctxCanvas.stroke();
```

Web-сценарий из листинга 22.2 рисует треугольник без заливки. Давайте рассмотрим последовательность действий.

1. Вызовом метода `beginPath` сообщаем Web-обозревателю, что собираемся рисовать контур сложной фигуры.
2. Методом `moveTo` устанавливаем перо в точку, где начнется рисование.
3. С помощью метода `lineTo` рисуем две линии, которые станут сторонами треугольника.
4. Третью сторону мы рисовать не будем, а лучше вызовем метод `closePath`, чтобы Web-обозреватель сам нарисовал ее, замкнув нарисованный нами контур.
5. Вызываем метод `stroke`, чтобы закончить рисование треугольника без заливки.

Дуги

Дуги рисуются тоже довольно просто. Для этого используется метод `arc`:

```
<контекст рисования>.arc(<горизонтальная координата>,
↳<вертикальная координата>, <радиус>, <начальный угол>, <конечный угол>,
↳true|false)
```

Первые два параметра задают горизонтальную и вертикальную координаты центра рисуемой дуги в виде числа в пикселах. Третий параметр определяет радиус дуги, также в пикселах и в виде числа. Четвертый и пятый параметры задают начальный и конечный углы дуги в радианах в виде чисел; эти углы отсчитываются от горизонтальной оси. Если шестой параметр имеет значение `true`, то дуга рисуется против часовой стрелки, а если `false` — по часовой стрелке. Метод `arc` не возвращает результата.

Рисование дуги начинается в точке, где в данный момент установлено перо. После рисования дуги перо будет установлено в точке, где кончается эта дуга.

Как уже говорилось, начальный и конечный углы рисуемой дуги задаются в радианах, а не в градусах. Пересчитает величину угла из градусов в радианы нам следующее выражение JavaScript:

```
radians = (Math.PI / 180) * degrees;
```

Здесь переменная `degrees` хранит значение угла в градусах, а переменная `radians` будет хранить то же значение, но в радианах. Свойство `PI` объекта JavaScript `Math` хранит значение числа π.

Вот Web-сценарий, который рисует окружность без заливки:

```
ctxCanvas.beginPath();
ctxCanvas.arc(200, 150, 100, 0, Math.PI * 2, false);
ctxCanvas.stroke();
```

Отметим, какие параметры метода `arc`, в частности, значения начального и конечного угла, мы задавали в этом случае.

Кривые Безье

Кривые Безье — это линии особой формы, описываемые тремя или четырьмя точками: начальной, конечной и одной или двумя контрольными. Начальная и конечная точки, как и в случае прямой линии, задают начало и конец кривой Безье, а *контрольные точки* формируют касательные, определяющие форму этой кривой.

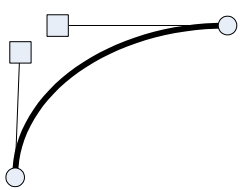


Рис. 22.1. Кривая Безье с двумя контрольными точками

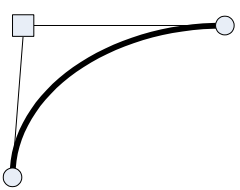


Рис. 22.2. Кривая Безье с одной контрольной точкой

На рис. 22.1 кривая Безье выделена утолщенной линией, ее начальная и конечная точки обозначены кружками, квадратики соответствуют контрольным точкам. Через каждую контрольную точку, а также через начальную и конечную точки кривой Безье проведены касательные (тонкие прямые линии) — они определяют форму кривой. Если мы мысленно переместим какую-либо из контрольных точек, то направление проведенной через нее касательной изменится, и, следовательно, изменится и форма кривой Безье.

На рис. 22.1 представлена кривая Безье с двумя контрольными точками. Такие кривые применяются чаще всего.

Но зачастую предпочтительнее использовать другую, "вырожденную", форму кривых Безье — с одной контрольной точкой (рис. 22.2).

На основе кривых Безье с одной контрольной точкой можно создавать дуги и рисовать секторы, в чем мы вскоре убедимся.

Для рисования кривых Безье с двумя контрольными точками предусмотрен метод `bezierCurveTo`:

```
<контекст рисования>.bezierCurveTo
↳(<горизонтальная координата первой контрольной точки>,
↳<вертикальная координата первой контрольной точки>,
↳<горизонтальная координата второй контрольной точки>,
↳<вертикальная координата второй контрольной точки>,
↳<горизонтальная координата конечной точки>,
↳<вертикальная координата конечной точки>)
```

Назначение параметров этого метода понятно из их описания. Все они задаются в пикселах в виде чисел. Метод не возвращает результата.

Рисование кривой Безье начинается в той точке, где в данный момент установлено перо. После рисования кривой перо устанавливается в ее конечную точку.

Вот Web-сценарий, рисующий кривую Безье с двумя контрольными точками:

```
ctxCanvas.beginPath();
ctxCanvas.moveTo(100, 100);
ctxCanvas.bezierCurveTo(120, 80, 160, 20, 100, 200);
ctxCanvas.stroke();
```

Рисование кривых Безье с одной контрольной точкой реализует метод `quadraticCurveTo`:

```
<контекст рисования>.quadraticCurveTo
↳(<горизонтальная координата контрольной точки>,
↳<вертикальная координата контрольной точки>,
↳<горизонтальная координата конечной точки>,
↳<вертикальная координата конечной точки>)
```

Описывать параметры этого метода также нет смысла — их назначение понятно. Все они задаются в пикселах в виде чисел. Метод не возвращает результата.

Рисование такой кривой Безье также начинается в той точке, где в данный момент установлено перо. После рисования кривой перо устанавливается в ее конечную точку.

Вот Web-сценарий, рисующий кривую Безье с одной контрольной точкой:

```
ctxCanvas.beginPath();
ctxCanvas.moveTo(100, 100);
ctxCanvas.quadraticCurveTo(200, 100, 200, 200);
ctxCanvas.stroke();
```

Получившаяся кривая будет иметь вид дуги.

Более сложный пример иллюстрирует листинг 22.3.

Листинг 22.3

```
ctxCanvas.beginPath();
ctxCanvas.strokeStyle = "red";
```

```
ctxCanvas.fillStyle = "red";
ctxCanvas.moveTo(100, 100);
ctxCanvas.quadraticCurveTo(200, 100, 200, 200);
ctxCanvas.lineTo(100, 200);
ctxCanvas.lineTo(100, 100);
ctxCanvas.fill();
```

Web-сценарий из листинга 22.3 рисует красный сектор окружности с красной же заливкой. Мы проводим кривую Безье с одной контрольной точкой, имеющую вид дуги, и соединяем ее начальную и конечную точки с центром воображаемой окружности.

Еще один пример приведен в листинге 22.4.

Листинг 22.4

```
ctxCanvas.beginPath();
ctxCanvas.moveTo(20, 0);
ctxCanvas.lineTo(180, 0);
ctxCanvas.quadraticCurveTo(200, 0, 200, 20);
ctxCanvas.lineTo(200, 80);
ctxCanvas.quadraticCurveTo(200, 100, 180, 100);
ctxCanvas.lineTo(20, 100);
ctxCanvas.quadraticCurveTo(0, 100, 0, 80);
ctxCanvas.lineTo(0, 20);
ctxCanvas.quadraticCurveTo(0, 0, 20, 0);
ctxCanvas.stroke();
```

Web-сценарий из листинга 22.4 рисует прямоугольник со скругленными углами. Попробуйте сами с ним разобраться.

Прямоугольники

Мы уже умеем рисовать прямоугольники, используя описанные ранее методы `strokeRect` и `fillRect`. Но прямоугольники, рисуемые этими методами, представляют собой независимые фигуры, не являющиеся частью какого-либо сложного контура. Если мы хотим нарисовать прямоугольник в составе сложного контура, нам придется прибегнуть к методу `rect`:

<контекст рисования>.rect(*<горизонтальная координата>*,
↳*<вертикальная координата>*, *<ширина>*, *<высота>*)

Первые два параметра задают горизонтальную и вертикальную координаты верхнего левого угла рисуемого прямоугольника в пикселах в виде чисел. Третий и четвертый параметры задают, соответственно, ширину и высоту прямоугольника, также в пикселах и тоже в виде чисел. Метод `rect` не возвращает результата.

После рисования прямоугольника методом `rect` перо устанавливается в точку с координатами [0,0], т. е. в верхний левый угол канвы.

Web-сценарий из листинга 22.5 рисует сложную фигуру, состоящую их трех накладывающихся друг на друга квадратов, и создает для нее заливку.

Листинг 22.5

```
ctxCanvas.beginPath();
ctxCanvas.rect(50, 50, 50, 50);
ctxCanvas.rect(75, 75, 50, 50);
ctxCanvas.rect(100, 100, 50, 50);
ctxCanvas.fill();
```

Задание стиля линий

Канва позволяет задать стиль линий, включающий в себя некоторые параметры, которые управляют формой их начальных и конечных точек и точек соединения линий друг с другом. Давайте их рассмотрим.

Свойство `lineCap` задает форму начальных и конечных точек линий. Его значение может быть одной из следующих строк:

- `"butt"` — начальная и конечная точки как таковые отсутствуют (значение по умолчанию);
- `"round"` — начальная и конечная точки имеют вид кружков;
- `"square"` — начальная и конечная точки имеют вид квадратиков.

Web-сценарий из листинга 22.6 рисует толстую прямую линию, начальная и конечная точки которой имеют вид кружков.

Листинг 22.6

```
ctxCanvas.beginPath();
ctxCanvas.lineWidth = 10;
ctxCanvas.lineCap = "round";
ctxCanvas.moveTo(20, 20);
ctxCanvas.lineTo(180, 20);
ctxCanvas.stroke();
```

Свойство `lineJoin` задает форму точек соединения линий друг с другом. Его значение может быть одной из следующих строк:

- `"miter"` — точки соединения имеют вид острого или тупого угла (значение по умолчанию);
- `"round"` — точки соединения, образующие острые углы, скругляются;
- `"bevel"` — острые углы, образуемые соединяющимися линиями, как бы срезаются.

Web-сценарий из листинга 22.7 рисует контур треугольника толстыми линиями, причем точки соединения этих линий, образующие острые углы, будут срезаться.

Листинг 22.7

