Владимир Дронов

bhv

JavaScript и AJAX в Web-дизайне

2-е издание

- HTML, CSS, JavaScript и AJAX
- Web-сценарии и события
- Управление Web-обозревателем и содержимым Web-страницы
- Графика и анимация
- Web-формы и базы данных
- Фильтры и преобразования

Наиболее полное руководство

В ПОДЛИННИКЕ®

Владимир Дронов

JavaScript и AJAX в Web-дизайне

2-е издание

Санкт-Петербург

«БХВ-Петербург»

2008

УДК 681.3.06
ББК 32.973.26-018.2
Д75

Дронов В. А.

Д75 JavaScript и AJAX в Web-дизайне: 2-е изд., перераб. и доп. — СПб.: БХВ-Петербург, 2008. — 736 с.: ил. — (В подлиннике)

ISBN 978-5-9775-0251-1

В книге описывается все, что должен знать Web-дизайнер: принципы создания Web-страниц, язык JavaScript, основы написания Web-сценариев, работа с содержимым Web-страницы, обработка данных, введенных в Web-форму, особенности различных Web-обозревателей, использование баз данных, фильтров и преобразований, графика, анимация и пр. Изложение сопровождается большим количеством подробно разобранных примеров и полезных советов. Особое внимание уделено вопросам совместимости Web-сценариев с различными Web-обозревателями.

Второе издание книги, ранее выходившей под названием "JavaScript в Web-дизайне", полностью переработано и дополнено с учетом современных технологий, дан вводный курс AJAX.

Для Web-дизайнеров

УДК 681.3.06
ББК 32.973.26-018.2

Группа подготовки издания:

Главный редактор	*Екатерина Кондукова*
Зам. главного редактора	*Евгений Рыбаков*
Зав. редакцией	*Григорий Добин*
Редактор	*Екатерина Капалыгина*
Компьютерная верстка	*Натальи Караваевой*
Корректор	*Виктория Пиотровская*
Дизайн серии	*Инны Тачиной*
Оформление обложки	*Елены Беляевой*
Зав. производством	*Николай Тверских*

Лицензия ИД № 02429 от 24.07.00. Подписано в печать 27.06.08.
Формат $70\times100^{1}/_{16}$. Печать офсетная. Усл. печ. л. 59,36.
Тираж 2500 экз. Заказ №
"БХВ-Петербург", 194354, Санкт-Петербург, ул. Есенина, 5Б.

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

ISBN 978-5-9775-0251-1

Оглавление

Введение

В далеком уже 2001 году автор написал книгу о языке JavaScript и его использовании в Web-дизайне. Эта книга стала первой и одной из самых активно продаваемых в карьере автора.

Но время идет, интернет-стандарты меняются, интернет-технологии развиваются, выходят новые версии Web-обозревателей, имеющие новые возможности, книги устаревают. Давно устарела и первая книга "JavaScript в Web-дизайне". Вот поэтому автор снова, образно говоря, взялся за перо.

Новая книга — "JavaScript и AJAX в Web-дизайне" — это обновленное издание, включающее описание некоторых новых возможностей и технологий и очищенное от уже устаревшего материала (описывающего, в основном, давно почивший в бозе Netscape Navigator). Вооружившись самой "свежей" документацией и самыми последними версиями Web-обозревателей Internet Explorer, Opera и Firefox, автор как следует в них покопался и теперь представляет на суд читателей результат этих "раскопок".

Грамотные интернетчики могут пропустить следующую главу. Остальным лучше ее прочитать, чтобы узнать, о чем же эта книга.

О чем эта книга?

Что мы чаще всего делаем в Интернете? Правильно, смотрим Web-страницы. А что такое Web-страница? Это особым образом подготовленный документ, содержащий текст, изображения, таблицы и пр. Мы вводим интернет-адрес страницы в специальное поле ввода Web-обозревателя и наблюдаем все это в его окне.

Web-страницы создаются с помощью особых языков — HTML и CSS. Первый язык описывает собственно содержимое страницы (какой текст, какие изображения и какие таблицы на ней будут присутствовать), второй — ее оформление (размер шрифта, цвет текста и фона, толщину рамки и пр.). Если короче, то языки HTML и CSS задают представление страницы.

Но ни HTML, ни CSS не позволяют задать поведение страницы. Например, мы не сможем только с их помощью сделать так, чтобы при наведении курсора мыши на изображение оно менялось или чтобы какой-то из элементов страницы скрывался и вновь показывался в ответ на нажатие кнопки. И уж тем более мы не сможем сделать какой-то из элементов страницы анимированным. Страница статична — загрузившись, она остается неизменной, что бы мы с ней не делали.

Почему? А потому, что языки HTML и CSS не предусматривают никаких средств, чтобы описать поведение элемента страницы. Они, как было сказано ранее, описывают только ее представление.

Но ведь мы часто встречаем страницы с меняющимися в ответ на движения мыши изображениями, скрывающимися и показывающимися элементами и даже анимацией! Как все это делается?

С помощью Web-сценариев — особых программ, вставляющихся прямо в страницу и выполняющихся Web-обозревателем. Они как раз и реализуют все эти чудеса.

Web-сценарии пишутся, как правило, на языке JavaScript. (Иногда для этого используются и другие языки, но крайне редко.) Его мы и будем изучать на протяжении всей этой книги.

Что мы можем сделать с помощью Web-сценариев? О, довольно много…

- Управлять Web-обозревателем (открывать и закрывать его окна, менять их размеры и местоположение на экране и др.).
- Управлять содержимым Web-страницы (добавлять новые элементы и удалять ненужные, изменять содержимое элементов и их параметры).
- Работать с произвольными данными (сохранять и пересылать другим страницам).
- Работать с данными, введенными посетителем в Web-форму.
- Реализовывать специальные эффекты (анимация, создание "красивостей" наподобие тени, эффекта размытия и пр.).

И все это мы изучим благодаря данной книге.

Какие программы будут использоваться 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.
- Internet Explorer 7 — Web-обозреватель, самый популярный на данный момент.
- Opera 9.26 — Web-обозреватель.
- Firefox 2.0.0.11 — Web-обозреватель.

Все примеры тестировались на перечисленных ранее Web-обозревателях. В некоторых случаях примеры также тестировались на более старых Web-обозревателях, в частности, на Netscape Navigator 7 preview 1.

Другие программы автором в работе практически не использовались.

Типографские соглашения

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

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

В угловые скобки (<>) заключаются названия значений атрибутов, параметров или фрагментов кода, которые, в свою очередь, набраны курсивом. В код реального сценария, разумеется, должны быть подставлены реальное значение, реальный параметр или реальный код. Например:

```
<MAP NAME="<имя карты>">
  <набор описаний горячих областей>
</MAP>
```

Здесь вместо подстроки *<имя карты>* должно быть подставлено реальное имя карты, а вместо подстроки *<набор описаний горячих областей>* — реальный код, описывающий горячие области.

В квадратные скобки ([]) заключаются необязательные фрагменты кода. Например:

```
USEMAP=[<интернет-адрес страницы>]#<имя карты>
```

Здесь *<интернет-адрес страницы>* может присутствовать, а может и отсутствовать.

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

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

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

Слишком длинные, не помещающиеся на одной строке выражения JavaScript автор разрывает на несколько строк и в местах разрывов ставит знаки ↳. Например:

```
var langDataObj = xhr2Obj.responseXML.
↳getElementsByTagName("langdata")[0];
```

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

> ***ЕЩЕ РАЗ ВНИМАНИЕ!***
>
> Все приведенные ранее типографские соглашения имеют смысл только в форматах написания тегов HTML, атрибутов стилей CSS и выражений JavaScript. В коде примеров используется только знак ↳.

Благодарности

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

- Губине Наталье Анатольевне, начальнику отдела АСУ Волжского гуманитарного института (г. Волжский Волгоградской обл.), где работает автор, — за понимание и поддержку.
- Всем работникам отдела АСУ — 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.
- Всем своим читателям и почитателям — за прекрасные отзывы о моих книгах.
- Всем, кого я забыл здесь перечислить, — за все хорошее.

Часть I

Введение в Web-дизайн и Web-программирование

Глава 1

Что такое Интернет и как он работает

Первая часть этой книги будет посвящена рассмотрению всех интернет-технологий, используемых для создания Web-страниц. Ведь согласитесь — глупо начинать с Web-программирования, не научившись делать сами Web-страницы.

Также глупо начинать с создания Web-страниц, не выяснив, что такое Интернет и как он работает. Так что первая глава книги будет посвящена именно Интернету и основным принципам его работы.

Основные принципы работы Интернета

С принципов работы Интернета мы и начнем. И первым же делом выясним, что же такое Интернет.

Что такое Интернет

В самом деле, что такое *Интернет*? Всемирная компьютерная сеть. (Ее, кстати, так часто и называют: Всемирная сеть, или даже просто Сеть с большой буквы.) Протянутая по всему земному шару паутина медных проводов, волоконно-оптических кабелей и радиоканалов, связывающих друг с другом многочисленные компьютеры, — вот что такое Интернет. Разумеется, все здесь подчиняется общим стандартам (о которых мы поговорим далее) — иначе эта суперсеть просто не будет работать.

Если же быть совсем точным, то Интернет — это не единая сеть, а совокупность более мелких сетей, связанных друг с другом общими каналами и стандартами. Таких сетей превеликое множество: огромные территориальные сети, раскинувшиеся на целые области, штаты и государства, ведомственные сети, объединяющие родственные организации, локальные компьютерные

сети отдельных организаций и так называемые кампусные сети — сети, объединяющие компьютеры одного или нескольких близлежащих районов города. Благодаря проложенным между ними каналам связи они и составляют единое целое, имя которому Интернет.

Даже частные пользователи, подключающиеся к Интернету по модему, выделенной линии, радиоканалу или поддерживающему такую возможность сотовому телефону, тоже по сути дела являются частью Сети. Так что когда мы включаем наш модем и дозваниваемся до нашего *интернет-провайдера* (организации, предоставляющей доступ в Интернет), то приобщаемся к единому целому. А что, разве это не повод для законной гордости?

Сеть Интернет имеет одну замечательную особенность — она очень устойчива к сбоям. Так, если где-то порвется провод, мы этого не заметим. А все потому, что данные, которые мы запрашиваем, пойдут в этом случае по другому проводу. Специалисты говорят, что Интернет децентрализован — он не имеет единого центра, из которого ведется управление пересылкой данных, поэтому в случае аварии автоматически переконфигурируется и продолжает нормально работать.

Еще одна замечательная особенность Интернета — его глобальность, всемирность. Не вставая из-за компьютера, мы можем совершить путешествие по всему миру, побывать в США, Австралии, Германии, Зимбабве, на Огненной Земле и даже в Антарктиде! Для этого нужно всего лишь набрать нужный нам интернет-адрес.

Интернет имеет достаточно долгую и бурную историю. Он появился еще в конце 60-х годов XX века, когда Министерство обороны США финансировало проект создания компьютерной сети, устойчивой к сбоям. Разумеется, создавалась эта сеть для нужд обороны, да и название имела другое — *ARPANET*. Позднее, в начале 80-х, эта сеть отошла к ученым, а военные приступили к созданию другой сети, которой пользуются до сих пор. И в то же самое время ARPANET был переименован в Internet, или, если по-русски, Интернет.

Первоначально, еще во времена ARPANET, эта сеть использовалась для пересылки электронной почты и обмена файлами. Web-странички, ради которых мы, в основном, и путешествуем по Сети, появились только в конце 80-х. Именно тогда Интернет и "пошел в народ", перестав быть сетью ученых и превратившись в сеть для всех.

В Россию, точнее, в СССР, Интернет официально пришел в 1991 году, но популярность среди широких масс компьютерщиков приобрел только в середине 90-х. В настоящее же время в России, наверное, и не найти человека, не слышавшего об Интернете. Вы такого встречали? Автор — еще нет.

НА ЗАМЕТКУ

Говорят, в первой польской энциклопедии, изданной, кажется, в XVII столетии, термин "лошадь" описывался так: "что такое лошадь, знают все". То же самое можно сейчас сказать об Интернете. (Вот только можно ли сейчас сказать то же самое о лошади?..)

Сервисы Интернета

Раз уж мы заговорили об услугах, предоставляемых Интернетом, или, как говорят профессионалы, *сервисах* Интернета, то давайте узнаем о них побольше. В конце концов, нам ими пользоваться…

Самый старый и самый популярный до сих пор сервис Интернета — это электронная почта (e-mail). Ежедневно в мире отправляются и принимаются сотни миллионов электронных писем, и это количество в будущем будет только увеличиваться. В самом деле, электронная почта доступна, удобна, быстра и бесплатна, в отличие от почты "бумажной", которую пользователи Интернета уже успели презрительно прозвать "улиточной" (по-английски — snail mail). Конечно, эти доступность, удобство, быстрота и бесплатность имеют и некоторые недостатки, вроде "спама" — несанкционированных рекламных рассылок, но эти недостатки вполне можно стерпеть.

Еще один сервис Интернета, почти такой же старый, как почта, — это пересылка файлов. Пользователи Интернета называют его *FTP* (File Transfer Protocol, протокол передачи файлов; почему так — мы узнаем чуть позже). Сейчас FTP уже не имеет той популярности, как на заре существования Интернета, но все еще довольно часто используется. Так, все крупные корпорации — Intel, Microsoft и др. — помимо Web-сайта, имеют и сервер FTP.

Третий сервис Интернета — это Всемирная паутина, или *WWW* (World Wide Web, повсеместно протянутая паутина), или просто *Web*, те самые Web-страницы и Web-сайты, которые мы просматриваем в Web-обозревателе. Появившийся значительно позже электронной почты и FTP, WWW стала самым популярным сервисом и, собственно, превратила Интернет из сети ученых в сеть для всех.

Об остальных сервисах Интернета (а их немало) мы только упомянем. Это новости UseNet, потоковое вещание, интернет-пейджеры, чаты, нашумевшие в последнее несколько лет файлообменные сети и некоторые другие, менее известные сервисы.

Клиенты и серверы

Но каким образом мы пользуемся всем тем богатством, что дает нам Всемирная сеть? С помощью особых программ! Это Web-обозреватель, клиент электронной почты, программа просмотра интернет-телевидения и прослушива-

ния интернет-радио, интернет-пейджер и "чатилка". Все они очень хорошо нам знакомы.

Но программ, используемых для предоставления нам сервисов Интернета, гораздо больше. И очень многие из них нам, если так можно сказать, "не видны", то есть мы не общаемся с ними напрямую. Вообще, существуют два совершенно разных вида интернет-программ. И сейчас мы о них поговорим.

Программы, относящиеся к первому виду, — это Web-обозреватели, клиенты электронной почты, чатов, интернет-пейджеры, в общем, все те, с которыми мы имеем дело непосредственно. Мы получаем с их помощью различную информацию из Сети и работаем с ней. Такие программы называются программами-*клиентами*, а компьютеры, на которых они работают, — наши с вами компьютеры! — *клиентскими*.

Да, но как программы-клиенты получают из Сети нужную нам информацию (Web-страницы, файлы, письма и пр.)? Очень просто — для этого они обращаются к другим программам, относящимся ко второму виду. Это программы-*серверы*, работающие на *серверных* компьютерах, где также хранится и запрашиваемая клиентами информация. Существуют Web-, FTP-серверы, серверы электронной почты, чата, интернет-пейджеров, потокового вещания и пр.

> ***НА ЗАМЕТКУ***
>
> Очень часто понятие "сервер" распространяется и на серверный компьютер, и на саму программу-сервер. Это, вообще-то, неправильно, так как на одном серверном компьютере может быть установлено несколько программ-серверов, но вошло в практику.

Процесс получения информации клиентами от сервера включает шесть шагов.

1. Пользователь запрашивает с помощью программы-клиента некую информацию, введя в программу интернет-адрес сервера. (Об интернет-адресах мы поговорим потом, а пока будем знать, что это особый адрес, однозначно идентифицирующий нужный нам компьютер в Интернете и работающую на нем серверную программу.)
2. Клиент устанавливает *соединение* (своего рода линию связи) с сервером и посылает тому особый информационный блок, называемый *клиентским запросом*. Этот запрос должен быть определенным образом оформлен, чтобы сервер его понял.
3. Сервер принимает запрос и расшифровывает его.
4. Сервер извлекает запрошенный файл или фрагмент данных, записанных в файле, и посылает его клиенту в составе другого информационного блока — *серверного ответа*. Разумеется, этот ответ также должен быть составлен определенным образом. Если же запрашиваемых клиентом дан-

ных не найдено или сервер почему-то не смог понять клиентский запрос, он возвращает *сообщение об ошибке* — информационный блок, содержащий *код* (числовой номер) и, возможно, текстовое описание возникшей ошибки.

5. Клиент получает ответ от сервера, расшифровывает его и выдает полученную информацию пользователю. Если получено сообщение об ошибке, клиент сообщает об этом пользователю либо предпринимает какие-то действия самостоятельно.
6. Клиент разрывает соединение с сервером.

Процесс отправки клиентом данных серверу также включает шесть шагов.

1. Пользователь вводит в программу-клиент информацию и интернет-адрес сервера, которому она должна быть отправлена.
2. Клиент устанавливает соединение с сервером и посылает тому отправляемую информацию в составе клиентского запроса. При этом отправляемая информация, как правило, особым образом кодируется.
3. Сервер принимает запрос, расшифровывает его и извлекает отправленную информацию.
4. Сервер записывает отправленную клиентом информацию в файл или обрабатывает каким-то образом. В случае успешной записи или обработки он отправляет клиенту так называемое *подтверждение* — информационный блок, сообщающий о том, что все прошло нормально. Если у сервера возникли проблемы с приемом информации, он отправляет сообщение об ошибке.
5. Клиент получает ответ от сервера, расшифровывает его и уведомляет пользователя об успешной или неуспешной отправке данных либо предпринимает какие-то действия самостоятельно.
6. Клиент разрывает соединение с сервером.

Весь процесс "общения" клиента и сервера, начиная с отправки клиентом запроса и заканчивая принятием им ответа от сервера, называется *сеансом*. А соединение между клиентом и сервером, устанавливаемое на время этого сеанса и разрываемое после его окончания, называется *сеансовым*, или *временным*.

Любое соединение между клиентом и сервером устанавливается только клиентом. Сервер установить соединение с клиентом не может. Можно сказать, что серверу здесь отведена подчиненная роль.

Мы только что познакомились с особой *архитектурой* (принципом построения компьютерных систем), называемой *двухзвенной*, или архитектурой

"клиент-сервер". Эта архитектура использует два вида программ — клиенты и серверы, — выполняющие разные роли. Она применяется для реализации почти всех современных интернет-сервисов и пока что себя оправдывает.

> ***НА ЗАМЕТКУ***
>
> Некоторые интернет-сервисы, в частности, файлообменные сети (Napster, Gnutella, Kazaa и др.), используют другую архитектуру — *однозвенную*. Здесь все компьютеры, подключенные к Интернету и реализующие этот сервис, фактически равны между собой; любой из них может выступать в роли как клиентского (запрашивать информацию у других компьютеров), так и серверного (предоставлять хранящуюся на нем информацию другим компьютерам). Само собой, здесь используется особое программное обеспечение, которое может работать и как клиент, и как сервер.

В отличие от клиента, "имеющего дело" с одним-единственным пользователем, сервер работает сразу с множеством пользователей, причем одновременно. Сведения о соединениях, данные, пересылаемые клиентам и принимаемые от клиентов, — все это активно отнимает системные ресурсы компьютера, и чем больше соединений и данных проходят через сервер, тем больше требуется ресурсов. Поэтому на серверных компьютерах, как правило, не экономят.

Серверные компьютеры — настоящие монстры, содержащие несколько процессоров, дисковые массивы впечатляющей емкости, быстрые каналы связи с Интернетом и специальное программное обеспечение, у которого достаточно "сил", чтобы управлять всей этой мощью. Все в них нацелено на то, чтобы обслужить как можно больше клиентов, обработать как можно больше запросов, чтобы пользователи получили запрошенную информацию за приемлемое время. Но часто, если клиентов и запросов оказывается слишком много, ресурсов серверного компьютера не хватает, и начинаются проблемы. Они могут проявляться в том, что сервер просто отказывается обслужить "лишних" клиентов, предлагая им подождать немного, когда нагрузка немного снизится, а то и в том, что могучий серверный компьютер просто-напросто "зависает". Такое тоже случается, и не так уж редко…

Ну да не будем о грустном! Не стоит начинать знакомство с таким притягательным миром интернет-технологий со столь печальных вещей, как системные сбои. Чем их меньше и чем реже они случаются, тем лучше для всех нас.

Протоколы

Люди, чтобы понимать друг друга, должны разговаривать на одном языке. Точно так и с компьютерами, подключенными к сети, неважно какой — всемирной или локальной. Обмен данными по этим сетям должен проходить по единым стандартам, иначе начнется новое вавилонское столпотворение.

Стандарт, согласно которому организуются передаваемые по сети данные и команды, управляющие передачей этих данных, называется *протоколом*. В Интернете для обмена данными используются довольно много протоколов, и некоторые мы здесь вкратце рассмотрим.

Самые, так сказать, фундаментальные протоколы Интернета — *IP* (Internet Protocol, межсетевой протокол) и *TCP* (Transfer Control Protocol, протокол управления передачей). Это так называемые *протоколы низкого уровня*, определяющие самые основные параметры передаваемых данных: длину отдельных порций (*пакетов*) данных, оформление интернет-адресов получателя и отправителя и средства защиты от ошибок. Эти протоколы выполняют самую "грязную" работу по пересылке данных, не вникая, что же именно они передают.

Протокол IP занимается тем, что "упаковывает" особым образом подготовленные данные в пакеты и помещает в каждый пакет интернет-адреса компьютера-отправителя и компьютера-получателя. Протокол TCP, базирующийся на IP, обеспечивает гарантированную отправку данных, то есть следит за тем, чтобы ни один пакет не потерялся в пути, а также занимается подготовкой данных для передачи протоколом IP, в частности, разбивает слишком объемные массивы данных на несколько пакетов и потом собирает их вновь. Эти два протокола настолько взаимосвязаны друг с другом, что часто эту парочку называют одним словом *TCP/IP*, а иногда даже считают за один протокол.

Кстати, протокол IP выполняет еще одну очень важную работу: он делит один реальный, физический, канал связи Интернета (кабель, волоконно-оптическую линию или радиоканал) на несколько воображаемых, виртуальных, "канальчиков", называемых *портами IP*. Всего таких портов предусмотрено 65 535; они идентифицируются по номеру, и любой из них может быть использован для установления соединения и пересылки данных. Более того, разные программы могут одновременно пересылать по одному физическому каналу разные потоки данных, используя разные порты.

TCP/IP применяется другими протоколами, уже *высокого уровня*. Эти протоколы описывают способы оформления клиентских запросов, серверных ответов, подтверждений и сообщений об ошибках, команды, пересылаемые клиентом серверу при запросе или передаче данных, и способ кодирования передаваемой информации. (Собственно передачей этих данных, как мы уже выяснили, занимается "дуэт" TCP/IP.)

> ***НА ЗАМЕТКУ***
>
> Строго говоря, существуют еще *протоколы физического уровня*, располагающиеся "ниже" даже TCP/IP. Они определяют электрические параметры сигнала, кабелей, разъемов и пр.

Каждый сервис Интернета использует свой собственный протокол высокого уровня, а то и несколько, предназначенных для разных задач или разработанных конкурирующими организациями. Давайте рассмотрим протоколы, с которыми мы столкнемся в будущем.

Начнем мы, конечно, с WWW. Для передачи данных Всемирная паутина использует протокол *HTTP* (HyperText Transfer Protocol, протокол передачи гипертекста). Он задает набор команд, отправляемых клиентом (Web-обозревателем) Web-серверу, и способы представления пересылаемых в обе стороны данных. Пожалуй, это самый широкоизвестный протокол Интернета — всем более-менее грамотным интернетчикам знакомы эти четыре буквы.

Сервис пересылки файлов FTP использует протокол, который так и называется — *FTP*. Он также определяет набор команд для управления файлами на сервере (загрузка с сервера, отправка на сервер, создание папки, копирование, перемещение, удаление файлов и папок и т. д.) и способы кодирования файлов для пересылки по каналам связи. В этом смысле протоколы HTTP и FTP весьма похожи.

А вот электронная почта использует целых два протокола. Первый протокол — *SMTP* (Simple Mail Transfer Protocol, простой протокол пересылки почты) — используется для пересылки почты клиентом серверу. Для получения же почты от сервера клиент общается с ним по протоколу *POP3* (Post-Office Protocol 3, протокол почты 3).

Существует еще один почтовый протокол — *IMAP* (Internet Message Access Protocol, протокол доступа к почте Интернета). "Коллега" и "наследник" более старого POP3, он предоставляет больше возможностей, но распространен не так широко.

Чуть раньше мы узнали о портах IP. Так вот, каждый существующий протокол высокого уровня использует для передачи данных свой собственный порт (так называемый *порт по умолчанию*). В табл. 1.1 перечислены некоторые протоколы и используемые ими порты по умолчанию.

Таблица 1.1. *Порты IP, используемые по умолчанию для передачи данных некоторых протоколов высокого уровня*

Протокол	Используемый порт IP
HTTP	80
FTP	21
SMTP	25
POP3	110

Но почему такое странное название — "порт по умолчанию"? Давайте разберемся.

Дело в том, что все более-менее серьезные серверы предоставляют возможность изменить порт, используемый протоколом, которые они обслуживают, на другой. Например, Web-сервер может быть настроен так, чтобы использовать для "общения" с клиентами не 80-й порт, а, скажем, 8000-й. Это весьма редко, только в особых случаях, но все же применяется. (Например, если на одном серверном компьютере работают два Web-сервера, один из них использует порт по умолчанию — 80-й, — а другой настраивается на использование другого порта — того же 8000-го.)

Интернет-адреса

Теперь давайте поговорим о том, каким образом идентифицируются компьютеры, подключенные к Интернету. А именно — об интернет-адресах.

Интернет-адрес — это числовое или строковое значение, позволяющее точно идентифицировать компьютер в Сети. Именно такой интернет-адрес (точнее, два — отправителя и получателя) подставляется в каждый отправляемый по Сети пакет IP, чтобы он успешно дошел до места назначения.

> ***НА ЗАМЕТКУ***
>
> Существует, правда, возможность дать одному компьютеру сразу несколько интернет-адресов. Но используется это нечасто и в особых случаях. В дальнейшем для простоты мы будем считать, что один интернет-адрес — это один компьютер.

На заре эпохи Интернета в качестве интернет-адреса использовался *IP-адрес* — числовое значение, идентифицирующее компьютер для протокола IP. IP-адрес замечательно подходит для компьютеров, но очень плохо — для людей. Вот пример интернет-адреса:

192.168.1.10

Не очень-то наглядно, правда? Именно поэтому с расширением Интернета была введена в строй новая система интернет-адресов, которой мы пользуемся до сих пор. Это так называемые доменные имена, о которых стоит поговорить подробно.

Но прежде чем мы начнем разговор о доменных именах, давайте выясним, что такое домен. *Домен*, или *доменная зона*, — это участок Интернета, созданный для удобства управления им. Такой участок может быть крупным, мелким или вообще состоять из одного компьютера. Каждый домен обозначается строкой текста, состоящей из английских букв.

Структура доменов похожа на матрешку: мелкие домены "вложены" внутрь крупных, а крупные, в свою очередь, — внутрь гигантских. Гигантские домены называются *доменами верхнего уровня*, а вложенные в них более мелкие — *доменами нижнего уровня*.

Домены верхнего уровня бывают интернациональными и национальными. *Интернациональные домены* объединяют компьютеры по роду деятельности организаций, которым они принадлежат; к ним относятся домены com и biz (коммерческие организации), edu (образовательные), mil (военные), org (организации, не занимающиеся компьютерами и Интернетом), net (организации, занимающиеся компьютерами и Интернетом), travel (туристические организации) и некоторые другие. *Национальные домены* объединяют компьютеры по территориальному признаку и выдаются отдельным странам; это домены us (США), uk (Великобритания), fr (Франция), de (Германия), ru (Россия) и др.

Домены нижнего уровня выдаются, как правило, отдельным организациям или, опять же, по территориальному признаку. Их текстовое обозначение часто совпадает с названием этой организации или территориальной единицы (области, района, штата, города и пр.).

Если теперь записать обозначения всех доменов, в которых находится нужный нам компьютер, в порядке от более мелких к более крупным, разделив их точками, мы получим *доменное имя* этого компьютера. Так, если у нас сам компьютер имеет имя comp45, отдел, в котором он стоит, — buh (бухгалтерия), организация, включающая этот отдел, — office, а страна — ru (Россия), то мы получим такое доменное имя:

comp45.buh.office.ru

Согласитесь — запомнить это гораздо проще, чем невразумительный IP-адрес.

Да, но проблема в том, что протокол IP не понимает доменные имена! Что делать? Как преобразовать доменное имя в понятный ему IP-адрес?

Для этого используется особый сервис Интернета, называемый *DNS* (Domain Name System, система доменных имен). Клиент отправляет *серверу DNS* запрос, содержащий доменное имя, и получает в виде ответа IP-адрес, соответствующий этому доменному имени. А уж с IP-адресом он знает, что делать.

Такие серверы DNS имеются в каждом домене. А несколько самых мощных в мире серверов DNS (*корневые серверы DNS*) обслуживают домены верхнего уровня.

Так, нужный нам компьютер мы задавать научились. А как задать нужную нам серверную программу? Очень просто: перед доменным именем ставим

название протокола, обслуживаемого этой программой, знак двоеточия и два знака / (обратный слеш), вот так (обозначение протокола подчеркнуто):

http://comp45.buh.office.ru

ftp://comp45.buh.office.ru

В первом случае мы обращаемся к Web-серверу, а во втором — к серверу FTP, находящемуся на одном и том же компьютере **comp45.buh.office.ru**.

Также имеется возможность указать номер порта IP, через который производится обмен данными. Номер порта записывается после доменного имени серверного компьютера через двоеточие, вот так (подчеркнут):

http://comp45.buh.office.ru:8000

Здесь мы обращаемся к Web-серверу, работающему на компьютере **comp45.buh.office.ru** и использующему для "общения" с клиентом порт 8000.

Многие серверы (почтовые, FTP и др.) требуют от пользователя ввода его имени и, возможно, пароля. Имя пользователя помещается между названием протокола и самим доменным именем и отделяется от последнего знаком @ ("коммерческое эт"). Вот два примера задания имени пользователя в доменном имени сервера (подчеркнуто):

ftp://user@comp45.buh.office.ru

account@server.ru

Последний пример демонстрирует нам обычный адрес электронной почты. Заметим, что название протокола здесь не указывается — почтовый клиент и почтовый сервер сами знают, какой протокол использовать.

Ну а пароль пользователя помещается между именем и знаком @ и отделяется от имени двоеточием — вот так (подчеркнут):

ftp://user:password@comp45.buh.office.ru

Ну вот, с основными принципами работы Интернета мы ознакомились. Теперь давайте сосредоточимся на WWW — именно этим сервисом мы будем пользоваться на протяжении всей книги.

Основные понятия WWW

Здесь мы узнаем все о Web-страницах и Web-сайтах, выясним, чем сайт отличается от страницы, поговорим о Web-обозревателях и Web-серверах и изучим множество новых терминов.

Web-страницы и Web-сайты

Что такое Web-страница? Ответить на этот вопрос могут многие. Это интернет-документ, предназначенный для распространения через Интернет посредством сервиса WWW. А если уж говорить по-простонародному, это то, что показывает в своем окне программа-клиент для просмотра Web-страниц — Web-обозреватель.

С технической точки зрения Web-страница — это обычный текстовый файл, который можно создать в любом текстовом редакторе, например Блокноте, стандартно поставляемом в составе Windows. Этот файл содержит собственно текст Web-страницы и различные команды форматирования этого самого текста. Команды форматирования называются *тегами*, а описывает их особый язык *HTML* (HyperText Markup Language, язык гипертекстовой разметки). Файл Web-страницы обязательно должен иметь расширение htm[l].

А что такое Web-сайт? Это набор Web-страниц, подчиненных общей тематике и объединенных в единое целое (как — будет рассказано в *главе 2*). Как видим, сугубо технических отличий у Web-страницы и Web-сайта не слишком много.

Web-сайт сохраняется на жестких дисках серверного компьютера, на котором работает Web-сервер (серверная программа, обеспечивающая работу сервиса WWW), в виде набора различных файлов. Прежде всего, это, разумеется, файлы Web-страниц, составляющих сайт. Многие сайты включают файлы графических изображений, помещенных на страницы (почему изображения хранятся отдельно от самих страниц, мы узнаем в *главе 2*). Также сайт может содержать файлы архивов и дистрибутивов программ и некоторые другие файлы.

Зачастую различные файлы, составляющие сайт, хранятся в папках. Конечно, папки использовать необязательно, но так удобнее, особенно если файлов много и все они разных типов.

Для хранения всех файлов, составляющих сайт, на диске серверного компьютера создается особая папка, называемая *корневой*. Все файлы и папки сайта должны находиться только в этой папке, без малейших исключений.

Корневую папку сайта на серверном компьютере создает человек, занимающийся настройкой и обслуживанием программы Web-сервера (или же всего серверного компьютера), — *администратор*. При этом он заносит полный путь этой папки в настройки Web-сервера, чтобы последний "знал", где ее найти.

НА ЗАМЕТКУ

Нужно отметить, что все серьезные программы Web-серверов позволяют создавать так называемые *виртуальные папки*. Виртуальная папка может нахо-

диться абсолютно в любом месте файловой системы компьютера, но Web-сервер считает ее частью сайта, словно она находится в его корневой папке. Виртуальные папки также создаются администратором Web-сервера.

Чтобы получить какой-либо файл с Web-сайта, Web-обозреватель посылает управляющему им Web-серверу в составе клиентского запроса полный путь к данному файлу. Предположим, мы набрали в поле ввода интернет-адреса Web-обозревателя вот что:

http://www.somesite.ru/somepage.html

то есть запросили у Web-сервера **http://www.somesite.ru** страницу somepage.html, хранящуюся в корневой папке сайта. В этом случае Web-обозреватель отправит данному Web-серверу такой запрос:

/somepage.html

Web-сервер, получив этот запрос, найдет в корневой папке файл somepage.html, загрузит его и отправит Web-обозревателю. Если же такого файла нет или Web-сервер почему-то не сможет его загрузить, он отправит Web-обозревателю сообщение об ошибке.

Мы уже знаем, что любой пакет IP содержит, кроме всего прочего, интернет-адрес отправителя. Кроме того, интернет-адрес клиентского компьютера посылается в составе клиентского запроса HTTP. Так что Web-сервер всегда сможет узнать, куда ему отправить запрошенный файл.

Если же мы наберем в Web-обозревателе вот такой интернет-адрес:

http://www.somesite.ru/download/archive.zip

запросив архивный файл archive.zip, находящийся в папке download, вложенной в корневую папку сайта, Web-обозреватель пошлет Web-серверу **http://www.somesite.ru** вот такой запрос:

/download/archive.zip

НА ЗАМЕТКУ

Для запроса файла, находящегося в виртуальной папке, используется аналогичный интернет-адрес:

http://www.somesite.ru/pictures/picture.jpg

Здесь pictures — виртуальная папка.

Так, все прекрасно, все замечательно и все исключительно ясно! Но мы ведь нечасто набираем в поле ввода интернет-адреса Web-обозревателя интернет-адреса конкретных файлов. Скажем, мы можем набрать вот такой интернет-адрес:

http://www.somesite.ru

Тогда Web-обозреватель отправит Web-серверу в клиентском запросе вот что:

/

Как поступит Web-сервер в таком случае?

Дело в том, что одна из страниц сайта указывается в качестве так называемой *страницы по умолчанию*. Именно она отправляется Web-обозревателю, если он не указал в клиентском запросе имя файла конкретной Web-страницы. Имя файла страницы по умолчанию задается администратором Web-сервера в его настройках — как правило, default.htm[l] или index.htm[l].

Так что в приведенном ранее случае Web-сервер отправит нам страницу по умолчанию, хранящуюся в корневой папке сайта. Это может быть, например, страница default.html.

Мы можем набрать в строке ввода интернет-адреса Web-обозревателя и такой интернет-адрес — **http://www.somesite.ru/articles**. Тогда Web-обозреватель отправит в составе клиентского запроса следующее:

/articles/

И Web-сервер в ответ пошлет нам страницу по умолчанию, хранящуюся в папке articles, что вложена в корневую папку сайта.

В *главе 2*, рассматривая принципы создания Web-страниц, мы еще вернемся к интернет-адресам, используемым в WWW. А пока что закончим на этом.

> ***НА ЗАМЕТКУ***
>
> Отдельные Web-страницы и целые Web-сайты также могут быть сохранены на жестком диске клиентского компьютера и открываться в Web-обозревателе прямо с него. В этом случае роль своеобразного Web-сервера выполняет файловая система.

Web-обозреватели

Мы уже знаем, что Web-обозреватели — это программы для просмотра Web-страниц и Web-сайтов. Основная их задача — это отправить Web-серверу корректно, в соответствии со всеми стандартами, сформированный клиентский запрос, принять серверный ответ и вывести полученную страницу на экран. Для этого окно Web-обозревателя содержит поле ввода интернет-адреса и область, в которой отображается содержимое полученной Web-страницы. (Разумеется, оно также содержит заголовок, меню и панели инструментов, как и многие окна приложений Windows.)

Обычно после получения от Web-сервера файла Web-страницы (да и любого другого файла, составляющего содержимое сайта) Web-обозреватель сохраняет их на жестком диске клиентского компьютера в особой области, назы-

ваемой *кэшем*. Этот кэш может иметь вид как обычной папки (кэш Microsoft Internet Explorer или Opera), так и большого файла (кэш Mozilla или Firefox).

Зачем это нужно? Да хотя бы затем, чтобы мы смогли впоследствии просмотреть данную страницу, не подключаясь к Интернету. Все современные Web-обозреватели поддерживают так называемый *автономный режим* (offline mode), когда они отображают только те страницы, что находятся в кэше. (Кстати — исключительно удобная вещь!) Если же мы попытаемся просмотреть страницу, которой нет в кэше, Web-обозреватель предложит нам подключиться к Интернету и загрузить ее.

Но даже если мы и подключены в данный момент к Интернету, Web-обозреватель все равно активно использует кэш. Перед тем как загрузить какой-либо файл, он проверяет, не изменился ли данный файл по сравнению с тем, что находится сейчас в его кэше (если, конечно, он там уже есть). Если не изменился, он загружает нужный файл прямо из кэша, что намного быстрее.

Теперь познакомимся с программами Web-обозревателей, имеющими в настоящее время наибольшую популярность. Все они, в общем, следуют одним и тем же стандартам и отличаются друг от друга только деталями, не оговоренными в этих стандартах, и удобством для пользователей.

Настоящий король виртуальных просторов — это, конечно, Microsoft Internet Explorer. Он имеется на любом компьютере, работающем под управлением Windows (что, как говорят злые языки, и обусловило его популярность). Однако это очень мощная, быстрая, весьма нетребовательная к ресурсам и исключительно удобная программа. Автор данной книги для просмотра Web-страниц пользуется именно Internet Explorer. В настоящее время доступна версия 7.0, которая входит в состав новой версии Windows — Windows Vista; также существует отдельная версия для старых версий Windows — XP и 2003.

> ***НА ЗАМЕТКУ***
>
> Начиная с версии 7.0 Internet Explorer официально сменил имя. Теперь он называется Microsoft Windows Internet Explorer. Но автор на протяжении этой книги будет пользоваться старым наименованием этой программы как более привычным.

Второе место по популярности занимает весьма амбициозный Web-обозреватель по имени Firefox. Эта программа распространяется бесплатно, более того, ее исходные тексты открыты для изучения и модификации. Она весьма быстра и компактна, поддерживает все Web-стандарты, нетребовательна к системным ресурсам и имеет множество интересных и весьма полезных возможностей, которыми пока не может похвастаться ни один из его конкурентов. Когда писалась эта глава, вышла версия 2.0.0.11, а к моменту выхода книги в свет должна появиться версия 3.0.

Третье место занимает предшественник Firefox под названием Mozilla. Он также распространяется бесплатно, исходные тексты его открыты, а по возможностям он примерно аналогичен Firefox. Самая последняя версия этой программы имеет номер 1.7. В настоящее время разработка Mozilla прекращена, и на его основе разрабатывается Web-обозреватель SeaMonkey, чья версия 1.0 на момент написания книги уже вышла.

Некогда властелин WWW Netscape Navigator устроился на четвертом месте. Да, последняя версия Navigator под номером 9.0.4 выглядит весьма неплохо, поддерживает все современные стандарты Интернета, корректно отображает большинство Web-страниц и не очень требовательна к системным ресурсам. Но все равно время его ушло.

Пятое место оккупировано разработкой фирмы Apple, производящей широко известные в узких кругах компьютеры Macintosh, — Safari. В настоящий момент имеет хождение версия Safari 2.0. Утверждается, что это самый быстрый в мире Web-обозреватель, но недавно вышедшая версия для Windows, как пишут, не впечатляет своим быстродействием.

На шестом месте отдыхает разработка норвежских программистов Opera. Эта достаточно мощная и очень быстрая программа (хотя и уступающая, по слухам, Safari), поддерживающая все официальные Web-стандарты, тем не менее, весьма требовательна к системным ресурсам и не всегда правильно отображает некоторые Web-страницы. Последняя вышедшая в свет версия носит номер 9.25.

В настоящее время просторы WWW "бороздят" практически только шесть перечисленных ранее программ. Существует, однако, еще несколько малоизвестных Web-обозревателей, а также довольно многочисленная когорта программ, построенных на основе программного ядра Internet Explorer. Мы не будем их рассматривать.

Осталось только сказать, что выбор Web-обозревателя — это личное дело каждого. Все они поддерживают одни и те же стандарты (правда, зачастую по-своему) и предоставляют пользователю примерно одинаковый набор возможностей. Так что, как в песне поется, "думайте сами, решайте сами"…

Web-серверы

Поскольку мы не только пользователи, но и уже наполовину разработчики, нас будут интересовать также и Web-серверы. Давайте поговорим и о них.

"Зоопарк" Web-серверов ничуть не меньше "зоопарка" (или, если учесть, что Web-обозреватели жестоко конкурируют друг с другом, "серпентария") Web-обозревателей, так что мы можем подобрать себе программу по вкусу.

И, в отличие от Web-обозревателей, среди Web-серверов нет безоговорочного лидера — даже самые распространенные из них не занимают больше половины рынка.

Начнем наш краткий обзор с двух программ фирмы Microsoft: Personal Web Server и Internet Information Services. Они поставляются в составе Microsoft Windows, первая — в составе Windows 98 и Me, а вторая — в составе Windows NT, 2000, XP, 2003 и Vista. Со своими обязанностями они справляются очень хорошо, не транжирят системные ресурсы, легко настраиваются, поддерживают множество передовых интернет-технологий и при надлежащей настройке легко затыкают за пояс конкурентов. Кроме собственно Web-сервера, они содержат также простейшие серверы FTP и почты, а также множество вспомогательных программ.

Web-сервер Apache — пожалуй, самый распространенный. Среди его достоинств: полная бесплатность (более того — его исходные тексты открыты), легкость настройки, довольно высокая производительность, хорошая поддержка. По крайней мере, для Web-сайтов с небольшой загрузкой — это идеальный выбор.

Есть еще один весьма примечательный Web-сервер — Sambar. Он поддерживает такое количество интернет-технологий (многие из них — эксклюзивные, не доступные больше ни в одной программе), что просто оторопь берет — как же всем этим богатством воспользоваться и куда его применить? Недостатка у Sambar всего два: малая известность и не очень удобная настройка. (Кстати, собственный сайт автора этой книги, доступный в сети института, где он работает, функционирует под управлением именно этого Web-сервера.)

Менее известные Web-серверы мы рассматривать не будем, так как их довольно много. В любом случае, они предоставляют примерно одинаковый набор возможностей.

Что дальше?

Вот и закончился наш вводной курс интернет-технологий. Пора переходить к рассмотрению языка HTML и принципов создания Web-страниц. Именно этому будет посвящена вся следующая глава.

Глава 2

Язык HTML. Создание Web-страниц

Вооружившись необходимыми теоретическими знаниями об Интернете и интернет-технологиях, перейдем к более практическим вещам, а именно — к созданию Web-страниц. Эти страницы мы во второй части книги будем "оживлять" с помощью Web-сценариев.

Итак, что мы уже знаем о Web-страницах из *главы 1*?

Мы знаем, что Web-страницы — суть обычные текстовые файлы, созданные в любом простейшем текстовом редакторе (том же Блокноте) и сохраненные с расширением htm[l].

Мы знаем, что эти файлы содержат текст, который составляет содержимое страницы, и особые команды, называемые тегами и используемые для задания внешнего вида или назначения тех или иных *элементов* Web-страницы. С помощью этих тегов можно, например, оформить фрагмент текста как отдельный абзац, выделить его полужирным шрифтом или курсивом или превратить в заголовок.

Мы знаем, что для создания Web-страниц используется язык HTML. Этот язык определяет набор тегов, их назначение, правила написания и расстановки в тексте страницы.

Язык HTML был разработан в начале 80-х годов прошлого века Тимом Бернерсом-Ли. Кроме самого HTML, этот примечательный человек также создал первые программы Web-обозревателя и Web-сервера — и первые Web-страницы. Можно сказать, что он создал WWW и Интернет в современном его виде. И даже удостоился за это рыцарского звания!

Шло время. HTML пополнялся новыми возможностями. Менялись номера версий этого языка. Самая последняя версия — 4.01 — вышла в конце 90-х годов прошлого века; все современные Web-обозреватели ее поддерживают.

Бернерс-Ли давно отошел от активных дел, передав язык HTML на попечение организации *World Wide Web Consortium* (сокращенно — *WWWC* или, что встречается чаще, *W³C*). Это название дословно можно перевести как "Комитет Всемирной паутины". W³C издает весьма увесистые труды, описывающие набор тегов HTML и необходимые требования к Web-обозревателям.

Введение в язык HTML

Язык HTML лучше всего изучать на примерах. Так что давайте не будем болтать впустую, а сразу же создадим нашу первую Web-страничку.

Основные понятия HTML

Операционная система Windows уже содержит все нужные инструменты. И первый из этих инструментов — текстовый редактор Блокнот. Запустим его и наберем приведенный далее текст (или, как говорят программисты, код) на языке HTML. Выглядит он устрашающе, но настоящего Web-дизайнера (и, тем более, Web-программиста) так просто не испугаешь.

```
<HTML>
  <HEAD>
    <TITLE>Web-страница</TITLE>
  </HEAD>
  <BODY>
    <H1>Пример Web-страницы</H1>
    <P>Это простейшая Web-страничка, созданная в стандартном
    <EM>Блокноте</EM> и отображенная в <EM>Microsoft Internet
    Explorer</EM>.</P>
  </BODY>
</HTML>
```

После этого проверим набранный код HTML на ошибки и сохраним в файле с именем 2.1.htm. Только когда будем вводить имя файла в стандартном окне сохранения, заключим его в кавычки, иначе Блокнот по доброте душевной добавит расширение txt, и наш файл получит имя 2.1.htm.txt.

Теперь откроем полученный файл в Web-обозревателе Internet Explorer, для чего достаточно дважды щелкнуть по нему мышью. Internet Explorer — второй инструмент Web-дизайна, который припасла нам Windows. То, что мы увидим в его окне, показано на рис. 2.1.

Вот мы и создали свою первую, совсем простенькую Web-страничку!

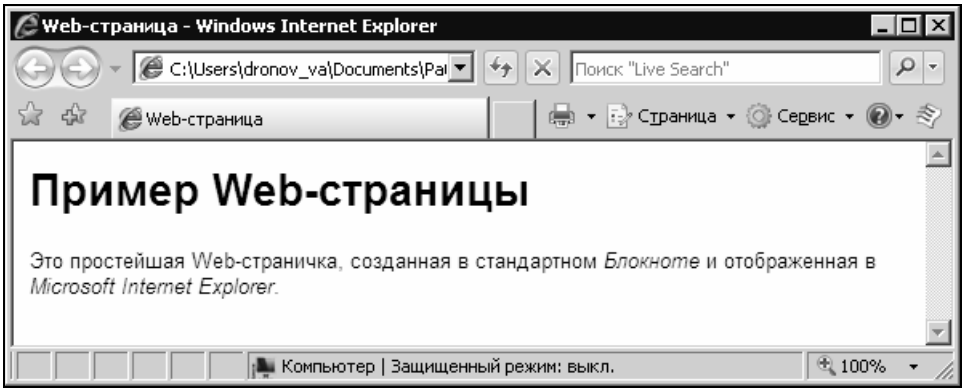

Рис. 2.1. Простейшая Web-страничка

Теперь займемся собственно языком HTML. Снова запустим Блокнот и откроем в нем только что созданную Web-страницу (проще всего это сделать, перетащив файл этой страницы в окно Блокнота). И найдем в ее HTML-коде вот этот фрагмент:

```
<H1>Пример Web-страницы</H1>
<P>Это простейшая Web-страничка, созданная в стандартном
<EM>Блокноте</EM> и отображенная в <EM>Microsoft Internet
Explorer</EM>.</P>
```

Он задает содержимое Web-страницы, видимое в окне Web-обозревателя. Помимо самого текста, здесь присутствуют какие-то слова, заключенные в символы < ("меньше") и > ("больше"). Это и есть теги HTML. Они задают форматирование текста. Скажем, строка "Блокноте" будет выведена курсивом, так как теги `<EM>` и `</EM>` задают курсивное начертание текста. Причем тег `<EM>` помечает начало курсивного фрагмента (*открывающий* тег), а тег `</EM>` — конец (*закрывающий*). А собственно фрагмент текста, заключенный между открывающим и закрывающим тегами, называется *содержимым* тега. Именно к содержимому применяется действие тега.

Также в приведенном фрагменте HTML-кода имеются теги `<P>` и `<H1>` (и соответствующие им закрывающие теги `</P>` и `</H1>`). Они создают соответственно обычный текстовый абзац и заголовок; при этом Web-обозреватель отобразит их надлежащим образом — см. рис. 2.1.

Еще один полезный и часто употребляемый тег — `<STRONG>`. Он выделяет текст полужирным шрифтом. Например, если мы изменим HTML-код нашей Web-страницы таким образом:

```
<H1>Пример Web-страницы</H1>
<P>Это простейшая Web-страничка, созданная в стандартном
```

```
<EM>Блокноте</EM> и отображенная в <STRONG>Microsoft Internet
Explorer</STRONG>.</P>
```

слова "Microsoft Internet Explorer" отобразятся полужирным шрифтом.

Теперь подытожим все, что узнали.

- Для создания Web-страниц используются особые слова — теги.
- Тег содержит имя, представляющее собой набор латинских букв и символов < и >, в которые заключается это имя.
- Закрывающий тег также включает в себя символ /, который помещается между символом < и именем тега.

Традиционно имена тегов набирают прописными буквами. Хотя стандарт HTML этого не предписывает, так HTML-код лучше читается.

Как видно, ничего особо сложного в языке HTML нет. Единственная сложность — это запомнить все нужные теги, но это вопрос времени, опыта и хорошей справочной литературы.

Вложенность тегов

Если внимательно посмотреть на HTML-код нашей страницы, можно заметить, что одни теги вложены в другие. В частности, тег <EM> вложен в тег <P>. Такая *вложенность* тегов в HTML — обычное дело.

Когда Web-обозреватель встречает тег, вложенный в другой тег, он складывает эффект от применения этих тегов. Так, в нашем случае слово "Microsoft Internet Explorer" будет частью абзаца и отобразится курсивным шрифтом. То есть Web-обозреватель сложит эффект от применения тегов <P> и <EM>.

Если мы заключим в тег <EM>, скажем, тег <STRONG> (как мы уже знаем, он задает полужирное начертание текста), то содержимое последнего будет выведено полужирным курсивом. Давайте так и сделаем. Измененный фрагмент нашей Web-страницы будет выглядеть так:

```
<P>Это простейшая Web-страничка, созданная в стандартном
<EM>Блокноте</EM> и отображенная в <EM><STRONG>Microsoft</STRONG>
Internet Explorer</EM>.</P>
```

Сохраним полученный файл под именем 2.2.htm и откроем его в Web-обозревателе. Слово "Microsoft" будет выведено полужирным шрифтом, как показано на рис. 2.2.

Обратим внимание на порядок, в котором следуют открывающие и закрывающие теги. Порядок следования закрывающих тегов должен быть обратным тому, в котором следуют открывающие теги. Говоря иначе, теги со всем

их содержимым должны полностью вкладываться в другие теги, не оставляя "хвостов" снаружи.

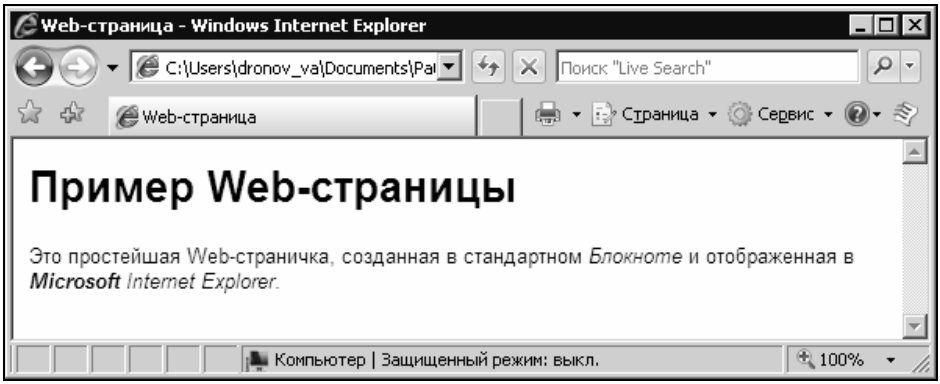

Рис. 2.2. Демонстрация эффекта вложенности тегов HTML

Если же мы нарушим это правило и напишем такой HTML-код (обратите внимание на специально перепутанный порядок следования закрывающих тегов `</STRONG>` и `</EM>`):

```
<P>Это простейшая Web-страничка, созданная в стандартном
<EM>Блокноте</EM> и отображенная в <STRONG><EM>Microsoft</STRONG>
Internet Explorer</EM>.</P>
```

то Web-обозреватель может и не отобразить наше творение правильно (хотя Internet Explorer славится своим умением исправлять мелкие ошибки Web-дизайнера). Так что об этом желательно не забывать.

Осталось только выучить несколько новых терминов. Тег, в который вложен данный тег, называется *родительским*. В свою очередь, тег, вложенный в данный тег, называется *дочерним*. Так, для тега `<EM>` в приведенном ранее 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 вложен одновременно в два тега).

У*ровень вложенности* того или иного тега показывает количество тегов, в которые он последовательно вложен. Так, если принять за точку отсчета тег `<P>`, то тег `<EM>` будет иметь первый уровень вложенности, так как он вложен непосредственно в тег `<P>`. Тег `<STRONG>` же будет иметь второй уровень вложенности, так как он вложен в тег `<EM>`, а тот, в свою очередь, — в тег `<P>`. В сложных Web-страницах уровень вложенности иных тегов может составлять несколько десятков.

Кстати, зачастую HTML-код набирается этакой лесенкой, где уровень вложенности тегов показывается величиной отступа слева. Такой код лучше читается, и в нем сразу видны некоторые ошибки (пропущенный закрывающий тег и пр.).

Две секции Web-страницы

Теперь давайте рассмотрим еще несколько тегов, используемых для служебных целей и не отображаемых Web-обозревателем. Они так и называются — *невидимые* теги. (Все остальные уже рассмотренные нами теги были *видимыми*.)

Посмотрим снова на HTML-код нашей Web-страницы. Мысленно удалим из него фрагмент, "отвечающий" за видимое содержимое. Получится вот что:

```
<HTML>
  <HEAD>
    <TITLE>Web-страница</TITLE>
  </HEAD>
  <BODY>
    . . .
  </BODY>
</HTML>
```

Видно, что все теги, задающие содержимое Web-страницы, помещаются внутри парного тега <BODY>. Этот тег используется для выделения так называемой *секции тела* Web-страницы. Все видимое содержимое страницы должно находиться в секции тела, то есть в теге <BODY>.

Другой парный тег — <HEAD> — задает *секцию заголовка* (не путайте с обычным текстовым заголовком, задаваемым тегом <H1>). В секции заголовка помещается служебная информация, описывающая саму Web-страницу и используемая Web-обозревателем для внутренних нужд. Среди этой служебной информации может присутствовать *название* Web-страницы, показываемое в заголовке окна Web-обозревателя; оно задается парным тегом <TITLE>. Собственно, секция заголовка нашей Web-страницы только ее название — "Web-страница" — и содержит.

И секция заголовка, и секция тега страницы находятся внутри парного тега <HTML>. Этот тег располагается на высшем (нулевом) уровне вложенности и не имеет родителя.

Такую структуру оформления HTML-кода страницы — с секциями заголовка и тела и тегами <HTML>, <HEAD> и <BODY> — предписывает стандарт HTML.

Лучше всегда ему следовать, иначе возможны проблемы с некоторыми Web-обозревателями.

Итак, первый шаг в изучении языка HTML мы сделали. Пора продолжить учебу: изучить новые теги и новые возможности, которые они предлагают.

Работа с текстом

Текст — это самое главное во многих Web-страницах. Именно ради текста в большинстве случаев они и создаются. Немногочисленные исключения — например, фотогалереи, — не в счет.

Форматирование фрагментов текста

Наше знакомство с HTML продолжится с рассмотрения тегов, предназначенных для форматирования фрагментов текста, а именно — выделения их особым начертанием.

Мы уже знаем, что парный тег `<STRONG>` выделяет фрагмент текста, являющийся его содержимым, полужирным шрифтом. Аналогично, парный тег `<EM>` выделяет фрагмент текста, являющийся его содержимым, курсивом.

```
Страница открыта в <STRONG>Microsoft</STRONG> <EM>Internet Explorer</EM>
```

Здесь слово "Microsoft" будет выделено полужирным шрифтом, а слова "Internet Explorer" — курсивом.

Вкладывая тег `<STRONG>` в тег `<EM>` (или наоборот), можно выделить фрагмент текста полужирным курсивом:

```
<STRONG><EM>Это полужирный курсив</EM></STRONG>
```

Не забываем только о порядке следования закрывающих тегов — он должен быть противоположным порядку открывающих тегов.

Теги `<STRONG>` и `<EM>` имеют еще одно назначение. Они позволяют задать степень важности указанных фрагментов относительно "обычного" текста, причем тег `<STRONG>` задает бóльшую важность, нежели тег `<EM>`. Так, если мы вернемся к приведенному ранее примеру

```
Страница открыта в <STRONG>Microsoft</STRONG> <EM>Internet Explorer</EM>
```

то слово "Microsoft" будет помечено как очень важное, а слова "Internet Explorer" — просто важные.

Зачем это нужно? Ну, хотя бы, затем, чтобы выделить слова, на которые посетитель страницы должен обратить внимание. Обычный Web-обозреватель выделит их полужирным шрифтом или курсивом (в зависимости от использованного нами тега), а программы чтения с экрана — скажем, интонацией.

Но что делать, если мы хотим выделить какой-либо фрагмент текста полужирным шрифтом или курсивом, не делая его при этом важным? Следует использовать стили CSS, о которых речь пойдет в *главе 3*. А мы вернемся к тегам HTML.

Следующие два тега, которые мы должны рассмотреть, — это `<SUB>` и `<SUP>`. Оба они парные. Первый выводит свое содержимое нижним индексом, а второй — верхним. Например:

```
H<SUB>2</SUB>O
E=mc<SUP>2</SUP>
```

В первом случае мы получим на "выходе" H_2O, а во втором — $E=mc^2$.

Стандарт HTML 4.01 определяет еще несколько тегов форматирования фрагментов текста, которые также дают этим фрагментам особый смысл, но используются не столь часто, как `<STRONG>` и `<EM>`. Все они перечислены в табл. 2.1.

Таблица 2.1. *Прочие теги форматирования фрагментов текста*

Тег	Смысл, даваемый им содержимому	Визуальное выделение содержимого
`<ABBR>`	Аббревиатура	Подчеркивание точечной линией (не всегда)
`<ACRONYM>`	Аналогичен `<ABBR>`	Не выделяется
`<ADDRESS>`	Адрес "обычной" почты	Курсив
`<CITE>`	Цитата	Курсив
`<CODE>`	Фрагмент исходного кода программы	Моноширинный шрифт уменьшенного размера
`<DEL>`	Фрагмент, помеченный для удаления (но не удаленный)	Зачеркивание
`<DFN>`	Термин, встречающийся в тексте впервые	Курсив
`<INS>`	Фрагмент, вставленный в текст страницы	Подчеркивание
`<KBD>`	Текст, который пользователь должен ввести с клавиатуры	Моноширинный шрифт
`<SAMP>`	Информация, выведенная какой-либо программой пользователю	Моноширинный шрифт
`<VAR>`	Обозначения в тексте имен переменных программы на каком-либо языке программирования	Курсив

Здесь нужно дать некоторые пояснения. Обычно Web-обозреватель выводит содержимое Web-страниц *пропорциональным* шрифтом, в котором символы имеют разную ширину. Такие шрифты используются для вывода текста практически всегда — и другими программами, и самой Windows, — так как набранный ими текст лучше читается. Но некоторые из перечисленных в табл. 2.1 тегов выводят текст *моноширинным* шрифтом, где все символы имеют одинаковую ширину. Моноширинные шрифты традиционно используются для набора исходных текстов программ и для программиста выглядят более привычно, поэтому содержимое тегов <CODE>, <KBD> и <SAMP> выводится именно моноширинным шрифтом.

Вообще, полезность тегов, перечисленных в табл. 2.1, весьма сомнительна. Современные Web-обозреватели не предлагают для них никакой особой поддержки, кроме выделения шрифтом, а этого легко достичь с помощью стилей CSS (см. *главу 3*). Но они присутствуют в стандарте HTML 4.01, и нам следует о них знать.

Форматирование абзацев

От фрагментов текста перейдем к более крупным "единицам" — абзацам. И посмотрим, что HTML припас нам для их форматирования.

Один тег для форматирования абзацев нам уже знаком. Это парный тег <P>, который мы использовали при написании нашей первой Web-страницы. Он, собственно, создает обычный текстовый абзац, не имеющий какого-либо особого значения.

```
<P>Это первый абзац.</P>
<P>А это второй абзац.</P>
```

При выводе на экран абзаца Web-обозреватели руководствуются следующими правилами:

- содержимое тега <P> становится содержимым абзаца;
- если содержимое абзаца не удается вывести в одну строку, выполняется перенос строк по словам;
- по умолчанию содержимое абзаца выравнивается по левому краю;
- от других абзацев он отделяется свободным пространством;
- программы чтения с экрана могут отделять один абзац от другого паузой.

Разбиение текста на абзацы существенно улучшает его читаемость, как и в случае других электронных документов (текстовых, Word и пр.) и обычных, "бумажных" книг. Это понятно.

Если нам нужно поместить в текст страницы заголовок, мы воспользуемся одним из парных тегов <Hn>, где *n* — номер от 1 до 6. Этот номер задает так называемый *уровень* заголовка, степень его "веса" относительно других заголовков и обычных абзацев. Так, уже знакомый нам тег <H1> создает заголовок первого уровня — самый "увесистый"; с его помощью создаются заголовки Web-страниц. Тег <H2> создает заголовок второго уровня, подходящий для заголовков отдельных частей большого текста, тег <H3> — заголовок третьего уровня (заголовки отдельных глав) и т. д.

```
<H1>Заголовок страницы</H1>
<H2>Заголовок первой части</H2>
<H3>Заголовок первой главы</H3>
<P>Первая глава...</P>
<H3>Заголовок второй главы<H3>
<P>Вторая глава</P>
<H2>Заголовок второй части</H2>
<H3>Заголовок третьей главы</H3>
<P>Третья глава...</P>
```

При выводе на экран заголовка правила таковы:

- содержимое тега <Hn> становится содержимым заголовка заданного уровня;
- если содержимое заголовка не удается вывести в одну строку, выполняется перенос строк по словам;
- текст заголовка выводится более крупным шрифтом, нежели содержимое обычного абзаца. Размер шрифта при этом зависит от уровня заголовка;
- по умолчанию содержимое заголовка выравнивается по левому краю;
- от других абзацев он отделяется свободным пространством, величина которого также зависит от уровня заголовка;
- программы чтения с экрана могут выделять заголовок интонацией и более длительной паузой.

Последний тег форматирования абзацев, который мы должны рассмотреть, — это <BLOCKQUOTE>. Этот тег применяется, в основном, для оформления больших цитат, вынесенных в отдельный абзац.

```
<BLOCKQUOTE>
Писать на книжках очень глупо.
Иван Фадеич Кандалупа.
(Саша Черный)
</BLOCKQUOTE>
```

Когда Web-обозреватель встретит этот тег, он примет во внимание вот что:

- содержимое тега <BLOCKQUOTE> становится содержимым абзаца, содержащего цитату;
- если содержимое абзаца не удается вывести в одну строку, выполняется перенос строк по словам;
- по умолчанию содержимое абзаца выравнивается по левому краю;
- от других абзацев он отделяется несколько большим, чем в случае тега <P>, свободным пространством;
- абзац выводится с заметным отступом слева и справа;
- программы чтения с экрана могут выделять такой абзац более длительной паузой.

Вот, собственно, и все теги форматирования абзацев.

Нужно сказать еще, что при выводе содержимого любого абзаца — обычного, заголовка или цитаты — Web-обозреватель учитывает следующие дополнительные правила:

- несколько пробелов, следующих друг за другом, трактуются как один пробел;
- переводы строк в HTML-коде трактуются как пробел.

Это позволяет избежать некоторых ошибок нерадивых операторов — тех же двойных пробелов, которые в электронных документах встречаются довольно часто.

Создание списков

Кроме абзацев и заголовков, HTML позволяет создавать списки, содержащие набор пунктов; пункты могут быть помечены либо маркерами, либо нумерацией. В первом случае говорят о *маркированном* списке, а во втором — о *нумерованном*.

Для создания маркированного списка используется парный тег <UL>. Внутри этого тега помещаются его пункты; каждый из этих пунктов должен помещаться внутри парного тега <LI>. Например:

```
<UL>
  <LI>HTML;</LI>
  <LI>CSS;<LI>
  <LI>JavaScript.</LI>
</UL>
```

Здесь мы создали маркированный список из трех пунктов. Каждый из этих пунктов по умолчанию помечается кружком с заливкой в качестве маркера.

Нумерованный список создается аналогично, но с помощью другого тега — <OL>. Вот пример HTML-кода, создающего нумерованный список из трех пунктов:

```
<OL>
  <LI>HTML;</LI>
  <LI>CSS;<LI>
  <LI>JavaScript.</LI>
</OL>
```

Пункты нумерованного списка по умолчанию нумеруются арабскими цифрами. Первый пункт имеет номер 1, второй — 2 и т. д.

А вот еще один пример нумерованного списка:

```
<OL START="3">
  <LI>HTTP;</LI>
  <LI>FTP;<LI>
  <LI>SMTP;</LI>
  <LI>POP3.</LI>
</OL>
```

Он содержит четыре пункта, причем нумерация их начинается с 3. Да, именно так!

Здесь мы столкнулись с так называемыми *атрибутами* — дополнительными параметрами тегов. Тег <OL> содержит атрибут START, указывающий номер, с которого начнется нумерация пунктов списка (*значение* атрибута).

Атрибуты тегов HTML — весьма примечательная вещь, с помощью которой мы можем в некоторых пределах менять обычное поведение тегов. Нужно только помнить следующее:

- атрибуты тегов записываются после имени тега через пробел, прямо внутри символов < и >;
- значение атрибута записывается после имени атрибута через знак двоеточия и обязательно указывается в кавычках.

Имена атрибутов, как и имена тегов, также набираются прописными буквами, а значения атрибутов — строчными. Хотя стандарт HTML этого не предписывает, так сложилась традиция, да и на читаемости HTML-кода это сказывается лучшим образом — все теги и атрибуты видны сразу.

Атрибут START может как присутствовать, так и не присутствовать в теге <OL>; во втором случае Web-обозреватель начнет нумерацию пунктов списка

со значения по умолчанию — единицы (см. пример, приведенный ранее). Говорят, что атрибут START *необязательный*.

Тег <LI>, создающий пункт списка, поддерживает атрибут VALUE. Этот необязательный атрибут указывает номер данного пункта; последующие пункты будут нумероваться начиная с заданного номера.

```
<OL>
  <LI>HTTP;</LI>
  <LI>FTP;<LI>
  <LI VALUE="5">SMTP;</LI>
  <LI>POP3.</LI>
</OL>
```

Здесь пункты "HTTP" и "FTP" будут иметь номера 1 и 2, а пункты "SMTP" и "POP3" — 5 и 6, так как для пункта "SMTP" был задан номер 5.

Понятно, что атрибут VALUE имеет смысл указывать только в случае нумерованного списка. В случае списка маркированного он будет проигнорирован.

Зачем может понадобиться задание начала нумерации пункта в списке? Например, если нам понадобится "разбить" большой список обычным абзацем, мы можем поступить следующим образом. Сначала мы создадим обычный нумерованный список, содержащий пункты, которые должны следовать до этого абзаца. Потом мы создадим сам абзац и другой список, содержащий пункты, которые должны следовать после абзаца. Напоследок задаем для второго списка начальный номер пункта, следующий за тем, на котором закончилась нумерация первого списка. Так мы обеспечим сквозную нумерацию в обоих списках.

Осталось только сказать, что пункты списков обоих видов выводятся с отступом слева. А промежуток между пунктами делается существенно меньше, чем промежуток между абзацами.

Стандарт HTML 4.01 предусматривает возможность создания списков еще одной разновидности — *списков определений*. Такой список содержит перечень неких терминов, требующих разъяснения, с самими этими разъяснениями. Используются такие списки редко, но нам следует о них знать.

Сам список определений создается парным тегом <DL>, внутри которого помещаются термины и разъяснения. Каждый термин помещается в парный тег <DT>. Разъяснение этого термина должно находиться сразу же после него в парном теге <DD>.

```
<DL>
  <DT>HTTP</DT>
  <DD>Протокол, используемый сервисом WWW</DD>
```

```
  <DT>FTP</DT>
  <DD>Протокол, используемый сервисом обмена файлами</DD>
</DL>
```

Здесь мы создали список определений, содержащий два термина с разъяснениями.

Пункты списка определений никак не помечаются и не нумеруются. Вообще, стандарт HTML 4.01 отдает оформление такого списка на откуп Web-обозревателям. Большинство из них оформляет их как обычные списки, за тем исключением, что разъяснения выводятся с бóльшим отступом слева, чем термины, а пункты не маркируются и не нумеруются.

Управление переносом строк

Мы уже знаем, что Web-обозреватель сам выполняет перенос строк по словам, если те не помещаются в окно целиком. Стандарт HTML 4.01 предусматривает некоторые средства по управлению процессом переноса строк.

Иногда бывает нужно выполнить перенос строки в заданном месте абзаца. Для этого служит весьма примечательный тег <BR>. Примечателен он тем, что является не двойным, как изученные нами теги, а *одинарным*, то есть не имеет закрывающей "пары".

Тег
 выполняет перенос строки в том месте текста абзаца, где он встретился. При этом перенесенная на другую строку часть текста все равно останется частью абзаца.

```
<P>Сейчас мы выполним перенос строк...<BR>Перенесли!</P>
```

Слово "Перенесли!" в этом абзаце будет перенесено на другую строку.

Тег
 "работает" в любом случае, даже если Web-обозревателю хватит места, чтобы разместить остаток текста абзаца, расположенного после этого тега, в той же строке. Поэтому достигаемый им эффект называют *жестким переносом строк*.

Рассмотрим другой случай. Мы пишем текст Web-страницы и хотим, чтобы в некоторой фразе перенос строк не выполнялся в любом случае. Такое может случиться, если мы, например, помещаем на Web-страницу фрагмент исходного кода программы на языке программирования, который не допускает переносов строк внутри выражений (о выражениях см. *главу 4*). (К таким языкам относится, в частности, Microsoft Visual Basic.) Можно ли указать Web-обозревателю не выполнять перенос строк внутри этой фразы?

Можно. И специально для этого предназначен парный тег <NOBR>. Помещаем нужную нам "непереносимую" фразу внутри этого тега и наслаждаемся полученным результатом.

```
<P>Эта фраза <NOBR>не будет перенесена на другую строку</NOBR>!</P>
```

> ***Внимание!***
>
> Тег <NOBR> не является частью стандарта HTML 4.01 (*нестандартный тег*). Это значит, что некоторые Web-обозреватели его могут не поддерживать. (Хотя самые распространенные все-таки поддерживают.)

Сказав о теге <NOBR>, нельзя не упомянуть об одинарном теге <WBR>. Этот тег может присутствовать только внутри тега <NOBR> и выполняет *мягкий перенос строк*. В месте "непереносимой" строки, где он встретился, Web-обозреватель может выполнить перенос строк, — именно в этом месте и больше ни в каком другом.

```
<P><NOBR>Эта фраза будет перенесена на другую строку только в<WBR>этом
↳месте.</NOBR></P>
```

Встретив приведенный фрагмент HTML-кода, Web-обозреватель, если понадобится, сможет выполнить перенос строк только между словами "в" и "этом".

В "непереносимых" строках, помещенных в тег <NOBR>, можно использовать и тег жесткого переноса <BR>. Только нужно ли?..

Специальные символы

Есть, кстати, еще один способ управления переносом строк. Он позволяет запретить перенос строки между определенными словами. Для этого достаточно между данными словами вставить не обычный пробел, а *неразрывный* — он как раз и запрещает Web-обозревателю выполнять перенос строк.

Неразрывный пробел обозначается особой последовательностью символов — *литералом* — . (Обратите внимание: символ точки с запятой в конце обязателен.)

```
<P>Между этими словами запрещен перенос строк.</P>
```

В приведенном примере между словами "этими" и "словами" перенос строк не будет выполняться в любом случае.

Здесь мы столкнулись с так называемыми *специальными символами* HTML. Такие символы не могут быть вставлены в HTML-код напрямую и требуют использования литералов.

Помимо неразрывного пробела, классическим примером специальных символов служат знаки < ("меньше") и > ("больше"). Напрямую мы их вставить в HTML-код никак не сможем — эти символы используются для обозначения тегов и обрабатываются Web-обозревателем особым образом. Так, если мы напишем

```
<P>x > y</P>
```

символ > не будет выведен на экран, да и не факт, что сам абзац будет правильно обработан. Нам придется использовать соответствующий литерал, обозначающий символ > (выделен полужирным шрифтом):

```
<P>x &gt; y</P>
```

Да, символ > обозначается литералом `>`. А символ < — литералом `<`.

В табл. 2.2 представлены наиболее употребляемые специальные символы HTML и соответствующие им литералы.

Таблица 2.2. *Некоторые специальные символы HTML и соответствующие им литералы*

Специальный символ	Литерал
Кавычка	`&quot;`
Знак "меньше"	`<`
Знак "больше"	`>`
Неразрывный пробел	` `
Амперсанд	`&`
Евро	`&#8364;`
©	`&copy;`
®	`&reg;`
Левая кавычка	`&#8220;`
Правая кавычка	`&#8221;`
Длинное тире	`&#8212;`

Заметим, что для вставки символов евро, левой и правой кавычек и длинного тире используется *код* этих символов. Это своего рода уникальный номер, которым каждый символ представлен в памяти компьютера. Мы еще поговорим о кодах символов в конце этой главы.

Осталось рассмотреть еще один специальный символ HTML, стоящий несколько особняком. Это *горизонтальная линия* — обычная горизонтальная линия, которая может использоваться, чтобы отделить одну часть страницы от другой.

Горизонтальная линия вставляется в страницу с помощью одинарного тега <HR> и помещается Web-обозревателем в том месте, где встретился этот тег.

```
<P>Одна часть страницы.</P>
<HR>
<P>Другая часть страницы.</P>
```

По умолчанию Web-обозреватель рисует горизонтальную линию в "трехмерном" виде, с ясно видимой тенью. Но если включить в тег <HR> особый атрибут NOSHADE, тень у линии рисоваться не будет. Этот атрибут интересен тем, что не принимает значения (*атрибут без значения*) — достаточно одного его присутствия в теге.

```
<HR NOSHADE>
<P>Это линия без тени.</P>
```

Текст фиксированного формата

Рассмотрим еще один способ оформления текста, доступный в HTML. Это последнее, что нам нужно узнать о работе с текстом.

Давайте напишем вот такую страницу:

```
<HTML>
  <HEAD>
    <TITLE>Эксперименты с фиксированным форматом</TITLE>
  </HEAD>
<BODY>
  <P>Это
небольшой
пример
Web-страницы.</P>
  </BODY>
</HTML>
```

Назовем ее "Эксперименты с фиксированным форматом" (см. содержимое тега <TITLE>). Сохраним его в файле 2.3.htm и откроем в Web-обозревателе. То, что мы увидим, показано на рис. 2.3.

Видно, что содержимое абзаца, разбитое нами на четыре строки, Web-обозреватель вывел в одну. Собственно, он так и должен был поступить, ведь стандарт HTML требует преобразовывать переводы строк в HTML-коде в пробелы и игнорировать несколько следующих подряд пробелов, преобразуя их в один. Web-обозреватель в данном случае делает все правильно.

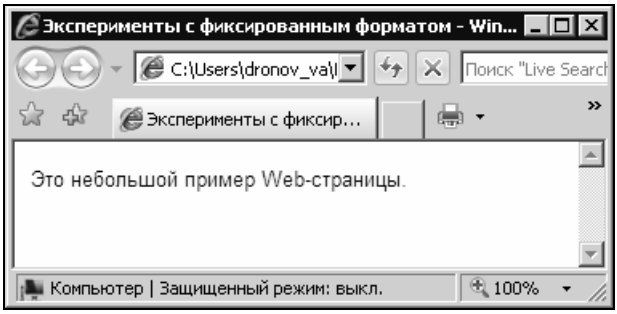

Рис. 2.3. Изначальная страница "Эксперименты с фиксированным форматом"

А теперь мы немного изменим приведенный ранее HTML-код следующим образом (исправления выделены полужирным шрифтом):

```
<HTML>
  <HEAD>
    <TITLE>Эксперименты с фиксированным форматом</TITLE>
  </HEAD>
  <BODY>
    <PRE>Это
небольшой
пример
Web-страницы.</PRE>
  </BODY>
</HTML>
```

Видно, что мы только изменили тег, с помощью которого создается абзац. Сохраним исправленный код в файле 2.4.htm. Если мы откроем новую страницу в Web-обозревателе, то увидим нечто совершенно отличное от виденного ранее (рис. 2.4).

Во-первых, Web-обозреватель вывел текст абзаца "как есть" — со всеми переводами строк. Во-вторых, он вывел этот текст моноширинным, а не пропорциональным шрифтом. Забегая вперед, скажем также, что, если бы мы использовали в тексте несколько следующих друг за другом пробелов, Web-обозреватель также вывел бы их "как есть", без предписанных стандартом HTML преобразований.

А все потому, что мы использовали в странице 2.4.htm тег `<PRE>` вместо тега `<P>`! Этот парный тег предписывает Web-обозревателю вывести заключенный в нем текст без всяких преобразований. То есть создать *текст фиксированного формата*.

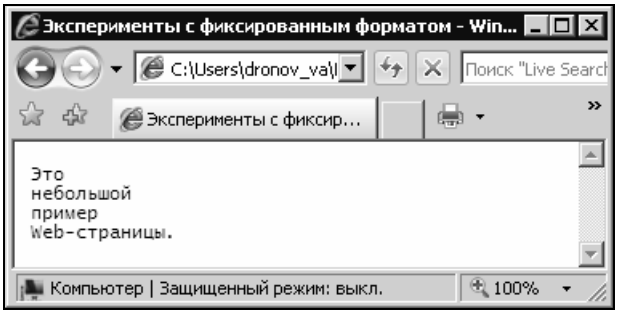

Рис. 2.4. Исправленная страница "Эксперименты с фиксированным форматом"

Текст фиксированного формата поможет нам в случае, если понадобится преобразовать в Web-страницу документ, набранный в обычном текстовом формате и сохраненный в файле с расширением txt. Для этого достаточно создать простую Web-страницу с минимумом необходимых тегов (<HTML>, <HEAD> и <BODY>), вставить в тело этой страницы тег <PRE> и поместить в него содержимое преобразуемого документа.

Кстати, никто и ничто не мешает использовать в таком тексте знакомые нам теги форматирования фрагментов текста, создания гиперссылок, вставки графических изображений и некоторые другие, о которых мы поговорим позже в этой главе.

На этом о работе с текстом все! Пора выяснить, каким образом Web-страницы связываются друг с другом, образуя сайт.

Работа с гиперссылками

Не только и не столько Web-страницы прославили Интернет. Свою роль внесло еще одно замечательное изобретение, буквально связавшее разрозненные документы в настоящую паутину. Благодаря этому изобретению мы можем с легкостью перемещаться по страницам, ведь любой уголок Всемирной сети находится от нас на расстоянии щелчка мыши.

Создание гиперссылок

Это *гиперссылки* — особые связи, ведущие от одной Web-страницы к другой. (Часто их называют просто *ссылками*.) Именно по ним мы щелкаем мышью, если хотим перейти на другую страницу.

Гиперссылки создаются с помощью особого парного тега <A>. Внутрь этого тега помещается текст, который станет собственно гиперссылкой и по кото-

рому посетитель будет щелкать мышью, чтобы перейти на другую страницу. (Эту страницу называют еще *целевой*.) Интернет-адрес целевой страницы указывается в атрибуте `HREF` тега `<A>`.

Атрибут `HREF` обязательно должен присутствовать в теге `<A>`, задающем гиперссылку, — в данном случае это *обязательный* атрибут. (Как мы выясним далее, тег `<A>` используется для создания не только гиперссылок, и в этом случае атрибут `HREF` не нужен.)

Вот пример гиперссылки:

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

Она указывает на страницу page125.html, хранящуюся в папке pages, вложенной в корневую папку сайта, на сайте **http://www.somesite.ru**.

А вот пример HTML-кода, создающего сразу две гиперссылки в одном абзаце:

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

Сама же страница, где присутствует этот код (*текущая страница*), вероятно, хранится в файле 23.html.

Гиперссылка необязательно должна указывать на Web-страницу. Например:

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

Данная гиперссылка указывает на архивный файл archive.zip, хранящийся в той же папке, что и текущая страница. При щелчке на этой гиперссылке Web-обозреватель предложит загрузить этот архивный файл и либо открыть его, либо сохранить на диске клиентского компьютера.

Тег `<A>` поддерживает также необязательный атрибут `TARGET`. Он указывает *цель гиперссылки*, задающую, 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 (целевая страница открывается в том же окне), нужно присвоить атрибуту `TARGET` значение "_self" (это его значение по умолчанию) или вообще убрать данный атрибут из тега `<A>`.

Для создания гиперссылки, указывающей на файл, что хранится на сервере FTP, используется все тот же тег <A>. Просто нужно подставить в атрибут HREF необходимый интернет-адрес. Например:

```
<A HREF="ftp://ftp.someserver.ru/public/archive.zip">Архив</A>
```

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

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

При щелчке на такой гиперссылке ничего не произойдет.

Web-обозреватель выводит гиперссылки согласно следующим правилам:

- 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* гиперссылки), выводится ярко-красным цветом;
- при помещении курсора мыши на гиперссылку Web-обозреватель меняет его форму на "указующий перст".

Это поведение по умолчанию можно изменить с помощью стилей CSS (см. *главу 3*).

Интернет-адреса в WWW

Мы уже рассмотрели три примера гиперссылок. Интернет-адреса целевых страниц (и любых других файлов) в них задаются двумя разными способами. Давайте поговорим о них.

В первой гиперссылке присутствует такой интернет-адрес:

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 используются, если нужно создать гиперссылку, указывающую на файл, что входит в состав другого сайта.

Однако если гиперссылка указывает на файл, входящий в состав того же сайта, что и текущая страница, будет сподручнее использовать *сокращенный* интернет-адрес. Такие интернет-адреса не содержат интернет-адреса Web-сервера (он и так известен Web-обозревателю), а только имя нужного файла. Они бывают двух типов, которые мы сейчас рассмотрим.

Сокращенные интернет-адреса первого типа задают путь к файлу, который нужно получить (*целевому* файлу), относительно корневой папки сайта. Они содержат в своем начале символ / (слеш), который и говорит Web-серверу, что путь нужно отсчитывать относительно корневой папки.

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

Например, интернет-адрес

/page3.html

указывает на файл page3.html, хранящийся в корневой папке сайта. А интернет-адрес

/articles/article1.html

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

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

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

Рассмотрим несколько примеров подобных интернет-адресов.

archive.zip

Этот интернет-адрес указывает на Web-страницу, хранящуюся в файле archive.zip в той же папке, что и файл текущей страницы. Собственно, этот интернет-адрес взят из третьей гиперссылки, которую мы рассмотрели ранее.

chapter3/page1.html

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

../contents.html

А этот указывает на 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 называются *относительными*. Они используются, если нужно создать гиперссылку на файл, хранящийся в той же папке, что

и текущая 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 использовать 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.

Почтовые гиперссылки

Мы также можем создать гиперссылку, указывающую на почтовый адрес (*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 гиперссылка будет выглядеть так:

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

Атрибут `TARGET` в этом случае игнорируется, даже если присутствует в теге `<A>`. И это понятно — все равно для написания писем используется на Web-обозреватель, а другая программа.

Якоря

Напоследок мы узнаем еще об одной, весьма специфичной, разновидности гиперссылок. Это так называемые *якоря* (anchors). В отличие от других гиперссылок, они не указывают на другую Web-страницу (иной файл, адрес электронной почты), а помечают некоторый фрагмент текущей страницы, чтобы другая гиперссылка могла на него сослаться.

Зачем это нужно? Ну, например, если ваша Web-страница содержит большой текст, разбитый на главы (кто знает, может, вы написали роман и выложили его в Сети), вы можете поместить в ее начале оглавление. При щелчке на пункте оглавления посетитель тотчас "перескакивает" на соответствующую главу вашего труда. Очень удобно!

Якоря создаются с помощью все того же (можно сказать — универсального) тега `<A>`. Только в данном случае атрибут `HREF` в нем присутствовать не должен (сказано же было, что якоря — весьма специфичная разновидность гиперссылок). Вместо него в тег `<A>` помещается атрибут `ID`, который в данном случае является обязательным.

А что же задает атрибут `ID`? А задает он особое, уникальное в пределах текущей страницы имя создаваемого якоря. Это имя однозначно идентифицирует данный якорь и должно состоять из латинских букв, цифр и знаков подчеркивания, причем начинаться должно с буквы.

И еще. Мы уже знаем, что тег `<A>` — парный и в случае гиперссылки в него помещается текст, который этой самой гиперссылкой и станет. В случае же якоря в этот тег не помещается ничего (так называемый *пустой тег*). По крайней мере, так чаще всего поступают.

Вот пример HTML-кода, создающего якорь:

```
<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 "романа".

Хорошо! Якорь готов. Как теперь на него сослаться с другой страницы? Очень просто. Для этого достаточно создать обычную гиперссылку, добавив в ее интернет-адрес имя нужного нам якоря. Имя якоря ставится в самый конец интернет-адреса и отделяется от него символом `#` ("решетка").

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

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

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

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

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

Внимание!

Старые Web-обозреватели не поддерживают атрибут `ID`, зато поддерживают атрибут `NAME`, имеющий то же назначение. Поэтому при создании якорей рекомендуется использовать оба этих атрибута:

```
<A ID="chapter3" NAME="chapter3"></A>
```

Работа с графикой

Первая версия языка HTML, разработанная еще Тимом Бернерсом-Ли, не поддерживала помещение на Web-страницы графических изображений. Возможности по работе с графикой появились только в HTML 2.0. Они не бог весть какие богатые, но их вполне достаточно.

Давайте выясним, что HTML предлагает нам в этом плане. И начнем мы с теории — разберемся, в каком виде хранятся графические изображения и какие форматы используются для публикации графики в Интернете.

Внедренные элементы

С чем мы имели дело ранее? В основном, с абзацами и заголовками, иначе говоря, с *текстовыми элементами* Web-страниц. Все текстовые элементы создаются с помощью соответствующих тегов языка HTML, изученных нами ранее — `<P>`, `<H1>`, `<STRONG>`, `<EM>`, `<A>` и пр., — и сохраняются в текстовых файлах с расширением htm[l], то есть в файлах собственно Web-страниц.

Но графические изображения в виде текста сохранить невозможно — они имеют другую природу. Попробуйте как-нибудь открыть в текстовом редакторе, том же Блокноте, графическое изображение. Вы увидите только мешанину непонятных символов — текстовый редактор будет стараться вывести данные изображения в виде текста, но это у него так и не получится.

Вывод: графические изображения не удастся сохранить в составе Web-страницы. Что же делать?

А не хранить их в самой Web-странице! Графические изображения, предназначенные для размещения на страницах, хранятся в отдельных файлах. А в коде HTML этих страниц с помощью особых тегов ставятся своего рода ссылки на эти файлы. Встретив такую ссылку, Web-обозреватель загружает нужный файл и выводит содержащееся в нем изображение в месте Web-страницы, где встретилась данная ссылка.

Из-за этого графические изображения называют *внедренными элементами* Web-страниц.

Теперь давайте поговорим о форматах графических файлов, поддерживаемых Web-обозревателями. (*Форматом файла* называется способ записи в файл массива каких-либо данных, в данном случае — изображения.) Именно в них должна храниться интернет-графика.

Форматы интернет-графики

Трудолюбивое человечество за весь срок существования компьютеров наплодило несколько десятков графических форматов файлов. Но для хранения интернет-графики используются только немногие из них.

Прежде всего, это, конечно, хорошо знакомые нам форматы *GIF* (Graphics Interchange Format, формат обмена графикой) и *JPEG* (Joint Photographic Experts Group, объединенная группа экспертов по фотографии). Еще стоит упомянуть весьма напористого новичка — формат *PNG* (Portable Network Graphics, перемещаемая сетевая графика), который сейчас отвоевывает позиции у более "старых" форматов.

Самые популярные форматы — GIF и JPEG. Хотя бы потому, что изображения хранятся в них в сжатом виде, поэтому занимают намного меньше места, нежели в формате, не поддерживающем сжатие. Вот только способы сжатия, поддерживаемые этими форматами, разные. Так, формат GIF сжимает изображение не очень сильно, но зато обеспечивает лучшее качество и поэтому применяется для хранения изображений — элементов оформления страниц (всяческие линейки, маркеры списков и т. п.) и штриховых изображений (карт, схем, рисунков карандашом и пр.). Формат JPEG сжимает изображения сильнее, но с некоторыми потерями качества и поэтому применяется для полутоновых изображений с большим количеством цветов (фотографии, картины и пр.), на которых такие потери не так заметны.

Формат GIF обладает особым свойством, за которое его любят Web-художники. В одном GIF-файле можно сохранить последовательность графических изображений, фактически — настоящий фильм (*анимированный GIF*). Бесчисленные "живые" (даже слишком) рекламные картинки — баннеры, — в последнее время заполонившие Интернет, хранятся, в основном, именно в этом формате.

Другое достоинство формата GIF — возможность задать *"прозрачный" цвет*. Закрашенные этим цветом области изображения станут своего рода "дырками", сквозь которые будет "просвечивать" фон родительского элемента.

Формат PNG, как говорят его создатели, объединяет возможности GIF и JPEG, не "прихватывая" заодно с собой их недостатки. Однако хранить в себе анимированные изображения он не может.

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

Кроме описанных ранее, в Web-дизайне широко используется еще один формат — *Shockwave/Flash*. Назвать это чудо программистского искусства графическим форматом не поворачивается язык — он позволяет хранить целые фильмы со звуковым сопровождением, причем фильмы интерактивные, реагирующие на действия пользователя. При этом файлы Shockwave/Flash, имеющие расширение swf, отличаются небольшими размерами и быстро загружаются даже по медленным каналам связи.

Формат Shockwave/Flash используется очень широко. С его помощью создаются те же баннеры, фильмы, элементы оформления Web-страниц и даже целые сайты. Вообще, и сам этот формат, и программный пакет Adobe Flash, в котором создается графика этого формата, заслуживают отдельный книги.

Вставка графических изображений

Чтобы поместить на страницу графическое изображение, следует использовать одинарный тег `<IMG>`. Для указания интернет-адреса файла с графическим изображением используется обязательный атрибут `SRC` этого тега.

```
<IMG SRC="/pics/picture4.jpg">
```

Встретив приведенный HTML-код, Web-обозреватель загрузит файл picture4.jpg, хранящийся в папке pics, вложенной в корневую папку сайта, и поместит его в то место, где встретился тег `<IMG>`.

Многие Web-обозреватели по умолчанию окружают графические изображения довольно толстой черной рамкой, что не очень красиво. Убрать ее или же задать для нее свои параметры (цвет и толщину) можно с помощью стилей CSS, о которых мы поговорим в *главе 3*.

Тег `<IMG>` поддерживает также два необязательных атрибута, которые мы сейчас рассмотрим. Один из них весьма полезен и даже рекомендован к использованию, другой — не очень.

Атрибут `ALT` задает так называемый *текст замены*. Зачем он нужен?

Дело в том, что Web-обозреватель, загрузив и выведя на экран саму страницу, не сразу выводит помещенные на нее графические изображения. Какое-то время уходит на загрузку файлов, в которых они хранятся, и это время может быть довольно большим, особенно если канал связи медленный, а изображения большие. Пока идет загрузка изображений, Web-обозреватель отображает на странице в тех местах, где они должны находиться, пустые рамки. И в этих-то пустых рамках выводится текст замены (если, конечно, он задан).

```
<IMG SRC="/pics/picture.jpg" ALT="Фотография">
```

ВНИМАНИЕ!

Если Web-обозреватель почему-то не сможет загрузить изображение, на странице в том месте, где оно должно присутствовать, так и останется пустая рамка с текстом замены.

Текст замены рекомендуется задавать для всех изображений, за исключением элементов оформления Web-страниц (рамок, кнопок и пр.). Так, по крайней мере, посетитель поймет, что в данном месте должно быть изображение, и выяснит его назначение.

Некоторые Web-обозреватели (в частности, Internet Explorer) поддерживают своего рода заменитель текста замены. Они позволяют задать для тега `<IMG>` изображение низкого качества и, соответственно, небольшого размера — оно будет загружено вскоре после загрузки страницы и выведено на нее на то время, пока будет загружаться "полноразмерное" изображение. Интернет-адрес такого изображения замены задается в атрибуте `LOWSRC`.

```
<IMG SRC="/pics/picture.jpg" LOWSRC="/previews/picture4_low.jpg">
```

Вообще, этот атрибут редко используется. Обычно, если нужно опубликовать на странице большое изображение, создают особую страницу "предпросмотра", на которую помещается изображение низкого качества и гиперссылка, ведущая на страницу с "полноразмерным" изображением. Посетитель сам сможет решить, нужно ли ему "полноразмерное" изображение, и, если решит, что нужно, щелкнет по этой гиперссылке.

Специальные изображения

Так, помещать на страницы графику мы научились. Теперь давайте познакомимся с двумя разновидностями специальных графических изображений: изображениями-гиперссылками и картами-изображениями.

Изображения-гиперссылки

Изображение-гиперссылка — это обычное изображение, выполняющее "по совместительству" функции гиперссылки. Такие изображения часто встречаются на современных Web-страницах — они заметно нагляднее, чем обычные, "текстовые" гиперссылки, а при удачном применении прекрасно вписываются в дизайн страницы.

Создать изображение-гиперссылку очень просто. Для этого достаточно поместить тег `<IMG>`, создающий изображение, в тег `<A>`, создающий гиперссылку.

```
<A HREF="/pages/page2.html"><IMG SRC="/pics/page2_logo.gif"></A>
```

При наведении курсора мыши на такое изображение курсор сменится на хорошо знакомый нам по гиперссылкам "указующий перст". А рамка, выводимая Web-обозревателем вокруг изображения по умолчанию, будет иметь цвет, используемый для выделения гиперссылок (по умолчанию синий для обычных, темно-красный для посещенных, ярко-красный для активных гиперссылок).

Карты-изображения

Карта-изображение — это обычное изображение, разбитое на части, каждая из которых представляет собой гиперссылку, указывающую на свой интернет-адрес. (Такие части изображений, имеющие функциональность гиперссылок, часто называют *горячими областями*.) Можно сказать, что это набор изображений-гиперссылок, объединенных в одном изображении.

Такие карты-изображения часто используются для создания заголовков сайта, фрагменты которого представляют собой гиперссылки, указывающие на определенную страницу. Вообще, областей применения у карт-изображений довольно много.

Карта-изображение состоит из двух частей. Первая часть — само изображение, которую мы превратим в карту-изображение. Вторая часть — это *карта*, определяющая форму, координаты и размеры отдельных горячих областей. С нее-то и начинается создание карты-изображения.

Карта создается с помощью парного тега <MAP>. Формат его написания таков:

```
<MAP NAME="<имя карты>">
  <набор описаний горячих областей>
</MAP>
```

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

С помощью обязательного атрибута NAME задается уникальное в пределах страницы имя карты. Это имя должно удовлетворять тем же требованиям, что и имя якоря. Впоследствии по нему мы сможем сослаться на карту.

Внутри тега <MAP> приводится набор описаний содержащихся в карте горячих областей. Каждое из этих описаний создается с помощью одинарного тега <AREA>. Формат его написания таков:

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

Давайте рассмотрим все атрибуты этого тега.

Необязательный атрибут SHAPE задает форму горячей области. Обязательный атрибут COORDS перечисляет координаты, необходимые для построения этой области. SHAPE может принимать следующие значения:

- "rect" — прямоугольная горячая область. Атрибут COORDS в этом случае имеет вид 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 имеет 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, он имеет смысл только в том случае, если мы создаем именно горячую область, а не "дырку" с атрибутом NOHREF.

Создав карту, можно приступать к подготовке изображения, которое станет картой-изображением. Оно создается точно так же, как обычное изображение. Единственное исключение — тег <IMG>, создающий это изображение, должен включать атрибут USEMAP, указывающий на созданную ранее карту.

Формат написания атрибута USEMAP таков:

```
USEMAP=[<интернет-адрес страницы>]#<имя карты>
```

то есть аналогичен формату атрибута HREF в гиперссылках, ссылающихся на якоря. В значении атрибута указываем имя карты, предварив его символом # ("решетка"). Если карта находится на другой странице, интернет-адрес этой страницы указываем перед символом "решетки".

Напоследок приведем пример карты:

```
<MAP NAME="simplemap">
  <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 страницы page1.html и page2.html, многоугольную область, не ссылающуюся никуда, и прямоугольную область, ссылающуюся на страницу appendix.html. Причем последняя горячая область при щелчке на ней откроет страницу в новом окне Web-обозревателя.

Теперь осталось создать изображение, которое станет картой-изображением:

```
<IMG HREF="/pics/map_image.gif" USEMAP="#simplemap">
```

Вот и все об изображениях-картах.

Работа с таблицами

Таблицы — пожалуй, идеальный выбор, если нужно расположить множество данных на небольшой книжной, журнальной или Web-странице. Собранные в таблицу данные отображаются очень наглядно, найти в них нужные сведения ничего не стоит. Кроме того, таблицы — это красиво!

Первые версии HTML не поддерживали таблицы ни в каком виде. Несчастные Web-дизайнеры пытались как-то выйти из положения, создавая их в виде текста фиксированного формата (см. ранее). Получалось, конечно, некрасиво, но что поделаешь. Но с появлением версии HTML 3.2 и поддерживающих ее Web-обозревателей они получили возможность создавать какие угодно таблицы. Чем и занимаются до сих пор.

Создание таблиц

Таблица HTML формируется в четыре этапа. Все они перечислены далее.

1. Создание самой таблицы.
2. Создание строк таблицы.
3. Создание ячеек таблицы.
4. Создание содержимого ячеек таблицы.

Давайте рассмотрим их по порядку.

Итак, этап первый — создание самой таблицы. Делается это с помощью парного тега `<TABLE>`.

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

Как видим, здесь все просто — достаточно написать только один этот тег.

Второй этап — создание строк таблицы — также не представляет особого труда. Каждая строка таблицы создается с помощью парного тега <TR>. Просто вставляем нужное количество (по числу строк таблицы) этих тегов в созданный ранее тег <TABLE>.

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

Обратим внимание, что само содержимое ячеек таблицы пока еще не задано. Мы только формируем, можно сказать, "скелет" таблицы, а именно его крупные "кости".

На третьем этапе этот "скелет" обрастает более мелкими "косточками" — внутри каждой строки создаются описания ячеек. Нужно сразу сказать, что табличные ячейки бывают двух видов: обычная и *ячейка заголовка*. Последняя отличается тем, что текст, составляющий ее содержимое, выделяется полужирным шрифтом и выравнивается по центру. Ячейки заголовки используются для формирования "шапки" таблицы.

Обычная ячейка таблицы создается с помощью парного тега <TD>. Для создания ячейки заголовка применяется парный тег <TH>.

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

Остается четвертый этап — наполнение "скелета" таблицы "мясом", то есть содержимым. Содержимое это будет находиться внутри тегов <TD> и <TH>, как показано далее.

```
<TABLE>
  <TR>
    <TH>Заголовок столбца № 1</TH>
```

```
    <TH>Заголовок столбца № 2</TH>
  </TR>
  <TR>
    <TD>Ячейка № 2.1</TD>
    <TD>Ячейка № 2.2</TD>
  </TR>
</TABLE>
```

А теперь запомним несколько правил, которые нарушать ни в коем случае не следует. В противном случае Web-обозреватель может отобразить таблицу неверно или вообще не отобразить.

- Тег `<TR>` может находиться только внутри тега `<TABLE>`.
- Теги `<TD>` и `<TH>` могут находиться только внутри тега `<TR>`.
- Содержимое таблицы может находиться только в тегах `<TD>` и `<TH>`.
- Ячейки таблицы должны иметь хоть какое-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 ` `).

А чем может быть содержимое ячеек таблицы? О-о-о, тут простор для Web-дизайнерской фантазии ничем не ограничен. Текст, в том числе разбитый на абзацы или фиксированного формата, гиперссылки, графические изображения и даже другие таблицы — все это можно поместить в ячейки. Чем и пользуются.

Все Web-обозреватели выводят таблицу согласно следующим правилам:

- между границами отдельных ячеек делается небольшой отступ;
- между содержимым каждой ячейки и ее границей будет сделан небольшой отступ;
- таблица выравнивается по левому краю;
- размеры таблицы делаются такими, чтобы полностью вместить ее содержимое;
- текст ячеек выравнивается по левому краю и по верху;
- текст ячеек заголовка выравнивается по центру по горизонтали и по вертикали и набирается полужирным шрифтом;
- рамки вокруг самой таблицы и вокруг отдельных ее ячеек не рисуются.

Это поведение по умолчанию можно изменить с помощью стилей CSS, о которых пойдет разговор в *главе 3*.

Способ формирования таблиц, предлагаемый HTML, весьма прост и нагляден. Единственный его недостаток — некоторая громоздкость получающегося HTML-кода.

Название и секции таблицы

Стандарт HTML преподносит Web-дизайнерам все новые и новые сюрпризы, полезные и не очень. Давайте рассмотрим два таких сюрприза, связанные с таблицами.

Прежде всего, мы имеем возможность вставить в таблицу *название*. Это название выводится в верхней части таблицы, выше самой первой строки, и выравнивается по центру.

Чтобы вставить в таблицу название, используется парный тег <CAPTION>, который помещается прямо в тег <TABLE>. Текст названия вставляется внутрь тега <CAPTION>.

```
<TABLE>
  <CAPTION>Простая таблица</CAPTION>
  . . .
</TABLE>
```

Запомним, что тег <CAPTION> может находиться только внутри тега <TABLE>.

А что второй сюрприз? А вот он не столь полезен. Стандарт HTML позволяет логически разбить таблицу на три *секции*:

- *секцию заголовка*, содержащую "шапку" таблицы;
- *секцию тела*, содержащую строки с обычными данными;
- *секцию "поддона"*, содержащую итоговые данные и различные примечания.

Секция заголовка задается с помощью парного тега <THEAD> и "охватывает" строки, входящие в "шапку" таблицы. (Эти строки обычно содержат ячейки заголовка.) Секции тела и "поддона" задаются, соответственно, с помощью парных тегов <TBODY> и <TFOOT>.

Вот пример HTML-кода таблицы, разбитой на секции:

```
<TABLE>
  <THEAD>
    <TR>
      <TH>Заголовок столбца № 1</TH>
      <TH>Заголовок столбца № 2</TH>
    </TR>
  </THEAD>
```

```
  <TBODY>
    <TR>
      <TD>Ячейка № 2.1</TD>
      <TD>Ячейка № 2.2</TD>
    </TR>
  </TBODY>
  <TFOOT>
    <TR>
      <TD>Итог по столбцу № 1</TD>
      <TD>Итог по столбцу № 2</TD>
    </TR>
  </TFOOT>
</TABLE>
```

Все эти три тега могут встречаться только внутри тега <TABLE> и должны содержать только теги <TR>.

Секции таблицы в настоящее время не поддерживаются ни одним Web-обозревателем, однако могут обрабатываться специальными программами, теми же системами чтения с экрана. Так что создавать в таблицах секции или нет — решать нам самим. (Хотя абсолютное большинство Web-дизайнеров этого не делает.)

> ***Внимание!***
>
> Даже если мы и не разбили таблицу на секции, Web-обозреватель все равно считает, что она содержит секцию тела. С этой особенностью мы столкнемся в *главе 5*.

Объединение ячеек таблиц

Самое время поговорить об одной интересной особенности языка HTML. Это так называемое *объединение ячеек* таблиц. Которое лучше всего изучить на конкретном примере.

Давайте создадим простую таблицу, HTML-код которой приведен далее.

```
<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>
```

Если этот фрагмент кода вставить в Web-страницу, в окне Web-обозревателя она будет показана, как представлено на рис. 2.5.

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

Рис. 2.5. Изначальная таблица, чьи ячейки подвергнутся объединению

<table>
<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>
<tr><td>11</td><td colspan="4">12+13+14+15</td></tr>
<tr><td>16</td><td>17</td><td>18</td><td>19</td><td>20</td></tr>
</table>

Рис. 2.6. Таблица, показанная на рис. 2.5, после объединения некоторых ячеек (объединенные ячейки обозначены сложением их номеров)

А теперь рассмотрим вот такую таблицу — см. рис. 2.6.

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 — выполняет объединение ячеек по горизонтали, второй — ROWSPAN — по вертикали.

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, в тег <TD> (<TH>) которой мы вписываем этот атрибут.
3. Удалить теги <TD> (<TH>) остальных объединяемых ячеек данной строки.

Давайте объединим ячейки 2 и 3 таблицы, чей HTML-код представлен ранее. Вот исправленный фрагмент кода, создающий первую строку этой таблицы:

```
<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.

Выполнить объединение ячеек 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, в тег <TD> (<TH>) которой мы вписываем этот атрибут.
4. Просмотреть последующие строки и удалить из них теги <TD> (<TH>) остальных объединяемых ячеек.

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

```
<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 ячейку 6, поскольку она объединилась с ячейкой 1.

Сейчас трудно встретить таблицу, в которой бы не было выполнено объединение ячеек. А уж в специализированных таблицах (например, таблицах разметки, используемой для правильного размещения отдельных частей Web-страницы) объединение используется сплошь и рядом.

Реализация всплывающих подсказок

Всем нам, много работающим с Windows-программами, знакомы так называемые *всплывающие подсказки* — небольшие окна, появляющиеся на экране при наведении курсора мыши на элемент управления и содержащие его краткое описание. HTML предлагает средства для создания точно таких же подсказок для элементов страницы. Создаются они очень просто.

Каждый видимый тег HTML поддерживает необязательный атрибут `TITLE`, в качестве значения которого указывается текст, который появится во всплывающей подсказке.

Небольшой пример:

```
<H1 TITLE="Это заголовок">Пример Web-страницы</H1>
```

При наведении курсора мыши на заголовок "Пример Web-страницы" на экране появится всплывающая подсказка с текстом "Это заголовок" (рис. 2.7).

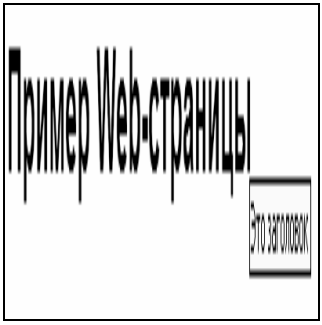

Рис. 2.7. Всплывающая подсказка

Естественно, в качестве текста всплывающей подсказки следует задавать что-то более вразумительное. В конце концов, цель такой подсказки — проинформировать посетителя о назначении данного элемента страницы, не так ли?

Использование атрибута `TITLE` — самый простой способ создать всплывающую подсказку для элемента страницы. Есть и другой способ — использовать Web-сценарии, о которых пойдет речь в *части 2* этой книги, — но он заметно сложнее.

> ***НА ЗАМЕТКУ***
>
> Для графических изображений в качестве текста всплывающей подсказки будет использоваться текст замены, заданный атрибутом `ALT`. Подробнее об этом см. соответствующий параграф данной главы.

Служебные теги HTML

Настала пора поговорить о *служебных тегах* HTML, используемых не для создания ее содержимого, а для указания сведений о самой странице и правильного ее оформления. Этих тегов не очень много.

Теги каркаса

Самые важные из служебных тегов — это те, что формируют сам "костяк" Web-страницы. Они так и называются — *теги каркаса*.

Стандарт HTML предусматривает три тега каркаса, которые, собственно, нам уже знакомы. Все они являются невидимыми.

Парный тег <HTML> используется для размещения HTML-кода страницы. Это дополнительное указание Web-обозревателю, что данный текстовый файл содержит Web-страницу и что ее нужно обработать как Web-страницу. (Основным указанием служит расширение данного файла — htm[l].) Весь HTML-код должен помещаться внутри этого тега.

Парный тег <HEAD> оформляет секцию заголовка Web-страницы. Внутри него должны размещаться служебные теги, несущие информацию о данной странице, — эту информацию Web-обозреватель использует для ее правильного вывода или какой-то дополнительной обработки.

Кстати, служебные теги, несущие информацию, которая не выводится на экран и описывает саму страницу, часто называют *метатегами*. Таких метатегов стандарт HTML предусматривает довольно много, но мы рассмотрим только важнейшие их них.

Парный тег <BODY> оформляет секцию тела Web-страницы — собственно ее видимого содержимого. А видимое содержимое страницы — это все, чем мы занимались в большей части этой главы.

Запомним несколько правил, которым нужно следовать при создании Web-страниц.

- Все служебные теги, несущие информацию о странице (то есть метатеги), должны помещаться в теге <HEAD> (секции заголовка).
- Все теги, несущие видимое содержимое страницы, должны помещаться в теге <BODY> (секции тела).
- Секция заголовка должна предшествовать секции тела.
- И секция заголовка, и секция тега должны находиться в теге <HTML>.

Вот, собственно, и все, что начинающему Web-программисту нужно знать о тегах каркаса. Теперь давайте рассмотрим два важнейших метатега, которые рекомендуется использовать во всех страницах.

Название Web-страницы

Как мы уже знаем, название страницы выводится в заголовке окна Web-обозревателя; также оно помещается в его *историю* (список уже посещенных

страниц). Это название задается парным тегом <TITLE> и помещается в секции заголовка страницы.

```
<HTML>
  <HEAD>
    <TITLE>Некая Web-страница</TITLE>
  </HEAD>
 . . .
```

Название страницы рекомендуется указывать всегда — так посетитель сразу сможет узнать, что за страница открыта в минимизированном окне Web-обозревателя, и нормально "путешествовать" по его истории. Более того: сейчас страница без названия воспринимается как плохой стиль Web-дизайна.

Задание кодировки страницы

Второй метатег, который мы рассмотрим в этой книге, задает очень важный параметр — кодировку текста, которым набрана Web-страница. Если его не задать, возможна ситуация, когда вместо нормального текста Web-обозреватель выводит набор непонятных закорючек. Сейчас такая ситуация встречается все реже, но еще несколько лет назад нечитающиеся страницы попадались частенько.

Вероятно, вы знаете, что каждый символ, который вводится с клавиатуры и отображается на экране, имеет уникальный номер, называемый кодом символа (об этом мы уже говорили в начале главы). Совокупность таких кодов вместе с описанием, какой код какому символу соответствует, образует *кодировку*. Каждая кодировка имеет свое общепринятое название, например 1251 или KOI8-R.

Поскольку любой язык использует свой набор символов, для каждого языка кодировки, как правило, различны. (Исключение — западноевропейские и центральноевропейские языки, использующие одну кодировку для всех.) Но на этом путаница с кодировками не кончается. Дело в том, что разные операционные системы используют разные кодировки. Например, западноевропейская версия Windows использует кодировку 1250, русская — 1251, американская версия MS-DOS — 437, а русская — 866. Как видите, русских кодировок уже две. А если добавить сюда еще кодировку, используемую русской версией операционной системы UNIX — KOI8-R и русской версией компьютеров Apple Macintosh — MacCyrillic, кодировок станет уже четыре. И это только главные — на памяти автора существовало еще несколько менее распространенных кириллических кодировок ("основная" кодировка ГОСТ, "болгарская", "американская", "югославская" и т. п.). Кроме того,

в последнее время появилась кодировка *Unicode*, поддерживающая ВСЕ имеющиеся на Земле языки. Настоящая тирания кодировок!..

Чем все это грозит? А вот чем. Вы, наверное, пытались открыть текстовый документ, созданный в Блокноте, в Norton Commander. Видели, что при этом получается — текст абсолютно нечитаем. А все потому, что русские кодировки 866 (MS-DOS), используемая Norton Commander, и 1251 (Windows), используемая Блокнотом, не совпадают! В них один и тот же код соответствует разным символам.

Каков же выход?

Выхода нет. Можно надеяться только на то, что какая-то из кодировок станет стандартом и постепенно вытеснит конкурентов. Пока что на роль такого (негласного) стандарта претендует 1251, хотя интернетчики старого поколения, пользующиеся UNIX-совместимыми системами, предпочитают KOI8-R. Во всяком случае, уже сейчас большинство Web-страниц, имеющихся в русском сегменте Сети, написано в кодировке 1251.

Но каким же образом задается кодировка Web-страницы? Для этого используется весьма специфический тег, формат написания которого приведен далее.

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

Видно, что обозначение кодировки указывается как часть значения атрибута `CONTENT` одинарного тега `<META>`. Другая часть этого значения (`"text/html"`) указывает тип электронного документа — Web-страница. Тег `<META>` должен присутствовать в секции заголовка страницы (теге `<HEAD>`).

Что касается самих кодировок, то самые полезные для нас представлены в табл. 2.3. Все они поддерживаются современными Web-обозревателями.

Таблица 2.3. *Некоторые кодировки, поддерживаемые современными Web-обозревателями*

Общепринятое название	Обозначение	Платформа, которая ее использует
Кириллица Windows	`windows-1251`	Русские версии Windows, начиная от версии 3.0
KOI8-R	`koi8-r`	Русские версии UNIX-совместимых систем
KOI8-U	`koi8-ru`	Украинские версии UNIX-совместимых систем
Кириллица DOS	`cp866`	Русская версия MS-DOS

Таблица 2.3 *(окончание)*

Общепринятое название	Обозначение	Платформа, которая ее использует
Кириллица ISO	`iso-8859-5`	Редко используемая русская кодировка; автору не встречалось ни одной страницы, написанной на ней
Западноевропейская	`iso-8859-1`	Используется большинством западноевропейских языков
Центральноевропейская Windows	`windows-1250`	Версии Windows для языков Центральной Европы
Центральноевропейская DOS	`ibm852`	Версии MS-DOS для языков Центральной Европы
Unicode UTF-8	`utf-8`	Разновидность Unicode, предназначенная специально для Web-страниц

Уже говорилось, что сейчас большинство Web-страниц хранятся в кириллической кодировке Windows. Но может случиться так, что какой-либо Web-сервер (точнее, его администратор) может потребовать, чтобы публикуемые на нем Web-страницы были представлены в определенной кодировке, например в KOI8-R. В этом случае выбора у нас нет — мы должны будем выбрать ту кодировку, которую требует Web-сервер.

Если же мы собираемся создавать Web-страницы на языках Западной или Центральной Европы, выбора у нас нет — только западноевропейская или центрально европейская кодировка соответственно. Также сейчас набирает популярность кодировка Unicode UTF-8; сохраненных в ней страниц становится все больше и больше, но, в основном, в англоязычной части Интернета.

Пролог

Пролог Web-страницы также относится к служебным тегам и задает разновидность языка HTML, на которой написана Web-страница. Исходя из пролога, Web-обозреватель обрабатывает код страницы тем или иным образом.

В настоящее время существуют две разновидности языка HTML. Первая из них называется *строгой*; такой HTML-код соблюдает все соглашения, предписанные стандартом HTML 4.01. Вторая разновидность HTML — *переходная*; "переходной" HTML-код написан с использованием некоторых тегов, присутствовавших в предыдущих версиях HTML, но в версии 4.01 объявленных устаревшими.