```
ctxCanvas.beginPath();
ctxCanvas.lineWidth = 10;
ctxCanvas.lineJoin = "bevel";
ctxCanvas.moveTo(20, 20);
ctxCanvas.lineTo(380, 20);
ctxCanvas.lineTo(200, 280);
ctxCanvas.closePath();
ctxCanvas.stroke();
```

Свойство `miterLimit` задает дистанцию, на которую могут выступать острые углы, образованные соединением линий, от точки соединения, когда для свойства `lineJoin` задано значение `"miter"`. Углы, выступающие на бóльшую дистанцию, будут срезаны.

Значение данного свойства задается в пикселах в виде числа. Каково его значение по умолчанию, автору выяснить не удалось.

Листинг 22.8 содержит исправленный вариант Web-сценария, приведенного ранее.

Листинг 22.8

```
ctxCanvas.beginPath();
ctxCanvas.lineJoin = "miter";
ctxCanvas.lineWidth = 10;
ctxCanvas.miterLimit = 1;
ctxCanvas.moveTo(20, 20);
ctxCanvas.lineTo(380, 20);
ctxCanvas.lineTo(200, 280);
ctxCanvas.closePath();
ctxCanvas.stroke();
```

Дистанция, на которую могут выступать острые углы, образованные соединением линий контура, не должна превышать одного пиксела. Углы, выступающие на бóльшую дистанцию, будут срезаны.

Вывод текста

Было бы странно, если канва не позволяла выводить произвольный текст. Существуют два метода и несколько свойств для вывода текста.

Метод `strokeText` выводит заданный текст в указанное место. Текст рисуется в виде контура, без заливки; цвет контура задается значением свойства `strokeStyle`:

```
<контекст рисования>.strokeText(<выводимый текст>,
↳<горизонтальная координата>, <вертикальная координата>
↳[, <максимальная ширина>])
```

Первый параметр этого метода задает выводимый текст в виде строки. Второй и третий параметры задают координаты точки, в которой будет выведен текст, в пикселах в виде чисел. По умолчанию выводимый текст будет выровнен по левому краю относительно этой точки.

Четвертый, необязательный, параметр определяет максимальное значение ширины, которую может принять выводимый на канву текст. Если выводимый текст получается шире, канва выводит его либо шрифтом с уменьшенной шириной символов (если данный шрифт поддерживает такое начертание), либо шрифтом меньшего размера.

Метод `strokeText` не возвращает результата.

Пример:

```
ctxCanvas.strokeStyle = "blue";
ctxCanvas.strokeText("Всем привет!", 200, 50, 100);
```

Метод `fillText` также выводит заданный текст в указанное место. Однако текст этот представляет собой одну заливку, без контура; цвет заливки задается значением свойства `fillStyle`:

```
<контекст рисования>.fillText(<выводимый текст>,
<горизонтальная координата>, <вертикальная координата>
[, <максимальная ширина>])
```

Формат вызова этого метода такой же, как и у метода `strokeText`:

```
ctxCanvas.fillStyle = "yellow";
ctxCanvas.fillText("Всем пока!", 50, 100);
```

Свойство `font` позволяет задать параметры шрифта, которым будет выводиться текст. Параметры шрифта указывают в том же формате, что и у значения атрибута стиля `font` (см. *главу 8*), в виде строки:

```
ctxCanvas.fillStyle = "yellow";
ctxCanvas.font = "italic 12pt Verdana";
ctxCanvas.fillText("Всем пока!", 50, 100);
```

Свойство `textAlign` позволяет задать горизонтальное выравнивание выводимого текста относительно точки, в которой он будет выведен (координаты этой точки задаются вторым и третьим параметрами методов `strokeText` и `fillText`). Это свойство может принимать следующие значения:

- `"left"` — выравнивание по левому краю;
- `"right"` — выравнивание по правому краю;
- `"start"` — выравнивание по левому краю, если текст выводится по направлению слева направо, и по правому краю в противном случае (значение по умолчанию);
- `"end"` — выравнивание по правому краю, если текст выводится по направлению слева направо, и по левому краю в противном случае;
- `"center"` — выравнивание по центру.

Пример:

```
ctxCanvas.fillStyle = "yellow";
ctxCanvas.font = "italic 12pt Verdana";
ctxCanvas.textAlign = "center";
ctxCanvas.fillText("Всем пока!", 100, 100);
```

Свойство `textBaseline` позволяет задать вертикальное выравнивание выводимого текста относительно точки, в которой он будет выведен. Это свойство может принимать следующие значения:

- `"top"` — выравнивание по верху прописных (больших) букв;
- `"hanging"` — выравнивание по верху строчных (маленьких) букв;
- `"middle"` — выравнивание по середине строчных букв;
- `"alphabetic"` — выравнивание по базовой линии букв европейских алфавитов (значение по умолчанию);
- `"ideographic"` — выравнивание по базовой линии иероглифических символов (она находится чуть ниже базовой линии букв европейских алфавитов);
- `"bottom"` — выравнивание по низу букв.

Пример:

```
ctxCanvas.fillStyle = "yellow";
ctxCanvas.font = "italic 12pt Verdana";
ctxCanvas.textAlign = "center";
ctxCanvas.textBaseline = "top";
ctxCanvas.fillText("Всем пока!", 100, 100);
```

Использование сложных цветов

Ранее для линий и заливок у нас были только *простые*, однотонные, цвета. Настала пора познакомиться со средствами канвы для создания и использования *сложных* цветов: градиентных и графических.

Линейный градиентный цвет

В *линейном градиентном* цвете (или просто *линейном градиенте*) один простой цвет плавно переходит в другой при движении по прямой линии. Пример такого цвета — окраска заголовка окна в Windows 2000 и более поздних версиях Windows при выборе классической темы; там синий цвет плавно перетекает в белый.

Линейный градиентный цвет создают в три этапа.

Первый этап — вызов метода `createLinearGradient` — собственно создание линейного градиентного цвета:

```
<контекст рисования>.createLinearGradient
↳(<горизонтальная координата начальной точки>,
```

↳*<вертикальная координата начальной точки>*,
↳*<горизонтальная координата конечной точки>*,
↳*<вертикальная координата конечной точки>*)

Параметры этого метода определяют координаты начальной и конечной точки воображаемой прямой, по которой будет "распространяться" градиент. Они отсчитываются относительно канвы и задаются в пикселах в виде чисел.

Метод `createLinearGradient` возвращает экземпляр объекта `CanvasGradient`, представляющий созданный нами линейный градиент. Мы используем его для указания цветов, формирующих градиент.

Вот выражение, создающее линейный градиент, который будет "распространяться" по прямой с координатами начальной и конечной точек [0,0] и [100,50], и помещающее его в переменную `lgSample`:

```
var lgSample = ctxCanvas.createLinearGradient(0, 0, 100, 50);
```

Второй этап — расстановка так называемых *ключевых точек* градиента, определяющих позицию, в которой будет присутствовать "чистый" цвет. Между ключевыми точками будет наблюдаться переход между цветами. Таких ключевых точек может быть сколько угодно.

Ключевую точку ставят, вызвав метод `addColorStop` объекта `CanvasGradient`:

<градиент>.addColorStop(*<положение ключевой точки>*, *<цвет>*)

Первый параметр определяет относительное положение создаваемой ключевой точки на воображаемой прямой, по которой "распространяется" градиент. Он задается в виде числа от 0.0 (начало прямой) до 1.0 (конец прямой). Второй параметр задает цвет, который должен присутствовать в данной ключевой точке, в виде строки; при этом допустимы все форматы описания цвета, упомянутые в начале этой главы.

Метод `addColorStop` не возвращает значения.

Пример:

```
lgSample.addColorStop(0, "black");
lgSample.addColorStop(0.4, "rgba(0, 0, 255, 0.5)");
lgSample.addColorStop(1, "#FF0000");
```

Этот Web-сценарий создает на полученном нами ранее линейном градиенте три ключевые точки:

- расположенную в начальной точке воображаемой прямой и задающую черный цвет;
- расположенную в точке, отстоящей на 40% длины воображаемой прямой от ее начальной точки, и задающую полупрозрачный синий цвет;
- расположенную в конечной точке воображаемой прямой и задающую красный цвет.

Третий этап — собственно использование готового линейного градиента. Для этого представляющий его экземпляр объекта `CanvasGradient` следует присвоить свойству

strokeStyle или fillStyle. Первое свойство, как мы помним, задает цвет линий контуров, а второе — цвет заливок:

```
ctxCanvas.fillStyle = lgSample;
```

А теперь нам нужно рассмотреть один очень важный вопрос. И рассмотрим мы его на примере созданного ранее градиента.

Предположим, что мы нарисовали на канве три прямоугольника и применили к ним наш линейный градиент. Первый прямоугольник нарисован в точке [0,0] (в начале воображаемой прямой градиента, в смысле, в его первой ключевой точке), второй — в точке [30,20] (во второй ключевой точке), третий — в точке [80,40] (в конце градиента — его третьей ключевой точке). Иначе говоря, мы "расставили" наши прямоугольники во всех ключевых точках градиента.

Как будут окрашены эти прямоугольники? Давайте посмотрим.

- Первый прямоугольник будет окрашен, в основном, в черный цвет, заданный в первой ключевой точке градиента.
- Второй прямоугольник будет окрашен, в основном, в полупрозрачный синий цвет, заданный во второй ключевой точке градиента.
- Третий прямоугольник будет окрашен, в основном, в красный цвет, заданный в третьей ключевой точке градиента.

Следовательно, созданный нами градиент не "втиснулся" в каждый из нарисованных прямоугольников целиком, а как бы зафиксировался на самой канве, а прямоугольники только "проявили" фрагменты этого градиента, соответствующие их размерам. Другими словами, градиент задается для целой канвы, а фигуры, к которым он применен, окрашиваются соответствующими его фрагментами.

Если мы захотим, чтобы какая-то фигура была окрашена градиентом полностью, придется задать для этого градиента такие координаты воображаемой прямой, чтобы он "покрыл" всю фигуру. Иначе фигура будет содержать только часть градиента.

Листинг 22.9