Для задания пролога используется одинарный тег <!DOCTYPE>. Этот тег примечателен тем, что предшествует любому HTML-коду, даже тегу <HTML>:

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<HTML>
  <HEAD>
  . . .
```

Строгую разновидность языка HTML задает пролог следующего вида:

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN"
"http://www.w3.org/TR/html4/strict.dtd">
```

А переходную — пролог

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
```

Да, тот самый, что приведен в примере ранее.

В принципе, особой разницы, какой пролог (и какую разновидность языка HTML) указывать в Web-странице, нет. Автор этой книги всегда использует переходную разновидность и пока что не имел проблем с отображением страниц в Web-обозревателях. Но вполне возможно, что в некоторых случаях придется указать строгую разновидность HTML.

Одно не подлежит сомнению — пролог следует указывать всегда. Иначе возможны проблемы при выводе страниц на экран, особенно если в страницах активно используются стили CSS.

> ***НА ЗАМЕТКУ***
>
> Хотя в самых простых страницах пролог можно и не указывать. Так, в большинстве страниц, приведенных в этой книге, пролог не указан, и, тем не менее, они работают.

Комментарии

Часто бывает нужно сохранить прямо в HTML-коде Web-страницы какие-то пометки для себя или своих коллег — Web-дизайнеров. Специально для таких случаев стандарт HTML предусматривает так называемые *комментарии*, которые могут содержать любой текст и никак не обрабатываются Web-обозревателем.

Комментарии HTML вставляются в код с помощью парного тега вида:

```
<!--
  <текст комментария>
-->
```

Например:

```
<H1>Часть I</H1>
<!-- Не забыть вписать сюда текст первой части -->
<H1>Часть II</H1>
```

Фреймы

Последнее, что мы рассмотрим в этой главе, — фреймы. Самая противоречивая возможность, предлагаемая HTML, до сих пор, кстати, не включенная в стандарт этого языка, вызвавшая в свое время настоящую революцию в Web-дизайне, а ныне благополучно забываемая.

Что такое фреймы

Дать определение фрейма проще всего на примере. Представим себе, что окно Web-обозревателя разбито на несколько частей, этаких "форточек", и в каждой "форточке" отображается своя Web-страница. То есть в этом случае в одном окне Web-обозревателя будет выводиться не одна страница, а сразу несколько. Причем "форточки" эти будут вести себя как независимые окна Web-обозревателя; так, если их содержимое будет слишком велико, чтобы поместиться в "форточку", у них появятся свои полосы прокрутки.

Как это сделать? Не так уж и сложно — HTML предусматривает все нужные теги и правила.

Сначала Web-дизайнер создает особую Web-страницу, в HTML-коде которой и задаются параметры "форточек", в частности, их размеры и интернет-адреса страниц, которые будут в них загружены. Ну и, разумеется, нужно создать все страницы, которые будут загружены в определенные таким образом "форточки"; они ничем не отличаются от страниц, что мы создавали ранее.

Предположим, что мы создали страницу с набором таких вот "форточек" и несколько обычных страниц. Одна из таких страниц содержит набор гиперссылок для "путешествия" по сайту, другая — заголовок сайта (текстовый или графический), третья — сведения об авторских правах разработчиков, а четвертая — какой-либо вводный текст. Если мы откроем страницу с набором "форточек" в Web-обозревателе, то увидим нечто похожее на представленное на рис. 2.8.

Теперь пусть посетитель нашего сайта решит перейти на другую страницу. Он найдет в одной из "форточек" набор гиперссылок и щелкнет нужную. Web-обозреватель прочитает из тега <A>, формирующего эту гиперссылку, интернет-адрес нужной страницы и — внимание! — ее цель. А цель этой

гиперссылки указывает Web-обозревателю, что страница эта должна быть загружена в "форточку", отведенную под вводный текст. Что Web-обозреватель и сделает.

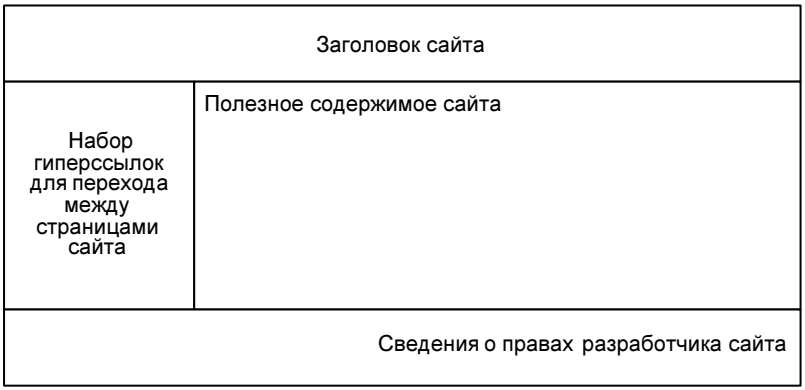

Рис. 2.8. Web-страница — классический набор из четырех фреймов

Что это нам дает? Давайте посмотрим. Обычно на всех страницах сайта помещается множество повторяющихся элементов: заголовок сайта, набор гиперссылок, сведения об авторских правах и пр. Зачастую такие повторяющиеся элементы занимают больше места, чем основное содержимое страницы. Понятно, что из-за этого страницы становятся очень большими и, следовательно, медленно загружаются, особенно по медленным каналам связи.

А так мы можем эти самые повторяющиеся элементы перенести из основных страниц во вспомогательные и поместить последние в отдельные "форточки". Пусть они постоянно присутствуют на экране, не обновляясь при переходах между страницами. А наши основные странички, лишенные "балласта", в результате сильно сбавят в размерах и будут грузиться быстрее.

Но пора, наконец, назвать вещи своими именами. "Форточки", на которые делится окно Web-обозревателя, — это и есть *фреймы* (frame, кадр). А Web-страницу, определяющую набор таких частей, назовем *набором фреймов* (frameset).

Фреймы — интересная и полезная вещь, которой от роду уже лет десять. Когда-то фреймы на Web-страницах были рядовым явлением. Их применяли так часто, что сейчас среди снобов от Web-дизайна они считаются дурным тоном и последние лет семь практически не применяются в новых сайтах. А зря!

Создание набора фреймов

Создание страницы с набором фреймов включает в себя три этапа.

1. Создание набора фреймов.
2. Создание отдельных фреймов.
3. Создание страниц — содержимого отдельных фреймов.

Давайте изготовим страницу, представляющую собой классический набор фреймов, показанный на рис. 2.8. И в процессе ее создания изучим все необходимые теги.

Сначала нам нужно создать "костяк" страницы с набором фреймов. Это будет вполне обычная страница с одним небольшим отличием от тех, что мы делали ранее.

```
<HTML>
  <HEAD>
    <TITLE>Набор фреймов</TITLE>
  </HEAD>
  <FRAMESET>
  </FRAMESET>
</HTML>
```

Исключение — это присутствие парного тега `<FRAMESET>` (на примере он выделен полужирным шрифтом) вместо тега `<BODY>`. Именно тег `<FRAMESET>`, присутствующий в странице вместо секции ее тела, указывает Web-обозревателю, что мы хотим создать набор фреймов.

Теперь давайте посмотрим на рис. 2.8 и посчитаем фреймы, которые нам нужно создать, в порядке сверху вниз. Их три:

- заголовок сайта;
- набор гиперссылок и полезное содержимое (пока что посчитаем их за один фрейм, а почему — узнаем чуть позже);
- сведения о правах.

Итого их три, и все они расположены по вертикали. Но Web-обозреватель об этом еще не знает, так что нам придется дать ему на этот счет соответствующие инструкции.

Для указания Web-обозревателю количества фреймов в наборе, их вертикального расположения и размеров (то есть значения высоты каждого из них) служит атрибут `ROWS` тега `<FRAMESET>`. В качестве значения этого атри-

бута указывается набор значений размеров фреймов, разделенных запятыми. Вот так:

```
<FRAMESET ROWS="100,400,100">
</FRAMESET>
```

Здесь мы указываем Web-обозревателю, что хотим иметь три расположенных вертикально фрейма со значениями высоты 100, 400 и 100 пикселов.

Мы можем указать значения размера фреймов и в процентах относительно текущей высоты страницы:

```
<FRAMESET ROWS="10%,80%,10%">
</FRAMESET>
```

Здесь средний фрейм займет 80% высоты страницы, а оба крайних — по 10%.

Мы также можем задать размер среднего фрейма вот таким образом:

```
<FRAMESET ROWS="100,*,100">
</FRAMESET>
```

Знак `*` ("звездочка") указывает Web-обозревателю, что под средний фрейм нужно отвести все свободное пространство, не занятое другими фреймами.

Можно также смешивать значения размеров, заданные в пикселах и в процентах:

```
<FRAMESET ROWS="80,*,10%">
</FRAMESET>
```

Здесь первый фрейм будет иметь высоту 80 пикселов, третий — 10% свободного пространства, не занятого первым фреймом, а второй фрейм займет все место, не занятое первым и третьим фреймами.

Поскольку первый и третий фреймы, в отличие от второго, у нас не будут "дробиться" на другие фреймы, мы можем сразу же их создать. Для создания самого фрейма, включенного в набор, мы используем одинарный тег `<FRAME>`. Вот так:

```
<FRAMESET ROWS="100,*,100">
  <FRAME>
  <FRAME>
</FRAMESET>
```

Теперь давайте остановимся и немного подумаем. Как следует из описанного ранее, один набор фреймов может содержать только фреймы, расположенные либо по горизонтали, либо по вертикали. Наш набор фреймов содержит фреймы, расположенные по вертикали. А еще не созданные фреймы с набором гиперссылок и полезным содержимым расположены по горизонтали. Что делать?

Вставить в только что созданный набор фреймов другой — описывающий два недостающих фрейма! Так мы получим *сложный набор фреймов*.

```
<FRAMESET ROWS="100,*,100">
  <FRAME>
  <FRAMESET>
  </FRAMESET>
  <FRAME>
</FRAMESET>
```

Здесь тег <FRAMESET>, создающий вложенный набор фреймов, выделен полужирным шрифтом. Зададим количество присутствующих в нем фреймов, их горизонтальное расположение и значения их размера (ширины). Для этого служит атрибут COLS тега <FRAMESET>, аналогичный атрибуту ROWS.

```
<FRAMESET ROWS="100,*,100">
  <FRAME>
  <FRAMESET COLS="100,*">
  </FRAMESET>
  <FRAME>
</FRAMESET>
```

И создадим во вложенном наборе два фрейма — набор гиперссылок и полезное содержимое.

```
<FRAMESET ROWS="100,*,100">
  <FRAME>
  <FRAMESET COLS="100,*">
    <FRAME>
    <FRAME>
  </FRAMESET>
  <FRAME>
</FRAMESET>
```

И последний этап — задание параметров фреймов, в частности, интернет-адресов страниц, которые будут загружены в них изначально. Для этого служит обязательный атрибут SRC тега <FRAME>.

```
<FRAMESET ROWS="100,*,100">
  <FRAME SRC="header.html" >
  <FRAMESET COLS="100,*">
    <FRAME SRC="links.html">
    <FRAME SRC="main.html" NAME="mainframe">
  </FRAMESET>
  <FRAME SRC="copyright.html">
</FRAMESET>
```

Набор фреймов готов. Но что это за странный атрибут NAME присутствует в теге <FRAME>, создающем фрейм с полезным содержимым? Сейчас мы это выясним.

Использование цели гиперссылки для указания фрейма

Фактически нам осталось только создать страницы, представляющие содержимое отдельных фреймов. Их мы создадим без труда — это обычные Web-страницы, не отличающиеся от тех, что мы создали ранее. Затруднения возникнут только со страницей links.html, содержащей набор гиперссылок.

По умолчанию, если посетитель щелкнет на гиперссылке, Web-обозреватель загрузит страницу, на которую указывает интернет-адрес этой гиперссылки, и загрузит ее в тот фрейм, в котором находится страница с данной гиперссылкой. Но нам-то нужно загрузить целевую страницу в другом фрейме текущей страницы — там, где должно отображаться полезное содержимое! Как это сделать?

Ранее мы познакомились с такой возможностью гиперссылок, как цель. Мы знаем, что эта цель позволяет указать, в каком окне Web-обозревателя будет открыта целевая страница — текущем или новом. Точно так же — с помощью цели гиперссылки — мы можем указать и фрейм, в котором должна быть открыта целевая страница. Для этого следует указать в качестве значения атрибута TARGET тега <A> имя нужного фрейма.

Имя фрейма указывается с помощью необязательного атрибута NAME тега <FRAME>. Посмотрим на приведенный ранее HTML-код, создающий наш набор фреймов, — мы уже указали имя фрейма полезного содержимого (mainframe):

```
<FRAME SRC="main.html" NAME="mainframe">
```

Так что нам остается только указать имя этого фрейма в качестве цели всех гиперссылок на странице links.html, указывающих на страницы нашего сайта:

```
<A HREF="news.html" TARGET="mainframe">Новости</A>
<A HREF="articles.html" TARGET="mainframe">Статьи</A>
<A HREF="files.html" TARGET="mainframe">Файлы</A>
. . .
```

Вот и все.

А если в качестве значения атрибута TARGET указать "_top", целевая страница будет загружена прямо во все окно Web-обозревателя и заменит набор фреймов. Такую цель следует указывать для гиперссылок, ведущих на другие сайты.

Для фреймов, содержимое которых всегда останется неизменным (в нашем случае — это фреймы с заголовком сайта, набором гиперссылок и сведениями об авторских правах), имя можно не указывать. Все равно оно никогда не будет использовано.

Дополнительные возможности фреймов и наборов фреймов

Теги `<FRAMESET>` и `<FRAME>` поддерживают довольно много необязательных атрибутов. Давайте их рассмотрим.

Начнем с тега `<FRAMESET>`, создающего набор фреймов. Полезных для нас необязательных атрибута здесь три.

Атрибут `FRAMEBORDER` позволяет включить или отключить рисование рамки вдоль границ отдельных фреймов набора. Значение `"yes"` включает рисование рамки, а значение `"no"` — отключает. По умолчанию рамка рисуется.

Атрибут `BORDER` задает толщину рамки в пикселах, если, конечно, вывод рамки включен. Значение по умолчанию зависит от Web-обозревателя, поэтому толщину рамки лучше всегда указывать явно.

Для задания параметров рамки следует указывать оба этих тега. Например, для указания толщины рамки, равной 5 пикселам, следует использовать такой код:

```
<FRAMESET . . . FRAMEBORDER="yes" BORDER="5">
```

Если нужно убрать рамку между фреймами, воспользуйтесь следующим кодом:

```
<FRAMESET . . . FRAMEBORDER="no" BORDER="0">
```

> ***НА ЗАМЕТКУ***
>
> Фреймы с видимыми рамками — плохой стиль Web-дизайна. Лучше их убрать или использовать совсем тонкие рамки — толщиной в 1 пиксел.

Атрибут `BORDERCOLOR` задает цвет рамки между фреймами. Значение цвета по умолчанию зависит от Web-обозревателя.

Цвет какого-либо элемента страницы в HTML задается в формате *RGB*. RGB расшифровывается как "Red, Green, Blue" — "красный, зеленый, синий". А сам формат задания цвета выглядит так — `#RRGGBB`, где `RR` — это шестнадцатеричное число от 0 до FF, задающее долю в окончательном цвете красной составляющей, `GG` — зеленой, а `BB` — синей. Пример задания цвета в формате RGB — `#336699` (это тускло-голубой цвет).

На заметку

Вряд ли вы помните RGB-код вашего любимого цвета. Поэтому лучше поискать в Интернете утилиты, позволяющие выбрать цвет и узнать его код. Также можно использовать набор цветов, созданный профессиональными дизайнерами. Где найти такой набор, указано в *заключении*.

С тегом <FRAMESET> все. На очереди — тег <FRAME>, создающий отдельный фрейм набора.

Атрибут SCROLLING запрещает или разрешает отображение полос прокрутки во фрейме. Значение "auto" задает их отображение лишь в том случае, если содержимое не помещается во фрейме; это поведение по умолчанию. Значение "yes" задает отображение полос прокрутки в любом случае, даже если они и не нужны, а значение "no" отключает их совсем.

Внимание!

Значение "no" атрибута SCROLLING следует использовать с осторожностью: кто знает, какими увидит посетитель наши страницы и увидит ли их вообще.

По умолчанию посетитель может изменить размеры фреймов, просто перетаскивая мышью рамки между ними. Атрибут без значения NORESIZE отключает эту возможность.

Атрибуты FRAMEBORDER и BORDERCOLOR работают аналогично одноименным атрибутам тега <FRAMESET>. При этом заданные с их помощью параметры отдельного фрейма перекрывают параметры, заданные аналогичными атрибутами для набора фреймов, в котором присутствует этот фрейм.

На заметку

Вообще-то, тег <FRAME> поддерживает также атрибуты MARGINWIDTH и MARGINHEIGHT, устанавливающие расстояние между границей фрейма и его содержимым соответственно по горизонтали и вертикали в пикселах. Но, судя по всему, современные Web-обозреватели игнорируют эти атрибуты.

Вот и все о фреймах.

Будущее HTML

Язык HTML существует уже два десятка лет. Срок немалый, особенно если вспомнить, сколько лет в среднем "живут" компьютерные технологии и стандарты. Что же ожидает его в будущем?

Судя по всему, версия 4.01 станет последней версией языка HTML. В дальнейшем он будет постепенно заменен своим потомком — языком *XHTML* (eXtensible HypetText Markup Language, расширяемый язык гипертекстовой

разметки). Этот язык основан на популярном языке описания данных *XML* (eXtensible Markup Language, расширяемый язык разметки). Его основные отличия от HTML перечислены далее.

- Имена тегов и атрибутов набираются **только** строчными буквами. Обратите внимание — это очень важно!
- Все теги исключительно двойные. Одинарные теги отсутствуют.
- Некоторые теги, присутствовавшие в старых версиях HTML, а в версии 4.01 объявленные как устаревшие, но еще поддерживаемые, полностью исключены.
- В каждой Web-странице обязательно должен присутствовать пролог. Собственно, именно пролог указывает Web-обозревателю, что страница написана на XHTML.

Скоро ли наступит эра XHTML? Вряд ли. Старый добрый HTML поцарствует еще не один год. Поэтому не стоит торопиться заучивать новые теги и переписывать уже написанные Web-страницы. Займитесь более полезными делами — на них-то времени всегда не хватает…

Что дальше?

Вот мы и познакомились с языком HTML, с помощью которого создаются Web-страницы. Мы рассмотрели многие его теги, но не все; о некоторых специфических тегах и предоставляемых ими возможностях мы узнаем в последующих главах.

Неоднократно в этой главе мы упоминали о стилях CSS. Что это такое? Давайте выясним — именно стилям посвящена вся *глава 3*.

Глава 3

Язык CSS. Каскадные таблицы стилей

В предыдущей главе мы вели разговор о языке HTML, используемом для создания Web-страниц, и активно практиковались в их создании. В этой главе мы продолжим изучение интернет-технологий, что применяются в Web-дизайне. На этот раз займемся каскадными таблицами стилей CSS.

Что это такое и зачем они нужны? Давайте разберемся.

Возможности HTML по оформлению текста, скажем прямо, не впечатляют. Всего-то и можем мы, что выделить текст полужирным шрифтом (тег `<STRONG>`) или курсивом (тег `<EM>`). Негусто, если сравнивать с современными текстовыми процессорами и издательскими системами…

Что же делать? Отказаться от всех красивостей? Как говорится в популярной рекламе, есть способ лучше. Мы можем использовать *каскадные таблицы стилей* (или просто *таблицы стилей*). По-английски они называются *CSS* (Cascading Style Sheets).

Каскадные таблицы стилей не относятся к языку HTML. Они пишутся на особом языке, который так и называется — CSS. В настоящее время существуют две версии этого языка: CSS1 и CSS2; идет работа над CSS3. Все современные Web-обозреватели полностью поддерживают CSS1 и частично или полностью — CSS2. Как правило, в большинстве случаев этого хватает.

Введение в каскадные таблицы стилей

Начнем мы с того, что разберем основные принципы создания таблиц стилей CSS. Более подробное рассмотрение предлагаемых языком CSS возможностей мы отложим на потом.

Создание таблиц стилей

Пусть, например, нам нужно выделить какой-то фрагмент текста красным шрифтом, дабы привлечь к нему внимание. Для этого мы сначала напишем такую таблицу стилей:

```
<STYLE>
  .redtext { color: #FF0000 }
</STYLE>
```

Таблица стилей формируется с помощью парного тега `<STYLE>` и помещается в секцию заголовка Web-страницы (внутрь тега `<HEAD>`). Внутри тега `<STYLE>` помещаются определения отдельных *стилей*, составляющих эту таблицу.

Каждый стиль, присутствующий в таблице стилей, обязан иметь уникальное имя. По этому имени мы сможем впоследствии на него сослаться. Правила именования стилей различаются в зависимости от того, к какой разновидности принадлежит данный стиль; мы обязательно рассмотрим эти правила.

Наш единственный стиль принадлежит к так называемым *стилевым классам*. Стилевой класс может быть применен (*привязан*) к любому тегу. Имя стилевого класса обязательно должно предваряться точкой — по этой точке Web-обозреватель и отличает стилевые классы от других разновидностей стилей. Наш стиль имеет имя `redtext`.

После имени стиля, через пробел, в фигурных скобках идет собственно набор параметров, задающий форматирование содержимого тега, к которому мы привяжем этот стиль. Он представляет собой список *атрибутов стиля* и их *значений*. Каждый атрибут стиля задает наименование параметра (размер шрифта, цвет текста и пр.), а значение — собственно значение параметра в принятой в CSS единице измерения. Атрибут отделяется от значения двоеточием, а отдельные пары "атрибут:значение" разделяются точкой с запятой. Причем после последней такой пары точка с запятой не ставится, иначе возможны проблемы с некоторыми Web-обозревателями.

Наш стиль содержит один-единственный атрибут — `color`, задающий цвет текста. Значение этого атрибута стиля равно `#FF0000`; этот RGB-код задает красный цвет.

Итак, стиль создан. Осталось привязать его к нужному тегу. Для этого мы применим необязательный атрибут `CLASS`, который поддерживается всеми видимыми тегами HTML:

```
<P CLASS="redtext">Это красный текст.</P>
```

Видно, что имя стилевого класса указывается в значении атрибута `CLASS`. Причем указывается уже без точки.

Что ж, понятие о стилях и таблицах стилей мы получили. Продолжим изучение стилей и их разновидностей.

Разновидности стилей

Первая разновидность стилей, которую мы знаем, — это стилевой класс. Сходна с ними вторая разновидность — *стиль-селектор*. Его определение записывается примерно так же, как определение стилевого класса:

```
#bigtext { font-size: large }
```

Имя стиля-селектора предваряется символом "решетки" (#). Еще не знакомый нам атрибут стиля `font-size` задает размер шрифта, а его значение `"large"` обозначает самый большой размер, определенный в CSS.

Стиль-селектор привязывается к тегу с помощью необязательного атрибута `ID`, также поддерживаемого практически всеми тегами:

```
<P ID="bigtext">Большой текст.</P>
```

Здесь нужно иметь в виду вот что. Значение атрибута `ID` должно быть уникальным в пределах данной Web-страницы. То есть если мы создали в ней любой тег со значением атрибута `ID`, равным `"bigtext"`, то второго тега с таким же значением данного атрибута мы создать не сможем. (Вообще-то, сможем, но Web-обозреватель не обработает его правильно.)

Стили-селекторы используются много реже, чем остальные разновидности стилей. Как правило, они применяются, если какой-то стиль нужно привязать к одному-единственному тегу на странице.

Третья разновидность стилей — *стили переопределения тега* — применяются очень часто. В отличие от стилевых классов и стилей-селекторов, которые могут быть привязаны к любому тегу, стили переопределения тега задают форматирование содержимого какого-либо определенного тега. Например:

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

Здесь мы создали стиль переопределения тега `<EM>`. После задания такого стиля любой текст, помещенный в тег `<EM>`, будет выделен полужирным зеленым шрифтом. RGB-код `#00FF00` задает зеленый цвет, атрибут стиля `font-weight` — степень "жирности" шрифта, а его значение `"bold"` — как раз полужирное начертание. Заметим, что имя стиля переопределения тега совпадает с именем тега, форматирование содержимого которого он задает, без символов < и >.

С помощью одного стиля мы можем переопределить форматирование сразу нескольких тегов. Для этого нам будет нужно просто перечислить имена нужных тегов через запятую, вот так:

```
P, H1, H2 { color: #00FF00 }
```

Здесь мы задали зеленый цвет шрифта сразу для трех тегов: <P>, <H1> и <H2>.

Мы можем задать и такой стиль переопределения тега:

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

Здесь мы задали черный цвет (RGB-код #000000) фона Web-страницы с помощью атрибута background-color.

Четвертая разновидность стилей представляет собой комбинацию трех рассмотренных нами ранее. Это *комбинированные стили*, примеры которых мы сейчас рассмотрим.

Стиль

```
H1 EM { color: #00FF00 }
```

будет применен к тексту, находящемуся в теге <EM>, вложенном, в свою очередь, в тег <H1>.

```
<H1><EM>Этот текст</EM> станет зеленым.</H1>
<H1>А этот не станет.</H1>
<P><EM>Этот</EM> - тоже.</P>
```

А такой стиль

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

будет применен только к тексту, помещенному внутрь тега <P> со значением атрибута CLASS, равным "mini". Значение "smaller" атрибута font-size задает размер шрифта, несколько меньший, чем размер шрифта родительского тега.

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

Пятая и последняя разновидность стилей — *встроенные стили*. Они помещаются прямо внутри тегов, к которым должны быть привязаны. Делается это с помощью необязательного атрибута STYLE, поддерживаемого, как и CLASS, практически всеми тегами.

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

Данный абзац будет выведен уменьшенным курсивом — значение "italic" атрибута стиля font-style задает курсивное начертание.

Вот и все разновидности стилей, поддерживаемые CSS. Настала пора узнать о разновидностях таблиц стилей.

Разновидности таблиц стилей

Определения стилей, как мы уже знаем, записываются в таблице стилей. Эти таблицы могут быть как помещены в HTML-код Web-страниц, так и хранить-

ся в отдельных файлах. Исходя из этого, они делятся на две разновидности, каждая из которых находит свое применение.

Уже знакомая нам *внутренняя* таблица стилей записывается прямо в HTML-код использующей ее страницы. Она заключается в парный тег <STYLE>, который помещается в секцию заголовка.

Вот пример внутренней таблицы стилей (ее CSS-код выделен полужирным шрифтом):

```
<HEAD>
  . . .
  <STYLE>
    .redtext { color: #FF0000 }
    #bigtext { font-size: large }
    EM { color: #00FF00;
         font-weight: bold }
    H1 EM { color: #0000FF }
  </STYLE>
  . . .
</HEAD>
```

Во внутреннюю таблицу стилей следует помещать стили, используемые только данной страницей, причем в нескольких ее тегах. Стиль, используемый в одном-единственном теге, лучше оформить в виде встроенного стиля.

Внешняя таблица стилей хранится отдельно от Web-страниц, в файле с "говорящим" расширением css. Особых отличий от внутренней таблицы стилей она не имеет, как видно из приводимого далее примера:

```
.redtext { color: #FF0000 }
#bigtext { font-size: large }
EM { color: #00FF00;
     font-weight: bold }
H1 EM { color: #0000FF }
```

Пожалуй, единственное ее отличие от внутренней таблицы стилей — отсутствие тега <STYLE>. В данном случае он не нужен, так как Web-обозреватель и так знает, что в данном файле хранится именно таблица стилей, а не что-то иное.

Внешняя таблица стилей привязывается к Web-странице с помощью особого одинарного тега <LINK>. Этот тег помещается в секцию заголовка Web-страницы.

```
<HEAD>
  . . .
  <LINK REL="stylesheet" HREF="<интернет-адрес файла таблицы стилей>"
```

```
  ↳TYPE="text/css">
  . . .
</HEAD>
```

Видно, что интернет-адрес файла с внешней таблицей стилей указывается в качестве значения обязательного атрибута `HREF`. Остальные атрибуты этого тега говорят Web-обозревателю, что тег ссылается как раз на таблицу стилей.

Во внешнюю таблицу стилей стоит помещать только стили, используемые во многих Web-страницах. Как правило, для всего сайта создается одна *глобальная* таблица стилей, которая используется всеми его страницами. Благодаря этому, мы можем изменять оформление всего сайта, просто отредактировав эту таблицу стилей.

В одной и той же Web-странице могут применяться сразу несколько таблиц стилей: несколько внешних и внутренняя.

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

Правила каскадности и приоритет стилей

Но что случится, если какой-то стиль будет определен одновременно в нескольких таблицах стилей? Давайте разберемся с этим.

Пусть мы создали вот такую внешнюю таблицу стилей:

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

После этого мы создали Web-страницу, содержащую внутреннюю таблицу стилей:

```
<STYLE>
  .redtext { color: #00FF00 }
  EM { font-size: smaller }
</STYLE>
```

А в самой странице содержится вот такой фрагмент HTML-кода:

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

Хорошо видно, что определения стилей как бы накладываются друг на друга. Так, во второй строке кода к тегу `<P>` привязаны и стиль-селектор `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 ранее HTML-кода (абзац с курсивным текстом "Это курсив."). Сначала Web-обозреватель загрузит и обработает внешнюю таблицу стилей, после чего сохранит ее в памяти. После этого он обработает внутреннюю таблицу стилей и добавит все содержащиеся в ней определения стилей к уже хранящимся во внешней таблице. Это значит, что стили переопределения тега `<EM>`, заданные в разных таблицах стилей, будут сложены, и результирующий стиль станет таким:

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

Именно он и будет применен ко всем тегам `<EM>`, что присутствуют в коде страницы.

Вторая строка кода, что содержит встроенный стиль, будет обработана так же. Web-обозреватель добавит к считанному из внешней таблицы стилей определению стиля-селектора `bigtext` определение встроенного стиля. Полученный стиль, если записать его на языке CSS, будет таким:

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

RGB-код `#FFFF00` задает желтый цвет.

И, наконец, самая трудная задача — первая строка кода (абзац "Это красный текст."). 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 во внешней таблице стиля, и заменит его тем, что задано в таблице внутренней, 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: #00FF00 }
```

И "красный текст" станет зеленым.

Здесь мы столкнулись с тем, что стили, заданные по-разному и 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*.

1. Внешняя таблица стилей, ссылка на которую (тег `<LINK>`) встречается в HTML-коде страницы позже, имеет приоритет перед той, на которую ссылаются раньше.
2. Внутренняя таблица стилей имеет приоритет перед внешними.
3. Встроенные стили имеют приоритет перед стилями, заданными в таблицах стилей.
4. Более конкретные стили имеют приоритет перед менее конкретными. Так, стилевой класс имеет приоритет перед стилем переопределения тега, поскольку стилевой класс привязывается к конкретным тегам. А комбинированный стиль по этой же причине имеет приоритет перед стилевым классом.

Что это значит? Это значит, что мы можем разделить используемые на наших страницах стили по их "близости" к конкретной странице. Так, более общие стили, используемые на всех страницах, мы можем вынести во внешнюю таблицу стилей. Если же на какой-то странице нам понадобится другой стиль, мы можем взять самый подходящий из внешней таблицы стилей и переопределить его во внутренней таблице стилей этой страницы: добавить новые атрибуты и изменить значения уже существующих. Таким образом, мы сможем уменьшить размер всех таблиц стилей — и внешних, и внутренних — до минимума.

Только чтобы понять все правила каскадности, "почувствовать их в руках", стоит поэкспериментировать. В конце концов, что может быть лучше хорошей практики!..

Атрибуты стилей CSS

Познакомившись со стилями CSS, давайте выясним, какие атрибуты поддерживаются Web-обозревателями на данный момент. Эти атрибуты мы сгруппируем по назначению: задающие параметры шрифта, параметры фона, параметры абзаца и пр.

Параметры шрифта

Поскольку текст на Web-страницах — всему голова, начнем мы наше знакомство с атрибутами стилей с тех из них, что задают параметры шрифта.

Атрибут `font-family` задает начертание шрифта текста.

```
font-family: "<список имен шрифтов, разделенных запятыми>";
```

Имена шрифтов задаются в виде `"Arial"` или `"Times New Roman"` (кавычки обязательны):

```
font-family: "Arial";
```

При этом подразумевается, что на клиентском компьютере такие шрифты присутствуют, иначе Web-обозреватель использует шрифт по умолчанию, и наша страница, возможно, будет выглядеть не так, как мы планировали. Впрочем, шрифты Arial и Times New Roman присутствуют на любом компьютере, работающем под управлением Windows.

Если атрибут `font-family` определен во встроенном стиле, названия шрифтов берутся в апострофы, а не кавычки.

```
<P STYLE="font-family: 'Arial'">
```

Допускается задавать несколько имен шрифтов через запятую:

```
font-family: "Verdana", "Arial";
```

В этом случае сначала на клиентском компьютере ищется первый шрифт из списка, в случае неудачного поиска — второй, потом третий и т. д.

Кроме 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"`, `"fantasy"` (декоративные шрифты) и `"monospace"` (моноширинные шрифты).

```
font-family: "Verdana", "Arial", "sans-serif";
```

Атрибут `font-size` определяет размер шрифта.

```
font-size: <размер>|xx-small|x-small|small|medium|large|x-large|
↳xx-large|larger|smaller;
```

Размер шрифта может быть задан в абсолютных и относительных величинах. Для этого используется одна из *единиц измерения*, поддерживаемая CSS, — все они перечислены в табл. 3.1.

Обозначение выбранной единицы измерения указывается после самого значения:

```
font-size: 10pt;
font-size: 1cm;
font-size: 150%;
```

Таблица 3.1. *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	%

Отметим, что все приведенные в табл. 3.1 единицы измерения используются для задания значений других атрибутов CSS.

Кроме числовых, атрибут `font-size` принимает и символьные значения. Так, значения от `"xx-small"` до `"xx-large"` задают семь предопределенных размеров шрифта. А значения `"larger"` и `"smaller"` представляют следующий размер шрифта соответственно по возрастанию и убыванию. Так, например, если для родительского тега определен шрифт размера `"medium"`, то значение `"larger"` установит для текущего элемента размер шрифта `"large"`.

Атрибут `color` задает цвет текста.

```
color: <цвет>;
```

Здесь значение цвета может быть задано как RGB-код или символьное наименование цвета (`"black"` для черного, `"white"` для белого, `"red"` для красного, `"green"` для зеленого, `"blue"` для синего и пр.).

Этот же атрибут — `color` — мы можем использовать для задания цвета горизонтальной линии HTML.

Атрибут `font-weight` устанавливает "жирность" шрифта.

```
font-weight: normal|bold|bolder|lighter|100|200|300|400|500|600|
↳700|800|900;
```

Здесь доступны семь абсолютных значений от 100 до 900, представляющих различную "жирность" шрифта, при этом обычный шрифт будет иметь "жирность" 400 (или `"normal"`), а полужирный — 700 (или `"bold"`). Значением по умолчанию является 400 (`"normal"`). Значения `"bolder"` и `"lighter"` являются

относительными и представляют следующие степени "жирности" соответственно в бóльшую и меньшую сторону.

Атрибут `font-style` задает начертание шрифта.

```
font-style: normal|italic|oblique;
```

Доступны три значения, представляющие обычный шрифт (`"normal"`, это значение по умолчанию) и курсив (`"italic"` или `"oblique"`). Последние два значения обрабатываются современными Web-обозревателями одинаково.

Атрибут `text-decoration` задает "украшение", которое будет применено к тексту: подчеркивание, надчеркивание или зачеркивание.

```
text-decoration: none|underline|overline|line-through;
```

Здесь доступны пять разных значений:

- `"none"` убирает все "украшения" (это поведение по умолчанию);
- `"underline"` подчеркивает текст;
- `"overline"` "надчеркивает" текст, то есть проводит линию над строками;
- `"line-through"` зачеркивает текст.

Атрибут `font-variant` задает, как будут выглядеть большие буквы шрифта.

```
font-variant: normal|small-caps;
```

Значение `"small-caps"` задает такое поведение шрифта, когда его строчные буквы выглядят точно так же, как прописные, просто меньшего размера. Значение по умолчанию — `"normal"`.

Атрибут `text-transform` позволяет изменить регистр символов текста.

```
text-transform: capitalize|uppercase|lowercase|none;
```

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"`, это поведение по умолчанию).

Атрибут `line-height` задает вертикальное расстояние между двумя строками (вернее, между базовыми линиями двух строк; *базовой линией* называется воображаемая линия, на которой "лежит" строка текста).

```
line-height: normal|<расстояние>;
```

Здесь допускаются абсолютные и относительные величины расстояния, указав соответствующую единицу измерения CSS (см. табл. 3.1). Если же мы ее не укажем, заданное нами значение умножается на высоту текущего шрифта и уже после этого используется. Таким образом, чтобы сделать стандартный отступ в два интервала, мы можем написать:

```
line-height: 2;
```

Значение "normal" этого атрибута устанавливает расстояния между строками по умолчанию.

Параметры фона

CSS позволяет нам задать фон для элементов страниц: абзацев, таблиц, отдельных ячеек таблиц или Web-страницы целиком. Для этого используются атрибуты, которые мы сейчас рассмотрим.

Атрибут background-color служит для задания цвета фона.

```
background-color: transparent|<цвет>;
```

Здесь можно задать либо сам цвет в виде RGB-кода или символьного наименования, либо значение "transparent", чтобы убрать фон совсем — тогда элемент страницы получит прозрачный фон. По умолчанию фон у элементов страницы отсутствует, а фон самой страницы задает Web-обозреватель.

Атрибут background-image позволяет использовать в качестве фона графическое изображение (*фоновое изображение*).

```
background-image: none|<интернет-адрес файла изображения>;
```

Здесь можно задать либо интернет-адрес файла, где хранится фоновое изображение, либо значение "none", убирающее графический фон (это поведение по умолчанию).

Если фоновое изображение меньше, чем элемент страницы (или сама страница), для которого оно задано, Web-обозреватель будет повторять это изображение, пока не "замостит" им весь элемент. Параметры этого повторения задает атрибут background-repeat.

```
background-repeat: no-repeat|repeat|repeat-x|repeat-y;
```

Здесь доступны четыре значения:

- "no-repeat" — фоновое изображение не будет повторяться никогда, в этом случае часть фона элемента страницы останется не занятой им;
- "repeat" — фоновое изображение будет повторяться по горизонтали и вертикали (поведение по умолчанию);
- "repeat-x" — фоновое изображение будет повторяться только по горизонтали;
- "repeat-y" — фоновое изображение будет повторяться только по вертикали.

Когда мы прокручиваем содержимое окна Web-обозревателя, то вместе с содержимым Web-страницы прокручивается и фоновое изображение (если оно

есть). Некоторые Web-обозреватели поддерживают одну забавную возможность: запрет прокрутки графического фона и фиксация его на месте. Фиксацией фона управляет атрибут `background-attachment`.

```
background-attachment: scroll|fixed;
```

Значение `"scroll"` заставляет Web-обозреватель прокручивать фон вместе с содержимым элемента страницы (это поведение по умолчанию). Значение `"fixed"` фиксирует фон на месте, и он не будет прокручиваться.

Также некоторые Web-обозреватели позволяют нам поместить фоновое изображение в нужное место элемента страницы. Выполняется это с помощью двух атрибутов, которые мы сейчас рассмотрим.

Атрибут `background-position-x` позволяет задать горизонтальную позицию фонового изображения.

```
background-position-x: <горизонтальная позиция>|left|center|right;
```

Здесь мы можем указать либо абсолютное или относительное (относительно ширины элемента страницы) местоположение фонового изображения, либо одно из предопределенных значений:

- `"left"` — фоновое изображение прижимается к левому краю элемента страницы;
- `"center"` — располагается по центру;
- `"right"` — прижимается к правому краю.

Значение по умолчанию — `0`, то есть фоновое изображение вплотную прижимается к левому краю элемента страницы.

Атрибут `background-position-y` позволяет задать вертикальную позицию фонового изображения.

```
background-position-y: <вертикальная позиция>|top|center|botton;
```

Здесь мы также можем указать либо абсолютное или относительное (относительно высоты элемента страницы) местоположение фонового изображения, либо одно из предопределенных значений:

- `"top"` — фоновое изображение прижимается к верхнему краю элемента страницы;
- `"center"` — располагается по центру;
- `"bottom"` — прижимается к нижнему краю.

Значение по умолчанию — `0`, то есть фоновое изображение вплотную прижимается к верхнему краю элемента страницы.

Параметры абзаца

Атрибуты, что мы рассмотрим здесь, применяются для форматирования абзацев текста, будь то обычные абзацы, заголовки или цитаты, созданные с помощью тега <BLOCKQUOTE>. Также они применимы для некоторых других элементов страницы: горизонтальных линий, изображений и пр.

Атрибут word-spacing позволяет задать промежуток между словами.

```
word-spacing: normal|<промежуток между словами>;
```

Здесь мы можем задать либо значение "normal" для стандартного интервала между словами (это значение по умолчанию), либо абсолютное или относительное значение этого самого интервала.

Атрибут letter-spacing задает промежуток между символами текста.

```
letter-spacing: normal|<промежуток между символами>;
```

Способы задания промежутка здесь те же, что и у предыдущего атрибута.

Атрибут text-align задает горизонтальное выравнивание текста.

```
text-align: left|right|center|justify;
```

Здесь доступны значения "left" (выравнивание по левому краю; поведение по умолчанию), "right" (по правому краю), "center" (по центру) и "justify" (полное выравнивание).

Атрибут vertical-align задает вертикальное выравнивание текста.

```
vertical-align: baseline|sub|super|top|text-top|middle|bottom|
↳text-bottom|<промежуток между базовыми линиями>;
```

Этот атрибут принимает восемь значений:

- "baseline" (значение по умолчанию) задает выравнивание базовой линии текста абзаца по базовой линии родителя;
- "sub" превращает текст в нижний индекс;
- "super" превращает текст в верхний индекс;
- "top" выравнивает верх абзаца по верхнему краю родительского элемента;
- "text-top" выравнивает верх текста абзаца по верху текста родителя;
- "middle" выравнивает центр абзаца по центру родителя;
- "bottom" выравнивает низ абзаца по низу родителя;
- "text-bottom" выравнивает низ текста абзаца по низу текста родителя.

Кроме того, мы можем указать абсолютное или относительное значение, задающее, насколько выше или ниже базовой линии родителя должна находиться базовая линия текста абзаца.

Этот же атрибут можно использовать для выравнивания не только текста в абзаце, но и других элементов страницы, например, графических изображений. И, скорее всего, при этом для достижения нужного результата придется поэкспериментировать.

Атрибут `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.

Атрибут `white-space` управляет переносом строк.

```
white-space: normal|pre|nowrap;
```

Здесь доступны три значения:

- "`normal`" устанавливает обычное поведение текста (это установка по умолчанию);
- "`pre`" задает такое поведение текста, словно он заключен в теги `<PRE>` и `</PRE>`;
- "`nowrap`" действует аналогично тегам `<NOWRAP>` и `</NOWRAP>`.

А вот атрибут `display` весьма примечателен. Он позволяет задать *поведение* элемента страницы.

```
display: none|inline|block|list-item|run-in|compact|marker|table|
↳inline-table|table-row-group|table-header-group|table-footer-group|
↳table-row|table-column-group|table-column|table-cell|table-caption|
↳inherit;
```

Доступных значений у этого атрибута довольно много:

- "none" — элемент вообще не будет отображаться на странице, словно он и не задан в ее HTML-коде;
- "`inline`" — элемент страницы ведет себя как отдельный символ текста (*встроенный элемент*);
- "`block`" — элемент страницы ведет себя как абзац текста (*блочный элемент*);
- "list-item" — элемент страницы ведет себя как пункт списка;
- "`run-in`" — *встраивающийся* элемент. Если за таким элементом следует блочный элемент, он становится первым символом блочного элемента, в противном случае он сам становится блочным элементом;
- "`compact`" — *компактный* элемент. Если за таким элементом следует блочный элемент, он форматируется как однострочный встроенный элемент

и помещается левее блочного элемента. В противном случае он сам форматируется как блочный элемент;

- "marker" — маркер списка;
- "table" — таблица;
- "inline-table" — таблица, отформатированная как встроенный элемент;
- "table-row-group" — секция тела таблицы;
- "table-header-group" — секция заголовка таблицы;
- "table-footer-group" — секция "поддона" таблицы;
- "table-row" — строка таблицы;
- "table-column-group" — определение *группы столбцов* таблицы (формируется с помощью парного тега <COLGROUP> и служит для задания параметров сразу нескольких столбцов);
- "table-column" — определение столбца таблицы (формируется с помощью парного тега <COL> и служит для задания параметров отдельного столбца);
- "table-cell" — ячейка таблицы;
- "table-caption" — название таблицы;
- "inherit" — элемент наследует поведение от родителя.

Значение по умолчанию зависит от конкретного элемента страницы.

> ***Внимание!***
>
> Не все Web-обозреватели поддерживают все значения атрибута display.

Атрибут visibility управляет видимостью элемента страницы. Он принимает следующие значения:

- "visible" — элемент страницы видим;
- "hidden" — элемент страницы скрыт;
- "inherit" — элемент страницы видим, если видим его родитель, и наоборот (значение по умолчанию).

Ранее мы выяснили, что атрибут display также может использоваться для скрытия элемента страницы — для чего ему достаточно присвоить значение "none". Но при этом элемент страницы исчезает совсем, словно он и не присутствует в HTML-коде страницы. Web-обозреватель при этом даже не выделяет под него место на странице. В случае использования для той же задачи атрибута visibility место под элемент страницы таки будет выделено.

Параметры размеров и размещения

Задают размеры и местоположение элементов страниц: абзацев, горизонтальных линий, изображений, таблиц и пр. Всего их четыре.

Прежде всего, это атрибуты `width` и `height`. Они позволяют задать, соответственно, ширину и высоту элемента страницы.

```
width: auto|<ширина>;
height: auto|<высота>;
```

Можно задать либо абсолютное или относительное значение соответствующего размера, либо значение `"auto"`, отдающее управление этим размером на откуп Web-обозревателю (поведение по умолчанию).

Атрибут `float` задает такое поведение блочного элемента страницы (например, изображения или таблицы), при котором он прижимается к левому или правому краю родителя, а остальное содержимое страницы обтекает его.

Доступных для этого атрибута значений три:

- `"left"` — элемент страницы прижимается к левому краю родителя, а остальное содержимое обтекает его справа;
- `"right"` — элемент страницы прижимается к правому краю родителя, а остальное содержимое обтекает его слева;
- `"none"` — поведение по умолчанию.

Этот атрибут применим только к блочным элементам страниц: абзацам, изображениям (если они не находятся в абзаце), таблицам и пр.

Атрибут `clear` работает только в совокупности с атрибутом `float`, который мы только что рассмотрели. Он позволяет указать, что данный элемент страницы должен располагаться ниже всех элементов с заданным значением атрибута `float`.

Здесь доступны четыре значения:

- `"left"` — элемент страницы должен располагаться ниже всех элементов, атрибут `float` которых имеет значение `"left"`;
- `"right"` — элемент страницы должен располагаться ниже всех элементов, атрибут `float` которых имеет значение `"right"`;
- `"both"` — элемент страницы должен располагаться ниже всех элементов, атрибут `float` которых имеет значение `"left"` или `"right"`;
- `"none"` — поведение по умолчанию, когда элементы страницы с одинаковым значением атрибута `float` могут выстроиться в одну строку.

Этот атрибут также применяется только к блочным элементам страниц.

Параметры отступов

Особые атрибуты стилей, предусмотренные стандартом CSS, позволяют задать отступы между содержимым блочного элемента страницы и его воображаемой границей, а также между его границей и границами соседних элементов. Такие атрибуты применяются, если нужно создать отступы между изображением и текстом, между границей ячейки таблицы и ее содержимым, между границами соседних ячеек таблицы и пр.

Атрибуты `padding-left`, `padding-top`, `padding-right` и `padding-bottom` позволяют задать величины отступов между содержимым элемента страницы и его воображаемой границей соответственно слева, сверху, справа и снизу.

```
padding-left|top|right|bottom: <отступ>;
```

Здесь доступны абсолютные и относительные (в виде процентов от соответствующего размера элемента страницы) значения.

Атрибуты `margin-left`, `margin-top`, `margin-right` и `margin-bottom` позволяют задать величины отступов между воображаемой границей элемента страницы и границей соседнего элемента соответственно слева, сверху, справа и снизу.

```
margin-left|top|right|bottom: <отступ>;
```

Здесь также доступны абсолютные и относительные (в виде процентов от соответствующего размера элемента страницы) значения.

Рассмотрим пару примеров. Приведенный далее стиль переопределения тега `<TD>` задает отступ от содержимого ячейки таблицы до ее границы в 4 пиксела, а между границами соседних ячеек — в 2 пиксела.

```
TD { padding-left: 4px;
     padding-top: 4px;
     padding-right: 4px;
     padding-bottom: 4px;
     margin-top: 2px;
     margin-left: 2px;
     margin-right: 2px;
     margin-bottom: 2px }
```

А следующий пример задает для абзаца отступ слева в 20% от его ширины.

```
<P STYLE="margin-left: 20%">Абзац с отступом слева.</P>
```

В реальности же практически всегда применяются абсолютные значения отступов.

Параметры рамки

А теперь с помощью CSS мы научимся делать то, чего и не снилось HTML, — создавать рамки вокруг элементов страниц. Такие рамки рисуются по воображаемой границе элемента страницы.

Для создания рамки используется целый набор особых атрибутов стилей. Сейчас мы их рассмотрим.

Атрибуты `border-left-width`, `border-top-width`, `border-right-width` и `border-bottom-width` задают толщину рамки.

```
border-left|top|right|bottom-width: thin|medium|thick|<толщина>;
```

Здесь можно задать либо числовое значение толщины рамки в пикселах или иных абсолютных единицах измерения, 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.

Теплая компания атрибутов `border-left-color`, `border-top-color`, `border-right-color` и `border-bottom-color` задает цвет рамки.

```
border-left|top|right|bottom-color: <цвет>;
```

Цвет рамки всегда следует задавать явно — в противном случае рамка может быть не нарисована.

Атрибуты `border-left-style`, `border-top-style`, `border-right-style` и `border-bottom-style` задают стиль линий, которыми будет нарисована рамка, или отсутствие рамки.

```
border-left|top|right|bottom-style: none|dotted|dashed|solid|double|
↳groove|ridge|inset|outset;
```

Здесь доступны следующие значения:

- `"none"` — рамка отсутствует (это поведение по умолчанию);
- `"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".

И опять — пара примеров. Приведенный далее стиль переопределения тега <TD> окружает каждую ячейку таблицы рамкой серого цвета (RGB-код #CCCCCC), сформированную из сплошных линий толщиной в 1 пиксел.

```
TD { border-left-width: 1px;
     border-left-color: #CCCCCC;
     border-left-style: solid;
     border-top-width: 1px;
     border-top-color: #CCCCCC;
     border-top-style: solid;
     border-right-width: 1px;
     border-right-color: #CCCCCC;
     border-right-style: solid;
     border-bottom-width: 1px;
     border-bottom-color: #CCCCCC;
     border-bottom-style: solid }
```

А этот HTML-код создает под абзацем толстую, в 5 пикселов, черную штриховую линию:

```
<P STYLE="border-bottom-width: 5px; border-bottom-color: #FFFFFF;
↳border-bottom-style: dashed">Абзац, подчеркнутый штриховой линией.</P>
```

Параметры списков

С помощью атрибутов стилей CSS можно также задать некоторые параметры списков. Давайте выясним какие.

Атрибут `list-style-type` задает вид маркера списка.

```
list-style-type: disc|circle|square|decimal|lower-roman|upper-roman|
↳lower-alpha|upper-alpha|none;
```

Доступны следующие значения этого атрибута:

- "disc" — черный кружок (поведение по умолчанию для маркированных списков);
- "circle" — светлый кружок;
- "square" — светлый квадратик;
- "decimal" — арабская цифра (поведение по умолчанию для нумерованных списков);
- "lower-roman" — маленькая римская цифра;
- "upper-roman" — большая римская цифра;
- "lower-alpha" — маленькая латинская буква;

- "upper-alpha" — большая латинская буква;
- "none" — маркер и нумерация отсутствуют (поведение по умолчанию для обычных абзацев, не являющихся списками).

Атрибут list-style-image позволяет задать в качестве маркера пунктов списка какое-либо графическое изображение (*графический маркер*). В этом случае значение атрибута list-style-type игнорируется.

```
list-style-image: none|<интернет-адрес файла изображения>;
```

Здесь можно задать интернет-адрес графического изображения, которое будет использоваться как маркер, или значение "none", убирающее графический маркер. По умолчанию значение этого атрибута равно "none".

Атрибут list-style-position позволяет установить местоположение маркера или нумерации в пункте списка.

```
list-style-position: inside|outside;
```

Если задано значение "inside", маркер (нумерация) будет располагаться внутри границ пункта списка, то есть принадлежать ему; в этом случае текст пунктов списка будет выглядеть несколько компактнее. Если же задано значение "outside", маркер (нумерация) будет располагаться за пределами границ пункта; пункты таких списков лучше читаются. Значение по умолчанию — "outside".

Параметры курсора

Для нас как начинающих Web-программистов также будет полезен атрибут стиля cursor. Он устанавливает форму курсора мыши при наведении его на данный элемент страницы.

```
cursor: hand|crosshair|text|wait|default|help|e-resize|ne-resize|
↳n-resize|nw-resize|w-resize|sw-resize|s-resize|se-resize|auto;
```

Все доступные значения этого атрибута перечислены в табл. 3.2.

Таблица 3.2. *Значения атрибута* `cursor` *и соответствующие им формы курсора мыши*

Значение атрибута `cursor`	**Форма курсора мыши**
"hand"	"Указующий перст"
"crosshair"	Перекрестье, "прицел"
"text"	Текстовый курсор
"wait"	Песочные часы, обозначающие, что Windows "думу думает"

Таблица 3.2 *(окончание)*

Значение атрибута `cursor`	**Форма курсора мыши**
`"default"`	Обычная стрелка
`"help"`	Обычная стрелка с вопросительным знаком справа
`"e-resize"`	Стрелка вправо, "на восток"
`"ne-resize"`	Стрелка вверх-вправо, "на северо-восток"
`"n-resize"`	Стрелка вверх, "на север"
`"nw-resize"`	Стрелка вверх-влево, "на северо-запад"
`"w-resize"`	Стрелка влево, "на запад"
`"sw-resize"`	Стрелка вниз-влево, "на юго-запад"
`"s-resize"`	Стрелка вниз, "на юг"
`"se-resize"`	Стрелка вниз-вправо, "на юго-восток"
`"auto"`	Управление формой курсора мыши передается Web-обозревателю. Поведение по умолчанию

На этом мы пока закончим рассмотрение атрибутов стилей, предлагаемых стандартом CSS. В *главе 10* мы познакомимся еще с несколькими атрибутами стилей, используемыми при создании свободно позиционируемых элементов. А сейчас давайте лучше рассмотрим некоторые моменты, непосредственно связанные со стилями.

Псевдостили

В самом начале этой главы мы рассмотрели пять разновидностей стилей, предусматриваемых стандартом CSS. Есть еще одна, теперь уже шестая разновидность, которая стоит несколько особняком. Это *псевдостили*, имеющие особые имена и задающие для тегов, к которым они применены, особое поведение.

В данный момент из всех определенных CSS псевдостилей широко поддерживаются только *псевдостили гиперссылок*. С их помощью мы можем задать параметры гиперссылок обычных и посещенных, параметры активной гиперссылки и гиперссылки, на которую указывает курсор мыши. Всего этих псевдостилей четыре, и они перечислены в табл. 3.3.

Таблица 3.3. *Псевдостили гиперссылок*

Псевдостиль	Описание
A:link	Применяется к обычным гиперссылкам
A:active	Применяется к активной гиперссылке
A:visited	Применяется к посещенным гиперссылкам
A:hover	Применяется к гиперссылке, на которую указывает курсор мыши

Вот пример таблицы стилей, содержащей псевдостили гиперссылок:

```
A:link     { color: #FF0000;
             text-decoration: none }
A:active   { color: #FF0000;
             text-decoration: underline }
A:visited  { color: #FF0000;
             text-decoration: line-through }
A:hover    { color: #FF0000;
             text-decoration: underline }
```

Здесь мы задали для всех гиперссылок красный цвет. Для обычных гиперссылок мы отключили подчеркивание (значение `"none"` атрибута стиля `text-decoration`), для активной гиперссылки и гиперссылки, на которую указывает курсор мыши, — установили подчеркивание (значение `"underline"` того же атрибута), а для посещенных гиперссылок задали зачеркивание (значение `"line-through"`). Вообще-то, зачеркивать гиперссылки не стоит — посетителя сайта это может обескуражить, — но это только для примера.

Также мы можем комбинировать псевдостили и стилевые классы, чтобы задать особое поведение не для всех гиперссылок страницы, а только для избранных. Например (имя стилевого класса выделено полужирным шрифтом):

```
A.navbar:link     { color: #0000FF;
                    text-decoration: none }
A.navbar:active   { color: #0000FF;
                    text-decoration: underline }
A.navbar:visited  { color: #0000FF;
                    text-decoration: line-through }
A.navbar:hover    { color: #0000FF;
                    text-decoration: underline }

. . .
```

```
<A HREF="somepage.htm">Обычная гиперссылка</A>
<A HREF="somepage.htm" CLASS="navbar">Гиперссылка, к которой был применен
↳стиль</A>
```

Обратим внимание, что между именем стилевого класса и знаком двоеточия не должно быть пробела, иначе Web-обозреватель не сможет обработать эти стили.

Кроме того, многие Web-обозреватели поддерживают еще два псевдостиля. Псевдостиль `:first-letter` применяет определенные в нем параметры форматирования к первому символу элемента страницы, а псевдостиль `:first-line` — к первой его строке. Оба эти псевдостиля всегда комбинируются с другими стилями: стилями переопределения тегов, стилевыми классами и др.

Вот пример использования этих псевдостилей:

```
P:first-letter { font-size: larger }
P.underlined:first-line { text-decoration: underline }
```

Первый стиль увеличивает размер шрифта, которым выводится первая буква каждого абзаца (тега `<P>`), на одну ступень, создавая так называемую буквицу. Второй стиль подчеркивает первую строку каждого абзаца, значение атрибута `CLASS` которого равно `"underlined"`.

Контейнеры

Мы научились привязывать стили к определенным тегам. Мы знаем, как сделать фон у текстового абзаца или заключить его в рамку. И уверены, что все это нам обязательно пригодится.

Но можно ли заключить в одну рамку сразу несколько абзацев? Можно. Для этого мы используем контейнеры.

Контейнер — это особый парный тег HTML, служащий как бы "упаковкой" для других тегов (тех же абзацев). Содержимое контейнера ведет себя как единое целое, а значит, мы можем творить с этим единым целым все, что нам заблагорассудится. Например, привязать к нему стилевой класс, создающий рамку.

Контейнеры бывают двух видов. *Блочный контейнер*, или просто *блок*, ведет себя как отдельный абзац текста, то есть отображается в отдельной строке и располагается на некотором расстоянии от других элементов страницы. Создается он при помощи парного тега `<DIV>`, внутри которого располагаются теги, создающие его содержимое.

```
<DIV>
  <P>Первый абзац, помещенный в контейнер.</P>
```

```
  <P>Второй абзац, помещенный в контейнер.</P>
  <P>Третий абзац, помещенный в контейнер.</P>
</DIV>
```

Поскольку тег <DIV> поддерживает атрибуты CLASS и STYLE, мы без проблем сможем привязать к нему любой стиль.

```
.ourcontainer { border-left-width: 1px;
                border-left-color: #000000;
                border-left-style: solid;
                border-top-width: 1px;
                border-top-color: #000000;
                border-top-style: solid;
                border-right-width: 1px;
                border-right-color: #000000;
                border-right-style: solid;
                border-bottom-width: 1px;
                border-bottom-color: #000000;
                border-bottom-style: solid }
. . .
<DIV CLASS="ourcontainer">
  <P>Первый абзац, помещенный в контейнер.</P>
  <P>Второй абзац, помещенный в контейнер.</P>
  <P>Третий абзац, помещенный в контейнер.</P>
</DIV>
```

Приведенный HTML-код создает контейнер, окруженный тонкой черной рамкой и вмещающий три абзаца.

Вторая разновидность контейнеров — это *встроенные контейнеры*. Такой контейнер ведет себя как часть абзаца и создается с помощью парного тега <SPAN>.

```
<P><SPAN>Этот текст</SPAN> - часть контейнера.</P>
```

Тег <SPAN> поддерживает атрибуты CLASS и STYLE, чем мы и воспользуемся.

```
<P><SPAN STYLE="font-weight: bold">Этот текст</SPAN> - часть
↳контейнера.</P>
```

Здесь текст "Этот текст" будет выделен полужирным шрифтом.

Контейнеры применяются очень часто. Особенно блочные — без них в современном Web-программировании никуда. И в *главе 10*, когда пойдет разговор о свободно позиционируемых элементах страниц, мы в этом убедимся.

Физическое и логическое форматирование

Изучая возможности языка CSS, вы, наверное, уже удивились. В самом деле, стандарт CSS предусматривает атрибут стиля `font-weight`, выделяющий текст полужирным шрифтом. Но ведь в языке HTML, который мы рассмотрели в *главе 2*, существует тег `<STRONG>`, делающий то же самое. Какая между ними разница?

В той же *главе 2* говорилось, что тег `<STRONG>` не просто выделяет текст полужирным шрифтом, но и придает ему особую важность. (Как и тег `<EM>`, но в его случае текст выделяется курсивом и имеет важность меньшей степени.) Если мы загрузим любую из созданных ранее Web-страниц в специальный Web-обозреватель для незрячих, который читает текст Web-страниц вслух, последний выделит содержимое тега `<STRONG>`, скажем, интонацией.

Говорят, что `<STRONG>` и `<EM>`, а также практически все остальные изученные нами теги HTML *форматируют текст логически*. Поэтому они называются *тегами логического форматирования*.

Что касается стилей CSS, то они просто применяют к тексту заданное нами форматирование, не давая ему никакого особого значения. То есть если мы определим для абзаца какой-либо стиль, например:

```
<P STYLE="font-weight: bold">Полужирный текст.</P>
```

то фактически скажем Web-обозревателю: "Просто выведи этот абзац полужирным шрифтом, но не давай ему никакого особого значения". Мы таким образом применим к абзацу *физическое форматирование*. Именно физическим форматированием занимаются таблицы стилей CSS.

Логическое форматирование задает *структуру* Web-страницы: расположение и порядок следования абзацев, графических изображений, строк и ячеек в таблице и особое значение отдельных фрагментов текста. Физическое же форматирование задает *представление* Web-страницы: каким шрифтом будут набраны обычный текст абзацев, каким цветом выделить заголовки, будет ли у таблицы рамка и пр.

Правила хорошего тона Web-дизайна требуют, чтобы представление Web-страницы было отделено от ее структуры. Поэтому профессиональные Web-дизайнеры по возможности выносят определения стилей CSS в отдельные таблицы стилей. К тому же, HTML-код, не загроможденный определениями стилей, становится более читабельным.

Вы спросите: если есть теги HTML логического форматирования, то, может быть, есть теги, выполняющие и форматирование физическое? Да, есть.

Их довольно много. Так, существует тег *физического форматирования* `<B>`, аналогичный тегу `<STRONG>`, и тег `<I>`, аналогичный тегу `<EM>`. Однако все эти теги объявлены комитетом W^3C устаревшими и не рекомендованными к использованию (хотя они все еще входят в стандарт языка HTML и поддерживаются всеми Web-обозревателями). Вместо них рекомендуется использовать стили CSS. О которых, собственно, разговор уже закончен.

Что дальше?

В этой главе мы рассмотрели таблицы стилей CSS и используемый для их написания одноименный язык. Теперь наши Web-странички будут выглядеть более презентабельно.

На очереди — рассмотрение самого языка JavaScript, на котором и пишутся Web-сценарии. Следующая глава, посвященная ему, будет весьма объемной. Не расслабляемся!

Глава 4

Язык JavaScript

В предыдущих главах мы только готовились к тому, чтобы заняться Web-программированием. Изучали современные интернет-технологии, языки HTML и CSS, создавали первые, пока еще самые простые, Web-страницы. Но любая подготовка рано или поздно подходит к концу, и наступает пора главного.

Вот и нам пришла пора заняться главным — языком JavaScript. Именно на нем пишутся Web-сценарии, добавляющие "жизни" Web-страницам. Именно с ним мы будем иметь дело на протяжении всей этой книги. Да и после ее прочтения желающие углубить свои познания и получить дополнительную практику будут иметь дело именно с этим языком.

Это очень большая глава. Наверное, самая большая в книге. Но читать ее нужно очень внимательно.

Введение в JavaScript

В этом разделе мы рассмотрим основные понятия языка программирования JavaScript — то, без чего нельзя приступать к написанию полноценных Web-сценариев.

Основные понятия JavaScript

Прежде всего, выясним, что такое *Web-сценарий* (или просто *сценарий*, для краткости). Это программа, помещенная прямо в HTML-код Web-страницы и выполняющая какие-то действия над ней. Более подробно принципы написания Web-сценариев мы рассмотрим в *главах 5* и *6*, а пока что нам будет достаточно этого краткого определения.

Давайте рассмотрим пример небольшого Web-сценария:

```
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 сценарием.

Давайте разберем приведенный сценарий по выражениям. Вот первое из них:

```
x = 4;
```

Здесь мы видим число 4. В JavaScript такие числа, а также фрагменты текста, присутствующие в выражениях, называются *константами* — постоянными величинами, поскольку их значение не меняется (в конце концов, 4 всегда 4). 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 знак равенства (`=`), поставленный между переменной и константой. А он здесь стоит не просто так! (Вообще, в коде программы каждый символ что-то да значит.) Это *оператор* — команда, выполняющая какие-то действия над данными сценария. А если точнее, то знаком `=` 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.

Переменная, созданная в каком-либо сценарии, будет доступна во всех остальных сценариях, присутствующих в данной странице. Об исключениях из этого правила мы поговорим потом.

Каждое выражение JavaScript должно оканчиваться символом точки с запятой (`;`). Этот символ, собственно, обозначает конец выражения; его отсутствие вызывает ошибку обработки сценария.

Рассмотрим следующее выражение:

```
y = 5;
```

Оно аналогично первому и присваивает переменной `y` константу 5. Подобные выражения часто называют *математическими*.

Третье же 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 значком звездочки (`*`). Это *арифметический оператор*.

В результате выполнения приведенного ранее 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 используются в Web-сценариях довольно часто.

Закончим пока с выражениями и операторами. Давайте поговорим о данных, обрабатываемых сценариями.

Типы данных JavaScript

Любая программа при своей работе оперирует некими данными. Такими данными могут быть интернет-адрес Web-страницы, размеры одного из ее элементов, цена какого-нибудь товара в интернет-магазине, величина атмосферного давления и пр. Конечно, не составляют исключения и Web-сценарии — уже наш первый сценарий, рассмотренный ранее, обрабатывал какие-то данные.

JavaScript может манипулировать данными, относящимися к разным видам, или, как говорят программисты, *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. Например, это могут быть такие строки:

```
"JavaScript"
"1234567"
'Строковые данные — это последовательности символов.'
```

Строки могут иметь любую длину (определяемую количеством составляющих их символов), ограниченную лишь объемом свободной памяти компьютера. Разумеется, теоретически 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. Все специальные символы, поддерживаемые JavaScript, и их коды приведены в табл. 4.1.

Таблица 4.1. *Специальные символы, поддерживаемые JavaScript, и их коды*

Символ	Описание	Код
`\f`	Прогон листа	12
`\n`	Перевод строки	10
`\r`	Возврат каретки	13
`\t`	Табуляция	9
`\"`	Двойная кавычка	22
`\'`	Одинарная кавычка	27
`\\`	Обратный слеш	92
`\999`	Любой символ по его восьмеричному коду (обозначен как `999`)	–
`\xFF`	Любой символ по его шестнадцатеричному коду (обозначен как `FF`)	–
`\xFFFF`	Любой символ по его коду Unicode (обозначен как `FFFF`)	–

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 все 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
```

Для записи дробных чисел может быть использована экспоненциальная форма вида `<мантисса>E<порядок>`. Вот примеры заданных таким образом чисел (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.

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 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, поддерживаемых JavaScript и не описанных здесь, мы рассмотрим позже.

Переменные

В начале этой главы мы кое-что узнали о переменных. Сейчас настало время поговорить о них подробнее.

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.

Есть много рекомендаций по поводу того, как следует именовать переменные (и не только переменные). Автор же этой книги рекомендует следовать простому правилу: имя должно быть "говорящим", то есть отражать назначение переменной. Например, переменную, в которой хранится интернет-адрес страницы, лучше всего назвать `pageAddress` — так будет сразу понятно, зачем она нужна. Но не стоит слишком усердствовать: имена типа `currentPageHTTPPortNumber` очень долго набирать. Для *временных* же переменных (хранящих данные, необходимые в данный конкретный момент и используемые только в одном сценарии на странице) лучше всего брать однобуквенные имена: `i`, `a`, `x`, `y` и т. п.

Объявление переменных

Перед использованием переменной в коде сценария рекомендуется выполнить ее *объявление*. Для этого используется *оператор объявления переменной* `var`, после которого указывается имя переменной. Вот так:

```
var x;
```

Теперь объявленной переменной можно присвоить какое-либо значение:

```
x = 1234;
```

и использовать в 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, siteAddress = "http://www.somesite.ru/";
```

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.

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. Давайте с ними познакомимся.

Арифметические операторы

Арифметические операторы служат для выполнения арифметических действий над числами. Все арифметические операторы, поддерживаемые JavaScript, перечислены в табл. 4.2.

Таблица 4.2. *Арифметические операторы*

Оператор	Описание
`-`	Смена знака числа
`+`	Сложение
`-`	Вычитание
`*`	Умножение
`/`	Деление
`%`	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 и в переменную s.

Операторы инкремента и декремента могут ставиться 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 используется в других вычислениях, 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;
```

> ***Внимание!***
>
> Операторы инкремента и декремента рекомендуется использовать, если значение какой-либо переменной нужно увеличить или уменьшить на единицу. Эти операторы выполняются быстрее, чем соответствующие математические выражения.

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".

Двоичные операторы

Двоичные операторы, называемые также *побитными*, затрагивают двоичное представление чисел и используются, в основном, для низкоуровневого программирования на JavaScript. Всего их семь и все они перечислены в табл. 4.3. Мы не станем подробно останавливаться на них, а просто рассмотрим.

Таблица 4.3. *Двоичные операторы*

Оператор	Описание
~	НЕ (двоичная инверсия)
&	И (двоичное умножение)
|	ИЛИ (двоичное сложение)
^	Исключающее ИЛИ
<<	Сдвиг влево с заполнением нулями младших разрядов
>>	Сдвиг вправо с заполнением старших разрядов содержимым знакового (самого старшего) разряда
>>>	Сдвиг вправо с заполнением старших разрядов нулями

Здесь оператор ~ — унарный, остальные операторы — бинарные. Причем в случае операторов сдвига второй операнд задает количество разрядов, на которые выполняется сдвиг.

Операторы присваивания

Один 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*. Такие операторы позволяют выполнять операцию присваивания одновременно с другой операцией:

```
a = a + b;
a += b;
```

Два этих выражения эквивалентны по результату. Просто во втором был использован оператор сложного присваивания +=.

Все операторы сложного присваивания, поддерживаемые JavaScript, и их эквиваленты приведены в табл. 4.4.

Таблица 4.4. *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 и возвращают логическое значение. Если условие сравнения выполняется, возвращается значение `true`, если не выполняется — `false`. Вот примеры выражений с операторами сравнения:

```
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`.

Все поддерживаемые JavaScript операторы сравнения приведены в табл. 4.5.

С первыми шестью операторами сравнения все понятно. Но на двух последних операторах — "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 несовместимости типов операндов возвращают `false`.

Таблица 4.5. *Операторы сравнения*

Оператор	Описание
<	Меньше
>	Больше
==	Равно
<=	Меньше или равно
>=	Больше или равно
!=	Не равно
===	Строго равно
!==	Строго не равно

Логические операторы

Логические операторы выполняют действия над логическими значениями. Все они приведены в табл. 4.6. А в табл. 4.7 и 4.8 показаны результаты выполнения этих операторов.

Таблица 4.6. *Логические операторы*

Оператор	Описание
!	НЕ (логическая инверсия)
&&	И (логическое умножение)
||	ИЛИ (логическое сложение)

Таблица 4.7. *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 4.8. *Результаты выполнения оператора НЕ*

Операнд	! (НЕ)
true	false
false	true

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);
```

Оператор получения типа *typeof*

Оператор получения типа `typeof` возвращает строку, описывающую тип данных операнда. Все значения, которые он может вернуть, перечислены в табл. 4.9.

Таблица 4.9. *Значения, возвращаемые оператором* `typeof`

Тип данных	Возвращаемое значение
Строковый	`"string"`
Числовой	`"number"`
Логический	`"boolean"`
Объектный (см. далее)	`"object"`
Функциональный (см. далее)	`"function"`
null	`"null"`
undefined	`"undefined"`

Оператор `typeof` может использоваться, например, так:

```
status = typeof(somevar);
```

Здесь мы присваиваем результат выполнения оператора `typeof` строковой переменной `status`. Впоследствии он может быть использован, например, в условном выражении.

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 операторов

```
var a, b, c, d, e, f;
a = 11;
b = "12";
c = a + b;
d = "JavaScript";
e = 2;
f = d + e;
```

значение переменной `b` при сложении с переменной `a` будет преобразовано в числовой тип; таким образом, переменная `c` будет содержать значение 23. Но так как значение переменной `d` не может быть преобразовано 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 сценария прерывается в связи с обнаружением ошибки совсем в другом его месте, на абсолютно верном операторе. Поэтому лучше всего не допускать подобных казусов, оперировать только переменными совместимых типов и самим выполнять нужные преобразования типов (как это сделается, мы рассмотрим потом).

Приоритет операторов

Последний вопрос, который мы здесь разберем, — это 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;
```

А в этом случае сначала будет выполнено умножение значения `c` на 10, а уже потом к полученному произведению будет прибавлено значение `b`. Оператор умножения имеет бóльший приоритет, чем оператор сложения, поэтому порядок "строго слева направо" будет нарушен.

Самый низкий приоритет имеют операторы 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. 4.10 перечислены все изученные нами операторы в порядке убывания их приоритетов.

Таблица 4.10. *Приоритет операторов (в порядке убывания)*

Операторы	**Описание**
`++ -- - ~ ! typeof`	Инкремент, декремент, смена знака, двоичное и логическое НЕ, определение типа
`* / %`	Умножение, деление, взятие остатка
`+ -`	Сложение и объединение строк, вычитание
`<< >> >>>`	Сдвиги
`< > <= >=`	Операторы сравнения
`== != === !==`	
`&`	Двоичное И
`^`	Двоичное исключающее ИЛИ
`|`	Двоичное ИЛИ
`&&`	Логическое И
`||`	Логическое ИЛИ
`?`	Условный оператор (см. ниже)
`= <оператор>=`	Присваивание, простое и сложное
`,`	Переключение приращений (см. далее)

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;
```

В этом случае 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 особым 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 заключаются в фигурные скобки, например:

```
{
  b = "12";
  c = a - b;
}
```

Как правило, блоки не используются 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*. Существует также другая, "вырожденная" разновидность условного выражения, содержащая только одно выражение, которое выполняется при выполнении условия и пропускается, если условие не выполнено.

Что может служить условием в условном выражении? Значение логической переменной или результат вычисления логического выражения. В качестве же выражений, которые должны быть выполнены или не выполнены, в зависимости от условия, могут быть использованы блоки.

Условное выражение имеет следующий формат:

```
if (<условие>)
  <блок "то">
else
  <блок "иначе">
```

"Вырожденный" формат его выглядит так:

```
if (<условие>)
  <блок "то">
```

Для написания условных выражений используются особые ключевые слова `if` и `else`. `Условие` — это и есть логическое выражение, в соответствии с которым JavaScript принимает решение, какой блок выполнить. Отметим, что оно всегда записывается в 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 выражении). А если `блок "иначе"` отсутствует, выполняется следующее выражение сценария.

Внимание!

Значения `null` или `undefined` преобразуются в `false`. Не забываем об этом.

Рассмотрим несколько примеров.

```
if (x == 1) {
  a = "Единица";
  b = 1;
}
 else {
  a = "Не единица";
  b = 22222;
}
```

Здесь мы сравниваем значение переменной `x` с единицей и в зависимости от результатов сравнения присваиваем переменным `f` и `h` разные значения.

Условие может быть довольно сложным:

```
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, так как фрагменты кода слишком просты, чтобы оформлять их в виде блоков.

Часто встречаются условные выражения такого вида:

```
if (str) {
  . . .
}
```

Причем переменная `str` содержит заведомо не логическое значение, а, скажем, строковое. В этом случае вступают в силу правила преобразования типов, изложенные в соответствующем параграфе этой главы. Если переменная `str` содержит строку, она будет преобразована в значение `true`; если же там хранится значение `null` или `undefined` (то есть эта переменная еще даже не объявлена), оно будет преобразовано в `false`. Подобные условные выражения, кстати, можно использовать для проверки, была ли объявлена та или иная переменная (или параметр функции; о функциях см. далее в этой главе).

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 обычного 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. Ниже него — только операторы присваивания.

Выражения выбора

Выражение выбора — это фактически несколько условных выражений, объединенных в одном. Его формат таков:

```
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 используются ключевые слова `switch`, `case` и `default`.

Давайте выясним, как выполняется выражение выбора. Результат вычисления `исходного выражения` последовательно сравнивается со `значением 1`, `значением 2` и т. д. и, если такое сравнение увенчалось успехом, 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).

Вот пример выражения выбора:

```
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"`.

Встретив *оператор прерывания* `break`, JavaScript прерывает выполнение блока, в котором оно присутствует, и начинает выполнение кода, следующего за выражением выбора. Если его опустить, то будет выполнен следующий блок. Так, если значение `условия` совпало со `значением 1` и был выполнен `блок 1`, не содержащий действия `break`, будет также выполнен `блок 2`.

Давайте уберем все действия `break` в нашем примере:

```
switch (a) {
  case 1 :
    out = "Единица";
  case 2 :
    out = "Двойка";
  case 3 :
    out = "Тройка";
  default :
    out = "Другое число";
}
```

В этом случае все блоки будут выполняться последовательно, один за другим. И переменной `out` всегда будет присваиваться строка `"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 используется, если какой-то код нужно выполнить строго определенное число раз. Вероятно, это наиболее часто используемый 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 используется для подсчета, 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 вернет `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.

Кроме того, допустимо использовать счетчик цикла в одном из выражений тела цикла — как это сделали мы. Счетчик `i` будет содержать последовательно возрастающие значения от 1 до 10, которые можно использовать 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.

> ***НА ЗАМЕТКУ***
>
> В особом "вырожденном" случае цикл `for` может даже не содержать тела. В этом случае "полезную нагрузку" цикла несет на себе выражение приращения.

Оператор переключения приращений

JavaScript предлагает особый оператор, используемый в выражениях `приращения` циклов со счетчиком. Это оператор , (запятая), называемый *оператором переключения приращений*. Он позволяет выполнять то одно, то другое выражение `приращения`. Его формат таков:

<первое приращение>, *<второе приращение>*

При первом прохождении цикла выполняется `первое приращение`, при втором — `второе приращение`, при третьем — снова `первое приращение` и т. д.

```
for (i = 0; j < 11; i++, j++) {
  a = i * 2 + 1;
  b = j / 2 - 3;
}
```

Оператор переключения приращений имеет самый низкий приоритет из всех операторов.

Цикл с постусловием

Цикл с постусловием во многом похож на цикл со счетчиком, а именно в том, что 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 используются ключевые слова `do` и `while`. Поэтому такие циклы часто называют "циклами do-while".

Вот пример цикла с постусловием:

```
do {
  a = a * i + 2;
  i = ++i;
} while (a < 100);
```

А вот еще один пример:

```
var a = 0, i = 1;
do {
  a = a * i + 2;
  i = ++i;
} while (i < 20);
```

Здесь мы используем счетчик, чье конечное значение ограничено. Хотя, конечно, в данном примере удобнее было бы использовать 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 используется 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 = ++i;
}
```

Прерывание и перезапуск цикла

Иногда бывает нужно прервать выполнение цикла. Для этого JavaScript предоставляет программистам операторы `break` и `continue`.

Уже знакомый нам по выражениям выбора 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.

```
while (a < 100) {
  i = ++i;
  if (i > 9 && i < 11) continue;
  a = a * i + 2;
}
```

Здесь мы пропускаем выражение вычисления `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 места любого сценария на данной странице (повторно используемый код, как его часто называют). Так что, если какой-то фрагмент кода встречается в нескольких местах вашего сценария или сценариев, 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 сценарии, ее нужно объявить. Объявление функции выполняется с помощью ключевого слова `function`.

```
function <имя функции>([<список параметров, разделенных запятыми>])
  <тело функции>
```

`Имя функции`, как уже говорилось, должно быть уникально в пределах 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. 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 помещается в круглые скобки, а сами параметры отделяются 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. Чтобы вернуть результат из функции в выражение, из которого она была вызвана, используется *оператор возврата* `return`:

```
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, вот так:

```
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`. (О вызове функций будет рассказано далее.) Для этого мы используем оператор получения типа `typeof` (он был описан ранее). Если он вернет строку `"undefined"` (то есть параметр `b` не был указан), мы присваиваем этому параметру число 2 — его значение 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

Приведенная ранее функция объявляет внутри своего тела свою собственную переменную `c`. Это так называемая *локальная* переменная. Такие переменные доступны только внутри тела функции, в котором они объявлены. При завершении выполнения функции значения всех локальных переменных теряются.

Разумеется, любая функция может обращаться к любой переменной, объявленной вне ее тела (переменной *уровня страницы*). При этом нужно помнить об одной вещи. Если существуют две переменные с одинаковыми именами, одна — уровня страницы, другая — локальная, то при обращении по этому имени будет получен доступ **к локальной** переменной. Одноименная переменная уровня страницы будет "замаскирована" своей локальной "тезкой".

Вызов функций

После того как мы объявили функцию, мы можем *вызвать* ее из любого сценария, присутствующего на этой же странице. Для этого используется следующий формат:

```
<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 сложном выражении.

> ***Внимание!***
>
> При вызове функции подставляйте именно фактические параметры, а не формальные, использованные 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.

```
s = 4 * divide(x, r) + y;
```

А здесь мы выполняем вызов функции с переменными в качестве фактических параметров.

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 результата, то она вызывается вот так:

```
initVars(1, 2, 3, 6);
```

Более того, таким образом 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 сценария.

Если функция не принимает параметров, при ее вызове все равно нужно указать пустые скобки. Иначе возникнет ошибка выполнения 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. А именно — присваивать функции переменным. Например:

```
var someFunc;
someFunc = cmp;
```

Здесь мы присвоили переменной `someFunc` объявленную ранее функцию `cmp`. Заметим, что в этом случае справа от оператора присваивания указывается 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.

Рекурсия

И еще один важный вопрос, связанный с написанием функций.

Мы уже знаем, что функции могут вызывать другие функции, конечно, если те уже объявлены. Более того, функции могут вызывать и сами себя. Такой прием программирования иногда может быть очень полезен и называется *рекурсией*.

В обычных условиях, если функция вызовет сама себя, то она войдет в бесконечный цикл вызовов, из которого нет выхода (так называемая *бесконечная рекурсия*). В конце концов, это завершится аварийной остановкой программы, а то и крахом операционной системы. Поэтому функция, предназначенная

для рекурсивного вызова, должна предусматривать возможность выхода из этого цикла вызовов. Стандартного способа сделать это не существует, поэтому решение для каждого конкретного случая нужно искать особо.

Рассмотрим пример функции, правильно использующей рекурсию:

```
function factorial(a)  {
  if (a == 0) {
    return 1;
  }
  else
    return (a * factorial(a - 1));
}
```

Эта функция вычисляет факториал числа `a`. Она вызывает сама себя для того, чтобы получить факториал числа `a - 1`. Заметим также, что в теле функции выполняется проверка условия равенства `a` нулю; если это условие истинно, возвращается единица, дальнейших рекурсивных вызовов не происходит, а все уже сделанные "хором" завершаются. Таким образом, эта функция имеет защиту от бесконечной рекурсии, ведь когда-нибудь она все равно получит в качестве параметра ноль.

А вот пример другой функции, использующей рекурсию неправильно:

```
function endless(a, b)  {
  var c;
  c = endless(a, b);
}
```

Эта функция будет вызывать себя бесконечно и в конце концов "обрушит" Web-обозреватель. А все потому, что в ней нет условия, которое прерывало бы бесконечный цикл рекурсивных вызовов.

Встроенные функции JavaScript

Мы выяснили, как пишутся и используются функции, узнали об их преимуществах и "подводных камнях", с которыми сталкиваются начинающие Web-программисты. Осталось выяснить еще кое-что — и с функциями можно заканчивать.

Язык JavaScript предоставляет разработчикам целый набор *встроенных* функций. Эти функции уже объявлены и реализованы в самом языке. Некоторые из них, поддерживаемые всеми современными Web-обозревателями, перечислены далее.

НА ЗАМЕТКУ

На самом деле встроенные функции JavaScript — это не совсем функции. Но об этом поговорим потом.

Функция `parseInt` позволяет преобразовать строку, содержащую число, в числовое значение.

`parseInt(`*<строка>*`[, `*<основание системы счисления>*`])`

Первым параметром этой функции передается сама строка, содержащая число. Вторым, необязательным, параметром передается число, являющееся основанием системы счисления, в которой представлено число в переданной первым параметром строке. Если этот параметр не указан, подразумевается, что число представлено в десятичной системе счисления.

Функция `parseInt` успешно преобразует в числовой тип строку, содержащую одно только число или число, за которым следуют символы, не являющиеся цифрами. Если же строка никак не может быть преобразована в число, эта функция вернет значение `NaN`.

Приведем несколько примеров использования функции `parseInt`:

```
a = parseInt("12");
b = parseInt("34rty");
c = parseInt("JavaScript");
```

После выполнения этих выражений в переменной `a` окажется число 12, в переменной `b` — число 34 (поскольку первые два символа переданной строки — цифры, их можно преобразовать в число), а в переменной `c` — `NaN` (строку "JavaScript" никак нельзя преобразовать в число).

Функция `parseFloat` выполняет преобразование строки в число с плавающей запятой. Формат ее вызова похож на формат вызова функции `parseInt`, за тем исключением, что задать основание системы счисления мы не можем. И ведет она себя так же, как функция `parseInt`.

`parseFloat(`*<строка>*`)`

Вот примеры ее использования:

```
a = parseFloat("1.2");
b = parseFloat("3.14pi");
c = parseFloat("WWW");
```

После выполнения этих выражений в переменной `a` окажется число 1,2, в переменной `b` — число 3,14, а в переменной `c` — `NaN`.

Функция `isNaN` возвращает `false`, если переданный ей единственный параметр представляет собой правильное число, и `true`, если не представляет.

Как правило, в качестве параметра этой функции используются математические выражения, корректность результатов вычисления которых нужно проверить.

```
x = 2;
y = 3;
z = 0;
s = isNaN(x + y);
t = isNaN(y / z);
```

Здесь в переменной `s` окажется значение `false` (так как при сложении 2 и 3 получится правильное число), а в переменной `t` — значение `true` (при делении 3 на 0 получится бесконечность, не являющаяся правильным числом).

Функция `inFinite` возвращает `true`, если переданный ей параметр представляет собой конечное число, и `false`, если математическую бесконечность. В этом смысле она по действию противоположна функции `isNaN`.

```
x = 2;
y = 3;
z = 0;
s = isNaN(x + y);
t = isNaN(y / z);
```

Здесь в переменной `s` окажется значение `true` (при сложении 2 и 3 получится конечное число), а в переменной `t` — значение `false` (при делении 3 на 0 получится бесконечность).

Функция `eval` принимает в качестве параметра выражение JavaScript, записанное в виде обычной строки, и возвращает результат его выполнения.

```
eval(<выражение JavaScript в виде строки>)
```

Приведенное далее выражение поместит в переменную `res` число 4.

```
res = eval("2 + 2");
```

Функция `escape` принимает в качестве параметра строку и преобразует ее к виду, пригодному для передачи в составе интернет-адреса. (О передаче данных в составе интернет-адреса мы поговорим в *главе 11*.) При этом символы пробела, двоеточия, кавычек, амперсанда и некоторые другие будут преобразованы в их коды, предваренные символом процента.

```
escape(<кодируемая строка>)
```

При выполнении выражения

```
encStr = escape("&");
```

переменная `encStr` получит строку `"%26"` — именно так кодируется для передачи символ амперсанда.

Функция `unescape` выполняет обратное преобразование — кодированную строку в изначальную.

```
unescape(<декодируемая строка>)
```

Приведенное далее выражение поместит в переменную `decStr` строку `"&"`.

```
decStr = escape("%26");
```

Массивы

Массив — это пронумерованный набор переменных (*элементов*), ведущий себя как одна переменная. 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;
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 (то есть будет содержать значение `undefined`).

Если будет нужно, мы легко сможем добавить к массиву еще один элемент, просто присвоив ему требуемое значение. Вот так:

```
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:

```
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*).

```
var hash;
hash["browser1"] = "Internet Explorer";
hash["browser2"] = "Firefox";
hash["browser3"] = "Opera";
```

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), в действительности 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 переменную 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 = ["Internet Explorer", "Opera", "Firefox"];
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] = "IE";
```

и обратимся к нему через переменную `newArray`:

```
s = newArray[0];
```

то в переменной `s` окажется строка `"IE"` — новое значение первого элемента этого массива. То есть фактически переменные `myArray` и `newArray` указывают на один и тот же массив.

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, действиями, 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

Объект — это сложный тип данных, включающий в себя множество переменных — *свойств* — и набор функций для манипулирования значениями этих переменных — *методов*. Здесь все достаточно просто: свойства хранят некие данные, а методы их обрабатывают, выдавая на выходе другие данные или совершая действия с самим объектом.

Каждый объект должен иметь уникальное имя, по которому к нему можно обратиться. К именам объектов предъявляются те же требования, что и к именам переменных и функций.

В качестве примера объекта можно привести Web-страницу. Она имеет свойства — интернет-адрес, размер файла, в котором хранится, имя фрейма, в котором открыто, — и набор методов — открыть в окне Web-обозревателя, обновить и пр. Мы можем изменить значения некоторых ее свойств и вызвать методы для манипуляций с ней.

Собственно объект только описывает набор доступных в нем свойств и методов. Его нельзя использовать для работы напрямую. Для этого мы должны создать на его основе *экземпляр объекта*, а уж его использовать для хранения реальных данных и манипуляций с ними.

> ***НА ЗАМЕТКУ***
>
> Экземпляры некоторых объектов нам самим создавать не нужно — это сделает либо JavaScript, либо Web-обозреватель. О таких объектах и их экземплярах речь пойдет в следующих главах этой книги.

В Web-сценариях, написанных на JavaScript, объекты и их экземпляры используются постоянно. Так что привыкаем к работе с объектами!

Работа с объектами и их экземплярами

Но оставим пока в покое экземпляры объектов, создаваемые Web-обозревателем. Давайте посмотрим, как осуществляется работа с объектами и их экземплярами, которыми нам придется манипулировать самостоятельно.

Итак, перед тем как начать работу с экземпляром какого-либо объекта, его нужно создать. Создание экземпляра объекта выполняется с помощью *оператора создания экземпляра* `new`:

```
new <имя объекта>([<список параметров, разделенных запятыми>])
```

После создания экземпляра объекта оператор `new` возвращает ссылку на него. (О ссылках было рассказано ранее.) Эта ссылка присваивается какой-либо переменной.

Список параметров может как присутствовать, так и отсутствовать. Обычно он содержит значения, которые присваиваются свойствам экземпляра при его создании.

Вот пример создания экземпляра `obj` некоего объекта `someObject`:

```
var obj;
obj = new someObject(a, b);
```

Теперь мы можем обращаться к свойствам и методам созданного экземпляра. Для этого используется так называемый синтаксис "с точкой":

```
<имя переменной, хранящей ссылку на экземпляр объекта>.<имя свойства>
<имя переменной, хранящей ссылку на экземпляр объекта>.<имя метода>
([<список фактических параметров, разделенных запятыми>])
```

То есть мы сначала указываем имя переменной, где хранится ссылка на экземпляр, потом ставим точку, а после точки пишем имя нужного свойства или метода. В случае метода также не забываем скобки со списком фактических параметров; если же метод не принимает параметров, указываем пустые скобки. Обратим внимание, что пробелы здесь недопустимы.

Вот несколько примеров:

```
obj.prop1 = 0;
a = obj.prop2 + 2;
obj.method1();
```

Иногда бывает, что один объект содержит внутри себя другие объекты (так называемые *внутренние*). Для обращения к свойствам и методам внутренних объектов используется тот же синтаксис, но обладающий, скажем так, "повышенной этажностью":

```
outerObj.innerObj.prop = 0;
```

где `outerObject` — экземпляр *внешнего объекта* (имеющего в своем составе внутренние объекты), а `innerObject` — экземпляр внутреннего объекта.

Выражение, описывающее цепочку вложенных друг в друга объектов, называется *путем*. Иногда путь бывает очень длинным:

```
outerObj.innerObj.innerInnerObj.prop = 10;
```

Как и в случае массива, мы можем присваивать ссылку на экземпляр объекта другой переменной:

```
obj2 = obj;
```

Как правило, экземпляры объектов удалять не нужно — их удалит сам JavaScript, как только страница со сценариями, создавшими эти экземпляры, будет выгружена Web-обозревателем из памяти. Но JavaScript на всякий пожарный случай предусматривает *оператор удаления экземпляра* `delete`.

```
delete <имя переменной, хранящей ссылку на экземпляр объекта>;
```

Например,

```
delete obj, obj2;
```

> ***НА ЗАМЕТКУ***
>
> Вообще, правила хорошего тона программирования предписывают удалять экземпляры объектов сразу же после окончания работы с ними. Но, как было сказано ранее, в случае Web-сценариев это необязательно.

Объект *Object* и использование его экземпляров

JavaScript предоставляет Web-программистам весьма специфический объект `Object`. Но в чем же его специфика?

Во-первых, он имеет только два свойства и четыре метода, да и те служебные. Но мы можем создавать свойства прямо в его экземплярах, просто присваивая им нужные значения (*свойства экземпляра* — они принадлежат не объекту, а только его экземпляру, в котором были созданы). Так что если нам нужно хранить некоторый набор данных в одном месте, экземпляр объекта `Object` — идеальный выбор (помимо массива).

Во-вторых, экземпляры этого объекта создаются с помощью особых выражений, называемых *инициализаторами*. Инициализатор имеет следующий формат вызова:

```
{<имя свойства 1>:<значение свойства 1>
[, <имя свойства 2>:<значение свойства 2>
[, <имя свойства 3>:<значение свойства 3>[. . .]]}
```

То есть мы создаем свойства экземпляра прямо в инициализаторе. А "на выходе" получаем все ту же ссылку на экземпляр, которую присваиваем переменной.

```
var otherObj;
otherObj = {prop1:3, prop2:4, prop3:"abc"};
```

Так мы получим экземпляр `otherObj` объекта `Object`, содержащий три свойства с уже заданными значениями.

Впоследствии мы можем добавить в этот экземпляр новые свойства, просто присвоив им нужные значения:

```
otherObj.prop4 = 5.87;
otherObj.prop10 = null;
```

В-третьих, к свойствам экземпляров объекта `Object` можно обращаться как к элементам ассоциативного массива (об ассоциативных массивах было рас-

сказано ранее). Индексы этих элементов будут совпадать с именами соответствующих им свойств.

```
otherObj["prop1"] = 11;
a = otherObj["prop3"] + "def";
```

А в-четвертых, на основе объекта `Object` можно создавать свои собственные объекты. Как — мы рассмотрим в конце этой главы.

Новые возможности JavaScript, применяемые при работе с объектами

А теперь самое время рассмотреть специфические возможности языка JavaScript, применяемые при работе с объектами и экземплярами.

Оператор проверки объекта `instanceof` проверяет, создан ли заданный экземпляр на основе заданного объекта. Если это так, возвращается `true`, в противном случае — `false`.

```
<экземпляр> instanceof <объект>
```

Здесь `экземпляр` задается в виде переменной, содержащей ссылку на него, а `объект` — в виде имени объекта.

```
if (obj instanceof SomeObject)
  . . .
```

Особая разновидность блочных выражений применяется при работе с несколькими свойствами и методами одного и того же экземпляра объекта. Это *блок with*. Формат его написания следующий:

```
with (<переменная, хранящая ссылку на экземпляр>) {
  <обращение к свойству экземпляра>
  . . .
  <обращение к методу экземпляра>
  . . .
}
```

Проиллюстрировать использование блока `with` лучше на примере. Пусть у нас есть такой фрагмент сценария:

```
someObj.prop1 = 1;
someObj.prop2 = 2;
someObj.prop3 = 3;
someObj.method1();
```

Здесь мы обращаемся к трем свойствам и одному методу экземпляра `someObj`. Выражения получаются довольно длинными. Но, используя блок `with`, мы можем их сократить.

```
with (someObj){
  prop1 = 1;
  prop2 = 2;
  prop3 = 3;
  method1();
}
```

Преобразованный с использованием блока `with` фрагмент кода стал заметно компактнее. Кроме того, он будет быстрее выполняться.

Особый цикл, называемый *циклом просмотра*, позволяет последовательно обратиться ко всем свойствам заданного экземпляра. Вот формат его написания:

```
for (<переменная-значение свойства> in <экземпляр объекта>)
  <тело цикла>
```

`Переменная-значение свойства` каждый раз получает значение очередного свойства `экземпляра объекта`. Мы можем использовать ее в `теле цикла`.

Как видно, для создания циклов просмотра используются ключевые слова `for` и `in`. Поэтому такие циклы еще называют циклами "for-in".

Вот пример цикла просмотра:

```
for (pr in someObj) {
  pr = "/"" + pr + "/"";
}
```

Этот сценарий просматривает все свойства экземпляра `someObj` и заключает их значения в кавычки. (Предполагается, что значения всех свойств этого экземпляра имеют строковый тип.)

> ***ВНИМАНИЕ!***
>
> Цикл просмотра поддерживается только для пользовательских объектов — тех, что создал сам программист. (О создании пользовательских объектов будет рассказано в конце этой главы.)

Встроенные объекты JavaScript

Все объекты, доступные в JavaScript, делятся на три вида:

- предоставляемые самим языком JavaScript;
- предоставляемые Web-обозревателем (о них мы будем говорить в последующих главах этой книги);
- созданные программистом (о них разговор пойдет в конце этой главы).

Объекты первого типа еще называются *встроенными*. Их довольно много. Сейчас мы их рассмотрим.

Объект строки *String*

Объект `String` представляет обычную строку, то есть константу или переменную строкового типа. Да-да, JavaScript рассматривает обычные типы данных — строки, числа, даже функции — как объекты и позволяет использовать их свойства и методы.

Так, написав выражение:

```
var str;
str = "JavaScript";
```

мы получим экземпляр `str` объекта `String`, содержащий строку `"JavaScript"`.

Экземпляр этого объекта также может быть создан с помощью уже знакомого нам оператора `new`:

```
new String([<строковая константа>]);
```

Строковая константа, являющаяся параметром, станет значением нового объекта. Например:

```
var str;
str = new String("JavaScript");
```

Если параметр опущен, будет создана пустая строка (не содержащая символов).

Объект `String` поддерживает единственное свойство `length`, возвращающее длину строки в символах.

```
var someStr = "WWW";
l1 = someStr.length;
l2 = "JavaScript".length;
```

После выполнения этого сценария в переменной `l1` окажется число 3 (длина строки `"WWW"`), а в переменной `l2` — 10 (длина строки `"JavaScript"`).

Теперь давайте рассмотрим многочисленные методы объекта `String`.

> ***Внимание!***
>
> Все эти методы не изменяют текущую строку, то есть значение самого экземпляра объекта `String`.

Метод `anchor` возвращает HTML-код, создающий якорь. Текущая строка станет содержимым этого якоря, а строка, переданная в качестве параметра, — его именем.

```
anchor(<имя якоря в строковом виде>)
```

Метод `charAt` возвращает символ строки, номер которого передан в качестве параметра.

```
charAt(<номер символа>)
```

> ***Внимание!***
>
> Символы в строке нумеруются, начиная с нуля.

```
c = "JavaScript".charAt(2);
```

После выполнения этого сценария в переменной `c` окажется символ `"v"` — третий по счету в строке `"JavaScript"`.

Метод `charCodeAt` возвращает код в кодировке Unicode символа строки, номер которого передан в качестве параметра.

```
charCodeAt(<номер символа>)
```

Довольно-таки бесполезный метод `concat` добавляет к текущей строке ту, что передана в качестве параметра, и возвращает результат.

```
concat(<добавляемая строка>)
```

Вместо него лучше использовать оператор объединения строк `+`.

Метод `fromCharCode` возвращает строку, составленную из символов, чьи коды Unicode переданы ей в качестве параметров.

```
String.fromCharCode(<код символа 1>, <код символа 2>, …)
```

Здесь мы видим, что метод `fromCharCode` может принять сколько угодно параметров. Подобные методы будут нам встречаться и у других объектов.

Отметим, что этот метод вызывается прямо у объекта `String`, а не у его экземпляра. Такие методы называются *статическими*.

```
str = String.fromCharCode(120, 121, 122);
```

После выполнения этого выражения в переменной `str` окажется строка `"xyz"`.

Метод `indexOf` возвращает номер позиции в текущей строке, в которой находится строка, переданная в качестве параметра.

```
indexOf(<искомая строка>[, <начальная позиция поиска>])
```

Первым параметром передается сама искомая строка (*подстрока поиска*). Вторым, необязательным, параметром может быть передан номер символа текущей строки, с которого начнется поиск подстроки поиска. По умолчанию поиск начинается с первого символа (то есть символа с номером 0). Если подстрока поиска не была найдена, возвращается значение –1.

```
n1 = "JavaScript".indexOf("a");
n2 = "JavaScript".indexOf("a", 2);
```

После выполнения этого сценария в переменной `n1` окажется число 1 (номер первого символа `"a"` в строке `"JavaScript"`), а в переменной `n2` — число 3 (номер второго символа `"a"` той же строки).

Метод `lastIndexOf` аналогичен методу `indexOf`, но ведет поиск подстроки, начиная с конца текущей строки.

```
lastIndexOf(<искомая строка>[, <начальная позиция поиска>])
n1 = "JavaScript".lastIndexOf("a");
n2 = "JavaScript".lastIndexOf("a", 2);
```

После выполнения этого сценария в переменной `n1` окажется число 3 (номер второго символа `"a"` в строке `"JavaScript"`), а в переменной `n2` — число 1 (номер первого символа `"a"` той же строки).

Метод `link` возвращает HTML-код, создающий гиперссылку. Текущая строка станет текстом этой гиперссылки, а строка, переданная в качестве параметра, — ее интернет-адресом.

```
link(<интернет-адрес гиперссылки>);
```

Метод `localeCompare` выполняет сравнение текущей строки и строки, переданной в качестве параметра, согласно заданным на компьютере клиента региональным установкам.

```
localeCompare(<сравниваемая строка>);
```

Если текущая строка "меньше" переданной, возвращается отрицательное число, если строки равны — 0, а если текущая строка "больше" переданной — положительное число.

Методы `match`, `replace` и `search` используются для работы с регулярными выражениями. Мы рассмотрим их в *главе 11*, когда станем говорить о работе с данными.

Метод `slice` возвращает заданный фрагмент текущей строки.

```
slice(<номер первого символа>[, <номер последнего символа>]);
```

Первым параметром передается номер первого символа возвращаемого фрагмента строки. Второй, необязательный, параметр задает номер его последнего символа. Если он опущен, возвращаемый фрагмент будет содержать все оставшиеся символы строки. Если же в качестве второго параметра указано отрицательное значение, заданное количество символов будет отсчитываться от конца строки.

```
var sourceStr = "JavaScript";
str1 = sourceStr.slice(4);
str2 = sourceStr.slice(4, 6);
str3 = sourceStr.slice(4, -2);
```

После выполнения этого сценария в переменной `str1` окажется строка `"Script"`, в переменной `str2` — строка `"Scr"`, а в переменной `str3` — строка `"Scrip"`.

Метод `split` разбивает текущую строку на фрагменты, формирует массив, элементами которого станут полученные фрагменты строки, и возвращает его.

```
split(<символ-разделитель>[, <предельное количество элементов массива>]);
```

Первым параметром методу передается символ-разделитель, по которому строка будет разбиваться на отдельные элементы. Вторым, необязательным, параметром передается максимальное количество элементов, которое будет содержать полученный массив; если он опущен, размер массива не ограничен.

```
var sourceStr = "Internet Explorer,Firefox,Opera";
arr1 = sourceStr.split(",");
arr2 = sourceStr.split(",", 2);
```

После выполнения данного сценария массив `arr1` будет содержать элементы `"Internet Explorer"`, `"Firefox"` и `"Opera"`, а массив `arr2` — элементы `"Internet Explorer"` и `"Firefox"`.

Метод `sub` заключает текущую строку в парный тег `<SUB>` (нижний индекс) и возвращает ее. Он не принимает параметров.

Метод `substr` возвращает заданный фрагмент текущей строки.

```
substr(<номер первого символа>[, <длина фрагмента>]);
```

Первым параметром передается номер первого символа возвращаемого фрагмента строки. Второй, необязательный, параметр задает длину возвращаемого фрагмента в символах. Если он опущен, возвращаемый фрагмент будет содержать все оставшиеся символы строки.

```
var sourceStr = "JavaScript";
str1 = sourceStr.substr(4);
str2 = sourceStr.substr(4, 2);
```

После выполнения этого сценария в переменной `str1` окажется строка `"Script"`, а в переменной `str2` — строка `"Sc"`.

Метод `substring` возвращает заданный фрагмент текущей строки. Это уже третий метод данной направленности.

```
substring(<номер первого символа>, <номер последнего символа>);
```

Первым параметром передается номер первого символа возвращаемого фрагмента строки. Второй параметр задает номер его последнего символа. При этом последний символ не будет включен в возвращаемый фрагмент.

```
var sourceStr = "JavaScript";
str = sourceStr.substr(4, 6);
```

После выполнения этого сценария в переменной `str1` окажется строка `"Sc"`.

Метод `sup` заключает текущую строку в парный тег `<SUP>` (верхний индекс) и возвращает ее. Он не принимает параметров.

Метод `toLocaleLowerCase` преобразует текущую строку в нижний регистр согласно заданным на компьютере клиента региональным установкам и возвращает ее. Он не принимает параметров.

Метод `toLocaleUpperCase` преобразует текущую строку в верхний регистр согласно заданным на компьютере клиента региональным установкам и возвращает ее. Он не принимает параметров.

Метод `toLowerCase` преобразует текущую строку в нижний регистр и возвращает ее. Он не принимает параметров.

Метод `toUpperCase` преобразует текущую строку в верхний регистр и возвращает ее. Он не принимает параметров.

Два абсолютно бесполезных метода `toString` и `valueOf` возвращают саму текущую строку.

Объект числа *Number*

Объект `Number` представляет обычное число — константу или переменную числового типа. Его экземпляр может быть создан как привычным для нас способом:

```
var num;
num = 12345;
```

так с помощью оператора `new`:

```
new Number(<числовая константа>);
```

Например:

```
var num;
num = new Number(12345);
```

Объект `Number` поддерживает несколько свойств, которые перечислены в табл. 4.11.

Все эти свойства принадлежат самому объекту `Number`, а не его экземплярам. Это так называемые *статические свойства*. Далее приведен пример обращения к свойству `MAX_VALUE`.

```
var max_number = Number.MAX_VALUE;
```

Метод `toExponential` возвращает экспоненциальное представление числа в строковом виде.

```
toExponential(<количество знаков после запятой>);
```

Таблица 4.11. *Свойства объекта* `Number`

Свойство	Описание
`MAX_VALUE`	Возвращает максимальное допустимое в JavaScript число. Оно примерно равно $1{,}79 \cdot 10^{308}$
`MIN_VALUE`	Возвращает минимальное допустимое в JavaScript число. Оно примерно равно $5 \cdot 10^{-324}$
`NaN`	Значение `NaN`
`NEGATIVE_INFINITY`	Математическая "минус бесконечность"
`POSITIVE_INFINITY`	Математическая "плюс бесконечность"

Единственный параметр этого метода задает количество знаков после запятой. Если заданное количество знаков меньше, чем присутствует в числе, оно будет округлено.

```
n = 12.34567;
sn1 = n.toExponential(10);
sn2 = n.toExponential(3);
```

После выполнения данного сценария в переменной `sn1` окажется строка `"1.2345670000E+1"`, а в переменной `sn2` — строка `"1.235E+1"`.

Метод `toFixed` возвращает обычное представление числа в строковом виде.

```
toFixed(<количество знаков после запятой>);
```

Единственный параметр этого метода задает количество знаков после запятой. Если заданное количество знаков меньше, чем присутствует в числе, оно будет округлено.

```
n = 12.34567;
sn1 = n.toFixed(10);
sn2 = n.toFixed(3);
```

После выполнения данного сценария в переменной `sn1` окажется строка `"12.3456700000"`, а в переменной `sn2` — строка `"12.346"`.

Метод `toLocaleString` возвращает строковое представление текущего числа с учетом заданных на компьютере клиента региональных установок. Он не принимает параметров.

Метод `toPrecision` возвращает обычное представление числа в строковом виде с заданным количеством знаков.

```
toPrecision(<общее количество знаков>);
```

Единственный параметр задает общее количество знаков числа (разделитель целой и дробной части числа также считается за знак). Если оно слишком

мало, число будет округлено. Если оно настолько мало, что его не хватит даже для вывода целой части числа, возвращенное число будет представлено в экспоненциальной нотации.

```
n = 12345.6789;
sn1 = n.toPrecision(10);
sn2 = n.toPrecision(8);
sn3 = n.toPrecision(4);
```

После выполнения данного сценария в переменной `sn1` окажется строка `"12345.6789"`, в переменной `sn2` — строка `"12345.68"`, а в переменной `sn3` — строка `"1E+4"`.

Здесь также поддерживаются два бесполезных метода `toString` и `valueOf`. Первый возвращает строковое представление текущего числа, второй — само это число.

Объект логического значения *Boolean*

Объект `Boolean` представляет логическое значение — константу или переменную логического типа. Экземпляр этого объекта может быть создан как привычным для нас способом:

```
var b;
b = (a == b);
```

так с помощью оператора `new`:

```
new Boolean(<логическая константа>);
```

Например:

```
var b;
b = new Boolean(a == b);
```

Объект `Boolean` поддерживает два метода, пользы от которых мало. Это все те же методы `toString` и `valueOf`. Первый метод возвращает строковое представление текущего значения объекта в виде строки `"true"` или `"false"`, второй — само его значение.

Объект значения даты *Date*

Объект `Date` служит для хранения значений даты и времени (время также включает миллисекунды). Такие значения "внутри" JavaScript представляют собой обычные числа, указывающие количество миллисекунд, прошедших с полуночи 1 января 1970 года. Отдельного типа даты JavaScript, в отличие от других языков программирования, не предусматривает.

Экземпляр объекта `Date` создается только с помощью оператора `new`. Формат его вызова таков:

```
new Date([<год>, <месяц>, <число>[, <часы>, <минуты>, <секунды>]])
```

Назначение параметров понятно из их названий. Так что при создании экземпляра объекта `Date` мы можем "собрать" дату и время из отдельных "частей". Если же ни один параметр не указан, созданный экземпляр получит значение текущей даты и времени.

```
d1 = new Date(2007, 12, 13);
d2 = new Date(2007, 12, 13, 15, 11, 40);
d3 = new Date();
```

Свойств объект `Date` не поддерживает. Зато методов у него очень много.

Метод `getDate` возвращает число из текущего значения даты. Он не принимает параметров.

Метод `getDay` возвращает номер месяца недели. Здесь число 0 обозначает воскресенье, число 1 — понедельник, 2 — вторник и т. д. Этот метод не принимает параметров.

Метод `getFullYear` возвращает номер года в виде числа из четырех цифр. Он не принимает параметров.

Метод `getHours` возвращает количество часов. Он не принимает параметров.

Метод `getMilliseconds` возвращает количество миллисекунд. Он не принимает параметров.

Метод `getMinutes` возвращает количество минут. Он не принимает параметров.

Метод `getMonth` возвращает номер месяца. Здесь число 0 обозначает январь, число 1 — февраль и т. д. Он не принимает параметров.

Метод `getSeconds` возвращает количество секунд. Он не принимает параметров.

Метод `getTime` возвращает "числовое" представление значения даты и времени в виде количества миллисекунд, прошедших с полуночи 1 января 1970 года. Он не принимает параметров.

Метод `getTimezoneOffset` возвращает разницу в минутах между временем по Гринвичу и локальным временем, принимая во внимание региональные настройки клиентского компьютера. Положительные значения соответствуют временны́м зонам, расположенным к западу от Гринвича, отрицательные — расположенным к востоку. Этот метод не принимает параметров.

Набор методов `getUTCDate`, `getUTCDay`, `getUTCFullYear`, `getUTCHours`, `getUTCMilliseconds`, `getUTCMinutes`, `getUTCMonth` и `getUTCSeconds` возвращают те же самые величины, что описанные ранее их "коллеги", но по Гринвичу.

Бесполезный для нас как JavaScript-программистов метод `getVarDate` возвращает значение даты в особом формате Windows, применяемом для про-

граммирования на *VBScript* (это другой язык написания Web-сценариев, на данный момент малопопулярный).

Такой же бесполезный метод `getYear` возвращает номер года, но в крайне неудобном формате из двух цифр. Кроме того, он по-разному обрабатывается разными Web-обозревателями. Лучше пользоваться более предсказуемым методом `getFullYear`.

Статический метод `parse` преобразует строковое представление даты и времени в числовое, принятое в JavaScript, которое и возвращает.

```
parse(<строковое представление даты и времени>)
```

В качестве строки, представляющей дату и время, можно использовать строку, возвращенную методами `toString`, `toGMTString` и `toLocaleString` (описаны далее).

```
var oldDate = new Date();
newDate = Date.parse(oldDate);
```

Смысла в приведенном примере немного, но он показывает, как вызывается метод `parse`.

Метод `setDate` изменяет число у значения даты и возвращает новое значение даты и времени.

```
setDate(<число>)
```

Видно, что новое значение числа передается в параметре этого метода.

Метод `setFullYear` изменяет год у значения даты и возвращает новое значение даты и времени.

```
setFullYear(<номер года>)
```

Метод `setHours` изменяет час у значения времени и возвращает новое значение даты и времени.

```
setHours(<час>)
```

Метод `setMilliseconds` изменяет миллисекунды у значения времени и возвращает новое значение даты и времени.

```
setMilliseconds(<количество миллисекунд>)
```

Метод `setMinutes` изменяет минуты у значения времени и возвращает новое значение даты и времени.

```
setMinutes(<минуты>)
```

Метод `setMonth` изменяет месяц у значения даты и возвращает новое значение даты и времени.

```
setMonth(<номер месяца>)
```

Не забываем, что нумерация месяцев в JavaScript начинается с нуля.

Метод `setSeconds` изменяет секунды у значения времени и возвращает новое значение даты и времени.

```
setSeconds(<секунды>)
```

Метод `setTime` задает новое значение даты и времени в виде количества миллисекунд, прошедших с полуночи 1 января 1970 года, и его же возвращает.

```
setTime(<количество миллисекунд>)
```

Набор методов `setUTCDate`, `setUTCFullYear`, `setUTCHours`, `setUTCMilliseconds`, `setUTCMinutes`, `setUTCMonth` и `setUTCSeconds` изменяют у значения даты и времени те же самые величины, что описанные ранее их "коллеги", но по Гринвичу.

Метод `setYear` полностью аналогичен `setFullYear`.

Метод `toDateString` возвращает строковое представление даты. Он не принимает параметров.

Метод `toGMTString` возвращает строковое представление даты и времени по Гринвичу. Он не принимает параметров.

> ***Внимание!***
>
> Метод `toGMTString` объявлен устаревшим. Вместо него следует использовать метод `toUTCString`, описанный далее.

Метод `toLocaleDateString` возвращает строковое представление даты с учетом региональных настроек клиентского компьютера. Он не принимает параметров.

Метод `toLocaleString` возвращает строковое представление даты и времени с учетом региональных настроек клиентского компьютера. Он не принимает параметров.

Метод `toLocaleTimeString` возвращает строковое представление времени с учетом региональных настроек клиентского компьютера. Он не принимает параметров.

Метод `toString` возвращает строковое представление даты и времени. Он не принимает параметров.

Метод `toTimeString` возвращает строковое представление времени. Он не принимает параметров.

Метод `toUTCString` возвращает строковое представление даты и времени по Гринвичу. Он не принимает параметров.

Статический метод `UTC` принимает "составные части" значения даты и времени и возвращает числовое значение даты и времени по Гринвичу.

```
UTC(<год>, <месяц>, <число>[, <часы>[, <минуты>[, <секунды>
↳[, <миллисекунды>]]]])
```

Назначение параметров этого метода понятно из их названий.

Метод `valueOf` возвращает текущее значение даты и времени.

Объект массива *Array*

Объект `Array` служит для представления массивов. Его экземпляр может быть создан как привычным для нас способом:

```
var arr;
arr = [1, 2, 3, 4];
```

так с помощью оператора `new`:

```
new Array([<список значений элементов массива, разделенных запятыми>]);
```

Например:

```
var arr;
arr = new Array(1, 2, 3, 4);
```

Объект `Array` поддерживает единственное свойство `length`, возвращающее размер массива.

А методов у этого объекта весьма много.

Метод `concat` создает новый массив, содержащий элементы текущего массива и значения всех переданных данному методу параметров, и возвращает его.

```
concat(<список добавляемых в новый массив элементов,
↳разделенных запятыми>);
```

В качестве параметров могут приниматься как обычные значения, так и массивы. Элементы этих массивов также будут добавлены в создаваемый массив.

```
var arr1, arr2;
arr1 = [1, 2, 3];
arr2 = arr1.concat(4, [5, 6]);
```

После выполнения этого сценария в переменной `arr2` окажется массив, содержащий элементы 1, 2, 3, 4, 5 и 6.

Метод `join` возвращает строку, полученную путем объединения значений всех элементов текущего массива; эти значения разделяются заданным символом-разделителем.

```
join(<символ-разделитель>);
var arr, s;
arr = [1, 2, 3];
s = arr1.join(",");
```

После выполнения этого сценария в переменной `s` окажется строка `"1,2,3"`.

Метод `pop` удаляет последний элемент текущего массива и возвращает его значение. Он не принимает параметров.

```
var arr, n;
arr = [1, 2, 3];
n = arr1.pop();
```

После выполнения этого сценария массив `arr` будет содержать два элемента — 1 и 2, — а в переменной `n` окажется число 3 — удаленный элемент массива `arr`.

Метод `push` помещает переданные ему в качестве параметров значения в конец текущего массива и возвращает последнее из них.

```
push(<список добавляемых значений, разделенных запятыми>);
var arr, n;
arr = [1, 2, 3];
n = arr.join(4, 5);
```

После выполнения этого сценария массив `arr` будет содержать пять элементов — 1, 2, 3, 4 и 5, — а в переменной `n` окажется число 5 — последний из добавленных элементов.

Метод `reverse` меняет порядок следования элементов текущего массива на противоположный и возвращает копию обновленного массива. Он не принимает параметров.

Метод `shift` удаляет первый элемент текущего массива и возвращает его значение. Он не принимает параметров.

```
var arr, n;
arr = [1, 2, 3];
n = arr1.shift();
```

После выполнения этого сценария массив `arr` будет содержать два элемента — 2 и 3, — а в переменной `n` окажется число 1 — удаленный элемент массива `arr`.

Метод `slice` извлекает заданные элементы текущего массива, формирует из них новый массив и возвращает его.

```
slice(<индекс первого элемента>[, <индекс последнего элемента>])
```

Назначение параметров понятно из их названия. Если последний параметр опущен, извлекаются все оставшиеся элементы массива.

```
var arr1, arr2, arr3;
arr1 = [1, 2, 3, 4, 5];
arr2 = arr1.slice(2);
arr3 = arr1.slice(2, 3);
```

После выполнения данного сценария массив `arr2` будет содержать элементы 3, 4 и 5, а массив `arr3` — элементы 3 и 4.

Метод `sort` сортирует текущий массив и возвращает его копию уже в отсортированном виде.

```
sort([<функция сортировки>])
```

По умолчанию этот метод представляет элементы массива в виде строк и сортирует эти строки. Но имеется возможность задать иные критерии сортировки. Для этого достаточно указать в качестве параметра `функцию сортировки`, которая будет выполнять сравнение двух элементов массива.

`Функция сортировки` должна удовлетворять двум требованиям. Во-первых, она должна принимать два параметра — два сравниваемых элемента массива. Во-вторых, она должны возвращать такие значения:

- отрицательное, если значение первого параметра меньше значения второго;
- 0, если значения обоих параметров равны;
- положительное, если значение первого параметра больше значения второго.

```
function sortFunction(a, b) {
  return a - b;
}
var arr;
arr = [5, 3, 2, 1, 4];
arr.sort(sortFunction);
```

После выполнения этого сценария массив `arr` будет отсортирован, причем элементы его будут сортироваться как числа.

Метод `splice` удаляет из текущего массива заданное количество элементов, возможно, добавляет в его конец новые элементы и возвращает копию уже измененного массива.

```
splice(<индекс первого удаляемого элемента>,
↳<количество удаляемых элементов>
↳[, <список добавляемых элементов, разделенных запятыми>])
```

Назначение параметров понятно из их названий.

```
var arr;
arr = [1, 2, 3, 4, 5];
arr.splice(2, 2, 6, 7);
```

После выполнения этого сценария массив `arr` будет содержать элементы 1, 2, 5, 6 и 7.

Метод `toLocaleString` возвращает строку, полученную путем объединения значений всех элементов текущего массива, разделенных запятой. При этом все числовые значения форматируются как денежные суммы согласно регио-

нальным настройкам клиентского компьютера. Этот метод не принимает параметров. Фактически он аналогичен вызову метода `join` с символом запятой в качестве параметра.

Метод `toString` возвращает строку, полученную путем объединения значений всех элементов текущего массива, разделенных запятой. Этот метод не принимает параметров. Фактически он аналогичен вызову метода `join` с символом запятой в качестве параметра.

Метод `unshift` добавляет в начало текущего массива значения, переданные в качестве параметров, и возвращает новый размер массива.

```
splice(<список добавляемых элементов, разделенных запятыми>)
var arr;
arr = [3, 4, 5];
arr.unshift(1, 2);
```

После выполнения этого сценария массив `arr` будет содержать элементы 1, 2, 3, 4 и 5.

Объект функции *Function*

Объект `Function` служит для представления функций. То есть, когда мы объявляем функцию

```
function someFunction(a, b) {
  return a + b;
}
```

мы создаем экземпляр `someFunction` объекта `Function`.

Мы также можем создать экземпляр этого объекта с помощью оператора `new`:

```
new Function(<набор строки, содержащей формальные аргументы>,
↳<тело функции в виде строки>)
```

Например:

```
someFunction = new Function("a", "b", "return a + b");
```

Впоследствии мы можем вызвать созданную таким образом функцию обычным способом.

У этого объекта представляют интерес, в основном, свойства. Все они перечислены в табл. 4.12.

Методы объект `Function` абсолютно бесполезны. Среди них можно выделить только `toString` и `valueOf`. Первый метод возвращает строковое представление функции, второй — ссылку на функцию.

Таблица 4.12. *Свойства объекта* `Function`

Свойство	Описание
`arguments`	Возвращает экземпляр объекта `Arguments` (его описание будет приведено далее в этой главе)
`caller`	Возвращает функцию, вызвавшую текущую функцию, в виде экземпляра объекта `Function`. Если функция была вызвана прямо из сценария, возвращается `null`
`length`	Возвращает количество параметров, принимаемых функцией

Объект массива аргументов *Arguments*

Единственный экземпляр объекта `Arguments` создается самим JavaScript при вызове функции и доступен только внутри ее тела. Он представляет массив, содержащий значения фактических параметров, полученных этой функцией.

Чтобы получить доступ к экземпляру объекта `Arguments`, следует воспользоваться свойством `arguments` объекта `Function`:

```
function someFunction(a, b) {
  return arguments[0] + arguments[1];
}
```

В приведенном примере мы обращаемся к обоим параметрам функции `someFunction` через экземпляр объекта `Arguments`, полученный путем вызова свойства `arguments` объекта `Function`.

Объект `Arguments` поддерживает два свойства. Свойство `callee` возвращает экземпляр объекта `Function`, представляющий текущую функцию. А знакомое нам по массивам свойство `length` возвращает количество переданных функции фактических параметров.

Объект *Math*

Объект `Math` представляет программистам-математикам тригонометрические и логарифмические функции, а также математические константы. Все это реализовано в виде большого набора статических свойств и методов. Так что создавать экземпляр данного объекта для использования всего этого богатства не нужно.

Все поддерживаемые объектом `Math` свойства перечислены в табл. 4.13.

А в табл. 4.14 приведен список всех методов, поддерживаемых объектом `Math`.

Таблица 4.13. *Свойства объекта* `Math`

Свойство	Возвращаемое значение
`E`	e
`LN10`	ln 10
`LN2`	ln 2
`LOG10E`	lg e
`LOG2E`	$\log_2 e$
`PI`	π
`SQRT1_2`	√0,5
`SQRT2`	√2

Таблица 4.14. *Методы объекта* `Math`

Метод	Описание
`abs(`*<параметр>*`)`	Возвращает абсолютное значение `параметра`
`acos(`*<параметр>*`)`	Возвращает арккосинус `параметра` в радианах
`asin(`*<параметр>*`)`	Возвращает арксинус `параметра` в радианах
`atan(`*<параметр>*`)`	Возвращает арктангенс `параметра` в радианах
`atan2(`*<X>*`, `*<Y>*`)`	Возвращает угол в радианах между горизонтальной осью и прямой, проведенной через начало координат и точку с координатами `X,Y`
`ceil(`*<параметр>*`)`	Возвращает ближайшее целое число, большее или равное `параметру`
`cos(`*<параметр>*`)`	Возвращает косинус `параметра`, заданного в радианах
`exp(`*<параметр>*`)`	Возвращает значение $e^{\text{параметр}}$
`floor(`*<параметр>*`)`	Возвращает ближайшее целое число, меньшее `параметра` или равное ему
`log(`*<параметр>*`)`	Возвращает натуральный логарифм `параметра`
`max(`*<параметр 1>*`, `*<параметр 2>*`)`	Возвращает наибольший из `параметров`
`min(`*<параметр 1>*`, `*<параметр 2>*`)`	Возвращает наименьший из `параметров`

Таблица 4.14 *(окончание)*

Метод	Описание
`pow(<основание>, <порядок>)`	Возвращает значение основание^{порядок}
`random()`	Возвращает псевдослучайное число от 0 включительно до 1 исключительно
`round(<параметр>)`	Возвращает значение `параметра`, округленное до ближайшего целого
`sin(<параметр>)`	Возвращает синус `параметра`, заданного в радианах
`sqrt(<параметр>)`	Возвращает квадратный корень из `параметра`
`tan(<параметр>)`	Возвращает тангенс `параметра`, заданного в радианах

Глобальный объект *Global*

Ранее мы познакомились со встроенными функциями языка JavaScript. Но на самом деле это не функции, а статические методы особого объекта `Global`. Причем объекта, так сказать, невидимого — нам не нужно указывать его имени при вызове нужной функции.

> ***НА ЗАМЕТКУ***
>
> Зачем разработчикам языка JavaScript понадобилось городить огород с глобальным объектом `Global`, непонятно. Могли бы прекрасно обойтись без него.

Кроме того, объект `Global` содержит три полезных для нас статических свойства. Свойство `Infinity` возвращает значение математической бесконечности (∞), относящееся к числовому типу. А свойства `NaN` и `undefined` возвращают знакомые нам одноименные значения.

Кроме рассмотренных нами, язык JavaScript поддерживает также объекты регулярного выражения и `RegExp`. Все они предназначены для работы с регулярными выражениями — весьма мощным средством обработки текста. Эти объекты мы рассмотрим в *главе 11*, когда будем говорить о работе с данными.

Еще один фундаментальный объект — `Object` — мы рассмотрим в конце этой главы, после разговора о создании пользовательских объектов. Так будет логичнее.

Пользовательские объекты

Еще до того, как приступить к рассмотрению встроенных объектов JavaScript, мы выяснили, что все объекты этого языка можно разделить на три вида. И те из них, что принадлежат к третьему виду, создаются самим программистом для собственных нужд. Что ж, ни один язык программирования, даже самый мощный, не способен предоставить средства на все случаи жизни…

Давайте же выясним, как создавать свои объекты в JavaScript. (Их еще называют *пользовательскими*.) Это не так уж и сложно, хоть и немного непривычно.

Создание простейших пользовательских объектов

Проще всего создать новый объект на основе встроенного объекта `Object`. Делается это следующим образом.

Давайте вспомним, как мы создавали экземпляры объектов. С помощью оператора `new`, — скажете вы. Вот так:

```
d = new Date(2007, 12, 13);
```

После оператора `new` мы указываем имя объекта, на основе которого создается экземпляр, и список параметров в скобках — эти параметры потом станут значениями его свойств. А "на выходе" получаем ссылку на только что созданный экземпляр объекта.

Но постойте, ведь это очень похоже на вызов функции! Да, так и есть. Экземпляр объекта создает оператор `new` совместно с особой функцией, называемой *конструктором*. Оператор `new` выполняет создание экземпляра объекта `Object`, а конструктор "наполняет" его необходимыми свойствами и методами.

Имя функции-конструктора должно совпадать с именем объекта. Более того, имя конструктора и есть имя объекта.

Все это значит, что для создания нового объекта мы должны всего лишь объявить его конструктор и записать в его теле выражения, объявляющие его свойства и методы. И сейчас мы выясним, как это делается.

Формат объявления функции-конструктора ничем не отличается от формата объявления обычной функции:

```
function <имя пользовательского объекта>
↳([<список параметров, разделенных запятыми>])
{
  <объявление свойств>
  <объявление методов>
}
```

Не забываем о том, что имя функции-конструктора должно совпадать с именем объекта, который мы хотим создать.

Формат выражения, создающего новое свойство пользовательского объекта, таков:

```
this.<имя свойства> = <значение свойства>;
```

То есть мы просто присваиваем новому свойству нужное значение. Ключевое слово `this` обозначает экземпляр объекта, создаваемый в данный момент конструктором. Если мы его опустим, то создадим простую переменную, а это не то, что нам нужно.

> ***Внимание!***
>
> Ключевое слово `this` имеет смысл только в теле функции-конструктора.

Формат выражения, создающего метод, аналогичен:

```
this.<имя метода> = <функция, реализующая этот метод>;
```

Здесь мы присваиваем свойству функцию, реализующую данный метод. (О том, что можем присваивать функции переменным, мы уже знаем, а свойство в этом смысле не очень отличается от переменной.) Разумеется, присваиваемая функция уже должна быть объявлена.

В функциях, реализующих методы создаваемого объекта, мы также можем использовать ключевое слово `this`. Собственно, даже не можем, а должны — ведь это единственный способ получить доступ к свойствам и методам этого объекта.

Давайте в качестве примера объявим объект, представляющий окружность. Этот объект будет содержать свойства `x`, `y` и `R`, представляющие ее горизонтальную и вертикальную координаты и радиус, и метод `getLength`, возвращающий длину окружности. Конструктор этого объекта будет выглядеть так:

```
function Round(px, py, pR) {
  this.x = px;
  this.y = py;
  this.R = pR;
  this.getLength = mGetLength;
}
```

Он принимает три параметра, которые станут значениями соответствующих свойств данного объекта. А функция `mGetLength`, реализующая метод `getLength`, будет выглядеть так:

```
function mGetLength() {
  return 2 * this.R * Math.PI;
}
```

НА ЗАМЕТКУ

Очень часто к именам формальных параметров любых функций, в том числе и конструкторов, добавляется буква "p" (от англ. parameter). Это делается, чтобы сразу отделить параметры от переменных и свойств.

Созданный таким образом объект мы можем использовать в своих сценариях:

```
var r;
r = new Round(100, 100, 10);
r.y = 200;
var rl = r.getLength();
delete r;
```

Как мы уже выяснили, конструкторы многих встроенных объектов позволяют опускать некоторые параметры (необязательные параметры). Мы можем предусмотреть такую же возможность в конструкторах своих объектов. Для этого достаточно проверить тип значения параметра с помощью уже знакомого нам оператора получения типа `typeof`. Если параметр опущен, этот оператор вернет строку `"undefined"`.

Давайте перепишем конструктор объекта `Round`, чтобы он позволял опускать значения параметров.

```
function Round(px, py, pR) {
  if (typeof(px) != "undefined")
    this.x = px
  else
    this.x = 100;
  if (typeof(py) != "undefined")
    this.y = py
  else
    this.y = 100;
  if (typeof(pR) != "undefined")
    this.R = pR
  else
    this.R = 10;
  this.getLength = mGetLength;
}
```

И пример использования обновленного объекта:

```
var r;
r = new Round(100, 50);
r.R = 20;
var rl = r.getLength();
delete r;
```

Наследование объектов

А теперь предположим, что мы решили создать объект `Cylinder`, представляющий цилиндр. Этот объект будет иметь свойства `x`, `y`, `R` и `h` (высота цилиндра) и методы `getLength` и `getVolume` (возвращает объем цилиндра). Мы можем написать соответствующий конструктор, объявляющий все эти свойства и методы, заново. А можем поступить по-другому.

Давайте посмотрим на список свойств и методов нового объекта `Cylinder`. Легко заметить, что большинство из них уже присутствует в созданном нами ранее объекте `Round`. Нельзя ли как-нибудь расширить этот объект, дав ему новые свойства и методы, характерные для цилиндра?

Можно! Мы создадим объект `Cylinder` на основе объекта `Round`, унаследовав все эти свойства и методы и добавив новые.

Но сначала введем несколько новых терминов. Назовем уже имеющийся объект, на основе которого нужно создать новый, *объектом-предком*, создаваемый на его основе объект — *объектом-потомком*, а сам процесс порождения нового объекта на основе старого — *наследованием*. Свойства и методы, которые объект-потомок получит "в наследство" от объекта-предка, пусть называются *унаследованными*. Этими терминами пользуются все профессиональные программисты.

Формат функции-конструктора для объекта-потомка будет таким:

```
function <имя объекта-потомка>
↳([<список параметров, разделенных запятыми>])
  {
    this.base = <имя объекта-предка>;
    this.base([<список параметров конструктора объекта-предка,
    ↳разделенных запятыми>]);
    <определение новых свойств>
    <определение новых методов>
    [<переопределение унаследованных свойств>]
    [<переопределение унаследованных методов>]
  }
<имя объекта-потомка>.prototype = new <имя объекта-предка>;
```

Первое выражение тела конструктора объекта-потомка присваивает функцию-конструктор объекта-предка свойству `base`. Это свойство будет содержать конструктор объекта-предка, который при вызове создаст все унаследованные свойства и методы и заполнит их значениями. Сам же вызов конструктора предка выполняется вторым выражением тела конструктора потомка.

Как создаются новые свойства и методы объекта-потомка, мы уже знаем. Кроме этого, мы можем переопределить значения унаследованных свойств и изменить унаследованные методы. Как изменить значение унаследованного свойства, понятно; для изменения же унаследованного метода достаточно присвоить ему другую функцию.

А теперь рассмотрим самое последнее выражение:

```
<имя объекта-потомка>.prototype = new <имя объекта-предка>;
```

Оно присваивает особому свойству `prototype` объекта-потомка экземпляр объекта-предка. Этим мы даем знать JavaScript, что данный объект является потомком данного объекта.

Осталось, собственно, создать объект `Cylinder`. Вот код его конструктора и функции `mGetValue`, реализующей метод `getValue`.

```
function Cylinder(px, py, pR, ph) {
  this.base = Round;
  this.base(px, py, pR);
  if (typeof(ph) != "undefined")
    this.h = ph
  else
    this.h = 2;
  this.getVolume = mGetVolume;
}
Cylinder.prototype = new Round;
function mGetVolume() {
  return Math.PI * this.R * this.R * this.h;
}
```

И пример его использования:

```
var c;
c = new Cylinder(100, 50, 20);
c.h = 5;
var ch = c.getVolume();
delete c;
```

Расширение встроенных объектов

Язык JavaScript также позволяет расширить набор свойств и методов своих встроенных объектов. Для этого используются выражения следующего формата:

```
<объект>.prototype.<имя свойства> = <значение свойства>;
<объект>.prototype.<имя метода> = <функция, реализующая этот метод>;
```

Давайте для примера расширим объект `String`, добавив метод, который будет заключать текущую строку в кавычки и возвращать ее, не изменяя при этом текущую строку. Для этого мы напишем такой сценарий:

```
function mGetQuoted() {
  return "\"" + this.valueOf() + "\"";
}
String.prototype.getQuoted = mGetQuoted;
```

После чего мы можем использовать этот метод так же, как и уже имеющиеся в объекте `String` методы:

```
var s1 = "JavaScript", s2;
s2 = s1.getQuoted();
```

Объект *Object* — общий предок

Все, абсолютно все объекты JavaScript — и встроенные, и пользовательские — являются потомками объекта `Object`, непосредственно или через других предков. А это значит, что все объекты JavaScript наследуют его свойства и методы и, как правило, переопределяют их.

Свойство `prototype` этого объекта уже знакомо нам и служит для задания объекта-предка. Также его можно использовать для получения ссылки.

Свойство `constructor` возвращает функцию-конструктор объекта и может использоваться, скажем, для определения, на основе какого объекта создан экземпляр.

```
if (r.constructor == Round) . . .
```

Метод `hasOwnProperty` принимает в качестве параметра имя свойства или метода в виде строки и возвращает `true`, если данное свойство (метод) поддерживается объектом, и `false` в противном случае.

```
hasOwnProperty(<имя свойства или метода в строковом виде>);
```

Например:

```
var c, out = "";
c = new Cylinder(100, 50, 20);
if (c.hasOwnProperty("R"))
  out += "Свойство R поддерживается\r\n";
if (c.hasOwnProperty("S"))
  out += "Свойство S поддерживается\r\n";
```

После выполнения данного сценария в переменной `out` окажется строка `"Свойство R поддерживается"`. В самом деле, объект `Cylinder` поддерживает свойство `R`, но не поддерживает свойство `S`.

Методы `toLocaleString`, `toString` и `valueOf` всегда переопределяются объектами-потомками. Они возвращают, соответственно, строковое представление экземпляра объекта с учетом региональных настроек клиентского компьютера, то же самое без учета региональных настроек и само значение экземпляра.

Комментарии

В *главе 2*, рассматривая язык HTML, мы познакомились с комментариями — особыми пометками, которые Web-дизайнер может оставлять для себя и своих коллег и которые не обрабатываются Web-обозревателем. Язык JavaScript тоже не отстает и предоставляет в распоряжение Web-программистов два *оператора вставки комментариев*.

Оператор вида

```
// <строка комментария>
```

позволяет вставить в код однострочный комментарий. Он может находиться, например, в конце какого-либо выражения:

```
a = b + c; // Складываем что-то с чем-то
```

Заметим, что такой комментарий должен помещаться после точки с запятой, обозначающей конец выражения.

А этот оператор позволяет вставить в код программы комментарий любого размера.

```
/*
  <текст комментария>
*/
```

Например:

```
/*
  В этом выражении мы складываем что-то с чем-то
  и помещаем результат куда-то
*/
a = b + c;
```

На самом деле, в комментариях следует писать что-то более осмысленное. Иначе толку от этих комментариев никакого.

Правила написания выражений

Напоследок давайте еще немного поговорим о выражениях JavaScript. А точнее, о правилах их написания.

JavaScript — язык весьма либеральный. Он позволяет нам писать выражения в несколько строк (в отличие от некоторых других языков программирования, например, Visual Basic). Зачастую так и пишут, перенося слишком длинные, не помещающиеся в окне текстового редактора выражения на несколько строк. Единственное — при этом нужно учитывать несколько правил, которые несложно запомнить.

- Разрешаются переносы строк между операндом и оператором. Под операндом здесь понимается константа, обращение к переменной или свойству, вызов функции или метода.
- 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 является символ точки с запятой (`;`). Собственно, об этом уже говорилось в начале этой главы.

В качестве примера давайте возьмем одно из выражений, опять же, 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;
```

Хотя это, конечно, перебор.

Но если мы выполним перенос строк внутри имени переменной `y2` —

```
y = y1 * y
2 + x1 * x2;
```

получим сообщение об ошибке.

На этом все о JavaScript.

Что дальше?

Вот мы и рассмотрели язык JavaScript в том объеме, который нам понадобится при написании Web-сценариев. Теоретический курс закончился. Пора приступать к практике.

В следующей главе мы рассмотрим, как пишутся Web-сценарии, как Web-обозреватель представляет страницу после загрузки и как получить доступ к ее элементам. Также мы рассмотрим написание сценариев, выполняющихся прямо во время загрузки страницы, — самую простую их разновидность.

Часть II

Базовые приемы JavaScript-программирования

Глава 5

Общие принципы написания Web-сценариев

HTML, CSS, JavaScript… Сколько же их, компьютерных языков! И все для чего-то служат. На HTML пишутся сами Web-страницы, CSS используется для написания таблиц стилей, а на JavaScript "разговаривают" Web-программисты, создающие Web-сценарии. И все эти языки мы знаем.

Изучить сразу три языка на протяжении всего трех глав книги — труд, достойный уважения. Но "мертвые", бесполезные знания ничего не стоят — имеют значение только знания, которые можно к чему-то приложить. "Мертвый капитал" — не капитал.

В *части II* этой книги мы начнем "оживлять" этот "мертвый капитал". Мы научимся писать Web-сценарии различного назначения, познакомимся, какие возможности предлагают нам как программистам Web-обозреватели, и рассмотрим решение множества типичных задач, с которыми мы как программисты обязательно столкнемся в будущем. То есть от теоретического курса Web-программирования перейдем к практическому.

И начнем мы с выяснения, как же пишутся Web-сценарии.

Как пишутся Web-сценарии

Давайте сразу же создадим небольшую Web-страничку, содержащую совсем простой Web-сценарий. Этот сценарий будет выводить текущую дату и время, отформатированную согласно региональным установкам клиентского компьютера.

Вот HTML-код этой страницы:

```
<HTML>
  <HEAD>
    <TITLE>Пример Web-сценария</TITLE>
```

```
  </HEAD>
  <BODY>
    <P>
      <SCRIPT>
        var d = new Date();
        document.write(d.toLocaleString());
      </SCRIPT>
    </P>
  </BODY>
</HTML>
```

Сохраним его в файле 5.1.htm и откроем в Web-обозревателе. То, что нам покажет Internet Explorer, видно на рис. 5.1.

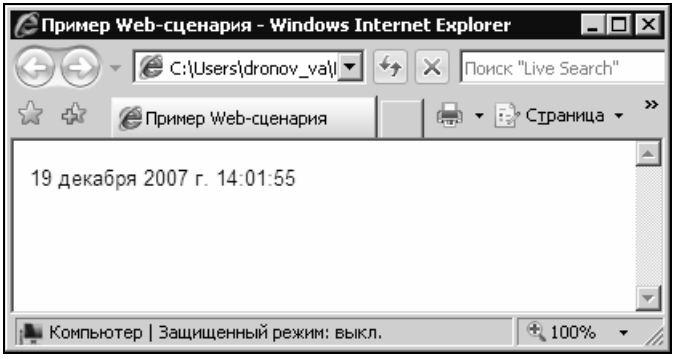

Рис. 5.1. Web-страница со сценарием, выводящим сегодняшнюю дату и время

То же самое покажут нам Opera и Firefox. А это значит, что наша первая "живая" страничка работает.

Теперь подробно разберем, что мы только что написали.

Сразу видно, что код Web-сценария помещается в парный тег `<SCRIPT>`. Встретив этот тег, Web-обозреватель поймет, что здесь присутствует Web-сценарий и его следует выполнить, а не выводить на экран.

> ***НА ЗАМЕТКУ***
>
> Тег `<SCRIPT>` поддерживает необязательный атрибут `TYPE`, имеющий в нашем случае чисто теоретический интерес. В качестве его значения задается тип сценария, содержащегося в этом теге, фактически — язык программирования, на котором он написан. Значение `"text/javascript"` задает язык JavaScript; этот же язык выбирается, если данный атрибут опущен. Internet Explorer также поддерживает значения `"text/vbscript"` и `"text/vbs"` этого атрибута, задающие язык VBScript.

Сценарий выполняется в том месте кода страницы, где присутствует. Так, мы поместили наш сценарий внутри тега `<P>`, создающего абзац, и именно в этом месте он будет выполнен. Если сценарий выводит на страницу какой-то текст (как в нашем случае), он выведет его именно в этом месте. Наш сценарий выведет строковое значение даты и времени внутри тега `<P>`, создав содержимое этого, изначально пустого, тега.

Сам же сценарий, как мы видим, создает экземпляр объекта `Date`, содержащий текущую дату и время (поскольку его конструктор не имеет параметров — см. приведенный ранее код). Далее он вызывает метод `write` экземпляра `document` объекта `HTMLDocument` (о нем вы поговорим позже) и передает ему в качестве параметра строковое представление даты и времени, учитывающее региональные настройки. (Строковое представление даты и времени с учетом региональных настроек возвращается методом `toLocaleString` объекта `Date` — это мы помним из *главы 4*.)

Объект `HTMLDocument` представляет саму Web-страницу. Экземпляр `document` этого объекта создается самим Web-обозревателем во время загрузки страницы. А метод `write` объекта `HTMLDocument` выводит переданную ему в качестве параметра строку в то место, где встретилось вызвавшее этот метод выражение.

Подробнее об объекте `HTMLDocument` и поддерживаемых им свойствах и методах мы поговорим потом. А пока вернемся к нашему сценарию.

Написанный нами сценарий целиком выполняется во время загрузки страницы. Это самая простая разновидность Web-сценариев — *загрузочные* сценарии.

Давайте немного изменим код нашей страницы, чтобы он выглядел вот так:

```
<HTML>
  <HEAD>
    <TITLE>Пример Web-сценария</TITLE>
    <SCRIPT>
      var d = new Date();
    </SCRIPT>
  </HEAD>
  <BODY>
    <P>
      <SCRIPT>
        document.write(d.toLocaleString());
      </SCRIPT>
    </P>
  </BODY>
</HTML>
```

Здесь мы для достижения того же эффекта применили два сценария. Первый сценарий помещается в секции заголовка страницы (в теге <HEAD>) и создает экземпляр объекта Date. Второй сценарий помещается в теге <P> и выполняет собственно вывод даты и времени.

Из *главы 4* мы знаем, что созданная в одном из присутствующих на странице сценариев переменная после этого останется доступной для всех остальных сценариев, присутствующих на этой же странице. Так что номер, что мы проделали ранее, сработает — объявленная в первом сценарии переменная d будет доступна и во втором сценарии.

В секции заголовка часто помещают сценарии, объявляющие переменные, которые должны быть доступны во всех сценариях данной страницы, или выполняющие какие-либо начальные установки. Это хороший стиль Web-программирования.

А теперь давайте рассмотрим еще один сценарий:

```
<HTML>
  <HEAD>
    <TITLE>Пример Web-сценария</TITLE>
  </HEAD>
  <BODY>
    <P ID="par">Красный текст.</P>
    <SCRIPT>
      par.style.color = "#FF0000";
    </SCRIPT>
  </BODY>
</HTML>
```

Этот сценарий получает доступ к тегу <P>, содержащему единственный абзац, и задает для него красный цвет текста. Но как он это делает?

Внутреннее представление страницы. Document Object Model (DOM)

А вот здесь мы вплотную подошли к внутреннему представлению страницы Web-обозревателем. Без понимания, как Web-обозреватель "раскладывает по полочкам" страницу и все ее элементы, заниматься серьезным Web-программированием невозможно.

Ранее мы выяснили, что сама страница представляется Web-обозревателем как экземпляр особого объекта HTMLDocument. Этот объект реализуется самим Web-обозревателем и является внешним по отношению к исполняющей

среде JavaScript, поэтому называется *внешним*. Поддерживаемые им свойства и методы позволяют получать различные сведения о странице (например, ее интернет-адрес) и выполнять над ней различные действия. (Так, ранее мы использовали метод `write` этого объекта для вывода произвольного текста на страницу.)

Во время загрузки страницы Web-обозреватель создает представляющий ее экземпляр объекта `HTMLDocument` и сохраняет его в переменной `document`. Эта переменная доступна во всех сценариях, присутствующих на данной странице. Поскольку она создается самим Web-обозревателем без участия программиста, то, по аналогии с встроенными функциями языка JavaScript, называется *встроенной*.

По мере загрузки страницы Web-обозреватель считывает код HTML, определяющий различные ее элементы, и также создает их внутреннее представление в виде экземпляров соответствующих объектов. Так, для абзаца создается экземпляр объекта `HTMLParagraphElement`, для гиперссылки — экземпляр объекта `HTMLLinkElement`, для изображения — экземпляр объекта `HTMLImageElement`, для таблицы — `HTMLTableElement` и др. Все эти объекты поддерживают свойства и методы, предназначенные для получения сведений обо всех этих элементах страницы и манипуляциях с ними.

Кроме того, внутреннее представление создается также для текстового содержимого тегов. Текстовое содержимое каждого тега представляется как экземпляр объекта `Text`.

Все объекты, представляющие элементы страницы, в том числе и объект `Text`, являются потомками объекта `HTMLElement`. Этот объект представляет свойства и методы, общие для всех элементов страницы: имя создающего его тега, имя привязанного к нему стилевого класса и пр.

Совокупность экземпляров различных объектов, представляющих саму страницу и все ее элементы, называется *DOM* (сокращение от *Document Object Model* — объектная модель документа). Все современные Web-обозреватели, поддерживающие Web-сценарии, также поддерживают и DOM, ведь именно она позволяет манипулировать содержимым страницы из сценариев.

Только вот поддерживают они DOM по-разному. Internet Explorer никак не обрабатывает "промежутки" между тегами, не заполненные текстом. Давайте в качестве примера рассмотрим фрагмент содержимого нашей первой страницы, созданной еще в *главе 2*.

```
<BODY>
  <P>Это простейшая Web-страничка, созданная в стандартном
  <EM>Блокноте</EM> и отображенная в <EM>Microsoft
  Internet Explorer</EM>.</P>
</BODY>
```

Internet Explorer создаст в памяти следующее представление в виде экземпляров соответствующих объектов (`HTMLElement` и `Text`):

```
Тег <BODY>
  Тег <P>
    Текст "Это простейшая Web-страничка, созданная в стандартном "
    Тег <EM>
      Текст "Блокноте"
    Текст " и отображенная в "
    Тег <EM>
      Текст "Microsoft Internet Explorer"
    Текст "."
```

А Opera и Firefox посчитают "промежутки" между тегами за пустой текст и создадут такое представление ("пустые" текстовые экземпляры объекта `Text` выделены полужирным шрифтом):

```
Тег <BODY>
  "Пустой" текст
  Тег <P>
    Текст "Это простейшая Web-страничка, созданная в стандартном "
    Тег <EM>
      Текст "Блокноте"
    Текст " и отображенная в "
    Тег <EM>
      Текст "Microsoft Internet Explorer"
    Текст "."
  "Пустой" текст
```

Кроме того, у разных Web-обозревателей различается представление самих Web-сценариев. Internet Explorer не создает представления для содержимого тега `<SCRIPT>`, но создает представление для самого этого тега. А Opera и Firefox создают представление и для того, и для другого.

Какой из этих подходов правилен и более точно соответствует стандартам DOM, неизвестно. Но нужно иметь эти различия в виду.

Именование элементов страницы

Ранее мы выяснили, что экземпляр объекта `HTMLDocument`, представляющего страницу, доступен из всех сценариев этой страницы под именем `document`. Это значит, что мы можем получить к нему доступ без всяких дополнительных действий.

Однако, в отличие от экземпляра объекта `HTMLDocument`, представляющего саму Web-страницу, экземпляры объектов, представляющие отдельные элементы этой страницы, изначально не доступны в сценариях. Они хранятся во внутренних структурах, формируемых Web-обозревателем в памяти, и добраться до них так же просто, как до экземпляра `document`, не получится.

Выход из положения — дать им имя.

Для именования элемента страницы достаточно вставить в определяющий его тег необязательный атрибут `ID` и в качестве значения этого атрибута указать нужное имя. Атрибут `ID` поддерживается всеми видимыми тегами.

Пример элемента страницы с именем:

```
<P ID="par">Красный текст.</P>
```

Это фрагмент нашей третьей по счету страницы, созданной на протяжении этой главы. Здесь мы дали тегу `<P>`, формирующему абзац "Красный текст.", имя `par`.

К именам элементов страниц предъявляются достаточно простые требования. Они должны содержать только латинские буквы, цифры и знаки подчеркивания (_), причем первым символом в имени должна быть буква или знак подчеркивания.

Имена элементов страницы должны быть уникальными в пределах данной Web-страницы. Если же мы создадим на странице два элемента с одинаковыми именами, Web-обозреватель не сможет обработать эту страницу правильно.

> ***Внимание!***
>
> Имена следует давать только тем элементам страницы, к которым будет осуществляться доступ из сценариев. Остальные элементы страницы прекрасно обойдутся без имен.

> ***На заметку***
>
> Как мы выяснили в *главе 3*, стили-селекторы привязываются к тегам также с помощью атрибута `ID`. Часто эта особенность данного атрибута используется для того, чтобы привязать к какому-либо элементу страницы стиль-селектор и одновременно дать ему имя для доступа из сценариев.

Получение доступа к элементу страницы

Хорошо, имя элементу страницы мы дали. Но как теперь получить к нему доступ из сценариев?

Здесь есть три способа. Давайте рассмотрим их.

Прямой доступ по имени

Самый простой способ добраться до элемента страницы в сценарии — обратиться прямо к нему по его имени. Вот так:

<имя элемента страницы>.<имя свойства>

и

<имя элемента страницы>.<имя метода>

↳([*<список фактических параметров, разделенных запятыми>*])

То есть используем такой же синтаксис, как и для доступа к обычному экземпляру объекта JavaScript, встроенного или пользовательского. (Об использовании объектов и их экземпляров см. *главу 4*.)

Например:

```
par.style.color = "#FF0000";
```

Здесь мы обратились к свойству `style` элемента страницы `par`. Это свойство хранит экземпляр объекта `CSSRule`, который представляет результирующий стиль данного элемента страницы, вычисленный на основе всех привязанных к нему стилей согласно правилам каскадности (см. *главу 3*). Объект `CSSRule` поддерживает свойство `color`, соответствующее одноименному атрибуту стиля и задающее цвет элемента страницы. Этому-то свойству мы и присваиваем значение `"#FF0000"` — RGB-код красного цвета.

Как правило, именно этот способ доступа к элементам страницы используется в большинстве случаев из-за его простоты и наглядности. Но мы им пользоваться не будем. И вот почему…

> ***Внимание!***
>
> Прямой доступ к элементам страницы по имени не соответствует стандартам DOM, объявлен устаревшим и не рекомендованным к использованию, хоть и поддерживается всеми Web-обозревателями. Вместо него рекомендуется использовать метод `getElementById` объекта `HTMLDocument` (см. далее).

Доступ через коллекции

Второй способ получить доступ к элементу страницы в сценарии — использовать коллекции. *Коллекция* JavaScript — это ассоциативный массив (об ассоциативных массивах см. *главу 4*), представляющий список экземпляров каких-либо объектов и сам являющийся особым объектом.

Объект `HTMLDocument` поддерживает достаточно большое количество коллекций, доступных через одноименные свойства. Все они перечислены в табл. 5.1.

Таблица 5.1. *Коллекции объекта* `HTMLDocument`

Коллекция	Описание
`all`	Все элементы страницы, включая теги `<HTML>`, `<HEAD>`, `<TITLE>` и `<BODY>`. Стандартами DOM объявлена устаревшей и не рекомендована к использованию, хотя поддерживается всеми Web-обозревателями
`anchors`	Все якоря страницы
`applets`	Все элементы ActiveX (см. *главу 9*)
`embeds`	Все модули расширения (см. *главу 9*)
`forms`	Все Web-формы (см. *главу 12*)
`images`	Все графические изображения
`links`	Все гиперссылки, включая горячие области
`scripts`	Все Web-сценарии. Поддерживается только Internet Explorer и Opera
`styleSheets`	Все таблицы стилей

Все эти коллекции являются экземплярами объекта `HTMLCollection`, за исключением `styleSheets` (`StyleSheetList`). Все они поддерживают свойство `length`, возвращающее *размер* коллекции — количество элементов в ней.

Мы можем получить доступ к нужному элементу коллекции так же, как к элементу обычного ассоциативного массива — по строковому индексу, который будет совпадать с именем элемента страницы. Например, чтобы получить доступ к абзацу `par`, мы напишем такое выражение:

```
parObj = document.all["par"];
```

А если мы знаем числовой индекс нужного элемента страницы, можем использовать его:

```
someObj = document.all[3];
```

Только учтем, что нумерация элементов коллекции, как и элементов массива, начинается с нуля.

Мы можем "обойти" все элементы страницы определенного типа, "перебирая" все элементы соответствующей коллекции. Так, чтобы "обойти" все гиперссылки на странице, мы просмотрим коллекцию `links`:

```
var i, linkObj;
for (i = 0; i < document.links.length; i++) {
  linkObj = document.links[i];
  // Выполняем манипуляции с полученной гиперссылкой
}
```

Обычно доступ через коллекции не используется. Он может быть полезен только в особых случаях, например, когда имя нужного элемента страницы не задано жестко, а вычисляется в сценарии. В остальных случаях рекомендуется использовать метод `getElementById` объекта `HTMLDocument` (см. далее).

> ***Внимание!***
>
> Коллекцией `all` рекомендуется пользоваться только в крайних случаях. Поскольку она содержит все элементы страницы, поиск в ней элемента по его имени может выполняться довольно долго. К тому же, как сказано в табл. 5.1, эта коллекция объявлена стандартами DOM устаревшей и не рекомендованной к использованию.

Доступ с помощью свойств и методов DOM

Последний способ доступа к элементу страницы, который мы рассмотрим, — пожалуй, самый громоздкий, но полностью соответствующий стандартам DOM. Это использование особых свойств и методов DOM, которые поддерживаются объектами `HTMLDocument` и `HTMLElement`.

Начнем с объекта `HTMLDocument`. Он поддерживает свойство `body`, возвращающее ссылку на экземпляр объекта `HTMLBodyElement`, соответствующий тегу `<BODY>`, то есть секции тела страницы. Получив этот экземпляр, мы можем просмотреть все элементы страницы, пользуясь свойствами и методами объекта `HTMLElement`, которые мы рассмотрим далее. (Объект `HTMLBodyElement` является потомком объекта `HTMLElement` и наследует все его свойства и методы.)

Очень важный метод `getElementById` этого же объекта позволяет получить ссылку на экземпляр объекта `HTMLElement`, соответствующего элементу страницы с заданным именем.

`getElementById(`*`<имя элемента страницы>`*`)`

`Имя элемента страницы` передается в виде строки.

```
var parObj = document.getElementById("par");
```

> ***Внимание!***
>
> Именно метод `getElementById` объекта `HTMLDocument` стандарты DOM рекомендуют использовать для получения доступа к элементу страницы. И, в основном, именно им мы будем в дальнейшем пользоваться при написании всех сценариев.

Мы уже знаем, что имя элемента страницы задается атрибутом `ID`, поддерживаемым практически всеми тегами. Но для этой же цели может использоваться атрибут `NAME`, имеющий аналогичное назначение.

```
<P NAME="par">Красный текст.</P>
```

Так вот, для получения доступа к элементу страницы, имя которого задано атрибутом `NAME`, служит метод `getElementsByName`.

```
getElementsByName(<имя элемента страницы>)
```

Сразу же отметим, что этот метод возвращает массив экземпляров объекта `HTMLElement`, соответствующих элементам страницы с заданным именем. Если на странице существует единственный элемент с таким именем, возвращенный массив будет иметь один элемент.

```
var parObjs = document.getElementsByName("par");
var parObj = parObjs[0];
```

Еще один полезный метод — `getElementsByTagName`. Он возвращает массив экземпляров объекта `HTMLElement`, содержащий все элементы страницы, которые сформированы с помощью заданного тега.

```
getElementsByTagName(<имя тега>)
```

Имя тега задается в строковом виде без символов < и >.

```
var parObjs = document.getElementsByTagName("P");
var parObj = parObjs[0];
```

Теперь обратимся к объекту `HTMLElement`. Рассмотрим его свойства и методы в порядке от самых интересных для нас к самым бесполезным.

Свойство `childNodes` возвращает ссылку на коллекцию `childNodes`. Эта коллекция является экземпляром объекта `NodeList` и содержит все элементы страницы, которые являются дочерними по отношению к текущему и при этом вложены непосредственно в него.

```
var parObj = document.body.childNodes[0];
```

После выполнения этого сценария в переменной `parObj` окажется ссылка на экземпляр объекта `HTMLElement`, представляющий абзац `par` (так как он является первым дочерним элементов в секции тела страницы — теге `<BODY>`).

Свойство `firstChild` возвращает ссылку на экземпляр объекта `HTMLElement`, представляющий элемент страницы, который является первым дочерним для текущего элемента. Если текущий элемент не содержит дочерних элементов, возвращается `null`.

```
var parObj = document.body.firstChild;
var textObj = parObj.firstChild;
```

После выполнения этого сценария в переменной `parObj` окажется ссылка на абзац `par` (он у нас первый дочерний элемент тега `<BODY>`), а в переменной `textObj` — экземпляр объекта `Text`, представляющий текстовое содержимое этого абзаца (он у нас вообще содержит один дочерний элемент — сам текст).

Свойство `lastChild` возвращает ссылку на экземпляр объекта `HTMLElement`, представляющий элемент страницы, который является последним дочерним для текущего элемента. Если текущий элемент не содержит дочерних элементов, возвращается `null`.

Свойство `nextSibling` возвращает ссылку на экземпляр объекта `HTMLElement`, представляющий элемент страницы, который является следующим "соседом" текущего элемента в его родителе. Если текущий элемент — последний в его родителе, возвращается `null`.

```
var parObj = document.body.firstChild;
var scriptObj = parObj.nextSibling;
```

После выполнения этого сценария в переменной `scriptObj` окажется ссылка на сам сценарий (тег `<SCRIPT>`).

Свойство `previousSibling` возвращает ссылку на экземпляр объекта `HTMLElement`, представляющий элемент страницы, который является предыдущим "соседом" текущего элемента в его родителе. Если текущий элемент — первый в его родителе, возвращается `null`.

```
var scriptObj = document.body.lastChild;
var parObj = parObj.previousSibling;
```

После выполнения этого сценария в переменной `parObj` окажется ссылка на абзац `par`.

Свойство `parentNode` возвращает ссылку на элемент страницы, являющийся родителем для текущего.

```
var bodyObj = par.parentNode;
```

После выполнения этого сценария в переменной `bodyObj` окажется ссылка на секцию тела страницы (тег `<BODY>`).

Объект `HTMLElement` поддерживает метод `getElementsByTagName`. Работает он так же, как аналогичный метод объекта `HTMLDocument`.

Метод `hasChildNodes` возвращает `true`, если текущий элемент страницы имеет дочерние элементы, и `false` в противном случае. Он не принимает параметров.

И еще четыре довольно-таки бесполезных свойства.

Свойство `all` возвращает ссылку на уже знакомую нам коллекцию `all`. Только теперь эта коллекция будет содержать только элементы, являющиеся дочерними по отношению к текущему.

Свойство `children` по назначению аналогично свойству `childNodes` и возвращает коллекцию `children`.

Свойство `parentElement` возвращает родительский элемент для текущего элемента страницы и полностью аналогично рассмотренному ранее свойству `parentNode`.

> ***Внимание!***
>
> Свойства `all`, `children` и `parentElement` объекта `HTMLElement` и соответствующие им коллекции поддерживаются только Internet Explorer и Opera.

Свойство `ownerDocument` возвращает ссылку на экземпляр объекта `HTMLDocument`, представляющий саму страницу, в которой находится текущий элемент. В общем, то же самое, что и переменная `document`.

В качестве примера давайте напишем страницу, которая с помощью особого сценария будет подсчитывать количество находящихся на ней элементов и выводить на экран. Ее код приведен далее.

```
<HTML>
  <HEAD>
    <TITLE>Коллекции</TITLE>
  </HEAD>
  <BODY>
    <P>Это простейшая Web-страничка, созданная в стандартном
    <EM>Блокноте</EM> и отображенная в <EM>Microsoft
    Internet Explorer</EM>.</P>
    <SCRIPT>
      function getCountOfElements(pContainerObj) {
        var count = 0, elementsCol, colLength;
        if (pContainerObj.hasChildNodes()) {
          elementsCol = pContainerObj.childNodes;
          colLength = elementsCol.length;
          for (var i = 0; i < colLength; i++) {
            count++;
            count += getCountOfElements(elementsCol[i]);
          }
        }
        return count;
      }
      document.write("<P>Количество элементов: " +
      ↳getCountOfElements(document.body).toString() + "</P>");
    </SCRIPT>
  </BODY>
</HTML>
```

В сценарии, подсчитывающем количество элементов страницы, мы использовали функцию, вызывающую саму себя рекурсивно. (О рекурсии говорилось в *главе 4*.) После ее открытия в Internet Explorer она покажет, что содержит 9 элементов, при открытии в Opera и Firefox — 12. Opera и Firefox посчитают за отдельные элементы "промежутки" между тегами `<BODY>` и `<P>` и `</P>` и `<SCRIPT>`, а также содержимое тега `<SCRIPT>` — сам сценарий.

Доступ к элементу страницы с помощью свойств и методов DOM применяется, опять же, в специальных случаях. Так, в приведенном ранее HTML-коде страницы эти свойства и методы использует сценарий, выполняющий таким способом "обход" всей страницы и подсчитывающий все ее элементы. Другим способом эту задачу вряд ли возможно выполнить.

Единственное исключение — метод `getElementById` объекта `HTMLDocument`, который стандарты DOM рекомендуют для получения доступа к элементам страницы и который отныне будем использовать мы. Надо сразу привыкать делать все согласно стандартам.

Особенности работы с таблицами

Таблица HTML представляется Web-обозревателем как экземпляр объекта `HTMLTableElement`. Этот объект является потомком объекта `HTMLElement`, а значит, наследует все его свойства и методы. Эти свойства и методы нам уже знакомы, и мы можем ими пользоваться для доступа к любому элементу таблицы — названию, секции, строке, ячейке или ее содержимому. (Подробнее о таблицах HTML было рассказано в *главе 2*.)

Предположим, что мы создали таблицу и дали ей имя `tab`. В этой таблице мы явно определили название и секции заголовка, тела и "поддона".

```
<TABLE ID="tab">
  <CAPTION>Таблица</CAPTION>
  <THEAD>
    <TR><TH>Столбец 1</TH><TH>Столбец 2</TH><TH>Столбец 3</TH></TR>
  </THEAD>
  <TBODY>
    <TR><TD>1</TD<TD>2</TD><TD>3</TD></TR>
    <TR><TD>4</TD<TD>5</TD><TD>6</TD></TR>
    . . .
  </TBODY>
  <TFOOT>
    <TR><TD>Итог 1</TD><TD>Итог 2</TD><TD>Итог 3</TD></TR>
  </TFOOT>
</TABLE>
```

После этого выполним выражение

```
var tabObj = document.getElementById("tab");
```

Оно поместит в переменную `tabObj` экземпляр объекта `HTMLTableElement`, представляющий данную таблицу.

Для доступа к названию и секциям этой таблицы мы можем использовать коллекцию `childNodes`. Как мы помним, она содержит все потомки текущего элемента страницы.

```
var captionObj = tabObj.childNodes[0];
var headObj = tabObj.childNodes[1];
var bodyObj = tabObj.childNodes[2];
var footObj = tabObj.childNodes[3];
```

Этот сценарий поместит:

- в переменную `captionObj` — экземпляр объекта `HTMLTableCaptionElement`, представляющий название таблицы (название является первым потомком таблицы; и не забываем, что нумерация элементов коллекции начинается с нуля);
- в переменную `headObj` — экземпляр объекта `HTMLTableSectionElement`, представляющий секцию таблицы, в данном случае секцию заголовка (второй потомок таблицы);
- в переменную `bodyObj` — экземпляр объекта `HTMLTableSectionElement`, представляющий секцию тела таблицы (третий потомок);
- в переменную `footObj` — экземпляр объекта `HTMLTableSectionElement`, представляющий секцию "поддона" таблицы (четвертый потомок).

Все эти объекты порождены от объекта `HTMLElement` и наследуют все его свойства и методы. В чем мы сейчас и убедимся.

Получив нужную секцию, мы можем через всю ту же коллекцию `childNodes` добраться до нужной строки, содержащейся в этой секции.

```
var firstHeadRowObj = headObj.firstChild;
```

Это выражение поместит в переменную `firstHeadRowObj` экземпляр объекта `HTMLTableRowElement`, представляющий строку таблицы, в данном случае первую и единственную строку ее секции заголовка.

```
var secondBodyRowObj = bodyObj.childNodes[1];
```

А это выражение поместит в переменную `secondBodyRowObj` экземпляр объекта `HTMLTableRowElement`, представляющий вторую строку секции тела таблицы.

Пользуясь все той же коллекцией `childNodes`, мы можем добраться до любой ячейки полученной строки

```
var cellObj = firstHeadRowObj.childNodes[0];
```

Это выражение поместит в переменную `cellObj` экземпляр объекта `HTMLTableCellElement`, представляющий ячейку таблицы, в данном случае первую ячейку единственной строки секции заголовка.

```
var cellObj = secondBodyRowObj.childNodes[2];
```

А это выражение поместит в переменную `cellObj` экземпляр объекта `HTMLTableCellElement`, представляющий третью ячейку второй строки секции тела таблицы.

Теперь предположим, что мы поленились и не стали создавать в таблице `tab` ни названия, ни секций. (Собственно, Web-дизайнеры так часто и делают, поскольку разбиение таблицы на секции никаких преимуществ на данный момент не дает, а название таблицы можно оформить по-другому, например, в виде обычного абзаца.)

```
<TABLE ID="tab">
  <TR><TH>Столбец 1</TH><TH>Столбец 2</TH><TH>Столбец 3</TH></TR>
  <TR><TD>1</TD<TD>2</TD><TD>3</TD></TR>
  <TR><TD>4</TD<TD>5</TD><TD>6</TD></TR>
  . . .
  <TR><TD>Итог 1</TD><TD>Итог 2</TD><TD>Итог 3</TD></TR>
</TABLE>
```

В этом случае Web-обозреватель предположит, что все строки таблицы входят в секцию тела, которую Web-дизайнер почему-то не создал явно, и создаст эту секцию сам. Созданная таким образом секция тела таблицы станет единственным потомком самой таблицы.

```
var tabObj = document.getElementById("tab");
var bodyObj = tabObj.firstChild;
```

После выполнения этого сценария в переменной `bodyObj` окажется экземпляр объекта `HTMLTableSectionElement`, представляющий неявно созданную Web-обозревателем секцию тела таблицы. От него-то мы и начнем "плясать", если пожелаем получить доступ к строкам и ячейкам таблицы.

```
var rowObj = bodyObj.childNodes[1];
var cellObj = rowObj.childNodes[2];
```

После выполнения этих выражений в переменной `rowObj` окажется экземпляр объекта `HTMLTableRowElement`, представляющий вторую строку таблицы, а в переменной `cellObj` — экземпляр объекта `HTMLTableCellElement`, представляющий третью ячейку этой строки.

> ***Внимание!***
>
> Не забываем, что в случае отсутствия у таблицы явно определенных секций Web-обозреватель неявно создаст секцию тела и поместит в нее все строки этой таблицы.

Разумеется, для доступа к элементам таблицы можем использовать и другие свойства и методы DOM из описанных ранее. Просто коллекцией `childNodes` в таких случаях удобнее пользоваться (по крайней мере, для автора).

Перечисленные ранее объекты поддерживают набор специфических свойств и методов, которые мы можем использовать для "путешествия" по секциям, строкам и ячейкам таблицы. Иногда их использование может упростить код. Поэтому рассмотрим их.

Свойство `rows` объекта `HTMLTableElement` возвращает ссылку на одноименную коллекцию, содержащую все строки таблицы в виде экземпляров объекта `HTMLTableRowElement`. При этом коллекция `rows` представляет все строки таблицы во всех ее секциях.

Коллекция `rows`, как и все упоминаемые далее коллекции, является экземпляром объекта `HTMLCollection`. Мы уже знаем, что этот объект поддерживает свойство `length`, возвращающее количество элементов коллекции; в случае коллекции `rows` это будет количество строк во всех секциях таблицы.

```
var tabObj = document.getElementById("tab");
var rowCount = tabObj.rows.length;
```

Этот сценарий поместит в переменную `rowCount` количество строк во всех секциях таблицы `tab`.

Свойство `tBodies` объекта `HTMLTableElement` возвращает ссылку на одноименную коллекцию, содержащую все секции тела таблицы в виде экземпляров объекта `HTMLTableSectionElement`.

```
var tabObj = document.getElementById("tab");
var bodyObj = tabObj.tBodies[0];
```

Этот сценарий поместит в переменную `bodyObj` экземпляр объекта `HTMLTableSectionElement`, представляющий первую (а скорее всего, единственную) секцию тела таблицы `tab`.

> ***НА ЗАМЕТКУ***
>
> Непонятно, зачем для представления секции тела понадобилось формировать коллекцию. Ведь секция тела в таблице, по идее, всего одна — можно было бы обойтись свойством вида `tBody`.

Свойства `tHead` и `tFoot` объекта `HTMLTableElement` возвращают ссылки на экземпляры объекта `HTMLTableSectionElement`, представляющие, соответственно, секцию заголовка и "поддона" таблицы.

Свойство `tCaption` объекта `HTMLTableElement` возвращает ссылку на экземпляр объекта `HTMLTableCaptionElement`, представляющий название таблицы.

Рассмотрим теперь объект `HTMLTableSectionElement`, представляющий отдельную секцию таблицы. Он поддерживает уже знакомое нам свойство `rows`, возвращающую ссылку на одноименную коллекцию, которая содержит строки, что входят только в данную секцию. Больше тут говорить не о чем.

А вот объект `HTMLTableRowElement`, представляющий отдельную строку таблицы, поддерживает три полезных для нас свойства. Давайте их рассмотрим.

Свойство `cells` возвращает ссылку на одноименную коллекцию ячеек этой строки в виде экземпляров объекта `HTMLTableCellElement`.

```
var tabObj = document.getElementById("tab");
var footObj = tabObj.tFoot;
var rowObj = footObj.rows[0];
var cellObj = rowObj.cells[1];
```

После выполнения этого сценария в переменной `footObj` окажется ссылка на секцию "поддона" таблицы, в переменной `rowObj` — ссылка на единственную строку этой секции, а в переменной `cellObj` — ссылка на вторую ячейку этой строки. Разумеется, в этих переменных окажутся ссылки на экземпляры соответствующих объектов, но давайте говорить вот так, для краткости.

Свойство `rowIndex` возвращает числовой номер текущей строки в таблице без учета секций. Отметим, что нумерация строк в таблице начинается с нуля.

Свойство `sectionRowIndex` возвращает числовой номер текущей строки в секции таблицы, где эта строка находится. Нумерация строк в секции также начинается с нуля.

Что касается объекта `HTMLTableCellElement`, то нам будет здесь полезно только свойство `cellIndex`. Оно возвращает числовой номер ячейки в строке. Нумерация ячеек в строке также начинается с нуля.

Средства DOM для получения параметров элемента страницы

Напоследок рассмотрим еще несколько свойств объекта `HTMLElement`, которые могут нам пригодиться.

Свойство `id` задает или возвращает в строковом виде имя, заданное для элемента страницы через атрибут `ID`. А свойство `name` задает или возвращает также в строковом виде имя, заданное атрибутом `NAME`.

```
var parObj = document.getElementById("par");
parObj.id = "paragraph";
```

Этот сценарий меняет имя абзаца с `par` на `paragraph`. И мы сможем обращаться к этому абзацу по новому имени.

Весьма примечательное, но не очень полезное свойство `nodeName` возвращает одно из трех строковых значений:

- имя тега для элемента страницы, представляющего тег;
- имя атрибута для экземпляра объекта, представляющего атрибут тега (о доступе к атрибутам тега см. *главу 8*);
- текст, являющийся содержимым тега, для элемента страницы, представляющего текстовое содержимое тега.

Кроме того, имя тега, создающего текущий элемент страницы, позволит узнать свойство `tagName`. Именно его, в основном, для этого и используют.

```
var tabObj = document.getElementById("tab");
var footObj = tabObj.tFoot;
var footTagName = footObj.tagName;
```

Этот сценарий поместит в переменную `footTagName` имя тега, с помощью которого создана секция "поддона" таблицы `tab`, — `"TFOOT"` или `"tfoot"` в зависимости от регистра символов, в котором было набрано имя этого тега.

Свойство `nodeType` также примечательно. Оно возвращает одно из следующих значений:

- число 1 для элемента страницы, представляющего тег;
- `null` для экземпляра объекта, представляющего атрибут тега;
- число 3 для элемента страницы, представляющего содержимое тега.

Это свойство можно использовать для определения, является ли данный элемент страницы тегом или фрагментом текста. И используют, кстати.

Свойство `nodeValue` возвращает одно из следующих значений:

- `null` для элемента страницы, представляющего тег, и для экземпляра объекта, представляющего атрибут тега, значение которого не задано;
- значение атрибута для экземпляра объекта, представляющего атрибут тега, который задан явно;
- текст, являющийся содержимым тега, для элемента страницы, представляющего текстовое содержимое тега.

В двух последних случаях этому свойству можно присвоить другое значение, которое станет, соответственно, новым значением атрибута или новым текстом.

Свойством `nodeValue` обычно пользуются для получения и изменения текста, содержащегося в каком-либо теге.

```
var parObj = document.getElementById("par");
var parTextObj = parObj.firstChild;
parTextObj.nodeValue = "Я - абзац!";
```

Этот сценарий изменит текст, присутствующий в абзаце `par`. Сначала мы получаем ссылку на сам этот абзац, потом — на содержащийся в нем текстовый элемент (это единственный потомок абзаца, поэтому мы используем свойство `firstChild`), и нам останется только присвоить новый текст свойству `nodeValue`.

```
var tabObj = document.getElementById("tab");
var bodyObj = tabObj.tBodies[0];
var rowObj = bodyObj.rows[1];
var cellObj = rowObj.cells[0];
cellObj.nodeValue = "Я - ячейка таблицы!";
```

Этот сценарий меняет текст, содержащийся в первой ячейке второй строки секции тела таблицы `tab`. Он аналогичен предыдущему сценарию.

```
var tabObj = document.getElementById("tab");
var captionObj = tabObj.tCaption;
captionObj.nodeValue = "Я - название таблицы!";
```

А этот сценарий меняет текст названия таблицы `tab`.

Последнее свойство, что мы здесь рассмотрим, — `textContent`. Оно задает или возвращает текстовое содержимое тега без учета HTML-форматирования. То есть все теги, что вложены в текущий, отбрасываются, и возвращается только "чистый" текст.

```
<P ID="par">Это <EM>курсивный</EM> текст.</P>
. . .
var parObj = document.getElementById("par");
var parText = parObj.textContent;
```

После выполнения этого сценария в переменной `parText` окажется строка `"Это курсивный текст."`. Теги `<EM>` и `</EM>` при этом будут отброшены.

Это же свойство можно использовать для задания нового текстового содержимого элемента страницы.

Файлы сценариев

Мы изучили вставку сценариев в HTML-код страницы в самом начале этой главы. Но один момент мы там не рассмотрели, хотя он очень важен. Это файлы сценариев.

Файл сценариев — это текстовый файл, содержащий только Web-сценарии. Они записываются без тега `<SCRIPT>`, поскольку Web-обозреватель и так знает, что содержится в этом файле. Файлы сценариев, написанных на языке JavaScript, имеют расширение js.

НА ЗАМЕТКУ

А файлы сценариев, написанных на языке VBScript, имеют расширение vbs.

Но как поместить находящийся в файле сценарий в страницу? Для этого используется особый формат написания уже знакомого нам тега <SCRIPT>:

```
<SCRIPT SRC="<интернет-адрес файла сценария>"></SCRIPT>
```

То есть тег <SCRIPT> оставляется пустым, и в него помещается обязательный в данном случае атрибут SRC, в качестве значения которого указывается интернет-адрес нужного файла сценария.

Сценарии, помещенные в файлы, обрабатываются точно так же, как сценарии, помещенные непосредственно в HTML-код страницы, — в том самом месте, где встретился определяющий их тег <SCRIPT>. Они также могут помещаться и в секции заголовка, и в любом месте секции тела страницы. Так что в этом смысле для Web-обозревателя они не отличаются от сценариев, являющихся "собственностью" страницы.

Обычно в файлы сценариев помещаются сценарии, содержащие некий общий для нескольких страниц код, в основном, объявления функций и объектов общего назначения. Здесь прослеживается аналогия с внешними таблицами стилей CSS (см. *главу 3*), куда выносятся стили, общие для нескольких страниц.

Для примера давайте возьмем приведенный в начале этой главы HTML-код и вынесем присутствующий в нем сценарий в файл. Содержимое этого файла будет таким:

```
var d = new Date();
document.write(d.toLocaleString());
```

Сохраним его под именем 5.1.js.

Теперь исправим HTML-код самой страницы. Исправления будут минимальными — смотрим сами:

```
<HTML>
  <HEAD>
    <TITLE>Использование файла сценария</TITLE>
  </HEAD>
  <BODY>
    <P>
      <SCRIPT SRC="5.1.js"></SCRIPT>
    </P>
  </BODY>
</HTML>
```

Сохраним этот код в файле 5.6.htm и откроем в Web-обозревателе. Вынесенный в файл сценария код будет успешно выполнен.

Что дальше?

Вот мы и познакомились с простейшими Web-сценариями, узнали, что такое DOM и как ее использовать для получения доступа к нужному нам элементу страницы — тегу или фрагменту текста, являющегося его содержимым. Завершившаяся глава была хоть и небольшой, но важной.

Ум жаждет знаний, душа жаждет успехов. В следующей главе мы столкнемся один на один с более сложной разновидностью Web-сценариев — обработчиками событий. И, разумеется, узнаем, что такое события и какую пользу они могут нам принести. Вперед!

Глава 6

Обработка событий

В *главе 5* мы познакомились с основными принципами написания Web-сценариев. Теперь мы знаем, с помощью какого тега сценарии помещаются в HTML-код страницы, как они выполняются и какими средствами следует пользоваться, дабы получить доступ к нужному элементу страницы. Первый шаг в практическое Web-программирование сделан.

Но мы изучили только самую простую разновидность сценариев — загрузочные. Они, конечно, используются в страницах, и весьма часто, но работают, так сказать, на подхвате: объявляют общие для всех сценариев переменные, выполняют предварительные установки, вносят в код страницы при ее загрузке поправки, которые впоследствии так и останутся неизменными и пр. В общем, готовят почву для работы совсем других сценариев.

Каких?

О, это совершенно особые сценарии и совершенно особый принцип программирования! Это обработчики событий — сценарии, выполняющиеся только после возникновения заданного события: щелчка мышью по элементу страницы, помещения курсора мыши над ним, окончание загрузки страницы или графического изображения и др. Они, именно они и никто другой дают "жизнь" страницам, заставляют их реагировать на действия пользователя.

В этой главе мы займемся исключительно обработчиками событий. Мы выясним, что такое события, как они порождаются Web-обозревателем, как пишутся сценарии их обработки и какие средства припасли для этого Web-обозреватели. В общем, сделав первый шаг в Web-программировании, сделаем и второй.

Начнем, разумеется, с азов. Сначала нам нужно выяснить, что такое событие.

События и обработчики событий

В Web-обозревателе постоянно что-то происходит. То завершится загрузка страницы, то посетитель поведет курсором мыши, то щелкнет по гиперссылке, то прокрутит содержимое окна, то переместит его в другое место экрана. Web-обозреватель — довольно "беспокойное" приложение (собственно, как и любое другое приложение Windows).

Когда в Web-обозревателе что-то происходит, он генерирует *событие* — своего рода уведомление о "происшествии", содержащее его описание и некоторые дополнительные данные. (О дополнительных данных, описывающих событие, мы поговорим особо.) Так, когда посетитель щелкает по гиперссылке, генерируется событие "щелчок по гиперссылке". Если посетитель щелкнет мышью на любом элементе страницы, не являющемся гиперссылкой, будет сгенерировано событие "щелчок левой кнопкой мыши" (это не то же самое, что "щелчок по гиперссылке"!). Также существуют события "окончание загрузки страницы", "прокрутка содержимого окна", "перемещение окна" и мн. др.

Сгенерировав событие, Web-обозреватель как-то на него реагирует. В случае "щелчка по гиперссылке" он выполняет переход на целевую страницу или открытие целевого файла. В случае "прокрутки содержимого окна" он, собственно, прокручивает это содержимое. В случае "перемещения окна" он перемещает это окно на новое место.

Но многие события не удостаиваются "внимания" Web-обозревателя. Так, при "щелчке левой кнопкой мыши" на элементе страницы, не являющемся гиперссылкой (или, вообще, любым элементом, который должен изначально реагировать на щелчки), он ничего не делает. Также он ничего не делает при "окончании загрузки страницы".

Но мы можем сделать так, чтобы Web-обозреватель как-то отреагировал на эти события. Для этого мы можем написать особый сценарий — *обработчик события* — и *привязать* его к этому событию. И после наступления заданного события Web-обозреватель выполнит этот сценарий, который сделает со страницей все, что мы пожелаем.

Точно так же мы можем дополнить или изменить поведение Web-обозревателя при возникновении одного из событий, на которые он должен реагировать сам (например, "щелчок по гиперссылке"). Мы можем выполнить какие-то дополнительные действия перед переходом на целевую страницу, перенаправить Web-обозреватель на другую страницу или вообще отменить переход, оставив открытой текущую страницу (вот будет сюрприз для посетителя!). Все это делается в обработчике события.

Теперь уясним следующие моменты, связанные с обработчиками событий.

- Обработчик события привязывается к конкретному элементу страницы, в котором возникают события, требующие обработки. Так, если нужно обработать событие "щелчок левой кнопкой мыши" в текстовом абзаце, обработчик привязывается к этому абзацу.
- Обработчик события привязывается к конкретному событию. Так, если мы привязали обработчик к событию "щелчок левой кнопкой мыши", он будет срабатывать только при возникновении именно этого события. Другие события, скажем, "двойной щелчок левой кнопкой мыши", обрабатываться им не будут.
- Обработчик события выполняется только при возникновении заданного события в элементе страницы, к которому он привязан. Во время загрузки страницы он не выполняется.
- Мы можем привязать один обработчик сразу к нескольким элементам страницы и нескольким событиям. Так, один и тот же обработчик может обрабатывать событие "щелчок левой кнопкой мыши" в абзаце и "щелчок по гиперссылке" в гиперссылке. Кстати, так часто и делают.
- В обработчике события мы можем производить со страницей любые действия. Так, при наступлении события "щелчок левой кнопкой мыши" в абзаце мы можем менять цвет текста этого абзаца или окружать его рамкой.

Вот, собственно, и все начальные сведения о событиях и их обработчиках. Рассмотрим теперь конкретную реализацию обработки событий.

Сразу уясним для себя, что существуют две разные модели обработки событий: модель Internet Explorer и модель Firefox. Первая поддерживается всеми современными Web-обозревателями — и Internet Explorer, и Opera, и Firefox, вторая — только Opera и Firefox. Первая заметно проще и нагляднее, чем вторая, но вторая предоставляет несколько больше возможностей.

Разумеется, мы рассмотрим обе этих модели. И начнем с модели Internet Explorer.

Обработка событий по модели Internet Explorer

Давайте напишем небольшую Web-страницу, реализующую модель обработки событий Internet Explorer. Создадим на ней текстовый абзац, который после щелчка мышью на нем будет менять свой цвет то на красный, то на черный.

Внимание!

Не забываем, что модель Internet Explorer поддерживается всеми Web-обозревателями.

HTML-код этой страницы приведен далее.

```
<HTML>
  <HEAD>
    <TITLE>Пример обработчика событий</TITLE>
    <SCRIPT>
      var flag = false;
      function parClick() {
        var parObj = document.getElementById("par");
        if (flag)
          parObj.style.color = "#000000"
        else
          parObj.style.color = "#FF0000";
        flag = !flag;
      }
    </SCRIPT>
  </HEAD>
  <BODY>
    <P ID="par" ONCLICK="parClick();">Щелкните здесь.</P>
  </BODY>
</HTML>
```

Сохраним страницу в файле 6.1.htm, откроем в любом Web-обозревателе и попробуем щелкнуть мышью на абзаце. После этого его текст должен сменить цвет с черного на красный. Если щелкнуть на нем еще раз, он сменит свой цвет снова на черный и т. д.

Теперь давайте разберем единственный сценарий, присутствующий на данной странице. Это и есть обработчик события "щелчок левой кнопкой мыши", привязанный к абзацу, носящему имя `par`.

Прежде всего, мы объявляем переменную `flag` и присваиваем ее логическое значение `false`. Эта переменная будет хранить признак того, красный цвет имеет текст абзаца (значение `true`) или черный (значение `false`). Изначально цвет абзаца черный, поэтому мы и присвоили этой переменной значение `false` при ее объявлении.

Далее мы объявляем функцию `parClick`. Эта функция и будет менять цвет текста абзаца `par`. Она проверяет значение текущей переменной `flag`; если

это false, то меняет цвет абзаца на красный, если true — на черный. После этого она инвертирует значение этой переменной и снова присваивает ей.

> ***Внимание!***
>
> Если нужно последовательно менять значение какого-либо свойства экземпляра объекта, представляющего элемент страницы, лучше отвести для хранения признака того, какое именно значение присвоено этому свойству, отдельную переменную логического или числового типа. Можно, конечно, вместо этого проверять значение самого свойства, но в этом случае нет никакой гарантии, что Web-обозреватель не изменит его после присваивания согласно каким-то своим представлениям. Так, после присваивания значения "#FF0000" свойству color объекта style Web-обозреватель изменит его на "red" — строковое представление красного цвета, и при последующем обращении к этому свойству мы получим именно строку "red", которая не равна строке "#FF0000".

Да, но где и как вызывается эта функция? Обратим внимание на HTML-код, формирующий абзац par.

```
<P ID="par" ONCLICK="parClick();">Щелкните здесь.</P>
```

Здесь в теге <P> мы видим не знакомый нам атрибут ONCLICK, которому в качестве значения присвоена строка "parClick();". Понятно, что эта строка — суть код обработчика события, он просто вызывает функцию parClick. А атрибут ONCLICK представляет событие onClick — именно так в стандартах HTML называется событие "щелчок левой кнопкой мыши".

Фактически мы просто присвоили сценарий — обработчик события onClick (давайте уж называть его так) одноименному атрибуту тега ONCLICK. Таких атрибутов, представляющих события, довольно много — по числу поддерживаемых объектом HTMLElement и его потомками событий. Имена этих атрибутов совпадают с именами соответствующих событий и традиционно набираются большими буквами.

Мы можем передавать в функции — обработчики событий любое количество параметров. Например, имя элемента страницы, к которому привязан обработчик.

```
<P ID="par" ONCLICK="parClick('par');">Щелкните здесь.</P>
```

Здесь мы передали функции — обработчику события parClick строку с именем абзаца, к которому привязан этот обработчик (par). Отметим, что мы заключили эту строку в одинарные кавычки, поскольку код обработчика уже заключен в двойные кавычки, и поместить внутрь нее еще пару двойных кавычек мы так просто не сможем (подробнее см. *главу 4*).

Впоследствии мы можем использовать переданный таким образом параметр внутри тела функции — обработчика события. Разумеется, сначала в объяв-

лении этой функции мы должны указать, что она принимает параметр (или параметры).

```
function parClick(pID) {
  var parObj = document.getElementById(pID);
  if (flag)
    parObj.style.color = "#000000"
  else
    parObj.style.color = "#FF0000";
  flag = !flag;
}
```

Это модификация приведенной ранее функции `parClick`, но уже использующей для доступа к элементу страницы его имя, переданное в качестве параметра.

Если наш обработчик события совсем небольшой, мы можем даже не создавать функцию, а просто присвоить его код нужному атрибуту тега, который создает элемент страницы, где это событие должно обрабатываться. Но так получается далеко не всегда — обработчики событий, как правило, весьма объемисты.

Есть еще один способ привязать обработчик к событию. Давайте рассмотрим его на очередном примере. Вот HTML-код другой страницы:

```
<HTML>
  <HEAD>
    <TITLE>Пример обработчика событий</TITLE>
    <SCRIPT>
      var flag = false;
      function parClick() {
        var parObj = document.getElementById("par");
        if (flag)
          parObj.style.color = "#000000"
        else
          parObj.style.color = "#FF0000";
        flag = !flag;
      }
    </SCRIPT>
  </HEAD>
  <BODY>
    <P ID="par">Щелкните здесь.</P>
    <SCRIPT>
```

```
      var parObj = document.getElementById("par");
      parObj.onclick = parClick;
    </SCRIPT>
  </BODY>
</HTML>
```

Видно, что атрибут `ONCLICK` тега `<P>`, формирующего абзац `par`, отсутствует. Вместо этого в небольшом сценарии, следующем за создающим данный абзац кодом HTML, мы присваиваем функцию — обработчик события свойству `onclick` этого абзаца. Это свойство, так же, как и атрибут `ONCLICK` тега, соответствует событию `onClick`.

Свойство `onclick`, а также многие другие свойства, соответствующие одноименным событиям, поддерживаются объектом `HTMLElement` и его потомками. Этим свойствам присваиваются функции, которые станут обработчиками этих событий. Имена этих свойств совпадают с именами соответствующих событий, но набираются маленькими буквами.

Сразу отметим, что, в отличие от первого способа привязки обработчика к событию, здесь обработчик обязательно оформляется в виде функции. А как его еще присвоить нужному свойству?

Такой способ для привязки обработчиков событий к элементам страницы используется довольно редко. Но это единственный способ привязать обработчик к событию, поддерживаемому служебными объектами Web-обозревателя, о которых мы поговорим в *главе 7*.

Теперь подытожим все, что сейчас узнали, об обработке событий по модели Internet Explorer.

- В случае привязки обработчиков к событиям через соответствующие атрибуты тега обработчики событий чаще всего оформляются в виде функций.
- В случае привязки обработчиков к событиям через соответствующие свойства объектов, представляющих элементы страницы, обработчики событий обязательно оформляются в виде функций.
- Для привязки обработчика к нужным элементу страницы и событию первым способом код этого обработчика присваивается "отвечающему" за это событие атрибуту соответствующего тега.
- Для привязки обработчика к нужным элементу страницы и событию вторым способом функция-обработчик присваивается "отвечающему" за это событие свойству элемента страницы.
- Объявления функций — обработчиков событий практически всегда помещаются в сценариях, располагающихся в секции заголовка страницы, поскольку они должны быть обработаны в самую первую очередь, еще до загрузки всей страницы.

Мы рассмотрим все события, поддерживаемые элементами страницы, и соответствующие им атрибуты и свойства в последующих главах этой книги. А пока что закончим рассмотрение модели обработки событий Internet Explorer. Перейдем к модели Firefox.

Обработка событий по модели Firefox

Стандарты DOM рекомендуют использовать для обработки событий именно модель Firefox как более прогрессивную. Рекомендуют-то рекомендуют, но сейчас ее поддерживают только Opera и Firefox; Internet Explorer ее не поддерживает. Так что мы будем использовать, в основном, именно модель Internet Explorer как более универсальную.

> ***Внимание!***
>
> Еще одно напоминание: модель обработки событий Firefox поддерживается только Opera и Firefox.

> ***На заметку***
>
> Вполне возможно, Internet Explorer 8, который сейчас только разрабатывается, будет поддерживать модель обработки событий Firefox. По крайней мере, Microsoft утверждает, что он будет полностью поддерживать стандарты DOM.

Для рассмотрения модели привязки событий Firefox мы также создадим небольшую страницу со сценарием — обработчиком события. Она будет аналогична странице 6.1.htm — текстовый абзац, после щелчка мышью на нем меняющий свой цвет то на красный, то на черный.

Вот HTML-код этой страницы:

```
<HTML>
  <HEAD>
    <TITLE>Пример обработчика событий</TITLE>
    <SCRIPT>
      var flag = false;
      function parClick() {
        var parObj = document.getElementById("par");
        if (flag)
          parObj.style.color = "#000000"
        else
          parObj.style.color = "#FF0000";
        flag = !flag;
      }
```

```
    </SCRIPT>
  </HEAD>
  <BODY>
    <P ID="par">Щелкните здесь.</P>
    <SCRIPT>
      var parObj = document.getElementById("par");
      parObj.addEventListener("click", parClick, false);
    </SCRIPT>
  </BODY>
</HTML>
```

Здесь мы видим два сценария. Первый, расположенный в секции заголовка страницы, объявляет функцию `parClick`, которая является обработчиком события. Второй, находящийся после HTML-кода, формирующего абзац `par`, выполняет привязку обработчика. Все примерно то же, что и в предыдущей рассмотренной нами странице, где обработчик привязывается к событию через свойство `onclick`.

Вот только в данном случае обработчик привязывается как-то непонятно. Мы видим вызов некоего метода `addEventListener` экземпляра объекта, представляющего наш абзац, и функцию `parClick` в списке параметров этого метода:

```
parObj.addEventListener("click", parClick, false);
```

Что бы это значило?

Метод `addEventListener` объекта `HTMLElement` подключает к элементу страницы заданную функцию в качестве функции-слушателя указанного события. *Функция-слушатель* — суть все тот же обработчик события, реализованный в виде функции и выполняемый при возникновении данного события в заданном элементе страницы.

```
addEventListener(<имя события в формате DOM>, <функция-слушатель>,
↳<перехватывать события, возникающие в дочерних элементах страницы>);
```

Первым параметром методу `addEventListener` передается имя события, которое нужно обрабатывать, в виде строки формата DOM. Эта строка представляет собой имя события в формате HTML, но набранное строчными буквами и без символов "on" в начале. Например, в случае события `onClick` это будет строка "click" (см. приведенный ранее сценарий).

Вторым параметром передается сама функция-слушатель. Здесь все понятно.

Третий параметр указывает Web-обозревателю, следует ли перехватывать события, возникающие в дочерних по отношению к текущему элементах страницы. О перехвате событий мы поговорим в конце этой главы, а пока

что уясним, что значение `true` включает перехват, `false` — отключает, перехватом событий мы пока заниматься не будем и поэтому третьим параметром метода `addEventListener` всегда будем указывать `false`.

Метод `addEventListener` не возвращает никакого значения.

Обработка событий по модели Firefox имеет одну весьма примечательную особенность, которая может стать преимуществом. Мы можем подключить к одному элементу страницы и к одному событию сразу несколько функций-слушателей, и все они будут выполняться при возникновении данного события в том порядке, в котором были подключены. Мы можем даже подключить к одному элементу страницы и к одному событию одну функцию-слушатель дважды: в первый раз — с задействованным перехватом событий, во второй — с незадействованным. Обрабатывая события по модели Internet Explorer, мы такого сделать не сможем. Там принцип строг: одному событию в данном элементе страницы — один обработчик.

Метод `removeEventListener` объекта `HTMLElement` позволяет удалить подключенную ранее функцию-слушателя.

```
removeEventListener(<имя события в формате DOM>, <функция-слушатель>,
↳<перехватывать события, возникающие в дочерних элементах страницы>);
```

С первыми двумя параметрами все понятно. Третьим же параметром должно быть передано значение `true` или `false`, в зависимости от того, был ли при подключении функции-слушателя задействован перехват событий. Дело в том, что, как мы уже узнали, к элементу страницы можно подключить сколько угодно функций-слушателей, в том числе дважды подключить одну функцию — с задействованным и незадействованным перехватом событий. Так вот, если мы так сделаем, нам придется удалять отдельно функцию-слушатель с задействованным перехватом и отдельно — с незадействованным.

Метод `removeEventListener` также не возвращает никакого значения.

Подведем итог полученным знаниям о модели Firefox.

- Обработчики событий оформляются в виде функций-слушателей.
- Для подключения функции-слушателя к заданным элементу страницы и событию используется метод `addEventListener` объекта `HTMLElement`.
- Мы можем подключить к одному элементу страницы и к одному событию сразу несколько функций-слушателей. Выполняться они при этом будут в том порядке, в котором были подключены.
- Для удаления ранее подключенной функции-слушателя используется метод `removeEventListener` объекта `HTMLElement`.

С обработкой событий по модели Firefox мы закончили. Но разговор о событиях еще далек от завершения.

Получение дополнительной информации о событии

Очень часто нужно получить о событии некоторую дополнительную информацию. Например, это могут быть координаты точки, в которой посетитель щелкнул мышью, код введенного с клавиатуры символа и др.

Специально для представления дополнительной информации о событии стандарты DOM предусматривают особый объект `Event`. Этот объект поддерживает весьма обширный набор свойств, представляющих различные сведения о наступившем событии. Давайте поговорим о нем.

И сразу же заметим, что получение информации о событии различается в зависимости от Web-обозревателя. В Internet Explorer и Opera это выполняется одним способом, а в Firefox — другим. И еще заметим, что модель обработки события в этом случае роли не играет, а играет роль именно Web-обозреватель.

Получение информации о событии в Internet Explorer и Opera

В Internet Explorer и Opera в теле любой функции — обработчика события доступен экземпляр `event` объекта `Event`. Этот экземпляр неявно создается самим Web-обозревателем.

> ***Внимание!***
>
> Не забываем, что экземпляр `event` объекта `Event` доступен только в теле функции — обработчика события.

Объект `Event` в Internet Explorer и Opera поддерживает весьма обширный набор свойств. Большинство из них перечислено в табл. 6.1; свойства, не перечисленные там, а также практические применение описанных там свойств мы рассмотрим в следующих главах этой книги.

Таблица 6.1. *Свойства объекта* `Event` *в Internet Explorer и Opera*

Свойство	Описание
`altKey`	Возвращает `true`, если была нажата клавиша <Alt>
`altLeft`	Возвращает `true`, если была нажата левая клавиша <Alt>, и `false`, если правая
`button`	Возвращает номер кнопки мыши, нажатой посетителем. Список возвращаемых значений приведен в табл. 6.2

Таблица 6.1 *(продолжение)*

Свойство	Описание
`cancelBubble`	Задает, прерывать или не прерывать всплытие событий. О всплытии событий мы поговорим далее в этой главе
`clientX`	Возвращает горизонтальную координату курсора мыши относительно клиентской области окна Web-обозревателя (без учета рамок, заголовка, строки меню, панелей инструментом и строки состояния)
`clientY`	Возвращает вертикальную координату курсора мыши относительно клиентской области окна Web-обозревателя (без учета рамок, заголовка, строки меню, панелей инструментов и строки состояния)
`ctrlKey`	Возвращает `true`, если была нажата клавиша <Ctrl>
`ctrlLeft`	Возвращает `true`, если была нажата левая клавиша <Ctrl>, и `false`, если правая
`fromElement`	Возвращает элемент страницы, на котором ранее находился курсор мыши
`keyCode`	Возвращает код нажатой на клавиатуре клавиши или введенного символа в Unicode
`offsetX`	Возвращает горизонтальную координату курсора мыши относительно элемента страницы, в котором наступило это событие
`offsetY`	Возвращает вертикальную координату курсора мыши относительно элемента страницы, в котором наступило это событие
`propertyName`	Возвращает имя свойства элемента страницы, значение которого изменилось
`repeat`	Возвращает `true`, если событие "нажатие клавиши" наступило повторно вследствие того, что пользователь удерживает клавишу нажатой, и `false` в противном случае
`returnValue`	Разрешает или отменяет поведение по умолчанию для элемента страницы. Подробнее об этом мы поговорим далее в этой главе
`screenX`	Возвращает горизонтальную координату курсора мыши относительно экрана
`screenY`	Возвращает вертикальную координату курсора мыши относительно экрана
`shiftKey`	Возвращает `true`, если была нажата клавиша <Shift>
`shiftLeft`	Возвращает `true`, если была нажата левая клавиша <Shift>, и `false`, если правая

Таблица 6.1 *(окончание)*

Свойство	Описание
`srcElement`	Возвращает элемент страницы, в котором возникло данное событие (*источник события*)
`toElement`	Возвращает элемент страницы, на который был перемещен курсор мыши
`type`	Возвращает имя события в формате DOM
`wheelDelta`	Возвращает величину, на которую было прокручено колесо мыши. Если колесо было прокручено по направлению от посетителя, возвращается положительная величина, если в направлении к посетителю — отрицательная
`x`	Возвращает горизонтальную координату курсора мыши относительно родителя элемента страницы, в котором наступило это событие
`y`	Возвращает вертикальную координату курсора мыши относительно родителя элемента страницы, в котором наступило это событие

Таблица 6.2. *Значения, возвращаемые свойством* `button` *объекта* `Event`

Значение	Описание
0	Ни одна из кнопок мыши не была нажата
1	Была нажата левая кнопка
2	Была нажата правая кнопка
3	Были одновременно нажаты левая и правая кнопки
4	Была нажата средняя кнопка
5	Были одновременно нажаты левая и средняя кнопки
6	Были одновременно нажаты правая и средняя кнопки
7	Были одновременно нажаты все кнопки

Методов объект `Event` в Internet Explorer и Opera не поддерживает.

Давайте рассмотрим небольшую страницу, содержащую сценарий, который использует некоторые из приведенных ранее свойств. Вот ее HTML-код:

```
<HTML>
  <HEAD>
    <TITLE>Пример обработчика событий</TITLE>
    <SCRIPT>
      function parClick() {
```

```
        var parTextObj = event.srcElement.firstChild;
        parTextObj.nodeValue = "offsetX: " + event.offsetX.toString() +
        ", offsetY: " + event.offsetY.toString();
      }
    </SCRIPT>
  </HEAD>
  <BODY>
    <P ID="par" ONCLICK="parClick();">Щелкните в любом месте этого
    абзаца.</P>
  </BODY>
</HTML>
```

Рассмотрим единственный сценарий этой страницы, а именно тело функции — обработчика события `parClick`.

Сначала мы получаем экземпляр объекта `Text`, представляющий текстовое содержимое элемента страницы, в котором возникло событие, — абзаца `par`. Для этого мы обращаемся к свойству `srcElement` экземпляра `event` объекта `Event`. Полученную ссылку мы присваиваем переменной `parTextObj`. Этим занимается первое выражение тела функции.

Второе выражение тела функции `parClick`, воспользовавшись свойством `nodeValue` объекта `Text`, помещает в абзац `par` новое содержимое. Это строка, содержащая координаты точки, на которой посетитель щелкнул мышью, вычисленные относительно элемента страницы, в котором возникло событие, — самого этого абзаца `par`. Для получения числовых значений координат мы обращается к свойствам `offsetX` и `offsetY` объекта `Event`, которые преобразуем в строковый вид.

Обратим внимание, как мы получаем строковое представление чисел, возвращенных свойствами `offsetX` и `offsetY` объекта `Event`. Сначала мы получаем само значение свойства (`event.offsetX`), потом к этому значению применяем метод `toString` (`event.offsetX.toString()`). JavaScript позволяет выполнять такие вот "цепочечные" вызовы методов.

Как мы видим, Internet Explorer и Opera достаточно "дружественны" к Web-программисту. Чтобы получить сведения о событии, нам не придется совершать дополнительных "телодвижений". В отличие от Firefox…

Получение информации о событии в Firefox

При выполнении функции — обработчика события Firefox неявно передает ей единственный параметр — экземпляр объекта `Event`. Пользуясь им, мы сможем добраться до интересующих нас сведений о возникшем событии.

Здесь нужно помнить одну вещь. Если мы в Firefox выполняем обработку события по модели Internet Explorer, помещая вызов функции — обработчика события в атрибут тега, соответствующий событию, то должны указать в качестве единственного параметра этой функции экземпляр `event` объекта `Event`. Этот экземпляр создаст сам Firefox при возникновении события.

```
<P ID="par" ONCLICK="parClick(event);">Щелкните здесь.</P>
```

где `parClick` — функция обработчика.

Если мы присваиваем функцию — обработчик события свойству элемента страницы, соответствующему нужному событию, или выполняем обработку событий по модели Firefox, нам ничего делать не нужно — Firefox сам передаст в функцию этот параметр.

Сведения, которые мы можем получить о событии в Firefox, также весьма обширны. Посмотрим на табл. 6.3 — там перечислены свойства объекта `Event`, которые будут нам полезны.

Таблица 6.3. *Свойства объекта* `Event` *в Firefox*

Свойство	Описание
`altKey`	Возвращает `true`, если была нажата клавиша <Alt>
`bubbles`	Возвращает `true`, если событие может всплывать. О всплытии событий будет рассказано далее в этой главе
`button`	Возвращает номер кнопки мыши, нажатой пользователем. Список возвращаемых значений приведен в табл. 6.2
`cancelable`	Возвращает `true`, если имеется возможность отменить поведение по умолчанию для данного события. О поведении по умолчанию и его отмене будет рассказано далее в этой главе
`charCode`	Возвращает код введенного с клавиатуры символа в Unicode. Подробнее это свойство будет описано в *главе 8*
`clientX`	Возвращает горизонтальную координату курсора мыши относительно клиентской области окна Web-обозревателя (без учета рамок, заголовка, строки меню, панелей инструментов и строки состояния)
`clientY`	Возвращает вертикальную координату курсора мыши относительно клиентской области окна Web-обозревателя (без учета рамок, заголовка, строки меню, панелей инструментов и строки состояния)
`ctrlKey`	Возвращает `true`, если была нажата клавиша <Ctrl>
`currentTarget`	Возвращает элемент страницы, в котором в данный момент происходит обработка событий. Может не совпадать с элементом страницы, в котором возникло событие, если используется перехват событий или событие всплыло (см. далее)

Таблица 6.3 *(окончание)*

Свойство	Описание
`detail`	Возвращает количество щелчков мышью, выполненных посетителем на данном элементе страницы
`eventPhase`	Возвращает 1, если событие было перехвачено; 2, если событие обрабатывается в элементе страницы, где возникло; 3, если событие всплыло
`keyCode`	Возвращает код нажатой на клавиатуре клавиши в Unicode. Подробнее это свойство будет описано в *главе 8*
`layerX`	Возвращает горизонтальную координату курсора мыши относительно элемента страницы, в котором наступило это событие
`layerY`	Возвращает вертикальную координату курсора мыши относительно элемента страницы, в котором наступило это событие
`pageX`	Возвращает горизонтальную координату курсора мыши относительно всей страницы. При этом принимается во внимание то, что страница может быть прокручена по горизонтали
`pageY`	Возвращает вертикальную координату курсора мыши относительно всей страницы. При этом принимается во внимание то, что страница может быть прокручена по вертикали
`relatedTarget`	Возвращает элемент страницы, с которого переместился курсор мыши, или элемент страницы, на который был перемещен курсор мыши, в зависимости от возникшего события. Подробнее см. *главу 8*
`screenX`	Возвращает горизонтальную координату курсора мыши относительно экрана
`screenY`	Возвращает вертикальную координату курсора мыши относительно экрана
`shiftKey`	Возвращает `true`, если была нажата клавиша <Shift>
`target`	Возвращает элемент страницы, в котором возникло событие (источник события)
`type`	Возвращает имя события в формате DOM
`which`	Возвращает код введенного с клавиатуры символа или нажатой клавиши в Unicode. Подробнее это свойство будет описано в *главе 8*

На заметку

Вообще-то, объект `Event` в Firefox поддерживает еще и свойство `isChar`, возвращающее `true`, если была нажата алфавитно-цифровая клавиша, и `false`, если была нажата клавиша управляющая. Но в реальности из-за не исправленной на момент написания этой книги ошибки это свойство всегда возвращает `false`.

Кроме того, объект `Event` в Firefox поддерживает два метода, которые мы рассмотрим далее в этой главе. Они применяются для служебных целей.

В качестве примера исправим приведенный ранее HTML-код так, чтобы он работал в Firefox.

```
<HTML>
  <HEAD>
    <TITLE>Пример обработчика событий</TITLE>
    <SCRIPT>
      function parClick(evt) {
        var parTextObj = evt.target.firstChild;
        parTextObj.nodeValue = "layerX: " + evt.layerX.toString() +
        ", layerY: " + evt.layerY.toString();
      }
    </SCRIPT>
  </HEAD>
  <BODY>
    <P ID="par" ONCLICK="parClick(event);">Щелкните в любом месте этого
    абзаца.</P>
  </BODY>
</HTML>
```

Здесь мы видим, что функция — обработчик события принимает параметр — экземпляр объекта `Event`. Этот экземпляр мы передаем функции при ее вызове в сценарии, присвоенном атрибуту `ONCLICK`. Именно он будет нести всю информацию о событии, что нам нужна. В остальном данный код аналогичен приведенному ранее, за тем исключением, что используются свойства `layerX` и `layerY` объекта `Event`.

Всплытие событий

А теперь поговорим о таком интересном явлении, как всплытие события. Оно позволяет нам написать обработчик события, который будет обрабатывать события, возникающие сразу в нескольких элементах страницы, обойдя тем самым ограничение "один элемент страницы, одно событие, один обработчик".

Когда в каком-либо элементе страницы возникает событие, сначала выполняется обработчик, привязанный к этому элементу (если он, конечно, есть). После этого событие перемещается в родитель данного элемента страницы, и выполняется обработчик, привязанный уже к нему (опять же, если он есть).

Далее событие перемещается в родитель родителя и т. д., пока не достигнет секции тела страницы (тега <BODY>), где его прохождение завершается. Это прохождение события "снизу вверх" и называется *всплытием*.

Давайте рассмотрим пример страницы со сценарием, выполняющим обработку события onClick сразу в трех абзацах. Обработчик этого события мы привяжем к блочному контейнеру, в который поместим эти абзацы. (О блочных контейнерах говорилось в *главе 3*.)

```
<HTML>
  <HEAD>
    <TITLE>Всплытие событий</TITLE>
    <SCRIPT>
      function contClick() {
        var outputObj = document.getElementById("output");
        var outputTextObj = outputObj.firstChild;
        outputTextObj.nodeValue = "Щелкнули!";
      }
    </SCRIPT>
  </HEAD>
  <BODY>
    <DIV ID="cont" ONCLICK="contClick();">
      <P>Абзац 1.</P>
      <P>Абзац 2.</P>
      <P>Абзац 3.</P>
    </DIV>
    <P ID="output">Щелкните на одном из расположенных выше абзацев.</P>
  </BODY>
</HTML>
```

Комментировать тут особо нечего. Обработчик события onClick, привязанный к контейнеру cont, обрабатывает события, возникающие в содержащихся в нем абзацах, и выводит в абзац output текст "Щелкнули!". Отметим, что эта страница будет нормально обрабатываться всеми Web-обозревателями.

> ***НА ЗАМЕТКУ***
>
> Есть еще один способ обойти ограничение "один элемент страницы, одно событие, один обработчик" — привязать один и тот же обработчик события сразу к нескольким элементам страницы. Собственно, об этом уже говорилось.

Если мы пишем обработчик всплывающего события, нам может понадобиться получить доступ к элементу страницы, в котором это событие возникло.

Для этого мы воспользуемся свойством `srcElement` объекта `Event` Internet Explorer и Opera и свойством `target` этого же объекта в Firefox, возвращающие элемент страницы — источник события. Оба этих свойства были рассмотрены ранее.

Если и к элементу страницы, в котором возникло событие, и к его родителю были привязаны обработчики, нам, возможно, понадобится узнать, не получил ли обработчик события, привязанный к родителю, всплывшее событие. Для этого проще всего сравнить значение свойства `srcElement` или `target` и сам родитель. Сначала мы получим родительский элемент страницы (предположим, что он имеет имя `parent`):

```
var parentObj = document.getElementById("parent");
```

Потом получим значение свойства `srcElement` или `target`:

```
var targetObj = event.srcElement; //Для Internet Explorer и Opera
var targetObj = evt.target; //Для Firefox. evt — экземпляр объекта Event,
                            //переданный функции — обработчику события
                            //через параметр
```

И сравним их:

```
if (parentObj == targetObj) {
  //Событие возникло в самом родителе
} else {
  //Событие всплыло из дочернего элемента
}
```

Кроме того, можно проверить значение свойства `eventPhase` объекта `Event` в Firefox. Если событие возникло в самом этом элементе страницы, оно вернет число 2, если всплыло из дочернего элемента — число 3.

Объект `Event` в Firefox поддерживает еще одно полезное свойство — `bubbles`. Дело в том, что некоторые события не всплывают — они начинают и оканчивают свою "жизнь" в том элементе страницы, в котором возникли. Узнать, может ли событие всплывать, и поможет это свойство: если событие может всплывать, оно вернет `true`, в противном случае — `false`.

Но что делать, если нужно прервать всплытие события в одном из обработчиков? Для этого Internet Explorer и Opera поддерживают свойство `cancelBubble` объекта `Event`. Чтобы прервать всплытие события, достаточно в обработчике присвоить этому свойству значение `true`; значение `false` разрешает дальнейшее всплытие события.

```
event.cancelBubble = true;
```

Firefox для этих же целей предлагает вызвать метод `stopPropagation` объекта `Event`, не принимающий параметров и не возвращающий значения.

```
evt.stopPropagation();
```

где `evt` — экземпляр объекта `Event`, переданный функции — обработчику события через параметр.

Перехват событий в дочерних элементах в модели обработки событий Firefox

В модели обработки событий Firefox существует еще одно явление — перехват событий. Сейчас мы о нем поговорим.

Перехват событий — это такой способ обработки событий, когда обработчик, привязанный к родителю, обрабатывает все события, возникшие в дочерних элементах, в самую первую очередь, а уже потом событие обрабатывается в дочернем элементе, в котором реально возникло. То есть последовательность обработки событий в данном случае такова:

1. Выполняется привязанный к родителю обработчик события, для которого был задействован перехват событий.
2. Выполняется обработчик события, привязанный к дочернему элементу, в котором возникло событие.
3. Начинается всплытие события согласно описанным ранее правилам.

Указать, задействовать ли перехват событий, можно при подключении к элементу страницы функции-слушателя (которая и станет обработчиком события) — в вызове метода `addEventListener` объекта `HTMLElement`. Как мы помним, он принимает третий параметр, которым передается логическая величина (выделен полужирным шрифтом).

```
addEventListener(<имя события в формате DOM>, <функция-слушатель>,
↳<перехватывать события, возникающие в дочерних элементах страницы>);
```

Так вот, этот параметр и указывает Web-обозревателю, задействовать ли для данной функции-слушателя перехват событий. Значение `true` указывает задействовать перехват, значение `false` отключает его, то есть задействует поведение по умолчанию.

Если впоследствии понадобится удалить функцию-слушатель, мы используем метод `removeEventListener` объекта `HTMLElement`. Этот метод также принимает третий параметр, указывающий, была ли данная функция-слушатель подключена с задействованием перехвата событий (значение `true`) или нет (`false`).

```
removeEventListener(<имя события в формате DOM>, <функция-слушатель>,
↳<перехватывать события, возникающие в дочерних элементах страницы>);
```

Это необходимо, так как модель Firefox позволяет подключить одну и ту же функцию-слушатель к элементу страницы дважды: с задействованием перехвата событий и без него.

Для примера давайте рассмотрим HTML-код, приведенный далее:

```
<HTML>
  <HEAD>
    <TITLE>Перехват событий</TITLE>
    <SCRIPT>
      function contClick() {
        var outputObj = document.getElementById("output");
        var outputTextObj = outputObj.firstChild;
        var s = outputTextObj.nodeValue;
        outputTextObj.nodeValue = s + " contClick,";
      }
      function contInterceptClick() {
        var outputObj = document.getElementById("output");
        var outputTextObj = outputObj.firstChild;
        var s = outputTextObj.nodeValue;
        outputTextObj.nodeValue = s + " contInterceptClick,";
      }
      function parClick() {
        var outputObj = document.getElementById("output");
        var outputTextObj = outputObj.firstChild;
        var s = outputTextObj.nodeValue;
        outputTextObj.nodeValue = s + " parClick,";
      }
    </SCRIPT>
  </HEAD>
  <BODY>
    <DIV ID="cont">
      <P ID="par1">Абзац 1.</P>
      <P ID="par2">Абзац 2.</P>
      <P ID="par3">Абзац 3.</P>
    </DIV>
    <P ID="output">Последовательность выполнения обработчиков
    событий:</P>
    <SCRIPT>
      var contObj = document.getElementById("cont");
```

```
      contObj.addEventListener("click", contClick, false);
      contObj.addEventListener("click", contInterceptClick, true);
      var parObj = document.getElementById("par1");
      parObj.addEventListener("click", parClick, false);
      var parObj = document.getElementById("par2");
      parObj.addEventListener("click", parClick, false);
      var parObj = document.getElementById("par3");
      parObj.addEventListener("click", parClick, false);
    </SCRIPT>
  </BODY>
</HTML>
```

Здесь присутствуют три функции — обработчика событий; каждая из них выводит в последнем абзаце (`output`) свое имя. Функция `contClick` привязывается к контейнеру `cont`, функция `contInterceprClick` — также к контейнеру `cont`, но с задействованием перехвата событий, а `parClick` — к абзацам `par1`, `par2` и `par3`, вложенным в контейнер `cont`.

Если мы сохраним приведенный ранее код в HTML-файле, откроем его в Firefox и щелкнем по любому из абзацев `par1`, `par2` или `par3`, то в последнем абзаце (который `output`) увидим такую последовательность вызова функций — обработчиков событий: `contInterceptClick`, `parClick`, `contClick`. То есть сначала будет выполнена функция `contInterceptClick` (привязанная к контейнеру `cont` с задействованным перехватом событий), потом — `parClick` (привязанная к абзацам `par1`, `par2` и `par3`), а самой последней — `contClick` (привязанная к контейнеру `cont`).

Осталось сказать совсем немного. Свойство `eventPhase` объекта `Event` в Firefox позволит нам узнать, задействован ли для данной функции — обработчика перехват событий. Если это так, данное свойство вернет число 1, в противном случае — число 2 (если событие обрабатывается в элементе страницы, в котором возникло) или 3 (если событие всплыло из дочернего элемента).

> ***Внимание!***
>
> Перехват событий поддерживается только в модели обработки событий Firefox.

Поведение по умолчанию и его отмена

Как мы выяснили в самом начале этой главы, для некоторых событий существует *поведение по умолчанию* — действие, предпринимаемое в ответ на него самим Web-обозревателем. Классический пример — гиперссылка; для нее это переход на целевую страницу или загрузка целевого файла.

Мы можем отменить поведение по умолчанию совсем или временно. Как — сейчас выясним.

Самый простой способ отменить поведение по умолчанию — использовать в качестве обработчика события выражение `return false;`, вписав его прямо в соответствующий атрибут. Этот способ работает во всех Web-обозревателях.

```
<A HREF="somepage.html" ONCLICK="return false;">Никуда не веду</A>
```

При щелчке на гиперссылке "Никуда не веду" перехода на страницу somepage.html не происходит.

Для отмены поведения по умолчанию мы также можем использовать обработчики события вида:

```
<A HREF="somepage.html"
ONCLICK="someFunc(); return false;">Никуда не веду</A>
```

Здесь сначала будет вызвана функция `someFunc`, а потом выполнено выражение `return false`, собственно отменяющее поведение по умолчанию. В дальнейшем мы иногда будем применять подобный подход.

Другой способ несколько сложнее и предполагает использование объекта `Event`. Он подойдет, если нужно запрещать поведение по умолчанию на время, в зависимости от некоторого условия.

Объект `Event` в Internet Explorer и Opera поддерживает свойство `returnValue`. Если ему присвоить значение `false`, поведение по умолчанию для данного события будет отменено.

```
<SCRIPT>
  function aClick() {
    event.returnValue = false;
  }
</SCRIPT>
. . .
<A HREF="somepage.html" ONCLICK="aClick();">Никуда не веду</A>
```

Изначальное значение свойства `returnValue` — `true`, разрешающее поведение по умолчанию.

В Firefox для той же цели используется вызов метода `preventDefault` объекта `Event`, не принимающего параметров и не возвращающего значения.

```
<SCRIPT>
  function aClick(evt) {
    evt.preventDefault();
  }
```

```
</SCRIPT>
. . .
<A HREF="somepage.html" ONCLICK="aClick(event);">Никуда не веду</A>
```

Объект `Event` в Firefox также поддерживает свойство `cancellable`, позволяющее выяснить, можно ли для данного события отменить поведение по умолчанию (фактически — существует ли для данного события и элемента страницы поведение по умолчанию). Если это так, данное свойство вернет значение `true`, в противном случае — `false`.

На этом мы пока закончим с событиями. В последующих главах мы обязательно к ним вернемся и рассмотрим события, поддерживаемые различными объектами Web-обозревателей, и не рассмотренные здесь возможности объекта `Event`. О, он таит еще немало сюрпризов!..

Что дальше?

Вот мы и познакомились с событиями, обработчиками событий и всем, что с ними связано. Теперь мы готовы к "оживлению" наших страниц. Нам недостает совсем немного — знания, какие события поддерживают объекты Web-обозревателя и как их использовать.

Эти знания мы получим в последующих главах этой книги. И в следующей же главе начнем изучение объектов, представляющих различные возможности самого Web-обозревателя. Мы научимся управляться с его окнами, историей, интернет-адресами страниц и фреймами. Держись, Web-обозреватель!

Глава 7

Работа с Web-обозревателем

Вот и настала пора, вооружившись всеми полученными в предыдущих главах знаниями, приступить к написанию сценариев, работающих с Web-обозревателем и открытой в нем Web-страницей. И начнем мы с Web-обозревателя, так как работать с ним проще.

Итак, что же мы сможем сделать с Web-обозревателем? Довольно много…

- Получить сведения о самой программе Web-обозревателя (версия, программное ядро, операционная система, язык и пр.).
- Управлять окнами Web-обозревателя (открывать, закрывать, перемещать их, изменять их размеры и пр.).
- Получить сведения об интернет-адресе, откуда была загружена текущая страница.
- Двигаться по истории Web-обозревателя (например, открыть предыдущую просмотренную страницу).
- Получить сведения о видеоподсистеме клиентского компьютера (в частности, разрешение и цветность экрана).

Все это выполняется с помощью особых объектов и их экземпляров, предоставляемых самим Web-обозревателем в наше полное распоряжение. Нам остается только правильно воспользоваться свойствами, методами и событиями, поддерживаемыми этими объектами. Этому-то как раз и посвящена данная глава.

Получение сведений о Web-обозревателе

Начнем мы с самого простого — получения сведений о самом Web-обозревателе. Это может быть полезно, например, если мы предусмотрели различные версии одних и тех же страниц для разных программ Web-обозревателей и теперь

должны выяснить, каким Web-обозревателем пользуется посетитель, дабы перенаправить его на нужную страницу. (О перенаправлении посетителя на другую страницу будет рассказано далее в этой главе.)

Все сведения о Web-обозревателе, в котором открыта текущая страница, сосредоточены в свойствах объекта `Navigator`. Единственный экземпляр этого объекта — `navigator` — создается самим Web-обозревателем и доступен в любом сценарии, присутствующем на странице, через одноименную переменную. Можно сказать, Web-обозреватель сам "представляется" нам.

Объект `Navigator` поддерживает множество свойств, которые перечислены в табл. 7.1, и один малополезный метод. Событий он не поддерживает.

Таблица 7.1. *Свойства объекта* `Navigator`

Свойство	Описание
`appCodeName`	Возвращает имя исходного кода программного ядра Web-обозревателя. Полное описание возвращаемого этим свойством значения приведено далее
`appName`	Возвращает имя программы Web-обозревателя. Полное описание возвращаемого этим свойством значения приведено далее
`appVersion`	Возвращает версию программы Web-обозревателя. Полное описание возвращаемого этим свойством значения приведено далее
`browserLanguage`	Возвращает код языка программы Web-обозревателя (например, "ru" для русского языка, "en" для английского и пр.). Поддерживается только Internet Explorer и Opera
`cookieEnabled`	Возвращает `true`, если Web-обозревателю разрешен пользователем прием cookie, и `false` в противном случае. Поддерживается только Internet Explorer
`cpuClass`	Возвращает наименование процессора клиентского компьютера, например, "x86" или "Alpha". Поддерживается только Internet Explorer
`language`	Возвращает код языка программы Web-обозревателя. Аналогично `browserLanguage`. Поддерживается только Opera и Firefox
`onLine`	Возвращает `true`, если клиент в настоящий момент подключен к Интернету, и `false`, если отключен от него (автономный режим). Поддерживается только Internet Explorer и Firefox
`platform`	Возвращает обозначение операционной системы клиентского компьютера, например, "Win32"

Таблица 7.1 *(окончание)*

Свойство	Описание
`systemLanguage`	Возвращает код языка операционной системы клиента. Поддерживается только Internet Explorer
`userAgent`	Возвращает строку, идентифицирующую Web-обозреватель. Полное описание возвращаемого этим свойством значения приведено далее
`userLanguage`	Судя по всему, то же самое, что `browserLanguage`. Поддерживается только Internet Explorer и Opera

Не принимающий параметров метод `javaEnabled` объекта `Navigator` возвращает `true`, если Web-обозреватель может выполнять сценарии JavaScript, и `false` в противном случае. Этот метод для нас бесполезен, поскольку, если пользователь отключил в настройках безопасности Web-обозревателя исполнение сценариев, то никакой сценарий и так выполнен не будет.

А теперь нужно дать пояснения по поводу некоторых из приведенных в табл. 7.1 свойств. Всего этих свойств четыре: `appCodeName`, `appName`, `appVersion` и `userAgent`.

Начнем со свойства `appCodeName`. Для всех Web-обозревателей оно вернет строку "Mozilla" — название старой программы, на исходном коде которой основаны все современные Web-обозреватели. Современный Web-обозреватель Mozilla (ныне — SeaMonkey) — это, можно сказать, дальнейшее развитие старого Mozilla.

Фактически свойство `appCodeName` — дань памяти современных Web-обозревателей "старичкам". Для нас оно совершенно бесполезно.

Со свойствами `appName`, `appVersion` и `userAgent` все много сложнее. Для каждой из рассмотренных в книге программ возвращаемые ими значения будут разными.

Так, свойство `appName` будет возвращать следующие строковые значения:

- "Microsoft Internet Explorer" для Internet Explorer и Opera в режиме совместимости с Internet Explorer;
- "Netscape" для Firefox, Opera в режиме совместимости с Firefox, Mozilla и Navigator;
- "Opera" для Opera в режиме представления "своим именем".

> ***Внимание!***
>
> Для Mozilla, Firefox и Opera в режиме совместимости с Firefox значение, возвращаемое свойством `appName`, будет равно "Netscape". Дело в том, что Web-обозреватели Mozilla и Firefox основаны на исходных текстах Netscape Navigator.

Теперь обратимся к свойству `appVersion`. Для Internet Explorer возвращаемое этим свойством значение будет самым запутанным:

<версия исходного кода ядра Web-обозревателя>

↳(compatible; *<версия самого Web-обозревателя>*;

↳*<сведения об операционной системе>*[; *<данные непонятного назначения>*])

Для нас будут полезны только `версия самого Web-обозревателя` и `сведения об операционной системе`. `Версия исходного кода ядра` для всех современных версий Internet Explorer — "4.0" и для нас бесполезна. Остальные данные также бесполезны, а зачастую и непонятны.

На компьютере автора значение свойства `appVersion` таково:

```
4.0 (compatible; MSIE 7.0; Windows NT 6.0; Mozilla/4.0 (compatible;
↳MSIE 6.0; Windows NT 5.1; SV1) ; SLCC1; .NET CLR 2.0.50727;
↳.NET CLR 3.0.04506; .NET CLR 1.1.4322)
```

Здесь "MSIE 7.0" — это Internet Explorer 7.0, "Windows NT 6.0" — Windows Vista. (Windows 2000 обозначается как "Windows NT 5.0", а Windows XP — как "Windows NT 5.1".) Строки ".NET CLR 2.0.50727", ".NET CLR 3.0.04506" и ".NET CLR 1.1.4322" обозначают версии установленных на компьютере исполняющих сред Microsoft .NET — 2.0, 3.0 и 1.0 соответственно. Назначение остальных данных, присутствующих в этой строке, автору книги установить не удалось.

Перейдем к Opera. Если в ее настройках задан режим совместимости с Internet Explorer, значение свойства `appVersion` будет немного другим. Вот его формат:

<версия исходного кода ядра Web-обозревателя> (compatible; MSIE 6.0;

↳*<сведения об операционной системе>*; *<обозначение языка>*)

Например:

```
4.0 (compatible; MSIE 6.0; Windows NT 6.0; ru)
```

Отметим, что Opera в этом случае всегда "представляется" Internet Explorer 6.0.

В случае если мы включим режим совместимости Opera с Firefox, то получим значение такого формата:

<версия исходного кода ядра Web-обозревателя>

↳(*<сведения об операционной системе>*; U|I; *<обозначение языка>*;

↳rv:*<ближайшая версия Mozilla>*)

`Версия исходного кода ядра` здесь — "5.0". Буква "U", по идее, обозначает американскую версию программы, а буква "I" — интернациональную. В реальности же там почему-то всегда присутствует буква "U". `Ближайшая версия Mozilla` — всегда "1.8.0".

Например:

```
5.0 (Windows NT 6.0; U; ru; rv:1.8.0)
```

В случае если в настройках Opera задан режим представления "своим именем", мы получим такое значение:

```
<версия Opera> (<сведения об операционной системе>; U|I;
↳<обозначение языка>)
```

Например:

```
9.25 (Windows NT 6.0; U; ru)
```

Лаконичные Firefox, Mozilla и Navigator не принесут нам таких сюрпризов, как "болтливый" Internet Explorer и многоликий (даже чересчур) Opera. Формат выдаваемого ими значения свойства `appVersion` таков:

```
<версия исходного кода ядра> (<сведения об операционной системе>;
↳<обозначение языка>)
```

Например, автор получил такое значение для русской версии Firefox 2.0.0.11:

```
5.0 (Windows; ru)
```

для Mozilla 1.7 и Navigator 7 preview 1:

```
5.0 (Windows; en-US)
```

> ***Внимание!***
>
> Mozilla и Navigator выдают пятибуквенный код языка, например, "ru-RU" для русского или "en-US" для американского английского. Все остальные Web-обозреватели выдают двухбуквенный код языка, например, "ru" или "en".

Осталось разобраться со свойством `userAgent`. Возвращаемое им значение также зависит от Web-обозревателя. Internet Explorer, например, выдаст вот что:

```
<значение свойства appCodeName>/<значение свойства appVersion>
```

Например:

```
Mozilla/4.0 (compatible; MSIE 7.0; Windows NT 6.0; Mozilla/4.0
↳(compatible; MSIE 6.0; Windows NT 5.1; SV1) ; SLCC1; .NET CLR
2.0.50727;
↳.NET CLR 3.0.04506; .NET CLR 1.1.4322)
```

Web-обозреватель Opera, "притворяющийся" Internet Explorer, при обращении к свойству `userAgent` выдаст такую же строку:

```
<значение свойства appCodeName>/<значение свойства appVersion> Opera
↳<версия Opera>
```

Например:

```
Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 6.0; ru) Opera 9.25
```

Заметим, что здесь в конце строки указывается версия Opera, так что мы всегда сможем выяснить, действительно ли это Opera.

Если включить в Opera режим совместимости с Firefox, при обращении к свойству `userAgent` он выдаст вот такую строку:

```
Mozilla/<версия исходного кода ядра Web-обозревателя>
↳(<версия операционной системы>; U|I; <обозначение языка>;
↳rv:<ближайшая версия Mozilla>) Gecko/<версия программного ядра>
↳Firefox/<версия Firefox> Opera <версия Opera>
```

Здесь "Gecko" — название программного ядра Firefox (и Mozilla, кстати), а `ближайшая версия Mozilla` — также всегда "1.8.0". Например:

```
Mozilla/5.0 (Windows NT 6.0; U; ru; rv:1.8.0) Gecko/20060728
↳Firefox/1.5.0 Opera 9.25
```

Opera, "представляющийся" самим собой, выдаст такое значение этого свойства:

```
<значение свойства appCodeName>/<значение свойства appVersion>
```

т. е. как и Internet Explorer. Например:

```
Opera/9.25 (Windows NT 6.0; U; ru)
```

Значение свойства `userAgent` в случае Firefox будет таким:

```
Mozilla/<версия исходного кода ядра Web-обозревателя>
↳(<название операционной системы>; U|I; <версия операционной системы>;
↳<обозначение языка>; rv:<ближайшая версия Mozilla>)
↳Gecko/<версия программного ядра> Firefox/<версия Firefox>
```

Например:

```
Mozilla/5.0 (Windows; U; Windows NT 6.0; ru; rv:1.8.1.11) Gecko/20071127
↳Firefox/2.0.0.11
```

Заметим, что ближайшая версия Mozilla здесь другая.

Рассмотрим "за компанию" также Mozilla и Navigator. Mozilla мало отличается от своего "потомка" Firefox и выдаст вот что:

```
Mozilla/<версия исходного кода ядра Web-обозревателя>
↳(<название операционной системы>; U|I; <версия операционной системы>;
↳<обозначение языка>; rv:<версия Mozilla>)
↳Gecko/<версия программного ядра>
```

Например:

```
Mozilla/5.0 (Windows; U; Windows NT 5.1; en-US; rv:1.7) Gecko/20040616
```

Что касается "ветерана" Navigator, то вот формат его значения свойства `userAgent`:

```
Mozilla/<версия исходного кода ядра Web-обозревателя>
↳(<название операционной системы>; U|I; <версия операционной системы>;
↳<обозначение языка>; rv:<ближайшая версия Mozilla>)
↳Gecko/<версия программного ядра> Netscape/<версия Navigator>
```

На своем компьютере автор получил вот что:

```
Mozilla/5.0 (Windows; U; Windows NT 5.1; en-US; rv:1.0rc2) Gecko/20020512
↳Netscape/7.0b1
```

Подведем итоги. Прежде всего, со свойствами `appVersion` и `userAgent` каши явно не сваришь — больно уж сложен и разнообразен формат возвращаемых ими строк. Поэтому для идентификации Web-обозревателя лучше пользоваться свойством `appName`. А для определения языка (это может понадобиться, если наш сайт имеет разные версии для разных языков — очень частая ситуация) мы воспользуемся свойствами `browserLanguage` (для Internet Explorer и Opera) и `language` (для Firefox).

Поскольку мы собираемся серьезно заниматься Web-программированием, то сразу давайте напишем три функции, которые пригодятся нам в дальнейшем. Они будут возвращать различные сведения о Web-обозревателе в более удобном виде, нежели аналогичные свойства объекта `Navigator`.

Первая из этих функций — `getBrowser` — будет возвращать обозначение Web-обозревателя с учетом режима совместимости, включенного в Opera:

- "IE" — для Internet Explorer и Opera с включенным режимом совместимости с Internet Explorer;
- "O" — для Opera;
- "FF" — для Firefox, Opera с включенным режимом совместимости с Firefox, Mozilla и Navigator.

Эта функция не принимает параметров. А вот ее код:

```
function getBrowser() {
  switch (navigator.appName) {
    case "Microsoft Internet Explorer":
      return "IE";
      break;
    case "Opera":
      return "O"; //Это буква "O", а не ноль
      break;
```

```
    case "Netscape":
      return "FF";
  }
}
```

Здесь мы просто проверяем значение свойства `appName` объекта `Navigator` в выражении выбора и возвращаем соответствующую строку из перечисленных ранее.

Пример использования этой функции:

```
if (getBrowser() == "FF") {
  //Посетитель использует Firefox
}
```

Вторая функция — `getBrowserStrict` — также не будет принимать параметров и будет возвращать обозначение Web-обозревателя, не учитывая режим совместимости, включенный в Opera. Ее код также несложен:

```
function getBrowserStrict() {
  if (navigator.userAgent.indexOf("Opera") > -1)
    return "O" //Это буква "O", а не ноль
  else
    return getBrowser();
}
```

Сначала мы проверяем, присутствует ли в значении свойства `userAgent` объекта `Navigator` строка "Opera" — как мы уже знаем, именно она в любом режиме совместимости идентифицирует одноименный Web-обозреватель. Если таковая строка присутствует, мы возвращаем букву "O" (то есть это однозначно Opera); в противном случае мы вызываем функцию `getBrowser`, написанную ранее.

Последняя функция, которую мы напишем, — это `getLanguage`. Она также не принимает параметров и возвращает двухбуквенное обозначение языка: "ru" для русского, "en" для английского и пр.