```
var lgSample = ctxCanvas.createLinearGradient(0, 0, 100, 50);
lgSample.addColorStop(0, "black");
lgSample.addColorStop(0.4, "rgba(0, 0, 255, 0.5)");
lgSample.addColorStop(1, "#FF0000");
ctxCanvas.fillStyle = lgSample;
ctxCanvas.fillRect(0, 0, 200, 100);
```

Web-сценарий из листинга 22.9 рисует прямоугольник и заполняет его линейным градиентом, аналогичным созданному ранее в этом разделе. Попробуйте изменить координаты и размеры рисуемого прямоугольника и посмотрите, какая часть градиента в нем появится.

Радиальный градиентный цвет

Радиальный градиентный цвет (радиальный градиент) описывается двумя вложенными друг в друга окружностями. "Распространяется" он из внутренней окружности по направлению к внешней во все стороны. Ключевые точки такого градиента расставлены между этими окружностями.

Радиальный градиентный цвет также создают в три этапа.

Первый этап — вызов метода `createRadialGradient` — создание радиального градиентного цвета:

```
<контекст рисования>.createRadialGradient
↳(<горизонтальная координата центра внутренней окружности>,
↳<вертикальная координата центра внутренней окружности>,
↳<радиус внутренней окружности>,
↳<горизонтальная координата центра внешней окружности>,
↳<вертикальная координата центра внешней окружности>,
↳<радиус внешней окружности>)
```

Параметры этого метода задают координаты центров и радиусы обеих окружностей, описывающих радиальный градиент. Они задаются в пикселах в виде чисел.

Метод `createRadialGradient` возвращает экземпляр объекта `CanvasGradient`, представляющий созданный нами радиальный градиент.

Пример:

```
var rgSample = ctxCanvas.createRadialGradient(100, 100, 10, 150, 100, 120);
```

Это выражение создает радиальный градиент и помещает его в переменную `rgSample`. Созданный градиент будет "распространяться" от внутренней окружности, центр которой находится в точке с координатами [100,100], а радиус равен 10 пикселам, к внешней окружности с центром в точке [150,100] и радиусом 120 пикселов.

Второй этап — расстановка ключевых точек — выполняется с помощью уже знакомого нам метода `addColorStop` объекта `CanvasGradient`:

```
<градиент>.addColorStop(<положение ключевой точки>, <цвет>)
```

Первый параметр определяет относительное положение создаваемой ключевой точки на промежутке между внутренней и внешней окружностями. Он задается в виде числа от 0.0 (начало промежутка, т. е. внутренняя окружность) до 1.0 (конец промежутка, т. е. внешняя окружность). Второй параметр, как мы уже знаем, задает цвет, который должен присутствовать в данной ключевой точке.

Как и линейный градиент, радиальный может содержать сколько угодно ключевых точек.

Пример:

```
rgSample.addColorStop(0, "#CCCCCC");
rgSample.addColorStop(0.8, "black");
rgSample.addColorStop(1, "#00FF00");
```

Этот Web-сценарий создает на полученном нами ранее радиальном градиенте три ключевые точки:

- расположенную в начальной точке воображаемого промежутка между окружностями (т. е. на внутренней окружности) и задающую серый цвет;
- расположенную в точке, отстоящей на 80% длины воображаемого промежутка между окружностями от его начальной точки, и задающую черный цвет;
- расположенную в конечной точке воображаемого промежутка между окружностями (т. е. на внешней окружности) и задающую зеленый цвет.

Третий этап — использование готового радиального градиента — выполняется так же, как для линейного градиента, т. е. присваиванием его свойству `strokeStyle` или `fillStyle`:

```
ctxCanvas.fillStyle = rgSample;
```

Радиальный градиент ведет себя точно так же, как линейный — фиксируется на канве и частично "проявляется" на фигурах, к которым применен.

Листинг 22.10 иллюстрирует пример.

Листинг 22.10

```
var rgSample = ctxCanvas.createRadialGradient(100, 100, 10, 150, 100,
120);
rgSample.addColorStop(0, "#CCCCCC");
rgSample.addColorStop(0.8, "black");
rgSample.addColorStop(1, "#00FF00");
ctxCanvas.fillStyle = rgSample;
ctxCanvas.fillRect(0, 0, 200, 200);
```

Web-сценарий из листинга 22.10 рисует прямоугольник и заполняет его радиальным градиентом, аналогичным созданному ранее в этом разделе. Отметим, что центры внутренней и внешней окружностей, описывающих этот градиент, различаются, за счет чего достигается весьма примечательный эффект, который лучше видеть своими глазами.

Графический цвет

Графический цвет — это обычное графическое изображение, которым закрашиваются линии или заливки. Таким графическим изображением может быть содержимое как обычного графического файла, так и другой канвы.

Графический цвет создают в три этапа.

Первый этап необходим только в том случае, если мы используем в качестве цвета содержимое графического файла. Файл нужно как-то загрузить, удобнее всего — с помощью объекта Web-обозревателя `Image`, который представляет графическое изображение, хранящееся в файле.

Сначала с помощью знакомого нам по *главе 14* оператора `new` нам потребуется создать экземпляр объекта `Image`:

```
var imgSample = new Image();
```

Объект `Image` поддерживает свойство `src`, задающее интернет-адрес загружаемого графического файла в виде строки. Если присвоить этому свойству интернет-адрес какого-либо файла, данный файл тотчас будет загружен:

```
imgSample.src = "graphic_color.jpg";
```

В дальнейшем мы можем использовать данный экземпляр объекта `Image` для создания графического цвета.

Второй этап — собственно создание графического цвета с помощью метода `createPattern`:

```
<контекст рисования>.createPattent(<графическое изображение или канва>,
↳<режим повторения>)
```

Первый параметр задает графическое изображение в виде экземпляра объекта `Image` или канву в виде экземпляра объекта `HTMLCanvasElement`.

Часто бывает так, что размеры заданного графического изображения меньше, чем фигуры, к которой должен быть применен графический цвет. В этом случае изображение повторяется столько раз, чтобы полностью "вымостить" линию или заливку. Режим такого повторения задает второй параметр метода `createPattern`. Его значение должно быть одной из следующих строк:

- `"repeat"` — изображение будет повторяться по горизонтали и вертикали;
- `"repeat-x"` — изображение будет повторяться только по горизонтали;
- `"repeat-y"` — изображение будет повторяться только по вертикали;
- `"no-repeat"` — изображение не будет повторяться никогда; в этом случае часть фигуры останется не занятой им.

Метод `createPattern` возвращает экземпляр объекта `CanvasPattern`, представляющий созданный нами графический цвет:

```
var cpSample = ctxCanvas.createPattern(imgSample, "repeat");
```

Третий этап — использование готового графического цвета — выполняется так же, как для градиентов, т. е. присваиванием его свойству `strokeStyle` или `fillStyle`.

Пример:

```
ctxCanvas.fillStyle = cpSample;
ctxCanvas.fillRect(0, 0, 200, 100);
```

Этот Web-сценарий рисует прямоугольник с заливкой на основе созданного нами ранее графического цвета.

Графический цвет не фиксируется на канве, а полностью применяется к рисуемой фигуре. В этом его принципиальное отличие от градиентов.

НА ЗАМЕТКУ

В приведенном ранее примере мы предположили, что файл graphic_color.jpg имеет небольшие размеры или уже присутствует в кэше Web-обозревателя и поэтому загрузится очень быстро. Но если он, так сказать, "задержится в пути", Web-сценарий не выполнится. Поэтому изображения, хранящиеся в других файлах, выводятся по иной методике, которую мы скоро рассмотрим.

Вывод внешних изображений

Внешним по отношению к канве называется графическое изображение, хранящееся в отдельном файле, или содержимое другой канвы. Канва предоставляет довольно мощные средства для вывода таких изображений: мы можем изменить размеры изображения и даже вывести только его часть.

Вывести внешнее изображение на канву проще всего методом `drawImage`, точнее, его сокращенным форматом:

```
<контекст рисования>.drawImage(<графическое изображение или канва>,
↳<горизонтальная координата>, <вертикальная координата>)
```

Первый параметр задает графическое изображение в виде экземпляра объекта `Image` или канву в виде экземпляра объекта `HTMLCanvasElement`. Второй и третий параметры определяют координаты точки канвы, где должен находиться верхний левый угол выводимого изображения; они задаются в пикселах в виде чисел. Метод `drawImage` не возвращает результата.

Вот Web-сценарий, который загружает изображение из файла someimage.jpg и выводит его на канву так, чтобы его верхний левый угол находился в точке [0,0], т. е. в верхнем левом углу канвы:

```
var imgSample = new Image();
imgSample.src = "someimage.jpg";
ctxCanvas.drawImage(imgSample, 0, 0);
```

Если нам нужно при выводе внешнего изображения изменить его размеры, к нашим услугам расширенный формат метода `drawImage`:

```
<контекст рисования>.drawImage(<графическое изображение или канва>,
↳<горизонтальная координата>, <вертикальная координата>
↳<ширина>, <высота>)
```

Первые три параметра нам уже знакомы. Четвертый и пятый параметры задают, соответственно, ширину и высоту выводимого изображения в пикселах в виде чисел.

Вот пример Web-сценария, который выводит загруженное ранее изображение, растягивая его так, чтобы занять канву целиком:

```
ctxCanvas.drawImage(imgSample, 0, 0, 400, 300);
```

Рассмотрим теперь самый сложный случай — вырезание из внешнего изображения фрагмента и вывод его на канву с изменением размеров. Для этого применяется третий по счету формат метода `drawImage`:

```
<контекст рисования>.drawImage(<графическое изображение или канва>,
↳<горизонтальная координата вырезаемого фрагмента>,
↳<вертикальная координата вырезаемого фрагмента>
↳<ширина вырезаемого фрагмента>, <высота вырезаемого фрагмента>,
↳<горизонтальная координата вывода>, <вертикальная координата вывода>,
↳<ширина вывода>, <высота вывода>)
```

Первый параметр нам уже знаком и задает внешнее изображение.

Второй и третий параметры определяют координаты верхнего левого угла вырезаемого из внешнего изображения фрагмента. Они задаются относительно внешнего изображения в пикселах в виде чисел.

Четвертый и пятый параметры определяют ширину и высоту вырезаемого из внешнего изображения фрагмента. Они также задаются относительно внешнего изображения в пикселах в виде чисел.

Шестой и седьмой параметры определяют координаты точки канвы, где должен находиться верхний левый угол выводимого фрагмента внешнего изображения. Они задаются относительно канвы в пикселах в виде чисел.

Восьмой и девятый параметры определяют ширину и высоту выводимого фрагмента внешнего изображения в пикселах в виде чисел.

Вот Web-сценарий, который вырезает из загруженного ранее изображения фрагмент с верхним левым углом в точке [20,40], шириной 40 и высотой 20 пикселов и выводит этот фрагмент на канву, растягивая его так, чтобы занять канву целиком:

```
ctxCanvas.drawImage(imgSample, 20, 40, 40, 20, 0, 0, 400, 300);
```

Все приведенные ранее примеры подразумевают, что файл, хранящий внешнее изображение, загружается очень быстро (так может случиться, если файл имеет небольшие размеры или уже находится в кэше Web-обозревателя.) Но чаще всего случается так, что файл не успевает загрузиться к тому моменту, когда начнет выполняться выводящий его на канву код, и мы получим ошибку выполнения Web-сценария. Как избежать этого?

Очень просто. Объект `Image` поддерживает событие `onload`, возникающее после окончания загрузки изображения. Данному событию соответствует одноименное свойство, которому следует присвоить функцию — обработчик этого события. Web-сценарий из листинга 22.11 иллюстрирует сказанное.

Листинг 22.11

```
var imgSample = new Image();
function imgOnLoad() {
  ctxCanvas.drawImage(imgSample, 20, 40, 40, 20, 0, 0, 400, 300);
}
imgSample.src = "someimage.jpg";
imgSample.onload = imgOnLoad;
```

НА ЗАМЕТКУ

Именно так выполняется привязка обработчиков к событиям некоторых объектов Web-обозревателя — присваиванием функции-обработчика свойству, которое соответствует нужному событию. Можно ли использовать для этого методы библиотеки Ext Core, автор не проверял.

Создание тени у рисуемой графики

Еще канва позволяет создавать тень у всех рисуемых фигур. Для задания ее параметров применяют четыре свойства, которые мы сейчас рассмотрим.

Свойства `shadowOffsetX` и `shadowOffsetY` задают смещение тени, соответственно, по горизонтали и вертикали относительно фигуры в пикселах в виде чисел. Положительные значения смещают тень вправо и вниз, а отрицательные — влево и вверх. Значения этих свойств по умолчанию — 0, т. е. фактически отсутствие тени.

Пример:

```
ctxCanvas.shadowOffsetX = 2;
ctxCanvas.shadowOffsetY = -2;
```

Свойство `shadowBlur` задает степень размытия тени в виде числа. Чем больше это число, тем более размыта тень. Значение по умолчанию — 0, т. е. отсутствие размытия.

Пример:

```
ctxCanvas.shadowBlur = 4;
```

Свойство `shadowColor` задает цвет тени. Цвет задается в виде строки в любом из форматов, описанных в начале этой главы. Значение по умолчанию — черный непрозрачный цвет.

Пример:

```
ctxCanvas.shadowColor = "rgba(128, 128, 128, 0.5)";
```

После того как мы зададим параметры тени, они будут применяться ко всей графике, которую мы далее нарисуем. На параметры тени уже нарисованной графики они влияния не окажут.

Пример:

```
ctxCanvas.fillText("Двое: я и моя тень.", 150, 50);
```

Преобразования системы координат

Преобразования — это различные действия (изменение масштаба, поворот и перемещение точки начала координат), которые мы можем выполнить над системой координат канвы.

При выполнении преобразования изменяется только система координат канвы. Рисуемая после этого графика будет создаваться в измененной системе координат; а нарисованная графика остается неизменной.

Сохранение и загрузка состояния

Первое, что нам нужно рассмотреть применительно к преобразованиям, — сохранение и загрузка состояния канвы. Эти возможности нам очень пригодятся в дальнейшем.

При сохранении состояния канвы сохраняются:

- все заданные трансформации (будут описаны далее);
- значения свойств `globalAlpha`, `globalCompositeOperation` (будет описано далее), `fillStyle`, `lineCap`, `lineJoin`, `lineWidth`, `miterLimit` и `strokeStyle`;
- все заданные маски (будут описаны далее).

Сохранение состояния канвы выполняет метод `save`. Он не принимает параметров и не возвращает результата.

Состояние канвы сохраняется в памяти компьютера и впоследствии может быть восстановлено. Более того, сохранять состояние канвы можно несколько раз; при этом все предыдущие состояния остаются в памяти и их также можно восстановить.

Восстановить сохраненное ранее состояние можно вызовом метода `restore`. Он не принимает параметров и не возвращает результата.

При вызове этого метода будет восстановлено самое последнее из сохраненных состояний канвы. При последующем его вызове будет восстановлено предпоследнее сохраненное состояние и т. д. Этой особенностью часто пользуются для создания сложной графики.

Перемещение начала координат канвы

Мы можем переместить начало координат канвы в любую другую ее точку. После этого все координаты будут отсчитываться от нового начала координат.

Для перемещения начала координат канвы в другую точку достаточно вызвать метод `translate`:

```
<контекст рисования>.translate(<горизонтальная координата>,
↳<вертикальная координата>)
```

Параметры метода `translate` определяют координаты точки, в которой должно находиться новое начало координат канвы. Они отсчитываются от текущего начала координат, измеряются в пикселах и задаются в виде чисел. Метод не возвращает результата.

При перемещении начала координат канвы будут учитываться все трансформации, примененные к канве ранее. Значит, если мы ранее переместили начало координат в точку [50,50] и теперь снова перемещаем его, уже в точку [100,50], в результате начало координат окажется в точке [150,100], если отсчитывать от верхнего левого угла канвы (начала системы координат по умолчанию).

Листинг 22.12 иллюстрирует пример.

Листинг 22.12

```
ctxCanvas.save();
ctxCanvas.translate(100, 100);
ctxCanvas.fillRect(0, 0, 50, 50);
ctxCanvas.translate(100, 100);
ctxCanvas.fillRect(0, 0, 50, 50);
ctxCanvas.restore();
ctxCanvas.fillRect(0, 0, 50, 50);
```

Web-сценарий из листинга 22.12 делает следующее:

1. Сохраняет текущее состояние канвы.
2. Перемещает начало координат в точку [100,100].
3. Рисует квадрат размерами 50×50 пикселов, верхний левый угол которого находится в точке начала координат.
4. Опять перемещает начало координат в точку [100,100]. Обратим внимание, что координаты этой точки теперь отсчитываются от нового начала системы координат, установленного предыдущим вызовом метода `translate`.
5. Опять рисует квадрат размерами 50×50 пикселов, верхний левый угол которого находится в начале координат.
6. Восстанавливает сохраненное состояние канвы, в том числе и положение начала системы координат (это положение по умолчанию, т. е. верхний левый угол канвы).
7. Рисует третий по счету квадрат размерами 50×50 пикселов, верхний левый угол которого находится в начале координат.

В результате мы увидим три квадрата, расположенные на воображаемой диагонали, тянущейся от верхнего левого угла канвы вниз и вправо.

> ***Внимание!***
>
> К сожалению, не существует простого способа вернуться к системе координат по умолчанию. Единственная возможность — сохранить состояние системы координат перед любыми трансформациями и потом восстановить его.

Поворот системы координат

Метод `rotate` позволяет повернуть оси системы координат на произвольный угол вокруг точки начала координат; при этом поворот будет выполняться по часовой стрелке:

<контекст рисования>.rotate(*<угол поворота>*)

Единственный параметр метода задает угол поворота системы координат в виде числа в радианах; этот угол отсчитывается от горизонтальной оси. Метод не возвращает результата.

При повороте системы координат будут учитываться все трансформации, примененные к канве ранее. Так, если мы ранее переместили начало системы координат в другую точку и теперь поворачиваем систему координат на какой-то угол, она будет повернута вокруг нового начала координат. А если мы после этого снова повернем систему координат на какой-то угол, она будет повернута относительно текущего положения горизонтальной оси — уже повернутой ранее.

Листинг 22.13 иллюстрирует пример.

Листинг 22.13

```
ctxCanvas.translate(200, 150);
for (var i = 0; i < 3; i++) {
  ctxCanvas.rotate(Math.PI / 6);
  ctxCanvas.strokeRect(-50, -50, 100, 100);
}
```

Web-сценарий из листинга 22.13 сдвигает начало координат в центр канвы (точку [200,150]), после чего трижды поворачивает систему координат на π/6 радиан (30°) и рисует в центре канвы квадрат без заливки. Обратим внимание, что каждый последующий поворот системы координат выполняется с учетом того, что она уже была повернута ранее.

Изменение масштаба системы координат

Метод `scale` дает возможность изменить масштаб системы координат канвы в бóльшую или меньшую сторону:

<контекст рисования>.scale(*<масштаб по горизонтали>*,
↳*<масштаб по вертикали>*)

Параметры этого метода задают масштаб для горизонтальной и вертикальной оси системы координат в виде чисел. Числа меньше 1.0 задают уменьшение масштаба, а числа больше 1.0 — увеличение; если нужно оставить масштаб по какой-то из осей неизменным, достаточно указать значение 1.0 соответствующего параметра. Метод `scale` не возвращает результата.

При изменении масштаба координат канвы будут учитываться все трансформации, примененные к канве ранее: перемещения начала координат, повороты и изменения масштаба.

Листинг 22.14 иллюстрирует пример.

Листинг 22.14

```
ctxCanvas.save();
ctxCanvas.strokeRect(0, 0, 50, 50);
ctxCanvas.scale(3, 1);
ctxCanvas.strokeRect(0, 0, 50, 50);
ctxCanvas.restore();
```