```
function getLanguage() {
  var sl;
  if (getBrowserStrict() == "FF")
    sl = navigator.language
  else
    sl = navigator.browserLanguage;
  if (sl.length > 2) sl = sl.substr(1, 2);
  return sl;
}
```

Сначала мы выясняем, каким Web-обозревателем пользуется посетитель, обращаясь к написанной ранее функции `getBrowserStrict`. Если это Firefox, мы обращаемся к свойству `language` объекта `Navigator`, чтобы получить обозначение языка, в противном случае — к свойству `browserLanguage` того же объекта. Напоследок мы проверяем, не длиннее ли полученное обозначение языка двух символов (свойство `length` объекта `String`), и, если это так, заимствуем из него только первые два символа (метод `substr` объекта `String`). Объект `String` со всеми его свойствами и методами нам уже знаком по *главе 4*.

Написанные нами три функции лучше всего поместить в особый файл сценариев (о файлах сценариев говорилось в *главе 5*). Данный файл мы будем подключать к каждой странице, где нам понадобятся эти функции.

Работа с окнами Web-обозревателя

Что ж, начали мы неплохо: узнали, как получить сведения о Web-обозревателе, и написали целых три полезных функции. На очереди — окна Web-обозревателя. Мы научимся получать о них различные сведения, перемещать, менять размеры, открывать и закрывать.

Текущее окно Web-обозревателя представляется как экземпляр объекта `Window`, созданный самим Web-обозревателем и хранящийся в переменной `window`. Этот объект поддерживает очень много свойств, методов и событий, позволяющих выполнять над окнами Web-обозревателя различные манипуляции. Давайте выясним, какие именно.

Управление размерами и местоположением окна

Сначала выясним, как получить и задать размеры и местоположение окна Web-обозревателя. Надо же с чего-то начинать…

Чтобы узнать координаты левого верхнего угла окна, мы обратимся к следующим свойствам объекта `Window`:

- `screenLeft` — горизонтальная координата (поддерживается Internet Explorer и Opera);
- `screenTop` — вертикальная координата (поддерживается Internet Explorer и Opera);
- `screenX` — горизонтальная координата (поддерживается Firefox);
- `screenY` — вертикальная координата (поддерживается Firefox).

```
if (getBrowserStrict() == "FF") {
  x = window.screenX;
  y = window.screenY;
```

```
} else {
  x = window.screenLeft;
  y = window.screenTop;
}
```

И здесь нам пригодилась написанная ранее функция `getBrowserStrict`.

А вот узнать размеры окна мы можем только в Opera и Firefox. Internet Explorer, точнее, его разработчики здесь дали маху, не предусмотрев в объекте `Window` соответствующих свойств или методов. Плохо…

Горизонтальный и вертикальный размеры окна Web-обозревателя возвращают свойства `outerWidth` и `outerHeight` объекта `Window` соответственно. Еще раз отметим, что эти свойства поддерживаются только Opera и Firefox.

```
if (getBrowserStrict() != "IE") {
  width = window.outerWidth;
  height = window.outerHeight;
}
```

Перейдем теперь конкретно к управлению размерами и местоположением окон Web-обозревателя. Для этого используются особые методы объекта `Window`.

> ***Внимание!***
>
> К сожалению, в Opera можно программно управлять размерами и местоположением только тех окон, которые вынесены за пределы главного окна. Окна, представленные в виде вкладок в главном окне этой программы, не откликаются на вызов описанных далее методов.

Метод `moveTo` задает новое местоположение окна.

```
moveTo(<горизонтальная координата>, <вертикальная координата>);
```

Новые координаты левого верхнего угла окна передаются в качестве параметров этого метода.

Метод `moveBy` позволяет сместить окно относительно текущего его местоположения.

```
moveBy(<смещение по горизонтали>, <смещение по вертикали>);
```

Смещение левого верхнего угла окна по горизонтали и вертикали также передаются в качестве параметров. При этом положительные величины указывают смещение вправо или вниз, а отрицательные — влево или вверх.

Приведем два примера использования этих методов.

```
window.moveTo(100, 100);
```

Помещаем левый верхний угол окна в точку с координатами {100, 100}.

```
window.moveBy(10, -20);
```

Смещаем левый верхний угол окна на 10 пикселов вправо и на 20 пикселов вверх.

Метод `resizeTo` меняет размеры окна.

```
resizeTo(<ширина>, <высота>);
```

Метод `resizeBy` увеличивает или уменьшает размеры окна на заданные величины приращения.

```
resizeBy(<приращение ширины>, <приращение высоты>);
```

Если задана положительная величина какого-либо `приращения`, данный размер увеличивается, если отрицательная — уменьшается.

```
window.resizeTo(300, 200);
```

Устанавливаем ширину окна в 300 пикселов, а высоту — в 200 пикселов.

```
window.resizeBy(-10, 5);
```

Уменьшаем ширину окна на 10 пикселов и увеличиваем его высоту на 5 пикселов.

Firefox также поддерживает не принимающий параметров метод `sizeToContent`. Он задает размеры окна так, чтобы в нем полностью поместилось все содержимое страницы, но не больше.

> ***НА ЗАМЕТКУ***
>
> Такой разнобой в свойствах и методах объекта `Window` и их поддержке разными Web-обозревателями вызван тем, что этот объект, равно как и все остальные объекты, рассмотренные в этой главе, не включены в стандарт DOM и вообще не стандартизированы. Так что разработчики Web-обозревателей стараются кто на что горазд...

Прокрутка содержимого окна

Еще мы можем программно прокрутить содержимое окна (то есть саму страницу) до заданной точки. Для этого используются два особых метода объекта `Window`.

Но прежде всего нужно выяснить положение полос прокрутки окна, горизонтальной и вертикальной. Неплохо было бы также выяснить, насколько мы можем прокрутить страницу по горизонтали и по вертикали, то есть полный размер страницы в соответствующем направлении. Проще всего это узнать, обратившись к четырем свойствам экземпляра объекта `HTMLElement`, представляющего секцию тела страницы; его можно получить из свойства `body` объекта `HTMLDocument`.

Вот эти свойства:

- `scrollHeight` — возвращает полную высоту страницы;
- `scrollLeft` — возвращает расстояние, на которое прокручена страница по горизонтали;
- `scrollTop` — возвращает расстояние, на которое прокручена страница по вертикали;
- `scrollWidth` — возвращает полную ширину страницы.

Свойства `scrollLeft` и `scrollTop` в случае секции тела страницы фактически возвращают положение бегунка на горизонтальной и вертикальной полосе прокрутки соответственно.

```
var scrolledX = document.body.scrollLeft;
var scrolledY = document.body.scrollTop;
```

Opera и Firefox поддерживают свойства `pageXOffset` и `pageYOffset` объекта `Window`. Эти свойства возвращают размер скрытой от посетителя в результате прокрутки части страницы, соответственно, по горизонтали и вертикали. Фактически они аналогичны свойствам `scrollWidth` и `scrollHeight`.

Firefox также поддерживает еще четыре свойства объекта `Window`, относящиеся к прокрутке содержимого окна. Свойства `scrollMaxX` и `scrollMaxY` возвращают максимальное расстояние в пикселах, на которое может быть прокручена страница, соответственно, по горизонтали и вертикали; фактически это горизонтальный или вертикальный размер самой страницы за вычетом соответствующего размера окна. А свойства `scrollX` и `scrollY`, насколько удалось выяснить автору, аналогичны уже рассмотренным свойствам `scrollLeft` и `scrollTop`.

Так, положение полос прокрутки и расстояние, на которое мы можем прокрутить содержимое окна, мы выяснили. Приступим непосредственно к прокрутке.

Метод `scrollTo` выполняет прокрутку содержимого окна до точки с заданными координатами.

```
scrollTo(<горизонтальная координата>, <вертикальная координата>);
```

Здесь координаты точки задаются относительно самой страницы, а не окна Web-обозревателя. Это понятно, поскольку точка-то присутствует на странице.

Метод `scrollBy` выполняет прокрутку содержимого окна на заданные величины смещений.

```
scrollBy(<смещение по горизонтали>, <смещение по вертикали>);
```

Положительные величины смещения вызывают прокрутку содержимого окна вправо и вниз, а отрицательные — влево и вниз.

Традиционно рассмотрим два примера.

```
window.scrollTo(0, 200);
```

Прокручиваем в точку с координатами {0, 200}.

```
window.scrollBy(0, -20);
```

Прокручиваем окно на 20 пикселов вверх.

Для прокрутки содержимого страницы до заданного элемента мы можем использовать метод `scrollIntoView` объекта `HTMLElement`. Его следует вызвать для того элемента, до которого мы хотим прокрутить страницу.

```
scrollIntoView([<выровнять по верху или по низу окна>])
```

Этот метод может принимать в качестве необязательного параметра логическую величину. Значение `true` прокручивает содержимое окна так, чтобы текущий элемент страницы появился у верхнего его края; такой же эффект даст пропуск этого параметра. Значение `false` прокручивает окно так, что элемент страницы будет находиться у нижнего края окна.

```
var parObj = document.getElementById("par");
parObj.scrollIntoView();
```

Данный сценарий прокручивает содержимое текущего окна так, чтобы абзац `par` появился в верхней его части.

> ***Внимание!***
>
> Метод `scrollIntoView`, по идее, поддерживается всеми Web-обозревателями, но почему-то не работает в Opera.

Firefox поддерживает еще два метода объекта `Window`, которые нам не мешало бы знать. Метод `scrollByLines` прокручивает содержимое окна по вертикали на заданное количество строк.

```
scrollByLines(<количество строк>)
```

Положительное значение параметра вызывают прокрутку вниз, отрицательное — вверх.

А метод `scrollByPages` прокручивает содержимое окна по вертикали на заданное количество страниц.

```
scrollByPages(<количество страниц>)
```

В терминологии Windows "строка" — это величина, на которую прокручивается содержимое страницы при нажатии на клавиши "стрелка вверх" и "стрелка вниз", а "страница" — величина, на которую прокручивается содержимое страницы при нажатии на клавиши <PgUp> и <PgDn>. Положительное значение параметра вызывает прокрутку вниз, отрицательное — вверх.

НА ЗАМЕТКУ

А самый простой способ выполнить прокрутку страницы до необходимого элемента — использовать якоря (см. *главу 2*).

Создание нового окна

Объект `Window` поддерживает замечательный метод `open`. Этот метод позволит нам создать новое окно Web-обозревателя и загрузить в него другую страницу.

```
open(<интернет-адрес вторичной страницы>, <имя создаваемого окна>,
↳[<список параметров создаваемого окна, разделенных запятыми>]);
```

Первым параметром передается `интернет-адрес вторичной страницы` в строковом виде. (*Вторичной* называется страница, загружаемая в созданном программно окне. Страница, загруженная в текущем окне и содержащая сценарий, который создал новое окно, называется *первичной*.) Если вместо него передать пустую строку, будет создано "пустое" окно.

Второй параметр задает имя создаваемого окна в строковом виде. Впоследствии мы можем использовать это имя в качестве цели гиперссылки (атрибут `TARGET` тега `<A>`; подробнее см. в *главе 2*). Если же этого не требуется, вместо имени можно передать пустую строку.

Третьим, необязательным, параметром методу передается список различных параметров создаваемого окна, разделенных запятыми, в строковом виде. Все эти параметры перечислены в табл. 7.2.

Таблица 7.2. *Доступные параметры окна, передаваемые методу* `open`

Параметр	Описание
`channelmode=yes|no`	Если `yes`, то создаваемое окно будет отображаться с панелью каналов (так называемый "режим театра"). Поддерживается только Internet Explorer
`fullscreen=yes|no`	Если `yes`, то создаваемое окно займет весь экран (так называемый "режим киоска"). Поддерживается только Internet Explorer
`height=<высота>`	Задает высоту создаваемого окна в пикселах
`left=<горизонтальная координата>`	Задает горизонтальную координату левого верхнего угла создаваемого окна. Поддерживается только Internet Explorer
`location=yes|no`	Если `yes`, созданное окно будет содержать панель ввода интернет-адреса

Таблица 7.2 *(окончание)*

Параметр	Описание
`menubar=yes|no`	Если `yes`, созданное окно будет содержать главное меню
`replace=yes|no`	Если `yes`, то интернет-адрес вторичной страницы заместит в списке истории интернет-адрес первичной страницы. Поддерживается только Internet Explorer
`resizeable=yes|no`	Если `yes`, размеры созданного окна можно будет изменять
`scrollbars=yes|no`	Если `yes`, созданное окно будет содержать полосы прокрутки
`status=yes|no`	Если `yes`, созданное окно будет содержать строку статуса
`titlebar=yes|no`	Если `yes`, созданное окно будет иметь заголовок. Поддерживается только Internet Explorer версий 4.*x* и 5.*x*
`toolbar=yes|no`	Если `yes`, созданное окно будет содержать панель инструментов
`top=`*<вертикальная координата>*	Задает вертикальную координату левого верхнего угла создаваемого окна. Поддерживается только Internet Explorer
`width=`*<ширина>*	Задает ширину создаваемого окна в пикселах

Если опустить третий параметр метода `open` или передать ему пустую строку, созданное программно окно будет иметь случайные размеры и местоположение, а также содержать заголовок, строку меню, панель инструментов, панель ввода интернет-адреса и строку статуса.

А теперь — внимание! Метод `open` возвращает экземпляр объекта `Window`, соответствующий созданному программно окну. С его помощью мы можем получить доступ к этому окну и работать с ним.

Теперь — несколько примеров.

```
var secondWnd = window.open("page3.html", "second_window");
```

Это выражение создаст новое окно Web-обозревателя, задаст для него имя `second_window` и выведет в нем страницу page3.html. Никакие дополнительные параметры создаваемого окна здесь не заданы.

Отметим, что экземпляр объекта `Window`, соответствующий созданному программно окну, мы сохранили в переменной `secondWnd`. Впоследствии мы можем использовать его для работы с данным окном и его содержимым.

Если же мы больше не собираемся трогать это окно, то можем и не сохранять возвращенный методом `open` экземпляр объекта `Window`.

```
var secondWnd = window.open("page3.html", "second_window", "height=200,
↳location=no, menubar=no, resizeable=no, scrollbars=no, status=no,
↳titlebar=no, toolbar=no, width=300");
```

А это выражение создаст окно с размерами 300×200 пикселов, без заголовка (в Internet Explorer версий 4.*x* и 5.*x*), строки меню, панели инструментов, панели адреса, строки статуса, полос прокрутки и без возможности изменить размеры окна.

> ***Внимание!***
>
> Окна Web-обозревателя без строки меню, панели инструментов, панели адреса, строки статуса, полос прокрутки и возможности изменить их размеры — очень плохой стиль Web-программирования. Лучше их избегать.

Работа с программно созданными окнами

Итак, новое окно Web-обозревателя мы создали. Как теперь с ним работать?

Мы уже знаем, что метод `open`, создающий новое окно, возвращает экземпляр объекта `Window`, соответствующий созданному программно окну. Этот экземпляр нам будет нужно сохранить в какой-либо переменной, если мы, конечно, хотим манипулировать новым окном и его содержимым.

```
var secondWnd = window.open("page3.html", "second_window");
```

Прежде всего, мы можем использовать уже изученные свойства и методы объекта `Window` для управления этим окном: перемещения, изменения размеров и прокрутки его содержимого.

```
secondWnd.moveTo(0, 0);
secondWnd.resizeTo(400, 300);
```

Далее, мы можем обращаться к переменным и функциям, объявленным в сценариях, присутствующих во вторичной странице. Формат обращения к ним таков:

<программно созданное окно>.<переменная> = <значение>;

и

<программно созданное окно>.<функция>([*<параметры>*]);

То есть такой же, как и для доступа к свойствам и методам экземпляра объекта.

```
secondWnd.someVar = 100;
secondWnd.someFunc(4, 7, "ABC");
```

Единственное — нужно, чтобы сценарии вторичного окна, объявляющие эти переменные и функции, к данному времени были выполнены. Поэтому будет лучше поместить их в секцию заголовка вторичной страницы.

Свойство `document` объекта `Window` возвращает экземпляр объекта `HTMLDocument`, представляющего загруженную в окно Web-обозревателя страницу. Этот объект и некоторые его свойства и методы были рассмотрены в *главе 5*.

> ***НА ЗАМЕТКУ***
>
> В той же *главе 5* мы узнали, что `document` — вроде как не свойство, а обычная переменная. На самом деле это в любом случае свойство, просто в случае текущей страницы Web-обозреватель для нашего удобства позволяет обращаться к нему и как к переменной.
>
> `window.document.`*`<свойство или метод>`*
>
> `document.`*`<свойство или метод>`*
>
> Оба приведенных обращения к экземпляру объекта `HTMLDocument`, представляющему текущую страницу, равноправны. Но если нам нужно обратиться к странице, открытой в созданном программно окне Web-обозревателя, мы должны использовать свойство `document` экземпляра объекта `Window`, возвращенного методом `open`:
>
> `secondWnd.document.`*`<свойство или метод>`*

В качестве примера предположим, что во вторичной странице присутствует абзац с именем `par`. Тогда в сценарии, находящемся в первичной странице, мы можем изменить его текст с помощью таких выражений:

```
var parObj = secondWnd.document.getElementById("par");
parObj.firstChild.nodeValue = "Новый текст";
```

Часто бывает нужно получить доступ из созданного программно окна к окну, его создавшему. Для этого предназначено свойство `opener` объекта `Window`. Оно возвращает экземпляр объекта `Window`, представляющий окно, которое создало данное окно. Понятно, что использовать его имеет смысл только в сценариях, присутствующих в страницах, которые будут открываться в созданных программно окнах в качестве вторичных.

```
var parObj = window.opener.document.getElementById("par");
parObj.firstChild.nodeValue = "Привет от окна, созданного программно!";
```

Приведенный сценарий, будучи помещенным во вторичную страницу, ищет на странице, создавшей новое окно, абзац `par` и задает для него новый текст.

Переключение между окнами

Объект `Window` позволяет активизировать заданное окно Web-обозревателя и деактивизировать его, убрав "в тень". Этого практически всегда бывает достаточно.

Метод `focus` активирует текущее окно. Он не принимает параметров.

```
window.focus();
secondWnd.focus();
```

Здесь переменная `secondWnd` хранит экземпляр объекта `Window`, соответствующий созданному программно окну.

Также не принимающий параметров метод `blur` деактивирует текущее окно и активирует другое окно Web-обозревателя.

```
secondWnd.blur();
```

> ***НА ЗАМЕТКУ***
>
> Предугадать, какое окно будет активировано после вызова метода `blur`, невозможно. Это может быть даже окно не Web-обозревателя, а какой-то другой программы.

Закрытие окна

Если объект `Window` позволяет программно создать новое окно Web-обозревателя, по идее, он должен предусматривать средства для его закрытия. Этим занимается не принимающий параметров метод `close`.

```
secondWnd.close();
```

> ***ВНИМАНИЕ!***
>
> Окна Web-обозревателя, созданные программно (методом `open`), при вызове метода `close` будут закрыты без предупреждения. Но при закрытии окон, открытых самим посетителем, на экране появится окно-предупреждение, спрашивающее, действительно ли следует закрыть это окно. Вывод этого предупреждения невозможно запретить ни из Web-сценариев, ни в настройках Web-обозревателя.

Если мы собираемся программно создавать и закрывать окна Web-обозревателя, нам также пригодится свойство `closed` объекта `Window`. Это свойство возвращает `true`, если текущее окно было закрыто посетителем или вызовом метода `close`, и `false`, если оно еще присутствует на экране.

```
if (secondWnd.closed) {
  var parObj = secondWnd.document.getElementById("par");
  parObj.firstChild.nodeValue = "Меня еще не закрыли!";
}
```

Данный сценарий проверяет, присутствует ли на экране окно `secondWnd`, созданное программно, и, если это так, выводит в абзаце `par` этого окна новый текст.

Прочие манипуляции с окнами

Давайте рассмотрим еще несколько свойств и методов объекта `Window`, которые пригодятся нам при манипуляциях с окнами Web-обозревателя.

Метод `home` вызывает открытие в Web-обозревателе "домашней" страницы, заданной в его настройках. Он не принимает параметров и поддерживается только Opera и Firefox.

Метод `print` выводит на экран диалоговое окно печати Web-обозревателя, позволяющее запустить печать текущей страницы на принтере. Он не принимает параметров и поддерживается, по идее, всеми Web-обозревателями, но почему-то не работает в Opera.

Метод `stop` прерывает загрузку страницы в текущем окне. Он не принимает параметров и поддерживается только Opera и Firefox.

Свойство `history` возвращает ссылку на экземпляр объекта `History`, представляющий историю Web-обозревателя. Этот объект представляет набор методов и свойств, с помощью которых мы можем перемещаться по истории. Мы рассмотрим его далее в этой главе.

Свойство `location` возвращает ссылку на экземпляр объекта `Location`, представляющий сведения об интернет-адресе, с которого была загружена текущая страница. Этот объект мы рассмотрим чуть позже.

Свойство `name` возвращает имя окна, заданное при его создании как один из параметров метода `open` (см. ранее). Если окно не имеет имени, возвращается пустая строка.

Свойство `navigator` аналогично по назначению переменной `navigator` и возвращает экземпляр объекта `Navigator`, предоставляющий сведения о Web-обозревателе. Данный объект был рассмотрен в начале этой главы.

> ***НА ЗАМЕТКУ***
>
> Здесь та же самая ситуация, что и со свойством `document`. На самом деле переменная `navigator` суть свойство объекта `Window`, просто в случае текущей страницы Web-обозреватель для нашего удобства позволяет обращаться к нему и как к переменной, и как к свойству.

Свойство `screen` возвращает ссылку на экземпляр объекта `Screen`, представляющий сведения о видеоподсистеме клиентского компьютера. Мы рассмотрим его в конце этой главы.

События объекта *Window*

Когда мы только что познакомились с объектом `Window`, то узнали, что он поддерживает также и некоторые события. Настала пора поговорить о них.

Объект `Window` поддерживает достаточно много событий. Большинство из них перечислено в табл. 7.3.

Таблица 7.3. *События объекта* `Window`

Событие	Строка DOM	Описание
`onAbort`	"abort"	Возникает при прерывании загрузки Web-страницы. Поддерживается только Opera и Firefox
`onAfterPrint`		Возникает сразу после печати страницы. Поддерживается только Internet Explorer
`onBeforePrint`		Возникает непосредственно перед печатью страницы. Поддерживается только Internet Explorer
`onBeforeUnload`		Возникает непосредственно перед уходом с текущей страницы. Поддерживается только Internet Explorer и Firefox
`onBlur`	"blur"	Возникает при деактивации текущего окна
`onError`	"error"	Возникает при неудачной загрузке страницы или ошибке в загрузочном Web-сценарии
`onFocus`	"focus"	Возникает при активации текущего окна
`onHelp`		Возникает, когда посетитель нажимает клавишу <F1>, чтобы вывести на экран интерактивную справку, перед собственно выводом справки. Поддерживается только Internet Explorer
`onLoad`	"load"	Возникает после завершения загрузки страницы
`onResize`	"resize"	Возникает при изменении размеров окна Web-обозревателя посетителем
`onScroll`	"scroll"	Возникает при прокрутке содержимого текущего окна
`onUnload`	"unload"	Возникает после ухода с текущей Web-страницы

Кроме того, объект `Window` в Opera и Firefox поддерживает события, поддерживаемые объектами `HTMLElement`, `HTMLFormElement` и объектами, которые представляют элементы управления. (Об объекте `HTMLFormElement` представляющем HTML-форму, и об объектах, представляющих элементы управления, пойдет речь в *главе 12*. О событиях, поддерживаемых объектом `HTMLElement`, будет рассказано в *главе 8*.)

Для обработки событий объекта `Window` можно использовать как модель Internet Explorer (только соответствующие свойства, ведь мы не можем привязать событие к окну через атрибуты), так и модель Firefox. Объект `Window`, как и объект `HTMLElement`, поддерживает методы `addEventListener` и `removeEventListener`.

```
// Используется модель Internet Explorer
window.onload = windowOnload;
// Используется модель Firefox
window.addEventListener("load", windowOnload, false);
```

Здесь `windowOnload` — объявленная ранее функция — обработчик события `onLoad`.

Теперь в качестве примеров давайте рассмотрим две Web-страницы, содержащие обработчики различных событий объекта `Window`.

Далее приведен HTML-код страницы, отслеживающей изменение размеров окна Web-обозревателя, в котором она загружена, и выводящей их на экран. К сожалению, она не будет работать в Internet Explorer, который не позволяет узнать размеры окна.

```
<HTML>
  <HEAD>
    <TITLE>Сведения о Web-странице</TITLE>
    <SCRIPT>
      function windowResize() {
        var parObj = document.getElementById("par");
        parObj.firstChild.nodeValue = "Ширина: " + window.outerWidth +
          "px, высота: " + window.outerHeight + "px";
      }
      window.onresize = windowResize;
    </SCRIPT>
  </HEAD>
  <BODY>
    <P ID="par"> </P>
  </BODY>
</HTML>
```

В единственном сценарии этой страницы мы объявляем функцию `windowResize`, которая станет обработчиком события `onResize` объекта `Window`. Эта функция просто выводит текущие размеры окна в абзац `par`.

В теле функции `windowResize` мы использовали свойства `outerWidth` и `outerHeight` объекта `Window`, возвращающие, соответственно, ширину и вы-

соту текущего окна. Как мы помним из сказанного ранее, эти свойства поддерживают только Opera и Firefox — вот поэтому данная страница и не будет работать в Internet Explorer.

Далее мы привязываем функцию `windowResize` к событию `onResize` окна, присваивая ее свойству `onresize` объекта `Window`, "ответственному" за это событие. Все, теперь объявленная нами функция — обработчик события будет выполняться при изменении размеров окна.

Заметим также, что мы поместили в абзац `par` текстовое содержимое, в нашем случае — неразрывный пробел:

```
<P ID="par"> </P>
```

(Вообще, мы можем поместить в него любое содержимое — в данном случае это непринципиально.) Это содержимое будет изменяться функцией `windowResize`, а значит, уже должно присутствовать в абзаце.

Вторая страница, которую мы рассмотрим, содержит абзац `par`, который при деактивации окна скрывается, а при активации снова появляется. Эта страница будет работать во всех Web-обозревателях.

```
<HTML>
  <HEAD>
    <TITLE>Прячущийся абзац</TITLE>
    <SCRIPT>
      function windowFocus() {
        var parObj = document.getElementById("par");
        parObj.style.display = "block";
      }
      function windowBlur() {
        var parObj = document.getElementById("par");
        parObj.style.display = "none";
      }
      window.onfocus = windowFocus;
      window.onblur = windowBlur;
    </SCRIPT>
  </HEAD>
  <BODY>
    <P ID="par">Привет!</P>
  </BODY>
</HTML>
```

Этот эффект достигается обработкой событий `onFocus` и `onBlur` объекта `Window`. Обработчик первого события выводит абзац `par` на экран, а обработчик второго события скрывает его. Сейчас мы рассмотрим, как это делается.

Сначала мы получаем доступ к экземпляру объекта `CSSStyle`, представляющему стиль CSS, привязанный к абзацу `par`. Этот экземпляр доступен через свойство `style`. Поскольку мы не привязали к этому абзацу никакого стиля, сам Web-обозреватель формирует стиль, содержащий параметры, привязываемые к данному элементу страницы по умолчанию.

Получив экземпляр объекта `CSSStyle`, мы обращаемся к его свойству `display`, соответствующему одноименному атрибуту стиля. (О стилях CSS и их атрибутах было рассказано в *главе 3*.) Данный атрибут стиля управляет поведением элемента на странице и, в частности, может скрыть его, как будто данный элемент вообще не присутствует в HTML-коде страницы.

Для скрытия абзаца `par` мы присваиваем свойству `display` объекта `CSSRule` значение `"none"` — это выполняется в обработчике события `onBlur`. А обработчик события `onFocus`, чтобы снова вывести этот абзац на экран, присваивает свойству `display` объекта `CSSRule` значение `"block"`.

Кстати, если нужно временно скрыть какой-то элемент страницы из Web-сценария, а потом его снова вывести на страницу, зачастую используется именно такой подход. Нужно только заметить, что значение `"none"` свойства (и атрибута стиля) `display` скрывает элемент страницы так, словно он вообще не объявлен в HTML-коде. (Собственно, об этом уже говорилось в *главе 3*.) Иногда это бывает полезно, иногда — нет.

> ***НА ЗАМЕТКУ***
>
> Другой способ временно скрыть элемент страницы — использовать свойство `visibility` объекта `CSSStyle`, соответствующее одноименному атрибуту стиля (см. *главу 3*). Значение `"hidden"` данного свойства скрывает элемент страницы, причем выделенное под него место на странице останется за ним; значение `"visible"` делает элемент страницы видимым.

На этом с управлением окнами Web-обозревателя можно закончить. Впоследствии мы еще вернемся к этому вопросу, когда будем говорить о работе с данными и рассматривать специфические особенности различных Web-обозревателей. О-о-о, они еще таят для нас сюрпризы!

Работа с интернет-адресом текущей страницы

Следующий вопрос, который нам нужно рассмотреть, — это получение сведений об интернет-адресе страницы, открытой в данный момент в Web-обозревателе, — текущей страницы. Что нам припасли для этого Web-обозреватели?

Для программного доступа к интернет-адресу текущей страницы и к отдельным его частям используется объект `Location`. Единственный экземпляр этого объекта создается самим Web-обозревателем и доступен через свойство `location` объекта `Window`. Это свойство мы уже рассмотрели, когда говорили об управлении окнами Web-обозревателя.

Для нашего удобства Web-обозреватель позволяет обращаться к свойству `location` экземпляра объекта `Window`, представляющего текущую страницу, как к переменной:

`window.location.`*`<свойство или метод>`*

`location.`*`<свойство или метод>`*

Мы также можем обратиться к экземпляру объекта `Location`, соответствующему программно созданному окну:

`<программно созданное окно>``.location.`*`<свойство или метод>`*

Благо программно создавать новые окна Web-обозревателя мы уже умеем.

Объект `Location` поддерживает множество свойств и несколько методов. Все его свойства перечислены в табл. 7.4. Они возвращают различные составляющие интернет-адреса текущей страницы: интернет-адрес Web-сервера, номер порта и пр. Все эти значения возвращаются в строковом виде.

Таблица 7.4. *Свойства объекта* `Location`

Свойство	Описание
`hash`	Имя якоря, если оно есть, без символа `#`
`host`	Интернет-адрес Web-сервера с номером порта, если порт указан
`hostname`	Интернет-адрес Web-сервера
`href`	Полный интернет-адрес страницы
`pathname`	Путь и имя файла, в котором хранится текущая страница, с начальным символом слеша
`port`	Номер порта без символа двоеточия. Если порт не указан, возвращается номер порта по умолчанию для протокола HTTP — "80"
`protocol`	Обозначение протокола с конечным символом двоеточия
`search`	Строка параметров, если она есть, с начальным символом вопросительного знака (о строке параметров будет рассказано далее)

Для примера предположим, что мы загрузили страницу с интернет-адреса

http://www.somesite.ru:8000/pages/page2.html

Тогда различные свойства объекта `Location` вернут следующие значения:

- `hash` — `null`;
- `host` — "www.somesite.ru:8000";
- `hostname` — "www.somesite.ru";
- `href` — "http://www.somesite.ru:8000/pages/page2.html";
- `pathname` — "/pages/page2.html";
- `port` — "8000";
- `protocol` — "http:";
- `search` — `null`.

Если мы воспользуемся якорем `anchor2` для перехода на другую часть этой страницы, то впоследствии свойство `hash` вернет значение "anchor2".

А теперь рассмотрим такой интернет-адрес:

http://www.othersite.ru/engine/search.exe?key=str

Здесь после вопросительного знака приведена так называемая *строка параметров*, которые передаются программе search.exe, хранящейся и выполняющейся прямо на серверном компьютере. (О серверных программах и передаче им параметров будет рассказано в *главе 12*.) В полученной в результате странице мы получим такие значения свойств объекта `Location`:

- `hash` — `null`;
- `host` — "www.othersite.ru";
- `hostname` — "www.othersite.ru";
- `href` — "http://www.othersite.ru/engine/search.exe?key=str";
- `pathname` — "/engine/search.exe";
- `port` — "80" (номер порта HTTP по умолчанию);
- `protocol` — "http:";
- `search` — "?key=str".

Свойство `href` можно использовать для принудительной загрузки другой страницы или, как говорят бывалые интернетчики, *перенаправления* посетителя на другую страницу. Для этого достаточно присвоить нужный интернет-адрес этому свойству:

```
location.href = "http://www.thirdsite.ru/";
```

Осталось поговорить о методах объекта `Location`. Всего их два.

Метод `replace` загружает в Web-обозревателе страницу, интернет-адрес которой был передан в качестве единственного параметра. Фактически он также выполняет перенаправление на другую страницу.

```
replace(<интернет-адрес целевой страницы>)
```

У этого метода есть одна интересная особенность. Он заменяет в истории Web-обозревателя интернет-адрес текущей страницы на интернет-адрес целевой. Это может быть полезно в случае, если страница по умолчанию нашего сайта выполняет определение программы Web-обозревателя и ее языка и перенаправление посетителя на соответствующую версию сайта. Если так, то странице по умолчанию особо нечего делать в истории Web-обозревателя, не так ли?

```
if (getBrowser() == "FF") {
  location.replace("index_ff.html")
else
  location.replace("index_ie.html");
```

Здесь мы перенаправляем посетителя на разные страницы в зависимости от Web-обозревателя. Для определения Web-обозревателя мы используем написанную ранее функцию `getBrowser`.

Метод `reload` перезагружает текущую страницу.

```
reload([<загрузить напрямую с Web-сервера>])
```

В качестве единственного необязательного параметра этот метод принимает логическое значение. Значение `false` предписывает загрузить страницу из кэша Web-обозревателя, если, конечно, данная страница там присутствует (это поведение по умолчанию, если параметр опущен); в противном случае страница будет загружена с Web-сервера. Значение `true` выполняет загрузку страницы прямо с Web-сервера, даже если она присутствует в кэше.

```
secondWnd.location.reload(true);
```

Это выражение перезагружает страницу, открытую в программно созданном окне `secondWnd`. Причем перезагрузка выполняется прямо с Web-сервера, даже если данная страница присутствует в кэше.

Работа с историей Web-обозревателя

История Web-обозревателя — это список страниц, посещенных посетителем за сеанс работы с Web-обозревателем. Посетитель может перемещаться по этому списку с помощью кнопок **Вперед** и **Назад** (**Forward** и **Back**) панели инструментов Web-обозревателя, соответствующих пунктов его главного меню или вызвав окно со списком истории и щелкнув в этом списке пункт, соответствующий нужной странице.

Web-обозреватель также предоставляет средства для работы со списком истории из Web-сценариев. Для этого используются свойства и методы объекта `History`, который как раз и предоставляет программный доступ к истории. Единственный экземпляр этого объекта создается самим Web-обозревателем и доступен через свойство `history` объекта `Window`.

Для нашего удобства Web-обозреватель позволяет обращаться к свойству `history` экземпляра объекта `Window`, представляющего текущую страницу, как к переменной:

```
window.history.<свойство или метод>
history.<свойство или метод>
```

Мы также можем обратиться к экземпляру объекта `History`, соответствующему программно созданному окну:

```
<программно созданное окно>.history.<свойство или метод>
```

Свойство `length` объекта `History` возвращает количество страниц, находящихся в списке истории.

Метод `back` загружает страницу, являющуюся предыдущей в списке истории.

```
back([<дистанция>])
```

Как видно, этот метод может принимать один необязательный параметр. Это число указывает относительную позицию списка истории, на которой находится требуемая страница, отсчитываемую от текущей страницы к началу списка ("назад").

```
history.back();
```

Это выражение загружает в текущее окно предыдущую страницу в списке истории.

```
history.back(3);
```

А это выражение загрузит страницу, находящуюся на третьей по счету позиции списка истории, если считать от текущей страницы "назад".

Метод `forward` загружает страницу, являющуюся следующей в списке истории.

```
forward([<дистанция>])
```

Назначение единственного необязательного параметра то же, что у метода `back`. Только в этом случае относительная позиция списка истории отсчитывается от текущей страницы к концу списка ("вперед").

```
history.forward();
```

Загружаем следующую страницу в списке истории.

```
history.forward(2);
```

Загружаем страницу, находящуюся на второй по счету позиции списка истории, если считать от текущей страницы "вперед".

Метод `go` позволяет загрузить любую страницу в списке истории.

```
go(<интернет-адрес>|<дистанция>)
```

Как видим, здесь мы можем указать в качестве параметра либо интернет-адрес нужной страницы в виде строки, либо дистанцию в списке истории в виде числа, как в случае методов `back` и `forward`. В последнем случае, если указать положительное число, отсчет в списке истории будет вестись от текущей страницы к его концу ("вперед"), а если отрицательное — к началу списка ("назад").

```
history.go("http://www.somesite.ru");
```

Загружаем страницу с интернет-адреса **http://www.somesite.ru**.

```
history.go(-3);
```

Загружаем страницу, находящуюся на третьей по счету позиции списка истории, если считать от текущей страницы "назад".

```
history.go(1);
```

Загружаем следующую страницу в списке истории.

Вместо последних двух выражений мы можем использовать следующие:

```
history.back(2);
history.forward();
```

Просто метод `go` более универсален.

Получение сведений о видеоподсистеме клиентского компьютера

Очень часто бывает нужно получить сведения о видеоподсистеме клиентского компьютера: разрешении экрана, количестве отображаемых цветов (цветности) и пр. Для этого используется объект `Screen`, свойства которого возвращают нужные данные. Единственный экземпляр этого объекта создается самим Web-обозревателем и доступен через свойство `screen` объекта `Window`.

Для нашего удобства Web-обозреватель позволяет обращаться к свойству `screen` экземпляра объекта `Window`, представляющего текущую страницу, как к переменной:

```
window.screen.<свойство>
screen.<свойство>
```

Объект `Screen` поддерживает только свойства. Большинство из них, для нас самые полезные, приведены в табл. 7.5.

Таблица 7.5. *Свойства объекта* `Screen`

Свойство	Описание
`availHeight`	Возвращает высоту свободной области экрана, не занятой панелью задач и прочими панелями
`availLeft`	Возвращает горизонтальную координату верхней левой точки свободной области экрана, не занятой панелью задач и прочими панелями. Поддерживается только Firefox
`availTop`	Возвращает вертикальную координату верхней левой точки свободной области экрана, не занятой панелью задач и прочими панелями. Поддерживается только Firefox
`availWidth`	Возвращает ширину свободной области экрана, не занятой панелью задач и прочими панелями
`colorDepth`	Возвращает число, обозначающее цветность экрана. Для 16 цветов возвращается значение 2, для 256 — 8, для 16,7 миллионов цветов (режим HiColor) — 32
`fontSmoothingEnabled`	Возвращает `true`, если в настройках операционной системы клиента включено сглаживание шрифтов, и `false` в противном случае. Поддерживается только Internet Explorer
`height`	Возвращает полную высоту экрана
`left`	Возвращает горизонтальную координату верхней левой точки экрана. Поддерживается только Firefox
`top`	Возвращает вертикальную координату верхней левой точки экрана. Поддерживается только Firefox
`width`	Возвращает полную ширину экрана

Для примера давайте напишем страницу, которая будет "растягивать" окно Web-обозревателя вдоль правого края экрана клиентского компьютера.

```
<HTML>
  <HEAD>
    <TITLE>Позиционирование окна</TITLE>
    <SCRIPT>
      function bodyLoad() {
        var screenWidth = screen.availWidth;
        var screenHeight = screen.availHeight;
        var windowWidth = 300;
        window.moveTo(screenWidth - windowWidth, 0);
```

```
        window.resizeTo(windowWidth, screenHeight);
      }
    </SCRIPT>
  </HEAD>
  <BODY ONLOAD="bodyLoad();">
    <P>Привет!</P>
  </BODY>
</HTML>
```

Здесь мы принудительно устанавливаем ширину окна равной 300 пикселов. Остальной код не требует пояснений.

> ***На заметку***
>
> Лучше использовать для выяснения разрешения экрана свойства `availWidth` и `availHeight`, а не `width` и `height`. Последние два свойства не учитывают, что системная панель задач у пользователя может быть настроена так, чтобы присутствовать на экране постоянно, в результате полезная площадь экрана уменьшится. Кроме того, на экране могут присутствовать другие панели, созданные как самой Windows, так и сторонними программами.

Доступ к содержимому фреймов

Напоследок мы поговорим о фреймах, точнее — о доступе к их содержимому. Если мы собираемся использовать фреймы, это нам пригодится.

Прежде всего нужно уяснить, что фрейм с точки зрения Web-обозревателя — своего рода "форточка", независимое окно, в котором загружена отдельная страница. (Об этом уже говорилось в *главе 2*, в параграфе, посвященном фреймам.) Эту "форточку" представляет экземпляр объекта `Window`, созданный самим Web-обозревателем и помещенный в переменную `window`. То есть если мы откроем какую-либо страницу во фрейме, переменная `window` в сценариях на этой странице будет ссылаться не на окно Web-обозревателя, а на текущий фрейм.

```
window.location.href = "page2.html";
location.href = "page2.html";
```

Так, любое из приведенных выражений откроет страницу page2.html в текущем фрейме. (Как говорилось ранее, при обращении к переменной `location` фактически происходит обращение к одноименному свойству экземпляра объекта `Window`, сохраненному в переменной `window`.)

Но что делать, если нам понадобится получить доступ к странице, загруженной в другом фрейме набора? Или если нам нужно загрузить новую страницу

во всем окне Web-обозревателя, заменив ей набор фреймов? Давайте выясним это.

Прежде всего, нам следует познакомиться с двумя свойствами объекта `Window`, имеющими отношение к фреймам и наборам фреймов:

- `parent` возвращает экземпляр объекта `Window`, представляющий окно Web-обозревателя или фрейм, в котором загружена страница набора фреймов;
- `top` возвращает экземпляр объекта `Window`, представляющий окно Web-обозревателя, в котором загружена страница самого верхнего набора фреймов в иерархии.

> ***НА ЗАМЕТКУ***
>
> Объект `Window` поддерживает также свойство `self`. Оно возвращает экземпляр объекта `Window`, предоставляющий текущий фрейм или текущее окно. Зачем оно нужно — непонятно.

Свойства `parent` и `top` имеют одно различие, о котором мы сейчас поговорим. Предположим, что мы создали страницу frameset1.html, содержащую набор из двух фреймов: `frame1` и `frame2`. Во фрейм `frame1` мы загрузили обычную Web-страницу page1.html, а во фрейм `frame2` — страницу frameset2.html, содержащую другой набор фреймов. Последний содержит фреймы `frame3` (в него загружена страница page3.html) и `frame4` (страница page4.html).

Теперь, если мы обратимся к свойству `parent` объекта `Window` из сценария страницы page1.html (фрейм `frame1` "внешнего" набора frameset1.html), то получим экземпляр объекта `Window`, представляющий окно Web-обозревателя, в который загружен "внешний" набор фреймов frameset1.html. То же самое мы получим, если обратимся к свойству `top` из этой же страницы. Видно, что в этом случае свойства `parent` и `top` выполняют одинаковую функцию.

Совсем другая картина появится, если мы попытаемся обратиться к этим свойствам из сценария, помещенного в страницу page3.html или page4.html (соответственно фреймы `frame3` и `frame4` "внутреннего" набора frameset2.html). Здесь свойство `parent` вернет экземпляр объекта `Window`, представляющий фрейм `frame2` "внешнего" набора frameset1.html, в котором загружен "внутренний" набор frameset2.html. Свойство `top` же вернет экземпляр объекта `Window`, представляющий окно Web-обозревателя, в котором загружен "внешний" набор frameset1.html.

> ***НА ЗАМЕТКУ***
>
> Вообще, отдельная страница набора фреймов, открытая во фрейме другого набора, — не очень хороший стиль Web-дизайна. Хотя иногда такая ситуация встречается. Например, некоторые бесплатные хостинг-провайдеры открывают

все страницы пользователя во фрейме (другие фреймы могут быть заняты рекламой), и если сайт пользователя построен на основе фреймов, мы получим как раз такой случай.

Итак, экземпляр объекта `Window`, представляющий окно, в котором открыт набор фреймов, мы получили. Как теперь добраться до нужного фрейма?

Специально для такого случая объект `Window` поддерживает свойство `frames`. Оно возвращает одноименную коллекцию, которая является экземпляром объекта `WindowCollection` и содержит все фреймы данного набора, также представленные в виде экземпляров объекта `Window`. (О коллекциях было рассказано в *главе 5*.) Для поиска нужного фрейма в этой коллекции мы можем использовать как численный индекс нужного фрейма, так и его буквенное имя, заданное атрибутом `NAME` тега `<FRAME>`.

Внимание!

Имена фреймов следует задавать только с помощью атрибута `NAME` тега `<FRAME>`, который хоть и объявлен устаревшим и не рекомендован к использованию, но поддерживается всеми Web-обозревателями. Что касается стандартного атрибута `ID`, то для тега `<FRAME>` он почему-то не поддерживается. Фреймы до сих пор толком не стандартизованы...

Приведем пару примеров, взяв в качестве "подопытного кролика" описанный ранее набор фреймов.

```
// Страница page1.html
var frame3Obj = window.parent.frames["frame3"];
```

Данное выражение, присутствующее в сценарии на странице page1.html "внешнего" набора фреймов frameset1.html, поместит в переменную `frame3Obj` экземпляр объекта `Window`, представляющий фрейм `frame3` "внутреннего" набора frameset2.html.

```
// Страница page4.html
var frame2Obj = window.top.frames["frame2"];
```

А это выражение из сценария на странице page4.html "внутреннего" набора фреймов frameset2.html поместит в переменную `frame2Obj` экземпляр объекта `Window`, представляющий фрейм `frame2` "внешнего" набора frameset1.html.

Поскольку фрейм суть экземпляр объекта `Window`, мы можем пользоваться всеми знакомыми нам свойствами и методами этого объекта для управления содержимым фрейма.

```
// Страница page1.html
var frame3Obj = window.parent.frames["frame3"];
frame3Obj.document.write("<H1>Всем привет от соседа!</H>");
```

Это выражение поместит на страницу page3.html, загруженную во фрейме `frame3`, текст приветствия от "соседнего" фрейма `frame1`.

```
// Страница page4.html
var frame1Obj = window.top.frames["frame1"];
frame1Obj.location.href = "page5.html";
```

Это выражение откроет во фрейме `frame1` страницу page5.html.

```
window.top.location = "http://www.somesite.ru";
```

А это выражение загрузит во все окно Web-обозревателя страницу по умолчанию сайта **http://www.somesite.ru**, заменив набор фреймов.

Коллекция `frames`, как и все остальные коллекции, рассмотренные нами в *главе 5*, поддерживает свойство `length`, возвращающее количество фреймов в наборе. Если же фреймы отсутствуют, возвращается число 0.

Зачем это может пригодиться? Иногда бывает нужно выяснить, загружена ли какая-то страница во всем окне Web-обозревателя или же во фрейме. Для этого достаточно проверить значение, возвращаемое свойством `length`: если оно равно 0, значит, страница загружена во все окно (то есть фреймов нет).

```
if (window.top.frames.length == 0)
  // Страница загружена во все окно Web-обозревателя
else
  // Страница загружена во фрейм
```

Вот, пожалуй, и все о доступе к содержимому фреймов.

Что дальше?

В этой главе мы научились работать с Web-обозревателем: получать сведения о нем самом, об интернет-адресе загруженной в нем страницы, о видеоподсистеме клиентского компьютера, управлять его окнами и "гулять" по его истории. Насыщенной получилась глава, хоть и не очень длинной.

После Web-обозревателя на очереди — сама Web-страница. В следующей главе мы научимся изменять ее содержимое из сценариев. Вот она как раз будет довольно длинной. Морально готовимся!

Глава 8

Управление содержимым Web-страницы

Вот мы и подошли к самому интересному — управлению содержимым Web-страницы, загруженной в Web-обозревателе. В этой главе мы научимся менять содержимое страницы "на лету", во время ее загрузки или в ответ на действия посетителя: добавлять новые элементы, изменять и удалять уже существующие.

Все ключевые объекты, представляющие различные элементы страницы и саму страницу, а также основные способы работы с ними нам уже знакомы по *главе 5*. Давайте кратко их перечислим.

- Объект `Window` представляет окно Web-обозревателя. Экземпляр этого объекта, представляющий текущее окно, создается самим Web-обозревателем и хранится в переменной `window`.
- Объект `HTMLDocument` представляет саму Web-страницу. Единственный экземпляр этого объекта создается самим Web-обозревателем и хранится в свойстве `document` объекта `Window`.
- Объект `HTMLElement` представляет базовую функциональность элемента страницы: абзаца, заголовка, графического изображения, ячейки таблицы и даже самой секции тела страницы.
- Объекты, представляющие конкретные элементы страницы (так, объект `HTMLParagraphElement` представляет абзац, `HTMLHeadingElement` — заголовок, `HTMLImageElement` — изображение, а `HTMLBodyElement` — секцию тела страницы), являются потомками объекта `HTMLElement`. Они предоставляют функциональность, специфичную для данного элемента страницы.
- Для каждого из элементов страницы Web-обозреватель создает экземпляр соответствующего объекта. Для доступа к нему мы можем использовать любой из способов, описанных в *главе 5*, наиболее подходящий в данный момент.

- Объект `HTMLDocument` и потомки объекта `HTMLElement` представляют набор свойств и методов, позволяющих нам получать различные параметры элемента страницы и самой страницы и менять их. Также эти объекты поддерживают довольно широкий набор событий, которые мы можем обрабатывать. (О событиях и их обработке было рассказано в *главе 6*.)

Так, фундаментальные знания мы получили. Осталось выяснить, как применять их на практике. Этим мы сейчас и займемся.

Работа с содержимым страницы

Начнем мы с изменения содержимого страницы, уже загруженной в окно Web-обозревателя. Web-программисты занимаются этим постоянно.

Изменение названия страницы

Проще всего изменить название страницы. Для этого достаточно воспользоваться свойством `title` объекта `HTMLDocument`.

```
document.title = "Новое название";
```

Это выражение меняет название текущей страницы.

```
window.top.document.title = "Это набор фреймов";
```

А это выражение, будучи размещенным в странице, загруженной в одном из фреймов, меняет название страницы, в которой определен сам набор фреймов.

В *главе 2* было описано создание фреймов. Как мы помним, в окно Web-обозревателя загружается страница с набором фреймов, а страницы, включающие фрагменты содержимого, открываются в отдельных фреймах этого набора. В заголовке окна Web-обозревателя при этом отображается название страницы с набором фреймов, и впоследствии оно не меняется. Но, пользуясь описанным здесь приемом, мы все-таки можем его изменить.

Изменение содержимого страницы

Поменять содержимое страницы также не очень сложно. Здесь Web-обозреватели предоставляют нам целых три способа. Выбирай — не хочу!

Использование методов *write* и *writeln* объекта *HTMLDocument*

Самый простой способ изменить содержимое страницы — воспользоваться уже знакомым нам по *главе 5* методом `write` объекта `HTMLDocument`.

Метод `write` в качестве единственного параметра принимает строку с HTML-кодом. Web-обозреватель обработает этот HTML-код и вставит результат его обработки в то место кода страницы, где встретился вызов данного метода.

```
write(<вставляемая строка>);
```

Приведем пару примеров.

```
write("<P>Это новый абзац.</P>");
```

Это выражение поместит на страницу новый абзац в том месте, где оно присутствует.

```
write("<TABLE>");
write("<TR>");
write("<TD>Ячейка 1</TD>");
write("<TD>Ячейка 2</TD>");
write("</TR>");
write("<TR>");
write("<TD>Ячейка 3</TD>");
write("<TD>Ячейка 4</TD>");
write("</TABLE>");
```

А этот сценарий поместит на страницу таблицу из четырех ячеек.

А теперь — очень важный момент! Метод `write` ведет себя по-разному в зависимости от того, в какой момент выполняется его вызов.

- Если метод `write` вызывается во время загрузки страницы (в загрузочном сценарии), он вставит новое содержимое на страницу в том месте, где встретился его вызов. Уже присутствующее на странице содержимое, определенное в ее HTML-коде или созданное предыдущими вызовами метода `write`, будет сохранено.
- Если метод `write` вызывается после окончания загрузки страницы (например, в обработчике события), он вставит новое содержимое на страницу, предварительно удалив старое, определенное в ее HTML-коде или созданное вызовами метода `write` во время ее загрузки.

Давайте для примера создадим новое окно Web-обозревателя и загрузим в него какую-либо страницу.

```
var secondWnd = window.open("somepage.html", "second_window");
```

Полюбуемся на открытую в новом окне страницу somepage.html и в первичной странице каким-либо образом (например, в обработчике событий) выполним следующий сценарий:

```
secondWnd.document.write("<P>Привет!</P>");
```

Мы увидим, что первоначальное содержимое страницы somepage.html, определенное в ее HTML-коде, пропадет, и его заменит абзац с текстом "Привет!".

Как видим, вызов метода `write` после загрузки страницы полностью удалил ее содержимое.

Метод `writeln` объекта `HTMLDocument` выполняет то же действие, что метод `write`, за одним исключением. Он после каждой строки, вставленной в HTML-код страницы, выполняет перенос строк. Поскольку Web-обозреватель игнорирует все переносы строк в HTML-коде (если, конечно, они не присутствуют в содержимом тега `<PRE>`; подробнее см. в *главе 2*), результат действия этого метода полностью аналогичен таковому у метода `write`.

```
writeln(<вставляемая строка>);
```

> ***НА ЗАМЕТКУ***
>
> Некоторые Web-обозреватели позволяют просмотреть результирующий HTML-код страницы, который получится после выполнения всех сценариев, изменивших ее содержимое. Так что, вероятно, лучше все-таки использовать метод `writeln`, так как сформированный им код выглядит более читабельным.

Осталось рассмотреть еще один метод объекта `HTMLDocument`, который может быть нам полезен. Это метод `close`; он заставляет Web-обозреватель немедленно обработать и вывести на страницу весь HTML-код, вставленный предыдущими вызовами методов `write` и `writeln`. Применяется он редко, но нам следует о нем знать.

Методы `write` и `writeln` практически всегда используются для дополнения содержимого страницы во время ее загрузки и применяются в загрузочных сценариях, что понятно. Если же требуется исправить содержимое страницы в ответ на событие, лучше использовать другие способы работы со страницей, которые мы сейчас рассмотрим.

Использование методов DOM

Использование для правки содержимого страницы методов DOM — самый универсальный способ, позволяющий сделать все что угодно и работающий во всех Web-обозревателях. Методы DOM жестко стандартизированы и вполне логичны, вот только написанный с их использованием JavaScript-код выглядит зачастую слишком громоздким.

Методы DOM позволяют выполнить следующее:

- найти нужный элемент страницы;
- создать новый элемент страницы, представляющий собой тег или текстовый элемент;
- поместить созданный элемент страницы в другой, фактически выведя его на страницу;
- удалить ненужный элемент страницы.

Методы, выполняющие поиск нужного элемента страницы, нам уже знакомы. Мы можем обратиться к *главе 5*, где они были описаны, чтобы все о них вспомнить.

Создание нового элемента страницы — тега или текстового элемента — выполняют два метода объекта `HTMLDocument`. Сейчас мы их рассмотрим.

Метод `createElement` выполняет создание нового тега.

```
createElement(<имя тега>)
```

`Имя тега` передается в строковом виде и без символов < и >. Метод возвращает экземпляр объекта — потомка объекта `HTMLElement`, соответствующий созданному элементу страницы.

```
var pObj = document.createElement("P");
```

Приведенное выражение создаст и поместит в переменную `pObj` экземпляр объекта `HTMLParagraphElement`, соответствующий абзацу.

Отметим сразу, что метод `createElement` в случае парного тега создает пустой тег. Кроме того, атрибутам созданного тега присваиваются значения по умолчанию. Наполнить созданный тег содержимым и задать нужные значения его атрибутов нам придется самим. Как это сделать, мы выясним потом.

Метод `createTextNode` выполняет создание текстового элемента и занесение в него содержимого.

```
createTextNode(<содержимое>)
```

Он возвращает экземпляр объекта `Text`, соответствующий созданному текстовому элементу.

```
var textObj = document.createTextNode("Это абзац.");
```

> ***НА ЗАМЕТКУ***
>
> Для занесения содержимого в текстовый элемент мы можем применить свойство `nodeValue`, рассмотренное в *главе 5*. Достаточно присвоить строку с текстовым содержимым этому свойству.

А теперь — еще одна очень важная вещь. Оба метода — и `createElement`, и `createTextNode` — не выполняют размещение созданного элемента страницы на самой этой странице. Они только создают экземпляр соответствующего объекта в памяти компьютера. Чтобы вывести созданный с помощью описанных ранее методов элемент на страницу, нам придется совершить дополнительные действия. Какие — мы сейчас рассмотрим.

Начнем с того, что объект `HTMLElement` (а значит, и все порожденные от него объекты) поддерживает метод `appendChild`. Он помещает элемент страницы, переданный ему в качестве параметра, в текущий элемент, делая первый потомком последнего.

```
appendChild(<элемент страницы — будущий потомок>)
```

При этом элемент страницы, переданный в качестве параметра, помещается в самый конец текущего элемента и становится самым последним его потомком.

```
pObj.appendChild(textObj);
```

Это выражение поместит созданный ранее текстовый элемент `textObj` в абзац `pObj`, который также был создан ранее. Таким образом, абзац `pObj` получит текстовое содержимое.

Этот же метод используется для вывода созданного элемента страницы на саму страницу. Достаточно поместить его внутрь любого элемента страницы, который уже присутствует на странице, например, в секцию тела страницы.

```
document.body.appendChild(pObj);
```

Это выражение поместит созданный ранее абзац `pObj` (он уже имеет текстовое содержимое) в секцию тела страницы. При этом данный абзац появится на странице и будет видим на экране.

Еще один весьма полезный метод, поддерживаемый объектом `HTMLElement`, — `insertBefore`. Он позволяет поместить новый элемент страницы в произвольное место внутри текущего.

```
insertBefore(<элемент страницы — будущий потомок>[,
↳<уже существующий потомок, перед которым будет помещен данный элемент>])
```

При вызове этого метода элемент страницы, переданный первым параметром, будет помещен в текущий элемент перед тем его потомком, который передан вторым параметром. Если второй параметр опущен, метод `insertBefore` работает так же, как уже знакомый нам метод `appendChild`.

Понятно, что метод `insertBefore` также помещает созданный элемент на страницу, если вызван для уже присутствующего на странице элемента.

```
var p2Obj = document.createElement("P");
var text2Obj = document.createTextNode("Это первый абзац.");
p2Obj.appendChild(text2Obj);
document.body.insertBefore(p2Obj, document.body.childNodes[0]);
```

Этот сценарий поместит на страницу еще один абзац, расположив его перед помещенным ранее. Обратим внимание, как мы указываем элемент страницы, перед которым нужно поместить созданный элемент, — через коллекцию `childNodes`. Так проще и надежнее.

Если нам нужно заменить один элемент страницы другим, мы используем метод `replaceChild`, также поддерживаемый объектом `HTMLElement`.

```
replaceChild(<заменяющий элемент страницы>, <заменяемый потомок>);
```

Этот метод возвращает экземпляр соответствующего объекта, представляющий заменяемый потомок. Впоследствии мы можем использовать его, чтобы, например, поместить в другой элемент страницы.

```
var h1Obj = document.createElement("H1");
var text3Obj = document.createTextNode("Это заголовок");
h1Obj.appendChild(text3Obj);
var p2Obj = document.body.replaceChild(h1Obj,
↳document.body.childNodes[0]);
document.body.appendChild(p2Obj);
```

Этот сценарий заменяет помещенный ранее на страницу абзац с текстом "Это первый абзац." заголовком первого уровня. Сам же замененный абзац помещается в конец страницы.

Следующий полезный метод объекта HTMLElement, который мы рассмотрим, — это cloneNode. Он возвращает полную копию текущего элемента страницы.

```
cloneNode(<включать потомки и атрибуты>)
```

Единственный параметр этого метода должен быть логической величиной. Если он равен true, в копию элемента страницы будут включены все его потомки и атрибуты, если они были созданы. Значение false предписывает выполнить копирование только самого этого элемента страницы. Метод возвращает экземпляр соответствующего объекта, представляющего копию текущего элемента страницы.

```
var p4Obj = document.body.childNodes[1].cloneNode(true);
p4Obj.nodeValue = "Это последний абзац.";
document.body.appendChild(p4Obj);
```

Этот сценарий создает копию второго по счету элемента страницы — абзаца с текстом "Это абзац.", — меняет его текст на "Это последний абзац." и помещает его в самый конец страницы.

Метод cloneNode пригодится, если нам понадобится получить множество слегка различающихся элементов страницы, например, ячеек таблицы. Достаточно будет создать одну ячейку, выставить ее параметры, после чего останется скопировать ее нужное число раз и для каждой копии задать свое содержимое. Это будет быстрее, чем создавать каждую ячейку заново.

Последний рассмотренный нами метод объекта HTMLElement подводит жирную черту под существованием элемента страницы. Это метод removeChild, удаляющий заданный потомок текущего элемента страницы.

```
removeChild(<удаляемый потомок>);
```

Единственным параметром этому методу передается потомок текущего элемента страницы, который нужно удалить. Метод возвращает экземпляр соответствующего объекта, представляющий удаляемый потомок. Впоследствии мы можем использовать его, чтобы, например, поместить в другой элемент страницы.

```
document.body.removeChild(document.body.childNodes[2]);
```

Это выражение удалит третий по счету элемент страницы — абзац с текстом "Это первый абзац.". Нечего ему там делать.

Internet Explorer и Opera поддерживают также метод insertAdjacentElement объекта HTMLElement, аналогичный рассмотренному выше методу insertBefore, но более гибкий.

```
insertAdjacentElement(<местоположение будущего потомка>,
↳<элемент страницы — будущий потомок>);
```

Первый параметр этого метода задает местоположение нового потомка в виде одной из перечисленных ниже строк:

- "afterBegin" — в самом начале текущего элемента страницы, после его открывающего тега;
- "afterEnd" — после текущего элемента страницы (его закрывающего тега);
- "beforeBegin" — перед текущим элементом страницы (его открывающим тегом);
- "beforeEnd" — в самом конце текущего элемента страницы, перед его закрывающим тегом.

Второй параметр задает сам элемент страницы, который станет потомком текущего элемента.

```
var p2Obj = document.createElement("P");
var text2Obj = document.createTextNode("Это первый абзац.");
p2Obj.appendChild(text2Obj);
document.body.insertAdjacentElement("afterBegin", p2Obj);
```

Здесь мы переписали пример, приведенный для метода insertBefore, с использованием только что изученного метода insertAdjacentElement. Но не забываем, что работать он будет только в Internet Explorer и Opera.

В качестве примера давайте возьмем самую первую написанную нами Web-страницу (см. *главу 2*). Вот ее HTML-код:

```
<HTML>
  <HEAD>
    <TITLE>Web-страница</TITLE>
  </HEAD>
```

```
  <BODY>
    <P>Это простейшая Web-страничка, созданная в стандартном
    <EM>Блокноте</EM> и отображенная в <EM>Microsoft
    Internet Explorer</EM>.</P>
  </BODY>
</HTML>
```

Удалим все ее содержимое и поместим в нее сценарий, создающий то же самое, но программно. HTML-код новой страницы таков:

```
<HTML>
  <HEAD>
    <TITLE>Страница, создаваемая программно</TITLE>
  </HEAD>
  <BODY>
    <SCRIPT>
      var pObj = document.createElement("P");
      var text1Obj = document.createTextNode("Это простейшая
      ↳Web-страничка, созданная в стандартном ");
      pObj.appendChild(text1Obj);
      var em1Obj = document.createElement("EM");
      var text2Obj = document.createTextNode("Блокноте");
      em1Obj.appendChild(text2Obj);
      pObj.appendChild(em1Obj);
      var text3Obj = document.createTextNode(" и отображенная в ");
      pObj.appendChild(text3Obj);
      var em2Obj = document.createElement("EM");
      var text4Obj = document.createTextNode("Microsoft Internet
      ↳Explorer");
      em2Obj.appendChild(text4Obj);
      pObj.appendChild(em2Obj);
      var text5Obj = document.createTextNode(".");
      pObj.appendChild(text5Obj);
      document.body.appendChild(pObj);
    </SCRIPT>
  </BODY>
</HTML>
```

Как видим, код получился громоздкий и сложный. Поэтому разбор его займет много времени.

Прежде всего, давайте посмотрим на HTML-код изначальной страницы и определим структуру тегов, наподобие той, что мы создавали в *главе 5* при рассмотрении DOM. Структура эта будет такова:

```
Тег <BODY>
  Тег <P>
    Текст "Это простейшая Web-страничка, созданная в стандартном "
    Тег <EM>
      Текст "Блокноте"
    Текст " и отображенная в "
    Тег <EM>
      Текст "Microsoft Internet Explorer"
    Текст "."
```

Здесь отступ слева обозначает вложенность одних элементов страницы в другие: элемент страницы, имеющий больший отступ, вложен в предыдущий элемент страницы, имеющий меньший отступ. Видно, что нам придется создать 8 элементов страницы; из них 3 тега и 5 текстовых элементов. Тег `<BODY>` уже присутствует в странице, и нам создавать его не требуется, поэтому его не считаем.

Теперь давайте рассмотрим код приведенного ранее сценария построчно.

```
var pObj = document.createElement("P");
```

Сначала создаем тег `<P>`, представляющий единственный абзац нашей новой страницы, и помещаем его в переменную `pObj`.

```
var text1Obj = document.createTextNode("Это простейшая Web-страничка,
созданная в стандартном ");
```

Далее очередь за первым потомком только что созданного абзаца — текстовым элементом, содержащим текст "Это простейшая Web-страничка, созданная в стандартном " (не забываем конечный пробел). Созданный текстовый элемент помещаем в переменную `text1Obj`.

```
pObj.appendChild(text1Obj);
```

Помещаем текстовый элемент `text1Obj` в абзац `pObj`. Процесс создания содержимого единственного абзаца нашей страницы пошел.

```
var em1Obj = document.createElement("EM");
```

Теперь создаем первый из присутствующих на странице тегов `<EM>` и помещаем его в переменную `em1Obj`.

```
var text2Obj = document.createTextNode("Блокноте");
```

Самое время создать текстовое содержимое первого тега `<EM>` — слово "Блокноте". Поместим его в переменную `text2Obj`.

```
em1Obj.appendChild(text2Obj);
```

Помещаем только что созданный текстовый элемент `text2Obj` в тег `<EM>` `em1Obj`.

```
pObj.appendChild(em1Obj);
```

И помещаем готовый тег `<EM>` `em1Obj`, уже включающий содержимое, в самый конец абзаца `pObj`, после вставленного ранее текстового элемента `text1Obj`.

```
var text3Obj = document.createTextNode(" и отображенная в ");
pObj.appendChild(text3Obj);
```

На создании следующего текстового элемента, который включит строку " и отображенная в " (начальный и конечный пробелы обязательны) и добавится в абзац `pObj`, мы останавливаться не будем. Все здесь то же самое, что и в случае первого текстового элемента `text1Obj`.

```
var em2Obj = document.createElement("EM");
var text4Obj = document.createTextNode("Microsoft Internet Explorer");
em2Obj.appendChild(text4Obj);
pObj.appendChild(em2Obj);
```

Создаем второй тег `<EM>`, помещаем в него текстовое содержимое со строкой "Microsoft Internet Explorer", после чего помещаем готовый тег с текстовым содержимым в абзац `pObj`.

```
var text5Obj = document.createTextNode(".");
pObj.appendChild(text5Obj);
```

Осталось создать последний элемент страницы — текстовый элемент с последней точкой предложения и поместить ее в абзац `pObj`. Все, абзац со всем его содержимым готов.

```
document.body.appendChild(pObj);
```

Осталось только поместить полностью созданный абзац `pObj` в секцию тела страницы. После этого Web-обозреватель выведет его на экран.

Приведем еще один пример страницы, чье содержимое формируется из сценария. Пусть она содержит таблицу из двух строк и двух столбцов.

```
<HTML>
  <HEAD>
    <TITLE>Страница, создаваемая программно</TITLE>
  </HEAD>
  <BODY>
    <SCRIPT>
      var tabObj = document.createElement("TABLE");
      var bodyObj = document.createElement("TBODY");
      var rowObj = document.createElement("TR");
      var cellObj = document.createElement("TD");
```

```
      var textObj = document.createTextNode("1");
      cellObj.appendChild(textObj);
      rowObj.appendChild(cellObj);
      var cellObj = document.createElement("TD");
      var textObj = document.createTextNode("2");
      cellObj.appendChild(textObj);
      rowObj.appendChild(cellObj);
      bodyObj.appendChild(rowObj);
      var rowObj = document.createElement("TR");
      var cellObj = document.createElement("TD");
      var textObj = document.createTextNode("3");
      cellObj.appendChild(textObj);
      rowObj.appendChild(cellObj);
      var cellObj = document.createElement("TD");
      var textObj = document.createTextNode("4");
      cellObj.appendChild(textObj);
      rowObj.appendChild(cellObj);
      bodyObj.appendChild(rowObj);
      tabObj.appendChild(bodyObj);
      document.body.appendChild(tabObj);
    </SCRIPT>
  </BODY>
</HTML>
```

Попробуйте сами разобраться, как работает этот сценарий. Автор только отметит, что здесь мы явно создаем секцию тела таблицы и уже к ней добавляем строки. По результатам его собственных экспериментов, в случае программного создания таблицы так будет лучше.

Методы DOM, предназначенные для работы с содержимым страницы, — исключительно гибкий и поддерживаемый всеми Web-обозревателями инструмент. У него есть только один недостаток — сложность и громоздкость получающегося JavaScript-кода, что мы и наблюдали в последнем примере. Но — увы! — на свете нет ничего совершенного…

Использование специфических свойств и методов Internet Explorer

Последний способ изменения содержимого страницы после ее загрузки в Web-обозревателе, который мы рассмотрим, — это использование особых свойств и методов объекта `HTMLElement`, которые поддерживаются только Internet Explorer и Opera. С их помощью мы можем писать весьма компактные

сценарии, правящие содержимое страницы "на лету", но, к сожалению, не работающие в Firefox. Давайте их рассмотрим.

Самое для нас полезное из свойств — конечно, `innerHTML`. Оно задает или возвращает HTML-код, определяющий содержимое текущего элемента страницы, в виде строки.

```
var parCode = document.getElementById("par").innerHTML;
```

Это выражение поместит в переменную `parCode` строку с HTML-кодом, представляющим содержимое абзаца `par`.

```
document.body.innerHTML = "<P>Это простейшая Web-страничка.</P>";
```

А это выражение поместит в секцию тела страницы абзац с текстом, заменив им предыдущее содержимое страницы.

> ***НА ЗАМЕТКУ***
>
> Надо отметить, что Firefox также поддерживает свойство `innerHTML` объекта `HTMLElement`, но оно доступно только для чтения. Это значит, что мы можем получить HTML-код, определяющий содержимое элемента страницы, но не можем его изменить. Достаточно бесполезная возможность…

Свойство `innerText` менее полезно. Оно задает или возвращает текстовое содержимое текущего элемента страницы также в виде строки.

```
document.getElementById("par").innerText = "Это абзац.";
```

Это выражение поместит в абзац `par` новый текст.

У свойства `innerText` есть две особенности, о которых нам обязательно нужно знать.

- При обращении к этому свойству оно вернет строку, представляющую все текстовое содержимое элемента страницы и его потомков. Никакие теги HTML в возвращаемой строке присутствовать не будут.
- При присвоении этому свойству строки, содержащей теги HTML, они будут выведены на страницу в виде текста без всякой обработки Web-обозревателем.

Поясним вторую особенность этого свойства. Пусть мы написали вот такое выражение:

```
document.getElementById("par").innerText = "<EM>Это абзац.</EM>";
```

Web-обозреватель выведет на страницу следующий текст:

```
<EM>Это абзац.</EM>
```

То есть теги `<EM>` и `</EM>` обработаны не будут, а выведутся на страницу "как есть".

Свойство `outerHTML` задает или возвращает HTML-код, определяющий и текущий элемент страницы, и его содержимое, в виде строки. Этим оно отличается от рассмотренного ранее свойства `innerHTML`.

```
document.getElementById("par").outerHTML = "<H1>Это заголовок</H1>";
```

Это выражение полностью заменит HTML-код, определяющий абзац `par`, кодом, создающим заголовок первого уровня. В результате на экране абзац пропадет, а вместо него появится заголовок.

Самое бесполезное свойство — `outerText` — задает или возвращает текстовое содержимое текущего элемента страницы в виде строки. От свойства `innerText` оно отличается тем, что при присвоении ему новой строки сам текущий элемент страницы будет удален, а на его месте появится содержимое строки, присвоенной свойству.

```
document.getElementById("par").outerText = "Это абзац.";
```

Это выражение удалит абзац `par` вместе с его содержимым и вставит на его место текст "Это абзац.". Данный текст станет содержимым родителя абзаца `par`: контейнера, ячейки таблицы или даже секции тела страницы.

> ***Внимание!***
>
> Текст, помещенный прямо в секцию тела страницы, вне абзацев, заголовков, ячеек таблицы и пр., — плохой стиль Web-дизайна.

Напоследок — небольшая памятка, которая поможет нам лучше запомнить, какое свойство за что отвечает.

- Свойства `innerHTML` и `outerHTML` работают с HTML-кодом, а свойства `innerText` и `outerText` — только с текстовым содержимым.
- Свойства `innerHTML` и `innerText` затрагивают только содержимое текущего элемента страницы, а свойства `outerHTML` и `outerText` — и сам текущий элемент.

Перейдем к методам. Всего их три.

Метод `insertAdjacentHTML` для нас будет самым полезным. Он вставляет заданный HTML-код в заданное место текущего элемента страницы.

```
insertAdjacentHTML(<местоположение вставляемого HTML-кода>,
↳<вставляемый HTML-код>);
```

Этот метод принимает два параметра. Первый из них задает местоположение вставляемого HTML-кода в виде одной из перечисленных далее строк:

- "afterBegin" — в самом начале текущего элемента страницы, после его открывающего тега;
- "afterEnd" — после текущего элемента страницы (его закрывающего тега);

- "beforeBegin" — перед текущим элементом страницы (его открывающим тегом);
- "beforeEnd" — в самом конце текущего элемента страницы, перед его закрывающим тегом.

Второй параметр задает вставляемый HTML-код также в виде строки.

```
var parObj = document.getElementById("par");
parObj.insertAdjacentHTML("afterBegin", "<EM>Привет!</EM> ");
parObj.insertAdjacentHTML("afterEnd", "<P>Пока!</P>");
```

Этот небольшой сценарий сначала вставит в начало абзаца `par` строку "Привет! " (присутствует конечный пробел), выделенную курсивом, а потом поместит после этого абзаца другой абзац с текстом "Пока!".

Метод `insertAdjacentText`, в отличие от предыдущего, занимается исключительно текстом. Формат его вызова такой же.

```
insertAdjacentText(<местоположение вставляемого текста>,
↳<вставляемый текст>);
```

Как и в случае свойств `innerText` и `outerText`, присутствующие во вставляемом тексте теги HTML будут выведены на страницу "как есть", без всякой обработки Web-обозревателем.

```
var parObj = document.getElementById("par");
parObj.insertAdjacentText("afterBegin", "Привет! ");
```

Этот сценарий вставит в начало абзаца `par` строку "Привет! " (присутствует конечный пробел).

Метод `replaceAdjacentText` позволяет заменить текст, являющийся содержимым текущего элемента страницы, другим.

```
replaceAdjacentText(<местоположение заменяемого текста>,
↳<заменяющий текст>);
```

Первый параметр этого метода задает местоположение заменяемого текста в виде одной из перечисленных далее строк:

- "afterBegin" — в самом начале текущего элемента страницы, между его открывающим тегом и открывающим тегом первого элемента-потомка;
- "afterEnd" — между закрывающим тегом текущего элемента страницы и открывающим тегом следующего элемента;
- "beforeBegin" — между закрывающим тегом предыдущего элемента страницы и открывающим тегом текущего элемента;
- "beforeEnd" — в самом конце текущего элемента страницы, между закрывающим тегом последнего элемента-потомка и закрывающим тегом самого текущего элемента.

Второй параметр задает заменяющий текст в виде строки. Метод возвращает строку, содержащую замененный текст.

```
var em1Obj = document.getElementById("em1");
em1Obj.replaceAdjacentText("beforeBegin", "Привет!");
```

Данный сценарий заменит весь текст, присутствующий перед открывающим тегом элемента страницы `em1` (это тег `<EM>` с содержимым), строкой "Привет!".

Метод `getAdjacentText` возвращает строку, содержащую текст из заданного места внутри текущего элемента страницы или рядом с ним.

```
getAdjacentText(<местоположение искомого текста>);
```

Единственный параметр этого метода указывает местоположение искомого текста в виде одной из строк, перечисленных ранее — см. описание метода `replaceAdjacentText`.

```
var em1Obj = document.getElementById("em1");
var str = em1Obj.getAdjacentText("afterEnd");
```

Этот сценарий поместит в переменную `str` текст, находящийся после элемента страницы `em1` (это тег `<EM>` с содержимым).

> ***Внимание!***
>
> Не забываем, что все описанные ранее свойства и методы поддерживаются только Internet Explorer и Opera.

На этом рассмотрение работы с содержимым страницы можно закончить. Пора приступать к работе с атрибутами тегов.

Работа с атрибутами тегов

Следующий шаг — работа с атрибутами элементов страницы, представляющих теги. (У текстовых элементов нет атрибутов.) Не умея с ними работать, мы не сможем изменить параметры элементов страницы.

Разработчики Web-обозревателей и здесь постарались и предоставили нам целых три способа работы с атрибутами тегов: создания их, задания для них значений и удаления. Похоже, три — счастливое число в Web-программировании.

Прямой доступ к атрибутам через свойства

Самый простой и используемый практически всегда способ добраться до значения нужного атрибута — это использовать соответствующее ему свойство объекта, представляющего данный элемент страницы. Уж что может быть проще, чем обратиться к свойству объекта!

Как мы уже знаем, базовую функциональность элемента страницы предоставляет объект HTMLElement. От этого объекта порождаются другие объекты, представляющие конкретные элементы страницы: абзацы, заголовки, гиперссылки, графические изображения, таблицы и их элементы и даже саму секцию тела страницы. Так, объект HTMLParagraphElement представляет абзац, объект HTMLHeadingElement — заголовок, объект HTMLImageElement — изображение и др.

Большинство из объектов, порожденных от объекта HTMLElement, не добавляют к нему никаких своих свойств, методов и событий. Это справедливо, в частности, для объектов HTMLParagraphElement и HTMLHeadingElement из упомянутых ранее — все их свойства, методы и события унаследованы от HTMLElement. И это понятно — как мы выяснили еще в *главе 2*, никаких специфических атрибутов теги <P> и <H*n*> не поддерживают.

Но некоторые из порожденных от объектов HTMLElement содержат, в частности, новые свойства, соответствующие специфическим для тега, создающего данный элемент страницы, атрибутам. Так, объект HTMLLinkElement, представляющий гиперссылку, добавляет свойства href и target, соответствующий атрибутам HREF и TARGET тега <A>. (Подробнее об этом объекте и его специфических свойствах и событиях мы поговорим ближе к концу главы.)

Исходя из этого, мы можем создать на странице гиперссылку и программно поменять интернет-адрес, на который она указывает:

```
<A ID="href1" HREF="page2.html">Страница 2</A>
. . .
<SCRIPT>
  var href1Obj = document.getElementById("href1");
  href1Obj.href = "page3.html";
  href1Obj.firstChild.nodeValue = "Страница 3";
</SCRIPT>
```

При желании мы также можем задать для нее новый текст, как и было продемонстрировано в примере ранее.

```
var pObj = document.createElement("P");
var textObj = document.createTextNode("Это абзац.");
pObj.appendChild(textObj);
pObj.id = "par";
document.body.appendChild(pObj);
```

А этот сценарий создает новый абзац, задает для него текстовое содержимое и имя par. Для задания имени здесь используется описанное еще в *главе 5* свойство id объекта HTMLElement, соответствующее атрибуту ID, который поддерживается всеми тегами.

Как правило, имена атрибутов и соответствующих им свойств совпадают, при этом имена свойств набираются маленькими буквами. А имена атрибутов мы уже знаем — они были описаны еще в *главе 2*.

Использование коллекции *attributes*

Второй способ получить доступ к нужному атрибуту — использовать коллекцию `attributes`. Эта коллекция является экземпляром объекта `NamedNodeMap` и содержит набор экземпляров объекта `Attr`, каждый из которых представляет один из заданных для данного тега атрибутов. Получить доступ к коллекции `attributes` можно через одноименное свойство объекта `HTMLElement`.

Что касается объекта `Attr`, представляющего отдельный атрибут, то он поддерживает два полезных для нас свойства.

- Свойство `name` возвращает имя атрибута.
- Свойство `value` задает или возвращает значение атрибута.

Здесь все просто: получаем доступ к нужному атрибуту и задаем его значение через свойство `value`.

К сожалению, для доступа к нужным элементам коллекции `attributes` мы не можем использовать строковые индексы — имена атрибутов. Доступно использование только числовых индексов, что — согласитесь — крайне неудобно. Чтобы получить атрибут по его имени, нам придется использовать метод `getNamedItem` объекта `NamedNodeMap`.

`getNamedItem(`*<имя нужного атрибута>*`)`

`Имя нужного атрибута` передается этому методу в виде строки. Возвращает он экземпляр объекта `Attr`, представляющий атрибут с заданным именем.

Давайте перепишем приведенный ранее пример сценария, меняющий интернет-адрес гиперссылки, с использованием коллекции `attributes`.

```
<A ID="href1" HREF="page2.html">Страница 2</A>
. . .
<SCRIPT>
  var href1Obj = document.getElementById("href1");
  var hrefAttrObj = href1Obj.attributes.getNamedItem("href");
  hrefAttrObj.value = "page3.html";
  href1Obj.firstChild.nodeValue = "Страница 3";
</SCRIPT>
```

К сожалению, таким способом мы можем изменить значение только того атрибута, который уже был задан в HTML-коде или в сценарии. Если нам

понадобится задать значение атрибута, который еще не был задан, нам придется его создать. Сейчас мы выясним, как это выполняется.

Прежде всего нам нужно создать новый атрибут. Этим занимается метод `createAttribute` объекта `HTMLDocument`.

```
createAttribute(<имя создаваемого атрибута>)
```

Как видим, `имя создаваемого атрибута` передается данному методу в виде строки. А возвращает этот метод экземпляр объекта `Attr`, представляющий созданный атрибут.

```
var nameAttrObj = document.createAttribute("name");
nameAttrObj.value = "anchor1";
```

Это выражение создаст экземпляр объекта `Attr`, представляющий атрибут `NAME`, и задает для него значение "anchor1".

Так, атрибут мы создали. Но он существует только в памяти компьютера и никак не привязан к нужному нам тегу. Нам нужно привязать его к тегу с помощью метода `setNamedItem` объекта `NamedNodeMap` (экземпляром этого объекта, как мы помним, является коллекция `attributes`).

```
setNamedItem(<привязываемый атрибут>)
```

Экземпляр объекта `Attr`, соответствующий созданному нами атрибуту, мы передаем в этот метод единственным параметром.

Если к данному тегу уже был привязан атрибут с таким же именем, метод `setNamedItem` заменит его новым и вернет экземпляр объекта `Attr`, представляющий старый атрибут. Если же атрибут с таким именем отсутствует в данном теге, возвращается `null`.

```
href1Obj.attributes.setNamedItem(nameAttrObj);
```

Это выражение добавит к гиперссылке `href1` атрибут `NAME` со значением "anchor1".

Еще один метод объекта `NamedNodeMap`, который мы должны знать, — это `removeNamedItem`. Он удаляет из текущего тега атрибут с заданным именем.

```
removeNamedItem(<имя удаляемого атрибута>)
```

`Имя удаляемого атрибута` передается в качестве параметра в строковом виде. Данный метод вернет экземпляр объекта `Attr`, представляющий удаленный атрибут; впоследствии мы можем привязать его к другому тегу, если у нас возникнет в этом нужда.

Осталось привести "большой" пример использования изученных нами методов. Давайте напишем сценарий, который удалит у гиперссылки `href1` атрибут `HREF` и создаст атрибут `NAME`, тем самым превратив гиперссылку в якорь.

```
<A ID="href1" HREF="page2.html">Страница 2</A>
. . .
```

```
<SCRIPT>
  var href1Obj = document.getElementById("href1");
  href1Obj.attributes.removeNamedItem("href");
  var nameAttrObj = document.createAttribute("name");
  nameAttrObj.value = "anchor1";
  href1Obj.attributes.setNamedItem(nameAttrObj);
</SCRIPT>
```

Способ работы с атрибутами тегов с использованием коллекции `attributes` хоть и соответствует полностью стандартам DOM, но на практике почти не применяется. Особых преимуществ Web-программистам он не дает, а сценарии, написанные с его использованием, получаются чересчур громоздкими. (Это, кстати, относится и к способу с использованием методов DOM, который мы рассмотрим далее.) В основном, применяется способ работы с атрибутами через соответствующие им свойства (описан ранее).

Использование методов DOM

Третий способ работы с атрибутами тегов — это использование соответствующих методов DOM. Эти методы позволяют получить экземпляр объекта `Attr`, соответствующий нужному атрибуту, по его имени, привязать созданный атрибут к тегу и удалить его без использования коллекции `attributes`. Все эти методы поддерживаются объектом `HTMLElement`, представляющим элемент страницы — тег.

Первый метод, который мы рассмотрим, — это `getAttributeNode`. Он возвращает экземпляр объекта `Attr`, соответствующий атрибуту с заданным именем.

`getAttributeNode(`*<имя нужного атрибута>*`)`

`Имя нужного атрибута` передается в виде строки.

```
<A ID="href1" HREF="page2.html">Страница 2</A>
. . .
<SCRIPT>
  var href1Obj = document.getElementById("href1");
  var hrefAttrObj = href1Obj.getAttributeNode("href");
  hrefAttrObj.value = "page3.html";
</SCRIPT>
```

Этот сценарий меняет интернет-адрес нашей многострадальной гиперссылки `href1`.

Следующий метод — `setAttributeNode`. Он привязывает переданный в качестве параметра атрибут к текущему тегу.

`setAttributeNode(`*<привязываемый атрибут>*`)`

В качестве параметра передается экземпляр объекта `Attr`, соответствующий созданному ранее атрибуту. Для его создания мы используем рассмотренный ранее метод `createAttribute` объекта `HTMLDocument`.

Если к данному тегу уже был привязан атрибут с таким же именем, метод `setAttributeNode` заменит его новым и вернет экземпляр объекта `Attr`, представляющий старый атрибут. Если же атрибут с таким именем отсутствует в данном теге, возвращается `null`. В этом метод `setAttributeNode` аналогичен рассмотренному ранее методу `setNamedItem` объекта `NamedNodeMap`.

```
var nameAttrObj = document.createAttribute("name");
nameAttrObj.value = "anchor1";
href1Obj.setAttributeNode(nameAttrObj);
```

Этот сценарий поместит в гиперссылку `href1` атрибут `NAME` со значением "anchor1".

Метод `removeAttribute` удаляет атрибут с заданным именем.

```
removeAttribute(<имя удаляемого атрибута>)
```

`Имя удаляемого атрибута` передается в качестве параметра в строковом виде.

И — пример.

```
<A ID="href1" HREF="page2.html">Страница 2</A>
. . .
<SCRIPT>
  var href1Obj = document.getElementById("href1");
  href1Obj.removeAttribute("href");
  var nameAttrObj = document.createAttribute("name");
  nameAttrObj.value = "anchor1";
  href1Obj.setAttributeNode(nameAttrObj);
</SCRIPT>
```

Нужно рассказывать, что он делает?

Стандарты DOM предусматривают еще два метода объекта `HTMLElement`, позволяющие быстро задать и изменить значение заданного атрибута. Пожалуй, их и стоит использовать в большинстве случаев.

Метод `getAttribute` позволяет быстро получить значение атрибута.

```
getAttribute(<имя нужного атрибута>)
```

`Имя нужного атрибута` передается в виде строки.

Метод `getAttribute` возвращает значение данного атрибута в том формате, в котором оно было задано: строка, число, логическая величина (в случае атрибутов без значения, описанных в *главе 2*; в случае присутствия данного

атрибута возвращается `true`). Если таковой атрибут отсутствует в теге, возвращается `null`.

```
var hrefStr = href1Obj.getAttribute("href");
```

Это выражение поместит в переменную `hrefStr` строку, содержащую интернет-адрес гиперссылки `href1`.

Метод `setAttribute` задает атрибуту значение. Если таковой атрибут отсутствует в теге, он будет создан.

`getAttribute(`*<имя нужного атрибута>*`, `*<его значение>*`)`

`Имя нужного атрибута` передается в виде строки, а его `значение` — в том формате, в котором оно должно быть задано: строка, число или логическая величина (в случае атрибутов без значения).

```
href1Obj.setAttribute("name", "anchor1");
```

Это выражение задаст для атрибута `NAME` гиперссылки `href1` значение "anchor1". Поскольку такового атрибута в теге `<A>`, создающем эту гиперссылку, нет, он будет создан.

О способе работы с атрибутами тегов с помощью методов DOM можно сказать то же самое, что и о способе с использованием коллекции `attributes`. Применяется он редко из-за громоздкости получающихся сценариев. В основном, используется способ прямого доступа к атрибутам через соответствующие свойства, который мы рассмотрели ранее.

Работа со стилями

Покончив с атрибутами тегов, займемся стилями CSS, привязанными к этим самым тегам. Способ работы со стилями мы рассмотрим только один, поскольку именно он используется всегда.

Для работы со стилями используется объект `CSSRule`. Экземпляр этого объекта, представляющий текущий, привязанный к элементу страницы стиль, создается самим Web-обозревателем и доступен через свойство `style` объекта `HTMLElement`. Если к данному элементу страницы не был привязан никакой стиль, Web-обозреватель создает своего рода стиль по умолчанию, хранящий некоторые изначальные параметры.

Но как добраться до атрибутов стиля? Здесь та же ситуация, что и с атрибутами тегов, — каждому атрибуту стиля соответствует особое свойство объекта `CSSRule`. Эти свойства набираются строчными буквами, без знаков -, а каждая буква, следующая в имени атрибута стиля за знаком -, делается прописной. Посмотрим на табл. 8.1 — там представлены имена некоторых свойств объекта `CSSRule` и имена соответствующих им атрибутов стиля.

Таблица 8.1. *Некоторые свойства объекта* `CSSRule` *и соответствующие им атрибуты стиля*

Атрибут стиля	Свойство объекта `CSSRule`
`background-color`	`backgroundColor`
`border-bottom-style`	`borderBottomStyle`
`color`	`color`
`text-align`	`textAlign`

Этим свойствам присваиваются значения в том же формате, что и соответствующим атрибутам стиля, и всегда в строковом виде. (Подробнее об атрибутах стиля и присваиваемых им значениях было рассказано в *главе 2*.) Так, свойству `backgroundColor` присваивается строка, содержащая RGB-код цвета (например, "#FF0000"), свойству `text-align` — строка, содержащая параметр выравнивания ("center") и т. п.

```
var hdObj = document.getElementById("hd");
hdObj.style.textAlign = "center";
```

Этот сценарий задает для заголовка `hd` выравнивание по центру.

```
var parObj = document.getElementById("par");
parObj.style.fontSize = "12pt";
```

А этот сценарий задает для абзаца `par` размер шрифта, равный 12 пунктам.

Также мы имеем возможность привязать к элементу страницы новый стилевой класс. Для этого служит свойство `className` объекта `HTMLElement` — оно как раз задает или возвращает имя стилевого класса в строковом виде.

```
var hdObj = document.getElementById("hd");
hdObj.className = "centeredHeader";
```

Этот сценарий привязывает к заголовку `hd` определенный в одной из таблиц стилей, внешних или внутренней, стилевой класс `centeredHeader`.

Для привязки к элементу страницы нового стиля-селектора мы можем использовать свойство `id` объекта `HTMLElement`. (Как мы помним из *главы 5*, это свойство также задает имя элемента страницы.)

```
var parObj = document.getElementById("par");
parObj.id = "someParagraph";
```

Данный сценарий привязывает к абзацу `par` определенный в одной из таблиц стилей, внешних или внутренней, стиль-селектор `someParagraph` и одновременно задает для абзаца новое имя, совпадающее с именем стиля-селектора.

Теперь настала пора рассмотреть примеры использования только что полученных знаний. Первым из них будет страница, содержащая сценарий, который

создаст у единственного присутствующего на этой странице заголовка толстую серую рамку. Ее HTML-код приведен далее.

```
<HTML>
  <HEAD>
    <TITLE>Работа со стилями</TITLE>
  </HEAD>
  <BODY>
    <H1 ID="hd">Привет!</H1>
    <SCRIPT>
      var hdObj = document.getElementById("hd");
      with (hdObj.style) {
        borderLeftColor = "#CCCCCC";
        borderLeftStyle = "solid";
        borderLeftWidth = "2px";
        borderTopColor = "#CCCCCC";
        borderTopStyle = "solid";
        borderTopWidth = "2px";
        borderRightColor = "#CCCCCC";
        borderRightStyle = "solid";
        borderRightWidth = "2px";
        borderBottomColor = "#CCCCCC";
        borderBottomStyle = "solid";
        borderBottomWidth = "2px";
      }
    </SCRIPT>
  </BODY>
</HTML>
```

Ничего особо сложного здесь нет — мы просто присваиваем нужным атрибутам стиля (точнее, представляющим их свойствам объекта `CSSRule`) нужные значения. Единственная примечательная деталь — мы использовали описанный в *главе 4* блок with, чтобы сократить длину выполняющих эти присваивания выражений. Если бы мы его не использовали, то фрагмент сценария с этими выражениями выглядел бы так:

```
hdObj.style.borderLeftColor = "#CCCCCC";
hdObj.style.borderLeftStyle = "solid";
hdObj.style.borderLeftWidth = "2px";
hdObj.style.borderTopColor = "#CCCCCC";
hdObj.style.borderTopStyle = "solid";
hdObj.style.borderTopWidth = "2px";
```

```
hdObj.style.borderRightColor = "#CCCCCC";
hdObj.style.borderRightStyle = "solid";
hdObj.style.borderRightWidth = "2px";
hdObj.style.borderBottomColor = "#CCCCCC";
hdObj.style.borderBottomStyle = "solid";
hdObj.style.borderBottomWidth = "2px";
```

Кому как, но у автора от таких конструкций рябит в глазах. Поэтому он старается везде применять блоки with JavaScript или их аналоги в других языках программирования.

Вторая страница, которую мы напишем, будет посложнее. Мы разместим на ней несколько абзацев, заголовок и горизонтальную линию и в сценарии зададим для них разные цвета: для абзацев — синий, для заголовка — красный, для горизонтальной линии — серый. Далее приведен HTML-код, который у нас получится.

```
<HTML>
  <HEAD>
    <TITLE>Работа со стилями</TITLE>
  </HEAD>
  <BODY>
    <H1>Здравствуйте!</H1>
    <P>Это пример работы с объектом CSSRule.</P>
    <P>Сейчас мы с помощью особого сценария изменим цвета всех элементов
    этой страницы.</P>
    <P>Обычные абзацы станут синими.</P>
    <P>Заголовок станет красным.</P>
    <P>А горизонтальная линия - серой.</P>
    <HR>
    <P>Вот так!</P>
    <SCRIPT>
      var childColl = document.body.childNodes;
      var childCount = childColl.length;
      var childObj;
      for (var i = 0; i < childCount; i++) {
        childObj = childColl[i];
        switch (childObj.tagName) {
          case "P" :
            childObj.style.color = "#0000FF";
            break;
          case "H1" :
```

```
            childObj.style.color = "#FF0000";
            break;
          case "HR" :
            childObj.style.color = "#CCCCCC";
        }
      }
    </SCRIPT>
  </BODY>
</HTML>
```

Для "обхода" всех элементов страницы мы используем коллекцию `childNodes` объекта `HTMLElement`. Секция тела страницы, как мы помним, представляет собой экземпляр объекта `HTMLBodyElement`, который порожден от объекта `HTMLElement`, а значит, наследует эту коллекцию. А свойство `tagName` того же объекта `HTMLElement`, описанное в *главе 5*, возвращает имя тега, с помощью которого создан данный элемент страницы.

Третья страница будет содержать таблицу из четырех строк и двух столбцов и сценарий, который задаст для каждой четной строки таблицы серый цвет фона.

```
<HTML>
  <HEAD>
    <TITLE>Работа со стилями</TITLE>
  </HEAD>
  <BODY>
    <TABLE ID="tab">
      <TR><TD>1</TD><TD>2</TD></TR>
      <TR><TD>3</TD><TD>4</TD></TR>
      <TR><TD>5</TD><TD>6</TD></TR>
      <TR><TD>7</TD><TD>8</TD></TR>
    </TABLE>
    <SCRIPT>
      var tabObj = document.getElementById("tab");
      var bodyObj = tabObj.tBodies[0];
      for (var i = 1; i < bodyObj.rows.length; i = i + 2) {
        bodyObj.rows[i].style.backgroundColor = "#CCCCCC";
      }
    </SCRIPT>
  </BODY>
</HTML>
```

Здесь мы сначала получаем саму таблицу `tab`, а потом — через коллекцию `tBodies` — ее секцию тела. Поскольку мы явно не создали никаких секций в таблице, секция тела будет сформирована неявно самим Web-обозревателем. Чтобы получить доступ к строкам этой секции, мы используем коллекцию `rows`. Больше рассказывать особо нечего: мы обходим в цикле все строки этой секции и задаем для них серый цвет фона (RGB-код — `#CCCCCC`). Отметим только два момента.

- В цикле мы задаем начальное значение переменной-счетчика `i` равным 1 — это позволит нам начать обход со второй строки. (Не забываем, что нумерация элементов коллекции начинается с нуля.)
- При каждом проходе цикла мы увеличиваем значение переменной-счетчика на 2, чтобы перейти на следующую четную строку.

Что ж, со стилями CSS мы закончили. Пора начать разговор о событиях, поддерживаемых элементами страницы, а точнее — объектом `HTMLElement`. Этот объект поистине неисчерпаем.

События элементов страницы и их обработка

Рассказ о событиях, поддерживаемых объектом `HTMLElement`, то есть элементами страницы, будет довольно долгим. Чтобы не запутаться, давайте рассмотрим все эти события по группам: события мыши, события клавиатуры и прочие события, не относящиеся к мыши и клавиатуре. Так нам будет проще.

События мыши

Посетитель, бродя по Web-сайтам, в основном орудует мышью, иногда — клавиатурой. Поэтому рассмотрение событий, поддерживаемых объектом `HTMLElement`, мы начнем именно с событий мыши.

Все "мышиные" события перечислены в табл. 8.2.

Таблица 8.2. *События мыши, поддерживаемые объектом* `HTMLElement`

Событие	Строка DOM для Firefox	Описание
`onClick`	"click"	Возникает при щелчке мышью на элементе страницы
`onDblClick`		Возникает при двойном щелчке мышью на элементе страницы

Таблица 8.2 *(окончание)*

Событие	Строка DOM для Firefox	Описание
onMouseDown	"mousedown"	Возникает при нажатии кнопки мыши, когда ее курсор находится над элементом страницы, перед событиями onMouseUp и onClick
onMouseEnter		Возникает, когда курсор мыши помещается на элемент страницы. В отличие от аналогичного события onMouseOver, не всплывает, и поведение по умолчанию для него не может быть отменено. Поддерживается только Internet Explorer
onMouseLeave		Возникает, когда курсор мыши уводится за границы элемента страницы. В отличие от аналогичного события onMouseOut, не всплывает, и поведение по умолчанию для него не может быть отменено. Поддерживается только Internet Explorer
onMouseMove	"mouse-move"	Возникает при перемещении курсора мыши над элементом страницы
onMouseOut	"mouseout"	Возникает, когда курсор мыши уводится за границы элемента страницы
onMouseOver	"mouseover"	Возникает, когда курсор мыши наводится на элемент страницы
onMouseUp	"mouseup"	Возникает при отпускании кнопки мыши, когда ее курсор находится над элементом страницы, после события onMouseDown и перед событием onClick
onMouseWheel		Возникает при вращении колесика мыши, когда ее курсор находится над элементом страницы. Поддерживается только Internet Explorer и Opera

Все эти события всплывают, за исключением событий onMouseEnter и onMouseLeave. Поведение по умолчанию для всех этих событий, за исключением onMouseEnter и onMouseLeave, может быть отменено. (О всплытии событий и поведении по умолчанию см. в *главе 6*.)

Для привязки событий мыши к элементам страницы мы можем использовать любой подход из описанных в *главе 6*. Это может быть использование соответствующих атрибутов тега или свойств объекта HTMLElement, "отвечающих" за события, в модели Internet Explorer или применение модели Firefox. Исключением являются только те события, для которых в табл. 8.2 не указана строка DOM — обрабатывать их можно только по модели Internet Explorer.

А теперь — кое-какие важные пояснения. Если мы посмотрим на табл. 8.2, то увидим, что существуют события `onClick`, `onMouseDown` и `onMouseUp`, сигнализирующие о том, что посетитель щелкнул мышью по элементу страницы. Эти события возникают в такой последовательности:

1. `onMouseDown`.
2. `onMouseUp`.
3. `onClick`.

Если же посетитель выполнит двойной щелчок мышью по элементу страницы, возникнет такая последовательность событий:

1. `onMouseDown`.
2. `onMouseUp`.
3. `onClick`.
4. `onMouseUp`.
5. `onDblClick`.

Заметим, что все обработчики, привязанные к событиям `onMouseDown`, `onMouseUp` и `onClick` (в последнем случае) будут выполнены. Это нужно иметь в виду.

Мы можем использовать для получения дополнительной информации о возникшем событии свойства объекта `Event`, описанные в *главе 6*. В данном случае нас, вероятно, будут интересовать свойства, указывающие местоположение курсора мыши, нажатую на ней кнопку и состояние клавиш <Alt>, <Shift> и <Ctrl> клавиатуры.

В случае событий `onMouseOver` и `onMouseOut` (а в случае Internet Explorer — также и `onMouseEnter` и `onMouseLeave`) пригодятся следующие свойства:

- `fromElement` — возвращает элемент страницы, на котором ранее находился курсор мыши, при возникновении события `onMouseOver` или `onMouseEnter` (поддерживается Internet Explorer и Opera);
- `toElement` — возвращает элемент страницы, на который был наведен курсор мыши, при возникновении события `onMouseOut` или `onMouseLeave` (поддерживается Internet Explorer и Opera);
- `relatedTarget` — возвращает элемент страницы, на котором ранее находился курсор мыши, при возникновении события `onMouseOver` или элемент страницы, на который был наведен курсор мыши, при возникновении события `onMouseOut` (поддерживается Firefox).

Теперь в качестве примеров давайте напишем три страницы, которые будут обрабатывать события мыши. Первая из этих страниц — самая простая — будет

обрабатывать щелчки и двойные щелчки мышью на абзаце. Ее HTML-код приведен далее.

```
<HTML>
  <HEAD>
    <TITLE>События</TITLE>
    <SCRIPT>
      function parClick() {
        var parObj = document.getElementById("par");
        parObj.firstChild.nodeValue = "Щелчок!";
      }

      function parDblClick() {
        var parObj = document.getElementById("par");
        parObj.firstChild.nodeValue = "Двойной щелчок!";
      }
    </SCRIPT>
  </HEAD>
  <BODY>
    <P ID="par" ONCLICK="parClick();"
    ONDBLCLICK="parDblClick();">Щелкните здесь один или два раза.</P>
  </BODY>
</HTML>
```

Думается, комментарии здесь излишни.

Вторая страница будет обрабатывать наведение курсора мыши на абзац, увод его прочь и щелчок мышью на абзаце. При этом наведении курсора мыши цвет текста абзаца будет меняться на синий, при уводе — снова на черный, а при щелчке — выполняться переход на другую страницу. То есть фактически мы программно создадим аналог обычной гиперссылки.

```
<HTML>
  <HEAD>
    <TITLE>Что-то наподобие гиперссылки</TITLE>
    <SCRIPT>
      function parClick() {
        location.href = "http://www.somesite.ru";
      }

      function parMouseOver() {
        var parObj = document.getElementById("par");
```

```
        parObj.style.color = "#0000FF";
      }

      function parMouseOut() {
        var parObj = document.getElementById("par");
        parObj.style.color = "#000000";
      }
    </SCRIPT>
  </HEAD>
  <BODY>
    <P ID="par" ONCLICK="parClick();" ONMOUSEOVER="parMouseOver();"
    ONMOUSEOUT="parMouseOut();">Перейти на сайт.</P>
  </BODY>
</HTML>
```

Здесь тоже все должно быть понятно. Единственное — для пущего сходства с гиперссылкой следует также сменить форму курсора мыши для абзаца `par` с обычного текстового на "указующий перст". Это можно сделать с помощью атрибута стиля `cursor` (см. *главу 3*).

Третья страница будет содержать три гиперссылки и абзац. При наведении курсора мыши на одну из гиперссылок в абзаце будет появляться поясняющий текст. Фактически мы создадим реализацию подсказок к элементам страницы, только не всплывающих. (О всплывающих подсказках HTML и их создании было рассказано в *главе 2*.)

```
<HTML>
  <HEAD>
    <TITLE>Подсказки</TITLE>
    <SCRIPT>
      var hints = [];
      hints["href1"] = "Перейти на первую страницу";
      hints["href2"] = "Перейти на вторую страницу";
      hints["href3"] = "Перейти на третью страницу";
      var defaultHint = "Наведите курсор мыши на любую гиперссылку";

      function hrefMouseOver(pID) {
        var outputObj = document.getElementById("output");
        outputObj.firstChild.nodeValue = hints[pID];
      }

      function hrefMouseOut() {
        var outputObj = document.getElementById("output");
```

```
        outputObj.firstChild.nodeValue = defaultHint;
      }
    </SCRIPT>
  </HEAD>
  <BODY>
    <P><A ID="href1" HREF="page1.html"
    ONMOUSEOVER="hrefMouseOver('href1');"
    ONMOUSEOUT="hrefMouseOut();">Страница 1</A></P>
    <P><A ID="href2" HREF="page2.html"
    ONMOUSEOVER="hrefMouseOver('href2');"
    ONMOUSEOUT="hrefMouseOut();">Страница 2</A></P>
    <P><A ID="href3" HREF="page3.html"
    ONMOUSEOVER="hrefMouseOver('href3');"
    ONMOUSEOUT="hrefMouseOut();">Страница 3</A></P>
    <P ID="output">Наведите курсор мыши на любую гиперссылку</P>
  </BODY>
</HTML>
```

Рассмотрим HTML-код нашей третьей страницы, поскольку здесь применяются очень интересные приемы, которыми мы будем пользоваться в дальнейшем.

Прежде всего, для хранения текста подсказок для гиперссылок мы используем ассоциативный массив `hints` (об ассоциативных массивах было рассказано в *главе 4*). В качестве индексов элементов этого массива мы используем имена гиперссылок, заданные атрибутом `ID`, — так нам будет проще получить к ним доступ.

Обратим внимание, как мы создаем и наполняем элементами этот массив. Сначала мы создаем пустой массив, присвоив переменной `hints` "пустые" квадратные скобки. Потом мы создаем элементы этого массива обычным, описанным в *главе 4* способом.

В переменной `defaultHint` хранится текст подсказки по умолчанию, выводящийся на экран, если курсор мыши не наведен ни на одну из гиперссылок.

Вывод подсказки выполняется функцией — обработчиком события `onMouseOver`. Это функция `hrefMouseOver`. Мы передаем ее в качестве единственного параметра имя гиперссылки, к которой она привязана. (О том, что функциям — обработчикам событий можно передавать параметры, мы узнали еще в *главе 6*.) Например, в случае первой гиперссылки:

```
ONMOUSEOVER="hrefMouseOver('href1');"
```

Функция `hrefMouseOver` использует его, чтобы извлечь из ассоциативного массива `hints` текст подсказки, соответствующий этой гиперссылке, и по-

местить его в абзац `output` — именно в нем выводятся подсказки. Отметим также, что строку с именем гиперссылки мы заключили в одинарные кавычки, так как код обработчика уже заключен в двойные кавычки, и поместить в него еще одну пару двойных кавычек мы так просто не сможем.

Передача имени элемента страницы, в котором возникло событие, в функцию-обработчик через параметр избавила нас от многих проблем. Если бы мы использовали для получения гиперссылки, в которой возникло событие `onMouseOver`, свойства объекта `Event`, то нам пришлось бы, дабы учесть различия в поддержке этого объекта в разных Web-обозревателях, писать два обработчика этого события: один — для Internet Explorer и Opera, второй — для Firefox. Лишняя и абсолютно ненужная работа…

После того как посетитель убрал курсор мыши с гиперссылки, в абзаце `output` следует вывести текст подсказки по умолчанию, хранящийся в переменной `defaultHint`. Этим занимается функция — обработчик события `onMouseOut` по имени `hrefMouseOut`. Она ничем не примечательна.

Это первый из самых часто используемых способов создать подсказки с помощью сценариев. Второй способ — с помощью свободно позиционируемых элементов — мы рассмотрим в *главе 10*.

События клавиатуры

Клавиатура — второй по значимости "виновник" возникновения событий. Поэтому специфическим событиям объекта `HTMLElement`, возникающим при нажатии клавиши на клавиатуре, мы посвятим целый параграф.

Все клавиатурные события перечислены в табл. 8.3. Как видим, их всего три.

Таблица 8.3. *События клавиатуры, поддерживаемые объектом* `HTMLElement`

Событие	Описание
`onKeyDown`	Возникает, когда посетитель нажимает любую клавишу клавиатуры. Если клавиша удерживается нажатой, возникает последовательно
`onKeyPress`	Возникает, когда посетитель нажимает алфавитно-цифровую клавишу. Если клавиша удерживается нажатой, возникает последовательно
`onKeyUp`	Возникает, когда посетитель отпускает нажатую ранее клавишу

Все события клавиатуры всплывают, и поведение по умолчанию для них может быть отменено. (О всплытии событий и поведении по умолчанию см. в *главе 6*.)

Строки DOM для этих событий отсутствуют. Это значит, что мы можем обрабатывать их только по модели Internet Explorer.

А теперь нужно дать необходимые пояснения. Все три перечисленные в табл. 8.3 события клавиатуры возникают при нажатии клавиши на клавиатуре. Но все зависит от типа нажатой клавиши. Так, при нажатии алфавитно-цифровой клавиши, то есть при вводе любого символа, возникает такая последовательность событий:

1. `onKeyDown`.
2. `onKeyPress`.
3. `onKeyUp`.

Если же посетитель нажмет управляющую клавишу (например, <Backspace>, <Ins>, <Del>, клавишу-стрелку), возникает укороченная последовательность событий:

1. `onKeyDown`.
2. `onKeyUp`.

То есть событие `onKeyPress` возникает только при нажатии алфавитно-цифровой клавиши — при вводе символа.

Часто бывает, что посетитель удерживает какую-то клавишу нажатой достаточно долгое время. В этом случае события `onKeyDown` и `onKeyPress` возникают последовательно, одно за другим, пока клавиша не будет отпущена. Событие `onKeyUp` в данном случае возникает всего один раз, когда посетитель отпустит нажатую клавишу.

При обработке событий клавиатуры мы можем использовать для получения дополнительной информации о возникшем событии свойства объекта `Event`, описанные в *главе 6*. Актуальными будут свойства, указывающие код нажатой клавиши, код введенного символа, состояние клавиш <Alt>, <Shift> и <Ctrl> и некоторые другие.

Сначала нужно дать понятие кода клавиши и кода символа. Это совершенно разные вещи, и нам обязательно нужно знать, в чем их различие.

Код символа обозначает символ, который посетитель ввел с клавиатуры. Так, если он нажал клавишу <A>, то будет введен символ "a", десятичный код которого — 97. (Предполагается, что режим CapsLock, управляемый одноименной клавишей, отключен.) Если же он нажмет клавишу <A>, удерживая нажатой клавишу <Shift>, то будет введен символ "A", чей десятичный код — 65.

Код клавиши обозначает код нажатой на клавиатуре клавиши, вне зависимости от того, включен ли режим CapsLock и удерживается ли нажатой клави-

ша <Shift>. Например, клавиша <A> имеет десятичный код 65 (как мы видим, он соответствует символу "A"), клавиша <Shift> — код 16, клавиша <Ctrl> — 17, <Alt> — 18, <CapsLock> — 20. При этом в случае нажатия клавиш <Shift>, <Ctrl>, <Alt> или <CapsLock>, как и других управляющих клавиш, никакого ввода символа не произойдет.

Разобравшись с кодами клавиш и символов, давайте кратко опишем свойства, с помощью которых мы можем выяснить, какая клавиша нажата на клавиатуре. Другие описанные в *главе 6* свойства объекта `Event` пояснений не требуют.

Начнем с объекта `Event`, поддерживаемого Internet Explorer и Opera. Здесь будет представлять интерес свойство `keyCode`. Оно возвращает:

- код нажатой клавиши в обработчиках событий `onKeyDown` и `onKeyUp`;
- код введенного символа в обработчике события `onKeyPress`.

Все эти коды возвращаются в кодировке Unicode.

Firefox же несколько мудреней. Свойства, "отвечающие" за код нажатой клавиши, в его случае требуют более подробного разбора. Чем мы сейчас и займемся.

Предположим, что посетитель нажал алфавитно-цифровую клавишу, то есть ввел символ. Тогда в обработчиках событий `onKeyDown` и `onKeyUp` свойство `keyCode` будет содержать код нажатой клавиши в кодировке Unicode. А в обработчике события `onKeyPress` свойство `charCode` будет содержать код уже введенного символа. В обработчиках всех трех событий свойство `which` также вернет код введенного символа.

Теперь предположим, что посетитель нажал управляющую клавишу. В обработчиках событий `onKeyDown` и `onKeyUp` все, как и в первом случае, свойство `keyCode` содержит код нажатой клавиши в кодировке Unicode. Событие `onKeyPress` в этом случае не возникает. Свойство `which` также вернет код нажатой клавиши.

Все сказанное ранее можно описать другими словами.

- Свойство `keyCode` в обработчиках событий `onKeyDown` и `onKeyUp` возвращает код нажатой клавиши.
- Свойство `charCode` в обработчике события `onKeyPress` возвращает код введенного символа.
- Свойство `which` в обработчиках всех трех событий возвращает код нажатой управляющей клавиши или введенного символа.

Повторим, что все это справедливо только для Firefox.

Теперь для примера напишем две страницы, которые будут определять коды нажатой клавиши и введенного символа: одну — для Internet Explorer и Opera, другую для Firefox. Эти страницы пригодятся нам в дальнейшем, когда мы будет обрабатывать события клавиатуры, и нам понадобятся коды клавиш и символов.

Вот HTML-код страницы для Internet Explorer и Opera:

```
<HTML>
  <HEAD>
    <TITLE>Коды клавиш и символов</TITLE>
    <SCRIPT>
      function bodyKeyDown() {
        var outputKeyObj = document.getElementById("outputKey");
        var outputCharObj = document.getElementById("outputChar");
        outputKeyObj.firstChild.nodeValue = "Код нажатой клавиши: " +
        event.keyCode.toString();
        outputCharObj.firstChild.nodeValue = " ";
      }

      function bodyKeyPress() {
        var outputCharObj = document.getElementById("outputChar");
        outputCharObj.firstChild.nodeValue = "Код введенного символа: " +
        event.keyCode.toString();
      }
    </SCRIPT>
  </HEAD>
  <BODY ONKEYDOWN="bodyKeyDown();" ONKEYPRESS="bodyKeyPress();">
    <P ID="outputKey"> </P>
    <P ID="outputChar"> </P>
  </BODY>
</HTML>
```

> ***НА ЗАМЕТКУ***
>
> Почему-то Opera не всегда корректно обрабатывает страницу, код которой приведен ранее. Так что лучше для выяснения кодов клавиш и символов использовать Internet Explorer, который, к тому же, стандартно поставляется в составе Windows.

Здесь обработчик события `onKeyDown` занимается выводом на страницу кода нажатой клавиши, а обработчик события `onKeyPress` — кода введенного символа. Оба этих обработчика привязаны к секции тела страницы (тегу `<BODY>`).

Код нажатой клавиши выводится в абзаце `outputKey`, а код введенного символа — в абзаце `outputChar`.

Отметим, что мы изначально создали содержимое абзацев `outputKey` и `outputChar` — неразрывный пробел. Впоследствии обработчики событий `onKeyDown` и `onKeyPress` будут его изменять. Если мы не создадим содержимое этих абзацев, то должны будем создать его программно — лишняя работа и лишний код.

Отметим, что обработчик события `onKeyDown` также очищает абзац `outputChar`, куда выводится код введенного символа, занося в него пробел (опять же, чтобы в нем уже присутствовало какие-то содержимое). Поясним, зачем это нужно. Пусть мы открыли эту страницу в Web-обозревателе и нажали любую алфавитно-цифровую клавишу. Тогда в абзаце `outputKey` появится код этой клавиши, а в абзаце `outputChar` — код соответствующего ей символа.

После этого мы нажмем управляющую клавишу. В абзаце `outputKey` будет выведен ее код — это обеспечит обработчик события `onKeyDown`. Но событие `onKeyPress` в случае нажатия управляющей клавиши не возникнет, и в абзаце `outputChar` останется код введенного ранее символа, что будет выглядеть некрасиво. Поэтому мы и очищаем абзац `outputChar` в обработчике события `onKeyDown`.

Остальной код комментариев не требует, так что перейдем к аналогичной странице, но для Firefox. Ее HTML-код приведен далее.