```
ctxCanvas.save();
ctxCanvas.scale(1, 3);
ctxCanvas.strokeRect(0, 0, 50, 50);
ctxCanvas.restore();
ctxCanvas.save();
ctxCanvas.scale(3, 3);
ctxCanvas.strokeRect(0, 0, 50, 50);
ctxCanvas.restore();
```

Web-сценарий из листинга 22.14 делает следующее:

1. Сохраняет текущее состояние канвы.
2. Рисует квадрат размерами 50×50 пикселов, верхний левый угол которого находится в начале координат.
3. Увеличивает масштаб горизонтальной координатной оси в 3 раза.
4. Рисует второй квадрат размерами 50×50 пикселов, верхний левый угол которого находится в начале координат.
5. Восстанавливает сохраненное ранее состояние канвы и сохраняет его снова.
6. Увеличивает масштаб вертикальной координатной оси в 3 раза.
7. Рисует третий квадрат размерами 50×50 пикселов, верхний левый угол которого находится в начале координат.
8. Восстанавливает сохраненное ранее состояние канвы и сохраняет его снова.
9. Увеличивает масштаб обоих координатных осей в 3 раза.
10. Рисует четвертый квадрат размерами 50×50 пикселов, верхний левый угол которого находится в начале координат.
11. Восстанавливает сохраненное ранее состояние канвы.

В результате мы увидим четыре прямоугольника с реальными размерами 50×50, 150×50, 50×150 и 150×150 пикселов.

Управление наложением графики

Когда мы рисуем новую фигуру на том месте канвы, где уже присутствует ранее нарисованная фигура, новая фигура накладывается на старую, перекрывая ее. Это поведение канвы по умолчанию, которое мы можем изменить.

Для управления наложением графики предусмотрено свойство `globalCompositeOperation`.

Вот его допустимые значения:

- `"source-over"` — новая фигура накладывается на старую, перекрывая ее (значение по умолчанию);
- `"destination-over"` — новая фигура перекрывается старой;

- "source-in" — отображается только та часть новой фигуры, которая накладывается на старую. Остальные части новой и старой фигур не выводятся;
- "destination-in" — отображается только та часть старой фигуры, на которую накладывается новая. Остальные части новой и старой фигур не выводятся;
- "source-out" — отображается только та часть новой фигуры, которая не накладывается на старую. Остальные части новой фигуры и вся старая фигура не выводятся;
- "destination-out" — отображается только та часть старой фигуры, на которую не накладывается новая. Остальные части новой фигуры и вся старая фигура не выводятся;
- "source-atop" — отображается только та часть новой фигуры, которая накладывается на старую; остальная часть новой фигуры не выводится. Старая фигура выводится целиком и находится ниже новой;
- "destination-atop" — отображается только та часть старой фигуры, которая накладывается на новую; остальная часть старой фигуры не выводится. Новая фигура выводится целиком и находится ниже старой;
- "lighter" — цвета накладывающихся частей старой и новой фигур складываются, результирующий цвет получается более светлым, окрашиваются накладывающиеся части фигур;
- "darker" — цвета накладывающихся частей старой и новой фигур вычитаются, в полученный цвет, который получается более темным, окрашиваются накладывающиеся части фигур;
- "xor" — отображаются только те части старой и новой фигур, которые не накладываются друг на друга;
- "copy" — выводится только новая фигура; все старые фигуры удаляются с канвы.

Заданный нами способ наложения графики действует только для графики, которую мы нарисуем после этого. На уже нарисованную графику он влияния не оказывает.

Листинг 22.15 иллюстрирует пример.

Листинг 22.15

```
ctxCanvas.fillStyle = "blue";
ctxCanvas.fillRect(0, 50, 400, 200);
ctxCanvas.fillStyle = "red";
ctxCanvas.globalCompositeOperation = "source-over";
ctxCanvas.fillRect(100, 0, 200, 300);
```

Web-сценарий из листинга 22.15 рисует два накладывающихся прямоугольника разных цветов и позволит изучить поведение канвы при разных значениях свойства `globalCompositeOperation`. Изменяем значение этого свойства, перезагружаем Web-страницу нажатием клавиши <F5> и смотрим, что получится.

Создание маски

О масках мы уже знаем из *главы 21*. В терминологии канвы так называется особая фигура, задающая своего рода "окно", сквозь которое будет видна часть графики, нарисованной на канве. Вся графика, не попадающая в это "окно", будет скрыта. При этом сама маска на канву не выводится.

Маской может быть только сложный контур, рисование которого описано ранее. И создается она примерно так же.

Вот последовательность действий.

1. Рисуем сложный контур, который станет маской.
2. Обязательно делаем его закрытым.
3. Вместо вызова методов `stroke` или `fill` вызываем метод `clip`, который не принимает параметров и не возвращает результата.
4. Рисуем графику, которая будет находиться под маской.

В результате нарисованная нами на шаге 4 графика будет частично видна сквозь маску. Требуемый результат достигнут.

Листинг 22.16 иллюстрирует пример.

Листинг 22.16

```
ctxCanvas.beginPath();
ctxCanvas.moveTo(100, 150);
ctxCanvas.lineTo(200, 0);
ctxCanvas.lineTo(200, 300);
ctxCanvas.closePath();
ctxCanvas.clip();
ctxCanvas.fillRect(0, 100, 400, 100);
```

Web-сценарий из листинга 22.16 сначала рисует маску в виде треугольника, а потом — прямоугольник, часть которого будет видна сквозь маску.

Создание графического логотипа Web-сайта

Вооружившись необходимыми знаниями о канве HTML 5, контексте рисования и его свойствах и методах, давайте попрактикуемся в Web-художествах. Создадим графический логотип для нашего Web-сайта, который поместим в контейнер `cheader` вместо маловыразительного текстового заголовка.

Сначала сформулируем требования.

- Логотип нашего Web-сайта будет представлять собой подчеркнутую надпись "Справочник по HTML и CSS" с тенью.
- Ширина логотипа будет такой, чтобы занимать все пространство между левым краем контейнера `cheader` и левым краем Web-формы поиска.

- Ширина логотипа будет меняться при изменении размеров окна Web-обозревателя.

Откроем Web-страницу index.htm в Блокноте, удалим все содержимое контейнера `cheader` и вставим в него такой HTML-код:

```
<CANVAS ID="cnv" WIDTH="600" HEIGHT="80"></CANVAS>
```

Он создает канву `cnv`, в которой и будет рисоваться логотип.

Так, канва у нас готова. Теперь нам нужно написать Web-сценарий, который будет получать размеры канвы и рисовать в ней логотип таким образом, чтобы он занял канву целиком.

Откроем файл Web-сценария main.js в Блокноте. И подумаем.

Где нам поместить код, выполняющий рисование логотипа? Может быть, в теле функции, передаваемой параметром методу `onReady` объекта `Ext`? Тогда логотип будет нарисован всего однажды — после загрузки Web-страницы.

Но мы хотим сделать так, чтобы ширина логотипа менялась при изменении ширины окна Web-обозревателя. Для этого нам следует сделать две вещи. Во-первых, придется в ответ на изменение ширины окна менять размеры канвы `cnv` — это очевидно. Во-вторых, понадобится после каждого изменения размеров канвы перерисовывать логотип — с учетом изменившихся размеров канвы.

Вывод: поместим код, выполняющий рисование логотипа, в тело функции `adjustContainers`. Эта функция, как мы помним, устанавливает размеры контейнеров, составляющих дизайн нашей Web-страницы, и выполняется при каждом изменении размеров окна Web-обозревателя — как раз то, что нам нужно.

Поместим в конец тела функции `adjustContainers` два выражения:

```
Ext.get("cnv").set({ width: elCSearch.getX() - 40 });
drawHeader();
```

Первое выражение устанавливает ширину канвы `cnv`, чтобы она заняла все пространство между левым краем контейнера `cheader` и левым краем Web-формы поиска. Нужное значение ширины получается следующим образом.

1. Берется значение горизонтальной координаты свободного контейнера `csearch`, в котором находится Web-форма поиска (см. *главу 21*). (Контейнер `csearch` хранится в переменной `elCSearch`.) Получается значение ширины, которую может занять канва, без учета внутренних отступов.
2. Из полученной ширины вычитается 20 — размер внутреннего отступа слева в пикселах, заданного в именованном стиле, привязанном к контейнеру `cheader`.
3. Из полученной разности вычитается еще 20 (итого получается 40) — размер отступа между правым краем канвы и левым краем контейнера `csearch` с Web-формой поиска. Это нужно, чтобы канва не примыкала к Web-форме поиска вплотную.

Полученное значение ширины присваивается атрибуту `WIDTH` тега `<CANVAS>` с помощью метода `set` объекта Ext Core `Element` (см. *главу 15*).

Второе выражение вызывает функцию `drawHeader`, которая и выполнит рисование логотипа. Листинг 22.17 содержит код, который объявляет эту функцию; мы можем поместить его в любое место файла main.js.

Листинг 22.17

```
function drawHeader() {
  var elCanvas = Ext.get("cnv");
  var cnvWidth = elCanvas.getAttribute("width");
  var ctx = elCanvas.dom.getContext("2d");
  ctx.beginPath();
  ctx.strokeStyle = "#B1BEC6";
  ctx.moveTo(0, 60);
  ctx.lineTo(cnvWidth, 60);
  ctx.stroke();
  ctx.shadowOffsetX = 2;
  ctx.shadowOffsetY = 2;
  ctx.shadowBlur = 2;
  ctx.shadowColor = "#CDD9DB";
  ctx.font = "normal 20pt Arial";
  ctx.textAlign = "right";
  ctx.textBaseline = "bottom";
  ctx.fillStyle = "#3B4043";
  ctx.scale(2, 1.3);
  ctx.fillText("Справочник по HTML и CSS", cnvWidth / 2, 60 / 1.3,
  cnvWidth / 2);
}
```

Рассмотрим листинг 22.17 по частям.

Сначала получаем канву `cnv`:

```
var elCanvas = Ext.get("cnv");
```

Затем получаем текущую ширину канвы:

```
var cnvWidth = elCanvas.getAttribute("width");
```

Рисуем горизонтальную линию, которая "вытянется" на всю ширину канвы и подчеркнет текст заголовка, который мы выведем потом:

```
ctx.beginPath();
ctx.strokeStyle = "#B1BEC6";
ctx.moveTo(0, 60);
ctx.lineTo(cnvWidth, 60);
ctx.stroke();
```

Задаем параметры тени для текста:

```
ctx.shadowOffsetX = 2;
ctx.shadowOffsetY = 2;
ctx.shadowBlur = 2;
ctx.shadowColor = "#CDD9DB";
```

Задаем шрифт текста. Берем его параметры из стиля переопределения тега `<H1>`, созданного нами в таблице стилей main.css:

```
ctx.font = "normal 20pt Arial";
```

Задаем для текста горизонтальное выравнивание по правому краю и вертикальное выравнивание по нижнему краю символов:

```
ctx.textAlign = "right";
ctx.textBaseline = "bottom";
```

Так нам будет проще вычислять координаты для вывода текста.

Задаем цвет заливки — он станет цветом выводимого текста:

```
ctx.fillStyle = "#3B4043";
```

Увеличиваем масштаб системы координат канвы:

```
ctx.scale(2, 1.3);
```

Отметим, что масштаб у горизонтальной оси координат больше, чем у вертикальной, — значит, текст будет растянут по горизонтали.

Выводим текст "Справочник по HTML и CSS" в виде заливки без контура:

```
ctx.fillText("Справочник по HTML и CSS", cnvWidth / 2, 60 / 1.3,
cnvWidth / 2);
```

Отметим несколько моментов.

- В качестве горизонтальной координаты вывода текста мы указали ширину канвы. Если учесть, что ранее мы задали горизонтальное выравнивание по правому краю, текст будет выровнен по правому краю канвы.
- В качестве вертикальной координаты вывода текста мы указали позицию, на которой находится уже нарисованная нами горизонтальная линия. Если учесть, что мы ранее задали вертикальное выравнивание по нижнему краю символов, получится так, что текст будет находиться выше этой линии; таким образом, линия подчеркнет текст.
- Мы указали максимальную ширину выводимого текста, равной ширине канвы. Благодаря этому текст при любом изменении ширины окна Web-обозревателя не вылезет за края канвы (подробнее см. в разделе, посвященном выводу текста).
- Вероятно, это самый важный момент. Текст мы вывели после изменения масштаба системы координат, но значения координат, которые указали при выводе, принадлежат старой системе координат. Чтобы преобразовать их в новой системе координат, мы разделим их на соответствующие величины масштабов (2 и 1,3).

Выполнение функции `drawHeader` закончено.

Откроем Web-страницу в Web-обозревателе. Посмотрим на новый логотип. Конечно, если постараться, можно сделать и лучше. Пусть это будет вашим домашним заданием.

На этом автор прощается с вами. Счастливо вам доделать Web-сайт — справочник по HTML и CSS! И успехов на поприще современного Web-дизайна!

Заключение

Вот и закончилась эта книга...

Программное ядро (или, на жаргоне программистов, "движок") нашего Web-сайта — справочника по HTML и CSS — готово и отлажено. Осталось только создать остальные Web-страницы, описывающие многочисленные теги HTML, атрибуты стиля CSS и примеры их использования. Мы ведь их так и не написали, увлекшись Web-дизайном и Web-программированием...

Мы познакомились с азами Web-дизайна, языками HTML, CSS и JavaScript, принципами Web-программирования и библиотекой Ext Core. И не только познакомились, но и применили полученные знания на практике, создав Web-сайт. Наш проект, хоть и невелик по объему, но зато использует самые "горячие" новинки Web-дизайна и Web-программирования. Подгружаемое и генерируемое содержимое, базы данных, семантическая разметка, интерактивные элементы — такое не везде встретишь.

А еще мы активно применяли новинки, которые несут нам HTML 5 и CSS 3. Многие Web-обозреватели уже сейчас поддерживают некоторые нововведения этих версий, а вскоре станут поддерживать и остальные их возможности. Ведь время идет, "черновые" стандарты становятся окончательными, а разработчики создают новые программы, которые будут поддерживать эти стандарты.

Конечно, не все было описано в этой книге. Многое осталось нерассмотренным. Но ведь окончательные редакции стандартов HTML 5 и CSS 3 еще далеки от завершения, да и Web-обозревателей, полностью соответствующих даже их "черновым" редакциям, также пока нет. Автор просто описал то, что, по его скромному мнению, понадобится даже неопытным Web-дизайнерам, и пригодится уже сейчас. А все прочее...

А все прочее описано в других материалах, которых хватает в Интернете. В табл. 3.1 приведены интернет-адреса некоторых Web-сайтов, которые будут очень полезны Web-дизайнеру любого уровня подготовки и которыми пользовался сам автор при написании этой книги. Правда, все они на английском языке, но этот язык должен знать любой грамотный компьютерщик — вопрос даже не обсуждается!

Таблица 3.1. *Некоторые Web-сайты с материалами по теме книги*

Интернет-адрес	Описание
https://developer.mozilla.org/en	Mozilla Developers Network (MDN). Раздел для разработчиков на Web-сайте сообщества Mozilla, создателя Web-обозревателя Firefox. Справочники по HTML, CSS, JavaScript, DOM, многочисленные статьи с примерами. Крайне рекомендуется, несмотря на несколько запутанную навигацию
http://dev.opera.com/	Opera Developer Community (DevOpera). Раздел для разработчиков на Web-сайте Opera, создателей одноименного Web-обозревателя. Многочисленные интереснейшие статьи, в том числе по HTML 5 и CSS 3
http://www.w3.org/	Web-сайт World Wide Web Consortium (W3C). Все стандарты, окончательные и "черновые", статьи для разработчиков с примерами
http://www.sencha.com/products/core/?ref=family	Раздел на Web-сайте, посвященный библиотеке Ext Core. Web-страница загрузки, руководство разработчика с примерами, справочник, форумы поддержки

Вот теперь действительно все! До свидания!

Владимир Дронов

ПРИЛОЖЕНИЕ

Расширения CSS

Расширения CSS — это атрибуты стиля, которые поддерживаются определенными Web-обозревателями и, можно сказать, расширяют возможности текущей версии CSS. Эти атрибуты стиля могут быть включены в черновую редакцию будущей версии CSS (на данный момент CSS 3), а могут вообще отсутствовать в любых документах, описывающих стандарт. В последнем случае расширения CSS служат для поддержки специфических возможностей того или иного Web-обозревателя.

Стандарт CSS требует, чтобы имена всех расширений CSS начинались с особого префикса, обозначающего Web-обозреватель, который их поддерживает. В этом приложении нам встретятся три таких префикса:

- `-moz-` — обозначает Mozilla Firefox;
- `-o-` — обозначает Opera;
- `-webkit-` — обозначает Web-обозреватели, основанные на программном ядре Webkit. Самые известные среди них — Google Chrome и Apple Safari.

Постараемся их запомнить.

Расширения CSS позволят нам задействовать возможности Web-обозревателя, которые не поддерживаются стандартом CSS или появятся только в будущей его версии. Однако, во-первых, расширения CSS зачастую поддерживаются только одним Web-обозревателем; другие Web-обозреватели могут и не "знать" об их существовании. Во-вторых, разработчики Web-обозревателя в любой момент могут лишить свое детище поддержки тех или иных расширений CSS, посчитав их ненужными.

> ***НА ЗАМЕТКУ***
>
> На Web-странице **https://developer.mozilla.org/en/CSS/CSS_transitions** описаны расширения CSS, предназначенные для создания анимационных эффектов — так называемых переходов CSS. Тем не менее, насколько удалось выяснить автору, последние версии Firefox их не поддерживают, хотя, по идее, должны. Вероятно, это как раз случай, когда разработчики Firefox посчитали эту возможность ненужной и удалили ее поддержку в очередной версии программы.
>
> Вполне возможно, что из описанных автором расширений CSS некоторые уже не поддерживаются.

В настоящем приложении мы рассмотрим некоторые расширения CSS, самые полезные для Web-дизайнеров, на взгляд автора.

Многоцветные рамки

Мы уже знаем, что с помощью особых атрибутов стиля CSS можно создавать одноцветные рамки у любых элементов Web-страницы.

Однако для рамок, толщина которых превышает один пиксел, мы можем задать сразу несколько цветов. В этом случае рамка будет представлена Web-обозревателем как набор вложенных друг в друга рамок толщиной в один пиксел. Первый из заданных нами цветов будет применен к самой внешней рамке, второй — к вложенной в нее рамке, третий — к рамке, вложенной в ту, которая вложена во внешнюю, и т. д.

Многоцветные рамки создают с помощью расширений CSS `-moz-border-left-colors`, `-moz-border-top-colors`, `-moz-border-right-colors` и `-moz-border-bottom-colors`. Они задают цвета, соответственно, для левой, верхней, правой и нижней стороны рамок.

```
-moz-border-left-colors|-moz-border-top-colors|-moz-border-right-colors|
-moz-border-border-colors: <набор цветов, разделенных пробелами>|none
```

Значение `none` убирает "многоцветье" у соответствующей стороны рамки. Это значение по умолчанию.

Данные расширения CSS поддерживаются только Firefox и не включены в стандарт CSS.

Пример:

```
#cheader { width: 1010px;
           padding: 0 20px;
           border-bottom: medium dotted;
           -moz-border-bottom-colors: #B1BEC6 #F8F8F8 #B1BEC6 }
```

Здесь мы задаем для контейнера `cheader` рамку, состоящую из одной нижней стороны, средней толщины, с тремя цветами.

Рамки со скругленными углами

Рамки с прямоугольными углами встречаются очень часто и уже успели намозолить глаза. Вот рамки со скругленными углами — другое дело!

Расширения CSS `-moz-border-radius-topleft` (Firefox), `border-top-left-radius` (Opera и стандарт CSS 3) и `-webkit-border-top-left-radius` (Web-обозреватели на программном ядре Webkit) задают радиус скругления верхнего левого угла рамки:

```
-moz-border-radius-topleft|border-top-left-radius|
-webkit-border-top-left-radius: <значение 1> [<значение 2>]
```

Если указано одно значение, оно задаст радиус скругления и по горизонтали, и по вертикали. Если указаны два значения, то первое задаст радиус скругления по горизонтали, а второе — по вертикали, создавая тем самым скругление эллиптической формы. Радиус скругления может быть задан в любой единице измерения, поддерживаемой CSS.

Для указания радиуса скругления остальных углов рамки предназначены расширения CSS, перечисленные далее.

- `-moz-border-radius-topright` (Firefox), `border-top-right-radius` (Opera и стандарт CSS 3) и `-webkit-border-top-right-radius` (Web-обозреватели на программном ядре Webkit) — радиус скругления верхнего правого угла рамки.
- `-moz-border-radius-bottomright` (Firefox), `border-bottom-right-radius` (Opera и стандарт CSS 3) и `-webkit-border-bottom-right-radius` (Web-обозреватели на программном ядре Webkit) — радиус скругления нижнего правого угла рамки.
- `-moz-border-radius-bottomleft` (Firefox), `border-bottom-left-radius` (Opera и стандарт CSS 3) и `-webkit-border-bottom-left-radius` (Web-обозреватели на программном ядре Webkit) — радиус скругления нижнего левого угла рамки.

Формат их использования такой же, как у расширений CSS, описанных в начале этого раздела (листинг П1).

Листинг П1