```
<HTML>
  <HEAD>
    <TITLE>Коды клавиш и символов</TITLE>
    <SCRIPT>
      function bodyKeyDown(evt) {
        var outputKeyObj = document.getElementById("outputKey");
        var outputCharObj = document.getElementById("outputChar");
        outputKeyObj.firstChild.nodeValue = "Код нажатой клавиши: " +
        evt.keyCode.toString();
        outputCharObj.firstChild.nodeValue = " ";
      }

      function bodyKeyPress(evt) {
        var outputCharObj = document.getElementById("outputChar");
        outputCharObj.firstChild.nodeValue = "Код введенного символа: " +
```

```
        evt.charCode.toString();
      }
    </SCRIPT>
  </HEAD>
  <BODY ONKEYDOWN="bodyKeyDown(event);"
  ONKEYPRESS="bodyKeyPress(event);">
    <P ID="outputKey"> </P>
    <P ID="outputChar"> </P>
  </BODY>
</HTML>
```

Здесь практически все то же самое, что и приведенном ранее коде. За исключением — и это понятно — особенностей объекта `Event` в Firefox, которые были описаны в *главе 6*.

Прочие события

На закуску оставим все остальные события, не относящиеся к клавиатуре и мыши. Их всего два, обрабатываются они крайне редко, только в особых случаях, и поддерживаются не всеми Web-обозревателями.

Internet Explorer поддерживает событие `onContextMenu`. Оно возникает, когда посетитель щелкает по элементу страницы правой кнопкой мыши, чтобы вывести контекстное меню, но перед выводом самого этого меню.

Поведение по умолчанию для этого события — вывод контекстного меню, — но его можно отменить, воспользовавшись одним из способов, описанных в *главе 6*. Кроме того, это событие всплывает.

Событие `onContextMenu` может нам пригодится, если мы захотим запретить для своей страницы вывод контекстного меню. Для этого достаточно вставить в код, формирующий секцию тела страницы (тег `<BODY>`), такой обработчик этого события:

```
<BODY . . . ONCONTEXTMENU="return false;">
```

То есть мы фактически отменим поведение по умолчанию для этого события.

Internet Explorer поддерживает также событие `onPropertyChange`, возникающее при изменении любого свойства текущего элемента страницы. В обработчике этого события для получения имени свойства, значение которого изменилось, можно использовать свойство `propertyName` объекта `Event` — оно как раз и вернет это имя в строковом виде. Данное событие не имеет поведения по умолчанию и не всплывает.

НА ЗАМЕТКУ

Полезность события `onPropertyChange` вызывает большие сомнения. В самом деле, если какой-то сценарий изменяет значение свойства элемента страницы, то зачем ему или другому сценарию, расположенному на этой же странице, знать, что оно было изменено? Он и так это знает!

Прочие свойства и методы элементов страницы

Напоследок рассмотрим несколько свойств и методов, поддерживаемых объектом `HTMLElement`, о которых мы еще не говорили. Их не так уж и много.

Начнем со свойств. Все они перечислены в табл. 8.4 и доступны только для чтения.

Таблица 8.4. *Прочие свойства объекта* `HTMLElement`

Свойство	Описание
`clientHeight`	Возвращает высоту клиентской области элемента страницы
`clientLeft`	Возвращает расстояние от левой границы клиентской области родителя до левой границы клиентской области текущего элемента страницы по горизонтали
`clientTop`	Возвращает расстояние от верхней границы клиентской области родителя до верхней границы клиентской области текущего элемента страницы по вертикали
`clientWidth`	Возвращает ширину клиентской области элемента страницы
`offsetHeight`	Возвращает полную высоту элемента страницы
`offsetLeft`	Возвращает расстояние от левой границы клиентской области родителя до левой границы текущего элемента страницы по горизонтали
`offsetTop`	Возвращает расстояние от верхней границы клиентской области родителя до верхней границы текущего элемента страницы по вертикали
`offsetWidth`	Возвращает полную ширину элемента страницы
`offsetParent`	Возвращает элемент страницы, относительно которого вычисляются значения свойств `offsetLeft` и `offsetTop`. Как правило, это родитель текущего элемента страницы

Таблица 8.4 *(окончание)*

Свойство	Описание
`parentNode`	Возвращает элемент страницы — родитель текущего элемента
`scrollHeight`	Возвращает полную высоту содержимого элемента страницы
`scrollLeft`	Возвращает позицию бегунка горизонтальной полосы прокрутки элемента страницы
`scrollTop`	Возвращает позицию бегунка вертикальной полосы прокрутки элемента страницы
`scrollWidth`	Возвращает полную ширину содержимого элемента страницы
`sourceIndex`	Возвращает индекс текущего элемента страницы в коллекции `all`. Поддерживается только Internet Explorer и Opera

Клиентская область элемента страницы — это свободное пространство внутри него, доступное для размещения содержимого потомков. Клиентская область включает "внутренние" отступы, но не включает "внешние" отступы, рамку и полосы прокрутки (если они есть). (О задании отступов и рамок с помощью соответствующих атрибутов стиля см. *главу 3*.)

Internet Explorer и Opera поддерживают метод `contains`, позволяющий узнать, содержит ли текущий элемент страницы заданный в качестве параметра.

```
contains(<возможный потомок>)
```

Он возвращает `true`, если заданный элемент страницы является потомком текущего, и `false` в противном случае.

```
<DIV ID="cont">
  <P ID="par">Это абзац.</P>
</DIV>
. . .
<SCRIPT>
  var contObj = document.getElementById("cont");
  var parObj = document.getElementById("par");
  var flag = contObj.contains(parObj);
</SCRIPT>
```

Приведенный сценарий поместит в переменную `flag` значение `true`, так как абзац `par` является потомком контейнера `cont`.

Вот, пожалуй, и все об управлении содержимым Web-страницы.

Что дальше?

В этой главе мы занимались тем, что меняли содержимое уже загруженной в Web-обозреватель страницы как нам заблагорассудится и обрабатывали события, поддерживаемые элементами страницы. Теперь любая страница в нашей власти!

В следующих главах мы продолжим работать с содержимым страниц. Мы изучим специфические возможности, предлагаемые графическими изображениями, таблицами, контейнерами, формами, элементами управления и мультимедийными элементами. (О формах, элементах управления и мультимедийных элементах разговора еще не было, но будет.) И начнем с графических изображений и мультимедиа.

Глава 9

Управление графикой и мультимедийными элементами

В прошлой главе мы научились изменять содержимое страницы уже после ее загрузки в Web-обозревателе. В этой главе мы продолжим заниматься тем же и сосредоточимся на графических изображениях и мультимедийных элементах.

Помещать на страницы графические изображения мы научились еще в *главе 2*. В *главе 5* мы узнали, что каждое графическое изображение, помещенное на страницу, представляется Web-обозревателем как экземпляр объекта `HTMLImageElement`, производного от объекта `HTMLElement`. Объект `HTMLImageElement` добавляет некоторые дополнительные свойства, специфические для графического изображения, поскольку создающий его тег `<IMG>` поддерживает некоторые дополнительные атрибуты (см. *главу 2*). Также он поддерживает дополнительные события, отражающие особую природу интернет-графики — ведь, как мы помним, изображения хранятся в отдельных файлах и загружаются Web-обозревателем отдельно от самой страницы.

Одной из разновидностей графических изображений являются карты-изображения, также описанные в *главе 2*. Каждая карта-изображение представляется Web-обозревателем как экземпляр объекта `HTMLImageElement`, поскольку ничем не отличается от обычного изображения. Карта, описывающая конфигурацию горячих областей, представляется как экземпляр объекта `HTMLMapElement`, а каждая горячая область, описанная в карте, — как экземпляр объекта `HTMLAreaElement`. Эти объекты и все поддерживаемые ими свойства мы обязательно рассмотрим.

Мультимедийные элементы — аудио- и видеоролики, помещенные на страницу, — это особый разговор. Давайте отложим их рассмотрение на потом, иначе вводная часть к этой главе выйдет слишком длинной. Ведь мы даже еще не знаем, как поместить их на страницу.

Работа с обычными графическими изображениями

Начнем с обычных графических изображений. Именно они сейчас наиболее популярны и чаще всего используются в Web-программировании для достижения различных эффектов. К тому же, работать с ними проще всего.

Как уже было сказано, каждое присутствующее на странице изображение представляется как экземпляр объекта `HTMLImageElement`. Этот объект порожден от объекта `HTMLElement`, который нам уже знаком по *главе 8*. Нам осталось только рассмотреть специфические свойства и события, поддерживаемые этим объектом. Их не так уж и много.

Чтобы закрепить полученные знания, мы создадим два примера: изображение, меняющееся при наведении на него курсора мыши, и полоса навигации — набор изображений-гиперссылок. Если мы впоследствии займемся Web-дизайном, то умение создать полосу навигации нам очень пригодится.

Свойства и события объекта *HTMLImageElement*

Специфических свойств объект `HTMLImageElement` поддерживает всего пять, из них только четыре поддерживаются всеми Web-обозревателями.

Свойство `src` задает или возвращает интернет-адрес файла, где хранится графическое изображение. С его помощью мы можем сменить изображение, отображаемое на странице, другим.

```
var someImgObj = document.getElementById("someimg");
someImgObj.src = "otherimg.gif";
```

Этот сценарий как раз и меняет изображение.

Свойство `alt` задает или возвращает текст замены, заданный для текущего изображения.

Свойство `lowsrc` задает или возвращает интернет-адрес файла, где хранится графическое изображение пониженного качества (подробнее см. *главу 2*).

Если нам нужно быстро узнать, загружено ли изображение, мы можем воспользоваться свойством `complete`. Это свойство возвращает `true`, если изображение загружено и выведено на страницу, и `false` в противном случае.

Internet Explorer также поддерживает свойство `readyState`. Оно возвращает состояние изображения в виде одной из перечисленных далее строк:

- "uninitialized" — изображение еще не загружено;
- "loading" — изображение загружается;

- "loaded" — изображение загружено, но еще не выведено на страницу;
- "complete" — изображение загружено и выведено на страницу.

Вероятно, удобнее в этом случае использовать свойство `complete`, нежели `readyState`. С ним все проще.

Специфических событий объект `HTMLImageElement` поддерживает четыре. Все они перечислены в табл. 9.1.

Таблица 9.1. *События объекта* `HTMLImageElement`

Событие	Строка DOM	Описание
`onAbort`	"abort"	Возникает при прерывании загрузки изображения
`onError`	"error"	Возникает при неудачной загрузке изображения
`onLoad`	"load"	Возникает после завершения загрузки изображения
`onReadyStateChange`		Возникает при изменении состояния изображения, фактически — при изменении значения свойства `readyState`. Поддерживается только Internet Explorer

Ни одно из этих событий не всплывает. Поведение по умолчанию может быть прервано только у событий `onAbort` и `onError`. У события `onAbort` поведение по умолчанию — прерывание загрузки, у события `onError` — вывод сообщения об ошибке.

Кроме того, объект `HTMLImageElement` поддерживает все события объекта `HTMLElement`, поскольку является его "наследником". Как правило, именно события объекта `HTMLElement` и обрабатываются.

Ну что, рассмотрим обещанные примеры?

Горячее изображение

Первый пример — так называемое *горячее изображение*. Это изображение, которое меняется при наведении курсора мыши. Средствами HTML его создать невозможно, а с помощью сценариев — запросто.

Подготовим два графических изображения и дадим файлам, в которых они хранятся, имена default.jpg и hover.jpg (если это изображения в формате JPEG; в случае форматов GIF или PNG они будут иметь другие расширения, соответствующие формату). Первое изображение будет выводиться на стра-

ницу изначально (*изначальное изображение*), второе — при наведении на горячее изображение курсора мыши (*заменяющее изображение*).

> ***Внимание!***
>
> Очень важно, чтобы и изначальное, и заменяющее изображения имели одинаковые ширину и высоту. Иначе получится некрасиво.

HTML-код страницы, содержащей горячее изображение, приведен далее.

```
<HTML>
  <HEAD>
    <TITLE>Горячее изображение</TITLE>
    <SCRIPT>
      function imgMouseOver() {
        var hotImgObj = document.getElementById("hotimg");
        hotImgObj.src = "hover.jpg";
      }

      function imgMouseOut() {
        var hotImgObj = document.getElementById("hotimg");
        hotImgObj.src = "default.jpg";
      }
    </SCRIPT>
  </HEAD>
  <BODY>
    <P><IMG ID="hotimg" SRC="default.jpg" ONMOUSEOVER="imgMouseOver();"
    ONMOUSEOUT="imgMouseOut();"></P>
  </BODY>
</HTML>
```

Здесь особо нечего комментировать. Обработчик события `onMouseOver` меняет изначальное изображение на заменяющее, а обработчик события `onMouseOut` выполняет обратную замену. Оба этих события мы рассмотрели в *главе 8* — это события мыши.

Отметим, что в теге `<IMG>`, создающем изображение, мы указали файл изначального изображения (значение "default.jpg" атрибута `SRC`). Благодаря этому после загрузки страницы изначальное изображение уже будет на ней присутствовать.

Полученное нами горячее изображение мы можем тут же превратить в изображение-гиперссылку. Как сказано в *главе 2*, для этого достаточно поместить его в тег `<A>`.

НА ЗАМЕТКУ

Так часто и поступают на практике — изображение-гиперссылку превращают в горячее изображение. Такое изображение-гиперссылка намного нагляднее, чем обычное, "негорячее".

Первый написанный нами пример слишком прост. Давайте сделаем что-нибудь посложнее. А именно — полосу навигации.

Полоса навигации

Полоса навигации — это набор горячих изображений-гиперссылок, как правило, ссылающихся на разные страницы сайта. Эти изображения выстраиваются в полосу (отсюда и название), горизонтальную или вертикальную; чаще всего встречаются вертикальные полосы навигации. Практически всегда для создания полосы навигации используются таблицы HTML: создается таблица из одной строки или столбца, и горячие изображения, составляющие полосу навигации, помещаются в ее ячейки.

Полосы навигации применяются очень часто, так что наш второй пример будет более приближенным к практике.

При создании полосы навигации нам придется решить несколько задач.

1. При наведении курсора мыши на изображение-гиперссылку поместить на его место заменяющее изображение. Это мы уже умеем делать.
2. При щелчке на изображении-гиперссылке нужно сменить его особым заменяющим изображением, отличным от тех, что применяются для других изображений-гиперссылок.
3. Изображение-гиперссылка, соответствующее текущей странице, должно отличаться от других. Как правило, для его выделения используют то же графическое изображение, что и для выделения изображения-гиперссылки, на котором щелкнули мышью.
4. При наведении курсора мыши на изображение-гиперссылку, соответствующее текущей странице, нужно подставить на его место заменяющее изображение, отличное от заменяющих изображений для других изображений-гиперссылок.
5. Нужно как-то дать знать сценариям, реализующим полосу навигации, какое изображение-гиперссылка соответствует текущей странице. Это нужно для того, чтобы его выделить.

Исходя из всего сказанного ранее, нам для каждого изображения-гиперссылки понадобятся четыре графических изображения:

- изначальное;
- заменяющее, выводимое при щелчке мышью, а также в случае изображения-гиперссылки, представляющей текущую страницу;
- заменяющее, выводимое при наведении курсора мыши на изображение-гиперссылку в обычном случае;
- заменяющее, выводимое при наведении курсора мыши на изображение-гиперссылку, представляющую текущую страницу.

Давайте для примера создадим страницу с полосой навигации, включающей три изображения-гиперссылки. Для каждого из них заготовим четыре графических изображения. Их имена будут начинаться с цифры, от 1 до 4, далее будет следовать знак подчеркивания, а за ним — одна из приведенных далее строк:

- "up" — изначальное;
- "down" — заменяющее, выводимое при щелчке мышью, а также в случае изображения-гиперссылки, представляющей текущую страницу;
- "hover" — заменяющее, выводимое при наведении курсора мыши на изображение-гиперссылку в обычном случае;
- "hover_down" — заменяющее, выводимое при наведении курсора мыши на изображение-гиперссылку, представляющую текущую страницу.

Эта система наименования файлов придумана автором для собственного удобства. Если вам она покажется неудобной, можете использовать другую систему.

Предположим, что наши изображения хранятся в формате JPEG; тогда все эти файлы будут иметь расширение jpg. Если они хранятся в другом формате, расширение будет соответствовать этому формату.

HTML-код самой страницы приведен далее.

```
<HTML>
  <HEAD>
    <TITLE>Полоса навигации</TITLE>
    <SCRIPT>
      var ups = [];
      ups["link1"] = "1_up.jpg";
      ups["link2"] = "2_up.jpg";
```

```
    ups["link3"] = "3_up.jpg";

    var downs = [];
    downs["link1"] = "1_down.jpg";
    downs["link2"] = "2_down.jpg";
    downs["link3"] = "3_down.jpg";

    var hovers = [];
    hovers["link1"] = "1_hover.jpg";
    hovers["link2"] = "2_hover.jpg";
    hovers["link3"] = "3_hover.jpg";

    var hover_downs = [];
    hover_downs["link1"] = "1_hover_down.jpg";
    hover_downs["link2"] = "2_hover_down.jpg";
    hover_downs["link3"] = "3_hover_down.jpg";

    function imgMouseOver(pHref) {
      var aObj = document.getElementById(pHref);
      if (aObj == currentAObj)
        aObj.firstChild.src = hover_downs[pHref]
      else
        aObj.firstChild.src = hovers[pHref];
    }

    function imgMouseOut(pHref) {
      var aObj = document.getElementById(pHref);
      if (aObj == currentAObj)
        aObj.firstChild.src = downs[pHref]
      else
        aObj.firstChild.src = ups[pHref];
    }

    function imgClick(pHref) {
      var aObj = document.getElementById(pHref);
      aObj.firstChild.src = downs[pHref]
    }
  </SCRIPT>
</HEAD>
<BODY>
```

```
    <H1>Это первая страница</H1>
    <P><A ID="link1" HREF="9.1.htm"><IMG SRC="1_down.jpg"
    ONMOUSEOVER="imgMouseOver('link1');"
    ONMOUSEOUT="imgMouseOut('link1');"
    ONCLICK="imgClick('link1');"></A></P>
    <P><A ID="link2" HREF="9.2.htm"><IMG SRC="2_up.jpg"
    ONMOUSEOVER="imgMouseOver('link2');"
    ONMOUSEOUT="imgMouseOut('link2');"
    ONCLICK="imgClick('link2');"></A></P>
    <P><A ID="link3" HREF="9.3.htm"><IMG SRC="3_up.jpg"
    ONMOUSEOVER="imgMouseOver('link3');"
    ONMOUSEOUT="imgMouseOut('link3');"
    ONCLICK="imgClick('link3');"></A></P>
    <SCRIPT>
      var currentAObj = document.getElementById("link1");
    </SCRIPT>
  </BODY>
</HTML>
```

Сохраним эту страницу в файле с именем 9.1.htm. После этого создадим на ее основе еще две страницы с именами, соответственно, 9.2.htm и 9.3.htm. Их код будет почти таким же за небольшими исключениями. Так, код страницы 9.2.htm, формирующий секцию ее тела, будет таким (изменения выделены полужирным шрифтом):

```
<H1>Это вторая страница</H1>
<P><A ID="link1" HREF="9.1.htm"><IMG SRC="1_up.jpg"
ONMOUSEOVER="imgMouseOver('link1');"
ONMOUSEOUT="imgMouseOut('link1');"
ONCLICK="imgClick('link1');"></A></P>
<P><A ID="link2" HREF="9.2.htm"><IMG SRC="2_down.jpg"
ONMOUSEOVER="imgMouseOver('link2');"
ONMOUSEOUT="imgMouseOut('link2');"
ONCLICK="imgClick('link2');"></A></P>
<P><A ID="link3" HREF="9.3.htm"><IMG SRC="3_up.jpg"
ONMOUSEOVER="imgMouseOver('link3');"
ONMOUSEOUT="imgMouseOut('link3');"
ONCLICK="imgClick('link3');"></A></P>
<SCRIPT>
  var currentAObj = document.getElementById("link2");
</SCRIPT>
```

А код, формирующий секцию тела страницы 9.3.htm, будет таким (изменения также выделены полужирным шрифтом):

```
<H1>Это третья страница</H1>
<P><A ID="link1" HREF="9.1.htm"><IMG SRC="1_up.jpg"
ONMOUSEOVER="imgMouseOver('link1');"
ONMOUSEOUT="imgMouseOut('link1');"
ONCLICK="imgClick('link1');"></A></P>
<P><A ID="link2" HREF="9.2.htm"><IMG SRC="2_up.jpg"
ONMOUSEOVER="imgMouseOver('link2');"
ONMOUSEOUT="imgMouseOut('link2');"
ONCLICK="imgClick('link2');"></A></P>
<P><A ID="link3" HREF="9.3.htm"><IMG SRC="3_down.jpg"
ONMOUSEOVER="imgMouseOver('link3');"
ONMOUSEOUT="imgMouseOut('link3');"
ONCLICK="imgClick('link3');"></A></P>
<SCRIPT>
  var currentAObj = document.getElementById("link3");
</SCRIPT>
```

Загрузим в Web-обозреватель страницу 9.1.htm и наведем курсор мыши на любое изображение-гиперссылку. Изображение должно смениться. Теперь щелкнем по любому изображению-гиперссылке, и в Web-обозревателе будет загружена другая страница — 9.2.htm или 9.3.htm.

Осталось разобрать код этих страниц. Ничего особо нового для нас там не будет.

Прежде всего, для хранения имен файлов, где находятся изображения, мы используем ассоциативные массивы. Таких массивов четыре:

- `ups` — хранит имена файлов с изначальными изображениями;
- `downs` — хранит имена файлов с заменяющими изображениями, выводимыми при щелчке мышью, а также в случае изображения-гиперссылки, представляющей текущую страницу;
- `hovers` — хранит имена файлов с заменяющими изображениями, выводимыми при наведении курсора мыши на изображение-гиперссылку в обычном случае;
- `hover_downs` — хранит имена файлов с заменяющими изображениями, выводимыми при наведении курсора мыши на изображение-гиперссылку, представляющую текущую страницу.

В качестве индексов мы используем имена гиперссылок.

В HTML-коде, формирующем изображения-гиперссылки, мы изначально проставляем те имена файлов с изображениями, которые должны отображаться изначально. Так, на странице 9.1.htm в качестве первого изображения-гиперссылки должно присутствовать изображение 1_down.jpg, поскольку эта гиперссылка указывает на страницу 9.1.htm, то есть текущую страницу, а в качестве второго и третьего изображений-гиперссылок — изображения 2_up.jpg и 3_up.jpg.

Далее мы сохраняем гиперссылку, представляющую текущую страницу, в переменной `currentAObj` — она нам потом пригодится. Отметим, что сценарий, выполняющий это сохранение, находится после HTML-кода, формирующего все гиперссылки. Ведь, чтобы элемент страницы был доступен для сценария, он должен быть сначала сформирован.

Работу полосы навигации обеспечивают три функции — обработчика событий. Кратко рассмотрим их.

- `imgMouseOver` — обработчик события `onMouseOver`. Она меняет изначальное изображение на заменяющее.
- `imgMouseOut` — обработчик события `onMouseOut`. Она возвращает на место изначальное изображение.
- `imgClick` — обработчик события `onClick`. Она подставляет на место изначального изображения соответствующее заменяющее.

Эти функции — обработчики событий привязаны не к гиперссылкам (тегам `<A>`), а к изображениям (тегам `<IMG>`). Хотя можно было привязать их и к гиперссылкам, но тогда код несколько усложнится.

В качестве единственного параметра мы передаем этим трем функциям имя соответствующей гиперссылки. (Как мы выяснили в *главе 6*, функциям — обработчикам событий можно передавать любые параметры.) Это имя будет использовано в теле этих функций для идентификации изображения-гиперссылки, в которой наступило событие.

> ***НА ЗАМЕТКУ***
>
> Конечно, мы можем использовать для выяснения, в какой гиперссылке наступило событие, знакомые нам свойства объекта `Event`. Но тогда нам придется создавать обработчики событий отдельно для Internet Explorer и Opera и отдельно для Firefox.

В теле функций — обработчиков событий `onMouseOver` и `onMouseOut` не происходит ничего особо мудреного. Сначала получается доступ к гиперссылке, имя которой было передано в качестве параметра. Далее проверяется, не та ли это гиперссылка, что указывает на текущую страницу, для чего она срав-

нивается со значением переменной `currentAObj`. После этого получается доступ к "вложенному" в нее изображению (ее возвращает свойство `firstChild`, поскольку изображение — единственный потомок гиперссылки) и, в зависимости от результатов сравнения, о котором говорилось ранее, свойству `href` присваивается имя нужного файла, которое берется из соответствующего ассоциативного массива по индексу — имени гиперссылки.

А тело функции — обработчика события `onClick` даже описывать не стоит. Настолько там все просто.

Рассмотренный нами способ создания полосы навигации — не единственный из применяемых на практике. Способов сделать такой симпатичный и практичный элемент страницы много, но все они основаны на одном принципе — подмене изображений в ответ на возникновение событий мыши.

Предзагрузка графических изображений

Хорошо, полосу навигации для своего сайта мы сделали. И даже выложили сайт в Интернет — пусть народ полюбуется. Работа сделана!

Сделана, да не совсем. Мы не предусмотрели одну небольшую деталь, которая может испортить у посетителя все впечатление от нашей красивой полосы навигации. Давайте выясним, что это за деталь.

Предположим, на наш сайт зашел посетитель. Он загрузил страницу по умолчанию, дождался, пока загрузятся все изображения, составляющие полосу навигации, полюбовался ей и решил перейти на другую страницу сайта. Когда он навел курсор мыши на соответствующее изображение-гиперссылку, выполнился обработчик события, подменяющий изначальное изображение заменяющим. Web-обозреватель проверил, присутствует ли заменяющее изображение в его кэше, обнаружил, что не присутствует, и начал его загружать. Канал доступа в Интернет у посетителя небыстрый, загрузка идет не шатко не валко, а пока она продолжается, посетитель наблюдает вместо заменяющего изображения пустой прямоугольник. Малопривлекательное зрелище — кажется, что страница не доделана.

Как же быть? Может, есть способ указать Web-обозревателю загрузить все составляющие полосу навигации изображения сразу при загрузке страницы (выполнить *предзагрузку*)? Тогда все загруженные изображения попадут в кэш, и Web-обозревателю не придется лезть за ними в Интернет. И посетитель больше не будет наблюдать раздражающих пустых прямоугольников.

Да, есть такой способ. Для каждого из изображений, которые должны выводиться на страницу программно, из сценариев, создается экземпляр объекта

HTMLImageElement (для чего используется знакомый нам по *главе 4* оператор new), его свойству src присваивается интернет-адрес файла с нужным изображением и все. Все это выполняет особый загрузочный сценарий, помещающийся в секции заголовка страницы.

А теперь — внимание! Для Web-обозревателя объекта HTMLImageElement не существует. Существует объект Image, выполняющий те же функции. Для простоты будем считать, что "Image" — сокращенная форма "HTMLImageElement", а полную форму Web-обозреватель почему-то не "знает".

Далее приведен сценарий, выполняющий предзагрузку файла 1_up.jpg (этот файл используется в созданной нами ранее полосе навигации).

```
var preloadImageObj = new Image();
preloadImageObj.src = "1_up.jpg";
```

Как уже говорилось, его нужно поместить в секцию заголовка страницы до всех сценариев, выполняющих манипуляции с графическими изображениями.

Если нужно выполнить предзагрузку сразу нескольких файлов (а так чаще всего и бывает), можно повторить приведенный ранее код, меняя в нем только интернет-адрес файла. А можно поступить по-другому, создав более компактный сценарий. Например, такой, какой приведен далее.

```
var images = new Array("1_up.jpg", "2_up.jpg", "3_up.jpg",
"1_down.jpg", "2_down.jpg", "3_down.jpg",
"1_hover.jpg", "2_hover.jpg", "3_hover.jpg",
"1_hover_down.jpg", "2_hover_down.jpg", "3_hover_down.jpg");
var preload = [];
for (var i = 0; i < images.length; i++) {
  preload [i] = new Image();
  preload [i].src = images[i];
}
```

Здесь мы создаем два массива. Первый — images — содержит интернет-адреса всех файлов, для которых нужно выполнить предзагрузку. Второй — preload — изначально пуст, но впоследствии наполнится экземплярами объекта Image, соответствующими предзагружаемыми изображениями. После этого в цикле мы "проходим" по массиву images, для каждого его элемента создаем экземпляр объекта Image, помещаем его в массив preload и присваиваем его свойству src содержимое текущего элемента массива images. Все!

Этот сценарий можно вставить в написанные нами ранее страницы 9.1.htm, 9.2.htm и 9.3.htm. Только — это нужно помнить — он должен быть загрузочным и присутствовать в странице самым первым.

Внимание!

Предзагрузка изображений заметно замедляет загрузку страницы. Поэтому ее стоит применять только в крайних случаях, когда без нее не обойтись (например, если на странице присутствуют горячие изображения или полоса навигации).

Работа с картами-изображениями

О картах-изображениях мы узнали еще в *главе 2*. Вкратце — это изображение, разбитое на части (горячие области), каждая из которых представляет собой отдельную гиперссылку, указывающую на свой интернет-адрес и имеющую свою цель. Карты-изображения часто используются для создания красивых заголовков сайтов, работающих так же, как полоса навигации, и в специальных случаях (например, на странице присутствует карта области; при щелчке на одном из городов загружается страница, содержащая сведения об этом городе).

В этом параграфе мы продолжим разговор о картах-изображениях. Мы научимся управлять ими программно, из сценариев.

Давайте сначала вспомним, с помощью каких тегов формируется карта-изображение. Их три.

- Одинарный тег `<IMG>`, создающий само изображение. В необязательном атрибуте `USEMAP` этого тега в особом формате (а точнее, в формате гиперссылок, указывающих на якоря; подробнее см. *главу 2*) указывается имя карты и интернет-адрес страницы, где она определена (если карта создана на другой странице).
- Парный тег `<MAP>`, создающий карту — описание набора горячих областей. Обязательный в данном случае атрибут `NAME` задает имя карты, которое указывается в атрибуте `USEMAP` тега `<IMG>`. (Атрибут `ID` в этом случае использовать не стоит, а еще лучше указать имя карты и в атрибуте `NAME`, и в атрибуте `ID`.)
- Одинарный тег `<AREA>`, создающий горячую область. Набор этих тегов помещается в тег `<MAP>`. Параметры горячей области задает набор атрибутов тега `<AREA>`, рассмотренных в *главе 2*.

Изображение, как мы уже знаем, представляет объект `HTMLImageElement`. Карту представляет объект `HTMLMapElement`, а горячую область — объект `HTMLAreaElement`. Двумя последними объектами мы как раз и займемся.

Но сначала — еще немного об объекте `HTMLImageElement`. Он поддерживает свойство `useMap`, соответствующее атрибуту `USEMAP` тега `<IMG>` и задающее имя карты и интернет-адрес страницы, где она определена (если карта созда-

на на другой странице). Значение этого свойства должно быть задано в строковом виде.

Обратимся к объекту `HTMLMapElement`. Помимо свойства `name`, соответствующего атрибуту `NAME` и задающего в строковом виде имя карты, он поддерживает свойство `areas`. Это свойство возвращает экземпляр объекта `HTMLCollection` — коллекцию, содержащую все присутствующие в карте горячие области в виде экземпляров объекта `HTMLAreaElement`. Для доступа к нужному элементу этой коллекции мы можем использовать численные или строковые индексы; в качестве строковых индексов используются имена горячих областей, заданные атрибутами `ID` или `NAME`.

Что касается объекта `HTMLAreaElement`, то он поддерживает целых пять полезных для нас свойств. Все они перечислены в табл. 9.2.

Таблица 9.2. *Свойства объекта* `HTMLAreaElement`

Свойство	Описание
`coords`	Задает или возвращает местоположение и размеры горячей области в виде строки. Соответствует атрибуту `COORDS`
`href`	Задает или возвращает интернет-адрес целевого файла в виде строки. Соответствует атрибуту `HREF`
`nohref`	Если `true`, то при щелчке на горячей области ничего происходить не будет; если `false`, при щелчке будет выполнена загрузка целевого файла. Соответствует атрибуту `NOHREF`. Подробнее использование этого свойства будет описано при рассмотрении второго примера
`shape`	Задает или возвращает форму горячей области в виде строки. Соответствует атрибуту `SHAPE`
`target`	Задает или возвращает цель гиперссылки в виде строки. Соответствует атрибуту `TARGET`

Описания соответствующих атрибутов и их значений приведены в *главе 2*.

Мы можем программно создавать и удалять карты и горячие области, используя для этого знакомые нам по *главе 8* свойства и методы DOM. Каких-либо особых хитростей и "подводных камней" здесь нет — все стандартно.

> ***Внимание!***
>
> Один "подводный камень" таки есть. При создании новой карты (тег `<MAP>`) ее имя следует присвоить и свойству `name`, и свойству `id`. Иначе созданная программно карта-изображение не будет работать в Internet Explorer.

Осталось рассмотреть пару примеров. Первый будет совсем простым — мы возьмем карту-изображение, созданную в *главе 2*, и программно изменим форму одной из горячих областей.

```
<MAP NAME="simplemap">
  <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>
. . .
<SCRIPT>
  var simpleMapObj = document.getElementById("simplemap");
  var areaObj = simpleMapObj.areas[0];
  areaObj.shape = "rect";
  areaObj.coords = "50,50,100,100";
</SCRIPT>
```

Здесь мы меняем форму первой из присутствующих в карте `simplemap` горячей области на прямоугольную и задаем для нее новые координаты.

Второй пример будет посложнее. Мы создадим страницу с обычным изображением и сценарием, который будет формировать карту-изображение с двумя горячими областями в виде круга. Первая горячая область будет указывать на страницу page1.html, вторая — не указывать никуда.

```
<HTML>
  <HEAD>
    <TITLE>Карта-изображение</TITLE>
  </HEAD>
  <BODY>
    <P><IMG ID="mappedimg" SRC="map.jpg"></P>
    <SCRIPT>
      var mapObj = document.createElement("MAP");
      mapObj.name = "simplemap";
      mapObj.id = mapObj.name;
      var areaObj = document.createElement("AREA");
      areaObj.shape = "circle";
      areaObj.coords = "30,30,30";
      areaObj.href = "page1.html";
      mapObj.appendChild(areaObj);
      var areaObj = document.createElement("AREA");
```

```
      areaObj.shape = "circle";
      areaObj.coords = "90,90,30";
      areaObj.nohref = true;
      mapObj.appendChild(areaObj);
      document.body.appendChild(mapObj);
      var imgObj = document.getElementById("mappedimg");
      imgObj.useMap = "#" + mapObj.name;
    </SCRIPT>
  </BODY>
</HTML>
```

Здесь нужно отметить три момента.

Во-первых, когда мы задаем имя создаваемой карты, присваиваем его сразу свойствам `name` и `id`. Если мы присвоим его только свойству `name`, карта-изображение не будет работать в Internet Explorer. Впрочем, об этом уже было сказано.

Во-вторых, когда мы создаем первую горячую область, указывающую на страницу page1.html, то присваиваем интернет-адрес этой страницы свойству `href`. А когда мы создаем вторую горячую область, никуда не указывающую, то присваиваем свойству `nohref` значение `true`. Никакого значения свойству `href` в этом случае присваивать не нужно — оно все равно будет проигнорировано.

В-третьих, когда мы привязываем готовую карту к изображению, то присваиваем свойству `useMap` имя этой карты, обязательно предварив его символом `#`. Об этом также говорилось ранее.

Больше о работе с картами-изображениями рассказывать нечего. Как, собственно, и о работе с изображениями вообще.

Работа с мультимедийными данными

Настала пора познакомиться с мультимедийными данными и способами поместить их на страницу и управлять ими из сценариев. Этой темы мы еще нигде не касались.

Мультимедиа ("multimedia" — "многосредность") — так называют информацию, отличающуюся от простого текста и графики. В настоящее время так называют аудио- и видеофильмы, хранящиеся в различных форматах. Это, в частности, аудиофайлы MP3 и WMA, видеофильмы AVI, QuickTime, WMV и FLV, анимация в формате Shockwave/Flash и мн. др., наиболее часто встречающиеся на Web-страницах.

Мода на мультимедиа захлестнула мир еще в начале 90-х годов. Сейчас же мы наблюдаем настоящий бум: интернет-радио и интернет-телевидение, сервисы публикации видео наподобие Youtube.com, аудио- и видеоролики, встроенные прямо в страницы, на сайтах попроще. Музыка и кино пришли в Интернет и пока что не собираются уходить оттуда.

Мультимедиа, мультимедиа, мультимедиа… Имя ему — легион.

Рассказ о мультимедийных возможностях HTML и JavaScript мы начнем с теории. Не зная ее, мы вряд ли сможем приступить к практике.

Поддержка мультимедийных данных

Сразу же нужно сказать, что мультимедийные данные, помещаемые на страницу, — такие же внедренные элементы, как и графические изображения. Это значит, что они хранятся в отдельных файлах, а в HTML-коде страницы с помощью особых тегов записываются интернет-адреса этих файлов.

В отличие от Web-страниц и традиционной интернет-графики, мультимедийные данные не поддерживаются Web-обозревателем непосредственно. Дело в том, что форматов хранения мультимедийных данных так много, что просто невозможно создать программу, которая бы их все поддерживала. Проблема решается использованием дополнительных программ, работающих совместно с Web-обозревателям и "отвечающих" каждая за свой формат.

Такие дополнительные программы, расширяющие возможности Web-обозревателя, делятся на две разновидности. Различаются они только способом "встраивания" в Web-обозреватель и некоторыми другими техническими деталями. С точки зрения обычного интернетчика они абсолютно одинаковы.

Первая разновидность — самая старая. Это *модули расширения* (plugins) Web-обозревателя, появившиеся еще в середине 90-х годов. Впервые их начал поддерживать Netscape Navigator 2.0; в Internet Explorer поддержка их появилась в версии 3.0, а Opera и Firefox поддерживают их с самых первых версий. Все современные Web-обозреватели поддерживают модули расширения.

Модули расширения выполняются в виде динамических библиотек Windows (DLL). Они задействуются Web-обозревателем автоматически, когда он встретит в HTML-коде страницы ссылку на мультимедийный файл. Если нужный модуль расширения на компьютере не установлен, Web-обозреватель может сам загрузить и установить его.

Вторая разновидность дополнительных программ появилась ближе к концу 90-х годов и носит название *элементов ActiveX*. Они поддерживаются Internet Explorer, начиная с версии 3.0, и всеми версиями Opera; их поддержка в Firefox заявлена, но реально они не поддерживаются.

ВНИМАНИЕ!

В Opera элементы ActiveX поддерживаются как-то странно — то работают, то не работают. Так что лучше на поддержку элементов ActiveX в Opera не рассчитывать.

Элементы ActiveX также выполняются в виде динамических библиотек Windows. Они задействуются Web-обозревателем автоматически и также могут быть при необходимости им загружены и установлены.

Все современные Web-обозреватели поставляются вместе с модулями расширения и элементами ActiveX для поддержки многих мультимедийных форматов данных. (Хотя некоторые типы данных, например, Web-страницы, текстовые файлы и графические изображения GIF, JPEG и PNG, Web-обозреватели "понимают" сами.) В частности, все Web-обозреватели изначально включают в себя модули расширения и элементы ActiveX, обрабатывающие графику Shockwave/Flash, фильмы AVI и MPEG и звукозаписи WAV.

Модули расширения Web-обозревателя

Чтобы поместить на страницу мультимедийный элемент с использованием модуля расширения Web-обозревателя, используется парный тег `<EMBED>`. Формат его записи таков:

```
<EMBED SRC="<интернет-адрес файла с мультимедийными данными>"
↳[PLUGINSPAGE="<интернет-адрес Web-страницы с дистрибутивом модуля
↳расширения>"] [TYPE="<MIME-тип мультимедийных данных>"]
↳[<дополнительные параметры модуля расширения>]
</EMBED>
```

Самый важный атрибут этого тега — `SRC`. Он задает интернет-адрес файла, хранящего сами мультимедийные данные (аудио, видеоклип, изображение или фильм Shockwave/Flash), и является обязательным.

Необязательный атрибут `PLUGINSPAGE` задает интернет-адрес Web-страницы, где посетитель может найти дистрибутив модуля расширения. Это будет очень полезно, если на компьютере посетителя данный модуль расширения не установлен. В частности, для проигрывателя Shockwave/Flash значение этого атрибута будет таким:

http://www.macromedia.com/go/getflashplayer

Другой необязательный атрибут `TYPE` задает так называемый *тип MIME* (Multipurpose Internet Mail Extensions, многоцелевые расширения почты Интернета). Он задает тип данных, хранящихся в заданном файле: звукозапись WAV, MP3, видеоклип AVI, MPEG или QuickTime либо графика Shockwave/Flash. Фактически значение этого параметра указывает на программу,

воспроизводящую данный файл. Типы MIME для некоторых форматов мультимедийных файлов приведены в табл. 9.3.

Таблица 9.3. *Некоторые типы MIME и соответствующие им форматы мультимедийных данных*

Тип файлов	Тип MIME
Аудио- или видеозапись ASF	`video/x-ms-asf`
Аудио- или видеозапись WMV	`video/x-ms-wmv`
Аудиозапись AIFF	`audio/aiff`
Аудиозапись AU	`audio/basic`
Аудиозапись MIDI	`audio/mid`
Аудиозапись MP3	`audio/mpeg`
Аудиозапись WAV	`audio/wav`
Аудиозапись WMA	`audio/x-ms-wma`
Видеозапись AVI	`video/avi`
Видеозапись Indeo (IVF)	`video/x-ivf`
Видеозапись MPEG	`video/mpeg`
Графический файл ART	`image/x-jg`
Графический файл BMP	`image/bmp`
Графический файл GIF	`image/gif`
Графический файл JPEG	`image/jpeg`
Графический файл Shockwave/Flash	`application/x-shockwave-flash`
Графический файл TIFF	`image/tiff`

Если атрибут `TYPE` отсутствует, Web-обозреватель попытается выяснить формат мультимедийных данных, хранящихся в заданном файле, по расширению этого файла.

На заметку

Сведения о расширениях файлов и соответствующих им типам MIME и модулям расширения Web-обозревателя хранятся в Реестре Windows. При установке модуля расширения он сам запишет в Реестр сведения о себе, поддерживаемых им форматах данных и расширениях файлов, так что ошибки тут практически исключены.

Также в теге <EMBED> могут присутствовать атрибуты, задающие дополнительные параметры самого модуля расширения. Они задаются в виде пар *<имя атрибута, соответствующего параметру>*="*<значение параметра>*" и будут рассмотрены нами позже.

Модуль расширения представляется Web-обозревателем как экземпляр объекта HTMLObjectElement. Он порожден от объекта HTMLElement, а значит, поддерживает все его свойства и методы. Кроме того, он поддерживает специфические свойства, о которых будет отдельный разговор.

Свойство src соответствует рассмотренному ранее одноименному атрибуту тега <EMBED> и задает или возвращает интернет-адрес файла мультимедийных данных в виде строки. Оно поддерживается только Internet Explorer и Opera.

Свойство type соответствует рассмотренному ранее одноименному атрибуту тега <EMBED> и задает или возвращает тип MIME мультимедийных данных в виде строки. Оно также поддерживается только Internet Explorer и Opera.

Свойство pluginspage самое мудреное. Понятно, что оно соответствует рассмотренному ранее одноименному атрибуту тега <EMBED> и задает или возвращает интернет-адрес страницы с дистрибутивом модуля расширения в виде строки. Но полноценно оно поддерживается только Firefox; Internet Explorer поддерживает его только для чтения, а Opera его вообще не поддерживает.

К дополнительным параметрам, заданным для модуля расширения, мы никак не сможем добраться из сценариев. По крайней мере, у автора не получилось это сделать.

Кроме того, для объекта HTMLObjectElement заявлена поддержка всех событий, поддерживаемых объектом HTMLLinkElement и перечисленных в табл. 9.1. Но, похоже, ни один Web-обозреватель этого не делает.

Как видим, тег <EMBED> и соответствующий ему объект HTMLObjectElement очень плохо подходят для программирования. Так что лучше создать модуль расширения в HTML-коде или из сценария, с помощью методов DOM, и больше его не трогать.

В качестве примера давайте напишем небольшую страницу, содержащую рекламный баннер в формате Shockwave/Flash. Небольшой баннер из тех, что сейчас заполонили Интернет.

```
<HTML>
  <HEAD>
    <TITLE>Модуль расширения</TITLE>
  </HEAD>
  <BODY>
```

```
    <EMBED SRC="banner.swf" TYPE="application/x-shockwave-flash">
    </EMBED>
  </BODY>
</HTML>
```

Здесь мы явно указали тип MIME-файла с мультимедийными данными (он у нас имеет имя banner.swf) — "application/x-shockwave-flash". Хотя этого можно и не делать — Web-обозреватель сам выяснит, какой модуль расширения использовать, по расширению файла данных.

Элементы ActiveX

Чтобы поместить на страницу мультимедийные данные с использованием элемента ActiveX, нам понадобятся уже два тега. Первый — парный тег `<OBJECT>`, служащий собственно для помещения на Web-страницу элемента ActiveX, с помощью которого посетитель будет просматривать или прослушивать наши мультимедийные данные. Второй — одинарный тег `<PARAM>`, задающий дополнительные параметры для данного элемента ActiveX (интернет-адрес файла с данными, необходимость наличия интерфейса управления и пр.).

Формат тега `<OBJECT>` таков:

```
<OBJECT CLASSID="<GUID элемента ActiveX>"
[CODEBASE="<интернет-адрес дистрибутива элемента ActiveX>"]
[CODETYPE="<MIME-тип мультимедийных данных>"]>
  <Теги <PARAM>, задающие дополнительные параметры элемента ActiveX>
</OBJECT>
```

Атрибут `CLASSID` очень важен. Он задает так называемый *GUID* (Global Unique IDentifier, глобальный уникальный идентификатор) элемента ActiveX, однозначно его идентифицирующий. С помощью GUID мы даем Web-обозревателю понять, какой элемент ActiveX хотим использовать для создания мультимедийного элемента.

Но как узнать нужный GUID?! Очень просто. GUID всех установленных в системе элементов ActiveX записывается в "ветви"

```
HKEY_CLASSES_ROOT/CLSID
```

системного Реестра. Web-обозреватель, встретив в HTML-коде страницы GUID, отыскивает его в указанной "ветви" Реестра, выясняет, какой элемент ActiveX нужно использовать, загружает его и запускает на выполнение.

> ***НА ЗАМЕТКУ***
>
> Чтобы найти в указанной "ветви" системного Реестра нужный GUID, лучше всего воспользоваться встроенной в программу Редактор реестра возможностью поиска.

Так, стандартно поставляемый в составе Windows проигрыватель Window Media (в смысле, соответствующий ему элемент ActiveX) имеет вот такой GUID:

```
22D6F312-B0F6-11D0-94AB-0080C74C7E95
```

Что касается проигрывателя Shockwave/Flash, то ему соответствует вот такой GUID:

```
d27cdb6e-ae6d-11cf-96b8-444553540000
```

А теперь — внимание! GUID в коде HTML должен быть записан таким образом:

```
clsid:<нужный нам GUID>
```

Символы "clsid:" в начале являются обязательными. Пробелы между двоеточием и самим GUID не допускаются.

Атрибут `CODEBASE` тега `<OBJECT>` является необязательным и задает интернет-адрес дистрибутива данного элемента ActiveX. Если данный элемент ActiveX не установлен на компьютере клиента, сам Web-обозреватель загрузит его дистрибутив, пользуясь указанным интернет-адресом, и сам же его установит.

Для проигрывателя Windows Media атрибут `CODEBASE` указывать не нужно — эта программа и так стандартно поставляется в составе Windows, начиная с версии 98. А вот для проигрывателя Shockwave/Flash его лучше указать; его значение должно быть таким:

http://fpdownload.macromedia.com/pub/shockwave/cabs/flash/swflash.cab#

↳version=9,0,0,0

Вдруг для просмотра какого-либо изображения понадобится более новая версия этого проигрывателя!

Необязательный атрибут `CODETYPE` задает тип MIME мультимедийных данных, хранящихся в указанном файле. Типы MIME для некоторых форматов мультимедийных файлов были приведены в табл. 9.3.

Пожалуй, о теге `<OBJECT>` больше рассказывать нечего. Обратимся к тегу `<PARAM>`.

Одинарный тег `<PARAM>` может присутствовать только внутри тега `<OBJECT>`. Он служит для задания дополнительных параметров модуля расширения. Вот формат его записи:

```
<PARAM NAME="<имя параметра>" VALUE="<значение параметра>">
```

Думается, комментировать здесь нечего.

По крайней мере, один тег `<PARAM>` всегда присутствует в теге `<OBJECT>` — он задает интернет-адрес файла с мультимедийными данными. В случае проиг-

рывателя Windows Media за это "отвечает" параметр `FileName`; проигрыватель Shockwave/Flash для этого использует параметр `movie`. В качестве значения обоих этих параметров указывается интернет-адрес файла данных.

> ***Внимание!***
>
> Обратим внимание — в теге `<EMBED>` для задания интернет-адреса файла с данными используется атрибут `SRC`, а в теге `<OBJECT>` его нет. В теге `<OBJECT>` файл с данными (как и все остальные дополнительные параметры) указывается с помощью тега `<PARAM>`.

Элемент ActiveX представляется Web-обозревателем как экземпляр объекта `HTMLObjectElement` — как и модуль расширения. Только в случае элемента ActiveX специфические свойства этого объекта будут другими. Давайте о них поговорим.

Итак, прежде всего, это свойство `classid`. Оно соответствует атрибуту `CLASSID`, задает или возвращает GUID элемента ActiveX в виде строки и поддерживается только Internet Explorer.

Свойство `codeBase` соответствует атрибуту `CODEBASE`, задает или возвращает интернет-адрес дистрибутива данного элемента ActiveX в виде строки и поддерживается и Internet Explorer, и Opera.

Свойство `codeType` соответствует атрибуту `CODETYPE`, задает или возвращает тип MIME мультимедийных данных в виде строки и поддерживается и Internet Explorer, и Opera. Причем, если элемент ActiveX создан в коде HTML страницы, это свойство доступно только для чтения; если же элемент ActiveX был создан программно, в сценарии, это свойство будет доступно и для чтения, и для записи.

А теперь — внимание! Объект `HTMLObjectElement` представляет прямой доступ к свойствам и методам элемента ActiveX, который он создает. Так, мы можем программно загрузить в элемент ActiveX новый файл Shockwave/Flash, присвоив его интернет-адрес свойству `movie`. Это свойство соответствует одноименному дополнительному параметру, устанавливаемому с помощью тега `<PARAM>` и задающему интернет-адрес файла с данными.

```
<OBJECT ID="o" CLASSID="clsid:d27cdb6e-ae6d-11cf-96b8-444553540000"
  CODEBASE="http://fpdownload.macromedia.com/pub/shockwave/cabs/
  ↳flash/swflash.cab#version=9,0,0,0">
  <PARAM NAME="movie" VALUE="banner.swf">
</OBJECT>
. . .
<SCRIPT>
```

```
  var oObj = document.getElementById("o");
  oObj.movie = "otherbanner.swf";
</SCRIPT>
```

К сожалению, такие штуки мы можем проделывать только в Internet Explorer. Opera не поддерживает управление элементами ActiveX из сценариев.

Может случиться так, что какой-то элемент ActiveX поддерживает свойство, чье имя совпадает с именем свойства объекта `HTMLObjectElement`. Скажем, какой-то элемент ActiveX обзавелся свойством `id`, до которого нам очень нужно добраться. В этом случае мы можем явно указать Web-обозревателю, что нам нужно именно свойство элемента ActiveX, а не объекта `HTMLObjectElement`, воспользовавшись свойством `object`. Это свойство объекта `HTMLObjectElement` возвращает ссылку на экземпляр объекта, представляющего элемент ActiveX. К сожалению, оно поддерживается только Internet Explorer (Opera, как мы уже выяснили, вообще не позволяет управлять элементом ActiveX из сценариев).

```
<OBJECT ID="o" CLASSID="<GUID элемента ActiveX>"
  . . .
</OBJECT>
. . .
<SCRIPT>
  var oObj = document.getElementById("o");
  oObj.object.id = <значение свойства id>;
</SCRIPT>
```

Также объект `HTMLObjectElement` в Internet Explorer поддерживает знакомое нам по объекту `HTMLImageElement` свойство `readyState`. Оно возвращает состояние элемента ActiveX, но уже в виде одного из перечисленных далее числовых значений:

- 0 — файл с данными еще не загружен;
- 1 — файл с данными загружается;
- 2 — файл с данными загружен, но данные еще не выведены на страницу;
- 3 — загруженные данные еще не выведены на страницу, но элементом ActiveX уже можно управлять из сценариев;
- 4 — данные выведены на страницу.

Только пользы от этого свойства немного, поскольку, насколько удалось выяснить автору, объект `HTMLObjectElement` не поддерживает событие `onReadyStateChange`, знакомое нам по объекту `HTMLImageElement`. Он поддерживает только события объекта `HTMLElement`, изученные нами в *главе 8*.

Давайте для примера напишем страницу, которая будет выводить рекламный баннер Shockwave/Flash, но с помощью элемента ActiveX.

```
<HTML>
  <HEAD>
    <TITLE>Элемент ActiveX</TITLE>
  </HEAD>
  <BODY>
    <OBJECT CLASSID="clsid:d27cdb6e-ae6d-11cf-96b8-444553540000"
    CODEBASE="http://fpdownload.macromedia.com/pub/shockwave/cabs/
    ↳flash/swflash.cab#version=9,0,0,0"
    CODETYPE="application/x-shockwave-flash">
      <PARAM NAME="movie" VALUE="banner.swf">
    </OBJECT>
  </BODY>
</HTML>
```

Здесь мы также явно указали тип MIME-файла с мультимедийными данными (banner.swf) — "application/x-shockwave-flash". Но этого можно и не делать.

Компромиссное решение: модель расширения + элемент ActiveX

Итак, что мы имеем? Во-первых, модули расширения, которые совместимы со всеми Web-обозревателями, но не предлагают никаких возможностей по программному управлению. Во-вторых, элементы ActiveX, поддерживаемые только Internet Explorer и Opera (но на Opera в этом случае, как было сказано ранее, рассчитывать не стоит), но весьма "благосклонные" к Web-программистам. Как говорится, или — или…

Но почему обязательно — или? Можно ведь сказать — и. Для этого следует использовать компромиссный вариант — объединить теги `<EMBED>` (создающий модуль расширения) и `<OBJECT>` (создающий элемент ActiveX), вложив первый внутрь второго. В случае нашего рекламного баннера это будет выглядеть так (необязательные атрибуты опущены):

```
<OBJECT CLASSID="clsid:d27cdb6e-ae6d-11cf-96b8-444553540000">
  <PARAM NAME="movie" VALUE="banner.swf">
  <EMBED SRC="banner.swf">
  </EMBED>
</OBJECT>
```

Предположим, этот код загружает Web-обозреватель, поддерживающий элементы ActiveX. Он прочитывает тег <OBJECT> и его атрибуты, загружает библиотеку, реализующую данный элемент ActiveX, и передает ей заданные в тегах <PARAM> дополнительные параметры. Тег <EMBED> в таком случае Web-обозревателю не нужен, и он его игнорирует.

Теперь "скормим" приведенный ранее код Web-обозревателю, который не поддерживает элементы ActiveX. Он читает тег <OBJECT> и, поскольку не "знает" его, пропускает и принимается за тег <EMBED>. Далее последовательность событий такая же, что описана ранее: загрузка файла с мультимедийными данными, определение и загрузка библиотеки, реализующей нужный модуль, и передача ей заданных в атрибутах тега <EMBED> параметров.

Кстати, стандарты HTML предписывают всем Web-обозревателям игнорировать все теги, что они не поддерживают (так называемые *неизвестные теги*). То же самое и с неподдерживаемыми атрибутами тегов и стилей — они должны быть проигнорированы. Эта особенность часто используется Web-программистами, в чем, собственно, мы уже убедились.

Разумеется, управлять модулем расширения из сценариев мы и в этом случае не сможем. Но зато сможем управлять элементом ActiveX!

Дополнительные параметры

О дополнительных параметрах модулей расширения и элементов ActiveX мы уже знаем. Они задают сведения о мультимедийных данных, режимах их воспроизведения, положения на странице и пр., не являющиеся жизненно необходимыми, но в некоторых случаях важными. Давайте рассмотрим их.

Прежде всего вспомним, как задаются дополнительные параметры.

- Дополнительные параметры модуля расширения задаются атрибутами тега <EMBED>.
- Дополнительные параметры элемента ActiveX задаются в тегах <PARAM> — по одному такому тегу на каждый параметр, — которые вкладываются в тег <OBJECT>.

Что касается самих дополнительных параметров, то каждая модель расширения или элемент ActiveX поддерживает свой набор. Поэтому давайте рассмотрим только некоторые параметры проигрывателей Windows Media и Shockwave/Flash, которые могут оказаться полезными для нас на первых порах. Дополнительные параметры проигрывателей перечислены в табл. 9.4 и 9.5.

Таблица 9.4. *Некоторые дополнительные параметры проигрывателя Windows Media*

Параметр	Описание
AutoRewind	Если "1", воспроизводимый мультимедийный файл будет автоматически перемотан к началу после окончания воспроизведения. Если "0", файл не будет перематываться. Значение по умолчанию — "1"
AutoStart	Если "1", воспроизведение мультимедийного файла будет запущено сразу же после его загрузки. Если "0", то посетитель сам должен запустить его воспроизведение. Значение по умолчанию — "1"
Balance	Задает величину стереобаланса. Принимает значения от –10 000 до 10 000. Значение по умолчанию — 0
EnableContextMenu	Если "1", после щелчка правой кнопкой мыши на проигрывателе появится контекстное меню, если "0", контекстное меню появляться не будет. Значение по умолчанию — "1"
Enabled	Если "1", посетитель может управлять проигрывателем с помощью кнопок, бегунка и контекстного меню, если "0", то не может. Значение по умолчанию — "1"
EnablePositionControls	Если "1", будут доступны кнопки перемотки, если "0", то не будут доступны. Значение по умолчанию — "1"
EnableTrackbar	Если "1", будет доступен бегунок, если "0", то не будет доступен. Значение по умолчанию — "1"
FileName	Задает интернет-адрес мультимедийного файла. Используется только в тегах <PARAM>; в теге <EMBED> следует применять атрибут SRC
Mute	Если "1", звук будет приглушен, если "0", звук будет воспроизводиться с изначальной громкостью. Значение по умолчанию — "0"
PlayCount	Задает количество повторов воспроизведения мультимедийного файла. Если 0, файл будет воспроизводиться бесконечно. Значение по умолчанию — 1 (т. е. однократное воспроизведение файла)

Таблица 9.4 *(продолжение)*

Параметр	Описание
`Rate`	Задает относительную частоту кадров воспроизводимого мультимедийного файла в виде множителя, на который нужно умножить изначальную частоту кадров. Например, если задать значение .5, то файл будет воспроизводиться с замедленной в два раза скоростью. Значение по умолчанию — 1.0
`SendErrorEvents`	Если "1", проигрыватель будет генерировать события `Error` (события проигрывателя Windows Media будут описаны далее), если "0", не будет генерировать. Значение по умолчанию — "1"
`SendKeyboardEvents`	Если "1", проигрыватель будет генерировать события клавиатуры, если "0", не будет генерировать. Значение по умолчанию — "0"
`SendMouseClickEvents`	Если "1", проигрыватель будет генерировать события `Click`, `DblClick`, `MouseDown` и `MouseUp`, если "0", не будет генерировать. Значение по умолчанию — "0"
`SendMouseMoveEvents`	Если "1", проигрыватель будет генерировать события `MouseMove`, если "0", не будет генерировать. Значение по умолчанию — "0"
`SendOpenStateChangeEvents`	Если "1", проигрыватель будет генерировать события `OpenStateChange`, если "0", не будет генерировать. Значение по умолчанию — "1"
`SendPlayStateChangeEvents`	Если "1", проигрыватель будет генерировать события `PlayStateChange`, если "0", не будет генерировать. Значение по умолчанию — "1"
`SendWarningEvents`	Если "1", проигрыватель будет генерировать события `Warning`, если "0", не будет генерировать. Значение по умолчанию — "1"
`ShowAudioControls`	Если "1", проигрыватель выведет на экран органы управления звуковым сопровождением, если "0", не выведет. Значение по умолчанию — "1"
`ShowControls`	Если "1", проигрыватель выведет на экран кнопки запуска, приостановки и остановки воспроизведения, если "0", не выведет. Значение по умолчанию — "1"

Таблица 9.4 *(окончание)*

Параметр	Описание
`ShowDisplay`	Если "1", проигрыватель выведет на экран область просмотра видео, если "0", не выведет. Значение по умолчанию — "0", если воспроизводится аудиоклип, и "1", если воспроизводится видеоклип
`ShowPositionControls`	Если "1", проигрыватель выведет на экран кнопки перемотки, если "0", то недоступны. Значение по умолчанию — "1"
`ShowStatusBar`	Если "1", проигрыватель выведет на экран строку статуса, если "0", не выведет. Значение по умолчанию — "0"
`ShowTracker`	Если "1", проигрыватель выведет на экран бегунок, если "0", не выведет. Значение по умолчанию — "1"
`TransparentAtStart`	Если "1", проигрыватель будет прозрачным перед началом и после окончания воспроизведения мультимедийного файла, если "0", не будет. Значение по умолчанию — "0"
`Volume`	Задает громкость звука в диапазоне от –10 000 (минимальная) до 0 (максимальная). Значение по умолчанию — –600
`WindowlessVideo`	Если "1", видео будет выводиться прямо на Web-странице (безоконный режим), если "0", видео будет выводиться в отдельном "окне", располагаемом на этой странице. Безоконный режим следует включить, если к воспроизводимому видео предполагается применять какие-либо эффекты. Значение по умолчанию — "0"

Таблица 9.5. *Некоторые дополнительные параметры проигрывателя Shockwave/Flash*

Параметр	Описание
`align`	Задает выравнивание "окна" проигрывателя на Web-странице. Доступны значения: "Default" (выравнивание по центру страницы), "L" (по левой границе), "T" (по верхней), "R" (по правой) и "B" (по нижней)
`loop`	Если "true", фильм будет зациклен, если "false", не будет. Значение по умолчанию — "true"

Таблица 9.5 *(окончание)*

Параметр	Описание
`menu`	Если "true", при щелчке мышью на проигрывателе будет выведено полное контекстное меню, если "false", это меню будет содержать только пункт **About Flash Player**, выводящий диалоговое окно сведений о проигрывателе. Значение по умолчанию — "true"
`movie`	Задает интернет-адрес файла с изображением или фильмом Shockwave/Flash. Используется только в тегах `<PARAM>`; в теге `<EMBED>` следует применять атрибут `SRC`
`play`	Если "true", воспроизведение фильма будет начато сразу же после его загрузки, если "false", посетитель сам должен запустить его воспроизведение. Значение по умолчанию — "true"
`quality`	Задает качество воспроизведения графики. Может принимать значения: "low" (самое низкое качество), "autolow", "autohigh", "medium", "high" и "best" (самое высокое качество). Значение по умолчанию — "high"
`salign`	Задает выравнивание изображения в "окне" проигрывателя. Доступны значения: "L" (по левой границе "окна"), "T" (по верхней), "R" (по правой), "B" (по нижней), "TL" (по верхней и левой), "TR" (по верхней и правой), "BL" (по нижней и левой) и "BR" (по нижней и правой). Если не задан, изображение располагается в центре "окна" проигрывателя
`scale`	Задает параметры масштабирования изображения в "окне" проигрывателя. Доступны значения: "showall" (масштабирование с сохранением пропорций, но могут появиться пустые области), "noborder" (то же самое, но без пустых областей — проигрыватель обрежет изображение, чтобы не допустить их появления) и "exactfit" (масштабирование без сохранения пропорций). Значение по умолчанию — "showall"
`wmode`	Задает параметры вывода изображения. Доступны значения "Window" (вывод изображения в собственном "окне" проигрывателя, которое будет располагаться на Web-странице), "Opaque" (вывод изображения прямо на странице, причем элементы страницы, располагающиеся под изображением, не будут видны) и "Transparent" (то же самое, но элементы страницы, располагающиеся под изображением, будут видны). Значение по умолчанию — "Window"

На заметку

Полный список параметров проигрывателя Windows Media можно узнать на посвященной ему странице сайта MSDN (**http://msdn.microsoft.com/library/en-us/wmp6sdk/htm/microsoftwindowsmediaplayercontrolversion64sdk.asp**). Полный список параметров проигрывателя Shockwave/Flash можно найти в интерактивной справке пакета Adobe Flash или на сайте фирмы Adobe (**http://www.adobe.com**) в разделе, посвященном Flash.

Осталось привести пару примеров применения дополнительных атрибутов на практике. Первый пример — страница, содержащая бесконечно воспроизводящийся фоновый звук, который мы поместим с помощью элемента ActiveX.

```
<HTML>
  <HEAD>
    <TITLE>Фоновая музыка</TITLE>
  </HEAD>
  <BODY>
    <OBJECT CLASSID="clsid:22D6F312-B0F6-11D0-94AB-0080C74C7E95">
      <PARAM NAME="FileName" VALUE="sound.wav">
      <PARAM NAME="PlayCount" VALUE="0">
      <PARAM NAME="ShowAudioControls" VALUE="0">
      <PARAM NAME="ShowControls" VALUE="0">
      <PARAM NAME="ShowDisplay" VALUE="0">
      <PARAM NAME="ShowPositionControls" VALUE="0">
      <PARAM NAME="ShowStatusBar" VALUE="0">
      <PARAM NAME="ShowTracker" VALUE="0">
    </OBJECT>
  </BODY>
</HTML>
```

Заметим, какие параметры мы задали для проигрывателя Windows Media. Во-первых, мы задали бесконечное повторение воспроизведения звукового клипа (значение "0" параметра `PlayCount`). Во-вторых, мы скрыли все элементы управления проигрывателя (значения "0" для параметров `ShowAudioControls`, `ShowControls`, `ShowDisplay`, `ShowPositionControls`, `ShowStatusBar` и `ShowTracker`). После этого невидимый проигрыватель будет постоянно играть один и тот же звук, действуя посетителям на нервы...

На заметку

Кстати, сказано верно. Ничто так не действует на нервы, как бесконечно пиликающая фоновая музыка. Так что используем приведенный ранее код только как пример, а не как руководство к действию.

Второй пример — страница с баннером Shockwave/Flash, растянутым по горизонтали. (Разумеется, мы растянули его только для примера.) Этот баннер мы поместим с помощью модуля расширения.

```
<HTML>
  <HEAD>
    <TITLE>Растянутый баннер</TITLE>
  </HEAD>
  <BODY>
    <EMBED SRC="banner.swf" SCALE="exactfit" STYLE="width: 400px">
    </EMBED>
  </BODY>
</HTML>
```

Здесь мы задали для "окна" проигрывателя Shockwave/Flash ширину, равную 400 пикселам, с помощью атрибута стиля `width`. После этого мы задали режим масштабирования изображения Shockwave/Flash без сохранения пропорций (значение "exactfit" параметра `scale` — он задается с помощью одноименного атрибута тега `<EMBED>`).

Что касается других модулей расширения и элементов ActiveX, не описанных в этой главе, то за сведениями о них нужно обращаться на сайт их разработчика. Там (по идее) должно быть представлено все, что нужно уважающему себя Web-программисту.

Управление элементами ActiveX из сценариев

Сказавший "а" да скажет "б"! Давайте рассмотрим возможности, предлагаемые некоторыми элементами ActiveX, в частности, проигрывателями Windows Media и Shockwave/Flash, по управлению ими из сценариев. Пригодится в дальнейшем.

Как уже говорилось ранее, модули расширения не поддерживают возможность доступа к ним из сценариев. К сожалению…

Управление проигрывателем Windows Media

Начнем с проигрывателя Windows Media. Он поддерживает множество полезных для нас свойств, методов и даже событий. Давайте их рассмотрим.

Свойства проигрывателя Windows Media весьма многочисленны. С большинством из них мы уже знакомы — см. табл. 9.4. Да, к перечисленным там параметрам мы можем обращаться и как к свойствам. Единственное — вместо значений "1" и "0" следует использовать логические значения `true` и `false` соответственно.

Другие свойства, поддерживаемые проигрывателем Windows Media, перечислены в табл. 9.6, методы — в табл. 9.7, события — в табл. 9.8.

Таблица 9.6. *Некоторые свойства, поддерживаемые проигрывателем Windows Media*

Свойство	Описание
`Bandwidth`	Возвращает скорость загрузки мультимедийного файла в битах в секунду в числовом виде
`BufferingCount`	Возвращает в числовом виде, сколько раз во время воспроизведения мультимедийного файла выполнялась его *буферизация* (предварительная загрузка в особую область памяти, называемую *буфером*)
`BufferingProgress`	Возвращает процент заполнения буфера в числовом виде
`BufferingTime`	Возвращает длительность в секундах фрагмента мультимедийного файла, загруженного в буфер. Значение возвращается в числовом виде
`CurrentPosition`	Задает текущую позицию бегунка в секундах в числовом виде
`Duration`	Возвращает продолжительность мультимедийного файла в секундах в числовом виде
`ErrorCode`	Возвращает в числовом виде код возникшей ошибки
`ErrorDescription`	Возвращает в строковом виде описание возникшей ошибки
`HasError`	Возвращает `true`, если в процессе загрузки или воспроизведения мультимедийного файла возникла ошибка, и `false` в противном случае
`ImageSourceHeight`	Возвращает высоту изображения видеоклипа в пикселах в числовом виде
`ImageSourceWidth`	Возвращает ширину изображения видеоклипа в пикселах в числовом виде
`IsBroadcast`	Возвращает `true`, если выполняется прием потокового аудио или видео, и `false`, если выполняется воспроизведение обычного мультимедийного файла
`IsDurationValid`	Возвращает `true`, если проигрыватель может правильно определить продолжительность аудио- или видеоклипа, и `false` в противном случае. Это свойство часто бывает нужно, т. к. при приеме потокового аудио или видео проигрыватель не всегда может определить его продолжительность

Таблица 9.6 *(окончание)*

Свойство	Описание
`LostPackets`	Возвращает количество потерянных *пакетов* (отдельных фрагментов данных) потокового аудио или видео в числовом виде
`OpenState`	Возвращает состояние открытия мультимедийного файла. Возвращаемые свойством значения: 0 (файл не открыт), 1 (идет загрузка файла описания потокового аудио или видео), 2 (идет загрузка файла описания интернет-радиостанции), 3 (поиск сервера), 4 (соединение с сервером), 5 (открытие файла) и 6 (файл открыт)
`PlayState`	Возвращает состояние воспроизведения мультимедийного файла. Возвращаемые свойством значения: 0 (воспроизведение остановлено), 1 (воспроизведение приостановлено), 2 (файл воспроизводится), 3 (идет буферизация потока), 4 (идет прокрутка вперед), 5 (идет прокрутка назад), 6 (выполняется переход на следующую позицию), 7 (выполняется переход на предыдущую позицию) и 8 (файл не был открыт)
`ReadyState`	Возвращает состояние готовности проигрывателя к воспроизведению мультимедийного файла. Возвращаемые свойством значения: 0 (файл не был задан, т. е. свойству `FileName` не был присвоен его интернет-адрес), 1 (идет загрузка файла), 3 (загружен не весь файл, но воспроизведение можно начинать), 4 (файл загружен полностью)
`ReceivedPackets`	Возвращает количество принятых пакетов потокового аудио или видео в числовом виде
`ReceptionQuality`	Возвращает процент принятых пакетов от общего количества пакетов потокового аудио или видео, пришедших за последние 30 секунд. Значение возвращается в числовом виде
`RecoveredPackets`	Возвращает количество восстановленных пакетов потокового аудио или видео в числовом виде

Таблица 9.7. *Некоторые методы, поддерживаемые проигрывателем Windows Media*

Метод	Описание
`FastForward()`	Выполняет быструю прокрутку воспроизводимого файла вперед
`FastReverse()`	Выполняет быструю прокрутку воспроизводимого файла назад

Таблица 9.7 *(окончание)*

Метод	Описание
`Open(<интернет-адрес файла>)`	Открывает мультимедийный файл, интернет-адрес которого был передан в качестве единственного параметра
`Pause()`	Приостанавливает воспроизведение мультимедийного файла
`Play()`	Запускает воспроизведение мультимедийного файла
`Stop()`	Останавливает воспроизведение мультимедийного файла

Таблица 9.8. *Некоторые события, поддерживаемые проигрывателем Windows Media*

Событие	Описание
`Buffering`	Возникает при начале или завершении буферизации данных. Обработчик принимает один параметр — значение `true`, если буферизация данных началась, и `false`, если она закончилась
`Click`	Возникает при щелчке мышью на проигрывателе. Обработчик принимает следующие параметры: обозначение нажатой кнопки мыши — сумма чисел 1 (левая кнопка), 2 (правая кнопка) и 4 (средняя кнопка); обозначение нажатой клавиши-модификатора — сумма чисел 1 (была нажата клавиша <Shift>), 2 (клавиша <Ctrl>) и 4 (клавиша <Alt); горизонтальная координата точки, в которой был выполнен щелчок мышью, и ее вертикальная координата (обе координаты отсчитываются относительно "окна" проигрывателя). Все параметры передаются в числовом виде. Это событие генерируется только в том случае, если свойству (параметру) `SendMouseClickEvents` проигрывателя присвоено значение `true` ("1")
`DblClick`	Возникает при двойном щелчке мышью на проигрывателе. Обработчик принимает следующие параметры: обозначение нажатой кнопки мыши — сумма чисел 1 (левая кнопка), 2 (правая кнопка) и 4 (средняя кнопка); обозначение нажатой клавиши-модификатора — сумма чисел 1 (была нажата клавиша <Shift>), 2 (клавиша <Ctrl>) и 4 (клавиша <Alt); горизонтальная координата точки, в которой был выполнен двойной щелчок мышью, и ее вертикальная координата (обе координаты отсчитываются относительно "окна" проигрывателя). Все параметры передаются в числовом виде. Это событие генерируется только в том случае, если свойству (параметру) `SendMouseClickEvents` проигрывателя присвоено значение `true` ("1")

Таблица 9.8 *(продолжение)*

Событие	Описание
`Disconnect`	Возникает при разрыве соединения с сервером. Обработчик принимает один параметр — числовое значение, обозначающее причину разрыва
`EndOfStream`	Возникает при окончании воспроизведения мультимедийного файла. Обработчик принимает один параметр — числовое значение, обозначающее состояние файла; если передано значение 0, то файл был воспроизведен без проблем
`Error`	Возникает, если проигрыватель столкнулся с проблемой при попытке воспроизвести мультимедийный файл
`KeyDown`	Возникает при нажатии клавиши, если проигрыватель был активен (имел фокус ввода). Обработчик принимает следующие параметры: код нажатой клавиши; обозначение нажатой клавиши-модификатора — сумма чисел 1 (была нажата клавиша <Shift>), 2 (клавиша <Ctrl>) и 4 (клавиша <Alt). Это событие генерируется только в том случае, если свойству (параметру) `SendKeyboardEvents` проигрывателя присвоено значение `true` ("1")
`KeyPress`	Возникает при нажатии и отпускании клавиши, если проигрыватель был активен (имел фокус ввода). Обработчик принимает единственный параметр — код ASCII нажатой клавиши. Это событие генерируется только в том случае, если свойству (параметру) `SendKeyboardEvents` проигрывателя присвоено значение `true` ("1")
`KeyUp`	Возникает при отпускании нажатой ранее клавиши, если проигрыватель был активен (имел фокус ввода). Обработчик принимает следующие параметры: код нажатой клавиши; обозначение нажатой клавиши-модификатора — сумма чисел 1 (была нажата клавиша <Shift>), 2 (клавиша <Ctrl>) и 4 (клавиша <Alt). Это событие генерируется только в том случае, если свойству (параметру) `SendKeyboardEvents` проигрывателя присвоено значение `true` ("1")
`MouseDown`	Возникает при нажатии кнопки мыши, если ее курсор находится над проигрывателем. Обработчик принимает следующие параметры: обозначение нажатой кнопки мыши — сумма чисел 1 (левая кнопка), 2 (правая кнопка) и 4 (средняя кнопка); обозначение нажатой клавиши-модификатора — сумма чисел 1 (была нажата клавиша <Shift>), 2 (клавиша <Ctrl>) и 4 (клавиша <Alt); горизонтальная координата точки, в которой находился курсор мыши, и ее вертикальная координата (обе координаты отсчитываются относительно "окна" проигрывателя). Все параметры передаются в числовом виде. Это событие генерируется только в том случае, если свойству (параметру) `SendMouseClickEvents` проигрывателя присвоено значение `true` ("1")

Таблица 9.8 *(продолжение)*

Событие	Описание
`MouseMove`	Возникает при перемещении курсора мыши над проигрывателем. Обработчик принимает следующие параметры: обозначение нажатой кнопки мыши (если она была нажата) — сумма чисел 1 (левая кнопка), 2 (правая кнопка) и 4 (средняя кнопка); обозначение нажатой клавиши-модификатора — сумма чисел 1 (была нажата клавиша <Shift>), 2 (клавиша <Ctrl>) и 4 (клавиша <Alt); горизонтальная координата точки, в которой находился курсор мыши, и ее вертикальная координата (обе координаты отсчитываются относительно "окна" проигрывателя). Все параметры передаются в числовом виде. Это событие генерируется только в том случае, если свойству (параметру) `SendMouseMoveEvents` проигрывателя присвоено значение `true` ("1")
`MouseUp`	Возникает при отпускании нажатой ранее кнопки мыши, если ее курсор находится над проигрывателем. Обработчик принимает следующие параметры: обозначение отпущенной кнопки мыши — сумма чисел 1 (левая кнопка), 2 (правая кнопка) и 4 (средняя кнопка); обозначение нажатой клавиши-модификатора — сумма чисел 1 (была нажата клавиша <Shift>), 2 (клавиша <Ctrl>) и 4 (клавиша <Alt>); горизонтальная координата точки, в которой находился курсор мыши, и ее вертикальная координата (обе координаты отсчитываются относительно "окна" проигрывателя). Все параметры передаются в числовом виде. Это событие генерируется только в том случае, если свойству (параметру) `SendMouseClickEvents` проигрывателя присвоено значение `true` ("1")
`NewStream`	Возникает при начале загрузки нового мультимедийного файла
`OpenStateChange`	Возникает при изменении состояния открытия мультимедийного файла. Обработчик принимает два параметра — предыдущее и новое состояния. Эти параметры могут принимать значения: 0 (файл не открыт), 1 (идет загрузка файла описания потокового аудио или видео), 2 (идет загрузка файла описания интернет-радиостанции), 3 (поиск сервера), 4 (соединение с сервером), 5 (открытие файла) и 6 (файл открыт). Это событие генерируется только в том случае, если свойству (параметру) `SendOpenStateChangeEvents` проигрывателя присвоено значение `true` ("1")

Таблица 9.8 *(окончание)*

Событие	Описание
`PlayStateChange`	Возникает при изменении состояния воспроизведения мультимедийного файла. Обработчик принимает два параметра — предыдущее и новое состояния. Эти параметры могут принимать значения: 0 (воспроизведение остановлено), 1 (воспроизведение приостановлено), 2 (файл воспроизводится), 3 (идет буферизация потока), 4 (идет прокрутка вперед), 5 (идет прокрутка назад), 6 (выполняется переход на следующую позицию), 7 (выполняется переход на предыдущую позицию) и 8 (файл не был открыт). Это событие генерируется только в том случае, если свойству (параметру) `SendPlayStateChangeEvents` проигрывателя присвоено значение `true` ("1")
`PositionChange`	Возникает при изменении посетителем позиции бегунка. Обработчик принимает два параметра — предыдущее и новое состояния в секундах, переданные в числовом виде
`ReadyStateChange`	Возникает при изменении состояния готовности проигрывателя к воспроизведению мультимедийного файла. Обработчик принимает единственный параметр — новое состояние. Он может принимать значения: 0 (файл не был задан, т. е. свойству `FileName` не был присвоен его интернет-адрес), 1 (идет загрузка файла), 3 (загружен не весь файл, но воспроизведение можно начинать), 4 (файл загружен полностью)
`Warning`	Возникает, когда проигрыватель сталкивается с возможной проблемой. Обработчик принимает следующие параметры: тип проблемы (0 — звуковое устройство недоступно, 1 — неизвестный формат файла, 2 — файл не может быть воспроизведен), дополнительная информация о проблеме в виде числа и строковое описание проблемы. Это событие генерируется только в том случае, если свойству (параметру) `SendWarningEvents` проигрывателя присвоено значение `true` ("1")

На заметку

Полный список свойств, методов и событий, поддерживаемых проигрывателем Windows Media, можно узнать на посвященной ему странице сайта MSDN (**http://msdn.microsoft.com/library/en-us/wmp6sdk/htm/microsoftwindowsmediaplayercontrolversion64sdk.asp**).

Свойства и методы проигрывателя Windows Media дополнительных пояснений не требуют. А вот с его событиями не все так просто. Для привязки к ним обработчиков нужно использовать не хорошо знакомые нам по *главе 6* приемы, а особый синтаксис, поддерживаемый только Internet Explorer. Сейчас мы его рассмотрим.

Вот формат записи обработчиков событий в стиле Internet Explorer:

```
<SCRIPT FOR="<имя тега <OBJECT>, создающего проигрыватель>"
EVENT="<событие и передаваемые им параметры>">
  <тело обработчика>
</SCRIPT>
```

Здесь используется все тот же хорошо знакомый нам тег `<SCRIPT>`. Он содержит два обязательных в данном случае атрибута. Атрибут `FOR` задает имя элемента ActiveX, к которому привязывается обработчик, — то самое имя, что задается атрибутом `ID`. Атрибут `EVENT` задает само событие, которое будет обрабатывать обработчик, и передаваемые обработчику параметры этого события. Значение атрибута `EVENT` записывается в таком формате:

```
<имя события>([<список передаваемых событием параметров, разделенных
запятыми>])
```

то есть как объявление функции. Переданные в обработчик параметры мы можем использовать в его теле.

Например (тело обработчика события опущено):

```
<SCRIPT FOR="o" EVENT="Disconnect(pReason)">
 . . .
</SCRIPT>
```

Этот обработчик будет обрабатывать событие `Disconnect`, возникающее в элементе ActiveX с именем `o` при разрыве соединения с сервером и передающее ему один параметр `pReason` — число, которое обозначает причину разрыва (см. табл. 9.8).

> ***НА ЗАМЕТКУ***
>
> Кстати, описанный ранее способ можно использовать и для привязки обработчика события к обычному элементу страницы: абзацу, гиперссылке, графическому изображению и пр. Но так практически никогда не делают, поскольку этот способ поддерживается только Internet Explorer.

Настала пора примеров — они помогут нам научиться работать со свойствами, методами и событиями проигрывателя Windows Media.

Первый пример — страница с помещенным на нее аудиоклипом. Она также содержит особый сценарий, который выводит на страницу продолжительность клипа в секундах после окончания его загрузки.

```
<HTML>
  <HEAD>
    <TITLE>Продолжительность клипа</TITLE>
```

```
  </HEAD>
  <BODY>
    <OBJECT CLASSID="clsid:22D6F312-B0F6-11D0-94AB-0080C74C7E95" ID="o">
      <PARAM NAME="FileName" VALUE="sound.wav">
    </OBJECT>
    <P ID="output"> </P>
    <SCRIPT FOR="o" EVENT="ReadyStateChange(pState)">
      var oObj = document.all["o"];
      var outputObj = document.all["output"];
      if (pState == 4)
        outputObj.firstChild.nodeValue = "Продолжительность клипа: " +
        oObj.Duration.toString() + " секунд.";
    </SCRIPT>
  </BODY>
</HTML>
```

Продолжительность клипа можно будет выяснить только после окончания его загрузки. Для этого необходимо обрабатывать событие `ReadyStateChange`, возникающее при изменении состояния загрузки клипа. Обработчику этого события передается один параметр — число, обозначающее новое состояние загрузки клипа (см. табл. 9.8). Когда с этим параметром передается число 4, значит, клип полностью загружен, и его параметры можно определить.

Исходя из этого, мы привязываем к элементу ActiveX с именем `o` обработчик события `ReadyStateChange`, пользуясь описанным ранее синтаксисом. В обработчик будет передан параметр `pState`, значением которого будет новое состояние загрузки клипа в виде числа. В теле обработчика мы проверяем значение этого параметра и, если оно равно 4 (клип загружен), выводим его продолжительность в абзаце `output`. Продолжительность клипа в секундах возвращает описанное в табл. 9.6 свойство `Duration`.

Второй пример будет представлять собой немного переделанную страницу с фоновым звуком, которую мы рассмотрели ранее. Она предоставит посетителю возможность приостанавливать воспроизведение фонового звука и запускать его снова.

```
<HTML>
  <HEAD>
    <TITLE>Управляемый фоновый звук</TITLE>
    <SCRIPT>
      function doPause() {
        var oObj = document.getElementById("o");
```

```
        oObj.Pause();
      }
      function doResume() {
        var oObj = document.getElementById("o");
        oObj.Play();
      }
    </SCRIPT>
  </HEAD>
  <BODY>
    <OBJECT CLASSID="clsid:22D6F312-B0F6-11D0-94AB-0080C74C7E95" ID="o">
      <PARAM NAME="FileName" VALUE="sound.wav">
      <PARAM NAME="PlayCount" VALUE="0">
      <PARAM NAME="ShowAudioControls" VALUE="0">
      <PARAM NAME="ShowControls" VALUE="0">
      <PARAM NAME="ShowDisplay" VALUE="0">
      <PARAM NAME="ShowPositionControls" VALUE="0">
      <PARAM NAME="ShowStatusBar" VALUE="0">
      <PARAM NAME="ShowTracker" VALUE="0">
    </OBJECT>
    <P><A HREF="#" ONCLICK="doPause();">Пауза</A></P>
    <P><A HREF="#" ONCLICK="doResume();">Пуск</A></P>
  </BODY>
</HTML>
```

Здесь мы используем те же параметры проигрывателя Windows Media, что и в предыдущей странице с фоновым звуком. Для приостановки и возобновления воспроизведения мы применяем две "пустые" гиперссылки, не указывающие никуда (для этого, как описано в *главе 2*, достаточно задать в качестве значения атрибута `HREF` тега `<A>` символ `#`). К этим гиперссылкам привязаны обработчики события `onClick` с именами `doPause` и `doResume`. Первый обработчик приостанавливает воспроизведение звука с помощью метода `Pause`, второй — возобновляет, используя метод `Play`. Оба этих метода были описаны в табл. 9.7.

Управление проигрывателем Shockwave/Flash

Проигрыватель Shockwave/Flash также поддерживает некоторое количество полезных для нас свойств и методов, но не событий. Поддерживаемые им свойства представлены в табл. 9.9, а методы — в табл. 9.10.

Таблица 9.9. *Некоторые свойства, поддерживаемые проигрывателем Shockwave/Flash*

Свойство	Описание
`AlignMode`	Задает или возвращает выравнивание "окна" проигрывателя на Web-странице. Доступны значения 0 (выравнивание по центру страницы), 1 (по левой границе) , 2 (по правой), 4 (по верхней) и 8 (по нижней)
`FrameNum`	Задает или возвращает номер текущего кадра фильма
`IsPlaying`	Возвращает `true`, если фильм в данный момент воспроизводится, и `false` в противном случае
`Loop`	Если `true`, фильм будет зациклен, если `false`, не будет
`Movie`	Задает или возвращает интернет-адрес файла с изображением Shockwave/Flash
`Quality`	Задает или возвращает качество вывода изображения. Доступны значения от 0 (самое низкое качество) до 3 (самое высокое качество)
`ReadyState`	Возвращает состояние, в котором находится изображение или фильм Shockwave/Flash. Значения этого свойства: 0 (изображение только начало загружаться), 1 (часть изображения загружена, но воспроизведение еще не началось), 2 (часть изображения загружена, и скоро начнется воспроизведение), 3 (часть изображения загружена, и изображение воспроизводится) и 4 (изображение полностью загружено)
`ScaleMode`	Задает или возвращает параметры масштабирования изображения в "окне" проигрывателя. Доступны значения: 0 (масштабирование с сохранением пропорций, но могут появиться пустые области), 1 (то же самое, но без пустых областей — проигрыватель обрежет изображение, чтобы не допустить их появления) и 2 (масштабирование без сохранения пропорций)
`TotalFrames`	Возвращает количество кадров фильма в числовом виде

Таблица 9.10. *Некоторые методы, поддерживаемые проигрывателем Shockwave/Flash*

Метод	Описание
`GotoFrame(`*<номер кадра>*`)`	Перемещает указатель на кадр с заданным в числовом виде номером. Нумерация кадров начинается с нуля
`PercentLoaded()`	Возвращает, сколько процентов изображения Shockwave/Flash загружено к данному моменту, в числовом виде

Таблица 9.10 *(окончание)*

Метод	Описание
`Play()`	Запускает воспроизведение фильма
`Rewind()`	Перематывает фильм на начало
`StopPlay()`	Останавливает воспроизведение фильма

> ***НА ЗАМЕТКУ***
>
> Полный список свойств и методов, поддерживаемых проигрывателем Shockwave/Flash, можно найти в разделе поддержки (**http://www.macromedia.com/support/flash**) Flash сайта фирмы Adobe.

Для закрепления полученных знаний напишем две страницы.

Первая страница будет возвращать количество кадров в баннере Shockwave/Flash, который на нее помещен. Поскольку мы не можем отследить окончание его загрузки (проигрыватель Shockwave/Flash не поддерживает события), создадим на странице абзац и привяжем к нему обработчик события `onClick`, который и будет выводить количество кадров в баннере. Ведь визуально мы можем отследить окончание загрузки баннера, не так ли?

```
<HTML>
  <HEAD>
    <TITLE>Количество кадров в баннере</TITLE>
    <SCRIPT>
      function pClick() {
        var oObj = document.getElementById("o");
        var outputObj = document.getElementById("output");
        outputObj.firstChild.nodeValue = "Количество кадров в баннере: "
        + oObj.TotalFrames.toString() + ".";
      }
    </SCRIPT>
  </HEAD>
  <BODY>
    <OBJECT CLASSID="clsid:d27cdb6e-ae6d-11cf-96b8-444553540000" ID="o">
      <PARAM NAME="movie" VALUE="banner.swf">
    </OBJECT>
    <P ONCLICK="pClick();">Получить количество кадров</P>
```

```
    <P ID="output"> </P>
  </BODY>
</HTML>
```

Комментировать здесь нечего, кроме того, что для получения количества кадров мы используем свойство `TotalFrames` (см. табл. 9.9).

Вторая страница позволит посетителю управлять воспроизведением помещенного на ней баннера.

```
<HTML>
  <HEAD>
    <TITLE>Управляемый баннер</TITLE>
    <SCRIPT>
      function doPause() {
        var oObj = document.getElementById("o");
        oObj.StopPlay();
      }
      function doResume() {
        var oObj = document.getElementById("o");
        oObj.Play();
      }
    </SCRIPT>
  </HEAD>
  <BODY>
    <OBJECT CLASSID="clsid:d27cdb6e-ae6d-11cf-96b8-444553540000" ID="o">
      <PARAM NAME="movie" VALUE="banner.swf">
    </OBJECT>
    <P><A HREF="#" ONCLICK="doPause();">Пауза</A></P>
    <P><A HREF="#" ONCLICK="doResume();">Пуск</A></P>
  </BODY>
</HTML>
```

Здесь мы также используем две "пустые" гиперссылки с привязанными к ним обработчиками события `onClick`. Эти обработчики и выполняют приостановку и запуск воспроизведения с помощью методов `StopPlay` и `Play`, описанных в табл. 9.10.

На этом все о мультимедийных элементах!

Что дальше?

В данной главе мы научились управлять графическими изображениями, картами-изображениями и создавать горячие изображения и полосы навигации. Также мы много узнали о мультимедийных элементах, модулях расширения и элементах ActiveX. Что ж, эру мультимедиа мы встретим во всеоружии!

В следующей главе мы вернемся к рассмотренным в *главе 3* стилям CSS и контейнерам и узнаем много нового о них. Мы познакомимся со свободно позиционируемыми элементами страницы, которые мы можем разместить в любом ее месте. И, разумеется, научимся управлять ими программно: позиционировать, задавать размеры, скрывать и даже перемещать по странице, создавая простую, но эффектную анимацию. Сделаем небольшой привал и снова двинемся вперед по пути Web-программирования!

Глава 10

Управление свободно позиционируемыми элементами. Анимация на Web-страницах

В *главе 3* мы познакомились со стилями CSS, позволяющими задавать для элементов страницы весьма богатое оформление. С их помощью мы можем изменить параметры шрифта, которым набран текст абзацев и заголовков, задать размеры элемента страницы, указать величину отступов от содержимого элемента страницы до его границы и между границами соседних элементов, создать рамки, настроить фон и пр. В общем, сделать с любым элементом страницы все, на что не способен HTML.

Также в *главе 3* мы узнали о контейнерах. Они позволяют нам объединить несколько элементов страницы, чтобы впоследствии применить к ним единый стиль. Так что, скажем, окружить несколько абзацев одной рамкой для нас сейчас не проблема.

В *главе 8* мы научились управлять стилями, привязанными к элементам страницы, из сценариев. Там же мы написали в качестве примеров несколько страниц с элементами, меняющими оформление в ответ на события мыши. Это оказалось совсем не сложно.

В этой главе узнаем о стилях и контейнерах кое-что новое. Можно даже сказать — революционное! Нечто такое, о чем до этого не догадывались. Но, как говорит Виктор Шендерович в своей радиопередаче "Плавленый сырок", обо всем по порядку…

Свободно позиционируемые элементы

Все элементы страниц, с которыми мы имели дело в предыдущих главах, имели одну особенность. Они всегда находились на том месте страницы, в котором присутствует создающий их HTML-код. Переместить на другое

место страницы без изменения ее HTML-кода, "ручного" или программного, мы не могли. (Хотя атрибуты стиля `float` и `clear` могут заставить элемент страницы прижаться к левой или правой стороне окна Web-обозревателя, радикально ситуацию они не меняют. Эти атрибуты описаны в *главе 3*.)

Что такое свободно позиционируемый элемент

Некоторые атрибуты стиля, не рассмотренные нами в *главе 3*, позволяют задать для элемента страницы произвольное местоположение, а также размеры и другие параметры. То есть с их помощью мы можем поместить элемент страницы туда, куда мы хотим, почти не меняя ее HTML-код, — достаточно написать соответствующий стиль и привязать его к данному элементу страницы.

Элементы страницы, чьи местоположение и размеры можно указать произвольно, называются *свободно позиционируемыми*, или просто *свободными*. Все остальные элементы, жестко привязанные к "своему" месту, назовем *непозиционируемыми*.

Свободно позиционируемым можно сделать не всякий элемент страницы. Как правило, в свободно позиционируемые элементы превращаются блочные контейнеры, создаваемые тегом `<DIV>`. (О блочных контейнерах см. *главу 3*.) Они подходят для этого наилучшим образом. Поэтому очень часто говорят о *свободно позиционируемых контейнерах*, подчеркивая "контейнерную" сущность свободных элементов.

Свободно позиционируемые элементы имеют множество интересных особенностей, которые Web-дизайнеры, разумеется, применяют на практике.

- Местоположение и размеры свободно позиционируемого элемента задаются произвольно в виде горизонтальной и вертикальной координат их верхнего левого угла, ширины и высоты.
- Местоположение и размеры свободно позиционируемых элементов не меняются ни в каких случаях, даже при изменении размеров окна Web-обозревателя. (Хотя мы можем менять их из сценариев; об этом будет рассказано далее.)
- Свободно позиционируемые элементы могут иметь любое содержимое, в том числе и другие свободно позиционируемые элементы.
- Если содержимое свободно позиционируемого элемента не помещается в нем, то элемент может как увеличить свои размеры, так и не увеличить; в последнем случае часть содержимого, не помещающаяся в элементе, не будет видна. Возможно также такое поведение свободного элемента, при котором в нем появляются полосы прокрутки для доступа к скрытой части

его содержимого (этакое независимое "окно" в окне Web-обозревателя, сделанное без использования фреймов).

- Свободно позиционируемые элементы находятся "выше" обычного, непозиционируемого содержимого страницы, как бы "плавают" над ним и перекрывают его. Из-за этого их также называют *плавающими элементами*.
- Свободно позиционируемые элементы могут перекрывать друг друга. При этом свободный элемент, созданный в HTML-коде позже, перекроет свободный элемент, созданный раньше. Непозиционируемое содержимое страницы свободные элементы перекрывают в любом случае.
- Слово "перекрывают" в предыдущих двух пунктах обозначает, что содержимое страницы, находящееся под свободно позиционируемым элементом, не будет видно — его скроет свободный элемент.
- Свободно позиционируемый элемент можно сделать невидимым (собственно, как и любой другой элемент страницы).

Как видим, свободно позиционируемые элементы обладают весьма примечательными особенностями. Из-за этого их так любят Web-программисты, занимающиеся созданием страниц, имитирующих приложения Windows: всяческие "окна" и "меню" с их помощью создаются достаточно просто (конечно, если не учитывать исключительную сложность сценариев, обеспечивающих их работу).

Настала пора изучить атрибуты стиля, с помощью которых создаются свободные элементы.

Создание свободно позиционируемых элементов

Рассмотрим небольшую Web-страничку, содержащую простой текст. Ее HTML-код приведен далее.

```
<HTML>
  <HEAD>
    <TITLE>Свободно позиционируемый элемент</TITLE>
  </HEAD>
  <BODY>
    <P>Это текст, который будет показан в окне Web-обозревателя. Это
      текст, который будет показан в окне Web-обозревателя. Это текст,
      который будет показан в окне Web-обозревателя. Это текст, который
      будет показан в окне Web-обозревателя. Это текст, который будет
      показан в окне Web-обозревателя. Это текст, который будет показан
      в окне Web-обозревателя.</P>
  </BODY>
</HTML>
```

Что мы увидим, если откроем данную страничку в Web-обозревателе? Правильно — обычный абзац с повторяющимся текстом.

Теперь преобразуем этот абзац в свободно позиционируемый элемент. Для этого изменим HTML-код страницы вот так (изменения выделены полужирным шрифтом).

```
<HTML>
  <HEAD>
    <TITLE>Свободно позиционируемый элемент</TITLE>
    <STYLE>
      #para {position: absolute;
             left: 50px;
             top: 50px;
             width: 200px;
             height: 100px;
             background-color: #CCCCCC;}
    </STYLE>
  </HEAD>
  <BODY>
    <DIV ID="para">Это текст, который будет показан в окне
      Web-обозревателя. Это текст, который будет показан в окне
      Web-обозревателя. Это текст, который будет показан в окне
      Web-обозревателя. Это текст, который будет показан в окне
      Web-обозревателя. Это текст, который будет показан в окне
      Web-обозревателя.</DIV>
  </BODY>
</HTML>
```

Сохраним готовую страницу в файле под именем 10.1.htm и откроем ее в Web-обозревателе. То, что мы увидим, показано на рис. 10.1.

Вот мы и сделали первый свободно позиционируемый элемент. И для этого нам потребовалось внести в исходный HTML-код совсем небольшие изменения. Давайте рассмотрим их подробнее.

Прежде всего, мы поместили текст абзаца в блочный контейнер, создаваемый тегом `<DIV>`. (Обычный абзац, созданный с помощью тега `<P>`, как и многие другие элементы страницы, не может быть позиционирован свободно.) Потом к полученному контейнеру мы привязали стиль-селектор `para`.

А вот этот стиль-селектор нужно рассмотреть подробнее. В нем мы использовали множество еще не знакомых нам атрибутов стиля.

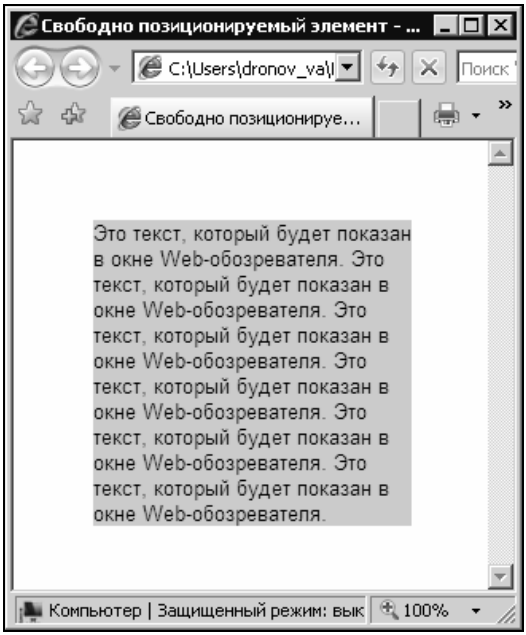

Рис. 10.1. Свободно позиционируемый элемент

Самый важный из них — это атрибут стиля `position`. Он задает способ позиционирования элемента страницы.

```
position: static | absolute | relative;
```

Как видим, этот атрибут может принимать следующие значения:

- "static" — элемент страницы является непозиционируемым (значение по умолчанию);
- "absolute" — элемент страницы является свободно позиционируемым. Его координаты отсчитываются относительно родителя;
- "relative" — элемент страницы является *относительно позиционируемым*. Координаты таких элементов страницы отсчитываются от той точки, где бы он находился, если бы был непозиционируемым.

Видно, что для превращения элемента страницы в свободно позиционируемый нужно присвоить атрибуту стиля `position` значение "absolute". Что мы и сделали —

```
position: absolute;
```

Для задания координаты верхнего левого угла свободного элемента служат атрибуты стиля `left` и `top`.

```
left | top: <значение координаты>;
```

Первый из этих атрибутов задает горизонтальную координату, второй — вертикальную. Обе эти координаты в случае свободного элемента отсчитываются от верхнего левого угла его родителя. Заданы они могут быть в любой единице измерения, поддерживаемой CSS (см. табл. 3.1).

```
left: 50px;
top: 50px;
```

В случае относительно позиционируемых элементов координаты левого верхнего угла элемента отсчитываются от той позиции, где бы он находился, если бы был непозиционируемым. Собственно, об этом уже говорилось ранее.

> ***Внимание!***
>
> Атрибуты стиля `left` и `top` имеют силу только для свободно позиционируемых и относительно позиционируемых элементов, то есть тех, у которых атрибут `position` имеет значение "absolute" или "relative" соответственно.

Для задания размеров используются уже знакомые нам по *главе 3* атрибуты стиля `width` и `height`. Они задают, соответственно, ширину и высоту свободного элемента в любой поддерживаемой CSS единице измерения.

```
width: 200px;
height: 100px;
```

Эти атрибуты имеют силу и для обычных элементов страницы.

Специально чтобы свободный элемент был лучше заметен, мы задали для него серый цвет фона. За цвет фона "отвечает" давно знакомый нам атрибут стиля `background-color`.

```
background-color: #CCCCCC;
```

Собственно, вот и весь стиль-селектор `para`. Небольшой — а каков результат!

Давайте рассмотрим остальные атрибуты стиля, которые применяются при создании свободных элементов и могут быть нам полезны.

Мы уже знаем, что свободные элементы могут перекрывать друг друга. Порядок их перекрытия друг другом задается числом и называется *z-индексом*. Так, самый первый свободный элемент, присутствующий на странице, будет иметь z-индекс 1 и располагаться в самом низу, второй — z-индекс 2 и располагаться выше первого, третий — z-индекс 3 и располагаться выше второго и т. д. Самый последний свободный элемент из созданных на странице будет иметь максимальный z-индекс и располагаться выше всех остальных свободных элементов.

Атрибут стиля `z-index` позволяет нам явно задать z-индекс свободного элемента.

```
z-index: <значение z-индекса> | auto;
```

Z-индекс задается в виде числа. Также может быть задано значение "auto" — тогда свободный элемент получит z-индекс по тем правилам, которые были описаны ранее. Кстати, "auto" — значение атрибута `z-index` по умолчанию.

```
z-index: 100;
```

Зададим для нашего свободного элемента очень большой z-индекс, чтобы он всегда был гарантированно выше всех остальных свободных элементов.

Как говорилось ранее, свободный элемент может иметь любое содержимое. И может случиться так, что это самое содержимое в нем не помещается. По умолчанию свободный элемент увеличит свои размеры (обычно — высоту), чтобы вместить содержимое полностью. Но мы можем изменить это поведение.

Атрибут `overflow` как раз и задает поведение свободного элемента в случае, если его содержимое в нем не помещается.

```
overflow: visible | scroll | hidden | auto;
```

Значения этого атрибута таковы:

- "visible" — свободный элемент увеличит свои размеры, чтобы полностью вместить содержимое (поведение по умолчанию);
- "scroll" — в свободном элементе будут присутствовать полосы прокрутки для доступа к скрытой части его содержимого. Причем полосы прокрутки будут присутствовать всегда, даже если содержимое свободного элемента полностью в нем помещается;
- "hidden" — содержимое, не помещающееся в свободном элементе, останется скрытым; размеры свободного элемента не изменятся и полосы прокрутки в нем не появятся;
- "auto" — в свободном элементе появятся полосы прокрутки для доступа к скрытой части его содержимого. Причем появятся они только в том случае, если содержимое свободного элемента не будет в нем помещаться.

Давайте создадим в нашем свободном элементе полосы прокрутки. Для этого достаточно добавить в определение стиля-селектора `para` следующую строку:

```
overflow: auto;
```

Результат показан на рис. 10.2.

Атрибут `visibility`, который нам уже знаком по *главе 3*, позволяет временно скрыть свободный элемент (как и любой другой элемент страницы). Значение "hidden" скрывает свободный элемент, значение "visible" — выводит на экран, а значение "inherit" заставляет его "наследовать" видимость или невидимость у родителя.

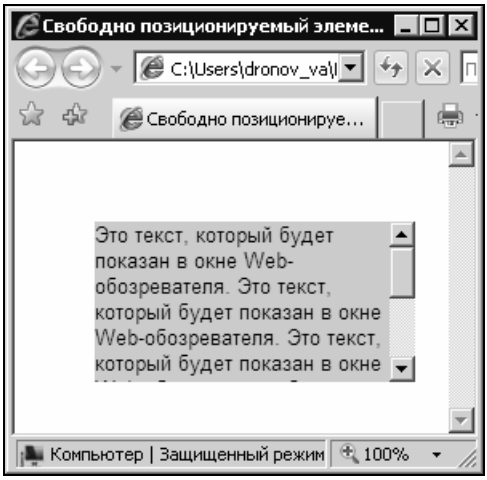

Рис. 10.2. Свободно позиционируемый элемент с полосами прокрутки

Еще один полезный атрибут, который мы здесь рассмотрим, — `clip`. Он позволяет задать координаты прямоугольной области, которая задает видимую часть свободного элемента ("прямоугольник видимости"). То есть та часть содержимого свободного элемента, что входит в заданный прямоугольник, будет видима на странице, а та, что не входит, — видима не будет.

```
clip: rect(<верхняя граница> <правая граница> <нижняя граница>
<левая граница>) | auto;
```

Здесь:

- `верхняя граница` — расстояние от верхней границы свободного элемента до верхней границы "прямоугольника видимости" по вертикали;
- `правая граница` — расстояние от левой границы свободного элемента до правой границы "прямоугольника видимости" по горизонтали;
- `нижняя граница` — расстояние от верхней границы свободного элемента до нижней границы "прямоугольника видимости" по вертикали;
- `левая граница` — расстояние от левой границы свободного элемента до левой границы "прямоугольника видимости" по горизонтали.

Координаты "прямоугольника видимости" задаются относительно свободного элемента. Также запомним порядок, в котором указываются координаты, — он должен быть именно таким.

Значение "auto" атрибута стиля `clip` убирает "прямоугольник видимости" и делает все содержимое свободного элемента видимым. Это значение по умолчанию.

Внимание!

Атрибут стиля `clip` имеет силу только для свободно позиционируемых элементов, то есть тех, у которых атрибут `position` имеет значение "absolute".

Давайте в качестве эксперимента создадим для нашего свободного элемента "прямоугольник видимости". Для этого добавим в определение стиля-селектора `para` такую строку:

```
clip: rect(20 200 50 50);
```

Результат показан на рис. 10.3.

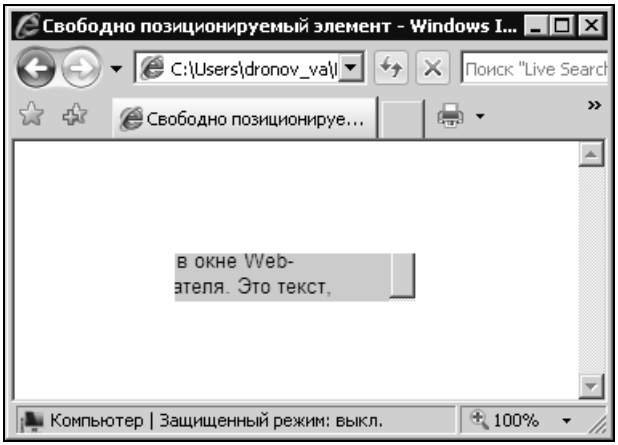

Рис. 10.3. Свободно позиционируемый элемент с полосами прокрутки и "прямоугольником видимости"

Да, получилось не очень аккуратно — видна часть бегунка полосы прокрутки. Именно поэтому "прямоугольник видимости" используют крайне редко, в основном, для создания специальных эффектов. Давайте его уберем, для чего исправим приведенную ранее строку CSS-кода следующим образом:

```
clip: auto;
```

то есть дадим атрибуту стиля `clip` значение по умолчанию "auto".

Собственно, это все атрибуты стиля, которые применяются для создания свободно позиционируемых элементов. Разумеется, мы можем использовать и другие атрибуты стиля, описанные в *главе 3*, для задания параметров текста, абзаца, отступов, параметров рамки и пр. И тогда наши свободные элементы будут выглядеть просто замечательно!

Управление свободно позиционируемыми элементами из сценариев

Особого толку в свободно позиционируемых элементах нет. В Web-дизайне они применяются редко, так как не всегда подходят для размещения содержимого страницы. (Обычные таблицы и непозиционируемые контейнеры для этого подходят куда лучше.) Совсем другое — свободные элементы, управляемые программно, из сценариев. Тут-то и раскрывается их подлинная мощь.

Ранее мы рассмотрели несколько атрибутов стилей, применяемых для создания свободных элементов. Программный доступ к их значениям осуществляется так же, как и к значениями других атрибутов стилей, — через свойства объекта `CSSRule`, представляющего стиль (см. *главу 8*). Правила, по которым формируются имена свойств объекта `CSSRule`, соответствующие различным атрибутам стилей, были рассмотрены там же. Так что на этом мы останавливаться не будем.

В *главе 8* приведена табл. 8.4, перечисляющая некоторые свойства объекта `HTMLElement`, не рассмотренные ранее. В основном, это свойства, позволяющие получить координаты самого элемента страницы, координаты его клиентской области (то есть пространства, в котором помещается содержимое элемента страницы, в том числе и свободного), положение полос прокрутки и пр. В случае непозиционируемого элемента страницы эти свойства малополезны, но в случае свободного элемента зачастую без них никуда. Так что вложим в книгу на странице, где приведена табл. 8.4, закладку.

И рассмотрим три примера страниц со свободными элементами и сценариями, которые ими управляют. В самом деле, хорошо подобранный пример — лучший учебник.

Первая страница будет иметь свободно позиционируемый элемент, который при любом изменении размера страницы будет находиться по ее центру.

```
<HTML>
  <HEAD>
    <TITLE>Свободно позиционируемый элемент</TITLE>
    <STYLE>
      #cont {position: absolute;
             left: 50px;
             top: 50px;
             width: 100px;
             height: 100px;
             background-color: #CCCCCC;}
    </STYLE>
```

```
    <SCRIPT>
      function winResize() {
        var contObj = document.getElementById("cont");
        var bodyObj = document.body;
        var contWidth = contObj.offsetWidth;
        var contHeight = contObj.offsetHeight;
        var bodyWidth = bodyObj.clientWidth;
        var bodyHeight = bodyObj.clientHeight;
        contObj.style.left = (bodyWidth - contWidth) / 2 + "px";
        contObj.style.top = (bodyHeight - contHeight) / 2 + "px";
      }
    </SCRIPT>
  </HEAD>
  <BODY>
    <DIV ID="cont">Этот свободно позиционируемый элемент всегда будет
    находиться в центре страницы.</DIV>
    <SCRIPT>
      window.onresize = winResize;
      winResize();
    </SCRIPT>
  </BODY>
</HTML>
```

Поскольку нам нужно, чтобы свободный элемент `cont` всегда находился в центре страницы, нам придется как-то отслеживать изменение размера окна Web-обозревателя. Для этого, как было сказано в *главе 7*, мы должны обрабатывать событие `onResize` объекта `Window`, а если точнее, то экземпляра этого объекта, представляющего текущее окно и доступного через встроенную переменную `window`. Нам нужно написать функцию — обработчик этого события и присвоить ее свойству `onresize` объекта `Window`. Что мы и делаем:

```
window.onresize = winResize;
winResize();
```

Да, обработчиком события `onResize` станет функция `winResize`. Отметим, что сразу же после присвоения ее свойству `onresize` выполняется вызов этой функции. Так мы позиционируем свободный элемент `cont` по центру страницы сразу же после ее загрузки.

Теперь рассмотрим функцию `winResize`. Там мы сначала получаем размеры нашего свободного элемента, воспользовавшись свойствами `offsetWidth` и `offsetHeight` объекта `HTMLElement`. Эти свойства, как было описано в табл. 8.4, возвращают полную ширину и высоту элемента страницы соот-

ветственно. Для получения размеров клиентской области секции тела страницы мы обращаемся уже к свойствам `clientWidth` и `clientHeight`, возвращающим, соответственно, ее ширину и высоту. Сама секция тела страницы доступна через свойство `body` экземпляра объекта `HTMLDocument`, представляющего текущую страницу и доступного через встроенную переменную `document`. Все это нам уже знакомо по *главам 5—7*.

```
var bodyObj = document.body;
var contWidth = contObj.offsetWidth;
var contHeight = contObj.offsetHeight;
var bodyWidth = bodyObj.clientWidth;
var bodyHeight = bodyObj.clientHeight;
```

Последние два выражения тела функции вычисляют новые координаты свободного элемента и присваивают их свойствам `left` и `top` экземпляра объекта `CSSRule`, представляющего привязанный к данному элементу стиль и доступного через свойство `style` объекта `HTMLElement`.

```
contObj.style.left = (bodyWidth - contWidth) / 2 + "px";
contObj.style.top = (bodyHeight - contHeight) / 2 + "px";
```

Отметим, что к числовому значению каждой координаты мы добавляем строку, обозначающую единицу измерения CSS, в нашем случае — пикселы (строка "px").

> ***НА ЗАМЕТКУ***
>
> Если в значении атрибута стиля не присутствует обозначение единицы измерения, Web-обозреватель считает, что это значение задано в пикселах. Но все же лучше задавать единицу измерения явно.

Вторая страница, которую мы напишем, будет содержать три гиперссылки; при наведении курсора мыши на любую из них на экране появится всплывающая подсказка. Это небольшое окно, содержащее поясняющий текст и знакомое нам по многим Windows-приложениям.

```
<HTML>
  <HEAD>
    <TITLE>Всплывающие подсказки</TITLE>
    <STYLE>
      #output {position: absolute;
               left: 50px;
               top: 50px;
               width: 100px;
               height: 20px;
               visibility: hidden;
```

```
            background-color: #FFFFCC}
    </STYLE>
    <SCRIPT>
      hints = [];
      hints["href1"] = "Перейти на первую страницу";
      hints["href2"] = "Перейти на вторую страницу";
      hints["href3"] = "Перейти на третью страницу";

      function hrefMouseOver(pID) {
        var hrefObj = document.getElementById(pID);
        var leftHint = hrefObj.offsetLeft;
        var topHint = hrefObj.offsetTop + hrefObj.offsetHeight;
        var outputObj = document.getElementById("output");
        outputObj.firstChild.nodeValue = hints[pID];
        with (outputObj.style) {
          left = leftHint + "px";
          top = topHint + "px";
          visibility = "visible";
        }
      }

      function hrefMouseOut() {
        var outputObj = document.getElementById("output");
        outputObj.style.visibility = "hidden";
      }
    </SCRIPT>
  </HEAD>
  <BODY>
    <DIV ID="output"> </DIV>
    <P><A ID="href1" HREF="page1.html"
    ONMOUSEOVER="hrefMouseOver('href1');"
    ONMOUSEOUT="hrefMouseOut();">Первая страница</A></P>
    <P><A ID="href2" HREF="page2.html"
    ONMOUSEOVER="hrefMouseOver('href2');"
    ONMOUSEOUT="hrefMouseOut();">Вторая страница</A></P>
    <P><A ID="href3" HREF="page3.html"
    ONMOUSEOVER="hrefMouseOver('href3');"
    ONMOUSEOUT="hrefMouseOut();">Третья страница</A></P>
  </BODY>
</HTML>
```

В принципе, очень многое здесь нам уже знакомо — в *главе 8* мы рассмотрели аналогичный сценарий, выводящий обычные, не всплывающие подсказки. Мы создали элемент страницы `output`, где выводится текст подсказок (только в этом случае он не абзац, а свободный элемент), для хранения текстов подсказок использовали ассоциативный массив `hints`, для вывода подсказки на экран — функцию `hrefMouseOver`, обрабатывающую событие `onMouseOver`, а для скрытия подсказки — функцию `hrefMouseOut`, обрабатывающую событие `onMouseOut`. Обе эти функции — обработчика событий мы привязали прямо к гиперссылкам, а в функцию `hrefMouseOver` передали в качестве параметра имя гиперссылки.

Что касается свободного элемента `output`, где выводится текст подсказок, то у него есть три особенности, о которых нужно обязательно поговорить.

- Мы задали для него желтый цвет фона (RGB-код `FFFFCC`). Если мы не зададим для него фон явно, он получит прозрачный фон (подробнее о параметрах фона см. *главу 3*), и тогда сквозь него будет просвечивать нижележащее содержимое страницы, что будет выглядеть некрасиво. В принципе, мы можем задать для него и белый цвет фона, но автор хотел сделать всплывающие подсказки на этой странице похожими на подсказки Windows.
- Мы задали для него очень маленькое значение высоты — всего 20 пикселов. Если его содержимое не будет в нем помещаться, он все равно растянется по вертикали. Если же мы зададим для него слишком большую высоту, может получиться так, что текст подсказки окажется совсем коротким и займет его не полностью, что также будет выглядеть некрасиво.
- Мы изначально сделали его невидимым, присвоив атрибуту стиля `visibility` значение "hidden". Это нужно, чтобы после загрузки страницы он не присутствовал на экране.

Теперь рассмотрим тело функции `hrefMouseOver`, выводящей подсказку на экран. Сначала мы получаем доступ к гиперссылке, чье имя передано в качестве параметра, и определяем горизонтальную координату его левой стороны (свойство `offsetLeft` объекта `HTMLElement`) и вертикальную координату ее нижней стороны (сумма значений свойств `offsetTop` и `offsetHeight` того же объекта; подробнее эти свойства описаны в табл. 8.4). Именно эти координаты будет иметь свободный элемент `output`. (Автор решил, что он должен выводиться прямо под гиперссылкой; может, это и не очень красиво, но реализуется проще.) Далее мы помещаем в этот свободный элемент текст подсказки, задаем для него полученные ранее координаты и делаем видимым, присвоив атрибуту стиля `visibility` значение "visible".

Функция `hrefMouseOut` очень проста — она только делает свободный элемент `output` снова невидимым, присвоив атрибуту стиля `visibility` значение "hidden". Вот и все.

Конечно, созданный нами пример весьма прост. Нам следовало бы задать для свободного элемента `output` такие параметры, чтобы он походил на всплывающие подсказки Windows, в частности, задать уменьшенный размер шрифта и тонкую серую рамку. Да и выводить его следовало бы в позиции курсора мыши. Но это заметно усложнило бы код, а мы ведь пока только учимся…

Анимация на Web-страницах

Если свободно позиционируемые элементы могут быть размещены в любом месте страницы и их местоположение может быть изменено программно, мы можем создавать с их помощью анимацию. Как — сейчас мы выясним.

Простейшая анимация

За счет чего достигается эффект анимации? Почему элемент страницы кажется движущимся по ней?

Дело в том, что координаты элемента периодически меняются. И меняются весьма быстро, не реже 12 раз в секунду, за счет чего мы и наблюдаем непрерывное движение. Такая частота выбрана оттого, что именно на ней человеческий глаз теряет способность различать отдельные приращения, "шаги" прерывистого движения. Короче говоря, за этим пределом прерывистое движение кажется человеку непрерывным. (На этом же принципе, кстати, работают кинематограф и телевидение.) Современные компьютеры достаточно мощны, и обеспечить такую частоту смены координат им вполне по силам.

Координаты анимированного элемента меняются таким образом, чтобы обеспечить его движение по определенной траектории. *Траектория* — суть обычная линия, прямая, кривая или ломаная, описывающая путь движения анимированного элемента. Она имеет *начальную точку*, откуда анимированный элемент начнет свое движение, и *конечную точку*, где он остановится.

Траектория практически любой сложности может быть с заданной точностью описана определенной математической функцией. Такая функция принимает некоторые параметры и возвращает координаты элемента. Давайте назовем ее *функцией траектории*.

Функция траектории, в общем случае, имеет такой вид:

```
{x, y, z} = f(Q, q, dq)
```

С возвращаемыми этой функцией результатами все просто. `x`, `y` и `z` — координаты анимированного элемента, соответственно, горизонтальная, вертикальная и порядок перекрытия (уже знакомый нам z-индекс). Конечно, функция, возвращающая сразу три координаты, — это общий случай. Обычно

изменяются две координаты элемента, `x` и `y`, а то и вовсе одна из них (в случае, если траектория представляет собой горизонтальную или вертикальную линию).

А вот с параметрами, принимаемыми функцией, все несколько сложнее. Всего, как мы видим, их три, и назначение их неочевидно. Давайте же рассмотрим все эти параметры по порядку.

Самый первый параметр — `Q`. Это *длина траектории* движения анимированного элемента, замеренная от начальной до конечной точек. Она может измеряться в пикселах, миллиметрах, градусах, радианах или каких-либо относительных единицах, например, процентах.

Вообще, единица измерения зависит от самой траектории: для прямолинейной больше подойдут пикселы или миллиметры, а для круговой — градусы или радианы. Важно одно: параметр `Q` должен каким-то образом обозначать полную длину траектории, по которой будет двигаться наш элемент.

Второй параметр (`q`) обозначает *текущую позицию*, занимаемую в данный момент времени анимированным элементом на траектории. Иными словами, это расстояние в единицах измерения траектории `Q`, которое этот элемент уже "пробежал" от начальной точки.

Последний, третий параметр — `dq`. Он задает *приращение*, на которое будет меняться величина `q` при каждом "шаге" анимированного элемента, фактически — скорость его движения.

Итак, что же должна делать функция траектории? Вот общий алгоритм, которому она должна следовать.

1. Принять начальные параметры и выполнить предварительные установки (прежде всего, установить анимированный элемент в начальную точку траектории движения).
2. Вычислить значения координат анимированного элемента на основании значения `q` и переместить этот элемент в точку с вычисленными координатами.
3. Проверить, дошел ли элемент до конечной точки траектории. Для этого нужно `q` сравнить с `Q`, и, если они равны либо `q` больше `Q`, перейти к пункту 6.
4. Увеличить значение `q` на величину `dq`.
5. Перейти к шагу 2.
6. Анимация закончена! Выполнить завершающие действия.

Основная сложность здесь заключается в реализации шага 2, то есть вычислении координат на основе значения `q`. Если траектория достаточно сложна,

описывающая ее функция (и, соответственно, "отвечающий" на нее сценарий) также сильно усложнится. Остальные же шаги реализуются очень просто и не составят трудности даже для начинающего Web-программиста.

Обычно, когда анимированный элемент достигает конца траектории (`q` становится равной или больше `Q`), анимация останавливается. Но так бывает не всегда. Функция траектории в этом случае может, например, снова выполнить начальные установки, поместив анимированный элемент в начало траектории, и запустить анимацию снова. Можно также сменить знак значения `dq` (изменить положительное значение на отрицательное или наоборот) и пустить анимацию задом наперед. Такая анимация, воспроизводящаяся постоянно, называется *зацикленной* или *бесконечной*.

По идее, сейчас следует создать пример страницы с анимированным элементом. Но мы не будем этого делать. А лучше поговорим, почему так делать ни в коем случае нельзя.

Анимация реального времени

Рассмотренный нами способ создания анимации имеет всего лишь одно более чем сомнительное достоинство и всего один, зато огромнейший недостаток.

Достоинство — простота, более того, очевидность реализации. Любой программист, даже малоквалифицированный, может сесть и в пять минут накропать Web-сценарий, реализующий движение элемента по прямолинейной траектории. И останется доволен.

Но давайте посмотрим на приведенный ранее алгоритм работы сценариев, реализующих эту анимацию. Видно, что он реализуется в виде цикла (о циклах было рассказано в *главе 4*), в теле которого выполняются все нужные действия: расчет новых координат, проверка, дошел ли анимированный элемент до конца траектории, и позиционирование анимированного элемента. Выполнение этого цикла отнимет у Web-обозревателя все выделенное ему операционной системой время. Абсолютно все! Он не сможет не то что "общаться" с пользователем, но даже вывести содержимое страницы на экран! Точнее, он-то ее выведет, но только после окончания анимации, когда анимированный элемент будет находиться в конечной точке траектории.

В случае зацикленной анимации ситуация еще хуже. Такая анимация реализуется в виде бесконечного цикла, который будет выполняться все время, пока запущен Web-обозреватель и в нем открыта данная страница. Web-обозреватель вообще не сможет вывести эту страницу на экран. Мы даже не сможем его корректно закрыть!

Это и есть единственный недостаток описанного ранее способа создания анимации. И какой недостаток!

Но что же делать? Выход довольно прост. И сейчас мы его рассмотрим.

В каждом компьютере имеется так называемый *системный таймер*, "тикающий" через равные промежутки времени. Операционная система обрабатывает эти "тики" и отслеживает таким образом текущее время. Кроме того, программы могут отслеживать момент "тика" и выполнять в этот момент какие-то действия.

К таким программам, "знающим" о существовании системного таймера, относится и Web-обозреватель. Мы можем указать ему определенный временной промежуток (*тайм-аут*) и функцию, которая будет выполнена по истечении этого промежутка. После выполнения этой функции отсчет временного промежутка начинается снова, после его окончания снова выполняется заданная функция и т. д.

Во время отсчета временного промежутка Web-обозреватель занимается своими делами: выводит страницу на экран и "общается" с посетителем. Времени у него на это предостаточно.

Теперь предположим, что заданная нами функция выполняет очередной "шаг" анимации. Отсчитав тайм-аут (давайте уж использовать этот термин, благо им пользуются профессиональные программисты), Web-обозреватель вызовет ее, и она изменит координаты анимированного элемента. После этого Web-обозреватель начинает отсчет следующего тайм-аута, а попутно изменит местоположение анимированного элемента согласно заданным функцией координатам и выполнит команды посетителя. Когда следующий тайм-аут истечет, снова будет вызвана функция, которая опять изменит координаты анимированного элемента. Когда начнется отсчет очередного, третьего по счету тайм-аута, Web-обозреватель снова обновит местоположение анимированного элемента. И так постоянно, пока мы специально не укажем прервать выполнение заданной функции.

Да это просто находка для Web-аниматоров!

Созданная таким образом анимация имеет еще одно достоинство, помимо того, что она не "нагружает" Web-обозреватель чрезмерно. Она жестко привязана к текущему времени, к "тикам" системного таймера. Поэтому ее часто называют *анимацией реального времени*.

Далее приведен алгоритм работы для функции траектории, созданной с использованием анимации реального времени.

1. Принять начальные параметры и выполнить предварительные установки (установить анимированный элемент в начальную точку траектории и запустить отсчет первого тайм-аута).

2. Когда тайм-аут закончится, вычислить значения координат анимированного элемента на основании значения `q` и переместить этот элемент в точку с вычисленными координатами.
3. Проверить, дошел ли элемент до конца траектории. Для этого следует q сравнить с `Q`, и, если они равны либо `q` больше `Q`, перейти к пункту 6.
4. Увеличить значение `q` на величину `dq`.
5. Завершить свою работу до окончания очередного тайм-аута и перейти к пункту 2.
6. Выполнить завершающие действия (в частности, остановить отсчет очередного тайм-аута).

Теперь давайте рассмотрим, как запустить и остановить отсчет тайм-аута. Для этого нам придется обратиться к объекту `Window`, представляющему окно Web-обозревателя и описанному в *главе 7*. Этот объект поддерживает четыре метода, которые мы пока что нигде не рассматривали.

Метод `setInterval` запускает отсчет тайм-аута, после окончания которого будет вызвана заданная функция. При этом после завершения выполнения тела этой функции отсчет тайм-аута начнется снова, после его окончания снова будет выполнена заданная функция и т. д.

```
setInterval(<функция>, <величина тайм-аута>)
```

Первым параметром этому методу передается функция, которая должна быть вызвана после очередного завершения тайм-аута. Вторым параметром передается числовое значение тайм-аута в миллисекундах.

Метод `setInterval` после выполнения создает в памяти компьютера особую структуру данных, называемую *таймером* (не путать с системным таймером!). Эта структура хранит значение тайм-аута, функцию, которую нужно выполнить после его окончания, и текущее время. Ее однозначно идентифицирует особое число, называемое *идентификатором таймера*, — оно-то и возвращается этим методом. Мы можем сохранить его в переменной и использовать впоследствии, чтобы удалить этот таймер и тем самым остановить отсчет тайм-аута.

```
var tm = window.setInterval(someFunction, 1000);
```

После выполнения этого выражения Web-обозреватель будет вызывать функцию `someFunction` каждую секунду (1000 миллисекунд). Идентификатор таймера, созданного вызовом метода `setInterval`, будет сохранен в переменной `tm`.

Для удаления таймера из памяти и, соответственно, прекращения отсчета тайм-аута следует использовать метод `clearInterval`. Он не возвращает значения.

```
clearInterval(<идентификатор таймера>)
```

Единственным параметром этому методу передается идентификатор таймера, возвращенный методом `setInterval`, который создал этот таймер.

Вообще, удалять таймер явно приходится довольно редко. Анимация на Web-страницах часто делается зацикленной, то есть таймер постоянно "занят делом". При закрытии же текущей страницы Web-обозреватель сам удаляет из памяти все объявленные в ее сценариях переменные, в том числе и идентификаторы таймеров, вместе с самими таймерами.

Мы также имеем возможность создать своего рода "вырожденный" таймер, "срабатывающий" только один раз. Такой таймер выполняет отсчет таймаута, после чего вызывает заданную функцию и сам удаляется из памяти. Для его создания используется метод `setTimeout`.

`setTimeout(`*<функция>*`, `*<величина тайм-аута>*`)`

Этот метод полностью аналогичен рассмотренному ранее методу `setInterval`: принимает те же параметры и возвращает все тот же идентификатор таймера.

А метод `cleatTimeout` удаляет таймер, созданный вызовом метода `setTimeout`.

`clearTimeout(`*<идентификатор таймера>*`)`

Он также аналогичен рассмотренному ранее методу `clearInterval`.

Удалять явно из памяти таймер, созданный вызовом метода `setTimeout`, приходится еще реже. Обычно это бывает нужно, чтобы прервать отсчет таймаута еще до того, как он закончился.

> ***Внимание!***
>
> Не забываем, что методы `setInterval` и `clearInterval` "отвечают" за таймер многократного срабатывания, а методы `setTimeout` и `clearTimeout` — за таймер однократного срабатывания. Названия этих методов часто вводят в заблуждение неопытных Web-программистов.

Теперь настала пора рассмотреть пример страницы с анимированным элементом. Пусть это будет графическое изображение, движущееся по странице по наклонной линии из точки с координатами {50, 50} в точку с координатами {200, 10}. Причем анимация будет зацикленной: при достижении конечной точки элемент начнет двигаться в обратную сторону, пока не достигнет начальной точки, а в начальной точке снова сменит направление движения и будет двигаться к конечной точке и т. д.

```
<HTML>
  <HEAD>
    <TITLE>Анимация</TITLE>
    <STYLE>
```

```
    #anim {position: absolute;
           left: 50px;
           top: 50px;
           width: 20px;
           height: 20px}
  </STYLE>
</HEAD>
<BODY>
  <DIV ID="anim"><IMG SRC="image.gif"></DIV>
  <SCRIPT>
    var x1 = 50;
    var y1 = 50;
    var x2 = 200;
    var y2 = 10;
    var qd = 5;
    var int = 100;

    if (x1 > x2) {
      var xmin = x2;
      var xmax = x1;
    } else {
      var xmin = x1;
      var xmax = x2;
    }
    var k = Math.abs((y2 - y1) / (x2 - x1));
    if (y1 > y2) k = -k;
    var x = x1;

    var animObj = document.getElementById("anim");

    function doStep() {
      if (x2 > x1)
        x += qd
      else
        x -= qd;
      animObj.style.left = x + "px";
      animObj.style.top = y1 + k * Math.abs(x - x1) + "px";
      if ((x <= xmin) || (x >= xmax)) qd = -qd;
    }
```

```
      animObj.style.left = x1 + "px";
      animObj.style.top = y1 + "px";

      window.setInterval(doStep, int);
    </SCRIPT>
  </BODY>
</HTML>
```

Сценарий, присутствующий на этой странице, весьма сложен, так что давайте рассмотрим его построчно, выражение за выражением.

```
var x1 = 50;
var y1 = 50;
var x2 = 200;
var y2 = 10;
var qd = 5;
var int = 100;
```

Сначала задаем координаты начальной (переменные `x1` и `y1`) и конечной (переменные `x2` и `y2`) точек, величину приращения по горизонтальной оси (переменная `qd`) и тайм-аут — 100 миллисекунд (0,1 секунда; переменная `int`). Обратим внимание, что мы задаем величину приращения по горизонтальной оси, чтобы упростить код; приращение по вертикальной оси мы будем вычислять.

```
if (x1 > x2) {
  var xmin = x2;
  var xmax = x1;
} else {
  var xmin = x1;
  var xmax = x2;
}
```

Вычисляем минимальную и максимальную координаты по горизонтальной оси. Это нужно, чтобы упростить код проверки достижения начальной или конечной точки.

```
var k = Math.abs((y2 - y1) / (x2 - x1));
```

Вычисляем отношение расстояний между начальной и конечной точками по вертикали и горизонтали. Ранее мы задали величину приращения анимации по горизонтальной оси; так вот, менять мы будем именно горизонтальную координату, а вертикальную — вычислять, пользуясь этим соотношением.

```
if (y1 > y2) k = -k;
```

Если вертикальная координата начальной точки больше вертикальной координаты точки конечной (то есть элемент движется снизу вверх), меняем знак полученного ранее соотношения.

```
var x = x1;
```

Задаем текущую горизонтальную координату анимированного элемента равной горизонтальной координате начальной точки.

```
var animObj = document.getElementById("anim");
```

Получаем доступ к анимированному элементу.

```
animObj.style.left = x1 + "px";
animObj.style.top = y1 + "px";
```

Устанавливаем анимированный элемент в начальную точку. Вообще, мы можем этого не делать, так как анимированный элемент уже находится в начальной точке (его координаты уже заданы в стиле-селекторе `anim`), но так будет нагляднее.

```
window.setInterval(doStep, int);
```

Запускаем отсчет тайм-аута. В качестве функции, вызываемой после каждого его окончания, зададим функцию `doStep`, которую рассмотрим далее. Идентификатор таймера мы не сохраняем, так как не будем его удалять — это сделает за нас Web-обозреватель, когда текущая страница будет закрыта (анимация все равно бесконечная).

Теперь рассмотрим функцию `doStep`.

```
function doStep() {
  if (x2 > x1)
    x += qd
  else
    x -= qd;
```

Если горизонтальная координата конечной точки больше горизонтальной координаты точки начальной (то есть элемент движется слева направо), увеличиваем текущую горизонтальную координату элемента на величину приращения. В противном случае (если элемент движется справа налево) уменьшаем текущую горизонтальную координату элемента на эту же величину. Фактически здесь мы выполняем сам "шаг" анимации.

```
  animObj.style.left = x + "px";
```

Задаем новую горизонтальную координату анимированного элемента.

```
  animObj.style.top = y1 + k * Math.abs(x - x1) + "px";
```

Задаем новую вертикальную координату анимированного элемента, которую вычисляем, пользуясь полученным ранее отношением.

```
  if ((x <= xmin) || (x >= xmax)) qd = -qd;
}
```

Проверяем, не достиг ли анимированный элемент начальной или конечной точки, и, если так, меняем знак приращения. Отметим, что здесь мы пользуемся полученными ранее величинами минимальной и максимальной горизонтальных координат. Если бы мы использовали непосредственно горизонтальные координаты начальной и конечной точек, это выражение получилось бы намного более сложным.

Это последнее выражение тела функции `doStep`, выполняющей очередной "шаг" анимации.

Приведенный ранее код весьма универсален. Мы можем менять анимацию как угодно, просто задавая новые координаты начальной и конечной точек. Единственное — этот код не будет работать в случае движения элемента строго по вертикальной линии, так как при вычислении выражения

```
var k = Math.abs((y2 - y1) / (x2 - x1));
```

возникнет ошибка деления на ноль (поскольку вертикальные координаты начальной и конечной точек будут равными).

Если вам нужно перемещать анимированный элемент по вертикальной линии, вы можете написать свой сценарий, взяв за основу приведенный ранее. Пусть это будет вашим домашним заданием.

А пока что рассмотрим еще один пример страницы с анимированным элементом. Это также будет графические изображение, но двигаться оно теперь будет по окружности радиусом 100 пикселов, центр которой находится в точке с координатами {200, 200}.

```
<HTML>
  <HEAD>
    <TITLE>Анимация</TITLE>
    <STYLE>
      #anim { position: absolute;
              left: 100px;
              top: 100px;
              width: 20px;
              height: 20px}
    </STYLE>
  </HEAD>
  <BODY>
    <DIV ID="anim"><IMG SRC="image.gif"></DIV>
    <SCRIPT>
      var x = 200;
      var y = 200;
```

```
      var qd = .2;
      var R = 100;
      var int = 100;

      var rr = Math.PI * 2;
      var a = 0;

      var animObj = document.getElementById("anim");

      function doStep() {
        a += qd;
        animObj.style.left = x + R * Math.sin(a) + "px";
        animObj.style.top = y + R * Math.cos(a) + "px";
        if (a >= rr) a = 0;
        }

      animObj.style.left = x + "px";
      animObj.style.top = y + R + "px";

      window.setInterval(doStep, int);
    </SCRIPT>
  </BODY>
</HTML>
```

Этот пример много проще предыдущего. Рассмотрим его построчно.

```
var x = 200;
var y = 200;
var qd = .2;
var R = 100;
var int = 100;
```

Задаем координаты центра окружности, по которой будет двигаться анимированный элемент (переменные `x` и `y`), величину приращения в радианах (переменная `qd`), радиус окружности (переменная `R`) и тайм-аут (переменная `int`).

```
var rr = Math.PI * 2;
var a = 0;
```

Вычисляем значение 2π радиан — конечное значение угла (переменная `rr`). Также задаем начальный угол — 0 радиан (переменная `a`).

```
var animObj = document.getElementById("anim");
```

Получаем доступ к анимированному элементу.

```
animObj.style.left = x + "px";
animObj.style.top = y + R + "px";
```

Устанавливаем анимированный элемент в начальную точку анимации — в самый низ окружности.

```
window.setInterval(doStep, int);
```

Запускаем отсчет тайм-аута. В качестве функции, вызываемой после каждого его окончания, задаем функцию `doStep`, которую рассмотрим далее. Идентификатор таймера мы также не сохраняем.

Рассмотрим функцию `doStep`. Она будет очень простой.

```
function doStep() {
  a += qd;
```

Увеличиваем текущий угол на величину приращения.

```
  animObj.style.left = x + R * Math.sin(a) + "px";
  animObj.style.top = y + R * Math.cos(a) + "px";
```

Задаем новые координаты анимированного элемента.

```
  if (a >= rr) a = 0;
}
```

Проверяем, не превысило ли значение текущего угла предельное значение, полученное нами ранее, и, если так, задаем начальное значение угла — 0 радиан.

Анимация по ключевым точкам

Всем хорош рассмотренный нами способ создания анимации. Кроме одного — далеко не всякую траекторию можно описать относительно простой функцией. Так, если мы захотим, чтобы анимированный элемент на нашей странице двигался по ломаной линии, нам придется поломать голову.

Но выход из этого тупика есть. (Вообще, из всякого тупика есть выход — его нужно только поискать.) И сейчас мы его рассмотрим.

Все мы в школе изучали математику и знаем, что любую линию — прямую, кривую или ломаную — можно с заданной степенью точности описать минимальным набором точек, называемых *ключевыми*. Давайте нарисуем на бумажке предполагаемую траекторию движения анимированного элемента, расставим на ней ключевые точки и пронумеруем их. То, что у нас получится, показано на рис. 10.4.

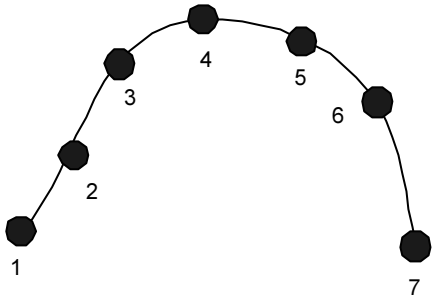

Рис. 10.4. Траектория, описанная набором ключевых точек (обозначены черными кружками)

Не забываем при этом, что ключевых точек должно быть ровно столько, чтобы описать нашу траекторию более-менее точно — но не больше, иначе их количество превысит все разумные пределы. Также обратим внимание, что в число ключевых точек входят и начальная и конечная точки анимации — ведь без них полностью описать траекторию не получится.

После этого проведем координатные оси — горизонтальную x и вертикальную y — и проградуируем их в пикселах (можно в миллиметрах или сантиметрах; просто пикселы в нашем, "компьютерном", случае привычнее). Пронумеруем все ключевые точки и выпишем на отдельную бумажку их координаты. В результате мы получим набор, или, как говорят программисты, *массив ключевых точек*.

Теперь проведем прямую линию и назовем ее *дорожкой анимации*. Эта дорожка будет соответствовать траектории анимации, но не в пространстве, а во времени. После этого расставим в нужных местах дорожки анимации ключевые точки (рис. 10.5).

Рис. 10.5. Дорожка анимации с набором ключевых точек (обозначены черными кружками)

Понятно, что мы сделали? Каждая ключевая точка на дорожке анимации задает момент времени, когда анимированный элемент должен находиться в заданных координатах. А координаты эти мы определили на предыдущем графике (см. рис. 10.4).

Нам остается проградуировать дорожку анимации, скажем, в секундах и выписать на бумажку с координатами ключевых точек соответствующие значения времени. В результате мы получим полный массив ключевых точек. Этот массив мы используем для создания анимации. Это будет *анимация по ключевым точкам*.

Для создания анимации по ключевым точкам используется другая функция траектории, имеющая такой вид:

```
{x, y, z} = f([p1, t1, p2, t2...], q)
```

Она принимает всего два параметра: созданный нами массив ключевых точек и, разумеется, текущую позицию. Длина траектории и приращение будут рассчитываться внутри этой функции на основании данных, хранящихся в массиве ключевых точек.

Вот алгоритм работы этой функции траектории.

1. Принять начальные параметры и выполнить предварительные установки (установить анимированный элемент в начальную точку траектории и запустить первый отсчет тайм-аута).
2. Когда тайм-аут закончится, проверить, дошел ли элемент до конца траектории. Если да, перейти к пункту 8.
3. Выяснить координаты следующей ключевой точки и соответствующее ей значение времени.
4. Проверить, совпадает ли текущее время со значением времени, соответствующим следующей ключевой точке, то есть дошел ли до нее анимированный элемент. Если да — перейти к пункту 7.
5. На основании координат следующей ключевой точки и ее значения времени вычислить приращение, определить значения координат анимированного элемента и переместить этот элемент в точку с вычисленными координатами.
6. Отметить текущее время.
7. Завершить свою работу и ждать очередного "тика" таймера. При его наступлении перейти к пункту 2.
8. Выполнить завершающие действия (в частности, остановить отсчет времени).

И, как обычно, рассмотрим пример страницы с анимированным элементом. Это опять будет графическое изображение, которое на этот раз будет двигаться по треугольной траектории.

```
<HTML>
  <HEAD>
    <TITLE>Анимация</TITLE>
```

```
  <STYLE>
    #anim { position: absolute;
            left: 100px;
            top: 100px;
            width: 20px;
            height: 20px}
  </STYLE>
</HEAD>
<BODY>
  <DIV ID="anim"><IMG SRC="image.gif"></DIV>
  <SCRIPT>
    var coords = [];
    coords[0] = [0, 100, 0];
    coords[1] = [0, 300, 2];
    coords[2] = [300, 0, 3];
    coords[3] = [0, 100, 0];
    var int = 100;

    var phaseNum = 0;
    var time = 0;

    var animObj = document.getElementById("anim");

    function doStep() {
      if ((phaseNum == 0) && (time == 0)) {
        animObj.style.left = coords[0][0] + "px";
        animObj.style.top = coords[0][1] + "px";
      }
      if (phaseNum < coords.length - 1) {
        var x1 = coords[phaseNum][0];
        var y1 = coords[phaseNum][1];
        var x2 = coords[phaseNum + 1][0];
        var y2 = coords[phaseNum + 1][1];
        var time1 = (phaseNum == 0) ? 0 : coords[phaseNum][2] * 1000;
        var time2 = coords[phaseNum + 1][2] * 1000;
        if (time > time2) {
          time = time2;
          phaseNum++;
        } else {
```

```
            if (x1 == x2) {
              var yd = Math.abs((time - time1) * (y2 - y1) /
              (time2 - time1));
              if (y1 > y2) yd = -yd;
              animObj.style.top = y1 + yd + "px";
            } else {
              var xd = Math.abs((time - time1) * (x2 - x1) /
              (time2 - time1));
              if (x1 > x2) xd = -xd;
              var k = Math.abs((y2 - y1) / (x2 - x1));
              if (y1 > y2) k = -k;
              var x = x1 + xd;
              animObj.style.left = x + "px";
              animObj.style.top = y1 + k * Math.abs(x - x1) + "px";
            }
            time += int;
          }
        } else {
          phaseNum = 0;
          time = 0;
        }
      }

      window.setInterval(doStep, int);
    </SCRIPT>
  </BODY>
</HTML>
```

Рассмотрим сценарий, реализующий анимацию, построчно.

```
var coords = [];
coords[0] = [0, 100, 0];
coords[1] = [0, 300, 2];
coords[2] = [300, 0, 3];
coords[3] = [0, 100, 0];
```

Для хранения координат ключевых точек используем массив `coords`, элементы которого также являются массивами. Каждый из этих вложенных массивов имеет три элемента:

- горизонтальную координату ключевой точки;
- вертикальную координату ключевой точки;

- отметка времени, когда анимированный элемент должен находиться в этой ключевой точке, в виде секунд, прошедших с начала анимации (секунды для нас привычнее, нежели миллисекунды).

```
var int = 100;
```

В переменной `int` сохраним значение тайм-аута.

```
var phaseNum = 0;
var time = 0;
```

Переменная `phaseNum` будет хранить номер уже пройденной элементом ключевой точки, а переменная `time` — текущее время.

```
var animObj = document.getElementById("anim");
```

Получаем доступ к анимированному элементу.

```
window.setInterval(doStep, int);
```

И запускаем отсчет тайм-аута.

Теперь рассмотрим функцию `doStep`, которая у нас традиционно занимается выполнением "шага" анимации.

```
function doStep() {
  if ((phaseNum == 0) && (time == 0)) {
    animObj.style.left = coords[0][0] + "px";
    animObj.style.top = coords[0][1] + "px";
  }
```

Если анимация только начинается (элемент прошел или проходит первую ключевую точку, имеющую номер 0, и значение текущего времени равно 0), принудительно устанавливаем анимированный элемент в первую ключевую точку.

```
  if (phaseNum < coords.length - 1) {
```

Проверяем, не дошла ли анимация до конца, то есть не перевалил ли номер ключевой точки, пройденной элементом, за общее количество ключевых точек (элементов массива `coords`). Если нет, выполняем следующий, довольно "увесистый", блок кода.

```
    var x1 = coords[phaseNum][0];
    var y1 = coords[phaseNum][1];
    var x2 = coords[phaseNum + 1][0];
    var y2 = coords[phaseNum + 1][1];
    var time1 = (phaseNum == 0) ? 0 : coords[phaseNum][2] * 1000;
    var time2 = coords[phaseNum + 1][2] * 1000;
```

Определяем значения координат и времени, соответствующих уже пройденной и следующей по счету ключевых точек. Они нужны нам, чтобы вычислить приращение.

```
if (time > time2) {
  time = time2;
  phaseNum++;
```

Если значение текущего времени превысило значение времени, соответствующее следующей ключевой точке, используем последнее как текущее время, считаем следующую ключевую точку пройденной, инкрементировав номер пройденной ключевой точки, и завершаем работу функции `doStep`.

```
} else {
```

В противном случае выполняем следующий код.

```
  if (x1 == x2) {
    var yd = Math.abs((time - time1) * (y2 - y1) /
    (time2 - time1));
    if (y1 > y2) yd = -yd;
    animObj.style.top = y1 + yd + "px";
```

Если в данный момент элемент движется строго по вертикали (горизонтальные координаты пройденной и следующей ключевых точек равны), вычисляем приращение движения по вертикали, новую вертикальную координату элемента (горизонтальная координата не меняется) и перемещаем элемент на новую позицию. Обратим внимание на выражение, вычисляющее приращение (переменная `yd`), — оно использует как значения вертикальных координат предыдущей и следующей ключевых точек, так и значения соответствующего им времени.

```
  } else {
    var xd = Math.abs((time - time1) * (x2 - x1) /
    (time2 - time1));
    if (x1 > x2) xd = -xd;
    var k = Math.abs((y2 - y1) / (x2 - x1));
    if (y1 > y2) k = -k;
    var x = x1 + xd;
    animObj.style.left = x + "px";
    animObj.style.top = y1 + k * Math.abs(x - x1) + "px";
  }
```

Если движение не строго вертикальное, работы у нас прибавляется. Хотя в данном случае комментировать особо нечего — этот код практически аналогичен приведенному в примере страницы с анимированным элементом, движущемся по наклонной линии. За исключением выражения, вычисляющего

приращение (переменная `xd`), и выражений, задающих новые координаты анимированного элемента — в этом случае меняются обе координаты.

```
      time += int;
    }
```

Не забываем увеличить значение текущего времени на значение тайм-аута. И завершаем работу функции.

```
  } else {
    phaseNum = 0;
    time = 0;
  }
}
```

Если же анимация дошла до конца (до последней ключевой точки), устанавливаем номер пройденной элементом ключевой точки и текущее время в 0. И анимация начнет воспроизводиться с начала.

Написанный нами сценарий весьма универсален. Мы можем создавать с его помощью любую анимацию, просто задавая в массиве `coords` параметры нужных нам ключевых точек. Вообще, нужно стараться писать универсальный код.

> ***Внимание!***
>
> Анимация на Web-страницах сейчас считается не очень хорошим стилем Web-дизайна. Поэтому используйте ее только в тех случаях, когда без нее действительно не обойтись.

Drag'n'drop

Drag'n'drop очень часто используется в Web-приложениях (программах, реализованных в виде Web-страницы); на обычных страницах он бесполезен. Так что давайте рассмотрим его реализацию — пригодится.

Что вообще такое drag'n'drop? Мы устанавливаем курсор мыши на какой-то элемент страницы, нажимаем левую кнопку и, не отпуская ее, перемещаем курсор, а элемент страницы следует за ним. Установив курсор мыши на нужное место, мы отпускаем кнопку мыши, и элемент страницы остается там. То же самое, что и в обычных Windows-приложениях.

Давайте перечислим все проблемы, которые нам придется решить, чтобы реализовать полноценный drag'n'drop.

- Отслеживать нажатие и отпускание кнопки мыши (как правило, левой) и, соответственно, запускать процесс перемещения элемента страницы и останавливать его.

- Отслеживать перемещение курсора мыши и перемещать элемент страницы вслед за ним.

Ничего сложного здесь нет. Понятно, что элемент страницы, поддерживающий drag'n'drop, должен быть свободно позиционируемым. Также понятно, что для реализации drag'n'drop следует обрабатывать события мыши: `onMouseDown`, `onMouseMove` и `onMouseUp`. Обработчики этих событий будут определять координаты курсора мыши и перемещать вслед за ним элемент страницы.

> ***НА ЗАМЕТКУ***
>
> Можно также реализовать drag'n'drop для непозиционируемых элементов страницы. В этом случае обработчики упомянутых ранее событий будут удалять перемещаемый элемент из предыдущей позиции HTML-кода страницы и вставлять его на новую. Но это много сложнее и поэтому редко применяется на практике.

Мы можем создать более сложную реализацию drag'n'drop. Так, мы можем предусмотреть возможность перемещения элемента строго в определенное место страницы, например, на определенный ее элемент. Также мы можем запретить перемещение элемента страницы на некоторые ее места. В обоих этих случаях нам придется хранить старые координаты элемента, чтобы при попытке переместить его на недопустимую позицию вернуть его назад. Также мы можем каким-то образом давать понять посетителю, что он пытается переместить элемент страницы на недопустимую или, наоборот, разрешенную позицию; это можно сделать, например, изменением цвета фона перемещаемого элемента страницы.

Рассмотрим страницу со свободно позиционируемым фрагментом текста, который можно перемещать мышью на любое место этой страницы. Это самая простая реализация drag'n'drop.

```
<HTML>
  <HEAD>
    <TITLE>Drag'n'drop</TITLE>
    <STYLE>
      #dnd { position: absolute;
             left: 100px;
             top: 100px;
             width: 100px;
             height: 100px }
    </STYLE>
    <SCRIPT>
      function getBrowser() {
```

```
  switch (navigator.appName) {
    case "Microsoft Internet Explorer":
      return "IE";
      break;
    case "Opera":
      return "O";
      break;
    case "Netscape":
      return "FF";
  }
}

function getBrowserStrict() {
  if (navigator.userAgent.indexOf("Opera") > -1)
    return "O"
  else
    return getBrowser();
}

var isDragging = false;
var xd = 0;
var yd = 0;

function dndMouseDownIEOpera() {
  isDragging = true;
  xd = event.offsetX;
  yd = event.offsetY;
}

function dndMouseDownFF(evt) {
  isDragging = true;
  xd = evt.layerX;
  yd = evt.layerY;
}

function bodyMouseMoveIEOpera() {
  if (isDragging) {
    dndObj.style.left = event.clientX - xd + "px";
    dndObj.style.top = event.clientY - yd + "px";
  }
```

```
      }

      function bodyMouseMoveFF(evt) {
        if (isDragging) {
          dndObj.style.left = evt.clientX - xd + "px";
          dndObj.style.top = evt.clientY - yd + "px";
        }
      }

      function dndMouseUp() {
        isDragging = false;
      }
    </SCRIPT>
  </HEAD>
  <BODY>
    <DIV ID="dnd">Этот элемент страницы можно перетаскивать мышью.</DIV>
    <SCRIPT>
      var dndObj = document.getElementById("dnd");
      if (getBrowserStrict() == "FF") {
        dndObj.onmousedown = dndMouseDownFF;
        document.body.onmousemove = bodyMouseMoveFF;
      } else {
        dndObj.onmousedown = dndMouseDownIEOpera;
        document.body.onmousemove = bodyMouseMoveIEOpera;
      }
      dndObj.onmouseup = dndMouseUp;
    </SCRIPT>
  </BODY>
</HTML>
```

Давайте рассмотрим код сценария, реализующего drag'n'drop. Хотя рассматривать тут особо нечего.

Прежде всего, отметим, что для выяснения текущей позиции курсора нам придется использовать свойства объекта `Event`, которые у разных Web-обозревателей различаются. Поэтому нам понадобятся написанные еще в *главе 7* функции `getBrowser` и `getBrowserStrict`, позволяющие выяснить, в каком Web-обозревателе открыта страница.

```
var isDragging = false;
var xd = 0;
var yd = 0;
```

Переменная `isDragging` будет хранить признак того, выполняется в данный момент перемещение элемента страницы (значение `true`) или нет (значение `false`). А переменные `xd` и `yd` будут хранить координаты курсора мыши относительно перемещаемого элемента — горизонтальную и вертикальную соответственно.

```
var dndObj = document.getElementById("dnd");
if (getBrowserStrict() == "FF") {
  dndObj.onmousedown = dndMouseDownFF;
  document.body.onmousemove = bodyMouseMoveFF;
} else {
  dndObj.onmousedown = dndMouseDownIEOpera;
  document.body.onmousemove = bodyMouseMoveIEOpera;
}
dndObj.onmouseup = dndMouseUp;
```

Выполняем привязку обработчиков к событиям.

- К событию `onMouseDown` перемещаемого элемента `dnd` привязываем функцию `dndMouseDownFF` в случае Firefox и `dndMouseDownIEOpera` в случае Internet Explorer и Opera.
- К событию `onMouseMove` секции тела страницы привязываем функцию `dndMouseMoveFF` в случае Firefox и `dndMouseMoveIEOpera` в случае Internet Explorer и Opera.
- К событию `onMouseUp` перемещаемого элемента `dnd` привязываем функцию `dndMouseUp`.

```
function dndMouseDownIEOpera() {
  isDragging = true;
  xd = event.offsetX;
  yd = event.offsetY;
}
function dndMouseDownFF(evt) {
  isDragging = true;
  xd = evt.layerX;
  yd = evt.layerY;
}
```

Функции — обработчики события `onMouseDown` присваивают переменной `isDraggind` значение `true`, сигнализируя, что процесс drag'n'drop начался. Также они определяют положение курсора мыши относительно перемещаемого элемента.

```
function bodyMouseMoveIEOpera() {
  if (isDragging) {
```

```
    dndObj.style.left = event.clientX - xd + "px";
    dndObj.style.top = event.clientY - yd + "px";
  }
}
function bodyMouseMoveFF(evt) {
  if (isDragging) {
    dndObj.style.left = evt.clientX - xd + "px";
    dndObj.style.top = evt.clientY - yd + "px";
  }
}
```

Функции — обработчики события `onMouseMove` перемещают элемент `dnd`, если процесс drag'n'drop начат (переменная `isDragging` хранит значение `true`).

```
function dndMouseUp() {
  isDragging = false;
}
```

Функция — обработчик события `onMouseUp` самая простая. Она останавливает процесс drag'n'drop, присваивая переменной `isDragging` значение `false`.

А вот пример посложнее — страница со свободно позиционируемым фрагментом текста, который можно перетаскивать только на строго определенные места. Такими местами являются два других свободных элемента ("приемника"). Когда перемещаемый элемент находится над одним их этих элементов-"приемников", цвет его фона меняется, сигнализируя о том, что операция drag'n'drop разрешена. Если же посетитель попытается перетащить перемещаемый элемент на любое другое место страницы, он вернется на старое место.

```
<HTML>
  <HEAD>
    <TITLE>Drag'n'drop</TITLE>
    <SCRIPT>
      function getBrowser() {
        switch (navigator.appName) {
          case "Microsoft Internet Explorer":
            return "IE";
            break;
          case "Opera":
            return "O";
            break;
          case "Netscape":
```

```
      return "FF";
  }
}

function getBrowserStrict() {
  if (navigator.userAgent.indexOf("Opera") > -1)
    return "O"
  else
    return getBrowser();
}

var isDragging = false;
var xd = 0;
var yd = 0;
var xbegin = 0;
var ybegin = 0;

function isReceived(pReceiver) {
  return ((dndObj.offsetLeft >= pReceiver.offsetLeft) &&
  (dndObj.offsetLeft + dndObj.offsetWidth <=
  pReceiver.offsetLeft + pReceiver.offsetWidth) &&
  (dndObj.offsetTop >= pReceiver.offsetTop) &&
  (dndObj.offsetTop + dndObj.offsetHeight <=
  pReceiver.offsetTop + pReceiver.offsetHeight));
}

function dndMouseDownIEOpera() {
  isDragging = true;
  xd = event.offsetX;
  yd = event.offsetY;
  xbegin = dndObj.offsetLeft;
  ybegin = dndObj.offsetTop;
}

function dndMouseDownFF(evt) {
  isDragging = true;
  xd = evt.layerX;
  yd = evt.layerY;
  xbegin = dndObj.offsetLeft;
```

```
    ybegin = dndObj.offsetTop;
  }

  function bodyMouseMoveIEOpera() {
    if (isDragging) {
      dndObj.style.left = event.clientX - xd + "px";
      dndObj.style.top = event.clientY - yd + "px";
      if ((isReceived(receiver1Obj)) || (isReceived(receiver2Obj)))
        dndObj.style.backgroundColor = "#FF0000"
      else
        dndObj.style.backgroundColor = "#CCCCCC";
    }
  }

  function bodyMouseMoveFF(evt) {
    if (isDragging) {
      dndObj.style.left = evt.clientX - xd + "px";
      dndObj.style.top = evt.clientY - yd + "px";
      if ((isReceived(receiver1Obj)) || (isReceived(receiver2Obj)))
        dndObj.style.backgroundColor = "#FF0000"
      else
        dndObj.style.backgroundColor = "#CCCCCC";
    }
  }

  function dndMouseUp() {
    isDragging = false;
    dndObj.style.backgroundColor = "#CCCCCC";
    if (!((isReceived(receiver1Obj)) || (isReceived(receiver2Obj))))
    {
      dndObj.style.left = xbegin + "px";
      dndObj.style.top = ybegin + "px";
    }
  }
</SCRIPT>
<STYLE>
  #receiver1 { position: absolute;
               left: 100px;
               top: 300px;
```

```
                width: 200px;
                height: 200px;
                background-color: #666666;
                color: #FFFFFF; }
    #receiver2 { position: absolute;
                left: 400px;
                top: 300px;
                width: 200px;
                height: 200px;
                background-color: #666666;
                color: #FFFFFF; }
    #dnd       { position: absolute;
                left: 100px;
                top: 100px;
                width: 100px;
                height: 100px;
                background-color: #CCCCCC; }
  </STYLE>
</HEAD>
<BODY>
  <DIV ID="receiver1">Тащите его сюда!</DIV>
  <DIV ID="receiver2">Тащите его сюда!</DIV>
  <DIV ID="dnd">Этот элемент страницы можно перетаскивать мышью.</DIV>
  <SCRIPT>
    var dndObj = document.getElementById("dnd");
    var receiver1Obj = document.getElementById("receiver1");
    var receiver2Obj = document.getElementById("receiver2");
    if (getBrowserStrict() == "FF") {
      dndObj.onmousedown = dndMouseDownFF;
      document.body.onmousemove = bodyMouseMoveFF;
    } else {
      dndObj.onmousedown = dndMouseDownIEOpera;
      document.body.onmousemove = bodyMouseMoveIEOpera;
    }
    dndObj.onmouseup = dndMouseUp;
  </SCRIPT>
</BODY>
</HTML>
```

Код сценария, реализующего здесь drag'n'drop, практически аналогичен коду из предыдущего примера. В данном случае только добавилось сохранение координат, по которым перемещаемый элемент находился в начале процесса drag'n'drop, чтобы при попытке переместить его на недопустимое место (то есть вне элементов-"приемников") вернуть его обратно. Также добавилась проверка, на допустимое ли место перемещен элемент (функция `isReceived`). Так что вы можете разобраться с этим примером самостоятельно.

> ***НА ЗАМЕТКУ***
>
> Drag'n'drop, реализованный описанным здесь способом с помощью обработки событий мыши, не всегда работает корректно. Так, Internet Explorer и Firefox некорректно обрабатывают перемещение элементов, содержащих графические изображения, а Opera — перемещение элементов, содержащих текст (как в нашем случае). Так что в любом случае drag'n'drop на Web-страницах лучше не увлекаться.
>
> В *главе 13* будет рассмотрена реализация drag'n'drop с использованием специфических средств, поддерживаемых Internet Explorer. Opera и Firefox, к сожалению, их не поддерживают (хотя Firefox 3, как утверждают его разработчики, будет поддерживать).

Что дальше?

Вот мы и закончили со свободно позиционируемыми элементами. Сколько мы всего узнали! Теперь создать всплывающие подсказки, анимацию или реализовать drag'n'drop для нас не проблема.

В следующей главе мы начнем более тесное "общение" с посетителем. А именно научимся выводить и вводить различные данные, сохранять данные на клиентском компьютере и передавать их между страницами. Кроме того, мы познакомимся с очень мощным средством обработки данных — так называемыми регулярными выражениями. Так сказать, изучим более сложные фигуры высшего пилотажа Web-программирования.

Глава 11

Работа с данными

В предыдущих главах мы были заняты, в основном, работой с содержимым страницы, загруженной в Web-обозреватель, и самим Web-обозревателем. Надо сказать, что большинство сценариев, которые сейчас пишут Web-программисты, именно этим и занимаются.

Большинство, но не все.

Часто бывает нужно принять от посетителя какую-то информацию, вывести результат ее обработки, а то и сохранить какие-то данные на диске клиентского компьютера либо передать их другой странице. Конечно, такая нужда (особенно сохранение данных) возникает нечасто, но все же может наступить момент, когда нам это понадобится. Что мы тогда будем делать?!

Поэтому давайте рассмотрим средства Web-обозревателей, позволяющие нам:

- принять от посетителя данные;
- вывести данные на экран;
- сохранить данные на диске клиентского компьютера;
- передать данные другой странице;
- использовать для обработки данных регулярные выражения — исключительно мощный, хоть и довольно сложный в освоении инструмент.

Материал немалый, и его описанию будет посвящена вся эта глава.

Вывод данных

Начнем мы с вывода данных, так как его реализовать проще всего. Здесь Web-обозреватели предлагают нам целых три способа.

Первый способ — использование для вывода данных элементы самой Web-страницы. Вспомним, как мы в предыдущих главах использовали для вывода

какой-либо информации, скажем, текста подсказки, абзацы и свободные элементы с неизменным именем `output`! Это, кстати, наиболее часто используемый способ вывода данных; применяя его, мы можем точно указать, в какое место страницы вывести эти данные и как их оформить.

Вывод данных в строке статуса

Второй способ — использование для вывода данных строки статуса окна Web-обозревателя. Строка статуса находится в самом низу окна и имеет вид серой панели, разделенной на секции, в которые может выводиться разнородная информация. Мы можем вывести любой текст в крайнюю левую ее панель.

За вывод текста в строку статуса "отвечают" два свойства объекта `Window`. Сейчас мы их рассмотрим.

Свойство `status` задает или возвращает текст, который в данный момент присутствует в крайней левой панели строки статуса. Это значит, что для того, чтобы вывести нужный текст в строку статуса, его нужно присвоить этому свойству.

```
window.status = "Привет всем!";
```

Это выражение выведет в строку статуса текущего окна Web-обозревателя текст "Привет всем!".

```
secondWnd.status = "Привет от первичного окна!";
```

А это выражение выведет в строку статуса вторичного окна, представляемого экземпляром объекта `Window`, который хранится в переменной `secondWnd`, текст "Привет от первичного окна!". О создании новых окон Web-обозревателя см. *главу 7*.

Свойство `defaultStatus` задает или возвращает строку, которая отображается в строке статуса по умолчанию. Оно пригодится, если мы захотим убрать свой текст из строки статуса, заменив его тем, что выводится Web-обозревателем.

```
window.status = window.defaultStatus;
```

Мы можем присвоить свойству `defaultStatus` любую строку, и тогда эта строка будет выводиться в строку статуса по умолчанию. Но делать этого не рекомендуется по причинам, изложенным в следующем абзаце.

> ***Внимание!***
>
> Произвольный текст, присутствующий в строке статуса, — очень плохой стиль Web-дизайна. В настоящее время считается, что строка статуса — полная "собственность" Web-обозревателя, куда он выводит различные сообщения

о ходе работы. Кроме того, многие современные Web-обозреватели позволяют блокировать исполнение сценариев, выводящих в строку статуса произвольный текст, — это нужно иметь в виду.

Вывод данных в окнах-сообщениях

Окно-сообщение — это небольшое окно с произвольным текстом, иконкой восклицательного знака и кнопкой **OK**. Это одно из стандартных окон Windows, знакомое нам по другим Windows-приложениям.

Чтобы вывести на экран окно-сообщение с заданным текстом, нужно воспользоваться методом `alert` объекта `Window`.

```
alert(<выводимый текст>)
```

Видно, что требуемый текст передается этому методу в качестве единственного параметра. Значения метод `alert` не возвращает.

```
window.alert("Всем привет!");
```

Это выражение выводит окно-сообщение с текстом "Всем привет!".

```
var d = new Date();
window.alert(d.toString());
```

А это выражение выводит на экран окно-сообщение со значением текущих даты и времени.

Вывод данных в окнах-сообщениях обычно применяется в двух случаях. Во-первых, это вывод критически важных данных. (Не критически важные данные лучше выводить прямо на страницу или, что, правда, не рекомендуется, в строку статуса.) При этом критически важные данные — это те данные, которые посетитель обязательно должен прочитать.

Во-вторых, окна-сообщения применяются при отладке сценариев, точнее, для выяснения значения какой-либо переменной или свойства в заданном месте сценария. Второй пример, представленный в этом параграфе, кстати, это и демонстрирует.

Ввод данных

Покончив с выводом данных, займемся их вводом. Здесь Web-обозреватели предлагают нам два способа. Первый способ — использование Web-форм — будет описан в *главе 13*, поскольку это слишком большой материал, чтобы рассмотреть его в небольшом параграфе. А вот второй способ — ввод с помощью специальных окон — мы рассмотрим прямо сейчас.

Прежде всего, мы можем указать Web-обозревателю вывести окно-сообщение с произвольным текстом, иконкой вопросительного знака и кнопками **OK**

и **Отмена** (Cancel). Посетитель может нажать одну из этих кнопок, а мы можем в зависимости от того, какая кнопка была нажата, выполнить какое-то действие. Это выполняется с помощью метода `confirm` объекта `Window`.

```
confirm(<выводимый текст>)
```

Текст, который должен присутствовать в окне-сообщении, передается этому методу в качестве единственного параметра. Метод `confirm` возвращает `true`, если посетитель нажал кнопку **OK**, и `false` в случае нажатия кнопки **Отмена** (Cancel).

```
var flag = window.confirm("Выполнить это действие?");
```

Это выражение выведет на экран окно-сообщение с кнопками и текстом "Выполнить это действие?" и поместит в переменную `flag` значение `true` или `false` в зависимости от нажатой посетителем кнопки этого окна.

Метод `prompt` объекта `Window` выводит на экран небольшое окно с полем ввода (*окно ввода*), в которое посетитель может ввести нужное значение. Для подтверждения ввода он должен будет нажать кнопку **OK** этого окна, для отмены сделанного ввода — кнопку **Отмена** (Cancel).

```
prompt(<текст сообщения>[, <изначальное значение>])
```

Первый параметр данного метода задает текст сообщения, который будет присутствовать в этом окне; это может быть требование ввести данные в определенном формате. Второй — необязательный — параметр задает значение, которое изначально будет присутствовать в поле ввода окна; если он опущен, в поле ввода будет занесено значение `undefined`.

> ***Внимание!***
>
> Следует всегда задавать изначальное значение для окна ввода, хотя бы пустую строку. В самом деле, если посетитель увидит в его поле ввода `undefined`, то будет немало обескуражен.

Если посетитель введет в поле ввода данного окна строковое значение и нажмет кнопку **OK**, метод `prompt` вернет введенное значение в виде строки (что понятно). Если посетитель введет число и нажмет **OK**, метод `prompt` вернет его в виде числового значения. Если же посетитель откажется от ввода данных, нажав кнопку **Отмена** (Cancel), метод `prompt` вернет значение `null`, даже если в поле ввода было что-то введено.

```
var input = window.alert("Введите строку или число", "");
if (typeof(input) == "string") {
  //Если введена строка, выполняем какие-то действия
} else {
  if (typeof(input) == "number") {
```

```
    //Если введено число, выполняем другие действия
  } else {
    //Если ничего не было введено, делаем что-то третье
  }
}
```

Приведенный ранее сценарий слишком отвлеченный, поэтому давайте рассмотрим страницу, которая будет запрашивать у посетителя имя с помощью окна ввода и в случае успешного ввода отображать его на странице.

```
<HTML>
  <HEAD>
    <TITLE>Ввод данных</TITLE>
  </HEAD>
  <BODY>
    <H1 ID="output"> </H1>
    <SCRIPT>
      var input = window.prompt("Введите свое имя", "");
      if (typeof(input) == "string") {
        var outputObj = document.getElementById("output");
        outputObj.firstChild.nodeValue = "Привет, " + input + "!";
      }
    </SCRIPT>
  </BODY>
</HTML>
```

Здесь для проверки, было ли что-то введено в окно ввода, мы используем описанный в *главе 4* оператор получения типа `typeof`. Если он вернет строку "string", значит, было введено строковое значение. Больше комментировать здесь нечего.

> ***На заметку***
>
> Ввод данных с помощью окон ввода применяется довольно редко. Как правило, от посетителя требуется ввести сразу несколько значений (например, имя и пароль), что удобнее выполнять с помощью Web-форм (см. *главу 12*).

Сохранение данных на клиентском компьютере

Возможность сохранения данных на клиентском компьютере сейчас не востребована. Однако современные Web-обозреватели все-таки ее предусматривают, как говорится, на всякий пожарный случай. Это использование так называемых cookies ("печенья"; единственное число — cookie).

Cookie — это небольшой фрагмент данных, сохраняемый Web-обозревателем на жестком диске клиентского компьютера (в отдельном текстовом файле, как Internet Explorer, или в одном большом файле, как Firefox). Этот cookie может содержать, в принципе, любые данные, которые можно преобразовать в строковый формат (числа, логические величины, дату и, разумеется, строки). Что нам и нужно.

Сохраненный cookie доступен всем Web-страницам, хранящимся в одной папке Web-сервера и всех вложенных в нее папках. Поиск нужного cookie при обращении к нему выполняется Web-обозревателем автоматически; нам об этом заботиться не придется.

При создании cookie мы можем указать дату, до которой он будет храниться на диске клиентского компьютера. "Просроченные" cookie будут автоматически удалены самим Web-обозревателем. Также мы можем не указывать дату хранения; в этом случае Web-обозреватель не будет сохранять cookie на диске — только в оперативной памяти (*временный cookie*). Разумеется, в этом случае после закрытия Web-обозревателя все хранимые в памяти cookie будут потеряны.

На cookie налагаются следующие ограничения:

- максимальный размер хранящихся в нем данных — 4 Кбайт;
- максимальное количество на один домен (о доменах см. *главу 1*) — 20;
- максимальное общее количество, хранимое Web-обозревателем, — 300.

Думается, этих ограничений нам хватит. Если, конечно, мы не собираемся запихивать в cookie что-то объемное…

Для создания cookie следует присвоить строку, содержащую представленные в особом формате данные, свойству `cookie` объекта `HTMLDocument`. Вот формат строки, присваиваемой этому свойству:

```
"<имя значения>=<само значение>[; expires=<дата хранения>]
[; path=<путь>][; domain=<домен>][; secure]"
```

`Имя значения` задает имя, под которым заданное `значение` будет храниться в cookie. Можно сказать, что в cookie для хранения каждого `значения` создается нечто вроде переменной с заданным `именем`.

А здесь нужно предупредить вот о чем. `Значение`, записываемое в cookie, должно быть закодировано в виде, применяемом для передачи данных в составе интернет-адреса (о передаче данных мы поговорим потом). Такое кодирование выполняется с помощью стандартной функции JavaScript `escape` (см. *главу 4*). Эта функция принимает единственный параметр — кодируемую строку — и возвращает ее в закодированном виде в качестве результата. Кстати, этот метод кодирования так и называется — *escape-кодирование*.

`Дата хранения` задает предельную дату, до которой cookie будет храниться на диске клиентского компьютера. Эта дата должна быть задана в строковом формате и по Гринвичу (*GMT*, Grinwitch Meridian Time — время по гринвичскому меридиану). Для получения такой строки с датой в формате GMT следует воспользоваться не принимающим параметров методом `toGMTString` объекта `Date` (также см. *главу 4*). Если `дата хранения` не задана, будет создан временный cookie (хранящийся в оперативной памяти компьютера).

Как уже говорилось, созданный в одной странице cookie доступен для всех страниц, хранящихся в той же папке Web-сервера и всех вложенных в нее папках. Если же мы хотим дать этому cookie доступ к страницам, хранящимся в другой папке Web-сервера (и вложенных в нее папках), то должны будем указать параметр `путь`, ведущий к нужной папке. Например, значение "/" этого параметра делает cookie доступным для всех страниц на данном Web-сервере (это значение как раз и обозначает корневую папку сайта). А значение "/chapters" делает cookie доступным для страниц, сохраненных в папке chapters корневой папки сайта и всех вложенных в нее папках.

Если мы хотим дать доступ к cookie страницам, хранящимся на других Web-серверах, то должны будем также указать параметр `домен`. Этот параметр должен представлять собой либо полное доменное имя Web-сервера без указания протокола (например, "www.othersite.ru" — тогда доступ к cookie получат страницы с Web-сервера **http://www.othersite.ru**), либо окончание доменного имени (".othersite.ru" — тогда доступ к cookie получат страницы с Web-серверов **http://www.othersite.ru**, **http://www2.othersite.ru**, **http://foreign.othersite.ru** и др., в общем, те, чье доменное имя оканчивается на "othersite.ru"). Заметим, что начальная точка в окончании доменного имени обязательна.

Параметр `secure` своим присутствием указывает на то, что для доступа к данным, хранящимся в этом cookie, нужно использовать защищенное соединение, т. е. протокол *HTTPS* (HyperText Transfer Protocol Secured, защищенный протокол передачи гипертекста). Этот параметр используется только в том случае, если cookie создается запросом от Web-сервера — в общем, не наш случай.

Мы можем поместить в cookie сколько угодно значений с заданными именами, просто присваивая содержащие их строки в приведенном ранее формате одну за другой свойству `cookie` объекта `HTMLDocument`. Все эти значения, сохраненные в cookie, будут существовать там, не "мешая" друг другу. Если одно из помещенных в cookie значений имеет то же имя, что и уже имеющееся в нем, новое значение заменит старое (но остальные значения останутся нетронутыми).

Удалить какое-либо значение из cookie, не затрагивая остальных, судя по всему, невозможно. Зато мы можем удалить cookie целиком. Для этого достаточно сохранить в нем какое-либо произвольное значение с именем, присутствующим в cookie, и в качестве даты хранения передать уже прошедшую дату, разумеется, в виде строки в формате GMT. Часто для этого используют строковое значение "Thu, 01-Jan-70 00:00:01 GMT", обозначающее одну секунду пополуночи первого января 1970 года (четверг). Получив такую дату, Web-обозреватель тут же удалит этот cookie.

> ***ВНИМАНИЕ!***
>
> Не забываем, что для удаления cookie нужно сохранить в нем новое значение под именем, уже присутствующим в cookie. То есть в cookie уже должно храниться значение с таким именем. В противном случае cookie не будет удален.

Чтобы получить из cookie сохраненные данные, достаточно, опять же, обратиться к свойству `cookie` объекта `HTMLDocument`. Оно вернет список всех сохраненных в cookie значений с их именами в виде пар *<имя>=<значение>*; эти пары будут отделены друг от друга точкой с запятой и пробелом.

Значения, возвращенные свойством `cookie`, будут закодированы с использованием escape-кодирования — мы ведь кодировали их функцией `escape` перед присвоением этому свойству. За декодирование строкового значения "отвечает" стандартная функция JavaScript `unescape`, принимающая кодированную строку в качестве единственного параметра и возвращающая ее в декодированном виде. Эта функция также была описана в *главе 4*.

> ***ВНИМАНИЕ!***
>
> При использовании cookie нужно иметь в виду три вещи. Во-первых, посетитель может отключить в настройках безопасности своего Web-обозревателя поддержку cookie. Во-вторых, даже если поддержка cookie в Web-обозревателе включена, cookie может быть заблокирован самим Web-обозревателем или какой-либо сторонней программой, работающей совместно с ним, как небезопасный. В-третьих, современные программы для удаления шпионского программного обеспечения (spyware) могут также удалить cookie, посчитав его вредоносным. (Хотя что такого может быть вредоносного в обычном фрагменте текста — непонятно...)

Работать напрямую со свойством `cookie` объекта `HTMLDocument` не очень удобно. Поэтому давайте напишем три функции, которые будут, соответственно, заносить в cookie заданное значение, извлекать его по имени и удалять cookie.

Функция `setCookie` заносит в `cookie` заданное значение под установленным именем. Если значение с таким именем уже существует, оно будет заменено новым.

```
setCookie(<имя значения>, <само значение>[, <дата хранения>
↳[, <путь>[, <домен>[, <защищенный доступ>]]]]);
```

Комментировать тут особо нечего — все параметры этой функции были рассмотрены ранее, в разговоре о свойстве `cookie` объекта `HTMLDocument`. `Дата хранения` указывается в виде экземпляра объекта `Date`. `Защищенный доступ` указывается в логическом формате: значение `true` требует защищенного соединения для доступа к данным cookie, значение `false` — не требует; пропуск этого параметра эквивалентен заданию значения `false`. Значения остальных параметров указываются в строковом формате, кроме `самого значения` — оно может быть задано в любом формате.

Значения функция `setCookie` не возвращает.

```
function setCookie(pName, pValue, pExpired, pPath, pDomain, pIsSecured) {
  var s = pName + "=" + escape(pValue.toString());
  if ((typeof(pExpired) != "undefined") && (typeof(pExpired) != "null"))
    s += "; expires=" + pExpired.toGMTString();
  if ((typeof(pPath) != "undefined") && (typeof(pPath) != "null"))
    s += "; path=" + pPath;
  if ((typeof(pDomain) != "undefined") && (typeof(pDomain) != "null"))
    s += "; domain=" + pDomain;
  if (pIsSecured)
    s += "; secured";
  document.cookie = s;
}
```

Здесь мы формируем строку на основании переданных функции параметров и присваиваем ее свойству `cookie`. При этом мы проверяем тип значений необязательных параметров: если он не равен "undefined" и "null", это значит, что параметр был передан, и мы включаем его значение в формируемую строку.

Функция `getCookie` извлекает из cookie значение с заданным именем.

```
getCookie(<имя значения>);
```

Она возвращает значение с заданным `именем` в виде строки, если оно существует в cookie; в противном случае возвращается `null`.

```
function getCookie(pName) {
  var sCookie = document.cookie;
  var sName = pName + "=";
  var i = sCookie.indexOf(sName);
  if (i > -1) {
    var j = sCookie.indexOf(";", i + sName.length);
    if (j == -1) j = sCookie.length;
    return unescape(sCookie.substring(i + sName.length, j));
```

```
  } else
    return null;
}
```

Здесь мы получаем строку, хранящуюся в свойстве `cookie`, и ищем в нем заданное `имя значения`. Если таковое находится, мы извлекаем фрагмент строки, находящийся между `именем` (плюс знак `=`) и ближайшим следующим символом точки с запятой; если такового символа больше нет (то есть это последнее значение в строке), мы извлекаем все следующие символы до конца строки. Извлеченный фрагмент возвращается как результат функции. Если же значения с заданным `именем` не найдено, возвращаем `null`.

Функция `deleteCookie` удаляет cookie.

```
deleteCookie(<имя любого уже присутствующего в cookie значения>
↳[, <путь>[, <домен>]]);
```

Параметры этой функции нам также знакомы. Значения она не возвращает.

```
function deleteCookie(pName, pPath, pDomain) {
  if (getCookie(pName)) {
    var s = pName + "=; expires=Thu, 01-Jan-70 00:00:01 GMT";
    if ((typeof(pPath) != "undefined") && (typeof(pPath) != "null"))
      s += "; path=" + pPath;
    if ((typeof(pDomain) != "undefined") && (typeof(pDomain) != "null"))
      s += "; domain=" + pDomain;
    document.cookie = s;
  }
}
```

Здесь мы сначала проверяем, присутствует ли значение с заданным `именем` в cookie (ведь для удаления cookie мы должны задать уже существующее `имя`), и, если присутствует, формируем строку, в которой указываем давно прошедшую дату хранения, и присваиваем эту строку свойству `cookie`. Все совсем просто.

В качестве примера рассмотрим страницу, которая считывает из cookie имя посетителя и выводит его на страницу. Если имя посетителя не сохранено в cookie, она запрашивает его у посетителя и сохраняет в cookie. Кроме того, на этой странице присутствует гиперссылка, при щелчке на которой cookie с сохраненным именем посетителя будет удален.

```
<HTML>
  <HEAD>
    <TITLE>Cookie</TITLE>
    <SCRIPT>
```

```
      //Здесь должен присутствовать код, объявляющий функции
      //setCookie, getCookie и deleteCookie
    </SCRIPT>
  </HEAD>
  <BODY>
    <H1 ID="output"> </H1>
    <P><A HREF="#" ONCLICK="deleteCookie('username');">Удалить
    сохраненное имя</A></P>
    <SCRIPT>
      var userName = getCookie("username");
      if (!(userName)) {
        userName = window.prompt("Введите свое имя", "");
        if (userName) {
          var expDate = new Date();
          var msNow = expDate.getTime();
          expDate.setTime(msNow + 1000 * 3600 * 24 * 7);
          setCookie("username", userName, expDate);
        }
      }
      if (userName) {
        var outputObj = document.getElementById("output");
        outputObj.firstChild.nodeValue = userName;
      }
    </SCRIPT>
  </BODY>
</HTML>
```

Давайте рассмотрим код сценария, реализующего запрос, сохранение и вывод имени посетителя построчно.

Прежде всего, в этой странице должен присутствовать код, объявляющий функции `setCookie`, `getCookie` и `deleteCookie`. Автор опустил его для экономии места, но вы должны его вставить в то место, где находится предписывающий сделать это комментарий (в секции заголовка страницы).

```
var userName = getCookie("username");
```

Пытаемся прочитать значение с именем `username`, сохраненное в cookie. Понятно, что `username` — это имя посетителя.

```
if (!(userName)) {
```

Если такового значения в cookie нет (функция `getCookie` вернула `null`), выполняем следующий блок кода.

Обратим внимание на условие в приведенном ранее выражении сравнения. Нам нужно проверить, содержится ли в переменной `userName` строка или значение `null`. В параграфе *главы 4*, посвященном преобразованию типов, сказано, что непустая строка преобразуется в значение `true`, а пустая строка и `null` — в `false`. Это мы и используем.

- Если в переменной `userName` хранится строка (имя посетителя было сохранено в cookie ранее), она преобразуется в значение `true`, которое оператор логического НЕ `!` превратит в `false`, и следующий блок кода, запрашивающий имя посетителя и сохраняющий его в cookie, выполнен не будет.
- Если в переменной `userName` хранится `null` (имя посетителя отсутствует в cookie), оно преобразуется в значение `false`, которое оператор логического НЕ `!` превратит в `true`. Следующий блок кода, запрашивающий имя посетителя и сохраняющий его в cookie, будет выполнен.

Собственно, подобные условные выражения уже были описаны в *главе 4*, в параграфе, посвященном условным выражениям. Но лучше рассказать об этом еще раз.

Настала пора рассмотреть этот блок кода.

```
userName = window.prompt("Введите свое имя", "");
```

Запрашиваем имя посетителя с помощью уже знакомого нам метода `prompt` объекта `Window`.

```
if (userName) {
```

Если посетитель его ввел, выполняем следующий блок кода. Снова обратим внимание на условие в выражении сравнения — оно работает на тех же принципах, что и описанное ранее.

```
    var expDate = new Date();
    var msNow = expDate.getTime();
    expDate.setTime(msNow + 1000 * 3600 * 24 * 7);
```

Вычисляем дату хранения cookie. Пусть это будет дата, отстоящая от текущей на семь дней. Чтобы ее получить, мы:

1. Создаем новый экземпляр объекта `Date`, содержащий текущую дату.
2. Получаем из него значение текущей даты в виде количества миллисекунд, прошедших с полуночи 1 января 1970 года (метод `getTime` объекта `Date`; подробнее см. *главу 4*).
3. Прибавляем к нему количество миллисекунд, соответствующее семи дням (1000 миллисекунд * 3600 секунд * 24 часа * 7 дней).

4. Помещаем полученную сумму в экземпляр объекта `Date` (метод `setDate` объекта `Date`). Требуемая дата хранения готова.

```
    setCookie("username", userName, expDate);
  }
}
```

Сохраняем имя посетителя в cookie под именем `username`. На этом блок кода, выполняющий запрос имени посетителя и его сохранение, закончен.

```
if (userName) {
  var outputObj = document.getElementById("output");
  outputObj.firstChild.nodeValue = userName;
}
```

Если в переменной `userName` присутствует имя посетителя, то есть посетитель ввел свое имя в окно ввода, выводим его на экран.

```
<P><A HREF="#" ONCLICK="deleteCookie('username');">Удалить
сохраненное имя</A></P>
```

Для удаления cookie с именем посетителя мы используем "пустую" гиперссылку, к событию `onClick` которой привязан в качестве обработчика вызов функции `deleteCookie`. Этой функции мы передаем в виде строки имя значения `username`, которое гарантированно присутствует в удаляемом cookie.

Cookie можно также использовать для хранения каких-то настроек. Так, можно предусмотреть возможность изменять оформление страницы (цвет текста и фона, шрифт и пр.) и хранить параметры оформления в cookie. Но этим редко пользуются.

Передача данных между страницами

Для передачи данных между страницами мы можем использовать два способа. Давайте их рассмотрим.

Первый способ — применение для передачи данных cookie. Все необходимые инструменты у нас уже есть. Мы сохраняем данные, подлежащие передаче, в cookie на одной странице, на другой странице читаем и, возможно, удаляем хранящий их cookie, если он нам больше не нужен.

Достоинство этого способа — возможность передачи данных более чем одной странице. В самом деле, если мы сохранили данные в cookie, они автоматически становятся доступными для всех страниц, хранящихся в той же папке на Web-сервере и во всех вложенных в нее папках. Недостаток — сохраненные в cookie данные остаются на клиентском компьютере (если мы специально не удалим хранящий их cookie) и теоретически могут быть использованы злоумышленниками.

Второй способ — передача данных в составе интернет-адреса в строке параметров (о строке параметров говорилось в *главе 7*). Строка параметров содержит пары вида `<имя>=<значение>`, разделенные символом апострофа `&`, помещается в самом конце интернет-адреса и отделяется от него символом вопросительного знака (`?`). Например (строка параметров подчеркнута):

http://www.somesite.ru/pages/page2.html?name1=Vasya&name2=Pupkin

Здесь мы передаем странице page2.html два значения с именами `name1` и `name2` — "Vasya" и "Pupkin" соответственно.

Данные, передаваемые в строке параметров, должны удовлетворять двум условиям. Во-первых, они должны быть закодированы с использованием escape-кодирования (как мы помним, за это "отвечает" встроенная функция JavaScript `escape`). Во-вторых, совокупная длина интернет-адреса (вместе со строкой параметров) не должна превышать 256 символов — это требование протокола TCP/IP (о протоколах Интернета см. *главу 1*).

Общий сценарий передачи данных в строке параметров следующий:

1. Из передаваемых данных формируем строку параметров, для чего кодируем их, объединяем в пары `<имя>=<значение>` и добавляем в конец интернет-адреса.
2. Выполняем переход по сформированному таким образом интернет-адресу. Для этого достаточно присвоить готовый интернет-адрес в строковом виде свойству `href` объекта `Location` (подробнее см. *главу 7*). Также можно использовать другие средства, описанные в той же главе, например, метод `open` объекта `Window`; в любом случае, это будет зависеть от конкретной задачи.
3. На новой странице мы получаем строку параметров (ее вернет свойство `search` объекта `Location`), разделяем ее на пары, раскодируем данные (встроенная функция JavaScript `unescape`) и используем в сценариях.

Достоинство этого способа в том, что данные передаются непосредственно нужной странице, не оставляя "следов" на клиентском компьютере. Недостаток — для передачи данных нескольким страницам нам придется повторять все перечисленные выше шаги снова и снова.

Так, сформировать строку параметров мы можем без особого труда. А вот найти в принятой строке параметров нужное значение не так просто. Поэтому давайте напишем функцию `getData`, которая нам очень в этом поможет.

`getData(<имя значения>)`

Эта функция будет принимать единственный параметр — `имя искомого значения` в строковом виде. Если значение с заданным `именем` присутствует в принятой

строке параметров, оно будет возвращено также в строковом виде; в противном случае функция вернет `null`.

```
function getData(pName) {
  var sSearch = location.search;
  var sName = pName + "=";
  var i = sSearch.indexOf(sName);
  if (i > -1) {
    var j = sSearch.indexOf("&", i + sName.length);
    if (j == -1) j = sSearch.length;
    return unescape(sSearch.substring(i + sName.length, j));
  } else
    return null;
}
```

Эта функция полностью аналогична рассмотренной ранее функции `getCookie`, так что вы сами сможете разобраться, как она работает. Единственное исключение обусловлено самой "природой" получаемого значения; строка, содержащая данные, извлекается из свойства `search` объекта `Location`, а отдельные пары *<имя>=<значение>* разделяются не точкой с запятой, а апострофом.

В качестве примера напишем две страницы; первая страница будет принимать данные от посетителя и передавать второй странице в строке параметров. Вторая страница для извлечения значений из строки параметров будет использовать функцию `getData`, что мы только что написали.

Начнем с первой страницы. Ее HTML-код приведен далее.

```
<HTML>
  <HEAD>
    <TITLE>Передача данных</TITLE>
    <SCRIPT>
      function hrefClick() {
        if ((name1) && (name2)) {
          var s = "11.4.htm?name1=" + escape(name1) + "&name2=" +
          escape(name2);
          location.href = s;
        }
      }
    </SCRIPT>
  </HEAD>
  <BODY>
    <H1 ID="output"> </H1>
```

```
    <P><A HREF="#" ONCLICK="hrefClick();">Передать данные</A></P>
    <SCRIPT>
      var name1 = window.prompt("Введите ваше имя", "");
      if (name1) {
        var name2 = window.prompt("Введите вашу фамилию", "");
        if (name1) {
          var outputObj = document.getElementById("output");
          outputObj.firstChild.nodeValue = "Привет, " + name1 + " " +
          name2 + "!";
        }
      }
    </SCRIPT>
  </BODY>
</HTML>
```

Мы видим загрузочный сценарий, который запрашивает у посетителя его имя и фамилию и, если они были введены, выводит их в заголовке `output`. Здесь нам все знакомо.

Для передачи данных другой странице мы используем "пустую" гиперссылку, к событию `onClick` которой привязана функция-обработчик `hrefClick`. Эта функция проверяет, были ли введены имя и фамилия посетителя, и, если были введены, формирует из них строку параметров, добавляет их к интернет-адресу второй страницы (11.4.htm) и открывает ее в том же окне Web-обозревателя.

```
function hrefClick() {
  if ((name1) && (name2)) {
    var s = "11.4.htm?name1=" + escape(name1) + "&name2=" +
    escape(name2);
    location.href = s;
  }
}
```

Обратим внимание на выражение, которое формирует интернет-адрес второй страницы и добавляет к нему строку параметров. Также отметим, что готовый интернет-адрес присваивается свойству `href` объекта `Location` — это самый простой способ загрузить вторую страницу.

Сохраним первую страницу в файле 11.3.htm и рассмотрим HTML-код второй страницы.

```
<HTML>
  <HEAD>
    <TITLE>Прием данных</TITLE>
```

```
    <SCRIPT>
      //Здесь должен присутствовать код, объявляющий функцию getData
    </SCRIPT>
  </HEAD>
  <BODY>
    <H1 ID="output"> </H1>
    <SCRIPT>
      var name1 = getData("name1");
      if (name1) {
        name2 = getData("name2");
        if (name2) {
          var outputObj = document.getElementById("output");
          outputObj.firstChild.nodeValue = "Привет, " + name1 + " " +
          name2 + "!";
        }
      }
    </SCRIPT>
  </BODY>
</HTML>
```

Не забываем, что в этой странице должен присутствовать код, объявляющий функцию `getData`. Автор опустил его для экономии места, но вы должны его вставить в то место, где находится предписывающий сделать это комментарий (в секции заголовка страницы).

В остальном, здесь совсем нечего комментировать. Мы получаем значения имени и фамилии из строки параметров, используя функцию `getData`, и, если они там присутствуют, выводим в заголовок `output`.

Сохраним вторую страницу в файле 11.4.htm. После этого загрузим в Web-обозреватель первую страницу 11.3.htm, введем имя и фамилию и щелкнем гиперссылку **Передать данные**. Если мы все сделали правильно, Web-обозреватель откроет вторую страницу 11.4.htm, в которой появятся введенные нами ранее имя и фамилия.

Обработка данных с использованием регулярных выражений

Последнее, что мы рассмотрим в этой главе, — использование регулярных выражений для обработки данных. Автор решил вынести этот материал в главу, посвященную работе с данными, хотя его, вероятно, следовало бы рассмотреть еще в *главе 4*, где описывается язык JavaScript, но она и так получилась слишком большой. Так что рассмотрим регулярные выражения здесь.

Введение в регулярные выражения

Регулярные выражения — это особые шаблоны для поиска последовательности символов в строке. Начать рассказ о них лучше всего с примера.

Предположим, что мы реализовали на нашей странице ввод посетителем какого-либо интернет-адреса и теперь хотим проверить, правильный ли интернет-адрес он ввел. Что мы будем делать? Можно, конечно, проверить, присутствуют ли во введенной строке символы "www", но ведь далеко не каждый интернет-адрес их содержит. Проверить, содержит ли введенная строка точки? Но точки могут присутствовать и в обычном тексте, не являющемся интернет-адресом. Символы "ru"? А если посетитель введет интернет-адрес из домена com? Проблема…

В этом случае нам на помощь придет регулярное выражение. Оно будет иметь следующий вид:

```
http://.+\.[a-z]{2,3}
```

Какой-то непонятный набор символов, в котором с трудом угадывается что-то знакомое… Признаться, автор и сам сначала с трудом в них разбирался. Ему помогали таблички, подобные табл. 11.1.

Таблица 11.1. *Регулярное выражение поиска интернет-адресов*

Символы	Описание
`http://`	Начало интернет-адреса — обозначение протокола HTTP
`.`	Точка обозначает любой символ
`+`	Плюс обозначает, что предыдущий символ должен повториться минимум один раз
`\.`	Обычная точка. Ее предваряет символ обратного слеша, так как точка в регулярных выражениях имеет особое значение — обозначает любой символ
`[a-z]`	Любой символ от "a" до "z", то есть латинская буква
`{2,3}`	Предыдущий символ должен повторяться от двух до трех раз

Итак, что же обозначает это регулярное выражение? Любую строку, удовлетворяющую определенным условиям. Перечислим их.

- Начальными символами этой строки должны быть "http://" (обозначение протокола HTTP).
- За ними должно следовать любое количество любых символов, но не менее одного.

- В конце этой строки должна присутствовать точка, за которой будут следовать две или три латинские буквы (обозначение домена верхнего уровня).

Таким образом, написанное нами регулярное выражение обозначает любой интернет-адрес, в начале которого присутствует обозначение протокола HTTP.

В регулярных выражениях используются специальные символы — *литералы*. Каждый такой литерал задает поиск того или иного символа или целых последовательностей символов. Так, символ точки `.` обозначает любой символ, а символ `+`, следующий за каким-либо символом, указывает, что предыдущий символ должен встретиться в тексте как минимум однажды.

Литералы могут содержать более чем один символ. Так, литерал `[a-z]` задает поиск любого символа от "a" до "z", то есть маленькой латинской буквы. А литерал `{2,3}` обозначает, что предыдущий символ должен встретиться в строке два или три раза.

Если же мы хотим найти какой-либо символ, совпадающий с литералом, мы должны будем предварить его обратным слешем. Например, чтобы найти в тексте точку, мы используем последовательность символов `\.`.

Рассмотрим еще одно регулярное выражение.

```
\w.+@.+\.com
```

Оно соответствует любому адресу электронной почты, находящемуся в домене com. Чтобы разобраться в нем, напишем таблицу, аналогичную табл. 11.1. Это будет табл. 11.2.

Таблица 11.2. *Регулярное выражение поиска адресов электронной почты*

Символы	Описание
`\w`	Любой алфавитно-цифровой символ или символ подчеркивания
`.`	Любой символ
`+`	Предыдущий символ должен повториться минимум один раз
`@`	Символ "@"
`.`	Любой символ
`+`	Предыдущий символ должен повториться минимум один раз
`\.`	Точка
`com`	Символы "com"

Это регулярное выражение более "строгое", чем написанное нами ранее. Так, оно предполагает, что адрес электронной почты должен начинаться с алфавитно-цифрового символа или подчеркивания (но, поскольку подчеркивание в начале интернет-адресов практически не используется, мы можем не принимать это во внимание). В почтовом адресе также должен присутствовать символ "@" — собственно, главный признак почтового адреса.

А вот еще одно регулярное выражение:

```
[\.!\?]$
```

Оно обозначает точку, восклицательный или вопросительный знак, присутствующий в конце абзаца. Снова напишем табличку, чтобы в нем разобраться (табл. 11.3).

Таблица 11.3. *Регулярное выражение поиска последнего знака препинания абзаца*

Символы	Описание
`[\.!\?]`	Один из возможных символов: точка, восклицательный и вопросительный знаки. Заметим, что вопросительный знак предварен обратной косой чертой, так как иначе JavaScript воспринял бы его как литерал
`$`	Конец строки (символ возврата каретки)

В регулярных выражениях мы можем использовать скобки для обособления каких-то их фрагментов. Рассмотрим, например, такое регулярное выражение:

```
(multi|hyper)media
```

Оно ищет в строке слова "multimedia" и "hypermedia", но не слово "media". Литерал `|` задает поиск либо первой, либо второй подстроки (в нашем случае либо "multi", либо "hyper"), а скобки здесь использованы для того, чтобы обособить фрагмент выражения, содержащий этот литерал. Если бы мы их не поставили, получилось бы выражение `multi|hypermedia`, совпадающее со словами "multi" и "hypermedia".

Скобки в регулярных выражениях также используются для создания так называемых *подвыражений*. Это отдельные фрагменты регулярного выражения, имеющие определенную степень "независимости". Фрагменты строки, которые соответствуют этим подвыражениям, помещаются JavaScript в особые ячейки памяти, из которых мы сможем их извлечь и использовать. Эти ячейки обозначаются номерами, совпадающими с порядковыми номерами присутствующих в регулярном выражении подвыражений; так, фрагмент строки, соответствующий первому по счету подвыражению, будет помещен в ячейку 1, второй — в ячейку 2 и т. д. Таких ячеек может быть 9.

Для примера давайте немного изменим написанное ранее регулярное выражение, предназначенное для поиска адреса электронной почты:

```
(\w.+)@.+\.com
```

Здесь мы превратили часть выражения, обозначающую имя пользователя (почтового ящика), в подвыражение. Поскольку это первое по счету подвыражение в регулярном выражении, соответствующий ему фрагмент строки будет помещен в ячейку 1. Для доступа к ней мы можем написать вот такой литерал:

```
\1
```

Мы можем использовать содержимое этой ячейки в том же самом регулярном выражении. Например, в таком:

```
(\w.+)@\1\.com
```

Это регулярное выражение будет соответствовать строке, удовлетворяющей перечисленным далее условиям.

- Она начинается с любого алфавитно-цифрового символа или символа подчеркивания.
- Затем должно следовать любое количество любых символов, но не менее одного.
- Все эти символы попадают в подвыражение, первое в данном регулярном выражении, и они будут помещены в ячейку 1.
- Далее должен идти символ `@`.
- После него должны следовать в точности те же символы, что находились до этого знака (тот же фрагмент строки, что соответствует первому подвыражению и находится теперь в ячейке 1).
- Оканчиваться эта строка должна символами ".com".

То есть это регулярное выражение будет соответствовать почтовым адресам вида **google@google.com**, **microsoft@microsoft.com** и т. п.

А теперь настало время перечислить все литералы регулярных выражений, поддерживаемые JavaScript (табл. 11.4).

Таблица 11.4. *Литералы, используемые в регулярных выражениях*

Литерал	Описание
`.`	Любой символ, за исключением перевода строки
`[abc]`	Любой символ из перечисленных в квадратных скобках. Можно задавать диапазоны символов, например, `[a-d]` заменяет `[abcd]`

Таблица 11.4 *(окончание)*

Литерал	Описание
`[^abc]`	Любой символ, кроме перечисленных в квадратных скобках. Можно задавать диапазоны символов, например, вместо `[^abcd]` задать `[^a-d]`
`\d`	Любая цифра. Эквивалентен `[0-9]`
`\B`	Любой символ, кроме цифры. Эквивалентен `[^0-9]`
`\w`	Любой алфавитно-цифровой символ или подчеркивание. Эквивалентен `[a-zA-Z0-9_]`
`\W`	Любой символ, кроме алфавитно-цифрового и подчеркивания. Эквивалентен `[^a-zA-Z0-9_]`
`\s`	Любой пробельный символ (пробел, табуляция, прогон страницы или перевод строки)
`\S`	Любой символ, кроме пробельного
`\b`	Граница слова (пробел или возврат каретки)
`\B`	Любой символ, кроме границы слова
`^`	Начало строки
`$`	Конец строки
`\f`	Прогон листа
`\n`	Перевод строки
`\r`	Возврат каретки
`\t`	Табуляция
`\<код>`	Символ с заданным восьмеричным `код`ом
`\x<код>`	Символ с заданным шестнадцатеричным `код`ом
`*`	Предыдущий символ должен встретиться один или больше раз или не встретиться вообще
`+`	Предыдущий символ должен встретиться один или больше раз
`?`	Предыдущий символ должен встретиться один раз или не встретиться вообще
`x|y`	Должен встретиться символ `x` или символ `y`
`{n}`	Предыдущий символ должен встретиться `n` раз
`{m,n}`	Предыдущий символ должен встретиться от `n` до `m` раз
`(<литералы>)`	Создает подвыражение, содержащее заданные `литералы`
`\<номер>`	Содержимое ячейки, содержащей соответствующий подвыражению с заданным `номер`ом фрагмент строки

Средства JavaScript для работы с регулярными выражениями

Рассмотрев сами регулярные выражения и используемые в них литералы, давайте поговорим о средствах, предлагаемых JavaScript для работы с ними. Это, прежде всего, два объекта-"тезки" `RegExp` и три метода объекта `String`, которые мы не рассмотрели в *главе 4*.

Первый объект `RegExp` представляет само регулярное выражение. Его экземпляр создается с помощью выражения вида:

```
new RegExp(<регулярное выражение>[, <флаги>])
```

Первым параметром передается само регулярное выражение в виде строки. Оно записывается в уже знакомом нам виде.

Второй — необязательный — параметр задает строку, содержащую флаги, — символы, задающие дополнительные параметры. Таких символов может быть три:

- "g" — поиск всех фрагментов, соответствующих регулярному выражению (если не задан, будет найден только первый фрагмент, после чего поиск остановится);
- "i" — при поиске регистр символов не будет приниматься во внимание (если не задан, регистр символов при поиске принимается во внимание);
- "m" — многострочный поиск, при котором будут учитываться символы возврата каретки и перевода строки. Если этот флаг задан, литерал `^` обозначает как самое начало строки, так и позицию, следующую за символами возврата каретки или перевода строки, а литерал `$` — как самый конец строки, так и позицию, предшествующую символам возврата каретки или перевода строки. Если этот флаг не установлен, литерал `^` обозначает только самое начало строки, а литерал `$` — только самый конец строки, то есть символы возврата каретки и перевода строки не будут приниматься во внимание. Поддерживается только Internet Explorer.

Экземпляр объекта `RegExp`, возвращенный оператором `new`, следует присвоить какой-либо переменной, чтобы его можно было использовать для дальнейшей работы.

```
var re = new RegExp("http://.+\.[a-z]{2,3}", "i");
```

Это выражение создаст экземпляр объекта `RegExp`, хранящий регулярное выражение для поиска интернет-адресов, первое из тех, что мы написали. Этот экземпляр объекта будет присвоен переменной `re`.

JavaScript поддерживает и другой, упрощенный, синтаксис выражения для создания экземпляра объекта `RegExp`. Вот его формат:

/<регулярное выражение>/[*<флаги>*]

Например:

```
var re = /http://.+\.[a-z]{2,3}/i;
```

Полученный экземпляр объекта `RegExp` можно использовать для поиска соответствующих им фрагментов строк. Для этого используются методы объекта `String`, которые мы сейчас рассмотрим.

Метод `match` позволяет выполнить поиск фрагмента строки, соответствующий заданному регулярному выражению.

`match(`*<регулярное выражение в виде экземпляра объекта RegExp>*`)`

Этот метод принимает единственный параметр — экземпляр объекта `RegExp`, хранящий регулярное выражение. Возвращает он массив, содержащий найденные фрагменты текста. Что это за массив и какие элементы он содержит, зависит от Web-обозревателя и от того, был ли при создании регулярного выражения (то есть экземпляра объекта `RegExp`) задан флаг "g".

Если флаг "g" был задан, возвращенный методом `match` массив будет одинаковым у всех Web-обозревателей. Его элементы будут представлять все найденные фрагменты строки, соответствующие регулярному выражению. А длина этого массива (то есть количество его элементов) укажет количество найденных фрагментов.

Если же флаг "g" не был задан, массив будет разным у разных Web-обозревателей. В случае Opera и Firefox он будет содержать только один элемент — первый из найденных фрагментов строки.

Internet Explorer, как всегда, идет своим путем. В его случае первый элемент возвращенного методом `match` массива будет содержать найденный фрагмент, а последующие элементы — фрагменты, соответствующие присутствующим в регулярном выражении подвыражениям (если они, конечно, там присутствуют). Кроме того, этот массив будет содержать три дополнительных свойства: свойство `index` вернет номер начального символа найденного фрагмента, свойство `input` — саму исходную строку, а свойство `lastIndex` — номер символа, следующего за найденным фрагментом.

Если ни одного подходящего фрагмента в строке не было найдено, во всех случаях возвращается `null`.

```
var str1 = "http://www.somesite.com";
var str2 = "someuser@somesite.com";
var result1 = str1.match(re);
var result2 = str2.match(re);
```

Этот сценарий выполнит поиск в строках `str1` и `str2` с использованием регулярного выражения `re`. В переменной `result1` окажется массив с единственным элементом, содержащим всю строку `str1` — результат успешного поиска, а в переменной `result2` — `null`, так как строка `str2` не соответствует регулярному выражению `re`.

Метод `replace` возвращает строку, в которой все фрагменты, соответствующие заданному регулярному выражению, заменены другим текстом. Сама текущая строка при этом не изменяется.

```
replace(<регулярное выражение в виде экземпляра объекта RegExp>,
↳<заменяющий текст>)
```

Первым параметром этому методу передается регулярное выражение в виде экземпляра объекта `RegExp`. Вторым параметром передается строка, которой будут заменены все фрагменты, соответствующие заданному регулярному выражению. Метод возвращает измененную строку; текущая строка, как уже говорилось ранее, не изменяется.

```
var str1 = "http://www.somesite.com";
var re = new RegExp("http");
var str2 = str1.replace(re, "ftp");
```

Этот сценарий заменит в строке `str1` идентификатор протокола HTTP на идентификатор протокола FTP и поместит измененную строку в переменную `str2`. Строка `str1` останется неизменной.

Метод `search` возвращает номер первого символа фрагмента строки, соответствующего заданному регулярному выражению. Если подходящий фрагмент не найден, возвращается –1. В этом смысле метод `search` аналогичен методу `indexOf`, рассмотренному в *главе 4*.

```
search(<регулярное выражение в виде экземпляра объекта RegExp>)
```

Единственный параметр этого метода задает регулярное выражение.

```
var str = "http://www.somesite.com";
var re = new RegExp("\.");
var n = str.search(re);
```

Это выражение поместит в переменную `n` число 10 — номер первого символа точки в строке `str` (не забываем, что символы в строке нумеруются, начиная с нуля).

Объект `RegExp` поддерживает также несколько свойств, которые могут быть нам полезны. Все они перечислены в табл. 11.5.

Таблица 11.5. *Свойства объекта* `RegExp`

Свойство	Описание
`global`	Возвращает `true`, если при создании регулярного выражения был задан флаг "g", и `false` в противном случае
`ignoreCase`	Возвращает `true`, если при создании регулярного выражения был задан флаг "i", и `false` в противном случае
`lastIndex`	Задает номер символа строки, с которого начнется поиск. Изначально, до выполнения самого первого поиска, равно 0
`multiline`	Возвращает `true`, если при создании регулярного выражения был задан флаг "m", и `false` в противном случае. Поддерживается только Internet Explorer
`source`	Возвращает строку, содержащую само регулярное выражение

```
var str = "http://www.somesite.com";
var re = new RegExp("\.");
re.lastIndex = 11;
var n = str.search(re);
```

Это выражение поместит в переменную `n` число 19 — номер второго символа точки в строке `str`.

Объект `RegExp` также поддерживает три полезных метода. Поговорим о них.

Метод `compile` может использоваться для изменения регулярного выражения уже после создания представляющего его экземпляра объекта `RegExp`.

`compile(`*<регулярное выражение>*`[, <флаги>])`

Все параметры этого метода нам уже знакомы. Возвращает он экземпляр объекта `RegExp`, представляющий новое регулярное выражение.

> ***НА ЗАМЕТКУ***
>
> В документации по JavaScript написано, что метод `compile` используется для преобразования регулярного выражения в некий внутренний формат, что ускоряет поиск. Но фактически его применяют для изменения регулярного выражения, да и то редко — проще создать новое регулярное выражение.

Метод `exec` позволяет выполнить поиск фрагмента строки, соответствующего регулярному выражению.

`exec(`*<строка, в которой выполняется поиск>*`)`

Единственным параметром этому методу передается строка, в которой выполняется поиск.

Если подходящий фрагмент найден, метод `exec` возвращает массив. Первый элемент этого массива будет содержать найденный фрагмент, а последующие элементы — фрагменты, соответствующие присутствующим в регулярном выражении подвыражениям (если они, конечно, там присутствуют). Кроме того, этот массив будет содержать два дополнительных свойства: свойство `index` вернет номер начального символа найденного фрагмента, а свойство `input` — саму исходную строку. Internet Explorer также поддерживает свойство `lastIndex` этого массива, возвращающее номер символа, следующего за последним найденным фрагментом.

Если подходящего фрагмента найдено не было, возвращается `null`.

Если при создании регулярного выражения был задан флаг "g", при последующих вызовах метода `exec` поиск будет выполняться, начиная с позиции, следующей за последним найденным фрагментом. Если флаг "g" не был задан, при последующих вызовах метода `exec` поиск всегда будет выполняться с начала строки.

```
var str = "http://www.somesite.com";
var re = new RegExp("\.");
var n1 = re.exec(str);
var n2 = re.exec(str);
```

Этот сценарий поместит в переменную `n1` число 10 — номер первого символа точки в строке `str`. А в переменной n2 окажется число 19 — номер второго символа точки в строке `str`.

Метод `test` позволяет проверить, присутствует ли в заданной строке хоть один фрагмент, соответствующий текущему регулярному выражению.

```
test(<строка, в которой выполняется поиск>)
```

Единственным параметром этому методу передается строка, которую нужно проверить на наличие подходящих фрагментов. Метод возвращает `true`, если хоть один фрагмент был найден, и `false` в противном случае.

```
var str = "http://www.somesite.com";
var re1 = new RegExp("http");
var re2 = new RegExp("ftp");
var f1 = re1.exec(str);
var f2 = re2.exec(str);
```

Этот сценарий поместит в переменную `f1` значение `true`, так как в строке `str` присутствуют символы "http", а в переменную `f2` — значение `false`, поскольку символов "ftp" в строке `str` нет.

Закончив с первым объектом `RegExp`, перейдем к его "тезке". Второй объект `RegExp` содержит только статичные свойства, значения которых содержат

некоторые сведения о найденных фрагментах строк и обновляются при каждом выполнении метода `exec` первого объекта `RegExp` и методов `match`, `replace` и `search` объекта `String`. (О статичных свойствах см. *главу 4*.)

> ***НА ЗАМЕТКУ***
>
> Документация по JavaScript утверждает, что это действительно разные объекты с одним именем `RegExp`.

Все статичные свойства второго объекта `RegExp` перечислены в табл. 11.6.

Таблица 11.6. *Свойства второго объекта* `RegExt`

Свойство	Описание
`$<`*номер*`>`	Возвращает фрагмент текста, соответствующий подвыражению с заданными номером
`index`	Возвращает номер первого символа самого последнего из найденных фрагментов
`input` или `$_`	Задает или возвращает строку, в которой выполняется поиск. Если поиск выполняется вызовом метода `exec` первого объекта `RegExp`, возвращается `null`
`lastIndex`	Задает или возвращает позицию, с которой начнется следующий поиск. Если поиск выполняется впервые, возвращает 0
`lastMatch` или `$@`	Возвращает фрагмент текста, найденный в результате самой последней операции поиска
`lastParen` или `$+`	Возвращает фрагмент текста, соответствующий самому последнему по счету подвыражению в регулярном выражении и найденный в результате самой последней операции поиска
`leftContext` или `$``	Возвращает фрагмент строки от ее начала до последнего найденного фрагмента, но не включая его
`rightContext` или `$'`	Возвращает фрагмент строки от последнего найденного фрагмента, но не включая его, до ее конца

```
var str = "http://www.somesite.com";
var re = new RegExp("(\w+).+");
re.exec(str);
var fostr = RegExp.$1;
var n = RegExp.index;
```

Этот сценарий поместит в переменную `fostr` строку "www" — фрагмент строки `str`, соответствующий первому (и единственному) подвыражению

в регулярном выражении `re`. В переменной `n` окажется число 7 — номер первого символа найденного фрагмента строки `str`, соответствующего регулярному выражению `re` (это фрагмент "www.somesite.com").

Регулярные выражения — очень мощное средство обработки данных, настолько мощное, что Web-программисты применяют его весьма нечасто. Обычно они довольствуются более простыми средствами — методами объекта `String`, возможностей которых вполне хватает в большинстве случаев. Но иногда без регулярных выражений не обойтись, например, если нам нужно проверить введенные посетителем данные на корректность, а эта данные имеют сложный формат (те же адреса электронной почты или телефонные номера).

Так или иначе, о работе с данными все.

Что дальше?

В этой главе мы занимались работой с данными. Мы научились выводить их посетителю, запрашивать их у посетителя, сохранять на клиентском компьютере и передавать другим страницам. Также мы рассмотрели регулярные выражения, которые помогут нам в сложных случаях обработки данных.

В следующей главе, которая будет последней в *части II*, мы продолжим рассмотрение способов ввода данных. Мы познакомимся с Web-формами, которые применяются почти повсеместно для запроса у посетителя целых наборов данных и в этом смысле не оставляют шансов рассмотренным нами в этой главе окнам ввода.

Глава 12

Работа с Web-формами

В предыдущей главе мы, в числе прочего, говорили и о реализации ввода данных. И даже немного попрактиковались в этом. Давайте вспомним все средства Web-обозревателей, что мы использовали, чтобы принять данные от посетителя.

Во-первых, это окна-сообщения, выводимые методом `confirm` объекта `Window`. Такие окна содержат заданный текст, иконку в виде вопросительного знака и кнопки **OK** и **Отмена** (Cancel). Посетитель может нажать любую из этих кнопок, а наш сценарий, в зависимости от этого, может выполнить тот или иной код.

Во-вторых, это окна ввода, выводимые методом `prompt` объекта `Window`. Они содержат поле ввода, заданный текст и кнопки **OK** и **Отмена** (Cancel). Посетитель может ввести в поле ввода любые данные и нажать кнопку **OK**, а может отказаться от ввода, нажав кнопку **Отмена** (Cancel). И, как в предыдущем случае, наш сценарий может, в зависимости от этого, выполнить тот или иной код.

Окна-сообщения используются, если нужно предоставить посетителю выбор из двух альтернатив, обычно — хочет ли он что-то сделать или не хочет. Окна ввода применяются, если нужно запросить у посетителя одно строковое или числовое значение. На большее их возможностей не хватит.

А теперь давайте вспомним наши путешествия по Интернету. Часто нам приходится вводить прямо на Web-странице какие-то данные: имя и пароль пользователя, почтовый адрес, анкетные данные и пр. Мы вводили их прямо на странице, в знакомые нам по Windows-приложениям элементы управления: поля ввода, списки, флажки, переключатели и др. После ввода данных мы нажимали кнопку **Отправить** или **OK** и попадали на страницу, где представлены какие-то данные, принадлежащие нам: письма электронной почты, списки заказов в интернет-магазине или что-то еще.

Ни окна-сообщения, ни окна ввода в этом случае не использовались. А использовались особые элементы страницы, называемые Web-формами и элементами управления. Именно с их помощью реализовывался ввод целого набора разнородных данных.

И именно им будет посвящена вся эта глава.

Создание Web-форм и элементов управления

Web-форма, или просто *форма*, — это особый элемент страницы, реализующий ввод, подготовку и отправку данных другой странице или специальной программе, работающей совместно с Web-сервером и обрабатывающей введенные посетителями данные (*серверная программа*). Web-форма содержит набор элементов управления — полей ввода, списков, флажков, переключателей и пр., — в которые и вводятся данные.

Начнем с рассмотрения работы Web-форм. Иначе нам будет трудно понять назначение некоторых атрибутов создающего их тега.

Как работают Web-формы

Давайте перечислим все задачи, которые должна выполнить Web-форма.

1. Принять данные от посетителя.
2. Подготовить их к отправке.
3. Отправить данные серверной программе или другой Web-странице.

Принять данные от посетителя — самая простая задача. Посетитель вводит их в элементы управления, присутствующие в форме. После ввода данных он нажимает особую кнопку, называемую *кнопкой отправки данных*; обычно эта кнопка имеет надпись **Отправить** (Submit). После ее нажатия форма запускает процесс подготовки и отправки данных.

> ***НА ЗАМЕТКУ***
>
> Также в форме может присутствовать *кнопка сброса формы*, имеющая надпись **Сброс** (Reset) и обнуляющая введенные пользователем данные. Надо сказать, пользы от нее немного, поэтому в последнее время кнопки сброса формы практически нигде не применяются.

Каждый элемент управления, присутствующий в форме, должен иметь уникальное имя. Единственное исключение из этого правила — переключатели, о них мы обязательно поговорим.

Итак, процесс отправки данных запущен. Первое, что делает форма, — это представляет введенные данные в виде набора пар *<имя элемента управления>=<введенное в него значение>*. Например, если в форме присутствуют поля ввода `name1`, `surname`, `name2` и `age`, в которые введены строки "Ivan", "Ivanovich", "Ivanov" и "30" соответственно, форма создаст такой набор:

```
name1=Ivan
surname=Ivanovich
name2=Ivanov
age=30
```

Первый шаг к отправке данных сделан.

Следующий шаг — это кодирование данных. Как они будут кодироваться, указывает *метод кодирования данных*. Таких методов формы поддерживают несколько, но какой из них будет реально применен, зависит от заданного метода передачи данных. Поэтому нам придется сначала рассмотреть их.

Метод передачи данных определяет способ, которым данные будут переданы по назначению. Стандарты HTML предусматривают всего два метода передачи данных: GET и POST.

Метод передачи данных *GET* нам уже знаком по *главе 11*. При его использовании данные передаются в строке параметров как часть интернет-адреса. Так, рассмотренный нами выше набор данных будет передан в составе вот такого интернет-адреса (сама строка параметров подчеркнута):

http://www.somesite.ru/bin/program.exe?name1=Ivan&surname2=Ivanovich&
↳name2=Ivanov&age=30

Здесь program.exe — имя файла серверной программы, которая обработает отправленные данные.

Метод кодирования данных здесь применяется всего один — escape-кодирование, которое нам также знакомо по *главе 11*.

> ***НА ЗАМЕТКУ***
>
> Ранее мы рассмотрели идеальный пример — посетитель ввел данные, содержащие только латинские буквы. В этом случае закодированная строка выглядит в точности так же, как незакодированная. Если бы посетитель ввел данные, содержащие символы, недопустимые в интернет-адресах (буквы кириллицы, пробелы, символы двоеточия, слеша, кавычек и др.), незакодированная и закодированная строки выглядели бы по-разному.

У метода GET есть два достоинства. Во-первых, передаваемые данные, что называется, на виду, и Web-программист всегда сможет проверить их правильность. Во-вторых, с его помощью можно передавать данные не только

серверной программе, но и другой Web-странице, что мы и рассмотрели в *главе 11*.

Недостатка у этого метода также два. Первый: с его помощью нельзя передавать большие объемы данных. Это происходит из-за ограничения, накладываемого протоколом TCP/IP на длину интернет-адреса — не более 255 символов. Вычтем отсюда длину собственно интернет-адреса серверной программы — и получим максимально допустимый размер наших данных. Второй недостаток метода GET — обратная сторона его достоинства. Данные, пересылаемые им, открыты для всеобщего обозрения и могут быть легко прочитаны в поле ввода интернет-адреса Web-обозревателя.

Метод GET стоит использовать, если пересылаемые серверной программе данные заведомо невелики и не являются секретными. В частности, он используется для пересылки ключевых слов поисковым машинам, служебных данных и т. п. Также это единственный метод, позволяющий передавать данные другим страницам. В остальных случаях лучше использовать второй метод передачи, называемый POST.

Метод *POST* передает данные серверной программе в виде так называемых *дополнительных данных* клиентского запроса. Поскольку размер дополнительных данных не ограничен (по крайней мере, он может быть очень велик), мы передадим все, что угодно, в каких угодно количествах. Таким способом Web-серверу можно передать даже файл. К сожалению, другим страницам данные этим способом передать нельзя.

Метод передачи данных POST поддерживает три метода кодирования данных:

- `application/x-www-form-urlencoded` — поддерживает кодирование строковых данных и используется в большинстве случаев;
- `multipart/form-data` — поддерживает также кодирование двоичных данных и используется, если нужно отправить файл;
- `text/plain` — представляет данные в виде текста и используется, если нужно передать данные в сообщении электронной почты.

> ***НА ЗАМЕТКУ***
>
> Метод кодирования данных фактически указывает тип MIME передаваемых данных. О типах MIME было рассказано в *главе 9*.

Достоинства метода POST: отсутствие ограничения на объем передаваемых данных и их "невидимость". Недостатки: сложность расшифровки данных и трудность отладки. Методом POST передаются, например, анкетные данные, адреса покупателей в электронных магазинах, литературные произведения на сайты **http://www.stihi.ru** и **http://www.proza.ru** и т. п. В общем, то, что имеет большие объемы.

НА ЗАМЕТКУ

Ходят слухи, что комитет W3C намерен со временем заставить Web-дизайнеров и Web-программистов вообще отказаться от метода GET и использовать только метод POST. Пока что метод GET просто объявлен не рекомендованным (deprecated) для использования во вновь создаваемых сайтах.

Создание Web-форм

Так, с Web-формами мы в основном познакомились. Настала пора выяснить, как же их создавать.

Для создания форм используется особый парный тег <FORM>.

```
<FORM ACTION="<интернет-адрес серверной программы или Web-страницы>"
[METHOD="get|post"] [ENCTYPE="<метод кодирования данных>"]
[TARGET="<цель>"] [AUTOCOMPLETE="on|off"]>
  . . . Теги, создающие элементы управления
</FORM>
```

Теги, создающие элементы управления, что входят в эту форму, помещаются внутри тега <FORM>. Мы рассмотрим эти теги потом.

Обязательный атрибут ACTION тега <FORM> задает интернет-адрес серверной программы или страницы, которой будут переданы данные. Мы также можем указать в качестве значения этого атрибута "#", тогда данные не будут никуда отправлены; это может пригодиться, например, если введенные в форму данные должны обрабатываться на этой же странице с помощью сценариев.

Необязательный атрибут METHOD задает метод передачи данных и может принимать значения "get" и "post". Если этот атрибут не указан, используется метод GET.

Атрибут ENCTYPE, задающий метод кодирования данных, имеет силу только при выборе метода передачи данных POST. Если этот атрибут не указан, используется метод кодирования данных application/x-www-form-urlencoded.

Ранее было сказано, что введенные в форму данные могут быть отправлены по электронной почте. Для этого нужно сделать следующее:

1. В атрибуте ACTION указать нужный адрес электронной почты в формате, описанном в *главе 2* ("mailto:<адрес>").
2. В атрибуте METHOD указать метод передачи данных POST (значение "post").
3. В атрибуте ENCTYPE указать метод кодирования данных text/plain (значение "text/plain").

Тогда при щелчке на кнопке отправки данных будет загружена программа почтового клиента, установленная в системе как клиент почты по умолчанию, сформировано новое сообщение, в строку адреса получателя подставлен заданный нами адрес, а в тело сообщения — текст, содержащий введенные данные в виде набора пар *<имя элемента управления>=<введенное в него значение>*.

Но мы отвлеклись. Необязательный атрибут TARGET давно нам знаком — он задает цель для загрузки страницы, которой были отправлены данные или которая была сформирована серверной программой на основе введенных данных. Значения этого атрибута были описаны в *главе 2*.

Необязательный атрибут AUTOCOMPLETE поддерживается только Internet Explorer. Он задает, будет ли задействована в данной форме функция автозаполнения. Значение "on" включает эту функцию, а значение "off" — отключает. Если этот атрибут не указан, функция автозаполнения не задействуется.

Кроме того, тег <FORM> поддерживает атрибуты ID, NAME, CLASS, STYLE и TITLE. Первые два атрибута, в частности, задают имя формы; если мы хотим получить к ней доступ из сценариев, нам следует их задать.

> ***Внимание!***
>
> Для задания имени формы необходимо использовать оба атрибута — и ID, и NAME. Иначе возможны проблемы с некоторыми Web-обозревателями.

Рассмотрим пример формы.

```
<FORM ACTION="http://www.somesite.ru/bin/program.exe" METHOD="post"
ENCTYPE="application/x-www-form-urlencoded" ID="frm" NAME="frm">
  . . .
</FORM>
```

Эта форма имеет имя frm; отметим, что мы задали его и в атрибуте ID, и в атрибуте NAME, чтобы избежать проблем с некоторыми Web-обозревателями. Форма отправит введенные данные серверной программе program.exe, хранящейся в папке bin корневой папки Web-сервера **http://www.somesite.ru**, с использованием метода кодирования данных application/x-www-form-urlencoded и метода передачи данных POST.

Создание элементов управления

Так, с формами мы разобрались. Пора начать разговор об элементах управления. Ведь форма без элементов управления — ничто.

Элементы управления создаются с помощью разных тегов HTML. Это тег <INPUT>, применяемый для создания большинства элементов управления, теги <TEXTAREA>, <SELECT> и <OPTION>. Разумеется, мы все их рассмотрим.

Все эти теги поддерживает уже знакомые нам атрибуты ID, NAME, CLASS, STYLE и TITLE. Первые два атрибута, в частности, задают уникальное имя элемента управления, которое обязательно нужно задать, иначе форма не сможет правильно сформировать набор данных для отправки серверной программе или другой странице. Об этом уже говорилось в самом начале данной главы, но, как говорится, повторенье — мать ученья.

> ***Внимание!***
>
> Для задания имени элемента управления следует использовать оба атрибута — и ID, и NAME. Иначе возможны проблемы с некоторыми Web-обозревателями.

А теперь рассмотрим все элементы управления, предлагаемые языком HTML, и теги, которые их создают.

Поле ввода

Поле ввода — наиболее часто используемый в Web-формах элемент управления. Он создается с помощью одинарного тега <INPUT>.

```
<INPUT [TYPE="text"] [VALUE="<изначальное значение>"] [SIZE="<размер>"]
[MAXLENGTH="<максимальное количество символов>"]
[TABINDEX="<номер в порядке обхода>"] [ACCESSKEY="<быстрая клавиша>"]
[DISABLED] [READONLY] [AUTOCOMPLETE="on|off"]>
```

Атрибут TYPE задает тип элемента управления. Значение "text" указывает Web-обозревателю создать именно поле ввода. Поле ввода также создается, если атрибут TYPE не указан (он является необязательным).

Необязательный атрибут VALUE задает значение, которое должно присутствовать в поле ввода изначально. Если этот атрибут не указан, поле ввода не будет содержать ничего.

Необязательный атрибут SIZE задает длину поля ввода в символах. Если он не указан, поле ввода будет иметь длину в 20 символов.

Необязательный атрибут MAXLENGTH задает максимальный размер строки, которую можно ввести в это поле ввода, в символах. Если этот атрибут не указан, в поле ввода можно будет ввести строку неограниченного размера.

Необязательный атрибут TABINDEX задает номер в порядке обхода. Что это такое? Давайте выясним.

Пользователи Windows знают, что "путешествовать" по элементам управления можно с помощью клавиатуры. Если активизировать какой-либо элемент управления в окне и нажать клавишу <Tab>, то активным станет следующий элемент управления. А если нажать комбинацию клавиш <Shift>+<Tab>, то активным станет предыдущий элемент управления.

Какой элемент управления станет активным при очередном нажатии клавиши <Tab> (или <Shift>+<Tab>), задает так называемый *порядок обхода*, в котором все элементы управления представлены номерами. Так, если в данный момент активен элемент управления, номер которого в порядке обхода 3, то при нажатии клавиши <Tab> будет активизирован элемент управления с номером в порядке обхода 4, а при нажатии комбинации клавиш <Shift>+<Tab> — с номером 2.

Тот же принцип действует и в Web-формах. Атрибут `TABINDEX` как раз и задает для элементов управления номер в порядке обхода.

Если этот атрибут не указан, Web-обозреватель сам назначит номер в порядке обхода для данного элемента управления — на единицу больший, чем номер предыдущего элемента управления. Если для всех элементов управления в форме номера в порядке обхода не указаны, порядок обхода будет совпадать с порядком, в котором элементы управления присутствуют в HTML-коде страницы.

В Windows-приложениях мы также можем активизировать некоторые элементы управления нажатием особой клавиши, удерживая при этом нажатой клавишу <Alt> (*быстрая*, или *горячая,* клавиша). То же самое доступно и в Web-формах. Быструю клавишу задает необязательный атрибут `ACCESSKEY`; в качестве его значения указывается наименование нужной клавиши (например, "a", "u", "7" и т. п.).

Атрибут без значения `DISABLED` позволяет сделать поле ввода недоступным; оно будет отображаться серым цветом, и посетитель не сможет даже его активизировать. Если этот атрибут присутствует в теге, поле ввода недоступно, если отсутствует — доступно.

Другой атрибут без значения — `READONLY` — позволяет сделать поле ввода доступным только для чтения; при этом посетитель все-таки сможет активизировать это поле, выделить содержащийся в нем текст и скопировать его в буфер обмена. Если этот атрибут присутствует, поле ввода будет доступно только для чтения, если отсутствует — доступно и для чтения, и для ввода.

Последний атрибут — `AUTOCOMPLETE` — уже нам знаком по тегу `<FORM>`. Он поддерживается только Internet Explorer.

При отправке данных для каждого поля ввода будет сформирована пара вида:

<имя поля ввода>=<значение, введенное в поле ввода>

Напоследок рассмотрим пример формы, имеющей два поля ввода.

```
<FORM ACTION="http://www.somesite.ru/bin/program.exe">
  Имя <INPUT TYPE="text" ID="n" NAME="n" SIZE="40" MAXLENGTH="40"
```

```
  TABINDEX="0" ACCESSKEY="a"><BR>
  Должность <INPUT TYPE="text" ID="e" NAME="e" SIZE="100"
  TABINDEX="1" ACCESSKEY="e" DISABLED><BR>
  . . .
</FORM>
```

Здесь создаются два поля ввода:

- `n` — поле ввода имени, его длина 40 символов, может принимать строки до 40 символов в длину, номер в порядке обхода 0, быстрая клавиша <A> (то есть комбинация клавиш <Alt>+<A>);
- `e` — поле ввода должности, его длина 100 символов, может принимать строки неограниченной длины, номер в порядке обхода 1, быстрая клавиша <E> (то есть комбинация клавиш <Alt>+<E>), недоступно.

Область редактирования

Область редактирования, в отличие от большинства других элементов управления, поддерживаемых HTML, создается парным тегом `<TEXTAREA>`.