```
#cheader { width: 1010px;
           padding: 0 20px;
           border-bottom: medium dotted;
           -moz-border-radius-bottomleft: 2px;
           -moz-border-radius-bottomright: 2px;
           border-bottom-left-radius: 2px;
           border-bottom-right-radius: 2px;
           -webkit-border-bottom-left-radius: 2px;
           -webkit-border-bottom-right-radius: 2px }
```

Здесь мы задаем для контейнера `cheader` рамку, состоящую из одной нижней стороны, имеющую радиусы скругления нижнего левого и нижнего правого углов, равные двум пикселам, сразу для всех Web-обозревателей, поддерживающих эту возможность. Таким образом, и Firefox, и Opera, и Web-обозреватели на программном ядре Webkit выведут эти углы рамки скругленными.

Расширения CSS `-moz-border-radius` (Firefox), `border-radius` (Opera и стандарт CSS) и `-webkit-border-radius` (Web-обозреватели на программном ядре Webkit) позволяют задать радиусы скругления сразу для всех углов рамки:

```
-moz-border-radius|border-radius|-webkit-border-radius:
<позиция 1 значение 1>[/<позиция 1 значение 2>]
[<позиция 2 значение 1>[/<позиция 2 значение 2>]
[<позиция 3 значение 1>[/<позиция 3 значение 2>]
[<позиция 4 значение 1>[/<позиция 4 значение 2>]]]]
```

Как видим, каждая позиция может содержать как одно значение, так и пару значений, разделенных слэшем `/`. Если она содержит одно значение, это значение задаст радиус скругления и по горизонтали, и по вертикали. Если же в позиции указать два значения, разделив их слэшем, первое задаст радиус скругления по горизонтали, второе — по вертикали.

Кроме того, можно указать от одной до четырех позиций.

- Если указана одна позиция, она задаст радиус скругления всех углов рамки.
- Если указаны две позиции, первая задаст радиус скругления верхнего левого и нижнего правого углов рамки, а вторая — верхнего правого и нижнего левого углов.
- Если указаны три позиции, первая задаст радиус скругления верхнего левого угла рамки, вторая — верхнего правого и нижнего левого, а третья — нижнего правого угла.
- Если указаны четыре позиции, первая задаст радиус скругления верхнего левого угла рамки, вторая — верхнего правого, третья — нижнего правого, а четвертая — нижнего левого угла.

Пример иллюстрирует листинг П2.

Листинг П2

```
#navbar LI { padding: 5px 10px;
             margin: 10px 0px;
             border: thin solid #B1BEC6;
             -moz-border-radius: 3px/1px 3px/1px 0px 0px;
             border-radius: 3px/1px 3px/1px 0px 0px;
             -webkit-border-radius: 3px/1px 3px/1px 0px 0px;
             cursor: pointer }
```

Здесь мы задаем для пунктов "внешних" списков, формирующих полосу навигации, рамки со скругленными верхними углами. Радиус скругления их по горизонтали будет 3 пиксела, а по вертикали — 1 пиксел.

Выделение со скругленными углами

Сказавший "а" да скажет "б"! Позволивший создавать рамки со скругленными углами да позволит скруглять углы у выделения!

Расширение CSS `-moz-outline-radius-topleft` задает радиус скругления верхнего левого угла выделения:

```
-moz-outline-radius-topleft: <значение>
```

Радиус скругления может быть задан в любой единице измерения, поддерживаемой CSS.

Для указания радиуса скругления остальных углов выделения применяются расширения CSS, перечисленные далее.

- `-moz-outline-radius-topright` — радиус скругления верхнего правого угла выделения.
- `-moz-outline-radius-bottomright` — радиус скругления нижнего правого угла выделения.

- `-moz-outline-radius-bottomleft` — радиус скругления нижнего левого угла выделения.

Формат их использования такой же, как у расширений CSS, описанных в начале этого раздела.

Листинг П3 иллюстрирует пример.

Листинг П3

```
DFN { outline: thin dotted #B1BEC6;
      -moz-outline-radius-topleft: 3px;
      -moz-outline-radius-topright: 3px;
      -moz-outline-radius-bottomright: 3px;
      -moz-outline-radius-bottomleft: 3px }
```

Здесь мы задаем для всех фрагментов текста, помеченных тегом `<DFN>`, выделение, все углы которого имеют радиус скругления 3 пиксела.

Расширение CSS `-moz-outline-radius` позволяет задать радиусы скругления сразу для всех углов выделения:

```
-moz-outline-radius: <значение 1> [<значение 2> [<значение 3>
[<значение 4>]]]|inherit
```

Здесь можно указать от одного до четырех значений.

- Если указано одно значение, оно задаст радиус скругления всех углов выделения.
- Если указаны два значения, первое задаст радиус скругления верхнего левого и нижнего правого углов выделения, а второе — верхнего правого и нижнего левого углов.
- Если указаны три значения, первое задаст радиус скругления верхнего левого угла выделения, второе — верхнего правого и нижнего левого, а третье — нижнего правого угла.
- Если указаны четыре значения, первое задаст радиус скругления верхнего левого угла выделения, второе — верхнего правого, третье — нижнего правого, а четвертое — нижнего левого угла.

Эти расширения CSS поддерживаются только Firefox и не включены в стандарт CSS 3.

Пример:

```
DFN { outline: thin dotted #B1BEC6;
      -moz-outline-radius: 3px }
```

Многоколоночная верстка

Многоколоночная верстка — это то, чего давно не хватало в Web-дизайне. Отдельные энтузиасты уже давно реализовывали ее с помощью таблиц или контейнеров.

Но теперь у них есть "законные" средства разбить текст на произвольное число колонок, воспользовавшись особыми расширениями CSS.

Расширения CSS `-moz-column-count` (Firefox) и `-webkit-column-count` (Web-обозреватели на программном ядре Webkit) задают желаемое число колонок для многоколоночной верстки:

```
-moz-column-count|-webkit-column-count: <число колонок>|auto
```

Реальное число колонок, которое выведет Web-обозреватель, может быть другим; например, на Web-странице может не оказаться места для указанного числа колонок или для размещения текста может потребоваться меньше колонок, чем было указано.

Значение `auto`, судя по всему, отменяет многоколоночную верстку.

Пример:

```
#cmain { -moz-column-count: 2;
         -webkit-column-count: 2 }
```

Здесь мы задаем для контейнера `cmain` две колонки.

Расширения CSS `-moz-column-width` (Firefox) и `-webkit-column-width` (Web-обозреватели на программном ядре Webkit) задают желаемую ширину колонок:

```
-moz-column-width|-webkit-column-width: <ширина колонок>|auto
```

Реальная ширина колонок, установленная Web-обозревателем, может быть больше или меньше и будет зависеть от ширины элемента Web-страницы, содержимое которого верстается в несколько колонок.

Значение `auto` возвращает управление шириной колонок Web-обозревателю.

Листинг П4 иллюстрирует пример.

Листинг П4

```
#cmain { -moz-column-count: 2;
         -webkit-column-count: 2;
         -moz-column-width: 300px;
         -webkit-column-width: 300px }
```

Задаем для контейнера `cmain` ширину колонок в 300 пикселов.

Расширения CSS `-moz-column-gap` (Firefox) и `-webkit-column-gap` (Web-обозреватели на программном ядре Webkit) задают ширину пространства между колонками:

```
-moz-column-gap|-webkit-column-gap:
<ширина пространства между колонками>|normal
```

Значение `normal` задает ширину пространства между колонками по умолчанию. Ее величина зависит от Web-обозревателя.

Листинг П5 иллюстрирует пример.

Листинг П5

```
#cmain { -moz-column-count: 2;
         -webkit-column-count: 2;
         -moz-column-width: 300px;
         -webkit-column-width: 300px;
         -moz-column-gap: 5mm;
         -webkit-column-gap: 5mm }
```

В листинге П5 задаем для контейнера `cmain` ширину пространства между колонками 5 мм.

Расширения CSS `-moz-column-rule-width` (Firefox) и `-webkit-column-rule-width` (Web-обозреватели на программном ядре Webkit) задают толщину линий, которыми колонки будут отделяться друг от друга:

```
-moz-column-rule-width|-webkit-column-rule-width: thin|medium|thick|
<толщина линий между колонками>
```

Здесь доступны те же значения, что и для атрибутов стиля, задающих толщину линий рамки и выделения (см. *главу 11*).

Листинг П6 иллюстрирует пример.

Листинг П6

```
#cmain { -moz-column-count: 2;
         -webkit-column-count: 2;
         -moz-column-width: 300px;
         -webkit-column-width: 300px;
         -moz-column-gap: 5mm;
         -webkit-column-gap: 5mm;
         -moz-column-rule-width: thin;
         -webkit-column-rule-width: thin }
```

Теперь между колонками в контейнере `cmain` будут проведены тонкие линии.

Расширения CSS `-moz-column-rule-style` (Firefox) и `-webkit-column-rule-style` (Web-обозреватели на программном ядре Webkit) задают стиль линий, которыми колонки будут отделяться друг от друга:

```
-moz-column-rule-style|-webkit-column-rule-style:
none|hidden|dotted|dashed|solid|double|groove|ridge|inset|outset
```

Здесь доступны те же значения, что и для атрибутов стиля, задающих стиль линий рамки и выделения (см. *главу 11*).

Листинг П7 иллюстрирует пример.

Листинг П7

```
#cmain { -moz-column-count: 2;
         -webkit-column-count: 2;
```

```
        -moz-column-width: 300px;
        -webkit-column-width: 300px;
        -moz-column-gap: 5mm;
        -webkit-column-gap: 5mm;
        -moz-column-rule-width: thin;
        -webkit-column-rule-width: thin;
        -moz-column-rule-style: dotted;
        -webkit-column-rule-style: dotted }
```

Теперь между колонками в контейнере `cmain` будут проведены тонкие точечные линии.

Расширения CSS `-moz-column-rule-color` (Firefox) и `-webkit-column-rule-color` (Web-обозреватели на программном ядре Webkit) задают цвет линий, которыми колонки будут отделяться друг от друга:

```
-moz-column-rule-color|-webkit-column-rule-color:
<цвет линий между колонками>
```

Листинг П8 иллюстрирует пример.

Листинг П8

```
#cmain { -moz-column-count: 2;
         -webkit-column-count: 2;
         -moz-column-width: 300px;
         -webkit-column-width: 300px;
         -moz-column-gap: 5mm;
         -webkit-column-gap: 5mm;
         -moz-column-rule-width: thin;
         -webkit-column-rule-width: thin;
         -moz-column-rule-style: dotted;
         -webkit-column-rule-style: dotted;
         -moz-column-rule-color: #B1BEC6;
         -webkit-column-rule-color: #B1BEC6 }
```

Теперь между колонками в контейнере `cmain` будут проведены тонкие точечные линии светло-синего цвета.

Расширения CSS `-moz-column-rule` (Firefox) и `-webkit-column-rule` (Web-обозреватели на программном ядре Webkit) задают сразу все параметры линий, которыми колонки будут отделяться друг от друга:

```
-moz-column-rule|-webkit-column-rule: <толщина> <стиль> <цвет>
```

Листинг П9 иллюстрирует пример.

Листинг П9

```
#cmain { -moz-column-count: 2;
         -webkit-column-count: 2;
         -moz-column-width: 300px;
```

```
        -webkit-column-width: 300px;
        -moz-column-gap: 5mm;
        -webkit-column-gap: 5mm;
        -moz-column-rule: thin dotted #B1BEC6;
        -webkit-column-rule: thin dotted #B1BEC6 }
```

К сожалению, Opera на данный момент не поддерживает многоколоночную верстку. А жаль...

Преобразования CSS

В *главе 22*, ведя разговор о канве и программном рисовании на ней, мы узнали о преобразованиях системы координат. С помощью особых расширений CSS мы можем проделать аналогичные действия над любым элементом Web-страницы: сместить его, повернуть, растянуть или сжать.

Для указания, какие именно преобразования следует выполнить над элементом Web-страницы, служат расширения CSS `-moz-transform` (Firefox), `-o-transform` (Opera) и `-webkit-transform` (Web-обозреватели на программном ядре Webkit):

```
-moz-transform|-o-transform|-webkit-transform: <преобразование>
```

Доступных преобразований не так уж и много. Сейчас мы их рассмотрим.

Преобразование `rotate` позволяет повернуть элемент Web-страницы на указанный угол по часовой стрелке:

```
rotate(<угол>)
```

Значение *угла* может быть задано в трех единицах измерения: градусах, радианах и градах. Если требуется указать угол в радианах, после самого числового значения угла ставят обозначение `deg`, в радианах — `rad`, в градах — `grad`. При этом между числом и обозначением единицы измерения угла не должно быть пробелов.

Здесь мы поворачиваем контейнер `cheader` на 3,14 радиан (примерно 180º):

```
#cheader { -moz-transform: rotate(3.14rad);
           -o-transform: rotate(3.14rad);
           -webkit-transform: rotate(3.14rad) }
```

А здесь будет выполнен поворот контейнера `cheader` на 30º:

```
#cheader { -moz-transform: rotate(30deg);
           -o-transform: rotate(30deg);
           -webkit-transform: rotate(30deg) }
```

Преобразование `scale` позволяет изменить масштаб элемента Web-страницы по горизонтали и вертикали, растянув его или сжав:

```
scale(<масштаб 1>[, <масштаб 2>])
```

Если указано одно значение, оно задает масштаб и по горизонтали, и по вертикали. Если указаны два значения, первое задает масштаб по горизонтали, второе — по вертикали. Значения больше 1 задают увеличение масштаба (растяжение), а значе-

ния меньше 1 — уменьшение (сжатие); значение 1 оставляет масштаб без изменений.

Здесь увеличиваем масштаб контейнера `cheader` вдвое по горизонтали и вертикали, тем самым растягивая его:

```
#cheader { -moz-transform: scale(2);
           -o-transform: scale(2);
           -webkit-transform: scale(2) }
```

А здесь увеличиваем масштаб контейнера `cheader` вдвое по горизонтали и уменьшаем его вдвое по вертикали:

```
#cheader { -moz-transform: scale(2, 0.5);
           -o-transform: scale(2, 0.5);
           -webkit-transform: scale(2, 0.5) }
```

Преобразования `scaleX` и `scaleY` позволяют изменить масштаб элемента Web-страницы, соответственно, только по горизонтали и только по вертикали:

```
scaleX|scaleY(<масштаб>)
#cheader { -moz-transform: scaleX(2);
           -o-transform: scaleX(2);
           -webkit-transform: scaleX(2) }
```

Преобразование `skew` позволяет сдвинуть элемент Web-страницы по горизонтальной и вертикальной оси на заданный угол, тем самым "скособочив" его:

```
skew(<угол 1>[, <угол 2>])
```

Если указано одно значение, оно задает угол сдвига и по горизонтальной, и по вертикальной оси. Если указаны два значения, первое задает угол сдвига по горизонтальной, второе — по вертикальной оси. Углы могут быть заданы в любых единицах измерения, описанных в начале этого раздела.

Сдвигаем контейнер `cheader` по горизонтальной оси на 10º. По вертикальной оси он сдвинут не будет, поскольку мы задали угол сдвига 0º:

```
#cheader { -moz-transform: skew(10deg, 0deg);
           -o-transform: skew(10deg, 0deg);
           -webkit-transform: skew(10deg, 0deg) }
```

Преобразования `skewX` и `skewY` позволяют сдвинуть элемент Web-страницы, соответственно, только по горизонтальной и только по вертикальной оси.

Пример:

```
skewX|skewY(<угол>)
#cheader { -moz-transform: skewX(10deg);
           -o-transform: skewX(10deg);
           -webkit-transform: skewX(10deg) }
```

Преобразование `translate` позволяет сместить элемент Web-страницы по горизонтальной и вертикальной оси на заданное расстояние:

```
translate(<расстояние 1>[, <расстояние 2>])
```

Если указано одно значение, оно задает расстояние для смещения и по горизонтальной, и по вертикальной оси. Если указаны два значения, первое задает расстояние для смещения по горизонтальной, второе — по вертикальной оси. Расстояния могут быть заданы в любых единицах измерения, поддерживаемых CSS.

Смещаем контейнер `cheader` на 5 мм по горизонтальной оси и на 2 мм по вертикальной:

```
#cheader { -moz-transform: translate(5mm, 2mm);
           -o-transform: translate(5mm, 2mm);
           -webkit-transform: translate(5mm, 2mm) }
```

Преобразования `translateX` и `translateY` позволяют сместить элемент Web-страницы, соответственно, только по горизонтальной и только по вертикальной оси.

Смещаем контейнер `cheader` на 2 мм по вертикальной оси:

```
translateX|translateY(<расстояние>)
#cheader { -moz-transform: translateY(2mm);
           -o-transform: translateY(2mm);
           -webkit-transform: translateY(2mm) }
```

Мы можем подвергнуть элемент Web-страницы сразу нескольким преобразованиям. Для этого следует записать эти преобразования одно за другим, разделив их пробелами. Преобразования будут применяться к элементу в том порядке, в котором они записаны.

Пример:

```
#cheader { -moz-transform: skewX(10deg) translateY(2mm);
           -o-transform: skewX(10deg) translateY(2mm);
           -webkit-transform: skewX(10deg) translateY(2mm) }
```

Здесь мы подвергаем контейнер `cheader` сначала сдвигу по горизонтальной оси на 10º, а потом — смещению по вертикальной оси на 2 мм.

По умолчанию все преобразования выполняются относительно центра элемента Web-страницы. Так, если мы повернем элемент на какой-то угол, он повернется относительно своего центра. Но мы можем указать другую точку, относительно которой будут выполняться все последующие преобразования. Для этого служат расширения CSS `-moz-transform-origin` (Firefox), `-o-transform-origin` (Opera) и `-webkit-transform-origin` (Web-обозреватели на программном ядре Webkit):

```
-moz-transform-origin|-o-transform-origin|-webkit-transform-origin:
<координата>|left|center|right [<координата>|top|center|bottom]
```

Если задано одно значение, оно укажет координату точки, относительно которой будут выполняться преобразования, по горизонтальной оси; координатой этой точки по вертикальной оси станет центр элемента Web-страницы. Если указаны два значения, первое задаст координату точки по горизонтальной оси, второе — по вертикальной.

Само значение координаты может быть задано в любой абсолютной или относительной единице измерения, поддерживаемой CSS. Также можно указать значения

`left` (левый край элемента Web-страницы), `center` (центр), `right` (правый край), `top` (верхний край) и `bottom` (нижний край).

Листинг П10 иллюстрирует пример.

Листинг П10

```
#cheader { -moz-transform: rotate(30deg);
           -o-transform: rotate(30deg);
           -webkit-transform: rotate(30deg);
           -moz-transform-origin: left;
           -o-transform-origin: left;
           -webkit-transform-origin: left }
```

Здесь мы поворачиваем контейнер `cheader` на 30º относительно точки, которая находится в середине его левого края. Мы задали в качестве координаты точки, относительно которой будут выполняться преобразования (в том числе и поворот), значение `left` (левый край контейнера) — это ее координата по горизонтали; в качестве вертикальной координаты, поскольку мы ее не указали, будет принята середина контейнера.

Еще один пример иллюстрирует листинг П11.

Листинг П11

```
#cheader { -moz-transform: rotate(30deg);
           -o-transform: rotate(30deg);
           -webkit-transform: rotate(30deg);
           -moz-transform-origin: right bottom;
           -o-transform-origin: right bottom;
           -webkit-transform-origin: right bottom }
```

Теперь контейнер `cheader` будет повернут на 30º относительно своего нижнего правого угла.

Предметный указатель

Z

A

Б

В

Г

Д

Е

З

И

К

Л

М

Н

О

П

Р

С

T

У

Ф

Х

Ц

Ч

Ш

Э

Я