```
<TEXTAREA [ROWS="<высота>"] [COLS="<ширина>"] [WRAP="off|soft|hard"]
[TABINDEX="<номер в порядке обхода>"] [ACCESSKEY="<быстрая клавиша>"]
[DISABLED] [READONLY]>
  <изначальное значение>
</TEXTAREA>
```

Первоначальное значение, которое должно присутствовать в области редактирования, помещается внутрь тега `<TEXTAREA>`. Это должен быть текст без всяких HTML-тегов.

Необязательный атрибут `ROWS` задает высоту области редактирования в строках. Если он не указан, Web-обозреватель задаст для нее произвольную высоту.

Необязательный атрибут `COLS` задает ширину области редактирования в символах. Если он не указан, Web-обозреватель задаст для нее произвольную ширину.

> ***ВНИМАНИЕ!***
>
> Значения ширины и высоты области редактирования, устанавливаемые при отсутствии в HTML-коде атрибутов `ROWS` и `COLS`, зависят от Web-обозревателя и операционной системы. Поэтому рекомендуется задавать их явно, с помощью упомянутых ранее атрибутов.

Необязательный атрибут `WRAP` позволяет указать, как область редактирования будет выполнять перевод строк и в каком виде введенное в него значение

будет отправлено серверной программе или другой странице. Этот атрибут может принимать три значения, перечисленные далее.

- "off" — перевод строк не выполняется. В этом случае, если текст не помещается в область редактирования по горизонтали, в ней появляется горизонтальная полоса прокрутки. При этом посетитель может вставить в текст *"жесткий" перевод строки*, нажав клавишу <Enter>.
- "soft" — область редактирования будет выполнять перевод слишком длинных строк, вставляя в текст так называемые *"мягкие" переводы строк*. При отправке значения области редактирования серверной программе эти переводы удаляются. (Собственно, никаких "мягких" переводов в текст вообще не помещается — это просто такой термин.)
- "hard" — перед отправкой значения все "мягкие" переводы строк область редактирования преобразует в "жесткие". При этом в нужных местах текста будут вставлены символы возврата каретки и перевода строки.

Если атрибут `WRAP` не указан, область редактирования будет вести себя так, словно используется значение "soft".

Какое значение атрибута `WRAP` выбрать в том или ином случае? Ответ на этот вопрос зависит от того, что мы хотим иметь на Web-странице и что должна будет обрабатывать серверная программа. Автор составил табл. 12.1, которая может вам помочь хотя бы на первых порах работы с формами.

Таблица 12.1. *Выбор значения атрибута* `WRAP`

Как должны отображаться данные	Как данные должны получаться серверной программой	Значение атрибута `WRAP`
Неизменными	Неизменными	"off"
Измененными (с "мягкими" переносами)	Неизменными	"soft"
Измененными (с "мягкими" переносами)	Измененными (с "жесткими" переносами строк)	"hard"

Остальные атрибуты, поддерживаемые тегом `<TEXTAREA>`, нам уже знакомы.

При отправке данных для каждой области редактирования будет сформирована пара вида:

```
<имя области редактирования>=<значение, введенное в область
↳редактирования>
```

И — пример формы с областью редактирования:

```
<FORM ACTION="http://www.somesite.ru/bin/program.exe">
  . . .
  Почтовый адрес<BR>
  <TEXTAREA ID="address" NAME="address" ROWS="4" COLS="100"
  TABINDEX="5"><BR>
  . . .
</FORM>
```

Это область редактирования для ввода почтового адреса, имеющая имя `address`, размеры 4 строки × 100 символов и номер в порядке обхода 5 (то есть будет одним из последних элементов управления в данной форме). Поскольку атрибут `WRAP` не указан, область редактирования будет выполнять перенос слишком длинных строк, вставляя в них "мягкие" переводы, но при отправке данных они будут удалены (поведение области редактирования по умолчанию).

Поле ввода пароля

Поле ввода пароля ничем не отличается от обычного поля ввода за тем исключением, что вместо вводимых символов в нем отображаются точки или звездочки, в зависимости от версии Windows. Такие поля ввода широко применяются для запроса паролей и других конфиденциальных данных.

Поле ввода пароля 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] [AUTOCOMPLETE="on|off"]>
```

Значение "password" атрибута `TYPE` указывает Web-обозревателю создать поле ввода пароля. Остальные атрибуты нам уже знакомы по обычному полю ввода.

При отправке данных для каждого поля ввода пароля будет сформирована пара вида:

```
<имя поля ввода пароля>=<значение, введенное в поле ввода пароля>
```

Понятно, что `значение` в этом случае не "маскируется" точками или звездочками.

Пример:

```
<FORM ACTION="http://www.somesite.ru/bin/program.exe">
  Имя пользователя <INPUT TYPE="text" ID="name" NAME="name"
```

```
  MAXLENGTH="20"><BR>
  Пароль <INPUT TYPE="password" ID="pass" NAME="pass" MAXLENGTH="20"><BR>
  . . .
</FORM>
```

Здесь все должно быть понятно без разъяснений.

Кнопка отправки данных

Кнопка отправки данных — пожалуй, второй по популярности элемент управления из применяемых в Web-формах. Она создается одинарным тегом `<INPUT>`.

```
<INPUT TYPE="submit" [VALUE="<надпись>"]
↳[TABINDEX="<номер в порядке обхода>"] [ACCESSKEY="<быстрая клавиша>"]
↳[DISABLED]>
```

Значение "submit" атрибута `TYPE` указывает Web-обозревателю создать кнопку отправки данных.

Необязательный атрибут `VALUE` в этом случае задает надпись для кнопки. Если он не указан, кнопка отправки данных будет иметь надпись **Отправить** (Submit).

Остальные атрибуты нам уже знакомы.

При отправке данных для кнопки отправки данных будет сформирована пара вида:

```
<имя кнопки>=<надпись кнопки>
```

Реально эти данные никогда не используются. Непонятно, зачем они вообще формируются.

Теперь мы можем полностью написать HTML-код, создающий форму для ввода имени пользователя и пароля.

```
<FORM ACTION="http://www.somesite.ru/bin/program.exe">
  Имя пользователя <INPUT TYPE="text" ID="name" NAME="name"
  MAXLENGTH="20"><BR>
  Пароль <INPUT TYPE="password" ID="pass" NAME="pass" MAXLENGTH="20"><BR>
  <INPUT TYPE="submit" ID="send" NAME="send" VALUE="Вход">
</FORM>
```

Кнопка сброса формы

Кнопка сброса формы применяется крайне редко, но все же давайте выясним, как ее создавать. Делается это все тем же тегом-"многостаночником" `<INPUT>`.

```
<INPUT TYPE="reset" [VALUE="<надпись>"]
↳[TABINDEX="<номер в порядке обхода>"] [ACCESSKEY="<быстрая клавиша>"]
↳[DISABLED]>
```

Значение "reset" атрибута TYPE указывает Web-обозревателю создать кнопку сброса формы.

Необязательный атрибут VALUE в этом случае задает надпись для кнопки. Если он не указан, кнопка сброса формы будет иметь надпись **Сброс** (Reset).

Остальные атрибуты знакомы нам по кнопке отправки данных.

При отправке данных для кнопки сброса формы будет сформирована пара вида:

<имя кнопки>=<надпись кнопки>

Эти данные также никогда не используются.

Дополним форму ввода имени пользователя и пароля кнопкой сброса. Просто для практики.

```
<FORM ACTION="http://www.somesite.ru/bin/program.exe">
  Имя пользователя <INPUT TYPE="text" ID="name" NAME="name"
  MAXLENGTH="20"><BR>
  Пароль <INPUT TYPE="password" ID="pass" NAME="pass" MAXLENGTH="20"><BR>
  <INPUT TYPE="submit" ID="send" NAME="send" VALUE="Вход"><BR>
  <INPUT TYPE="reset" ID="res" NAME="res">
</FORM>
```

Кнопка

HTML также позволяет создать обычную кнопку, не являющуюся ни кнопкой отправки данных, ни кнопкой сброса формы. Нажатие такой кнопки может быть обработано с помощью обработчика события onClick этой кнопки. Для ее создания также используется тег <INPUT>.

```
<INPUT TYPE="button" VALUE="<надпись>"
↳[TABINDEX="<номер в порядке обхода>"] [ACCESSKEY="<быстрая клавиша>"]
↳[DISABLED]>
```

Значение "button" атрибута TYPE указывает Web-обозревателю создать обычную кнопку. Атрибут VALUE, задающий надпись для кнопки, в этом случае является обязательным. Остальные атрибуты нам уже знакомы.

При отправке данных для обычной кнопки будет сформирована пара вида:

<имя кнопки>=<надпись кнопки>

Пример:

```
<FORM ACTION="http://www.somesite.ru/bin/program.exe">
  . . .
  <INPUT TYPE="button" ID="btn" NAME="btn" VALUE="Кнопка"
```

```
  ONCLICK="btnClick();">
  . . .
</FORM>
```

Здесь `btnClick` — функция, объявленная где-то на данной странице.

Флажок

Флажки используются в формах нечасто, в случаях, когда нужно дать посетителю возможность выбрать или не выбрать какую-то возможность. Для создания флажков применяется тег `<INPUT>`.

```
<INPUT TYPE="checkbox" [VALUE="<отправляемое значение>"] [CHECKED]
[TABINDEX="<номер в порядке обхода>"] [ACCESSKEY="<быстрая клавиша>"]
[DISABLED]>
```

Значение "checkbox" атрибута `TYPE` указывает Web-обозревателю создать именно флажок.

Необязательный атрибут `VALUE` задает значение, которое будет отправлено серверной программе или другой странице в случае, если флажок включен. Если этот атрибут не указан, флажок получит значение по умолчанию — "on".

Атрибут без значения `CHECKED` позволяет сделать флажок изначально включенным. Если этот атрибут присутствует, флажок будет включен изначально, если отсутствует — отключен.

Остальные атрибуты нам уже знакомы.

Как говорилось ранее, для включенного флажка при отправке данных будет сформирована пара вида:

```
<имя флажка>=<значение флажка>
```

Если флажок не был включен, такая пара не формируется и соответствующие флажку данные не отправляются. Об этом не следует забывать.

Вот пример использования флажка в форме:

```
<FORM ACTION="http://www.somesite.ru/bin/program.exe">
  . . .
  Я хочу получать письма со списком обновлений сайта
  <INPUT TYPE="checkbox" ID="chk" NAME="chk">
  . . .
</FORM>
```

Поскольку атрибут `VALUE` не указан, серверной программе будет отправлено значение "on". Этого будет достаточно, чтобы понять, включен ли флажок.

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 — гораздо удобнее в таких случаях использовать флажки.

А теперь — очень важная вещь! Ранее мы говорили, что каждый элемент управления должен иметь уникальное имя и что единственное исключение из этого правила — как раз переключатели. Так вот, имя задается для всей группы переключателей. Иными словами, переключатели, входящие в одну группу, должны иметь одинаковое имя, и это имя должно быть уникально в пределах формы.

Создается переключатель с помощью все того же тега `<INPUT>`.

```
<INPUT TYPE="radio" VALUE="<отправляемое значение>" [CHECKED]
[TABINDEX="<номер в порядке обхода>"] [ACCESSKEY="<быстрая клавиша>"]
[DISABLED]>
```

Значение "radio" атрибута `TYPE` указывает Web-обозревателю создать именно переключатель.

Атрибут `VALUE` задает значение, которое будет отправлено серверной программе или другой странице в случае, если данный переключатель включен. В этом случае атрибут `VALUE` обязателен.

Остальные атрибуты нам уже знакомы.

Для группы переключателей при отправке данных будет сформирована пара вида:

```
<имя группы переключателей>=<значение включенного переключателя>
```

Если ни один переключатель в группе не был включен, такая пара не формируется и соответствующие переключателю данные не отправляются.

> ***Внимание!***
>
> Хоть один переключатель в группе должен быть включен. Группа переключателей, в которой все переключатели отключены, — плохой стиль Web-программирования.

Приведем пример формы с группой переключателей.

```
<FORM ACTION="http://www.somesite.ru/bin/program.exe">
  . . .
  Укажите ваш пол:
  <INPUT TYPE="radio" ID="gender" NAME="gender" VALUE="m" CHECKED>
```

```
  мужской,
  <INPUT TYPE="radio" ID="gender" NAME="gender" VALUE="f"> женский
  . . .
</FORM>
```

Обратим внимание, что оба переключателя имеют одно имя — `gender`. Это значит, что они входят в одну группу. Первый переключатель этой группы включен изначально. Если он останется включенным, то серверной программе будет отправлено заданное для него значение — "m"; если посетитель включит второй переключатель, отправится значение второго переключателя — "f".

Список, обычный или раскрывающийся

Списки, как обычные, так и раскрывающиеся, создаются с помощью парного тега `<SELECT>`. Внутри этого тега помещаются 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]>
  . . . Теги <OPTION>, создающие пункты списка
</SELECT>
```

Необязательный атрибут `SIZE` задает высоту списка в пунктах (то есть позициях). Если этот атрибут имеет значение, отличное от 1, создается обычный список, имеющий такую высоту, чтобы полностью вместить в себя указанное в этом атрибуте количество пунктов. Если же этот атрибут имеет значение 1 или вообще отсутствует, создается раскрывающийся список.

Атрибут без значения `MULTIPLE` позволяет создать список, в котором можно выбрать сразу несколько пунктов. Если этот атрибут присутствует, в списке можно будет выбрать сразу несколько пунктов, если отсутствует — можно будет выбрать только один пункт. Понятно, что этот атрибут имеет смысл указывать только для обычных списков (если атрибут `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 [VALUE="<значение пункта>"] [SELECTED] [DISABLED]>
  <текст пункта>
<OPTION>
```

Текст пункта списка помещается внутри тега <OPTION>.

Необязательный атрибут VALUE задает значение, которое будет отправлено серверной программе или другой странице при выборе этого пункта. Если этот атрибут не указан, будет отправлен текст выбранного пункта.

Атрибут без значения SELECTED позволяет сделать данный пункт списка изначально выбранным. Если этот атрибут указан, пункт списка будет изначально выбранным, если не указан — не будет выбранным.

Уже знакомый нам атрибут без значения DISABLED позволяет сделать данный пункт недоступным для выбора.

Для списка при отправке данных будет сформирована пара вида:

<имя списка>=<значение выбранного пункта>

Если список позволяет выбрать сразу несколько пунктов, то указанная ранее пара будет сформирована для каждого из выбранных пунктов. Если же в списке не был выбран ни один пункт, то никаких данных отправлено не будет.

Пример формы со списком:

```
<FORM ACTION="http://www.somesite.ru/bin/program.exe">
  . . .
  Выберите ваш любимый язык программирования<BR>
  <SELECT ID="prlang" NAME="prlang" SIZE="4">
    <OPTION>JavaScript</OPTION>
    <OPTION>VBScript</OPTION>
    <OPTION VALUE="VB">Visual Basic</OPTION>
    <OPTION VALUE="Delphi">CodeGear Delphi</OPTION>
  </SELECT>
  . . .
</FORM>
```

Здесь мы формируем обычный список высотой в четыре пункта — как раз столько, сколько в нем задано. Первые два пункта не имеют значения (атрибут VALUE тега <OPTION> не указан), поэтому в их случае серверной программе будет отправлен текст этих пунктов. Третий и четвертый пункты списка имеют значения, которые и будут отправлены серверной программе.

Поле ввода имени файла

Поле ввода имени файла — совершенно особый элемент управления. Он позволяет выбрать файл, располагающийся на клиентском компьютере, и отправить этот файл (именно файл — не его имя!) серверной программе. Визуально он представляет собой обычное поле ввода, правее которого расположена

кнопка **Обзор** (Browse). При нажатии этой кнопки на экране появится стандартное диалоговое окно Windows открытия файла, в котором посетитель сможет выбрать файл, подлежащий отправке.

Если мы планируем реализовать на странице возможность отправки файлов какой-либо серверной программе, то должны выполнить два условия.

1. Для формы должен быть задан метод передачи данных POST.
2. Для формы должен быть выбран метод кодирования данных `multipart/form-data`.

Поле ввода имени файла создается тегом `<INPUT>`.

```
<INPUT TYPE="file" [SIZE="<размер>"]
↳[TABINDEX="<номер в порядке обхода>"] [ACCESSKEY="<быстрая клавиша>"]
↳[DISABLED]>
```

Значение "file" атрибута `TYPE` указывает Web-обозревателю создать именно поле ввода имени файла. Остальные атрибуты нам давно знакомы.

Для поля ввода имени файла при отправке данных будет сформирована пара вида:

```
<имя поля ввода>=<имя выбранного файла и его содержимое>
```

Обработать эти данные сможет только специально написанная серверная программа. Если же файл в этом поле выбран не был, никакие данные отправлены не будут.

Пример формы с полем ввода имени файла очень прост:

```
<FORM ACTION="http://www.somesite.ru/bin/program.exe">
  Выберите файл:<BR>
  <INPUT TYPE="file" ID="fl" NAME="fl" SIZE="200">
  . . .
</FORM>
```

Графическая кнопка отправки данных

Графическая кнопка отправки данных используется в формах довольно редко. Это обычная кнопка отправки данных, но представляющая собой графическое изображение; при щелчке на этом изображении форма начинает процесс передачи данных.

Создается графическая кнопка отправки данных тегом `<INPUT>`.

```
<INPUT TYPE="image"
↳SRC="<интернет-адрес графического файла с изображением>"
↳[ALT="<текст замены>"]
```

[LOWSRC="*<интернет-адрес файла с изображением пониженного качества>*"]
[TABINDEX="*<номер в порядке обхода>*"] [ACCESSKEY="*<быстрая клавиша>*"]
[DISABLED]>

Значение "image" атрибута TYPE указывает Web-обозревателю создать графическую кнопку отправки данных.

Здесь следует рассмотреть только атрибуты, задающие параметры графического изображения и знакомые нам по тегу <IMG> (см. *главу 2*). Это атрибуты SRC (интернет-адрес графического изображения), ALT (текст замены) и LOWSRC (интернет-адрес изображения пониженного качества, которое будет отображаться, пока не загрузится полноценное изображение, чей интернет-адрес задан атрибутом SRC). Остальные атрибуты знакомы нам по другим элементам управления.

Для графической кнопки отправки данных форма сформирует целых две пары вида:

<имя графической кнопки>.x=*<X>*
<имя графической кнопки>.y=*<Y>*

где X и Y — координаты точки на графической кнопке, на которой посетитель щелкнул мышью, в пикселах; они отсчитываются от верхнего левого угла кнопки. Серверная программа или другая страница может как-то обрабатывать эти значения, а может и игнорировать.

Пример формы с графической кнопкой отправки данных:

```
<FORM ACTION="http://www.somesite.ru/bin/program.exe">
  Имя пользователя <INPUT TYPE="text" ID="name" NAME="name"
  MAXLENGTH="20"><BR>
  Пароль <INPUT TYPE="password" ID="pass" NAME="pass" MAXLENGTH="20">
  <INPUT TYPE="file" ID="send" NAME="send" SRC="sendbutton.gif"
  ATL="Щелкните для входа на сайт"><BR>
  . . .
</FORM>
```

Это уже рассмотренная нами ранее форма ввода имени пользователя и пароля для входа на закрытый сайт, но с использованием графической кнопки вместо обычной кнопки отправки данных.

Скрытое поле

Собственно, *скрытое поле* — это даже не элемент управления, поскольку оно вообще не отображается на странице. Скрытые поля обычно используются для отправки серверной программе данных, так сказать, внутреннего пользования (например, внутреннего идентификатора посетителя, выполнившего успешный вход на сайт), которые не нужно афишировать.

Скрытое поле создается тегом <INPUT>.

```
<INPUT TYPE="hidden" VALUE="<значение>">
```

Значение "hidden" атрибута TYPE указывает Web-обозревателю создать именно поле ввода имени файла. Атрибут VALUE нам давно знаком.

Для скрытого поля форма всегда формирует пару вида:

```
<имя скрытого поля>=<значение скрытого поля>
```

Для примера рассмотрим фрагмент формы со скрытым полем:

```
<FORM ACTION="http://www.somesite.ru/bin/program.exe">
  . . .
  Выберите способ оплаты покупок:
  <INPUT TYPE="radio" ID="payment" NAME="payment" VALUE="p" CHECKED>
  наложенным платежом,
  <INPUT TYPE="radio" ID="payment" NAME="payment" VALUE="t">
  банковским переводом
  <INPUT ID="cid" NAME="cid" VALUE="6534267">
  . . .
</FORM>
```

Это форма для ввода способа оплаты покупок в интернет-магазине. Здесь скрытое поле cid хранит внутренний идентификатор зарегистрированного посетителя, выполнившего успешный вход. Такие идентификаторы формируются самой серверной программой, обеспечивающей функциональность интернет-магазина, фактически являются закрытой информацией и не должны отображаться на странице. Когда посетитель введет данные и нажмет кнопку отправки, серверная программа получит идентификатор и сразу определит, что и какой посетитель отправил эти данные.

Надпись

Надпись — это особый элемент страницы, задающий для данного элемента управления текст, который описывает его назначение. Никаких данных серверной программе или другой странице она не отправляет.

Надпись создается с помощью парного тега <LABEL>.

```
<LABEL FOR="<имя элемента управления, к которому относится надпись>"
↳[TABINDEX="<номер в порядке обхода>"] [ACCESSKEY="<быстрая клавиша>"]>
  <текст надписи>
</LABEL>
```

Сам текст надписи указывается внутри этого тега. Он может быть отформатирован с использованием практически любых тегов HTML.

Обязательный атрибут FOR задает имя элемента управления, к которому относится надпись. Уже знакомые нам необязательные атрибуты TABINDEX и ACCESSKEY задают номер в порядке обхода и быструю клавишу для элемента управления, к которому относится надпись.

Пример формы с использованием надписей:

```
<FORM ACTION="http://www.somesite.ru/bin/program.exe">
  <LABEL FOR="name" TABINDEX="1"><STRONG>Имя</STRONG></LABEL>
  <INPUT TYPE="text" ID="name" NAME="name" MAXLENGTH="20"><BR>
  <LABEL FOR="password" TABINDEX="2"><STRONG>Пароль</STRONG></LABEL>
  <INPUT TYPE="password" ID="pass" NAME="pass" MAXLENGTH="20"><BR>
  <INPUT TYPE="submit" ID="send" NAME="send" VALUE="Вход">
</FORM>
```

Здесь текст надписей выделен полужирным шрифтом с помощью тегов <STRONG>.

Надписи в формах применяются очень редко, по крайней мере, автору они не встречались ни разу. Никакой дополнительной функциональности по сравнению с обычным текстом они не предлагают.

Группа

Группа — особый элемент страницы, позволяющий объединить несколько элементов управления по их назначению. Визуально она представляет собой рамку, окружающую элементы управления и, возможно, имеющую заголовок, расположенный прямо на ее верхней или нижней границе. Никаких данных серверной программе или другой странице она не отправляет.

Группа создается с помощью парного тега <FIELDSET>.

```
<FIELDSET>
  . . .Элементы управления, объединяемые в группу
</FIELDSET>
```

Видно, что теги, создающие элементы управления, которые должны быть объединены в группу, помещаются прямо в тег <FIELDSET>. Никаких полезных для нас атрибутов этот тег не поддерживает.

Кроме того, в теге <FIELDSET> может присутствовать парный тег <LEGEND>, создающий заголовок группы.

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

Текст заголовка помещается прямо внутри этого тега. Необязательный атрибут ACCESSKEY задает быструю клавишу для всей группы; при ее нажатии будет активизирован первый элемент управления в этой группе.

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.

И — пример формы, в которой используется группа.

```
<FORM ACTION="http://www.somesite.ru/bin/program.exe">
  <FIELDSET>
    <LEGEND>Имя и пароль</LEGEND>
    Имя <INPUT TYPE="text" ID="name" NAME="name" MAXLENGTH="20"><BR>
    Пароль <INPUT TYPE="password" ID="pass" NAME="pass" MAXLENGTH="20">
  </FIELDSET>
  . . .
</FORM>
```

Здесь группа включает поля ввода имени и пароля посетителя. Заголовок будет присутствовать на верхней границе группы, так как создающий его тег <LEGEND> находится сразу же после открывающего тега <FIELDSET>.

Группы применяются в формах достаточно редко. Того же успеха можно достичь, поместив нужные элементы управления в блочный контейнер и задав для него рамку, тем более что в таком случае мы сами можем управлять параметрами рамки и заголовка. К тому же, как правило, Web-формы содержат элементы управления, уже имеющие общее назначение, так что группировать их дополнительно нет смысла.

Это все элементы управления, поддерживаемые HTML.

Примеры Web-форм и страниц, получающих данные от Web-форм

Поскольку материал, посвященный Web-формам и элементам управления, весьма велик, давайте рассмотрим пару примеров, чтобы закрепить полученные знания. Каждый из этих примеров будет включать две страницы: одна будет содержать форму и, следовательно, отправлять данные, другая — принимать их и выводить на экран.

Web-форма для сбора анкетных данных

Начнем с примера попроще. А именно с формы, принимающей от посетителя некие анкетные данные. Пусть это будет имя и фамилия посетителя, его адрес электронной почты, пол, любимый язык программирования и признак,

работает ли он программистом профессионально. Для ввода имени, фамилии и адреса электронной почты мы используем поля ввода, для указания пола — набор переключателей, языка программирования — раскрывающийся список, а принадлежности посетителя к профессионалам от программирования будет достаточно флажка.

HTML-код страницы с формой приведен далее.

```
<HTML>
  <HEAD>
    <TITLE>Анкета</TITLE>
  </HEAD>
  <BODY>
    <FORM ACTION="12.2.htm" METHOD="get">
      Имя <INPUT TYPE="text" ID="name1" NAME="name1" MAXLENGTH="20"><BR>
      Фамилия <INPUT TYPE="text" ID="name2" NAME="name2"
      MAXLENGTH="20"><BR>
      E-mail <INPUT TYPE="text" ID="email" NAME="email" SIZE="50"><BR>
      Пол: <INPUT TYPE="radio" ID="gender" NAME="gender" VALUE="m"
      CHECKED> муж.,
      <INPUT TYPE="radio" ID="gender" NAME="gender" VALUE="f"> жен.<BR>
      Любимый язык программирования<BR>
      <SELECT ID="pl" NAME="pl">
        <OPTION>JavaScript</OPTION>
        <OPTION>VBScript</OPTION>
        <OPTION VALUE="VB">Visual Basic</OPTION>
        <OPTION VALUE="Delphi">CodeGear Delphi</OPTION>
      </SELECT><BR>
      <INPUT TYPE="checkbox" ID="profi" NAME="profi">
      Я - профессиональный программист<BR>
      <INPUT TYPE="submit" ID="sub" NAME="sub" VALUE="Ввод">
    </FORM>
  </BODY>
</HTML>
```

В принципе, полученных нами знаний хватит, чтобы понять, что делает этот код. (Тем более что сценариев здесь вообще нет.) Отметим только, что созданная на этой странице форма передает данные странице 12.2.htm (мы напишем еще чуть позже) методом GET — это единственный метод, позволяющий передать данные Web-странице.

Сохраним эту страницу в файле под именем 12.1.htm. И рассмотрим HTML-код страницы, которая примет данные от страницы 12.1.htm и выведет их на экран.

```
<HTML>
  <HEAD>
    <TITLE>Обработка анкеты</TITLE>
    <SCRIPT>
      //Здесь должен присутствовать код, объявляющий функцию getData.
      //Этот код можно найти в главе 11, в параграфе, посвященном
      //передаче данных
    </SCRIPT>
  </HEAD>
  <BODY>
    <P ID="name1"> </P>
    <P ID="name2"> </P>
    <P ID="email"> </P>
    <P ID="gender"> </P>
    <P ID="pl"> </P>
    <P ID="profi"> </P>
    <SCRIPT>
      var name1Obj = document.getElementById("name1");
      var name1 = getData("name1");
      name1Obj.firstChild.nodeValue = name1;
      var name2Obj = document.getElementById("name2");
      var name2 = getData("name2");
      name2Obj.firstChild.nodeValue = name2;
      var emailObj = document.getElementById("email");
      var email = getData("email");
      emailObj.firstChild.nodeValue = email;
      var genderObj = document.getElementById("gender");
      var gender = getData("gender");
      var s = "Пол - ";
      if (gender == "m")
        genderObj.firstChild.nodeValue = s + "мужской"
      else
        genderObj.firstChild.nodeValue = s + "женский";
      var plObj = document.getElementById("pl");
      var pl = getData("pl");
      plObj.firstChild.nodeValue = "Любимый язык программирования - " +
```

```
      pl;
      var profi = getData("profi");
      if (profi) {
        var profiObj = document.getElementById("profi");
        profiObj.firstChild.nodeValue = "Профессиональный программист";
      }
    </SCRIPT>
  </BODY>
</HTML>
```

Здесь также нечего комментировать, кроме того, что для "разбора" принятой от страницы 12.1.htm строки параметров, содержащей введенные посетителем данные, мы используем написанную в *главе 11* функцию `getData`. В то место, где присутствует предписывающий это сделать комментарий, мы должны вставить код, объявляющий эту функцию, иначе страница не будет работать.

Сохраним эту страницу в файле под именем 12.2.htm. Загрузим в Web-обозреватель страницу 12.1.htm, введем какие-либо данные в форму и нажмем кнопку **Ввод**. Web-обозреватель загрузит страницу 12.2.htm, которая отобразит введенные данные (если, конечно, мы не допустили в коде обеих страниц ошибок).

Web-форма со списком, позволяющим выбирать сразу несколько пунктов

Настала пора рассмотреть более сложный пример. Давайте создадим форму со списком, предусматривающим возможность выбора сразу нескольких пунктов. Этот список будет содержать языки программирования. А вторая страница, обрабатывающая введенные в эту форму данные, будет выводить выбранные языки программирования на экран.

Для получения выбранных в списке пунктов из строки параметров нам пригодится функция `getSelectedOptions`, которую мы сейчас напишем.

```
getSelectedOptions(<имя списка>);
```

Эта функция будет принимать единственный параметр — имя списка, значения выбранных пунктов которого нужно получить. Возвращать она будет массив, содержащая эти значения; если ни один из пунктов выбран не был, возвращается `null`.

```
function getSelectedOptions(pName) {
  var sSearch = location.search;
  var sName = pName + "=";
```

```
  var j = 0;
  var optIndex = -1;
  var opts = [];
  var i = sSearch.indexOf(sName);
  while (i > -1) {
    j = sSearch.indexOf("&", i + sName.length);
    if (j == -1) j = sSearch.length;
    optIndex++;
    opts[optIndex] = unescape(sSearch.substring(i + sName.length, j));
    i = sSearch.indexOf(sName, j);
  }
  if (optIndex > -1)
    return opts
  else
    return null;
}
```

Здесь мы последовательно ищем в строке параметров все значения с именем, заданным параметром `pName`, и добавляем их в массив `opts`. Переменная `optIndex` хранит индекс последнего добавленного в массив элемента и инкрементируется перед добавлением нового элемента. В конце мы проверяем, присутствует ли в массиве `opts` хоть один элемент (превышает ли значение переменной `optIndex` –1 — ее изначальное значение), и, если присутствует, возвращаем его; в противном случае возвращаем `null`. Так или иначе, эта функция не очень сложна, так что попробуйте разобраться в ней самостоятельно.

HTML-код страницы с формой приведен далее. Он совсем невелик.

```
<HTML>
  <HEAD>
    <TITLE>Анкета</TITLE>
  </HEAD>
  <BODY>
    <FORM ACTION="12.4.htm" METHOD="get">
      Языки программирования<BR>
      <SELECT ID="pl" NAME="pl" SIZE="4" MULTIPLE>
        <OPTION>JavaScript</OPTION>
        <OPTION>VBScript</OPTION>
        <OPTION VALUE="VB">Visual Basic</OPTION>
        <OPTION VALUE="Delphi">CodeGear Delphi</OPTION>
```

```
      </SELECT><BR>
      <INPUT TYPE="submit" ID="sub" NAME="sub" VALUE="Ввод">
    </FORM>
  </BODY>
</HTML>
```

Присутствующая на этой странице форма отправит данные странице 12.4.htm, которую мы скоро создадим. А пока что сохраним страницу с формой в файле с именем 12.3.htm.

А вот HTML-код страницы, которая будет принимать и обрабатывать данные, введенные на странице 12.3.htm.

```
<HTML>
  <HEAD>
    <TITLE>Обработка анкеты</TITLE>
    <SCRIPT>
      //Здесь должен присутствовать код, объявляющий функцию
      //getSelectedOptions
    </SCRIPT>
  </HEAD>
  <BODY>
    <P ID="pl"> </P>
    <SCRIPT>
      var pl = getSelectedOptions("pl");
      if (pl) {
        var plObj = document.getElementById("pl");
        plObj.firstChild.nodeValue = pl.join(", ");
      }
    </SCRIPT>
  </BODY>
</HTML>
```

Здесь мы выделяем из строки параметров значения, соответствующие выбранным в списке `pl` страницы 12.3.htm пунктам, формируем из них строку, где они будут присутствовать, разделенные символом-разделителем ", " (запятая и пробел), с помощью метода `join` объекта `String` (см. *главу 4*) и выводим на экран. Поскольку для выделения значений выбранных пунктов используется написанная нами ранее функция `getSelectedOptions`, нам надо будет включить объявляющий ее код в место, где находится предписывающий это сделать комментарий.

Сохраним эту страницу в файле под именем 12.4.htm. Загрузим в Web-обозреватель страницу 12.3.htm, выберем какие-либо пункты в списке и на-

жмем кнопку **Ввод**. Web-обозреватель загрузит страницу 12.4.htm, которая выведет значения, соответствующие всем выбранным нами пунктам.

На этом можно закончить рассмотрение средств HTML, предназначенных для создания Web-форм и элементов управления. Начнем разговор о средствах, предоставляемых Web-обозревателями для программной работы с ними. Мы ведь Web-программисты, и нам это обязательно понадобится!

Работа с Web-формами и элементами управления из сценариев

И Web-формы, и элементы управления представляются особыми объектами. Пользуясь свойствами и методами этих объектов, мы можем управлять ими. А если мы еще задействуем события, то сможем реагировать на действия посетителя. Все как обычно.

Работа с Web-формами

Чтобы получить доступ к форме из сценариев, можно воспользоваться свойствами и методами DOM, изученными нами еще в *главе 5*. Ими чаще всего как раз и пользуются.

Кроме свойств и методов DOM, мы можем использовать коллекцию `forms`. Эта коллекция доступна через одноименное свойство объекта `HTMLDocument`, содержит все присутствующие на странице формы, представляет собой экземпляр объекта `HTMLCollection` и поддерживает все свойства и средства доступа к ее элементам, характерные для других известных нам коллекций.

```
var firstFormObj = document.forms[0];
```

Это выражение присвоит переменной `firstFormObj` первую форму из присутствующих на странице.

```
var frmObj = document.forms["frm"];
```

Это выражение поместит в переменную `frmObj` форму с именем `frm`.

```
var controlCount = document.forms.lenght;
```

А это выражение поместит в переменную `controlCount` количество форм, присутствующих на странице. Как мы помним, поддерживаемое всеми коллекциями свойство `length` возвращает количество элементов коллекции, что в случае коллекции `forms` как раз и будет количеством форм на данной странице.

Форму представляет объект `HTMLFormElement`, созданный на основе объекта `HTMLElement` и наследующий его свойства, методы и события. Кроме того,

этот объект предоставляет свои свойства, методы и события, которые мы сейчас рассмотрим.

Прежде всего, это свойства `action`, `autocomplete`, `enctype`, `method` и `target`, соответствующие одноименным атрибутам тега `<FORM>`. Они задают или возвращают интернет-адрес серверной программы или другой страницы, которой будут отправлены данные, состояние функции автозаполнения, метод кодирования данных, метод передачи данных и цель соответственно в виде строки. Не забываем, что свойство `autocomplete`, как и одноименный атрибут, поддерживается только Internet Explorer.

```
frmObj.action = "http://www.othersite.ru/bin/program.exe";
```

Это выражение задает для формы `frm`, ссылка на которую хранится в переменной `frmObj`, другую серверную программу.

```
frmObj.autocomplete = "on";
```

А это выражение включает для той же формы режим автозаполнения.

Свойство `elements` предоставляет доступ к одноименной коллекции, содержащей все присутствующие в форме элементы управления. Эта коллекция представляет собой экземпляр объекта `HTMLCollection` и поддерживает все свойства и средства доступа к ее элементам, характерные для других известных нам коллекций.

```
var lastControlObj = frmObj.elements[frmObj.elemtns.length - 1];
```

Это выражение поместит в переменную `lastControlObj` последний элемент управления, присутствующий в форме `frmObj`. Мы получаем количество элементов коллекции `elements`, то есть количество элементов управления в данной форме, через свойство `length`, и вычитаем из него единицу, так как нумерация элементов коллекции начинается с нуля, и последний элемент будет иметь индекс, на единицу меньший, чем количество элементов управления в форме.

```
var name1Obj = frmObj.elements["name1"];
```

А это выражение поместит в переменную `name1Obj` элемент управления с именем `name1`.

Объект `HTMLFormElement` также поддерживает два очень полезных метода. Метод `submit` выполняет отправку введенных данных серверной программе или другой странице; его вызов аналогичен нажатию кнопки отправки данных. А метод `reset` очищает форму, подставляя во все ее элементы управления изначальные значения; он аналогичен нажатию кнопки сброса формы. Оба этих метода не принимают параметров и не возвращают результата.

Что касается событий, поддерживаемых объектом `HTMLFormElement`, то все они перечислены в табл. 12.2.

Таблица 12.2. *События, поддерживаемые объектом* `HTMLFormElement`

Событие	Описание
`onActivate`	Возникает, когда любой элемент управления в форме становится активным. Всплывает, но поведение по умолчанию (активация элемента управления) не может быть отменено. Поддерживается только Internet Explorer
`onBeforeActivate`	Возникает перед тем, как любой элемент управления в форме станет активным. Всплывает, поведение по умолчанию (активация элемента управления) может быть отменено. Поддерживается только Internet Explorer
`onBeforeDeactivate`	Возникает перед тем, как любой элемент управления в форме перестанет быть активным. Всплывает, поведение по умолчанию (деактивация элемента управления) может быть отменено. Поддерживается только Internet Explorer
`onDeactivate`	Возникает, когда любой элемент управления в форме перестает быть активным. Всплывает, но поведение по умолчанию (деактивация элемента управления) не может быть отменено. Поддерживается только Internet Explorer
`onFocusIn`	Возникает перед тем, как любой элемент управления в форме станет активным. Всплывает, но поведение по умолчанию (активация элемента управления) не может быть отменено — этим оно отличается от события `onBeforeActivate`. Поддерживается только Internet Explorer
`onFocusOut`	Возникает сразу после того, как любой элемент управления в форме перестанет быть активным. Всплывает, но поведение по умолчанию (деактивация элемента управления) не может быть отменено. Поддерживается только Internet Explorer
`onReset`	Возникает при очистке формы в результате нажатия кнопки сброса или вызова метода `reset` объекта `HTMLFormElement`. Не всплывает, поведение по умолчанию (очистка формы) может быть отменено
`onSubmit`	Возникает при отправке данных в результате нажатия кнопки отправки или вызова метода `submit` объекта `HTMLFormElement`. Не всплывает, поведение по умолчанию (отправка данных) может быть отменено

Здесь нужно дать некоторые пояснения. Когда посетитель активизирует какой-либо элемент управления в форме (щелчком мышью, нажатием клавиши <Tab> или комбинаций клавиш <Shift>+<Tab>), события в форме возникают в такой последовательности:

- `onBeforeActivate`;
- `onActivate`;
- `onFocusIn`.

Когда посетитель деактивизирует активный до этого времени элемент управления (щелчком мышью вне формы, нажатием клавиши <Tab> или комбинаций клавиш <Shift>+<Tab>), в форме возникнет такая последовательность событий:

- `onBeforeDeactivate`;
- `onDeactivate`;
- `onFocusOut`.

В обработчиках событий `onBeforeActivate` и `onBeforeDeactivate` мы можем отменить поведение по умолчанию — активацию или деактивацию элемента управления соответственно. Если мы создаем какое-то Web-решение, предназначенное для работы исключительно в Internet Explorer, то можем этим воспользоваться.

Вообще, в основном в формах обрабатывают события `onSubmit` и `onReset`. Во-первых, они поддерживаются всеми Web-обозревателями, а во-вторых, от них больше пользы. Так, в обработчике события `onSubmit` мы можем выполнить проверку корректности введенных в форму данных и, если данные некорректны, прервать их отправку и предупредить посетителя. В обработчике события `onReset` мы можем, например, подставлять в определенные поля ввода не изначальные значения, заданные в HTML-коде (атрибут `VALUE` тега `<INPUT>`), а значения, вычисленные в сценарии.

Вот простой пример формы с обработчиком события `onSubmit`:

```
<SCRIPT>
  function frmSubmit() {
    var frmObj = document.forms["frm"];
    frmObj.action = "http://www.othersite.ru/bin/program.exe";
    frmObj.submit();
  }
</SCRIPT>
  . . .
<FORM ID="frm" NAME="frm" ACTION="#"
```

```
ONSUBMIT="frmSubmit(); return false;">
  . . .
</FORM>
```

Обратим внимание, что в теге `<FORM>` не задан интернет-адрес серверной программы, которой должны быть отправлены введенные данные. Этот интернет-адрес мы задаем в обработчике события `onSubmit` — функции `frmSubmit` — и там же выполняем отправку данных вызовом метода `submit`. Кроме того, мы отменяем поведение по умолчанию для события `onSubmit`, чтобы форма случайно не отправила данные "в никуда", для чего вставляем прямо в обработчик данного события выражение `return false`.

Работа с элементами управления

Как правило, сценарии, реализующие обработку введенных в форму данных, работают непосредственно с элементами управления. Это единственный способ для них "добраться" до нужных данных. Так что нам следует уделить элементам управления больше времени, нежели формам.

Свойства, методы и события, общие для всех элементов управления

Различные элементы управления, поддерживаемые HTML, представляются разными объектами DOM. Все эти объекты порождены от объекта `HTMLElement`, а значит, поддерживают все его свойства, методы и события. И, разумеется, предоставляют свои свойства, методы и события, которые мы обязательно рассмотрим.

Начнем мы со свойств, методов и событий, общих для всех элементов управления. Их набор довольно обширен.

Все общие для всех элементов управления свойства перечислены в табл. 12.3.

Таблица 12.3. *Свойства, общие для всех элементов управления*

Свойство	Описание
`accessKey`	Задает или возвращает быструю клавишу в виде строки. Поддерживается всеми элементами управления, кроме скрытого поля и группы
`disabled`	Если `true`, элемент управления недоступен, если `false` — доступен для ввода. Поддерживается всеми элементами управления, кроме скрытого поля, надписи и группы
`form`	Возвращает форму, в которой находится данный элемент управления

Таблица 12.3 *(окончание)*

Свойство	Описание
`tabIndex`	Задает или возвращает номер в порядке обхода в виде числа. Поддерживается всеми элементами управления, кроме скрытого поля и группы
`type`	Задает или возвращает тип элемента управления, задаваемый атрибутом `TYPE` тега `<INPUT>`, в виде строки. Поддерживается всеми элементами управления, даже теми, что создаются с помощью других тегов (подробности будут приведены далее)

Как видим, в основном, это свойства, соответствующие различным атрибутам тега, с помощью которого создан элемент управления.

Методов, общих для всех элементов управления, всего два: метод `focus` активизирует текущий элемент управления, а метод `blur` делает его неактивным — при этом активизируется следующий в порядке обхода элемент управления. Оба этих метода не принимают параметров и не возвращают результата.

Событий, общих для всех элементов управления, пожалуй, больше всего. Они перечислены в табл. 12.4 и поддерживаются всеми элементами управления, кроме скрытого поля, надписи и группы.

Таблица 12.4. *События, общие для всех элементов управления*

Событие	Описание
`onActivate`	Возникает, когда текущий элемент управления становится активным. Всплывает, но поведение по умолчанию (активация элемента управления) не может быть отменено. Поддерживается только Internet Explorer
`onBeforeActivate`	Возникает перед тем, как текущий элемент управления станет активным. Всплывает, поведение по умолчанию (активация элемента управления) может быть отменено. Поддерживается только Internet Explorer
`onBeforeDeactivate`	Возникает перед тем, как текущий элемент управления перестанет быть активным. Всплывает, поведение по умолчанию (деактивация элемента управления) может быть отменено. Поддерживается только Internet Explorer
`onBlur`	Возникает, когда текущий элемент управления перестает быть активным. Не всплывает, и поведение по умолчанию (деактивация элемента управления) не может быть отменено

Таблица 12.4 *(окончание)*

Событие	Описание
`onDeactivate`	Возникает, когда текущий элемент управления перестает быть активным. Всплывает, но поведение по умолчанию (деактивация элемента управления) не может быть отменено. Поддерживается только Internet Explorer
`onFocus`	Возникает, когда текущий элемент управления становится активным. Не всплывает, и поведение по умолчанию (активация элемента управления) не может быть отменено
`onFocusIn`	Возникает перед тем, как текущий элемент управления станет активным. Всплывает, но поведение по умолчанию (активация элемента управления) не может быть отменено — этим оно отличается от события `onBeforeActivate`. Поддерживается только Internet Explorer
`onFocusOut`	Возникает сразу после того, как текущий элемент управления перестанет быть активным. Всплывает, но поведение по умолчанию (деактивация элемента управления) не может быть отменено. Поддерживается только Internet Explorer

Без пояснений и здесь не обойтись. Когда посетитель активизирует какой-либо элемент управления в форме (щелчком мышью, нажатием клавиши <Tab> или комбинацией клавиш <Shift>+<Tab>) либо это делается в сценарии вызовом метода `focus`, события в данном элементе управления возникают в такой последовательности:

1. `onBeforeActivate`.
2. `onActivate`.
3. `onFocusIn`.
4. `onFocus`.

Разумеется, это справедливо только для Internet Explorer. В случае других Web-обозревателей возникнет только событие `onFocus`.

Когда посетитель деактивизирует активный до этого времени элемент управления (щелчком мышью на другом элементе управления или вне формы, нажатием клавиши <Tab> или комбинацией клавиш <Shift>+<Tab>) либо это делается в сценарии вызовом метода `blur`, в данном элементе управления возникнет такая последовательность событий:

1. `onBeforeDeactivate`.
2. `onDeactivate`.

3. onFocusOut.

4. onBlur.

Это в случае Internet Explorer — другие Web-обозреватели "отметятся" только событием onBlur.

В обработчиках событий onBeforeActivate и onBeforeDeactivate мы можем отменить поведение по умолчанию — активацию или деактивацию элемента управления соответственно. Правда, сработает это только в Internet Explorer.

Вот и все о свойствах, методах и событиях, общих для всех элементов управления. Теперь поговорим о свойствах, методах и событиях, специфичных для определенных элементов управления.

Поле ввода

Поле ввода представляется экземпляром объекта HTMLInputElement (как и все элементы управления, создаваемые тегом <INPUT>). Этот объект поддерживает несколько свойств, методов и событий, которые могут нам пригодиться.

Прежде всего, это свойство value. Оно задает или возвращает текущее значение, введенное в поле ввода.

```
var name1Obj = document.getElementById("name1");
var str = name1Obj.value;
name1Obj.value = "Vasya";
```

Первое выражение присваивает переменной str текущее значение поля ввода name1. Второе выражение помещает в это поле ввода новое значение.

Свойство defaultValue задает или возвращает изначальное значение, которое должно присутствовать в поле ввода при загрузке страницы или после выполнения сброса формы. В HTML-коде это значение задается атрибутом VALUE тега <INPUT>.

```
var name2Obj = document.getElementById("name2");
name2Obj.defaultValue = "Pupkin";
```

Это выражение задает для поля ввода name2 новое изначальное значение.

Свойства autocomplete, maxLength, readOnly и size соответствуют одноименным атрибутам тега <INPUT> и имеют то же назначение. О назначении этих атрибутов было рассказано в параграфе, посвященном созданию полей ввода. Свойства maxLength и size принимают и возвращают значения в числовом виде, свойство readOnly — в логическом, а свойство autocomplete — в строковом.

```
name1Obj.size = 100;
name1Obj.readOnly = true;
```

Этот сценарий задает ширину поля ввода `name1` равной 100 символам и делает его доступным только для чтения (просто для примера).

Метод `select` выделяет все содержимое поля ввода. Он не принимает параметров и не возвращает значения.

```
name2Obj.select();
```

Событие `onChange` возникает при любом изменении значения в поле ввода посетителем. Оно не всплывает; его поведение по умолчанию (собственно изменение значения в поле ввода) может быть отменено.

```
<SCRIPT>
  function name1Change() {
    var name1Obj = document.getElementById("name1");
    var submObj = document.getElementById("subm");
    if (name1Obj.value != "")
      submObj.disabled = false
    else
      submObj.disabled = true;
  }
</SCRIPT>
. . .
<FORM . . .>
  <INPUT TYPE="text" ID="name1" NAME="name1">
  <INPUT TYPE="submit" ID="subm" NAME="subm">
</FORM>
```

Приведенный ранее обработчик события `onChange` делает доступной кнопку отправки данных только в том случае, если в поле ввода `name1` введено какое-либо значение.

Событие `onSelect` возникает при выделении посетителем значения в поле ввода. В Internet Explorer оно не всплывает, в Opera и Firefox всплывает; поведение по умолчанию (выделение значения в поле ввода) для него не может быть отменено.

> ***Внимание!***
>
> В документации по DOM написано, что поведение по умолчанию для события `onSelect` все-таки может быть отменено. Но, похоже, это не так.

Событие `onSelectStart` возникает, когда посетитель только начинает выделение значения в поле ввода. Оно всплывает; его поведение по умолчанию (выделение значения в поле ввода), в отличие от события `onSelect`, может быть отменено. Поддерживается оно только Internet Explorer.

```
<INPUT TYPE="text" ID="name1" NAME="name1" ONSELECTSTART="return false;">
```

Этот HTML-код создает поле ввода, содержимое которого посетитель выделить не сможет. Правда, непонятно, зачем это может понадобиться…

Область редактирования

Область редактирования представляется экземпляром объекта `HTMLTextAreaElement`. Сейчас мы его рассмотрим.

Прежде всего, этот объект поддерживает свойства `defaultValue`, `readOnly` и `value`, уже знакомые нам по полю ввода (см. ранее). Свойство `type` для области редактирования всегда возвращает строку "textarea".

```
var addressObj = document.getElementById("address");
var addr = addressObj.value;
addressObj.readOnly = true;
```

Этот сценарий присваивает текст, введенный в область редактирования `address`, переменной `addr` и делает эту область редактирования доступной только для чтения.

Свойство `cols` соответствует одноименному атрибуту тега `<TEXTAREA>` и задает или возвращает ширину области редактирования в символах в виде числа. Свойство `rows` также соответствует одноименному атрибуту тега `<TEXTAREA>`; оно задает или возвращает высоту области редактирования в строках в виде числа.

Свойство `wrap` соответствует одноименному атрибуту тега `<TEXTAREA>` и задает или возвращает обозначение режима выполнения областью редактирования перевода строк в виде строки.

Метод `select` также знаком нам по полю ввода. Он выделяет все содержимое области редактирования.

Метод `doScroll` выполняет прокрутку содержимого области редактирования, если, конечно, ее возможно выполнить. Он поддерживается только Internet Explorer.

```
doScroll(<обозначение способа прокрутки>)
```

Единственный параметр, который принимает этот метод, — строка, обозначающая способ выполнения прокрутки. Он может принимать следующие значения:

- "scrollbarDown" или "down" — щелчок кнопки "вниз" вертикальной полосы прокрутки;
- "scrollbarLeft" или "left" — щелчок кнопки "влево" горизонтальной полосы прокрутки;

- "scrollbarPageDown" или "pageDown" — щелчок на вертикальной полосе прокрутки ниже ее бегунка;
- "scrollbarPageLeft" или "pageLeft" — щелчок на горизонтальной полосе прокрутки левее ее бегунка;
- "scrollbarPageRight" или "pageRight" — щелчок на горизонтальной полосе прокрутки правее ее бегунка;
- "scrollbarPageUp" или "pageUp" — щелчок на вертикальной полосе прокрутки выше ее бегунка;
- "scrollbarRight" или "right" — щелчок кнопки "вправо" горизонтальной полосы прокрутки;
- "scrollbarUp" или "up" — щелчок кнопки "вверх" вертикальной полосы прокрутки.

Этот метод не возвращает результата.

Объект `HTMLTextAreaElement` поддерживает события `onChange`, `onSelect` и `onSelectStart`, знакомые нам по полю ввода.

Событие `onScroll` возникает при прокрутке содержимого области редактирования, "вручную" или вызовом метода `doScroll`. Оно не всплывает, и поведение по умолчанию для него (собственно прокрутка) не может быть отменено. Поддерживается это событие только Internet Explorer.

Поле ввода пароля

Поле ввода пароля представляется экземпляром объекта `HTMLInputElement`. Этот объект поддерживает свойства `autocomplete`, `defaultValue`, `maxLength`, `readOnly`, `size` и `value`, метод `select` и событие `onSelectStart`, знакомые нам по обычному полю ввода.

Кнопка отправки данных

Кнопка отправки данных представляется экземпляром объекта `HTMLInputElement`. В случае данного элемента управления доступно знакомое нам свойство `value`, задающее или возвращающее надпись на кнопке в виде строки.

Метод `click` позволяет имитировать щелчок на кнопке. Он не принимает параметров и не возвращает значения.

Каких-то специфических событий кнопка отправки данных не поддерживает. Поэтому, если нужно отследить щелчок на ней, обрабатывают либо событие `onClick` самой кнопки (оно было рассмотрено в *главе 8*), либо событие `onSubmit` формы, в которой находится эта кнопка (это событие рассматривалось ранее).

Кнопка сброса формы

Кнопка сброса формы представляется экземпляром объекта `HTMLInputElement` и поддерживает те же самые свойства, методы и события, что и рассмотренная ранее кнопка отправки данных. Для отслеживания щелчка на этой кнопке обрабатывают либо ее событие `onClick` (было рассмотрено в *главе 8*), либо событие `onReset` формы, в которой находится эта кнопка (это событие рассматривалось ранее).

```
<SCRIPT>
  function frmReset() {
    var dateObj = document.getElementById("date");
    var d = new Date();
    dateObj.value = d.toString();
  }
</SCRIPT>
. . .
<FORM ONRESET="frmReset();" . . .>
  . . .
  <INPUT TYPE="text" ID="date" NAME="date">
  . . .
  <INPUT TYPE="reset" ID="rst" NAME="rst">
  . . .
</FORM>
```

Приведенный HTML-код создает форму с полем ввода `date` и кнопкой сброса `rst`. При нажатии кнопки сброса формы в поле ввода `date` заносится строковое значение текущей даты и времени. Видно, что для этого обрабатывается событие `onReset` формы.

Кнопка

Обычная кнопка также представляется экземпляром объекта `HTMLInputElement` и поддерживает те же самые свойства, методы и события, что и рассмотренные ранее кнопки отправки данных и сброса формы. Для отслеживания щелчка на ней обрабатывают событие `onClick` этой кнопки (было рассмотрено в *главе 8*).

```
<SCRIPT>
  function btnClick() {
    var dateObj = document.getElementById("date");
    var d = new Date();
    dateObj.value = d.toString();
```

```
  }
</SCRIPT>
. . .
<FORM . . .>
  . . .
  <INPUT TYPE="text" ID="date" NAME="date">
  <INPUT TYPE="button" ID="btn" NAME="btn" VALUE="Текущая дата"
  ONCLICK="btnClick();">
  . . .
</FORM>
```

Приведенный HTML-код создает форму с полем ввода `date` и кнопкой `btn`, при нажатии которой в данное поле ввода заносится строка с текущей датой и временем.

Флажок

Флажок представляется экземпляром объекта `HTMLInputElement`. Он поддерживает свойство `value`, задающее или возвращающее значение флажка, и метод `click`, имитирующий щелчок на флажке.

Свойство `checked` задает или возвращает текущее состояние флажка, то есть включен он или выключен. Значение `true` обозначает, что флажок включен, значение `false` — что флажок выключен.

```
var chkObj = document.getElementById("chk");
var f = chkObj.checked;
```

Этот сценарий помещает в переменную `f` состояние флажка `chk`.

Свойство `defaultChecked` задает или возвращает изначальное состояние флажка, заданное в его HTML-коде атрибутом `CHECKED`. Здесь значение `true` также обозначает, что флажок изначально включен, значение `false` — что флажок изначально выключен.

Свойство `indeterminate` задает или возвращает признак того, что флажок находится в так называемом нейтральном состоянии, когда он включен и закрашен серым ("ни да, ни нет"). Значение `true` обозначает, что флажок находится в нейтральном состоянии (включен и закрашен серым); значение `false` — в обычном (то есть включен или выключен и не закрашен серым; в таком состоянии флажок находится по умолчанию). Это свойство поддерживается только Internet Explorer.

> ***На заметку***
>
> Нейтральное состояние флажка является для него стандартным, но на практике используется крайне редко. Вдобавок, флажок, присутствующий на странице, можно установить в нейтральное состояние только программно, из сценария.

Каких-либо специфических событий флажок не поддерживает. Для отслеживания щелчка на нем обрабатывается событие `onClick`.

```
<SCRIPT>
  function updatesClick() {
    var updatesObj = document.getElementById("updates");
    var emailObj = document.getElementById("email");
    emailObj.disabled = !(updatesObj.checked);
  }
</SCRIPT>
. . .
<FORM . . .>
  . . .
  <INPUT TYPE="checkbox" ID="updates" NAME="updates"
  ONCLICK="updatesClick();">
  Я хочу подписаться на рассылку<BR>
  Мой почтовый адрес
  <INPUT TYPE="text" ID="email" NAME="email" SIZE="100" DISABLED>
  . . .
</FORM>
```

Приведенный HTML-код создает форму с флажком `updates`, предлагающим посетителю подписаться на рассылку обновлений этого сайта, и полем ввода `email`, куда посетитель должен ввести свой почтовый адрес. Причем это поле ввода доступно только в том случае, если флажок `updates` включен. Чтобы отследить включение и отключение флажка, мы обрабатываем его событие `onClick`. Отметим также, что изначально поле ввода `email` недоступно (в создающем его теге `<INPUT>` присутствует атрибут без значения `DISABLED`), так как флажок `updates` изначально выключен (в создающем его теге `<INPUT>` отсутствует атрибут без значения `CHECKED`).

Переключатель

Переключатель представляется экземпляром объекта `HTMLInputElement`. Он поддерживает свойства `checked`, `defaultChecked`, `value` и метод `click`, знакомые нам по флажку.

Поскольку мы задаем имя сразу для всех переключателей, входящих в группу, то для доступа к конкретному переключателю в данной группе мы не сможем использовать хорошо нам знакомый метод `getElementById`. В этом случае лучше воспользоваться методом `getElementsByName`, описанным в *главе 5*. Этот метод принимает в качестве единственного параметра имя в виде строки и возвращает массив, содержащий все элементы страницы,

чье имя, заданное атрибутом NAME, совпадает со значением параметра этого метода. Получив с помощью этого метода массив переключателей, мы используем числовой индекс, чтобы добраться до нужного элемента массива — то есть переключателя.

```
<INPUT TYPE="radio" ID="pl" NAME="pl" VALUE="JavaScript"> JavaScript<BR>
<INPUT TYPE="radio" ID="pl" NAME="pl" VALUE="VBScript"> VBScript<BR>
<INPUT TYPE="radio" ID="pl" NAME="pl" VALUE="Delphi"> CodeGear Delphi
. . .
<SCRIPT>
  var plObjs = document.getElementsByName("pl");
  plObjs[0].checked = false;
  plObjs[1].checked = true;
  plObjs[2].checked = false;
  plObjs[2].disabled = true;
</SCRIPT>
```

Этот сценарий включает второй по счету переключатель в группе pl (**VBScript**) и делает недоступным третий переключатель (**CodeGear Delphi**). Обратим внимание, что мы, включив второй переключатель, явно отключили первый и третий — это нужно сделать, чтобы избежать проблем в Internet Explorer.

> ***Внимание!***
>
> Если мы программно включим какой-то переключатель, присвоив его свойству checked значение true, то остальные переключатели этой группы мы должны обязательно отключить, для чего достаточно присвоить их свойствам checked значение false. Иначе возможны проблемы в Internet Explorer.

Переключатель, как и флажок с кнопками, не поддерживает каких-то "своих" событий. Для отслеживания щелчка на нем обрабатывается "универсальное" событие onClick.

```
<SCRIPT>
  function updatesClick() {
    var updatesObjs = document.getElementByName("updates");
    var emailObj = document.getElementById("email");
    emailObj.disabled = !(updatesObjs[0].checked);
  }
</SCRIPT>
. . .
<FORM . . .>
  . . .
```

```
  Вы хотите подписаться на рассылку?
  <INPUT TYPE="radio" ID="updates" NAME="updates" VALUE="yes"
  ONCLICK="updatesClick();"> Да
  <INPUT TYPE="radio" ID="updates" NAME="updates" VALUE="no" CHECKED
  ONCLICK="updatesClick();"> Нет<BR>
  Введите почтовый адрес
  <INPUT TYPE="text" ID="email" NAME="email" SIZE="100" DISABLED>
  . . .
</FORM>
```

Мы переделали приведенный ранее пример формы, предлагающей посетителю подписаться на рассылку сайта, но с использованием группы из двух переключателей вместо флажка — это группа `updates`. Здесь мы сделали изначально включенным второй переключатель (**Нет**, обозначающий отказ от подписки). А для отслеживания включения какого-либо из переключателей группы `updates` мы обрабатываем событие `onClick` в обоих переключателях этой группы.

Список

Список и раскрывающийся список представляются экземпляром объекта `HTMLSelectElement`. Свойства, методы и события этого объекта мы сейчас рассмотрим.

Прежде всего, нужно упомянуть свойство `type`. В случае списка с возможностью выбора только одного пункта оно всегда возвращает строку "select-one". Если же список позволяет выбрать сразу несколько пунктов, оно всегда возвращает строку "select-multiple".

Свойство `size` соответствует одноименному атрибуту тега `<SELECT>` и задает или возвращает размер списка в пунктах (позициях) в виде числа.

Свойство `multiple` соответствует одноименному атрибуту тега `<SELECT>` и указывает, поддерживает ли список возможность выбора сразу нескольких пунктов. Значение `true` указывает, что в списке можно выбрать несколько пунктов одновременно, а значение `false` — только один пункт.

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 прока от свойства `selectedIndex` больше в том случае, если список позволяет выбирать одновременно только один пункт. Хотя в любом случае его можно применять для проверки, выбран ли в списке хоть один пункт.

```
var selObj = document.getElementById("sel");
selObj.selectedIndex = 0;
```

Этот сценарий делает первый пункт списка `sel` выбранным.

Свойство `options` возвращает одноименную коллекцию, содержащую все пункты текущего списка (они, как мы выясним позднее, представляются экземплярами объекта `HTMLOptionElement`). Эта коллекция представляет собой экземпляр объекта `HTMLOptionsCollection`. Мы можем использовать числовые индексы, чтобы получить доступ к нужному элементу этой коллекции, то есть нужному пункту списка.

```
var selObj = document.getElementById("sel");
selObj.options[1].disabled = true;
```

Этот сценарий делает второй пункт списка `sel` недоступным для выбора.

Событие `onChange` возникает при выборе посетителем любого пункта в списке. При выборе пункта программно, из сценария, это событие не возникает.

Объект `HTMLSelectElement`, кроме всего прочего, поддерживает еще два метода, которые мы рассмотрим потом. А пока обратимся к пунктам списка.

Пункт списка представляется экземпляром объекта `HTMLOptionElement`. Он поддерживает несколько полезных для нас свойств, которые мы сейчас рассмотрим.

Свойство `selected` задает или возвращает состояние пункта — выбран ли он в данный момент или нет. Значение `true` обозначает, что пункт выбран, значение `false` — не выбран.

```
var selObj = document.getElementById("sel");
var f = selObj.options[0].selected;
```

Этот сценарий помещает в переменную `f` логическое значение, обозначающее состояние первого пункта списка `sel`.

Свойство `defaultSelected` задает или возвращает изначальное состояние пункта списка, заданное в HTML-коде атрибутом `SELECTED` тега `<OPTION>`. Значение `true` обозначает, что пункт изначально выбран, значение `false` — изначально не выбран.

Свойство `value` соответствует одноименному атрибуту тега `<OPTION>` и задает или возвращает значение пункта списка в виде строки.

```
var selObj = document.getElementById("sel");
var val = selObj.options[2].value;
```

Этот сценарий помещает в переменную `val` значение третьего пункта списка `sel`.

```
var selObj = document.getElementById("sel");
var selectedOpts = [];
for (var i = 0; i < selObj.options.length; i++) {
  if (selObj.options[i].selected)
    selectedOpts[selectedOpts.length] = selObj.options[i].value;
}
```

А этот сценарий просматривает все пункты списка `sel` и заносит в массив `selectedOpts` значения тех из них, что в данный момент выбраны. (Подразумевается, что список `sel` позволяет выбирать сразу несколько пунктов.) Обратим внимание, как мы задаем индекс вновь добавляемого элемента массива `selectedOpts` — как текущую длину массива. В самом деле, если массив не содержит элементов, его длина, возвращаемая свойством `length`, равна 0, и если мы подставим это значение в качестве индекса, то создадим первый элемент этого массива с индексом 0 и т. д.

Свойство `text` задает или возвращает текст пункта в виде строки. Мы можем использовать это свойство, чтобы быстро получить доступ к тексту пункта.

Свойство `index` возвращает номер пункта в списке в виде числа. Вряд ли оно будет для нас полезно.

Никаких особых методов и событий объект `HTMLOptionElement` не поддерживает.

А теперь вернемся к объекту `HTMLSelectElement` и рассмотрим два его метода, о которых упоминалось ранее. Эти методы позволят нам добавить и удалить заданный пункт списка.

> ***Внимание!***
>
> К сожалению, использовать хорошо нам знакомый метод `appendChild` для добавления пункта в список мы можем только в Opera и Firefox. Internet Explorer не поддерживает этот метод в объекте `HTMLSelectElement`, представляющем список. При этом метод `removeChild` поддерживается во всех случаях. Иначе как недоработкой это назвать нельзя.

Метод `add` добавляет пункт в список. К сожалению, его формат различен в Internet Explorer и Firefox; Opera поддерживает оба этих формата.

В Internet Explorer формат метода `add` таков:

```
add(<добавляемый элемент>
↳[,<номер элемента, перед которым он будет помещен>])
```

Первый параметр задает элемент, который должен быть добавлен в список, в виде экземпляра объекта `HTMLOptionElement`. Он создается с помощью рассмотренного нами в *главе 8* метода `createElement` объекта `HTMLDocument`. Второй — необязательный — параметр задает номер элемента списка, перед которым будет помещен вновь добавленный, в виде числа; если он опущен, элемент будет помещен в конец списка. Значения этот метод не возвращает.

```
var optObj = document.createElement("OPTION");
optObj.text = "Microsoft Visual C++";
optObj.value = "CPP";
var plObj = document.getElementById("pl");
plObj.add(optObj, 3);
```

Этот сценарий добавляет в список `pl` пункт, представляющий реализацию языка программирования C++ от Microsoft. Причем этот пункт будет помещен перед четвертым по счету из уже присутствующих в списке пунктов.

А вот формат метода `add` в Firefox:

```
add(<добавляемый элемент>, <элемент, перед которым он будет помещен>)
```

Первый параметр также задает элемент, который должен быть добавлен в список, в виде экземпляра объекта `HTMLOptionElement`. А второй параметр задает элемент списка, перед которым будет помещен вновь добавленный, также в виде экземпляра объекта `HTMLOptionElement`; если он равен `null`, элемент будет помещен в конец списка. Метод `add` и в этом случае не возвращает значения.

```
var optObj = document.createElement("OPTION");
optObj.text = "Microsoft Visual C++";
optObj.value = "CPP";
var plObj = document.getElementById("pl");
plObj.add(optObj, plObj.options[3]);
```

Этот сценарий добавляет в список `pl` пункт, представляющий реализацию языка программирования C++ от Microsoft. Этот пункт также будет помещен перед четвертым по счету из уже присутствующих в списке пунктов.

Повторим, что Opera поддерживает оба формата написания метода `add`.

Метод `remove` удаляет заданный пункт из списка. Его формат одинаков во всех Web-обозревателях.

```
remove(<номер удаляемого пункта>)
```

Единственный параметр этого метода задает номер удаляемого пункта в виде числа. Метод `remove` не возвращает значения.

```
var plObj = document.getElementById("pl");
plObj.remove(2);
```

Этот сценарий удаляет из списка `pl` третий пункт, представляющий Microsoft Visual Basic.

Вот и все о списках и их пунктах.

Поле ввода имени файла

Поле ввода имени файла представляется экземпляром объекта `HTMLInputElement`. Этот объект поддерживает свойства `size` и `value`, метод `select` и событие `onSelectStart`, знакомые нам по полю ввода пароля.

Графическая кнопка отправки данных

Графическая кнопка отправки данных представляется экземпляром объекта `HTMLInputElement`. Она поддерживает свойства `alt`, `complete`, `lowsrc`, `readyState` и `src` и метод `click`, знакомые нам по обычной кнопке отправки данных и графическому изображению (см. *главу 9*).

Каких-то специфических событий графическая кнопка отправки данных не поддерживает. Поэтому, если нужно отследить щелчок на ней, обрабатывают либо событие `onClick` самой кнопки (оно было рассмотрено в *главе 8*), либо событие `onSubmit` формы, в которой находится эта кнопка (это событие рассматривалось ранее).

Скрытое поле

Скрытое поле представляется экземпляром объекта `HTMLInputElement`. Она поддерживает свойство `value` — вероятно, единственное полезное для нас. Никаких особых методов и событий оно не поддерживает.

Надпись

Надпись представляется экземпляром объекта `HTMLLabelElement`. Единственное полезное для нас свойство этого объекта — `htmlFor`, соответствующее атрибуту `FOR` тега `<LABEL>` и задающее или возвращающее имя элемента управления, к которому привязана надпись, в виде строки. Отметим, что данное свойство имеет имя "htmlFor", так как "for" совпадает с именем ключевого слова языка JavaScript (см. *главу 4*), что не допускается.

Группа

Группа представляется экземпляром объекта `HTMLFieldSetElement`. Никаких особых свойств, методов и событий он не поддерживает.

Заголовок группы представляется экземпляром объекта `HTMLLegendElement`. Никаких особых свойств, методов и событий он также не поддерживает.

Примеры Web-форм, управляемых сценариями

Во время рассмотрения объектов, представляющих саму форму и различные элементы управления, их свойств, методов и событий мы написали довольно много примеров, поясняющих их использование. Напоследок осталось привести два "больших" примера страниц с формами, которые управляются программно, из сценариев.

Web-форма со списком, заполняемым программно

Первый пример, который мы рассмотрим, — это форма со списком, чьи пункты генерируются программно в зависимости от того, какой переключатель включил посетитель. Довольно часто в формах приходится предоставлять посетителю разные списки в зависимости от выбранных им ранее условий, так что наш пример будет, так сказать, приближенным к жизни.

Наш список будет содержать языки программирования, принадлежащие к двум группам: используемые для написания сценариев и не используемые для этого. Сама группа выбирается путем включения соответствующего переключателя.

HTML-код этой страницы приведен далее.

```
<HTML>
  <HEAD>
    <TITLE>Список, заполняемый программно</TITLE>
    <SCRIPT>
      scs = [];
      scs[0] = new Array("JavaScript", "JavaScript");
      scs[1] = new Array("VBScript", "VBScript");
      noscs = [];
      noscs[0] = new Array("Delphi", "CodeGear Delphi");
      noscs[1] = new Array("VB", "Microsoft Visual Basic");
      noscs[2] = new Array("CPP", "Microsoft Visual C++");
      noscs[3] = new Array("CS", "Microsoft Visual C#");

      //Здесь должен помещаться код, объявляющий функции getBrowser и
      //getBrowserStrict. Этот код приведен в главе 7

      function fillSelect(pOptions) {
        var plObj = document.getElementById("pl");
        var optsObj = plObj.options;
        while (optsObj.length > 0) plObj.remove(0);
```

```
        for (var i = 0; i < pOptions.length; i++) {
          var optionObj = document.createElement("OPTION");
          optionObj.value = pOptions[i][0];
          optionObj.text = pOptions[i][1];
          if (getBrowserStrict() == "FF" )
            plObj.add(optionObj, null)
          else
            plObj.add(optionObj);
        }
      }
    </SCRIPT>
  </HEAD>
  <BODY>
    <H1>Языки программирования</H1>
    <FORM ACTION="#">
      <P><INPUT TYPE="radio" ID="group" NAME="group" VALUE="sc"
      ONCLICK="fillSelect(scs);">
      Используемые для написания Web-сценариев</P>
      <P><INPUT TYPE="radio" ID="group" NAME="group" VALUE="nosc"
      ONCLICK="fillSelect(noscs);">
      Не используемые для написания Web-сценариев</P>
      <P><SELECT ID="pl" NAME="pl" SIZE="4" MULTIPLE></SELECT></P>
    </FORM>
    <SCRIPT>
      var groupObjs = document.getElementsByName("group");
      groupObjs[0].checked = true;
      groupObjs[1].checked = false;
      fillSelect(scs);
    </SCRIPT>
  </BODY>
</HTML>
```

Сразу отметим, что приведенный код использует объявленные в *главе 7* функции `getBrowser` и `getBrowserStrict`. Поэтому их объявления нужно вставить в место, где присутствуют предписывающие это сделать комментарии.

Для хранения значений и текста пунктов, представляющих языки программирования, мы используем массивы `scs` (языки, используемые для написания сценариев) и `noscs` (не используемые для этого). Каждый элемент такого массива представляет собой массив, содержащий два элемента: значение пункта и его текст.

Для заполнения списка пунктами мы написали универсальную функцию `fillSelect`. Эта функция принимает в качестве единственного параметра массив, содержащий сведения о пунктах, — `scs` или `noscs`. В ее теле сначала выполняется очистка списка, потом для каждого элемента переданного ей массива создается пункт, наполняется данными из этого элемента массива и добавляется к списку. Этот код не очень сложен, так что вы можете разобраться с ним самостоятельно.

Отметим только выражение, выполняющее очистку списка.

```
while (optsObj.length > 0) plObj.remove(0);
```

Оно удаляет первый пункт списка, пока их количество не станет равным 0. Пожалуй, это самый простой способ очистить список.

Присутствующая на странице форма содержит группу из двух переключателей, выбирающих, какие языки программирования должны отображаться в списке. Для отслеживания включения этих переключателей мы обрабатываем их события `onClick`, в качестве обработчиков используя вызовы рассмотренной ранее функции `fillSelect` и передавая ей в качестве параметра нужный массив — `scs` или `noscs`.

Поскольку форма не будет никуда отправлять введенные в нее данные, мы не стали задавать интернет-адрес серверной программы. В противном случае его следует задать.

После того как форма будет загружена и выведена на страницу, мы принудительно включаем первый переключатель группы и так же принудительно отключаем второй, чтобы избежать проблем в Internet Explorer. И обязательно вызываем функцию `fillSelect`, передав ей в качестве параметра массив `scs`, — это нужно, чтобы изначально заполнить список языками программирования, используемыми при написании сценариев, соответственно включенному первому переключателю.

Web-форма, проверяющая правильность введенных в нее данных

Конечно, программно заполняемые списки тоже полезно уметь создавать. Но куда чаще приходится проверять, правильные ли данные ввел посетитель в форму. Давайте выясним, как это можно сделать.

Давайте возьмем созданную нами ранее страницу 12.1.htm, на которой посетитель вводил анкетные данные, и изменим ее так, чтобы она сама проверяла правильность этих данных. Пусть они должны удовлетворять таким условиям:

- имя, фамилия и почтовый адрес обязательно должны быть введены;

- в поле ввода почтового адреса должен быть введен именно почтовый адрес;
- в списке языков программирования должен быть выбран хотя бы один пункт.

Для проверки, действительно ли почтовый адрес был введен в соответствующее поле ввода, мы используем упрощенную проверку — будем выяснять, присутствует ли во введенных данных символ `@`. Если же нужно реализовать более строгую проверку, следует использовать регулярные выражения, описанные в *главе 11*.

HTML-код исправленной страницы 12.1.htm приведен далее.

```
<HTML>
  <HEAD>
    <TITLE>Анкета</TITLE>
    <SCRIPT>
      function frmSubmit() {
        var name1Obj = document.getElementById("name1");
        var name2Obj = document.getElementById("name2");
        var emailObj = document.getElementById("email");
        var plObj = document.getElementById("pl");
        if (name1Obj.value == "") {
          window.alert("Вы не ввели свое имя!");
          name1Obj.focus();
        } else {
          if (name2Obj.value == "") {
            window.alert("Вы не ввели свою фамилию!");
            name2Obj.focus();
          } else {
            if (emailObj.value == "") {
              window.alert("Вы не ввели свой адрес электронной почты!");
              emailObj.focus();
            } else {
              if (emailObj.value.indexOf("@") == -1) {
                window.alert("Адрес электронной почты введен
                ↳неправильно!");
                emailObj.focus();
              } else {
                if (plObj.selectedIndex == -1) {
                  window.alert("Вы не выбрали ни одного языка
```

```
                ↳программирования!");
                plObj.focus();
              } else {
                var frmObj = document.getElementById("frm");
                frmObj.submit();
              }
            }
          }
        }
      }
    </SCRIPT>
  </HEAD>
  <BODY>
    <FORM ID="frm" NAME="frm" ACTION="12.2.htm" METHOD="get"
    ONSUBMIT="frmSubmit(); return false;">
      Имя <INPUT TYPE="text" ID="name1" NAME="name1" MAXLENGTH="20"><BR>
      Фамилия <INPUT TYPE="text" ID="name2" NAME="name2"
      MAXLENGTH="20"><BR>
      E-mail <INPUT TYPE="text" ID="email" NAME="email" SIZE="50"><BR>
      Пол: <INPUT TYPE="radio" ID="gender" NAME="gender" VALUE="m"
      CHECKED> муж.,
      <INPUT TYPE="radio" ID="gender" NAME="gender" VALUE="f"> жен.<BR>
      Любимый язык программирования<BR>
      <SELECT ID="pl" NAME="pl" SIZE="4">
        <OPTION>JavaScript</OPTION>
        <OPTION>VBScript</OPTION>
        <OPTION VALUE="VB">Visual Basic</OPTION>
        <OPTION VALUE="Delphi">CodeGear Delphi</OPTION>
      </SELECT><BR>
      <INPUT TYPE="checkbox" ID="profi" NAME="profi">
      Я - профессиональный программист<BR>
      <INPUT TYPE="submit" ID="sub" NAME="sub" VALUE="Ввод">
    </FORM>
  </BODY>
</HTML>
```

Здесь мы последовательно проверяем, введены ли в соответствующие поля ввода имя, фамилия и почтовый адрес, действительно ли введен почтовый адрес и выбран ли в списке хоть один язык программирования. Если это так,

мы выполняем отправку данных; в противном случае выводим посетителю окно-сообщение с предупреждающим текстом и делаем элемент управления, в котором были введены некорректные данные, активным.

Заметим, что мы отменяем поведение события `onSubmit` формы по умолчанию (то есть отправку данных), поместив в обработчик этого события, помимо вызова функции, выполняющей все перечисленные ранее проверки, выражение `return false`. Это нужно, чтобы форма случайно не выполнила отправку некорректных данных. Если же введенные в форму данные прошли все проверки, мы, так или иначе, выполним отправку данных программно (вызовом метода `submit` формы).

На этом все о формах и элементах управления.

Что дальше?

В этой главе мы познакомились с Web-формами и элементами управления, с помощью которых реализуется ввод данных. Мы также узнали о том, как управлять Web-формами и элементами управления программно, из сценариев. Все-таки тема книги — Web-программирование…

На этом *часть II*, посвященная практическому использованию JavaScript, закончилась. Но ставить точку еще рано. Мы еще не познакомились со специфическими возможностями, предлагаемыми отдельными программами Web-обозревателей. И в *части III* мы начнем изучать этот предмет.

Часть III

Использование специфических возможностей Internet Explorer и Firefox

Глава 13

Взаимодействие с посетителем (Internet Explorer и Firefox)

В *части II* мы рассматривали, в основном, общие функции, предлагаемые Web-программистам всеми Web-обозревателями. Как правило, именно их и используют для написания сценариев, которые должны работать всегда и везде.

В *части III* мы будем рассматривать специфические функции, предлагаемые отдельными Web-обозревателями. Их используют, в основном, в корпоративных решениях, рассчитанных на работу в одном конкретном Web-обозревателе. В обычных Web-страницах, публикуемых в Интернете и рассчитанных на самый широкий круг посетителей, их используют редко. В самом деле, в корпорациях много проще контролировать программное обеспечение и заставить пользователей решений работать именно в том Web-обозревателе, на который эти решения рассчитаны. А попробуйте заставить обычного посетителя обычного сайта открыть его, скажем, в Firefox!

В части поддержки специфических функций отличились Internet Explorer и Firefox. Причем Internet Explorer здесь рекордсмен — он поддерживает столько всего, что просто диву даешься. Firefox в этом смысле заметно беднее. А Opera не поддерживает ничего сверх стандартов HTML и DOM, поэтому хуже подходит для разработки корпоративных решений.

В этой главе мы научимся использовать самые простые из специфических возможностей Internet Explorer и Firefox. Это работа с произвольными фрагментами текста, текстом, выделенным посетителем, Буфером обмена Windows, реализация "настоящего" drag'n'drop, использование диалоговых HTML-окон и создание HTML-приложений. Кто знает, может, это нам пригодится…

Работа с произвольными фрагментами текста

И начнем мы изучение этих возможностей с работы с произвольным фрагментом текста, находящимся на странице. Это может быть текст, являющийся содержимым определенного тега, или произвольный текст, объединяющий содержимое сразу нескольких тегов. Мы можем получить этот текст, изменить его и выполнить некоторые другие действия.

Работа с фрагментом текста в Internet Explorer

Фрагмент текста, являющегося содержимым некоторого тега, в Internet Explorer представляется экземпляром объекта `TextRange`. Он поддерживает ряд свойств и методов, с помощью которых выполняются манипуляции над представляемым им фрагментом текста.

Манипулировать с помощью объекта `TextRange` в Internet Explorer мы можем только содержимым секции тела страницы, поля ввода, области редактирования, надписи любой кнопки и значения скрытого поля. Но с помощью особого метода `moveToElementText` объекта `TextRange` (будет рассмотрен далее) мы можем поместить в его экземпляр текстовое содержимое любого тега. Кроме того, в виде экземпляра объекта `TextRange` может быть представлен текст, выделенный на странице посетителем; как это сделать, будет рассмотрено далее.

Экземпляр объекта `TextRange` создается с помощью метода `createTextRange` объекта, представляющего нужный нам элемент страницы. Этот метод не принимает параметров и возвращает экземпляр объекта `TextRange`, представляющий текст или значение данного элемента страницы.

```
var bodyTRObj = document.body.createTextRange();
```

Это выражение поместит в переменную `bodyTRObj` экземпляр объекта `TextRange`, представляющего весь текст, находящийся на странице (в ее секции тела).

```
var btnObj = document.getElementById("btn");
var btnTRObj = btnObj.createTextRange();
```

А этот сценарий поместит в переменную `btnTRObj` экземпляр объекта `TextRange`, представляющего надпись кнопки `btn`.

Все свойства объекта `TextRange` перечислены в табл. 13.1.

Таблица 13.1. *Свойства объекта* `TextRange`

Свойство	Описание
`boundingHeight`	Возвращает высоту воображаемого прямоугольника, охватывающего текст-содержимое текущего экземпляра объекта `TextRange`, в пикселах
`boundingLeft`	Возвращает расстояние по горизонтали между левой границей воображаемого прямоугольника, охватывающего текст-содержимое текущего экземпляра объекта `TextRange`, и левой границей элемента страницы, который содержит этот текст, в пикселах
`boundingTop`	Возвращает расстояние по вертикали между верхней границей воображаемого прямоугольника, охватывающего текст-содержимое текущего экземпляра объекта `TextRange`, и верхней границей элемента страницы, который содержит этот текст, в пикселах
`boundingWidth`	Возвращает ширину воображаемого прямоугольника, охватывающего текст-содержимое текущего экземпляра объекта `TextRange`, в пикселах
`htmlText`	Возвращает HTML-код, представляющий содержимое экземпляра объекта `TextRange`, в виде строки
`offsetLeft`	Возвращает горизонтальную координату верхнего левого угла воображаемого прямоугольника, охватывающего текст-содержимое текущего экземпляра объекта `TextRange`, относительно родителя в пикселах
`offsetTop`	Возвращает вертикальную координату верхнего левого угла воображаемого прямоугольника, охватывающего текст-содержимое текущего экземпляра объекта `TextRange`, относительно родителя в пикселах
`text`	Задает или возвращает текст, представляющий содержимое экземпляра объекта `TextRange`, в виде строки

Набор методов, поддерживаемых объектом `TextRange`, значительно больше. Сейчас мы их рассмотрим.

Метод `collapse` сжимает текущий текстовый фрагмент (то есть представляющий его экземпляр объекта `TextRange`) в точку и перемещает его в начало или конец текста, который был ранее его содержимым.

```
collapse([true|false])
```

Этот метод принимает один необязательный параметр. Если он равен `true` или вообще не указан, сжатый фрагмент перемещается в начало текстового

фрагмента, а если он равен `false`, сжатый фрагмент будет перемещен в его конец. Результата этот метод не возвращает.

```
btnTRObj.collapse();
btnTRObj.text = "Кнопка: ";
```

Этот сценарий добавляет к надписи кнопки `btn` текст "Кнопка:". Для этого он сжимает текстовый фрагмент, представляющий ее надпись, перемещает сжатый фрагмент в начало надписи и присваивает ему данный текст. Этот текст и станет содержимым текстового фрагмента.

Метод `compareEndPoints` сравнивает граничные точки (начало и конец) текущего текстового фрагмента с граничными точками другого.

```
compareEndPoints(<обозначение сравниваемых граничных точек>,
↳<второй текстовый фрагмент>)
```

Первый параметр этого метода указывает, какие именно граничные точки этих двух текстовых фрагментов будут сравниваться. Его значение может быть одной из перечисленных далее строк:

- "StartToStart" — начальная точка текущего текстового фрагмента сравнивается с начальной точкой второго;
- "StartToEnd" — начальная точка текущего текстового фрагмента сравнивается с конечной точкой второго;
- "EndToStart" — конечная точка текущего текстового фрагмента сравнивается с начальной точкой второго;
- "EndToEnd" — конечная точка текущего текстового фрагмента сравнивается с конечной точкой второго.

Второй параметр указывает текстовый фрагмент (экземпляр объекта `TextRange`, но давайте уж переходить на термин "текстовый фрагмент"), граничная точка которого будет сравниваться с граничной точкой текущего фрагмента.

Метод `compareEndPoints` возвращает одно из следующих числовых значений:

- –1, если указанная граничная точка текущего текстового фрагмента находится ближе к началу страницы, чем указанная граничная точка второго фрагмента;
- 0, если указанные граничные точки обоих текстовых фрагментов совпадают;
- 1, если указанная граничная точка текущего текстового фрагмента находится дальше к концу страницы, чем указанная граничная точка второго фрагмента.

Метод `duplicate` возвращает вновь созданный текстовый фрагмент, являющийся точной копией текущего. Параметров он не принимает.

Иногда может быть так, что текстовый фрагмент содержит в себе часть слова или предложения. Метод `expand` расширяет его таким образом, чтобы он включил все это слово или предложение.

```
expand(<обозначение части текста>)
```

Единственный параметр этого метода задает обозначение части текста, которую нужно включить во фрагмент полностью. Его значение может быть одной из перечисленных далее строк:

- "character" — символ (хм, нужно сильно постараться, чтобы поместить во фрагмент часть символа…);
- "word" — слово (то есть последовательность символов, ограниченная пробелами, символами табуляции, перевода строки и пр.);
- "sentence" — предложение (то есть последовательность символов, ограниченная знаками пунктуации);
- "textedit" — весь текст данного элемента страницы.

Метод `expand` возвращает `true`, если текстовый фрагмент был удачно расширен, и `false` в противном случае.

Метод `findText` позволяет выполнить поиск заданного текста внутри или вне текущего текстового фрагмента.

```
findText(<подстрока поиска>[, <диапазон поиска>[, <флаги>]])
```

Первый параметр задает подстроку поиска, то есть искомый текст, в виде строки.

Второй параметр задает диапазон поиска в виде количества символов, отсчитанных от начальной точки текстового фрагмента. Понятно, что задается он в виде числа. При этом:

- положительное число задает количество символов, среди которых будет выполняться поиск, отсчитанных от начальной точки текстового фрагмента к концу страницы. При этом поиск также будет продолжаться до конца страницы;
- 0 задает поиск только внутри текущего текстового фрагмента;
- отрицательное число задает количество символов, среди которых будет выполняться поиск, отсчитанных от начальной точки текстового фрагмента к началу страницы. При этом поиск будет вестись от начальной точки фрагмента до начала страницы, то есть в обратную сторону.

Если второй параметр не указан, поиск будет вестись внутри текущего текстового фрагмента, если он содержит какой-то текст, или до конца страницы, если он сжат в точку (например, методом `collapse`).

Третий параметр задает флаги — дополнительные параметры поиска — и является суммой следующих чисел:

- 1 — поиск будет вестись в обратную сторону, от конца страницы к ее началу;
- 2 — поиск только целых слов;
- 4 — поиск с учетом регистра символов.

Отсутствие какого-то из этих чисел в значении флага означает, что соответствующий параметр не будет иметь силы при поиске.

Если третий параметр не указан или равен 0, будет выполняться поиск любых последовательностей символов, совпадающих с заданной подстрокой поиска, от начала страницы к ее концу и без учета регистра символов.

Если поиск был успешным, то есть совпадающая с подстрокой поиска последовательность символов была найдена, текущий текстовый фрагмент включит ее, а метод `findText` вернет `true`. Если же поиск был неуспешным, текущий текстовый фрагмент не изменится, а метод `findText` возвращает `false`.

```
var f = bodyTRObj.findText("JavaScript");
```

Это выражение выполняет поиск строки "JavaScript" в секции тела страницы (содержащийся в ней текст представляет текстовый фрагмент `bodyTRObj`) и помещает в переменную `f` значение `true`, если он был успешен, и `false` в противном случае.

Метод `getBookmark` возвращает так называемую *закладку* — уникальную строку, позволяющую идентифицировать местоположение текущего текстового фрагмента. Он не принимает параметров.

```
var bodyTRBM = bodyTRObj.getBookmark();
```

Это выражение поместит в переменную `bodyTRBM` закладку, соответствующую текущему местоположению фрагмента `bodyTRObj`. Впоследствии с помощью этой закладки и метода `moveToBookmark` (будет рассмотрен далее) мы сможем восстановить местоположение этого фрагмента.

Метод `inRange` позволяет выяснить, содержится ли заданный текстовый фрагмент внутри текущего или совпадает с ним.

```
inRange(<второй текстовый фрагмент>)
```

Единственным параметром передается текстовый фрагмент (экземпляр объекта `TextRange`), который будет проверяться на предмет нахождения внутри

текущего. Метод `inRange` возвращает `true`, если этот фрагмент находится внутри текущего или совпадает с ним, и `false` в противном случае.

Метод `isEqual` позволяет выяснить, совпадает ли заданный текстовый фрагмент с текущим.

```
isEqual(<второй текстовый фрагмент>)
```

Принимаемый параметр и возвращаемое значение у него такое же, как у рассмотренного ранее метода `inRange`.

Метод `move` сжимает текущий текстовый фрагмент в точку и перемещает его на заданное количество частей текста: символов, слов или предложений.

```
move(<обозначение части текста>[, <количество частей текста>])
```

Первый параметр этого метода задает обозначение части текста, на заданное количество которых нужно переместить сжатый фрагмент. Его значение может быть одной из перечисленных далее строк:

- "character" — символ;
- "word" — слово;
- "sentence" — предложение;
- "textedit" — перемещает в начало или конец изначального текстового фрагмента.

Второй — необязательный — параметр задает количество указанных частей текста, на которое нужно переместить сжатый фрагмент, в виде числа. Это количество может быть как положительным, так и отрицательным; в первом случае перемещение выполняется по направлению к концу страницы, во втором — к ее началу. Если первый параметр этого метода имеет значение "textedit", то значение второго параметра 1 выполняет перемещение к концу изначального текстового фрагмента, а значение –1 — к его началу. Если второй параметр не указан, выполняется перемещение на одну часть текста к концу страницы.

Метод `move` возвращает количество частей текста, на которое был реально перемещен сжатый фрагмент.

```
bodyTRObj.move("textedit", -1);
```

Это выражение сжимает текстовый фрагмент `bodyTRObj` и перемещает его в начало изначального фрагмента (в самое начало страницы).

Методы `moveEnd` и `moveStart` перемещают, соответственно, конечную и начальную точку текущего текстового фрагмента. Их формат вызова, принимаемые параметры и возвращаемые значения такие же, как у метода `move`.

```
moveEnd|moveStart(<обозначение части текста>
↳[, <количество частей текста>])
bodyTRObj.moveEnd("word", 4);
```

Это выражение перемещает конечную точку текстового фрагмента `bodyTRObj` на четыре слова к концу страницы.

Метод `moveToBookmark` восстанавливает местоположение текстового фрагмента, соответствующее указанной закладке. Закладку можно получить вызовом метода `getBookmark` (был рассмотрен ранее).

```
moveToBookmark(<закладка>)
```

Единственный параметр этого метода задает саму закладку. Он возвращает `true` при успешном восстановлении местоположения и `false` в противном случае.

```
bodyTRObj.moveToBookmark(bodyTRBM);
```

Это выражение восстанавливает местоположение текстового фрагмента `bodyTRObj`, соответствующее полученной ранее закладке `bodyTRBM`.

Метод `moveToElementText` помещает в текущий текстовый фрагмент текст, являющийся содержимым заданного элемента страницы.

```
moveToElementText(<элемент страницы>)
```

Единственный параметр этого метода задает элемент страницы, текст которого нужно поместить в текущий текстовый фрагмент, в виде экземпляра объекта `HTMLElement` или производного от него. Значения этот метод не возвращает.

```
var paraObj = document.getElementById("para");
var paraTRObj = document.createTextRange();
paraTRObj.moveToElementText(paraTRObj);
```

Этот сценарий помещает в переменную `paraTRObj` текст, находящийся в абзаце `para`.

Метод `moveToPoint` сжимает текущий текстовый фрагмент в точку и перемещает его в точку с заданными координатами.

```
moveToPoint(<горизонтальная координата>, <вертикальная координата>)
```

Параметры этого метода задают координаты нужной точки в виде числа. Эти координаты отсчитываются относительно окна Web-обозревателя в пикселах. Значения этот метод не возвращает.

Метод `parentElement` возвращает элемент страницы, содержащий текущий текстовый фрагмент, в виде экземпляра объекта `HTMLElement` или производного от него. Если такового элемента нет, возвращается `null`. Параметров этот метод не принимает.

Метод `pasteHTML` помещает в текущий текстовый фрагмент новое содержимое в виде заданного HTML-кода. Предыдущее содержимое при этом пропадает.

```
pasteHTML(<новое содержимое в виде HTML-кода>)
```

Единственный параметр этого метода задает новое HTML-содержимое текущего текстового фрагмента в виде строки. Значения этот метод не возвращает.

```
paraTRObj.pasteHTML("<STRONG>Привет!</STRONG>");
```

Это выражение помещает в абзац `para` строку "Привет!", выделенную полужирным шрифтом.

Уже знакомый нам по *главе 7* метод `scrollIntoView` выполняет прокрутку страницы так, чтобы содержимое текущего текстового фрагмента стало видимым.

```
scrollIntoView([true|false])
```

Этот метод может принимать в качестве необязательного параметра логическую величину. Значение `true` прокручивает содержимое окна так, чтобы текущий элемент страницы появится у верхнего его края; такой же эффект даст пропуск этого параметра. Значение `false` прокручивает окно так, что элемент страницы будет находиться у нижнего края окна. Значения этот метод не возвращает.

```
paraTRObj.scrollIntoView(false);
```

Это выражение выполняет прокрутку страницы так, чтобы абзац `para` стал видимым и находился у нижнего края окна Web-обозревателя.

А метод `select` мы рассмотрели в *главе 8*. Он выделяет содержимое текущего текстового фрагмента. Параметров этот метод не принимает и значения не возвращает.

```
if (bodyTRObj.findText("JavaScript"))
  bodyTRObj.select();
```

Это выражение выполняет поиск строки "JavaScript" в секции тела страницы (текстовый фрагмент `bodyTRObj`) и, в случае успешного поиска, выделяет найденную строку.

Метод `setEndPoint` совмещает заданную граничную точку (начало или конец) текущего текстового фрагмента с заданной граничной точкой другого.

```
setEndPoint(<обозначение граничных точек>, <второй текстовый фрагмент>)
```

Первый параметр этого метода указывает, какие именно граничные точки этих двух текстовых фрагментов будут совмещаться. Его значение может быть одной из перечисленных далее строк:

- "StartToStart" — начальная точка текущего текстового фрагмента совмещается с начальной точкой второго;
- "StartToEnd" — начальная точка текущего текстового фрагмента совмещается с конечной точкой второго;
- "EndToStart" — конечная точка текущего текстового фрагмента совмещается с начальной точкой второго;

- "EndToEnd" — конечная точка текущего текстового фрагмента совмещается с конечной точкой второго.

Второй параметр указывает текстовый фрагмент (экземпляр объекта `TextRange`), граничная точка которого будет совмещена с граничной точкой текущего фрагмента.

Метод `setEndPoint` не возвращает значения.

Особой пользы от объекта `TextRange` нет. Единственное, где он может пригодиться, — если мы собираемся выполнять сложную обработку текста на странице или работать с выделенным текстом. Но об этом позже.

Работа с фрагментом текста в Firefox

Фрагмент текста, являющегося содержимым некоторого тега, в Firefox представляется экземпляром объекта `Range`. Работа с ним принципиально отличается от работы с уже рассмотренным объектом `TextRange`, поддерживаемым Internet Explorer.

Для создания "пустого" (не содержащего никакого текста) экземпляра объекта `Range` используется метод `createRange` объекта `HTMLDocument`. Он не принимает параметров и возвращает вновь созданный экземпляр.

```
var someTRObj = document.createRange();
```

Не забываем, что полученный текстовый фрагмент пуст, то есть не содержит ничего. Нам самим придется задать его начальную и конечную точки.

```
someTRObj.selectNodeContents(document.body);
```

Этот сценарий задает в качестве начальной точки начало секции тела страницы, а в качестве конечной точки — ее конец. Метод `selectNodeContents` будет рассмотрен далее.

Свойства объекта `Range` перечислены в табл. 13.2.

Таблица 13.2. *Свойства объекта* `Range`

Свойство	Описание
`collapsed`	Возвращает `true`, если начальная и конечная точка текстового фрагмента совпадают (то есть фрагмент сжат в точку), и `false` в противном случае
`commonAncestorContainer`	Возвращает элемент страницы, являющийся родителем для элементов, возвращаемых свойствами `startContainer` и `endContainer` (см. далее), фактически — элемент, полностью содержащий данный текстовый фрагмент

Таблица 13.2 *(окончание)*

Свойство	Описание
`endContainer`	Возвращает элемент страницы, внутри которого находится конечная точка текущего текстового фрагмента
`endOffset`	Если элемент, возвращаемый свойством `endContainer`, является текстовым, возвращает количество символов между его началом и конечной точкой текущего текстового фрагмента. Если элемент, возвращаемый свойством `endContainer`, является тегом, возвращает количество дочерних элементов между его началом и конечной точкой текущего текстового фрагмента
`startContainer`	Возвращает элемент страницы, внутри которого находится начальная точка текущего текстового фрагмента
`startOffset`	Если элемент, возвращаемый свойством `startContainer`, является текстовым, возвращает количество символов между его началом и начальной точкой текущего текстового фрагмента. Если элемент, возвращаемый свойством `startContainer`, является тегом, возвращает количество дочерних элементов между его началом и начальной точкой текущего текстового фрагмента

Здесь не обойтись без пояснений, которые лучше сделать на примерах. Давайте возьмем абзац, содержащий текст, часть которого является содержимым текстового фрагмента; подчеркнутые символы указывают начальную и конечную точки этого фрагмента.

```
<P>JavaScript и AJAX в Web-дизайне</P>
```

Тогда перечисленные ранее свойства (кроме `collapsed` — с ним все ясно) вернут такие значения:

- `commonAncestorContainer` — экземпляр объекта `HTMLParagraphElement`, представляющий данный абзац;
- `endContainer` — экземпляр объекта `Text`, представляющий текстовое содержимое данного абзаца;
- `endOffset` — 16 (номер символа, на котором находится конечная точка текстового фрагмента, в текстовом содержимом абзаца; и не забываем, что нумерация символов начинается с нуля);

- `startContainer` — тот же экземпляр объекта `Text`, что и возвращается свойством `endContainer`;
- `startOffset` — 11 (номер символа, на котором находится начальная точка текстового фрагмента, в текстовом содержимом абзаца).

Немного усложним наш абзац.

```
<P>JavaScript и <STRONG>AJAX</STRONG> в Web-дизайне</P>
```

Видно, что начальная и конечная точки текстового фрагмента находятся теперь в разных тегах. И перечисленные ранее свойства вернут такие значения:

- `commonAncestorContainer` — экземпляр объекта `HTMLParagraphElement`, представляющий данный абзац;
- `endContainer` — экземпляр объекта `Text`, представляющий текстовое содержимое тега `<STRONG>`;
- `endOffset` — 3 (номер символа, на котором находится конечная точка текстового фрагмента, в текстовом содержимом тега `<STRONG>`);
- `startContainer` — экземпляр объекта `Text`, представляющий текстовое содержимое данного абзаца;
- `startOffset` — 11 (номер символа, на котором находится начальная точка текстового фрагмента, в первом текстовом элементе, представляющем содержимое абзаца).

Еще немного усложним пример.

```
<P>JavaScript и <STRONG>AJAX</STRONG> в Web-дизайне</P>
```

Здесь начальная и конечная точки текстового фрагмента находятся в разных текстовых элементах, представляющих текстовое содержимое абзаца. И перечисленные ранее свойства вернут такие значения:

- `commonAncestorContainer` — экземпляр объекта `HTMLParagraphElement`, представляющий данный абзац;
- `endContainer` — экземпляр объекта `Text`, представляющий содержимое второго тестового элемента в данном абзаце;
- `endOffset` — 1 (номер символа, на котором находится конечная точка текстового фрагмента, во втором текстовом элементе);
- `startContainer` — экземпляр объекта `Text`, представляющий содержимое первого текстового элемента данного абзаца;
- `startOffset` — 11 (номер символа, на котором находится начальная точка текстового фрагмента, в первом текстовом элементе, представляющем содержимое абзаца).

И еще один пример.

```
<P>JavaScript и <STRONG>AJAX</STRONG> в Web-дизайне</P>
```

Здесь начальная точка текстового фрагмента находится на теге `<STRONG>`, а конечная — во втором текстовом элементе данного абзаца. Значения свойств будут такими:

- `commonAncestorContainer` — экземпляр объекта `HTMLParagraphElement`, представляющий данный абзац;
- `endContainer` — экземпляр объекта `Text`, представляющий содержимое второго тестового элемента в данном абзаце;
- `endOffset` — 1 (номер символа, на котором находится конечная точка текстового фрагмента, во втором текстовом элементе, то есть начало данного фрагмента);
- `startContainer` — экземпляр объекта `HTMLElement`, представляющий тег `<STRONG>`;
- `startOffset` — 1 (номер тега `<STRONG>` в списке дочерних элементов абзаца; номер 0 будет иметь первый текстовый элемент).

Вот такие тонкости… Вероятно, именно поэтому средства Firefox для работы с фрагментами текста используются довольно редко.

Настала пора рассмотреть весьма многочисленные методы объекта `Range`. Сначала разберем методы, с помощью которых выполняется позиционирование граничных точек текстового фрагмента.

Метод `setStart` позиционирует начальную точку текстового фрагмента.

```
setStart(<элемент страницы>, <смещение>)
```

Первый параметр задает элемент страницы, который должен содержать начальную точку. Второй параметр задает смещение от начала этого элемента до собственно начальной точки. При этом:

- если задан текстовый элемент, то смещение задает количество символов от его начала до начальной точки текстового фрагмента;
- если задан тег, то смещение задает количество его дочерних тегов от начала данного тега до начальной точки текстового фрагмента.

Значения метод `setStart` не возвращает.

Метод `setEnd` позиционирует конечную точку текстового фрагмента.

```
setEnd(<элемент страницы>, <смещение>)
```

Первый параметр задает элемент страницы, который должен содержать конечную точку. Второй параметр задает смещение от начала этого элемента до собственно конечной точки. При этом:

- если задан текстовый элемент, то смещение задает количество символов от его начала до конечной точки текстового фрагмента;
- если задан тег, то смещение задает количество его дочерних тегов от начала данного тега до конечной точки текстового фрагмента.

Значения метод `setEnd` также не возвращает.

```
<P ID="para">JavaScript и AJAX в Web-дизайне</P>
. . .
<SCRIPT>
  var paraObj = document.getElementById("para");
  var textTRObj = document.createRange();
  textTRObj.setStart(paraObj.firstChild, 13);
  textTRObj.setEnd(paraObj.firstChild, 17);
</SCRIPT>
```

Этот сценарий устанавливает начальную точку текстового фрагмента `textTRObj` на 13-й символ текстового содержимого абзаца `para`, а конечную точку — на его 17-й символ. То есть он помещает в текстовый фрагмент `textTRObj` слово "AJAX".

Методы `setStartBefore` и `setStartAfter` устанавливают начальную точку текстового фрагмента относительно заданного элемента страницы. Первый метод устанавливает начальную точку в начале заданного элемента страницы, второй — в начале следующего в списке дочерних элементов того же родителя.

```
setStartBefore|setStartAfter(<элемент страницы>)
```

Единственный параметр этих методов задает сам элемент страницы. Значения оба этих метода не возвращают.

Методы `setEndBefore` и `setEndAfter` устанавливают конечную точку текстового фрагмента относительно заданного элемента страницы. Первый метод устанавливает конечную точку в начале заданного элемента страницы, второй — в начале следующего в списке дочерних элементов того же родителя.

```
setEndBefore|setEndAfter(<элемент страницы>)
```

Единственный параметр этих методов задает сам элемент страницы. Значения оба этих метода также не возвращают.

```
<P ID="para">JavaScript и AJAX в Web-дизайне</P>
. . .
```

```
<SCRIPT>
  var paraObj = document.getElementById("para");
  var textTRObj = document.createRange();
  textTRObj.setStartBefore(paraObj);
  textTRObj.setEndAfter(paraObj);
</SCRIPT>
```

Этот сценарий помещает в текстовый фрагмент `textTRObj` весь абзац `para` с его содержимым.

Методы `selectNode` и `selectNodeContents` помещают в текущий текстовый фрагмент заданный элемент страницы с его содержимым и только содержимое данного элемента страницы соответственно.

```
selectNode|selectNodeContents(<элемент страницы>)
```

Единственный параметр этих методов задает сам элемент страницы. Значения оба этих метода не возвращают.

```
<P ID="para">JavaScript и AJAX в Web-дизайне</P>
. . .
<SCRIPT>
  var paraObj = document.getElementById("para");
  var textTRObj = document.createRange();
  textTRObj.selectNodeContents(paraObj);
</SCRIPT>
```

Этот сценарий помещает в текстовый фрагмент `textTRObj` содержимое абзаца `para`.

Метод `collapse` сжимает текущий текстовый фрагмент в точку и перемещает его в начало или конец текста, который был ранее его содержимым.

```
collapse(true|false)
```

Этот метод принимает единственный параметр. Если он равен `true`, сжатый фрагмент перемещается в начало изначального текстового фрагмента, а если он равен `false`, сжатый фрагмент будет перемещен в его конец. Результата этот метод не возвращает.

Теперь рассмотрим методы объекта `Range`, позволяющие изменить содержимое текстового фрагмента.

Метод `cloneContents` возвращает экземпляр особого объекта `DocumentFragment`, представляющего фрагмент страницы. Этот фрагмент может быть помещен в другое место страницы с помощью любого из рассмотренных в *главе 8* методов DOM, например, метода `appendChild`. Метод `cloneContents` не принимает параметров.

Возвращаемый методом `cloneContents` фрагмент страницы (то есть экземпляр объекта `DocumentFragment`) будет содержать все атрибуты, имеющиеся в текущем текстовом фрагменте, в том числе и атрибуты `ID` и `NAME`, с их значениями. Если текущий текстовый фрагмент содержит открывающие теги, не имеющие закрывающей "пары", то в возвращенном фрагменте страницы соответствующие закрывающие теги будут присутствовать; то же касается и "непарных" закрывающих тегов. Однако обработчики событий, привязанные методом `addEventListener` (см. *главу 6*), в возвращенном фрагменте страницы присутствовать не будут.

Метод `deleteContents` удаляет из страницы фрагмент ее содержимого, соответствующий содержимому текущего текстового фрагмента; при этом также удаляется содержимое текущего текстового фрагмента. Параметров он не принимает и результата не возвращает.

```
<P ID="para">JavaScript и AJAX в Web-дизайне</P>
. . .
<SCRIPT>
  var paraObj = document.getElementById("para");
  var textTRObj = document.createRange();
  textTRObj.selectNode(paraObj);
  textTRObj.deleteContents();
</SCRIPT>
```

После выполнения этого сценария абзац `para` будет удален из страницы. Также будет удалено содержимое текстового фрагмента `textTRObj`, хотя сам этот фрагмент останется в памяти.

Метод `extractContents` также удаляет из страницы фрагмент ее содержимого, соответствующий содержимому текущего текстового фрагмента, и содержимое текущего фрагмента. Но при этом он возвращает экземпляр объекта `DocumentFragment`, представляющий удаленный фрагмент. Этот фрагмент может быть снова помещен на страницу с помощью любого из рассмотренных в *главе 8* методов DOM, например, метода `appendChild`. Параметров метод `extractContents` не принимает.

```
<P ID="para">JavaScript и AJAX в Web-дизайне</P>
. . .
<SCRIPT>
  var paraObj = document.getElementById("para");
  var textTRObj = document.createRange();
  textTRObj.selectNode(paraObj);
  var frObj = textTRObj.extractContents();
```

```
  document.body.appendChild(frObj);
</SCRIPT>
```

Этот сценарий сначала удаляет со страницы абзац `para`, а потом снова помещает его на страницу.

Метод `insertNode` вставляет в начальную точку текущего текстового фрагмента заданный элемент страницы. При этом вставленный элемент страницы сразу же появится на странице.

```
insertNode(<вставляемый элемент страницы>)
```

Единственный параметр указывает вставляемый элемент страницы. Значения этот метод не возвращает.

```
<P ID="para">JavaScript и AJAX в Web-дизайне</P>
. . .
<SCRIPT>
  var paraObj = document.getElementById("para");
  var textTRObj = document.createRange();
  textTRObj.selectNode(paraObj);
  var pObj = document.createElement("P");
  var textObj = document.createTextNode("Книги…");
  pObj.appendChild(textObj);
  textTRObj.insertNode(pObj);
</SCRIPT>
```

Этот сценарий помещает перед абзацем `para` другой абзац с текстом "Книги…".

Метод `surroundContents` помещает содержимое текущего текстового фрагмента в заданный элемент страницы — тег, который также станет частью содержимого этого фрагмента. Все изменения при этом будут отражены на странице.

```
surroundContants(<элемент страницы — тег>)
```

Единственный параметр указывает элемент страницы — тег, в который будет помещено содержимое текущего фрагмента. Значения этот метод не возвращает.

```
<P ID="para">JavaScript и AJAX в Web-дизайне</P>
. . .
<SCRIPT>
  var paraObj = document.getElementById("para");
  var textTRObj = document.createRange();
  textTRObj.selectNodeContents(paraObj);
```

```
  var emObj = document.createElement("EM");
  textTRObj.surroundContents(emObj);
</SCRIPT>
```

Этот сценарий поместит все текстовое содержимое абзаца `para` в тег `<EM>`.

Осталось рассмотреть несколько вспомогательных методов объекта `Range`.

Метод `cloneRange` возвращает новый текстовый фрагмент, являющийся точной копией текущего; этим он отличается от рассмотренного ранее метода `cloneContents`. Метод `cloneRange` не принимает параметров.

Метод `compareBoundaryPoints` сравнивает граничные точки (начало и конец) текущего текстового фрагмента с граничными точками другого.

```
compareBoundaryPoints(<обозначение сравниваемых граничных точек>,
↳<второй текстовый фрагмент>)
```

Первый параметр этого метода указывает, какие именно граничные точки этих двух текстовых фрагментов будут сравниваться. В качестве его можно использовать одно из перечисленных далее статических свойств объекта `Range`:

- `Range.START_TO_START` — начальная точка текущего текстового фрагмента сравнивается с начальной точкой второго фрагмента;
- `Range.START_TO_END` — начальная точка текущего текстового фрагмента сравнивается с конечной точкой второго фрагмента;
- `Range.END_TO_START` — конечная точка текущего текстового фрагмента сравнивается с начальной точкой второго фрагмента;
- `Range.END_TO_END` — конечная точка текущего текстового фрагмента сравнивается с конечной точкой второго фрагмента.

Второй параметр указывает текстовый фрагмент (экземпляр объекта `TextRange`), граничная точка которого будет сравниваться с граничной точкой текущего фрагмента.

Метод `compareBoundaryPoints` возвращает одно из следующих числовых значений:

- –1, если указанная граничная точка текущего текстового фрагмента находится ближе к началу страницы, чем указанная граничная точка второго фрагмента;
- 0, если указанные граничные точки обоих текстовых фрагментов совпадают;
- 1, если указанная граничная точка текущего текстового фрагмента находится дальше к концу страницы, чем указанная граничная точка второго фрагмента.

Метод `comparePoint` позволяет узнать, где находится указанная точка: перед текущим текстовым фрагментом, внутри или после него. Эта точка задается как смещение относительно элемента страницы.

`comparePoint(<элемент страницы>, <смещение>)`

Первый параметр задает элемент страницы, который содержит точку, чье местоположение относительно текущего фрагмента нужно выяснить. Второй параметр задает смещение от начала этого элемента до собственно данной точки. При этом:

- если задан текстовый элемент, то смещение задает количество символов от его начала до точки, чье местоположение относительно текущего фрагмента нужно выяснить;
- если задан тег, то смещение задает количество его потомков от начала тега до точки, чье местоположение относительно текущего фрагмента нужно выяснить.

Метод `comparePoint` возвращает одно из следующих числовых значений:

- –1, если указанная точка находится перед текущим фрагментом;
- 0, если указанная точка находится внутри текущего фрагмента;
- 1, если указанная точка находится после текущего фрагмента.

Статический метод `createContextualFragment` принимает в качестве единственного параметра строку, содержащую HTML-код, и возвращает экземпляр объекта `DocumentFragment`, представляющий созданный на основе этого кода фрагмент страницы. Содержимое текущего текстового фрагмента не изменяется и никак не используется для работы этого метода.

`createContextualFragment(<HTML-код>)`

Единственный параметр этого метода задает саму строку с HTML-кодом.

> ***Внимание!***
>
> Чтобы метод `createContextualFragment` успешно отработал, текущий текстовый фрагмент должен иметь хоть какое-то содержимое. В случае сжатого текстового фрагмента вызов этого метода вызывает ошибку.

```
<SCRIPT>
  var someTRObj = document.createRange();
  someTRObj.selectNode(document.body);
  var sCode = "<P>JavaScript и AJAX в Web-дизайне</P>";
  var divObj = someTRObj.createContextualFragment(sCode);
```

```
  document.body.appendChild(divObj);
</SCRIPT>
```

Этот сценарий помещает на страницу абзац с текстом "JavaScript и AJAX в Web-дизайне". Обратим внимание, что перед вызовом метода `createContextualFragment` мы помещаем в текстовый фрагмент `someTRObj` секцию тела страницы, так как в случае сжатого текстового фрагмента (а он и создается изначально сжатым) вызов этого метода вызовет ошибку.

Метод `detach`, насколько удалось выяснить автору, освобождает системные ресурсы, занимаемые текущим текстовым фрагментом. Хотя при этом сам фрагмент остается в памяти, но работать с ним уже будет невозможно.

Метод `isPointInRange` является "быстрой" разновидностью рассмотренного ранее метода `comparePoint`. Он только позволяет выяснить, находится ли заданная точка внутри текущего текстового фрагмента.

`isPointInRange(`*<элемент страницы>*`, `*<смещение>*`)`

Оба параметра этого метода аналогичны параметрам метода `comparePoint`. Он возвращает `true`, если заданная точка находится внутри текущего текстового фрагмента, и `false` в противном случае.

Метод `toString` возвращает текстовое содержимое текущего текстового фрагмента в виде строки; все имеющиеся теги HTML присутствовать в нем не будут. Этот метод не принимает параметров.

Объект `Range` Firefox несколько полезнее объекта `TextRange` Internet Explorer. Хотя его чаще всего используют для работы с выделенным текстом (см. далее).

На этом довольно долгий рассказ о работе с произвольными фрагментами текста закончен.

Работа с выделенным текстом

И Internet Explorer, и Firefox позволяют получить из сценария доступ к фрагменту текста страницы, выделенному посетителем. Это может быть как текст на самой странице (в абзаце, заголовке, списке и пр.), так и часть содержимого поля ввода или области редактирования. Давайте рассмотрим средства, предлагаемые разными Web-обозревателями для работы с выделенным текстом.

Работа с выделенным текстом в Internet Explorer

В Internet Explorer выделенный на странице текст представляется объектом `Selection`. Этот объект содержит несколько статичных свойств и методов, с помощью которых мы можем манипулировать выделенным текстом.

Получить доступ к объекту можно через свойство `selection` объекта `HTMLDocument`. Только нужно учесть, что это свойство вернет объект `Selection` только в том случае, если на странице что-то выделено; в противном случае оно вернет `null`.

```
var selObj = document.selection;
```

Это выражение помещает объект `Selection` или `null` в переменную `selObj`.

Объект `Selection` поддерживает свойство `type`. Оно возвращает строку "text", если на странице выделен именно текст, и "none", если выделено было что-то другое.

Что касается методов объекта `Selection`, то их три, полезных для нас. Давайте их рассмотрим.

Метод `clear` удаляет выделенный текст со страницы. Он не принимает параметров и не возвращает значения.

Метод `createRange` создает и возвращает экземпляр объекта `TextRange`, представляющий выделенный текст. Он не принимает параметров.

```
var selectedTRObj = document.selection.createRange();
scrollIntoView();
```

Этот сценарий выполняет прокрутку страницы так, чтобы выделенный текст находился у верхнего края окна Web-обозревателя.

Метод `empty` убирает выделение, не убирая сам выделенный текст. Он не принимает параметров и не возвращает значения.

Для примера давайте создадим страницу с абзацем. При выделении фрагмента содержимого этого абзаца на странице будет выводиться выделенный фрагмент. HTML-код этой страницы приведен далее.

```
<HTML>
  <HEAD>
    <TITLE>Работа с выделенным текстом</TITLE>
    <SCRIPT>
      function parMouseUp() {
        var selObj = document.selection;
        var outputObj = document.getElementById("output");
        if (selObj == null)
          outputObj.firstChild.nodeValue = " "
        else {
          var selTRObj = selObj.createRange();
          outputObj.firstChild.nodeValue = selTRObj.text;
        }
```

```
      }
    </SCRIPT>
  </HEAD>
  <BODY>
    <P ONMOUSEUP="parMouseUp();">Выделите любой фрагмент этой строки.</P>
    <P ID="output"> </P>
  </BODY>
</HTML>
```

Для определения момента окончания выделения текста в абзаце мы обрабатываем событие `onMouseUp`, возникающее при отпускании кнопки мыши. В самом деле, выделить текст на странице можно только мышью, при этом посетитель должен нажать кнопку мыши в той точке, где он хочет начать выделение, переместить мышь в точку, где выделение должно закончиться, и отпустить кнопку мыши. Вот как раз момент отпускания кнопки и следует отслеживать, чтобы выяснить, был ли текст выделен.

Дальше все просто. Мы определяем, был ли текст действительно выделен (ведь посетитель мог просто щелкнуть на абзаце), сравнивая значение свойства `selection` с `null`. Если текст был выделен (значение свойства `selection` не равно `null`), мы создаем на его основе текстовый фрагмент и извлекаем сам выделенный текст, обратившись к свойству `text` объекта `TextRange`. Этот текст мы помещаем в абзац `output`, традиционно используемый нами для вывода различной информации. Если текст не был выделен (значение свойства `selection` равно `null`), мы помещаем в этот же абзац пробел.

В полях ввода и областях редактирования отслеживать момент выделения текста еще проще. Они поддерживают событие `onSelect`, описанное в *главе 12*.

Работа с выделенным текстом в Firefox

В Firefox выделенный на странице текст также представляется объектом `Selection`, содержащим ряд свойств и методов. Но доступ к нему осуществляется уже методом `getSelection` объекта `Window`. Причем объект `Selection` возвращается всегда, независимо от того, был ли действительно выделен текст.

```
var selObj = window.getSelection();
```

Сам выделенный текст уже представляется как текстовый фрагмент (экземпляр объекта `Range`, описанный ранее). Используя свойства и методы этого объекта, мы можем получить доступ к самому выделенному тексту. А используя сценарии, мы можем добавить в выделение дополнительные текстовые фрагменты, созданные программно.

Свойства объекта `Selection` в Firefox перечислены в табл. 13.3.

Таблица 13.3. *Свойства объекта* `Selection` *в Firefox*

Свойство	Описание
`anchorNode`	Возвращает элемент страницы, в котором находится начальная точка выделения
`anchorOffset`	Возвращает номер символа, начиная от начала элемента страницы, возвращенного свойством `anchorNode`, на котором находится начальная точка выделения
`focusNode`	Возвращает элемент страницы, в котором находится конечная точка выделения
`focusOffset`	Возвращает номер символа, начиная от начала элемента страницы, возвращенного свойством `focusNode`, на котором находится конечная точка выделения
`isCollapsed`	Возвращает `true`, если выделение сжато в точку (то есть ее начальная и конечная точки совпадают), и `false` в противном случае
`rangeCount`	Возвращает количество созданных на основе выделения текстовых фрагментов. Если никакой текст не был выделен, возвращается 0; в случае выделения текста возвращается 1, так как выделенный текст представляется в виде единственного текстового фрагмента, созданного самим Firefox

Используя свойство `rangeCount`, мы можем выяснить, был ли выделен текст. Если значение, возвращенное этим свойством, равно 0, то текст выделен не был; в противном случае будет возвращено значение 1 или большее (если мы программно добавили в выделение дополнительные текстовые фрагменты).

Настала пора рассмотреть методы объекта `Selection`, поддерживаемого Firefox. Их довольно много.

Метод `addRange` добавляет в выделение новый текстовый фрагмент. Это может понадобиться в специальных случаях.

```
addRange(<текстовый фрагмент>)
```

Единственный параметр этого метода задает добавляемый в выделение текстовый фрагмент в виде экземпляра объекта `Range`. Результата этот метод не возвращает.

```
var paraObj = document.getElementById("para");
var paraTRObj = document.createRange();
paraTRObj.selectNodeContents(paraObj);
```

```
var selObj = window.getSelection();
selObj.addRange(paraTRObj);
```

Этот сценарий создает текстовый фрагмент из содержимого абзаца `para` и добавляет его в текущее выделение.

Метод `collapse` сжимает выделение в точку и помещает его в заданное место заданного элемента страницы.

```
collapse(<элемент страницы>, <смещение>)
```

Первый параметр этого метода задает элемент страницы, в который нужно поместить выделение. Второй параметр задает номер символа, начиная от начала заданного элемента страницы, на котором должно находиться выделение. Результата этот метод не возвращает.

```
var selObj = window.getSelection();
selObj.collapse(document.body, 0);
```

Этот сценарий сжимает выделение в точку и помещает его на первый символ секции тела страницы.

Метод `collapseToEnd` переносит начальную точку выделения в то место, где находится ее конечная точка, сжимая таким образом выделение. Этот метод не принимает параметров и не возвращает результата.

Метод `collapseToStart`, наоборот, переносит конечную точку выделения в то место, где находится ее начальная точка, сжимая таким образом выделение. Этот метод также не принимает параметров и не возвращает результата.

Метод `containsNode` позволяет проверить, находится ли заданный элемент страницы в текущем выделении.

```
containsNode(<элемент страницы>, true|false)
```

Первый параметр этого метода задает элемент страницы, который нужно проверить на предмет нахождения в текущем выделении. Если значение второго параметра `true`, то заданный элемент страницы может находиться в выделении частично, если же оно равно `false`, этот элемент страницы должен полностью находиться в выделении.

Метод `containsNode` возвращает `true`, если заданный элемент страницы находится в текущем выделении, частично или полностью, в зависимости от значения второго параметра, и `false`, если он там не находится.

```
var selObj = window.getSelection();
var paraObj = document.getElementById("para");
var f = selObj(paraObj, false);
```

Этот сценарий поместит в переменную `f` значение `true`, если абзац `para` целиком находится в текущем выделении, и `false`, если он не выделен.

Метод `deleteFromDocument` удаляет со страницы выделенный текст. Он не принимает параметров и не возвращает результата.

```
var selObj = window.getSelection();
selObj.deleteFromDocument();
```

Этот сценарий удаляет со страницы текст, выделенный посетителем или программно, из сценария. Вот сюрприз будет…

Метод `extend` перемещает конечную точку выделения в заданное место заданного элемента страницы.

```
extend(<элемент страницы>, <смещение>)
```

Первый параметр этого метода задает элемент страницы, в который нужно поместить конечную точку выделения. Второй параметр задает номер символа от начала заданного элемента страницы, на котором должна находиться конечная точка. Результата этот метод не возвращает.

Метод `getRange` возвращает текстовый фрагмент, присутствующий в выделении, с заданным номером.

```
getRange(<номер текстового фрагмента>)
```

Единственный параметр этого метода задает номер текстового фрагмента, который нужно получить.

```
var selObj = window.getSelection();
var selTRObj = selObj.getRange(0);
var selText = selTRObj.toString();
```

Это выражение помещает в переменную `selText` текст, представляемый первым текстовым фрагментом в выделении. Если посетитель выделил текст вручную, он будет в выделении единственным текстовым фрагментом.

Метод `removeAllRanges` удаляет из выделения все имеющиеся в нем текстовые фрагменты, включая созданный самим Firefox при выделении текста, и те, что были добавлены в выделение программно, вызовом метода `addRange`. Выделение также пропадает. Этот метод не принимает параметров и не возвращает результата.

Метод `removeRange` удаляет из выделения заданный текстовый фрагмент.

```
removeRange(<текстовый фрагмент>)
```

Единственный параметр этого метода задает удаляемый из выделения текстовый фрагмент в виде экземпляра объекта `Range`. Результата этот метод не возвращает.

```
selObj.removeRange(paraTRObj);
```

Это выражение удаляет из выделения созданный ранее текстовый фрагмент, включающий содержимое абзаца `para`.

Метод `selectAllChildren` выделяет все дочерние элементы заданного элемента страницы, не выделяя сам этот элемент. Предыдущее выделение при этом пропадает.

```
selectAllChildren(<элемент страницы>)
```

Единственный параметр этого метода задает элемент страницы, все дочерние элементы которого должны быть выделены. Результата этот метод не возвращает.

```
var selObj = window.getSelection();
selObj.selectAllNode(document.body);
```

Этот сценарий выделяет все содержимое секции тела страницы.

Метод `toString` возвращает текстовое содержимое текущего выделения в виде строки. Этот метод не принимает параметров.

Ранее мы рассмотрели пример страницы с абзацем, выводящей выделенный в этом абзаце фрагмент. Этот пример был реализован под Internet Explorer. Давайте переделаем его под Firefox.

```
<HTML>
  <HEAD>
    <TITLE>Работа с выделенным текстом</TITLE>
    <SCRIPT>
      function parMouseUp() {
        var selObj = window.getSelection();
        var outputObj = document.getElementById("output");
        if (selObj.rangeCount == 0)
          outputObj.firstChild.nodeValue = " "
        else
          outputObj.firstChild.nodeValue = selObj.toString();
      }
    </SCRIPT>
  </HEAD>
  <BODY>
    <P ONMOUSEUP="parMouseUp();">Выделите любой фрагмент этой строки.</P>
    <P ID="output"> </P>
  </BODY>
</HTML>
```

Как видим, код получился даже немного проще, чем в случае Internet Explorer. Firefox позволяет нам получить содержимое выделения прямо из объекта `Selection`, воспользовавшись методом `toString`. Internet Explorer

такого не может — нам пришлось создавать на основе выделения текстовый фрагмент и получать выделенный текст из него.

Пожалуй, на этом можно закончить о работе с выделенным текстом. И на время расстаться с Firefox — все описанные далее в этой главе возможности будут работать только в Internet Explorer.

Работа с Буфером обмена (Internet Explorer)

Internet Explorer позволяет работать с системным Буфером обмена. Мы можем помещать в него данные, извлекать их и разрешать или запрещать для определенных элементов страницы операции вырезания, копирования и вставки. Давайте выясним, как это делается.

За работу с Буфером обмена "отвечает" особый объект `ClipboardData`. Его можно получить, обратившись к свойству `clipboardData` объекта `Window`.

```
var cldObj = window.clipboardData;
```

Выполнив это выражение, мы получим данный объект в переменной `cldObj`.

Объект `ClipboardData` поддерживает три статических метода, которые мы сейчас рассмотрим. Свойств он не поддерживает.

Метод `clearData` удаляет данные из Буфера обмена.

```
clearData([<формат удаляемых данных>])
```

Буфер обмена Windows может содержать одни и те же данные сразу в нескольких форматах. Различные программы могут получить данные в том или ином формате, поддерживаемом данной конкретной программой; так, Microsoft Word может получить фрагмент страницы в формате HTML, а Блокнот — в виде обычного текста. Данные в различных форматах хранятся в Буфере обмена независимо друг от друга, и манипулировать ими также можно независимо.

Так вот, единственный параметр метода `clearData` может задавать формат данных, которые нужно удалить из Буфера обмена. Его значение может быть одной из перечисленных далее строк:

- "Text" — обычный текст;
- "URL" — гиперссылка;
- "File" — файл (вероятно, двоичное содержимое файла);
- "HTML" — HTML-код;
- "Image" — графическое изображение.

Также можно задавать сразу несколько форматов удаляемых данных, перечисляя приведенные ранее строки через запятую. Например:

```
cldObj.clearData("Text, HTML");
```

Это выражение удалит из Буфера обмена данные в текстовом формате и в формате HTML, оставив данные в других форматах (если они там изначально присутствовали).

Если второй параметр метода `clearData` не указан, будут удалены все данные во всех форматах. Результата этот метод не возвращает.

Метод `getData` позволяет получить данные из Буфера обмена.

```
getData(<формат получаемых данных>)
```

Единственный параметр этого метода задает формат получаемых данных в виде строки "Text" (обычный текст) или "URL" (гиперссылка). Метод возвращает полученные данные в виде строки.

```
var s = cldObj.getData("Text");
```

Это выражение помещает в переменную `s` строку, содержащую текстовые данные из Буфера обмена.

Метод `setData` заносит данные в Буфер обмена в указанном формате.

```
setData(<формат заносимых данных>, <заносимые данные>)
```

Первый параметр этого метода задает формат заносимых данных в виде строки "Text" (обычный текст) или "URL" (гиперссылка). Второй параметр указывает собственно заносимые данные также в виде строки.

Метод `setData` возвращает `true`, если данные были успешно занесены в Буфер обмена, и `false` в противном случае.

```
cldObj.setData("Text", "JavaScript");
```

Это выражение заносит в Буфер обмена строку "JavaScript" в виде обычного текста.

А теперь ненадолго отвлечемся от объекта `ClipboardData` и обратимся к элементам страницы, точнее к объекту `HTMLElement`. Он поддерживает целый набор событий, возникающих при выполнении операций с Буфером обмена: вырезания, копирования и вставки. Эти события перечислены в табл. 13.4.

Об этих событиях следует рассказать подробнее. Ведь именно в их обработчиках выполняется работа с содержимым Буфера обмена.

Начнем с событий `onBeforeCopy`, `onBeforeCut` и `onBeforePaste`. Эти события всплывают, и поведение по умолчанию для них может быть отменено. Поведение по умолчанию для этих событий — разрешение или запрет соответствующей операции: копирования, вырезания и вставки.

Таблица 13.4. *События объекта* `HTMLElement`*, возникающие при операциях с Буфером обмена*

Событие	Описание
`onBeforeCopy`	Возникает перед копированием выделенного содержимого элемента страницы в Буфер обмена
`onBeforeCut`	Возникает перед вырезанием выделенного содержимого элемента страницы в Буфер обмена
`onBeforePaste`	Возникает перед вставкой содержимого Буфера обмена в элемент страницы
`onCopy`	Возникает при копировании выделенного содержимого элемента страницы в Буфер обмена
`onCut`	Возникает при вырезании выделенного содержимого элемента страницы в Буфер обмена
`onPaste`	Возникает при вставке содержимого Буфера обмена в элемент страницы

По умолчанию все операции с Буфером обмена — копирование, вырезание и вставки — разрешены только для полей ввода и областей редактирования. Для всех других элементов страницы разрешено только копирование, а операции вырезания и вставки (те, что меняют содержимое страницы) запрещены. Но все это поведение по умолчанию, которое мы можем изменить. И изменим.

Итак, чтобы разрешить запрещенную операцию с Буфером обмена, нужно в обработчике соответствующего события — `onBeforeCopy`, `onBeforeCut` или `onBeforePaste` — присвоить значение `false` свойству `returnValue` объекта `Event` (подробнее об объекте `Event` см. *главу 6*). А присвоение значения `true` этому же свойству запретит соответствующую операцию. Еще раз: значение `false` разрешает операцию, а значение `true` — запрещает.

Например, чтобы разрешить операцию вырезания текста из абзаца, достаточно написать такой обработчик события `onBeforeCut` (выделен полужирным шрифтом):

```
<P ONBEFORECUT="event.returnValue = false;">Вырезаемый текст</P>
```

Если теперь выделить фрагмент текста в этом абзаце и щелкнуть на нем правой кнопкой мыши, пункт контекстного меню **Вырезать** (Cut), обычно недоступный, станет доступным. Также будет работать комбинация клавиш <Ctrl>+<X>, осуществляющая операцию вырезания. И, если ее выполнить, выделенный текст будет действительно вырезан!

Точно так же можно разрешить для другого абзаца операцию вставки:

```
<P ONBEFOREPASTE="event.returnValue = false;">Сюда можно что-то
вставить</P>
```

Или запретить для третьего абзаца операцию копирования:

```
<P ONBEFORECOPY="event.returnValue = true;">Врешь, не возьмешь!</P>
```

Теперь о событиях `onCopy`, `onCut` и `onPaste`. Все они всплывают, и их поведение по умолчанию — выполнение соответствующей операции с Буфером обмена — может быть отменено обычным путем, описанным в *главе 6*.

События `onCopy`, `onCut` и `onPaste` используются, если требуется дополнить или изменить данные, помещаемые в Буфер обмена или извлекаемые из него. Все это выполняется в обработчиках этих событий с помощью рассмотренных ранее методов объекта `ClipboardData`. Единственное — мы должны обязательно отменить поведение для этих событий по умолчанию, для чего достаточно дописать в конце вызова функции — обработчика события уже знакомое нам выражение `return false`.

Проще всего рассмотреть работу с Буфером обмена на примере. Предположим, что мы хотим, чтобы при копировании какого-то текста из абзаца к нему добавлялась строка "Взято с сайта:". Тогда мы должны написать такой код:

```
<SCRIPT>
  function parCopy() {
    var cldObj = window.clipboardData;
    var s = cldObj.getData("Text");
    cldObj.setData("Text", "Взято с сайта: " + s);
  }
</SCRIPT>
. . .
<P ONCOPY="parCopy(); return false;">Копируемый текст</P>
```

Здесь мы написали обработчик события `onCopy`, который извлекает данные в текстовом формате из Буфера обмена, добавляет к ним нужные слова и снова помещает в Буфер обмена. Также мы отменили поведение для события по умолчанию, вписав после вызова функции — обработчика события выражение `return false`, иначе наш код не будет работать правильно.

Точно так же мы можем изменить способ, которым данные из Буфера обмена будут вставляться на страницу. Например, пусть они полностью заменяют содержимое какого-либо абзаца. Реализующий это код может быть таким:

```
<SCRIPT>
  function parPaste() {
    var cldObj = window.clipboardData;
```

```
    var s = cldObj.getData("Text");
    var paraObj = document.getElementById("para");
    paraObj.innerText = s;
  }
</SCRIPT>
. . .
<P ID="para" ONBEFOREPASTE="event.returnValue = false;"
ONPASTE="parPaste(); return false;">Вставьте текст сюда.</P>
```

Здесь для вставки текста в абзац мы использовали свойство `innerText`, описанное в *главе 8*.

> ***Внимание!***
>
> Если в Internet Explorer выполняется вставка текста в элемент страницы, не являющийся полем ввода или областью редактирования, необходимо, чтобы некоторая часть уже имеющегося в этом элементе текста была выделена. Иначе операция вставки не будет доступна. Эта досадная недоработка кочует из одной версии Internet Explorer в другую.

Реализация drag'n'drop с переносом данных (Internet Explorer)

Собственно, drag'n'drop на страницах мы уже реализовывали в *главе 10*. Хоть это и была всего лишь имитация с использованием свободно позиционируемых элементов, но она нормально работала во всех Web-обозревателях. И, надо сказать, обычно большего не требуется.

Но многогранный Internet Explorer идет дальше и предоставляет возможность "настоящего" drag'n'drop с переносом не элементов страницы, а каких-либо данных от одного элемента страницы к другому. (Хорошо нам знакомый drag'n'drop в Windows суть также перенос данных между приложениями.) Для этого он предоставляет особый объект и целый ворох событий и приемов программирования, о которых сейчас пойдет речь.

Но сначала давайте перечислим все этапы выполнения операции drag'n'drop.

1. Посетитель наводит курсор мыши на выделенный фрагмент текста в элементе страницы, нажимает кнопку мыши и, не отпуская ее, начинает перемещение курсора. Операция drag'n'drop начинается именно в этот момент.
2. Когда посетитель наводит курсор мыши на другой элемент страницы, последний сигнализирует, можно или нельзя переместить задействованные в drag'n'drop данные в него. Это выполняется изменением формы курсора мыши.

3. Если данные, задействованные в drag'n'drop, могут быть перемещены в данный элемент страницы, посетитель отпускает кнопку мыши. Операция drag'n'drop на этом заканчивается успехом. Если же данные не могут быть сюда перемещены, операция drag'n'drop также заканчивается, но неуспешно.

> ***Внимание!***
>
> Internet Explorer позволяет выполнить операцию drag'n'drop только над выделенным фрагментом текста в элементе страницы.

Элемент страницы, из которого перемещаются данные, называется *источником* drag'n'drop. А элемент страницы, в который перемещаются данные, называется *приемником* drag'n'drop.

Прежде всего давайте рассмотрим события объекта `HTMLElement`, возникающие в источнике и приемнике во время различных этапов операции drag'n'drop. Этих событий довольно много, и все они перечислены в табл. 13.5.

Таблица 13.5. *События объекта* `HTMLElement`*, возникающие в процессе drag'n'drop*

Событие	Описание
`onDrag`	Возникает в источнике во время перемещения курсора мыши по странице, то есть во время выполнения операции drag'n'drop. Всплывает, поведение по умолчанию (начало drag'n'drop, если источник позволяет это сделать) может быть отменено
`onDragEnd`	Возникает в источнике, когда посетитель отпускает кнопку мыши, тем самым заканчивая операцию drag'n'drop. Всплывает, поведение по умолчанию (завершение drag'n'drop, если источник позволяет это сделать) может быть отменено
`onDragEnter`	Возникает в приемнике, когда посетитель во время операции drag'n'drop помещает курсор мыши над этим приемником. Всплывает, поведение по умолчанию (разрешение приема данных или отказ от этого, в зависимости от приемника) может быть отменено
`onDragLeave`	Возникает в приемнике, когда посетитель во время операции drag'n'drop убирает курсор мыши с этого приемника. Всплывает, поведение по умолчанию как таковое отсутствует
`onDragOver`	Возникает в приемнике, когда посетитель во время операции drag'n'drop перемещает курсор мыши над этим приемником. Всплывает, поведение по умолчанию как таковое отсутствует
`onDragStart`	Возникает в источнике, когда посетитель начинает операцию drag'n'drop. Всплывает, поведение по умолчанию (начало drag'n'drop, если источник позволяет это сделать) может быть отменено
`onDrop`	Возникает в приемнике, когда посетитель отпускает курсор мыши над этим приемником, тем самым заканчивая операцию drag'n'drop. Всплывает, поведение по умолчанию (завершение drag'n'drop, если приемник позволяет это сделать) может быть отменено

Последовательность возникновения этих событий приведена в табл. 13.6.

Таблица 13.6. *Порядок возникновения событий во время drag'n'drop*

Порядок	Действие	События источника	События приемника
1	Посетитель начинает операцию drag'n'drop	`onDragStart`	
2	Посетитель перемещает курсор мыши по странице	`onDrag`	
3	Посетитель помещает курсор мыши над приемником	`onDrag`	`onDragEnter`
4	Посетитель перемещает курсор мыши над приемником	`onDrag`	`onDragOver`
5	Посетитель убирает курсор мыши с приемника	`onDrag`	`onDragLeave`
6	Посетитель отпускает кнопку мыши над приемником, завершая тем самым операцию drag'n'drop	`onDragEnd`	`onDrop`

Немного о поведении по умолчанию для этих событий. Internet Explorer сам выполняет операцию drag'n'drop над текстом, выделенным на странице; этот текст может быть перемещен в поле ввода или область редактирования. Другие элементы страницы по умолчанию не могут принимать текст.

Теперь отвлечемся ненадолго от событий drag'n'drop и обратимся к объекту, который хранит данные, переносимые во время этой операции, и некоторые дополнительные параметры. Это объект `DataTransfer`, доступный через свойство `dataTransfer` объекта `Event`. Он поддерживает свойства и методы, с помощью которых мы можем в определенных пределах управлять операцией drag'n'drop.

Методы у объекта `DataTransfer` те же самые, что у рассмотренного ранее объекта `ClipboardData`: `clearData`, `getData` и `setData`. Форматы вызова, принимаемые параметры и возвращаемые результаты у них точно такие же.

Свойство `dropEffect` объекта `DataTransfer` задает тип операции drag'n'drop, который поддерживается приемником. Оно может принимать следующие строковые значения:

- "copy" — копирование данных;
- "link" — создание ссылки на данные;

- "move" — перемещение данных;
- "none" — приемник не может принять данные.

Так, если мы зададим тип операции "copy", то приемник может выполнить операцию копирования данных из источника, но не сможет выполнить перемещение ("move") и создание ссылки ("link").

Свойство `effectAllowed` объекта `DataTransfer` задает тип операции drag'n'drop, который может выполнить источник. Оно может принимать следующие строковые значения:

- "copy" — копирование данных;
- "link" — создание ссылки на данные;
- "move" — перемещение данных;
- "copyLink" — копирование данных и создание ссылки на них, в зависимости от типа операции, поддерживаемого приемником;
- "copyMove" — копирование и перемещение данных, в зависимости от типа операции, поддерживаемого приемником;
- "moveLink" — перемещение данных и создание ссылки на них, в зависимости от типа операции, поддерживаемого приемником;
- "all" — разрешены все типы операции drag'n'drop;
- "none" — все типы операции drag'n'drop запрещены;
- "uninitialized" — значение по умолчанию; если оно задано, будет выполнено действие по умолчанию.

Здесь, если мы зададим тип операции, скажем, "copy", то данные могут быть скопированы в поддерживающий эту операцию приемник. Но приемник, поддерживающий перемещение данных ("move"), не сможет их принять.

Что ж, сведем все полученные знания воедино и рассмотрим, как же реализуется поддержка drag'n'drop и какие события нам следует обрабатывать в этом случае.

Обратимся к источнику, так как именно он начинает drag'n'drop. Если нас устраивает его поведение по умолчанию — начало операции drag'n'drop с копированием выделенного в нем текста — мы можем не обрабатывать никаких событий. Если же мы хотим вовлечь в drag'n'drop другие данные, задать другой тип операции или вообще отменить ее, нам придется обработать событие `onDragStart` источника.

```
<SCRIPT>
  function sourceDragStart() {
    var sourceObj = document.getElementById("source");
```

```
    with (event.dataTransfer) {
      effectAllowed = "copyMove";
      setData("Text", sourceObj.innerText);
    }
  }
</SCRIPT>
. . .
<P ID="source" ONDRAGSTART="sourceDragStart();">Тащите меня!</P>
```

Здесь мы разрешаем операции перемещения и копирования данных из абзаца `source`, в качестве которых используем все его текстовое содержимое (к которому получили доступ через свойство `innerText` — это самый простой способ в Internet Explorer добраться до текстового содержимого элемента страницы). Заметим, что мы не отменили поведение по умолчанию для этого события, иначе операция drag'n'drop не начнется.

Чтобы запретить для какого-то элемента страницы операцию drag'n'drop, мы должны в обработчике события `onDragStart` присвоить свойству `returnValue` объекта `Event` значение `false`.

```
<P ID="nosource" ONDRAGSTART="event.returnValue = false;">Врешь, не
возьмешь!</P>
```

Теперь текст из абзаца `nosource` никуда нельзя будет перетащить.

> ***НА ЗАМЕТКУ***
>
> Конечно, для отмены операции drag'n'drop можно присвоить свойству `effectAllowed` объекта `DataTransfer` значение "none". Но проще и корректнее использовать для этого свойство `returnValue` объекта `Event`.

Итак, операция drag'n'drop начата. Теперь нужно как-то дать знать Internet Explorer, что данный элемент страницы может выступать как приемник и поддерживает определенный тип операции: копирование, перемещение и пр. Если этот элемент страницы — поле ввода или область редактирования, и нас устраивает его поведение по умолчанию (копирование данных из источника), нам, опять же, ничего делать не нужно. В противном случае нам придется обработать одно из событий приемника: `onDragEnter`, `onDragOver` или `onDragLeave`. Как правило, обрабатываются события `onDragEnter` и `onDragOver`, причем практически всегда для обоих этих событий используется один обработчик.

В этом обработчике нам нужно сделать две вещи. Во-первых, мы обязательно должны отменить поведение по умолчанию, присвоив свойству `returnValue` объекта `Event` значение `false`. Во-вторых, мы должны задать поддерживаемый приемником тип операции drag'n'drop, для чего используется свойство

dropEffect объекта DataTransfer. После этого элемент страницы собственно и станет приемником.

```
<SCRIPT>
  function receiverDragEnter() {
    event.returnValue = false;
    event.dataTransfer.dropEffect = "move";
  }
</SCRIPT>
. . .
<P ID="receiver" ONDRAGENTER="receiverDragEnter();"
ONDRAGOVER="receiverDragEnter();">Тащите все мне!</P>
```

Теперь абзац receiver сможет обрабатывать операции перемещения drag'n'drop.

Все, надоело посетителю, зажав сведенным судорогой пальцем кнопку мыши, таскать ее курсор по странице! Надо куда-то девать задействованные в операции drag'n'drop данные. В случае поля ввода или области редактирования достаточно просто их туда "бросить" — и Internet Explorer сам завершит drag'n'drop. Но другие элементы страницы по умолчанию не могут принимать данные; значит, нам их нужно об этом особо "попросить".

"Просьба" эта заключается в обработке события onDrop приемника. В обработчике данного события мы, во-первых, также должны отменить поведение по умолчанию, а во-вторых, что-то сделать с полученными данными, например, вставить их в требуемый элемент страницы как текстовое содержимое.

```
<SCRIPT>
  function receiverDragEnter() {
    event.returnValue = false;
    event.dataTransfer.dropEffect = "move";
  }

  function receiverDrop() {
    event.returnValue = false;
    var receiverObj = document.getElementById("receiver");
    receiverObj.innerText = event.dataTransfer.getData("Text");
  }
</SCRIPT>
. . .
<P ID="receiver" ONDRAGENTER="receiverDragEnter();"
```

```
ONDRAGOVER="receiverDragEnter();"
ONDROP="receiverDrop();">Тащите все мне!</P>
```

Здесь мы дополнили приведенный ранее пример обработчиком события `onDrop`. Он просто подставляет принятые данные в абзац `receiver`.

Теперь представим такую ситуацию. Мы выполняем перемещение данных с помощью drag'n'drop из источника в приемник, причем в качестве этих данных задали все текстовое содержимое источника. Захватив курсором мыши выделенный фрагмент содержимого источника, мы тащим его в приемник и "отпускаем" там. Данные перемещены, они появились в приемнике и теперь должны исчезнуть из источника. Но Internet Explorer удалит из источника только тот выделенный фрагмент, что мы перетаскивали, — остальной же текст так там и останется.

Если мы выполняем операцию перемещения и сами задаем для нее данные, то нам же придется позаботиться об их удалении из источника в случае нормального завершения этой операции. Для этого следует обработать событие `onDragEnd` источника.

```
<SCRIPT>
  function sourceDragStart() {
    var sourceObj = document.getElementById("source");
    with (event.dataTransfer) {
      effectAllowed = "copyMove";
      setData("Text", sourceObj.innerText);
    }
  }

  function sourceDragEnd() {
    var sourceObj = document.getElementById("source");
    sourceObj.innerText = "";
  }
</SCRIPT>
. . .
<P ID="source" ONDRAGSTART="sourceDragStart();"
ONDRAGEND="sourceDragEnd();>Тащите меня!</P>
```

Вот теперь все.

Для примера давайте рассмотрим две страницы, попроще и посложнее. Первая страница, которая попроще, будет содержать абзац-источник `sender` и абзац-приемник `receiver`. Посетитель может выделить часть текста в абзаце-

источнике и переместить ее в абзац-приемник. HTML-код этой страницы приведен далее.

```
<HTML>
  <HEAD>
    <TITLE>Drag'n'drop</TITLE>
    <SCRIPT>
      function receiverDragEnter() {
        event.returnValue = false;
        event.dataTransfer.dropEffect = "move";
      }

      function receiverDrop() {
        event.returnValue = false;
        var receiverObj = document.getElementById("receiver");
        receiverObj.innerText = event.dataTransfer.getData("Text");
      }
    </SCRIPT>
  </HEAD>
  <BODY>
    <P ID="sender"
    ONDRAGSTART="event.dataTransfer.effectAllowed = 'move';">Перетащите
    любой фрагмент этой строки...</P>
    <P ID="receiver" ONDRAGENTER="receiverDragEnter();"
    ONDRAGOVER="receiverDragEnter();"
    ONDROP="receiverDrop();">...сюда</P>
  </BODY>
</HTML>
```

Здесь все довольно просто.

- В обработчике события `onDragStart` источника мы задаем операцию, поддерживаемую источником, — перемещение.
- В обработчиках событий `onDragEnter` и `onDragOver` приемника мы отменяем поведение по умолчанию и задаем операцию, поддерживаемую приемником, — также перемещение.
- В обработчике события `onDrop` приемника мы помещаем задействованные в drag'n'drop данные (выделенный фрагмент текстового содержимого источника) в приемник. И не забываем отменить поведение по умолчанию.

Отметим, что самим удалять перемещенный фрагмент текста из источника нам не нужно. Поскольку мы сами не задавали данные для перемещения,

а оставили те, что задал Internet Explorer (то есть выделенный фрагмент текста источника), то Internet Explorer все сделает за нас.

Вторая страница, которая посложнее, будет содержать два источника и два приемника.

- Источник `sender1` поддерживает как копирование, так и перемещение.
- Источник `sender2` поддерживает только копирование.
- Приемник `receiver1` поддерживает копирование.
- Приемник `receiver2` поддерживает перемещение.

При этом в качестве данных, копируемых или перемещаемых операцией drag'n'drop, будет выступать все текстовое содержимое соответствующего источника.

```
<HTML>
  <HEAD>
    <TITLE>Drag'n'drop</TITLE>
    <SCRIPT>
      var isMove = false;

      function senderDragStart(pOp) {
        var senderObj = event.srcElement;
        with (event.dataTransfer) {
          effectAllowed = pOp;
          setData("Text", senderObj.innerText);
        }
      }

      function senderDragEnd() {
        if (isMove) {
          var senderObj = event.srcElement;
          senderObj.innerText = "";
          isMove = false;
        }
      }

      function receiverDragEnter(pOp) {
        event.returnValue = false;
        event.dataTransfer.dropEffect = pOp;
        isMove = (pOp == "move");
```

```
    }

    function receiverDrop() {
      event.returnValue = false;
      var receiverObj = event.srcElement;
      receiverObj.innerText = event.dataTransfer.getData("Text");
    }
  </SCRIPT>
 </HEAD>
 <BODY>
  <P ID="sender1" ONDRAGSTART="senderDragStart('copyMove');"
  ONDRAGEND="senderDragEnd();">Перетащите любой фрагмент этой...</P>
  <P ID="sender2" ONDRAGSTART="senderDragStart('copy');">или этой
  строки...</P>
  <P ID="receiver1" ONDRAGENTER="receiverDragEnter('copy');"
  ONDRAGOVER="receiverDragEnter('copy');"
  ONDROP="receiverDrop();">...сюда</P>
  <P ID="receiver1" ONDRAGENTER="receiverDragEnter('move');"
  ONDRAGOVER="receiverDragEnter('move');"
  ONDROP="receiverDrop();">...или сюда</P>
 </BODY>
</HTML>
```

Здесь мы объявили переменную `isMove`, которая будет обозначать, выполняется в данный момент операция копирования (значение `false`) или перемещения (значение `true`). К сожалению, другого способа отследить выполняемую с помощью drag'n'drop операцию, похоже, не существует.

- В обработчике события `onDragStart` источника мы задаем операцию, поддерживаемую источником, — только копирование или копирование и перемещение. Обозначение операции передается в функцию-обработчик единственным параметром — так проще. Также мы задаем в качестве данных, вовлеченных в drag'n'drop, все текстовое содержимое соответствующего источника.
- В обработчиках событий `onDragEnter` и `onDragOver` приемника мы отменяем поведение по умолчанию и задаем операцию, поддерживаемую приемником, — копирование или перемещение. Обозначение этой операции также передается в функцию-обработчик единственным параметром. Также мы задаем для переменной `isMove` значение `true`, если в качестве операции задано перемещение.

- В обработчике события `onDrop` приемника мы помещаем задействованные в drag'n'drop данные (текстовое содержимое источника) в приемник. И не забываем отменить поведение по умолчанию.
- В обработчике события `onDragEnd` мы проверяем значение переменной `isMove` и, если оно равно `true` (то есть выполняется операция перемещения), удаляем содержимое источника `sender1` (который, как мы помним, поддерживает перемещение) и присваиваем переменной `isMove` значение `false`.

Еще можно отметить, что для доступа к элементу страницы, в котором возникло событие, мы используем свойство `srcElement` объекта `Event`. Поскольку мы пишем под Internet Explorer, то можем смело использовать его специфические возможности, не заботясь о совместимости с другими Web-обозревателями.

Описание реализации drag'n'drop в Internet Explorer вышло довольно длинным. Описание других его возможностей вряд ли будет много короче…

Использование диалоговых окон HTML (Internet Explorer)

Похоже, разработчики Internet Explorer прочат свое творение в платформы для создания настоящих приложений… Поддержка drag'n'drop, работа с Буфером обмена, диалоговые окна, HTML-приложения (о них мы поговорим в конце этой главы), работа с базами данных (ей будет посвящена *глава 14*) — все это наводит на такие мысли.

Возьмем, например, диалоговые окна. Многие Windows-приложения используют их, чтобы запросить у пользователя какие-то данные. Вот и Internet Explorer позволяет выводить диалоговые окна, в качестве которых выступают обычные Web-страницы с формами и элементами управления, загружаемые в отдельном окне. Они так и называются — *диалоговые окна HTML*.

Как и диалоговое окна Windows-приложений, их "коллеги" мира HTML могут быть модальными и немодальными. *Модальное* диалоговое окно находится поверх создавшего его окна Web-обозревателя и не позволяет посетителю переключиться на него. *Немодальное* же диалоговое окно такую возможность предоставляет.

Модальные диалоговые окна HTML обычно используют для того, чтобы получить от посетителя какие-либо данные, необходимые для дальнейшей работы сценария (например, имя и пароль для входа в закрытый раздел сайта). Пока посетитель не введет необходимые данные в модальное окно и не нажмет

кнопку **OK** или **Отмена**, он не сможет продолжить работу с сайтом или Web-приложением. Область применения немодальных диалоговых окон заметно у́же — это ввод различных вспомогательных данных или задание параметров, влияющих на отображение информации на Web-странице.

Модальные диалоговые окна HTML

Начнем мы с модальных диалоговых окон HTML, так как применяются они чаще и работать с ними проще — Internet Explorer предоставляет для этого весьма удобные средства. Немодальные диалоговые окна создаются примерно так же, но работа с ними выполняется с помощью других средств.

Итак, что же нужно для создания диалогового окна как модального, так и немодального? Прежде всего, Web-страница, которая и станет содержимым диалогового окна. Эта страница должна содержать форму с элементами управления, в которые посетитель и будет вводить данные.

Для вывода на экран модального диалогового окна HTML используется метод `showModalDialog` объекта `Window`. Вот формат его вызова:

```
showModalDialog(<интернет-адрес Web-страницы, которая станет содержимым
↳диалогового окна>
↳[, <данные, передаваемые диалоговому окну>
↳[, <параметры диалогового окна>]]);
```

Первый параметр этого метода задает интернет-адрес Web-страницы, которая станет содержимым диалогового окна HTML. Понятно, что он должен быть задан в строковом виде.

Второй — необязательный — параметр задает данные, которые должны быть переданы диалоговому окну HTML (в смысле, открытой в нем странице). Это могут быть, например, первоначальные значения для элементов управления в форме (это обычная практика). Они передаются в любом нужном формате: единственная переменная строкового, числового или логического типа, массив или экземпляр какого-либо объекта, в том числе пользовательского.

Третий, также необязательный, параметр задает строку, содержащую список параметров создаваемого диалогового окна, разделенных запятыми. Здесь тот же принцип, что и у третьего параметра метода `open` объекта `Window`, рассмотренного нами в *главе 7*. Все доступные для задания параметры перечислены в табл. 13.7.

Для задания координат и размеров диалогового окна Internet Explorer поддерживает все единицы измерения, доступные в CSS (см. *главу 3*). По умолчанию, если обозначение единицы измерения не указано, используются пикселы.

Таблица 13.7. *Параметры диалогового окна HTML*

Свойство окна	Описание
`center=yes|no`	Если `yes`, то создаваемое окно будет находиться в центре экрана, если `no`, его местоположение будет задано самой Windows. Значение по умолчанию — `yes`
`dialogHeight:<значение>`	Высота создаваемого окна
`dialogHide=yes|no`	Если `yes`, то создаваемое окно будет скрываться при печати или предварительном просмотре перед печатью. Значение по умолчанию — `no`
`dialogLeft:<значение>`	Горизонтальная координата левого верхнего угла создаваемого окна относительно экрана
`dialogTop:<значение>`	Вертикальная координата левого верхнего угла создаваемого окна относительно экрана
`dialogWidth:<значение>`	Ширина создаваемого окна
`edge=sunken|raised`	Задает вид границы окна: вдавленный (`sunken`) или выпуклый (`raised`). Значение по умолчанию — `raised`
`help=yes|no`	Если `yes`, то создаваемое окно будет отображать в строке заголовка кнопку вызова контекстной справки. Значение по умолчанию — `yes`. Internet Explorer 7 этот параметр игнорирует
`resizable=yes|no`	Если `yes`, то посетитель сможет изменять размеры создаваемого окна. Значение по умолчанию — `no`
`scroll=yes|no`	Если `yes`, то создаваемое окно будет отображать полосы прокрутки, если содержимое Web-страницы не помещается в нем. Значение по умолчанию — `yes`
`status=yes|no`	Если `yes`, то создаваемое окно будет содержать строку статуса. Значение по умолчанию может в разных случаях быть различным, поэтому лучше задать это свойство явно
`unadorned=yes|no`	Если `yes`, то создаваемое окно будет "украшено". Что это значит, автору установить не удалось, т. к. никакой видимой разницы между "украшенным" и "неукрашенным" окном нет. Значение по умолчанию — `no`

Метод `showModalDialog` объекта `Window` сразу после вызова выводит на экран диалоговое окно HTML с заданной страницей и заданными параметрами. При этом выполнение сценария, содержащего вызов этого метода, приостанавливается, пока диалоговое окно не будет закрыто. После закрытия диалогового окна метод `showModalDialog` в качестве результата возвращает данные, переданные от диалогового окна (то есть от открытой в нем страницы).

```
var result = window.showModalDialog("form.html", [12, 45],
"resizable=no,scroll=no,status=no");
```

Это выражение выводит на экран модальное диалоговое окно HTML, содержащее страницу form.html, с неизменяемыми размерами, без полос прокрутки и строки статуса. Кроме того, оно передает созданному диалоговому окну массив с числами 12 и 45. После закрытия диалогового окна это выражение присваивает переменной `result` данные, переданные диалоговым окном, и выполнение сценария продолжается.

> ***НА ЗАМЕТКУ***
>
> Практически все диалоговые окна имеют неизменяемые размеры, не содержат полос прокрутки и строки статуса. Модальные диалоговые окна, кроме того, всегда выводятся в центре экрана. Так что приведенная ранее строка параметров диалогового окна является, можно сказать, стандартом. Правда, в ней не хватает размеров диалогового окна, но их можно подобрать только экспериментально.

Итак, модальное диалоговое окно HTML открыто. Обратимся к загруженной в нем странице (ее интернет-адрес, как мы уже знаем, передается первым параметром метода `showModalDialog`). Как она может получить отправленные через второй параметр этого метода данные? Очень просто — через свойство `dialogArguments` объекта `Window`. Вот так:

```
var parArr = window.dialogArguments;
```

И в переменной `parArr` окажется переданный вторым параметром метода `showModalDialog` массив с числами 12 и 45.

Можно написать и вот такой код:

```
var par1 = window.dialogArguments[0];
var par2 = window.dialogArguments[1];
```

то есть сразу же извлечь числа, благо свойство `dialogArguments` и так содержит массив. Тогда в переменной `par1` окажется число 12, а в переменной `par2` — число 45.

Теперь мы можем делать с полученными данными все, что пожелаем. Например, мы можем подставить эти данные в элементы управления Web-формы страницы, открытой в диалоговом окне, в качестве первоначальных значений.

Хорошо! Посетитель ввел данные в форму и нажал кнопку **OK**. Теперь диалоговое окно нужно закрыть и передать введенные в него данные окну, которое его открыло. Для этого случая объект `Window` поддерживает свойство `returnValue`. Присваиваем этому свойству данные, которые нужно вернуть в открывшую это окно страницу, например:

```
window.returnValue = 123;
```

или

```
window.returnValue = ["user", "password"];
```

после чего закрываем диалоговое окно:

```
window.close();
```

Метод `showModalDialog` в сценарии, создавшем диалоговое окно, возвращает переданные из диалогового окна данные. И мы можем их использовать.

> ***НА ЗАМЕТКУ***
>
> Кроме того, объект `Window` в Internet Explorer поддерживает свойства `dialogLeft`, `dialogTop`, `dialogWidth` и `dialogHeight`. Эти свойства соответствуют описанным в табл. 13.7 параметрам диалогового окна.

Правила хорошего тона программирования рекомендуют оснащать диалоговые окна также и кнопкой **Отмена**. При нажатии этой кнопки диалоговое окно должно просто закрываться вызовом метода `close` объекта `Window` без передачи каких-либо данных. Тогда метод `showModalDialog` вернет значение `null`.

Для примера давайте создадим две страницы, одна из которых является содержимым модального диалогового окна, позволяющего выбрать цвет текста второй страницы (основной) и сделать его полужирным.

HTML-код основной страницы, которая вызывает диалоговое окно, приведен далее.

```
<HTML>
  <HEAD>
    <TITLE>Вызов модального диалогового окна</TITLE>
    <SCRIPT>
      var textColor = "#000000";
      var isTextBold = false;

      function aClick() {
        var result = window.showModalDialog("13.6.htm",
        [textColor, isTextBold], "resizable=no,scroll=no,status=no");
        if (result) {
```

```
          textColor = result[0];
          isTextBold = result[1];
          var parObj = document.getElementById("par");
          parObj.style.color = textColor;
          parObj.style.fontWeight = isTextBold ? "bold" : "normal";
        }
      }
    </SCRIPT>
  </HEAD>
  <BODY>
    <P ID="par">Цвет этого текста можно изменить, воспользовавшись
    модальным диалоговым окном.</P>
    <P><A HREF="#" ONCLICK="aClick();">Задать параметры текста</A></P>
  </BODY>
</HTML>
```

Видно, что для вызова диалогового окна настройки параметров текста мы используем функцию — обработчик события `onClick` специальной гиперссылки. Эта функция выводит на экран диалоговое окно, чье содержимое хранится на странице 13.6.htm (это вторая страница в нашем примере), и передает этой странице в качестве параметров массив с двумя элементами: строковым (цвет) и логическим (полужирный или обычный шрифт) значениями. От страницы 13.6.htm обработчик получает также массив с двумя элементами и использует их значения для задания параметров шрифта абзаца.

Отметим, что мы сохраняем значение цвета и признак полужирности в особых переменных `textColor` и `isTextBold`. Это нужно, чтобы при последующих вызовах диалогового окна передать эти значения ему. Изначально эти переменные имеют значения "#000000" (черный цвет) и `false` (обычный шрифт).

Сохраним эту страницу в файл с именем 13.5.htm. И рассмотрим HTML-код второй страницы, задающей содержимое диалогового окна.

```
<HTML>
  <HEAD>
    <TITLE>Модальное диалоговое окно</TITLE>
    <SCRIPT>
      function okClick() {
        var boldObj = document.getElementById("bold");
        var colorObjs = document.getElementsByName("color");
        var color = "";
```

```
      for (var i = 0; i < colorObjs.length; i++) {
        if (colorObjs[i].checked) {
          color = colorObjs[i].value;
          break;
        }
      }
      window.returnValue = [color, boldObj.checked];
      window.close();
    }
  </SCRIPT>
</HEAD>
<BODY>
  <FORM ACTION="#">
    Цвет текста:<BR>
    <INPUT TYPE="radio" ID="color" NAME="color" VALUE="#000000">
    Черный<BR>
    <INPUT TYPE="radio" ID="color" NAME="color" VALUE="#FF0000">
    Красный<BR>
    <INPUT TYPE="radio" ID="color" NAME="color" VALUE="#00FF00">
    Зеленый<BR>
    <INPUT TYPE="radio" ID="color" NAME="color" VALUE="#0000FF">
    Синий<BR>
    Полужирный текст:
    <INPUT TYPE="checkbox" ID="bold" NAME="bold"><BR><BR><BR>
    <INPUT TYPE="button" ID="btnOK" NAME="btnOK" VALUE="OK"
    ONCLICK="okClick();"><BR><BR>
    <INPUT TYPE="button" ID="btnCancel" NAME="btnCancel" VALUE="Отмена"
    ONCLICK="window.close();">
  </FORM>
  <SCRIPT>
    var boldObj = document.getElementById("bold");
    var colorObjs = document.getElementsByName("color");
    var colorObj = null;
    for (var i = 0; i < colorObjs.length; i++) {
      colorObj = colorObjs[i];
      if (colorObj.value == window.dialogArguments[0]) break;
    }
    colorObj.checked = true;
```

```
      boldObj.checked = window.dialogArguments[1];
    </SCRIPT>
  </BODY>
</HTML>
```

Здесь мы принимаем от вызывающего окна параметры — массив с двумя элементами — и используем его для установки изначальных значений элементов управления в форме. Принципы работы с переключателями и флажками обсуждались в *главе 12*, так что ничего нового для нас здесь нет.

При нажатии кнопки **OK** функция — обработчик события `onClick` этой кнопки получает новые значения элементов управления, формирует из них массив с двумя элементами, присваивает его свойству `returnValue` объекта `Window` и закрывает текущее окно. Если посетитель нажал кнопку **Отмена**, мы просто закрываем текущее окно без всяких дополнительных действий.

Сохраним эту страницу в файле под именем 13.6.htm. После этого можем загрузить в Internet Explorer страницу 13.5.htm и попробовать созданный нами код в действии. Если мы не допустили ошибок, все должно работать.

Немодальные диалоговые окна HTML

Немодальные диалоговые окна HTML применяются значительно реже. Это связано, в частности, и с тем, что Internet Explorer не предлагает столь удобных средств для работы с ними, как с модальными окнами.

Содержимое немодального диалогового окна также описывает особая Web-страница. В этом смысле оба этих типа диалоговых окон схожи.

Для вывода немодального диалогового окна HTML на экран используется другой метод объекта `Window` — `showModelessDialog`. Вот формат его вызова:

```
showModelessDialog(<интернет-адрес Web-страницы, которая станет
↳содержимым диалогового окна>
↳[, <данные, передаваемые диалоговому окну>
↳[, <параметры диалогового окна>]]);
```

Как видим, параметры этого метода такие же, как и у его "коллеги" `showModalDialog`. Возвращает метод `showModelessDialog` экземпляр объекта `Window`, соответствующий созданному диалоговому окну.

Сразу после вызова метода `showModelessDialog` немодальное диалоговое окно создается и выводится на экран. При этом сценарий, содержащий вызов этого метода, продолжает выполняться.

Стоп, но как же нам передать данные из немодального диалогового окна странице, что его открыла? Стандартного способа сделать это нет... Придется изобретать очередной "финт ушами"...

Сначала в странице, которая выводит на экран немодальное диалоговое окно, нам будет нужно объявить набор переменных для обмена данными — по одной переменной на каждую единицу данных. Перед выводом диалогового окна мы присваиваем этим переменным все необходимые значения, например:

```
var param1 = 100;
var param2 = "";
```

После этого можно вывести на экран немодальное диалоговое окно, воспользовавшись таким синтаксисом:

```
window.showModelessDialog(<интернет-адрес Web-страницы>, window,
↳[, <параметры диалогового окна>]]);
```

Заметим, что вторым параметром метода `showModelessDialog` мы передаем текущее окно Web-обозревателя.

В странице, открытой в диалоговом окне, получаем переданное окно:

```
var parentWnd = window.dialogArguments;
```

после чего получаем сами переданные данные, извлекая их из объявленных ранее переменных:

```
var tempParam1 = parentWnd.param1;
var tempParam2 = parentWnd.param2;
```

Если же нам нужно передать какие-либо данные из диалогового окна создавшей его странице, мы просто присваиваем их объявленным ранее переменным:

```
parentWnd.param1 = 200;
parentWnd.param2 = "#FFFFFF";
```

Иногда нам нужно при пересылке данных из диалогового окна создавшей его страницы выполнять еще какие-то действия. Для этого мы можем объявить в странице, создающей окно, особую функцию:

```
function applyData() {
  // Здесь мы что-либо делаем с принятыми от диалогового окна данными
}
```

После передачи данных мы вызываем эту функцию из немодального диалогового окна:

```
parentWnd.param1 = 200;
parentWnd.param2 = "#FFFFFF";
parentWnd.applyData();
```

Так, кстати, часто и делают.

Обычно немодальное диалоговое окно отправляет данные при нажатии кнопки **OK** или **Применить**. Также правила хорошего тона программирования

рекомендуют оснащать немодальные окна кнопкой **Закрыть**, которая при нажатии закрывает это окно.

В качестве примера давайте возьмем созданные ранее страницы 13.5.htm и 13.6.htm и изменим их так, чтобы для задания параметров текста первой страницы использовалось немодальное диалоговое окно.

HTML-код исправленной первой страницы приведен далее.

```
<HTML>
  <HEAD>
    <TITLE>Вызов немодального диалогового окна</TITLE>
    <SCRIPT>
      var textColor = "#000000";
      var isTextBold = false;

      function aClick() {
        window.showModelessDialog("13.8.htm", window,
        "resizable=no,scroll=no,status=no");
      }
    </SCRIPT>
  </HEAD>
  <BODY>
    <P ID="par">Цвет этого текста можно изменить, воспользовавшись
    немодальным диалоговым окном.</P>
    <P><A HREF="#" ONCLICK="aClick();">Задать параметры текста</A></P>
    <SCRIPT>
      var parObj = document.getElementById("par");
    </SCRIPT>
  </BODY>
</HTML>
```

Сначала мы подготавливаем необходимые переменные: `textColor` — для хранения цвета, `isTextBold` — для хранения признака полужирности шрифта и `parObj` — для доступа к абзацу, параметры шрифта которого мы собираемся задавать. При щелчке на гиперссылке мы выводим немодальное диалоговое окно и передаем ему в качестве единственного параметра текущее окно.

Сохраним эту страницу в файл с именем 13.7.htm. И рассмотрим HTML-код измененной второй страницы — собственно немодального диалогового окна.

```
<HTML>
  <HEAD>
    <TITLE>Немодальное диалоговое окно</TITLE>
```

```
  <SCRIPT>
    function applyClick() {
      var colorObjs = document.getElementsByName("color");
      var color = "";
      for (var i = 0; i < colorObjs.length; i++) {
        if (colorObjs[i].checked) {
          color = colorObjs[i].value;
          break;
        }
      }
      var boldObj = document.getElementById("bold");
      parentWnd.textColor = color;
      parentWnd.parObj.style.color = parentWnd.textColor;
      parentWnd.isTextBold = boldObj.checked;
      parentWnd.parObj.style.fontWeight = boldObj.checked ? "bold" :
      "normal";
    }
  </SCRIPT>
</HEAD>
<BODY>
  <FORM ACTION="#">
    Цвет текста:<BR>
    <INPUT TYPE="radio" ID="color" NAME="color" VALUE="#000000">
    Черный<BR>
    <INPUT TYPE="radio" ID="color" NAME="color" VALUE="#FF0000">
    Красный<BR>
    <INPUT TYPE="radio" ID="color" NAME="color" VALUE="#00FF00">
    Зеленый<BR>
    <INPUT TYPE="radio" ID="color" NAME="color" VALUE="#0000FF">
    Синий<BR>
    Полужирный текст:
    <INPUT TYPE="checkbox" ID="bold" NAME="bold"><BR><BR><BR>
    <INPUT TYPE="button" ID="btnApply" NAME="btnApply"
    VALUE="Применить" ONCLICK="applyClick();"><BR><BR>
    <INPUT TYPE="button" ID="btnClose" NAME="btnClose" VALUE="Закрыть"
    ONCLICK="window.close();">
  </FORM>
  <SCRIPT>
```

```
      var parentWnd = window.dialogArguments;
      var boldObj = document.getElementById("bold");
      var colorObjs = document.getElementsByName("color");
      var colorObj = null;
      for (var i = 0; i < colorObjs.length; i++) {
        colorObj = colorObjs[i];
        if (colorObj.value == parentWnd.textColor) break;
      }
      colorObj.checked = true;
      boldObj.checked = parentWnd.isTextBold;
    </SCRIPT>
  </BODY>
</HTML>
```

Здесь мы получаем параметр, переданный диалоговому окну, — открывшее его окно — и через него "добираемся" до объявленных в том окне переменных. В остальном здесь все то же самое, что и в странице, формирующей модальное диалоговое окно из предыдущего примера.

Теперь нам останется сохранить эту страницу в файле под именем 13.8.htm, загрузить в Internet Explorer страницу 13.7.htm и попробовать обе эти страницы в действии. А о диалоговых окнах HTML больше рассказывать нечего.

HTML-приложения (Internet Explorer)

Последнее, что мы рассмотрим в данной главе, — это HTML-приложения Internet Explorer. Они используются довольно часто; например, утилиты инсталляции Microsoft Visual Studio 2008 Express Edition и Borland Turbo Delphi Explorer выполнены именно в таком виде.

HTML-приложения — это клиентские Windows-приложения, написанные с использованием HTML, CSS и JavaScript в виде Web-страницы и выполняющиеся Internet Explorer. Они запускаются так же, как обычные приложения Windows, отображаются в своем собственном окне, параметры которого мы можем менять в довольно широких пределах, выводятся в списке Диспетчера задач Windows и могут показываться на Панели задач. Также они имеют свое имя, под которым и отображаются в списке задач, и, возможно, свой значок, показываемый в Проводнике.

HTML-приложения отличаются от обычных Web-страниц тем, что Internet Explorer считает их содержимое изначально безопасным. Это значит, что мы можем помещать в HTML-приложения элементы ActiveX, и они будут выводиться без всяких предупреждений.

Но главных отличий HTML-приложений от обычных страниц два. Во-первых, файл HTML-приложения имеет расширение hta, а не htm[l]. Во-вторых, в коде HTML-приложения присутствует особый парный тег `HTA:APPLICATION`, атрибуты которого задают различные параметры данного приложения. Давайте его рассмотрим.

Итак, парный тег `HTA:APPLICATION`. Именно он превращает обычную Web-страницу в HTML-приложение. Располагается он в секции заголовка (то есть внутри парного тега `<HEAD>`) и не имеет в себе никакого содержимого. Зато он имеет множество атрибутов, с помощью которых задаются различные параметры HTML-приложения. Эти атрибуты перечислены в табл. 13.8.

Таблица 13.8. *Атрибуты тега* `HTA:APPLICATION`

Атрибут	Описание
`APPLICATIONNAME`	Задает имя HTML-приложения в строковом виде. Под этим именем приложение будет присутствовать в списке Диспетчера задач и Панели задач Windows
`BORDER`	Задает вид рамки окна HTML-приложения. Доступны значения: "thick" (толстая рамка с возможностью изменения размера окна; значение по умолчанию), "dialog" (толстая рамка без возможности изменения размеров окна), "thin" (тонкая рамка без возможности изменения размеров окна) и "none" (рамка отсутствует)
`BORDERSTYLE`	Задает вид обрамления содержимого окна HTML-приложения, т. е. самой страницы. Доступны значения: "normal" (обычное; значение по умолчанию), "raised" (трехмерное приподнятое), "sunken" (трехмерное вдавленное), "complex" (трехмерное приподнятое и вдавленное) и "static" (трехмерное, используемое, в основном, в приложениях, не требующих пользовательского ввода)
`CAPTION`	Если "yes" (значение по умолчанию), окно HTML-приложения будет содержать заголовок, если "no" — не будет
`CONTEXTMENU`	Если "yes" (значение по умолчанию), при щелчке правой кнопкой мыши по странице, являющейся HTML-приложением, появится контекстное меню, если "no" — не появится
`ICON`	Задает имя файла значка, используемого Проводником Windows для показа файла HTML-приложения. Используется обычный файл значков Windows с расширением ico; размеры значка — 32×32 пиксела
`INNERBORDER`	Если "yes" (значение по умолчанию), между границами окна HTML-приложения и содержимым выводимой в нем Web-страницы будет отображена рамка, если "no" — не будет

Таблица 13.8 *(окончание)*

Атрибут	Описание
MAXIMIZEBUTTON	Если "yes" (значение по умолчанию), заголовок окна HTML-приложения будет содержать кнопку максимизации, если "no" — не будет
MINIMIZEBUTTON	Если "yes" (значение по умолчанию), заголовок окна HTML-приложения будет содержать кнопку минимизации, если "no" — не будет
NAVIGABLE	Если "no" (значение по умолчанию), то при щелчке на гиперссылках в HTML-приложении целевые страницы будут загружены в новое окно Web-обозревателя, если "yes" — в то же самое окно
SCROLL	Если "yes", в окне HTML-приложения всегда будут присутствовать полосы прокрутки, если "no" — не будут присутствовать в любом случае, если "auto" — будут присутствовать, только если содержимое Web-страницы не помещается в окне
SCROLLFLAT	Если "no" (значение по умолчанию), полосы прокрутки будут иметь трехмерный вид, если "yes" — плоский вид
SELECTION	Если "yes" (значение по умолчанию), содержимое Web-страницы, являющейся HTML-приложением, можно выделить, если "no" — нельзя
SHOWINTASKBAR	Если "yes" (значение по умолчанию), HTML-приложение будет присутствовать в Панели задач Windows, если "no" — не будет. В списке Диспетчера задач оно будет присутствовать в любом случае
SINGLEINSTANCE	Если "no" (значение по умолчанию), пользователь может запустить одновременно несколько копий этого HTML-приложения, если "yes" — только одну копию
SYSMENU	Если "yes" (значение по умолчанию), заголовок окна HTML-приложения будет содержать системное меню, если "no" — не будет
VERSION	Задает версию HTML-приложения в строковом виде
WINDOWSTATE	Задает начальное состояние окна HTML-приложения, которое оно примет сразу же после открытия. Доступны значения: "normal" (обычное; значение по умолчанию), "minimize" (минимизированное) и "maximize" (максимизированное)

Само содержимое HTML-приложения помещается, как и в случае обычной Web-страницы, в секции тела (в теге <BODY>). Ведь, как мы помним, HTML-приложение — фактически обычная Web-страница.

ВНИМАНИЕ!

Если HTML-приложение содержит фреймы, в которых отображаются другие Web-страницы, их содержимое Web-обозреватель считает небезопасным. Со всеми вытекающими последствиями; так, если страница, открытая во фрейме, содержит элемент ActiveX, Web-обозреватель выдаст пользователю предупреждение о возможно небезопасном содержимом. Чтобы заставить Web-обозреватель "проникнуться доверием" к открытой во фрейме странице, нужно поместить в создающий этот фрейм тег `<FRAME>` атрибут `APPLICATION` и присвоить этому атрибуту значение "yes". (Значение этого атрибута по умолчанию — "no" — говорит о небезопасности содержимого фрейма.)

В сценарии тег `HTA:APPLICATION` представляется как экземпляр особого объекта. Автору не удалось выяснить имя этого объекта, поэтому назовем его `HTAApplication`. Мы можем обратиться к нему из сценария по его имени (да, тег поддерживает хорошо знакомый нам атрибут `ID`, с помощью которого задается его имя) и получить различные сведения о HTML-приложении с помощью различных свойств этого объекта. Все поддерживаемые им свойства перечислены в табл. 13.9. Методов и событий объект `HTAApplication` не поддерживает.

Таблица 13.9. *Свойства объекта* `HTAApplication`

Свойство	Описание
`applicationName`	Возвращает имя HTML-приложения в строковом виде
`border`	Возвращает вид рамки окна HTML-приложения в строковом виде. Все доступные значения перечислены в табл. 13.8
`borderStyle`	Возвращает вид обрамления содержимого окна HTML-приложения в строковом виде. Все доступные значения перечислены в табл. 13.8
`caption`	Возвращает "yes", если окно HTML-приложения содержит заголовок, и "no" в противном случае
`commandLine`	Возвращает в строковом виде параметры командной строки, использованные при запуске HTML-приложения
`contextMenu`	Возвращает "yes", если в HTML-приложении разрешен вывод контекстного меню при щелчке правой кнопкой мыши, и "no" в противном случае
`icon`	Возвращает имя файла значка, используемого Проводником Windows для вывода файла HTML-приложения
`innerBorder`	Возвращает "yes", если между границами окна HTML-приложения и содержимым отображаемой в нем Web-страницы присутствует рамка, и "no" в противном случае
`maximizeButton`	Возвращает "yes", если заголовок окна HTML-приложения содержит кнопку максимизации, и "no" в противном случае
`minimizeButton`	Возвращает "yes", если заголовок окна HTML-приложения содержит кнопку минимизации, и "no" в противном случае

Таблица 13.9 *(окончание)*

Свойство	Описание
navigable	Возвращает "yes", если при щелчке на гиперссылках, содержащихся на Web-странице, соответствующие им страницы загружаются в то же самое окно Web-обозревателя, и "no", если они загружаются в новое окно
scroll	Возвращает "yes", если в окне HTML-приложения всегда присутствуют полосы прокрутки, "no", если они не присутствуют в любом случае, и "auto", если они присутствуют, только когда содержимое Web-страницы не помещается в окне
scrollFlat	Возвращает "yes", если полосы прокрутки имеют плоский вид, и "no", если они имеют трехмерный вид
selection	Возвращает "yes", если содержимое Web-страницы можно выделить, и "no", если нельзя
showInTaskBar	Возвращает "yes", если HTML-приложение присутствует в панели задач Windows, и "no", если не присутствует
singleInstance	Возвращает "yes", если пользователь может запустить только одну копию этого HTML-приложения, и "no", если он может запустить одновременно несколько копий
sysMenu	Возвращает "yes", если заголовок окна HTML-приложения содержит системное меню, и "no", если не содержит
version	Возвращает версию HTML-приложения в строковом виде
windowState	Возвращает в строковом виде начальное состояние окна HTML-приложения, которое оно приняло сразу же после открытия. Все доступные значения перечислены в табл. 15.5

Как правило, HTML-приложения распространяются так же, как обычные приложения Windows, — через Интернет в архивах или дистрибутивных файлах. Также возможно их распространение через Интернет и как обычных Web-страниц: HTML-приложение публикуется на Web-сервере и открывается щелчком по гиперссылке на другой Web-странице или набором интернет-адреса в строке адреса Web-обозревателя.

В качестве примера давайте создадим небольшое HTML-приложение, выполняющее преобразование размеров из дюймов в миллиметры. Его HTML-код приведен далее.

```
<HTML>
  <HEAD>
    <TITLE>Конвертор</TITLE>
    <HTA:APPLICATION ID="app" APPLICATIONNAME="Конвертор" BORDER="dialog"
    ICON="icon.ico" MAXIMIZEBUTTON="no" SCROLL="no" VERSION="1.0">
```

```
    </HTA:APPLICATION>
    <SCRIPT>
      function convert() {
        var inchesObj = document.getElementById("inches");
        var millimetersObj = document.getElementById("millimeters");
        if (!isNaN(inchesObj.value))
          millimetersObj.firstChild.nodeValue = inchesObj.value * 25.4;
      }

      function showAbout() {
        var appObj = document.getElementById("app");
        window.alert(appObj.applicationName + " " + appObj.version);
      }

      window.resizeTo(500, 350);
    </SCRIPT>
  </HEAD>
  <BODY>
    <H1>Конвертор</H1>
    <P>Преобразование размеров из дюймов в миллиметры.</P>
    <P>Введите в поле ввода размер в дюймах и нажмите кнопку
    &amp;quot;Преобразовать&amp;quot;.</P>
    <FORM>
      <P>Дюймы: <INPUT TYPE="text" ID="inches" NAME="inches"
      VALUE="0"></P>
      <P><INPUT TYPE="button" VALUE="Преобразовать"
      ONCLICK="convert();"></P>
    </FORM>
    <P ID="millimeters">0</P>
    <FORM>
      <P><INPUT TYPE="button" VALUE="О программе"
      ONCLICK="showAbout();">&amp;nbsp;<INPUT TYPE="button" VALUE="Выход"
      ONCLICK="window.close();"></P>
    </FORM>
  </BODY>
</HTML>
```

Мы задаем для окна нашего HTML-приложения толстую рамку без возможности изменения размеров окна. Также мы отключаем кнопку максимизации — это настоятельно рекомендуется в случае окон неизменяемых размеров. Кнопка минимизации, однако, будет присутствовать, поскольку мы ее не отключали.

Если бы мы задали для нашего приложения рамку с возможностью изменения размеров окна, мы бы также включили кнопку максимизации. Но в этом случае нам бы пришлось обрабатывать изменение размеров окна, чтобы изменить размеры содержимого приложения.

Также мы убрали полосы прокрутки, поскольку впоследствии собираемся задать такие размеры окна приложения, чтобы все содержимое поместилось в нем полностью. Это выполняется вызовом метода `resizeTo` объекта `Window` (подробнее см в *главе 7*); сами размеры окна мы подобрали опытным путем.

Содержимое нашего приложения и сценарий, собственно выполняющий преобразование величин, мы рассматривать не будем. Там нам все давно знакомо. Единственное, на что стоит обратить внимание, — наличие кнопок **О программе** и **Выход**. Первая кнопка выводит на экран окно-сообщение с названием и версией программы (эти данные берутся из тега `HTA:APPLICATION`), вторая закрывает приложение. Наличие этих кнопок — хороший стиль программирования.

> ***НА ЗАМЕТКУ***
>
> Firefox также предлагает набор средств для создания приложений, которые работают под его управлением. В первую очередь, это язык описания интерфейса приложений XUL и особое расширение JavaScript. Здесь они описаны не будут, так как, пожалуй, потребуют отдельной книги.

Вот, собственно, и все об HTML-приложениях. Да и о взаимодействии с посетителем с помощью специфических средств Web-обозревателей тоже.

Что дальше?

В этой главе мы познакомились со специфическими средствами Internet Explorer и Firefox, с помощью которых можно взаимодействовать с посетителем. Мы научились работать с произвольными фрагментами страницы, выделенным текстом, Буфером обмена Windows, реализовывать "настоящий" drag'n'drop с переносом данных, использовать диалоговые окна HTML и создавать HTML-приложения. При создании корпоративных Web-приложений, которые будут работать под Internet Explorer или Firefox, это может пригодиться.

Как уже было сказано, Internet Explorer явно прочили в платформу для разработки как раз таких, "заточенных" под него корпоративных Web-приложений. И "напихали" в него столько возможностей, что просто диву даешься… Как вам, например, прямая работа с простыми базами данных? Не верите? Читайте следующую главу!

Глава 14

Работа с базами данных (Internet Explorer)

Да, Internet Explorer позволяет напрямую работать с базами данных! В этой главе мы выясним, как это делается.

Хотя, надо сказать, от этой возможности толку не очень много. Во-первых, базы данных поддерживаются самые простые — текстовые (чуть позже мы выясним, что это такое). Во-вторых, Internet Explorer фактически позволяет только их просматривать — добавление, правка и удаление данных недоступны. Уже понятно, что сложной системы, позволяющей просматривать и редактировать данные из базы, мы создать средствами одного только Internet Explorer не сможем.

Но вполне возможно, что для какого-то решения и этих крайне ограниченных возможностей хватит. Тем более что для его реализации не придется устанавливать на компьютер клиента никакого дополнительного программного обеспечения — достаточно одного лишь Internet Explorer.

Средства Internet Explorer для работы с базами данных весьма просты и наглядны. И мы сейчас в этом убедимся. Но сначала выясним, что такое текстовая база данных.

Введение в базы данных

Базы данных используются для хранения данных, организованных каким-либо образом. Применяются они настолько часто, что трудно найти программу делового назначения, не работающую с базами данных. Поэтому хотя бы базовые понятия о них мы должны получить.

Что такое база данных

Как правило, базы данных содержат данные, организованные в виде таблицы (так называемые *реляционные базы данных*). Такая база данных содержит

набор столбцов и строк; каждая строка представляет одну единицу данных, содержащую несколько характеристик, а каждый столбец — одну характеристику, входящую в столбец. Столбцы таблицы называются *полями*, а строки — *записями*.

В качестве примера, чтобы лучше понять организацию базы данных, давайте взглянем на табл. 14.1. Это небольшой пример базы данных, содержащей список различных языков программирования, интерпретируемых и компилируемых.

Таблица 14.1. *Пример базы данных — список языков программирования*

Название	Категория
JavaScript	Интерпретируемый
VBScript	Интерпретируемый
Java	Компилируемый
C++	Компилируемый
Delphi	Компилируемый
Visual Basic	Компилируемый
C#	Компилируемый

Каждая строка этой таблицы представляет одну единицу данных, в нашем случае — один язык программирования. Каждый столбец представляет одну характеристику, входящую в единицу данных, в нашем случае это имя или категория языка программирования. Так, характеристики первой единицы данных — "JavaScript" и "Интерпретируемый", третьей — "Java" и "Компилируемый" и т. д.

Осталось только заменить терминологию: вместо "таблица" сказать "база данных", вместо "столбец" — "поле", а вместо "строка" — "запись". И говорить так в дальнейшем.

> ***На заметку***
>
> Надо сказать, что таблица и база данных — разные вещи. Серьезная база данных может содержать множество таблиц, связанных друг с другом. Но в простейшем случае (как у нас) можно утверждать, что база данных и таблица — это одно и то же.

Легко видеть, что база данных со списком языков программирования имеет два поля: "Название" и "Категория". Следовательно, каждая запись этой таблицы будет содержать эти два поля, не больше и не меньше, — набор полей жестко определен. Записей же в таблице может быть сколько угодно.

Каждое поле имеет уникальное имя, по которому программа, работающая с базой, сможет получить к ней доступ. Так, поля нашей таблицы имеют имена "Название" и "Категория".

> ***НА ЗАМЕТКУ***
>
> Здесь мы рассматриваем таблицу с полями, имеющими русские имена, — так нагляднее. На самом деле, многие базы данных не поддерживают символы кириллицы в именах полей.

Кроме имени, поле имеет жестко заданный *тип*. Он определяет тип данных, хранимых в этом поле. (Здесь напрашивается аналогия с типом данных в JavaScript — см. *главу 4*.) Большинство баз данных поддерживают следующие типы полей:

- *строковый* (его еще часто называют *текстовым*);
- *целочисленный* (может хранить только целые числа — этим он коренным образом отличается от числового типа JavaScript, который может представлять любые числа);
- *с плавающей точкой* (может хранить только нецелые числа);
- *дата* (хранит значения даты);
- *логический*.

Более развитые технологии баз данных поддерживают множество более сложных типов, но мы их рассматривать не будем.

На этом наш краткий курс баз данных можно считать законченным. По крайней мере, теперь мы сможем ориентироваться в терминологии.

Текстовая база данных

Текстовая база данных — самый простой формат хранения баз данных. Она представляет собой обычный текстовый файл, созданный в простейшем текстовом редакторе (например, Блокноте) и имеющий расширение txt или csv. Текстовый формат баз данных часто называют *CSV* (Comma Separated Values — значения, разделенные запятыми); отсюда и одно из доступных расширений файла такой базы — csv.

Данные в текстовой базе записываются согласно трем простым правилам.

1. Каждая запись записывается в отдельной строке.
2. Значения каждого поля записываются в строке, формирующей запись, и отделяются друг от друга особым *символом-разделителем*, по умолча-

нию — запятой. Если значение поля содержит символ-разделитель, оно должно быть заключено в кавычки.

3. Первая строка может содержать имена полей, также отделенные друг от друга символом-разделителем.

Имена полей могут содержать только латинские буквы и цифры, причем начинаться должны с латинской буквы. Кроме того, после имени поля, через двоеточие, может указываться тип поля в виде обозначения:

- "text" — текстовое;
- "int" — целочисленное;
- "float" — с плавающей точкой;
- "date" — дата;
- "boolean" — логическое.

Значения даты указываются в формате *<месяц>/<число>/<год>*, где `месяц` указывается в виде порядкового номера, от 0 (январь) до 11 (декабрь). Значение логического поля "yes" или 1 соответствует обычному `true`, значение "no" или 0 — `false`.

Давайте запишем базу данных, представленную в табл. 14.1, в текстовом виде.

```
name:text,cat:text
JavaScript,интерпретируемый
VBScript,интерпретируемый
Java,компилируемый
C++,компилируемый
Delphi,компилируемый
Visual Basic,компилируемый
C#,компилируемый
```

Полю, содержащему название языка программирования, мы дали имя `name`, а полю категории — `cat`. Оба этих поля имеют текстовый тип.

Сохраним эту базу данных в файле с именем 14.1.txt. Впоследствии мы используем ее в примерах.

Реализация работы с базами данных

Итак, необходимые знания мы получили. И даже создали базу данных для экспериментов. Настала пора рассмотреть, как же в Internet Explorer реализуется работа с данными, хранящимися в базе.

Загрузка базы данных

Первое, что нам необходимо сделать, — загрузить базу данных. Это выполняется с помощью особого элемента ActiveX, называемого Tabular Data Control, или *TDC*. (Об элементах ActiveX и способе размещения их на странице см. *главу 9*.) Он поставляется в составе Internet Explorer, так что нам не нужно устанавливать его дополнительно.

TDC помещается на страницу с помощью вот такого HTML-кода:

```
<OBJECT CLASSID="clsid:333C7BC4-460F-11D0-BC04-0080C7055A83"
↳ID="<имя>" STYLE="width: 0px; height: 0px">
  <PARAM NAME="DataURL" VALUE="<интернет-адрес файла базы данных>">
  <PARAM NAME="UseHeader"
  ↳VALUE="<считать первую запись списком имен полей>">
</OBJECT>
```

Тег `<OBJECT>`, который помещает элемент ActiveX на страницу, знаком нам по *главе 9*. Отметим только три момента.

Во-первых, запомним GUID, который соответствует TDC:

```
333C7BC4-460F-11D0-BC04-0080C7055A83
```

Во-вторых, мы обязательно должны задать имя тега `<OBJECT>`, иначе потом не сможем выполнить привязку элементов страницы к данным из базы (о привязке будет рассказано позже). Имя элемента страницы задается, как мы помним, атрибутом `ID`.

В-третьих, крайне желательно задать для TDC нулевые ширину и высоту, что можно сделать с помощью встроенного стиля (см. ранее). Это нужно, чтобы TDC не выводился на странице.

Обязательными параметрами элемента TDC являются `DateURL`, задающий интернет-адрес файла с базой данных, и `UseHeader`, указывающий, считать первую запись базы списком имен ее полей (значение "1") или нет (значение "0"). Остальные параметры являются необязательными. Полный же список параметров элемента TDC приведен в табл. 14.2.

Таблица 14.2. *Параметры элемента TDC*

Параметр	Описание
`CaseSensitive`	Если "1" (значение по умолчанию), при фильтрации и сортировке записей будет учитываться регистр символов, если "0", регистр учитываться не будет. О фильтрации и сортировке записей будет рассказано далее

Таблица 14.2 *(окончание)*

Параметр	Описание
`CharSet`	Задает кодировку текста для данных текстовой базы. По умолчанию "windows-1525" (символы, используемые в большинстве западноевропейских языков). Чтобы нормально отображались символы кириллицы, следует использовать кодировку "windows-1251" (кириллица). Также поддерживаются значения "utf-7" и "utf-8", соответствующие кодировкам UTF-7 и UTF-8. Этот параметр следует использовать только в случае проблем с выводом данных
`DataURL`	Интернет-адрес файла с базой данных
`EscapeChar`	Символ, которым в значениях полей таблицы следует предварить символы, являющиеся *служебными* (используемыми для особых целей; к их числу относится символ-разделитель). Чаще всего задают значение "обратный слэш" (\)
`FieldDelim`	Символ-разделитель, используемый для отделения значений полей друг от друга. По умолчанию — запятая
`Filter`	Критерий фильтрации записей. О фильтрации см. далее
`RowDelim`	Символ-разделитель, используемый для отделения записей друг от друга. По умолчанию — значение "newline", обозначающее символ перевода строки
`Sort`	Критерий сортировки записей. О сортировке см. далее
`TextQualifier`	Символ, в который заключается значение поля, если оно содержит служебные символы. По умолчанию — двойная кавычка
`UseHeader`	Если "1", то первая запись базы данных считается списком имен и типов полей. Если "0", первая запись базы считается обычными данными, а поля получают имена по умолчанию: `Column1`, `Column2` и т. д. Значения по умолчанию у этого параметра нет, поэтому его всегда следует указывать явно

HTML-код, выполняющий загрузку нашей базы данных, будет выглядеть так:

```
<OBJECT CLASSID="clsid:333C7BC4-460F-11D0-BC04-0080C7055A83"
↳ID="langsTDC" STYLE="width: 0px; height: 0px">
  <PARAM NAME="DataURL" VALUE="14.1.txt">
  <PARAM NAME="UseHeader" VALUE="1">
</OBJECT>
```

Обработав его, Internet Explorer загрузит базу данных 14.1.txt в память клиентского компьютера. После этого хранящиеся в ней данные будут доступны для всех элементов страницы, что к ним привязаны.

Привязка элементов страницы к данным

Второй шаг — привязка элементов страницы к базе данных. Это выполняется с помощью особых атрибутов HTML, которые мы сейчас рассмотрим.

Привязать к данным можно абсолютное большинство элементов страницы. Все они делятся на две неравные группы.

К первой группе относятся контейнеры, гиперссылки и элементы управления, в общем, все элементы страницы, способные выводить в данный момент только содержимое одного поля одной записи базы данных. Это так называемая *текущая запись*, с которой в данный момент выполняется работа. Она обозначается *указателем текущей записи*, который можно с помощью особых средств перемещать, делая текущими другие записи.

> ***Внимание!***
>
> Ранее было сказано, что к базе данных можно привязать элементы управления. С помощью этих элементов управления посетитель сможет править содержимое базы данных. Но в самом начале этой главы говорилось, что Internet Explorer не позволяет править данные в базе. Что из этого верно?
>
> Верно и то, и другое. Посетитель может править содержимое базы данных, но эти изменения будут присутствовать только в памяти клиентского компьютера. В сам файл базы данных они перенесены не будут.

Ко второй группе относится таблица HTML, которая может выводить содержимое сразу нескольких полей множества записей базы. В данном случае "множество" означает или все записи, или некоторое их количество, заданное в HTML-коде.

Сначала мы рассмотрим привязку к данным элементов страницы из первой группы. Это выполняется проще всего.

Прежде всего мы подключим к элементу страницы сам TDC. Это выполняется с помощью обязательного в данном случае атрибута `DATASRC`, поддерживаемого всеми тегами, которые можно подключать к данным. В качестве значения этого атрибута указывается имя нужного TDC, предваренного символом `#` ("решетка").

Далее нам нужно указать поле, из которого элемент страницы будет брать данные. Это выполняется с помощью обязательного в данном случае атрибута `DATAFLD`, который также поддерживается всеми тегами, которые можно подключать к данным. Имя нужного поля указывается в качестве значения этого атрибута.

Все, тег привязан к данным! Осталось только удалить его содержимое, иначе возможны проблемы.

```
<P><SPAN DATASRC="#langsTDC" DATAFLD="name"></SPAN></P>
```

Этот код создает контейнер, привязанный к полю `name` базы данных, загруженной с помощью TDC `langsTDC`. Этот контейнер помещен внутри абзаца, так как абзац мы не может привязать к данным непосредственно. Отметим, что, во-первых, имя TDC, указанное в атрибуте `DATASRC`, предварено знаком `#`, а во-вторых, контейнер не имеет содержимого.

Нам также может пригодиться необязательный атрибут `DATAFORMATAS`, позволяющий задать формат выводимых данных. Формат задается в виде его значения и может быть таким:

- "text" — обычный текст без всякой обработки;
- "html" — HTML-код, который будет предварительно обработан;
- "localized-text" — обычный текст, выводимый с учетом региональных настроек операционной системы.

Если атрибут `DATAFORMATAS` не указан, данные выводятся как обычный текст без всякой обработки. Как правило, этого достаточно, если только в выводимом поле не хранится HTML-код.

```
<P><SPAN DATASRC="#langsTDC" DATAFLD="name"
DATAFORMATAS="html"></SPAN></P>
```

Теперь в этом контейнере будет выводиться HTML-код, хранящийся в поле `name`.

Чтобы привязать к базе данных элементы страницы второй группы (то есть таблицу), потребуется больше телодвижений. Прежде всего, мы должны создать сам тег `<TABLE>` и в нем уже знакомым нам атрибутом `DATASRC` задать TDC, из которого будут браться данные.

```
<TABLE DATASRC="#langsTDC">
</TABLE>
```

После этого мы должны создать единственную строку с ячейками; каждая ячейка этой строки будет соответствовать выводимому на странице полю базы данных.

```
<TABLE DATASRC="#langsTDC">
  <TR>
    <TD></TD>
    <TD></TD>
  </TR>
</TABLE>
```

Осталось указать для каждой ячейки нужное поле базы данных. Но, к сожалению, теги `<TD>` и `<TH>` не поддерживают атрибут `DATAFLD`. Так что нам при-

дется поместить в эти ячейки блочные или встроенные контейнеры и уже для них указать поля.

```
<TABLE DATASRC="#langsTDC">
  <TR>
    <TD><DIV DATAFLD="name"></DIV></TD>
    <TD><DIV DATAFLD="cat"></DIV></TD>
  </TR>
</TABLE>
```

Готово — таблица привязана к базе данных. Теперь единственная ее строка будет выведена столько раз, сколько записей присутствует в нашей базе данных. Об этом позаботится сам Internet Explorer.

Если мы захотим создать у такой таблицы "шапку", то должны явно разбить ее на секции. Секция заголовка будет содержать саму "шапку", а секция тела — единственную строку, выводящую записи базы данных.

```
<TABLE DATASRC="#langsTDC">
  <THEAD>
    <TR>
      <TH>Название</TH>
      <TH>Категория</TH>
    </TR>
  </THEAD>
  <TBODY>
    <TR>
      <TD><DIV DATAFLD="name"></DIV></TD>
      <TD><DIV DATAFLD="cat"></DIV></TD>
    </TR>
  </TBODY>
</TABLE>
```

Тег `<TABLE>` поддерживает необязательный атрибут `DATAPAGESIZE`. Он указывает количество записей, которое должно выводиться в таблице, или *размер страницы* (*страницей* называется набор записей, выводимый в таблице в данный момент). Если он опущен, выводятся все записи.

```
<TABLE DATASRC="#langsTDC" DATAPAGESIZE="5">
  <TR>
    <TD><DIV DATAFLD="name"></DIV></TD>
    <TD><DIV DATAFLD="cat"></DIV></TD>
  </TR>
</TABLE>
```

Теперь наша таблица будет выводить за один раз только пять записей.

Для перемещения между страницами используются особые методы объекта `HTMLTableElement`. Мы рассмотрим их потом.

Осталось подытожить полученные знания, создав страницу, которая будет выводить все записи из базы данных 14.1.txt. Ее HTML-код приведен далее.

```
<HTML>
  <HEAD>
    <TITLE>Языки программирования</TITLE>
  </HEAD>
  <BODY>
    <OBJECT CLASSID="clsid:333C7BC4-460F-11D0-BC04-0080C7055A83"
    ID="langsTDC" STYLE="width: 0px; height: 0px">
      <PARAM NAME="DataURL" VALUE="14.1.txt">
      <PARAM NAME="UseHeader" VALUE="1">
    </OBJECT>
    <TABLE DATASRC="#langsTDC">
      <THEAD>
        <TR>
          <TH>Название</TH>
          <TH>Категория</TH>
        </TR>
      </THEAD>
      <TBODY>
        <TR>
          <TD><DIV DATAFLD="name"></DIV></TD>
          <TD><DIV DATAFLD="cat"></DIV></TD>
        </TR>
      </TBODY>
    </TABLE>
  </BODY>
</HTML>
```

Комментировать здесь абсолютно нечего — все нам уже знакомо.

Программная привязка элементов страницы к данным

Настала пора рассмотреть средства программной привязки элементов страницы к данным. Для этого объект `HTMLElement` и производные от него поддерживают набор свойств и методов, которые могут быть нам полезны.

Прежде всего, объект `HTMLElement` поддерживает свойства `dataFld`, `dataFormatAs` и `dataSrc`, соответствующие описанным выше атрибутам. А объект `HTMLTableElement` также поддерживает свойство `dataPageSize`, соответствующее одноименному атрибуту. Все эти свойства доступны как для записи, так и для чтения.

Изменение TDC и формата вывода значений поля всегда обрабатывается нормально. Но вот с изменением имени поля все несколько сложнее. Для элементов страницы, входящих в первую группу (о группах элементов страницы, привязываемых к данным, см. ранее), изменение поля обрабатывается нормально. Но этот номер не проходит для ячейки таблицы, точнее, контейнера, вложенного в нее и собственно выводящего значение поля. Если мы просто изменим для него имя поля, ничего не произойдет. Нам придется выполнить следующие шаги.

1. Отвязать от данных саму таблицу, для чего достаточно присвоить ее свойству `dataSrc` пустую строку.
2. Присвоить имя нового поля свойству `dataFld` контейнера.
3. Снова привязать таблицу к данным, присвоив ее свойству `dataSrc` имя того же TDC.

Но и здесь не все слава богу! Судя по всему, Internet Explorer 7 некорректно обрабатывает изменение поля для контейнера, вложенного в ячейку таблицы. Даже если мы изменим поле, в данном столбце таблицы будут выводиться значения из старого поля. Что это, очередная ошибка?

Поэтому если мы хотим иметь таблицу, привязанную к данным, и программно менять поля, из которых выводятся данные в этой таблице, то должны использовать последний довод Web-программистов. А именно — удалить эту таблицу со страницы и сформировать ее заново, уже с новыми параметрами. Это можно сделать, воспользовавшись свойством `innerHTML` объекта `HTMLElement`, поддерживаемым Internet Explorer, или методами DOM. Все это было подробно описано в *главе 8*.

Объект `HTMLTableElement` поддерживает также четыре метода, о которых нам обязательно нужно знать, если мы ограничили количество отображаемых в таблице записей (задали атрибут `DATAPAGESIZE`). Эти методы перечислены далее.

- `firstPage` — выводит на экран самую первую страницу базы данных (о страницах было рассказано ранее).
- `lastPage` — выводит на экран самую последнюю страницу базы данных.
- `previousPage` — выводит на экран предыдущую страницу базы данных.
- `nextPage` — выводит на экран следующую страницу базы данных.

Все эти методы не принимают параметров и не возвращают результата.

Давайте для примера создадим страницу с таблицей, которая будет выводить в данный момент содержимое только одного поля базы данных 14.1.txt — `name` или `cat`. Изначально в ней будет выводиться содержимое поля `name`, а для переключения между полями мы используем небольшую форму с раскрывающимся списком.

HTML-код этой страницы приведен далее.

```
<HTML>
  <HEAD>
    <TITLE>Языки программирования</TITLE>
    <SCRIPT>
      function langChange() {
        var contObj = document.getElementById("cont");
        contObj.innerHTML = "";
        var langObj = document.getElementById("lang");
        var optObj = langObj.options[langObj.selectedIndex];
        var s = "<TABLE ID='tbl' DATASRC='#langsTDC'>";
        s += "<THEAD>";
        s += "<TR>"
        s += "<TH>" + optObj.text + "</TH>";
        s += "</TR>"
        s += "</THEAD>";
        s += "<TBODY>";
        s += "<TR>"
        s += "<TD><DIV DATAFLD='" + optObj.value + "'></DIV></TD>";
        s += "</TR>"
        s += "</TBODY>";
        s += "</TABLE>";
        contObj.innerHTML = s;
      }
    </SCRIPT>
  </HEAD>
  <BODY>
    <OBJECT CLASSID="clsid:333C7BC4-460F-11D0-BC04-0080C7055A83"
    ID="langsTDC" STYLE="width: 0px; height: 0px">
      <PARAM NAME="DataURL" VALUE="14.1.txt">
      <PARAM NAME="UseHeader" VALUE="1">
    </OBJECT>
```

```
    <FORM ACTION="#">
      <SELECT ID="lang" NAME="lang" ONCHANGE="langChange();">
        <OPTION VALUE="name" SELECTED>Название</OPTION>
        <OPTION VALUE="cat">Категория</OPTION>
      </SELECT>
    </FORM>
    <DIV ID="cont"></DIV>
    <SCRIPT>
      langChange();
    </SCRIPT>
  </BODY>
</HTML>
```

Сама таблица с данными выводится в блочном контейнере `cont`. Это нужно, чтобы упростить код: обратившись к этому контейнеру и использовав свойство `innerHTML`, мы можем поместить в него любой код HTML, который будет тотчас обработан Internet Explorer и выведен на экран.

Для выбора поля базы данных мы используем, как говорилось ранее, раскрывающийся список `lang` и обрабатываем его событие `onChange`, возникающее при выборе пункта в списке. Функция — обработчик этого события `langChange` выполняет удаление старой таблицы и вывод новой.

Сама функция `langChange` сначала удаляет старую таблицу, присвоив свойству `innerHTML` контейнера `cont` пустую строку. После этого она формирует строку, содержащую HTML-код новой таблицы, и присваивает ее этому же свойству. И новая таблица тотчас выводится на страницу.

Осталось только сказать, что при загрузке страницы на экран выводится таблица, содержащая значения из поля `name` базы данных. Это выполняется в два этапа. Сначала мы делаем первый пункт списка `lang` изначально выбранным, вставив в формирующий его тег `<OPTION>` атрибут без значения `SELECTED`. Потом мы в особом загрузочном сценарии (выполняющемся при загрузке страницы) вызываем функцию `langChange`. Теперь сразу после загрузки страницы посетитель может просмотреть названия языков программирования.

Ну что, рассмотрим еще один пример? Пусть теперь это будет таблица, выводящая содержимое все той же многострадальной базы данных 14.1.txt, но теперь только по пять записей. То есть мы зададим размер страницы, равный пяти записям, и теперь будем выводить содержимое базы данных постранично. А для "листания" таблицы от страницы к странице используем набор гиперссылок.

HTML-код страницы с "листанием" приведен далее.

```
<HTML>
  <HEAD>
    <TITLE>Языки программирования</TITLE>
  </HEAD>
  <BODY>
    <OBJECT CLASSID="clsid:333C7BC4-460F-11D0-BC04-0080C7055A83"
    ID="langsTDC" STYLE="width: 0px; height: 0px">
      <PARAM NAME="DataURL" VALUE="14.1.txt">
      <PARAM NAME="UseHeader" VALUE="1">
    </OBJECT>
    <TABLE ID="tbl" DATASRC="#langsTDC" DATAPAGESIZE="5">
      <THEAD>
        <TR>
          <TH>Название</TH>
          <TH>Категория</TH>
        </TR>
      </THEAD>
      <TBODY>
        <TR>
          <TD><DIV DATAFLD="name"></DIV></TD>
          <TD><DIV DATAFLD="cat"></DIV></TD>
        </TR>
      </TBODY>
    </TABLE>
    <P><A HREF="#" ONCLICK="tblObj.firstPage();">&lt;&lt;</A>
    <A HREF="#" ONCLICK="tblObj.previousPage();">&lt;</A>
    <A HREF="#" ONCLICK="tblObj.nextPage();">&gt;</A>
    <A HREF="#" ONCLICK="tblObj.lastPage();">&gt;&gt;</A></P>
    <SCRIPT>
      var tblObj = document.getElementById("tbl");
    </SCRIPT>
  </BODY>
</HTML>
```

Комментировать здесь совершенно нечего.

Средства управления TDC из сценариев

Конечно, программная привязка элементов страницы к данным — полезная вещь. Но куда полезнее возможность программно управлять самим TDC.

Прежде всего нужно сказать, что сам TDC поддерживает набор свойств, соответствующих перечисленным в табл. 14.2 параметрам. Эти свойства имеют те же имена, что и соответствующие им параметры. Единственное — вместо строковых значений "1" и "0" для задания логических величин следует применять привычные нам `true` и `false`.

```
<OBJECT CLASSID="clsid:333C7BC4-460F-11D0-BC04-0080C7055A83"
ID="langsTDC" STYLE="width: 0px; height: 0px">
  <PARAM NAME="DataURL" VALUE="14.1.txt">
  <PARAM NAME="UseHeader" VALUE="1">
</OBJECT>
. . .
<SCRIPT>
  var langsObj = document.getElementById("langsTDC");
  langsObj.DataURL = "14.2.txt";
</SCRIPT>
```

Этот сценарий загружает с помощью TDC `langsTDC` базу данных 14.2.txt.

Кроме того, TDC поддерживает свойство `recordset`. Это свойство возвращает экземпляр особого объекта `Recordset`, представляющий *набор записей* — своего рода программное представление базы данных со всеми ее полями и записями. Именно этот объект для нас наиболее полезен.

Прежде всего, для нас представляют интерес четыре метода, выполняющие перемещение на другую запись, которая после перемещения станет текущей (что такое текущая запись, описано ранее). Так что если мы собирается в данный момент выводить на странице содержимое только одной записи, эти методы нам очень пригодятся.

- `MoveFirst` — перемещает указатель на самую первую запись базы данных, делая ее текущей.
- `MoveLast` — перемещает указатель на самую последнюю запись базы данных, делая ее текущей.
- `MovePrevious` — перемещает указатель на предыдущую запись базы данных, делая ее текущей.
- `MoveNext` — перемещает указатель на следующую запись базы данных, делая ее текущей.

Эти методы не принимают параметров и не возвращают результата.

Также нам будут полезны свойства `BOF` и `EOF`. Первое возвращает `true`, если текущей в данный момент является самая первая запись базы данных, и `false` в противном случае. Второе возвращает `true`, если текущей в данный момент является самая последняя запись базы данных, и `false` в противном случае.

ВНИМАНИЕ!

Если мы находимся на первой записи базы данных и вызовем метод `MovePrevious`, то получим сообщение об ошибке. То же самое будет, если переместиться на последнюю запись базы и вызвать метод `MoveNext`. Так что при написании сценариев, манипулирующих TDC, перед собственно перемещением на другую запись всегда следует выполнять проверку значений свойств `BOF` и `EOF`.

```
if (!(langsObj.recordset.EOF)) langsObj.recordset.MoveNext();
```

Это выражение проверяет, не является ли текущей последняя запись базы данных, и, если нет, выполняет переход на следующую запись.

Свойство `Fields` возвращает одноименную коллекцию, содержащую все поля базы данных. Эта коллекция является экземпляром объекта, который называется так же — `Fields`. Для доступа к нужному полю мы можем использовать имя этого поля в качестве строкового индекса. Отметим только, что вместо квадратных скобок, как в других знакомых нам коллекциях, следует использовать круглые.

```
var nameFldObj = langsObj.recordset.Fields("name");
```

Это выражение помещает в переменную `nameFldObj` экземпляр объекта `Field`, представляющий поле базы данных, в нашем случае — поле `name`. Не забываем, что в этом случае следует использовать круглые скобки вместо квадратных.

Объект `Field` поддерживает полезное для нас свойство `Value`. С помощью этого свойства мы можем получить значение данного поля текущей записи или задать его.

```
var name = nameFldObj.Value;
```

Это выражение помещает в переменную `name` значение полученного ранее одноименного поля текущей записи.

Кроме того, нам могут быть полезны еще два свойства объекта `Recordset`. Свойство `AbsolutePosition` возвращает номер записи, являющейся в данный момент текущей. А свойство `RecordCount` возвращает общее количество записей в базе данных.

НА ЗАМЕТКУ

Полное описание объектов, применяемых для работы с базами данных, можно найти на посвященной им странице MSDN по адресу **http://msdn2.microsoft.com/en-us/library/ms675532(VS.85).aspx**.

Кроме того, TDC поддерживает несколько специфических событий, перечисленных в табл. 14.3.

Таблица 14.3. *События TDC*

Событие	Описание
`onDataAvailable`	Возникает всякий раз, когда TDC готов предоставить новые данные из базы. Всплывает, поведение по умолчанию как таковое отсутствует
`onDatasetChanged`	Возникает, когда параметры TDC изменяются, например, после задания критериев фильтрации или сортировки (о фильтрации и сортировке будет рассказано далее). Всплывает, поведение по умолчанию как таковое отсутствует
`onDatasetComplete`	Возникает после того, как TDC получит всю базу данных целиком. Всплывает, поведение по умолчанию как таковое отсутствует
`onRowEnter`	Возникает после перехода на текущую запись базы данных. Всплывает, поведение по умолчанию как таковое отсутствует
`onRowExit`	Возникает перед переходом с текущей записи на другую. Всплывает, поведение по умолчанию (собственно переход на другую запись) может быть отменено

Эти события можно использовать по-разному. Например, событие `onRowEnter` можно использовать для вывода информации, которая не хранится в базе данных, а рассчитывается на основе ее содержимого.

Кстати, если уж мы собираемся обрабатывать перечисленные в табл. 14.3 события, то для нас может быть интересно свойство `recordset` объекта `Event`, то есть самого события. Оно возвращает набор записей (экземпляр объекта `Recordset`), соответствующий TDC, в котором возникло данное событие. Это позволит нам быстро выполнить любой из перечисленных методов объекта `Recordset` или обратиться к его свойствам, не добираясь до набора записей через TDC.

Пример, который мы здесь рассмотрим, будет весьма сложным. Более того, он будет использовать другую базу данных, содержимое которой представлено далее.

```
name:text,cat:int
JavaScript,0
VBScript,0
Java,1
C++,1
```

```
Delphi,1
Visual Basic,1
C#,1
```

Здесь мы изменили тип поля `cat` на целочисленный. Число 0 обозначает интерпретируемый язык, а число 1 — компилируемый.

Сохраним исправленную базу данных в файле под именем 14.2.txt. И создадим страницу, которая будет выводить ее на экран.

Страница эта будет делать следующее:

- выводить одновременно содержимое только одной записи;
- иметь возможность перемещения от записи к записи. Это естественно — иначе посетитель не сможет ее полностью просмотреть;
- для компилируемого языка на странице будет выводиться слово "Компилируемый", отсутствующее в случае интерпретируемого языка.

HTML-код этой страницы приведен далее.

```
<HTML>
  <HEAD>
    <TITLE>Языки программирования</TITLE>
    <SCRIPT>
      function langsRowEnter() {
        isCompObj.style.visibility = (recObj.Fields('cat').Value == 1) ?
        'visible' : 'hidden';
      }
    </SCRIPT>
  </HEAD>
  <BODY>
    <OBJECT CLASSID="clsid:333C7BC4-460F-11D0-BC04-0080C7055A83"
    ID="langsTDC" STYLE="width: 0px; height: 0px"
    ONROWENTER="langsRowEnter();">
      <PARAM NAME="DataURL" VALUE="14.2.txt">
      <PARAM NAME="UseHeader" VALUE="1">
    </OBJECT>
    <P>Язык программирования: <SPAN DATASRC="#langsTDC"
    DATAFLD="name"></SPAN></P>
    <P ID="isComp">Компилируемый</P>
    <P><A HREF="#" ONCLICK="recObj.MoveFirst();">&lt;&lt;</A>
    <A HREF="#"
    ONCLICK="if (!(recObj.BOF)) recObj.MovePrevious();">&lt;</A>
```

```
    <A HREF="#"
    ONCLICK="if (!(recObj.EOF)) recObj.MoveNext();">&gt;</A>
    <A HREF="#" ONCLICK="recObj.MoveLast();">&gt;&gt;</A></P>
    <SCRIPT>
      var langsObj = document.getElementById("langsTDC");
      var recObj = langsObj.recordset;
      var isCompObj = document.getElementById("isComp");
    </SCRIPT>
  </BODY>
</HTML>
```

Здесь для вывода или скрытия слова "Компилируемый" мы используем обработчик события `onRowEnter` TDC. Он сравнивает значение поля `cat` базы данных с единицей и, если равенство верно, выводит данное слово, в противном случае скрывает его. Больше ничего здесь пояснений не требует — все нам уже знакомо.

Фильтрация и сортировка записей средствами TDC

Напоследок рассмотрим возможности TDC по фильтрации и сортировке записей. *Фильтрацией* разработчики баз данных традиционно называют отбор записей по определенным условиям (*критериям*), например, равенства или неравенства значения поля заданному значению. Ну, а с сортировкой все ясно.

Критерий фильтрации задается с помощью свойства или параметра `Filter` TDC (см. табл. 14.2). Ему присваивается строка, содержащая условное выражение (об условных выражениях применительно к JavaScript было рассказано в *главе 4*), которое и является критерием.

```
langsObj.Filter = "cat=0";
```

Это выражение задает критерий фильтрации, отбирающий из базы данных 14.2.txt только интерпретируемые языки (значение поля `cat` равно 0).

В табл. 14.4 перечислены все операторы сравнения и логические операторы, которые мы можем использовать в выражениях, задающих критерии фильтрации.

> ***Внимание!***
>
> Заметим, что логический оператор равенства в критериях фильтрации обозначается одним символом =, а не двумя, как в JavaScript.

Таблица 14.4. *Операторы сравнения и логические операторы, используемые для задания критериев фильтрации записей*

Оператор	Описание
<	Меньше
>	Больше
=	Равно
<=	Меньше или равно
>=	Больше или равно
<>	Не равно
&	Логическое И
|	Логическое ИЛИ

Чтобы вообще отключить фильтрацию, достаточно присвоить свойству `Filter` TDC пустую строку.

Для задания критерия сортировки предназначены свойство и параметр `Sort` TDC (см. табл. 14.2). Этому свойству присваивается строка, содержащая список имен полей, по которым должна выполняться сортировка, разделенных точкой с запятой. Например:

```
langsObj.Sort = "cat;name";
```

После выполнения этого выражения база данных будет отсортирована по полю `cat`. Записи же, содержащие одинаковые значения поля `cat`, будут дополнительно отсортированы по полю `name`.

По умолчанию сортировка выполняется по возрастанию значения заданного поля. Чтобы задать сортировку по убыванию, нужно перед именем поля поставить знак "минус":

```
langsObj.Sort = "cat;-name";
```

В этом случае сортировка по полю `name` будет выполняться по убыванию значения этого поля.

Опять же, чтобы вообще отключить сортировку, достаточно присвоить свойству `Sort` TDC пустую строку.

После задания критериев фильтрации и (или) сортировки обязательно следует вызвать метод `Reset` TDC, который выполнит обновление всех элементов страницы, привязанных к данным. Этот метод не принимает параметров и не возвращает значения.

В качестве примера давайте рассмотрим страницу, выводящую список языков программирования по категориям: только интерпретируемые или только

компилируемые. Для выбора категории мы предусмотрим раскрывающийся список. Дополнительно мы отсортируем список языков по их названию.

```
<HTML>
  <HEAD>
    <TITLE>Языки программирования</TITLE>
    <SCRIPT>
      function catChange() {
        var catObj = document.getElementById("cat");
        var optObj = catObj.options[catObj.selectedIndex];
        var langsObj = document.getElementById("langsTDC");
        langsObj.Filter = "cat=" + optObj.value;
        langsObj.Reset();
      }
    </SCRIPT>
  </HEAD>
  <BODY>
    <OBJECT CLASSID="clsid:333C7BC4-460F-11D0-BC04-0080C7055A83"
    ID="langsTDC" STYLE="width: 0px; height: 0px">
      <PARAM NAME="DataURL" VALUE="14.2.txt">
      <PARAM NAME="UseHeader" VALUE="1">
      <PARAM NAME="Filter" VALUE="cat=0">
      <PARAM NAME="Sort" VALUE="name">
    </OBJECT>
    <FORM ACTION="#">
      <SELECT ID="cat" NAME="cat" ONCHANGE="catChange();">
        <OPTION VALUE="0" SELECTED>Интерпретируемые</OPTION>
        <OPTION VALUE="1">Компилируемые</OPTION>
      </SELECT>
    </FORM>
    <TABLE DATASRC="#langsTDC">
      <THEAD>
        <TR>
          <TH>Название</TH>
        </TR>
      </THEAD>
      <TBODY>
        <TR>
          <TD><DIV DATAFLD="name"></DIV></TD>
```

```
        </TR>
      </TBODY>
    </TABLE>
  </BODY>
</HTML>
```

Здесь мы выводим только названия языков программирования — поскольку посетитель сам задает их категорию, она ему уже известна.

Для отслеживания выбора пункта в раскрывающемся списке `cat`, где задается категория языка программирования, мы обрабатываем событие `onChange` этого списка. Функция `catChange` — обработчик этого события формирует строку с новым критерием фильтрации и присваивает ее свойству `Filter` TDC. После этого следует вызов метода `Reset` TDC, чтобы он обновил данные согласно заданному критерию фильтрации.

Вот и все о работе с базами данных в Internet Explorer. Слишком многого от них ждать не приходится, но, кто знает, может, они пригодятся нам когда-нибудь.

> ***НА ЗАМЕТКУ***
>
> Internet Explorer также поддерживает более развитые средства для работы с настоящими базами данных, позволяющими также, в частности, править данные. Но, насколько удалось выяснить автору, они требуют дополнительного программного обеспечения, в частности, Internet Information Services. Описание этих средств можно найти на сайте MSDN (**http://msdn2.microsoft.com/en-us/library/default.aspx**). К сожалению, оно разбросано там по разным страницам, которые придется поискать.

Что дальше?

Internet Explorer велик и могуч! Он даже позволяет работать с базами данных (правда, с самыми простыми и без возможности правки данных), в чем мы и убедились в этой главе.

В следующей главе мы продолжим изучать богатые возможности Internet Explorer, на этот раз — в плане создания дополнительных "красивостей" на странице. Это так называемые фильтры и преобразования. Элементы страницы после применения к ним таких "красивостей" выглядят просто потрясающе!

Глава 15

Фильтры и преобразования (Internet Explorer)

Как платформа для создания корпоративных решений Internet Explorer, может, и не очень котируется. Но как средство создания впечатляющих графических эффектов он вполне даже ничего! И скоро мы в этом убедимся.

Эта глава будет посвящена использованию фильтров и преобразований Internet Explorer. Именно с их помощью создаются всевозможные "красивости", которые зачастую можно встретить в Интернете: полупрозрачные элементы страницы, элементы страницы с тенью, элементы страницы, плавно появляющиеся и исчезающие и мн. др. И все это делается только средствами HTML и CSS плюс немного JavaScript, без использования графики!

Заинтригованы? Что ж, читайте дальше.

Фильтры

Начнем мы, пожалуй, с фильтров. Их создавать проще всего — не требуется никакого программирования.

Создание фильтров

Фильтр Internet Explorer — это особый графический эффект, применяемый к элементу страницы, причем эффект статический, неподвижный. Это может быть полупрозрачность, тень, размытие, негативное изображение и др.

Фильтр можно применить практически к любому элементу страницы, в том числе и к секции ее тела. Единственное — некоторые элементы страницы требуют явного задания размеров (это можно сделать с помощью атрибутов стиля `width` и `height`), а другие должны быть свободно позиционированы (о свободно позиционируемых элементах рассказывалось в *главе 10*). К элементам страницы, к которым никак нельзя применить фильтр, относятся секции

и строки таблицы, списки и их пункты и модули расширения Web-обозревателя (описаны в *главе 9*).

Применить к элементу страницы фильтр можно с помощью особого атрибута стиля `filter`. Его значение должно быть записано в таком формате:

```
progid:DXImageTransform.Microsoft.<имя фильтра>
([<список свойств фильтра и их значений>])
```

`Имя фильтра` однозначно указывает на фильтр, который мы применяем к элементу страницы.

Необязательные `свойства` задают значения дополнительных параметров фильтра. Список свойств записывается в виде набора пар `<свойство>=<значение>`, разделенных запятыми, после имени фильтра в круглых скобках. Если список свойств отсутствует, то скобки ставить не нужно.

```
<IMG SRC="image.gif" STYLE="height: 100; width: 100; filter:
progid:DXImageTransform.Microsoft.MotionBlur(Strength=50)">
```

Этот HTML-код создает на странице графическое изображение и применяет к нему фильтр "размытие". Этот фильтр имеет имя `MotionBlur` и поддерживает свойство `Strength`, задающее дистанцию размытия и имеющее значение 50 пикселов.

Мы можем применить к одному элементу страницы сразу несколько фильтров. Для этого определения этих фильтров записываются в значении атрибута `filter` и отделяются друг от друга пробелами.

```
<IMG SRC="image.gif" STYLE="height: 100; width: 100; filter:
progid:DXImageTransform.Microsoft.MotionBlur(Strength=50)
DXImageTransform.Microsoft.DropShadow">
```

Этот HTML-код создает на странице графическое изображение и применяет к нему одновременно фильтры "размытие" и "тень" (`DropShadow`). Обратим внимание, что для последнего фильтра мы не указали никаких свойств и, следовательно, не поставили после его имени скобки.

Вот и все. Больше ничего для создания фильтра от нас не требуется.

Осталось только привести список всех фильтров (табл. 15.1), поддерживаемых Internet Explorer, и их свойств (табл. 15.2).

Таблица 15.1. *Фильтры, поддерживаемые Internet Explorer*

Фильтр	Описание
`Alpha`	Делает элемент страницы прозрачным
`AlphaImageLoader`	"Подкладывает" графическое изображение под содержимое элемента страницы

Таблица 15.1 *(окончание)*

Фильтр	Описание
`BasicImage`	Делает элемент страницы черно-белым, как бы просвеченным рентгеновскими лучами, и поворачивает его. Отдельно можно задавать угол поворота, степень "просвеченности" и т. п.
`Blur`	Делает элемент страницы размытым
`Chroma`	Делает заданный цвет элемента страницы прозрачным
`Compositor`	Объединяет цвета двух элементов страницы и выводит результат объединения
`DropShadow`	Добавляет к элементу страницы тень заданного цвета, причем отображаться она будет отдельно от самого элемента
`Emboss`	Отображает элемент страницы выпуклым
`Engrave`	Отображает элемент страницы вдавленным
`FlipH`	Выполняет зеркальное отражение элемента страницы по горизонтали
`FlipV`	Выполняет зеркальное отражение элемента страницы по вертикали
`Glow`	Отображает элемент страницы так, что он кажется тлеющим
`Gradient`	Градиентно закрашивает элемент страницы
`Light`	Отображает элемент страницы так, что он кажется освещенным
`MaskFilter`	Отображает прозрачный цвет элемента страницы цветом, заданным свойством `Color`, а все непрозрачные цвета делает прозрачными
`Matrix`	Изменяет размеры элемента страницы, поворачивает или инвертирует его, используя матричные преобразования
`MotionBlur`	Делает элемент страницы размытым, словно быстро движущимся в заданном направлении
`Pixelate`	Разбивает элемент страницы на отдельные пикселы
`Shadow`	Добавляет к элементу страницы тень заданного цвета, "отбрасываемую" в заданном направлении
`Wave`	Создает волнистое искажение элемента страницы

Таблица 15.2. *Свойства фильтров*

Свойство	Фильтр	Описание
`Add`	`MotionBlur`, `Wave`	Если `true`, исходное содержимое элемента страницы перекрывает новое, если `false` (значение по умолчанию) — наоборот

Таблица 15.2 *(продолжение)*

Свойство	Фильтр	Описание
`Bias`	`Emboss`, `Engrave`	Процентное значение, добавляемое к цвету элемента для получения цвета затенения. Значение по умолчанию — 0.7
`Color`	`DropShadow`, `Glow`, `MaskFilter`, `Shadow`	Цвет. Задается в RGB-формате как строка (например, #FFFFFF). Значения по умолчанию различаются у разных фильтров
`Direction`	`MotionBlur`, `Shadow`	Направление. Задается в градусах и должно быть кратно 45. Значения по умолчанию различаются у разных фильтров
`Enabled`	Все фильтры	Если `true` (значение по умолчанию), то фильтр применяется к элементу страницы, если `false` — не применяется
`EndColor`	`Gradient`	Конечный цвет градиентной закраски. Задается в RGB-формате как число (например, 0xFFFFFF). По умолчанию — черный
`EndColorStr`	`Gradient`	Конечный цвет градиентной закраски. Задается в RGB-формате как строка (например, #FFFFFF). По умолчанию — черный
`FinishOpacity`	`Alpha`	Конечный уровень градиентной прозрачности; задается в виде числа от 0 (полная прозрачность; значение по умолчанию) до 100 (полная непрозрачность). Значение по умолчанию — 0
`FinishX`, `FinishY`	`Alpha`	Горизонтальная и вертикальная координаты позиции, в которой заканчивается область градиентной прозрачности, в пикселах. Значения по умолчанию — 0
`Freq`	`Wave`	Количество "волн". Значение по умолчанию — 3
`Function`	`Compositor`	Номер функции преобразования. Список всех возможных значений приведен в табл. 15.3. Значение по умолчанию — 0
`GradientType`	`Gradient`	Если 1 (значение по умолчанию), градиентная заливка располагается по горизонтали, если 0 — по вертикали
`GrayScale`	`BasicImage`	Если 1, элемент страницы отображается черно-белым, если 0 (значение по умолчанию) — цветным

Таблица 15.2 *(продолжение)*

Свойство	Фильтр	Описание
`Invert`	`BasicImage`	Если 1, элемент страницы отображается с инвертированными цветами, если 0 (значение по умолчанию) — как обычно
`LightStrength`	`Wave`	Постоянство окраски "волн". Может быть от 0 до 100 (значение по умолчанию)
`MakeShadow`	`Blur`	Если `true`, элемент страницы будет затенен, если `false` (значение по умолчанию) — не будет
`Mask`	`BasicImage`	Если 1, прозрачный цвет элемента страницы будет заменен значением свойства `MaskColor`, если 0 (значение по умолчанию) — не будет
`MaskColor`	`BasicImage`	Цвет, на который будет заменен прозрачный цвет элемента страницы. Задается в RGB-формате в виде числа (например, 0xFFFFFF). По умолчанию — черный
`Mirror`	`BasicImage`	Если 1, элемент страницы будет отображен зеркально, если 0 (значение по умолчанию) — как обычно
`OffX`, `OffY`	`DropShadow`	Горизонтальное и вертикальное смещения тени в пикселах. Значение по умолчанию — 5
`Opacity`	`Alpha`	Начальный уровень градиентной прозрачности; задается в виде числа от 0 (полная прозрачность; значение по умолчанию) до 100 (полная непрозрачность). Значение по умолчанию — 0
`Opacity`	`BasicImage`	Уровень прозрачности элемента страницы. Может быть от 0.0 до 1.0 (значение по умолчанию)
`Phase`	`Wave`	Фаза "волн". Может быть от 0 (значение по умолчанию) до 100
`PixelRadius`	`Blur`	Размер области "размытия". Может быть от 1.0 до 100.0. Значение по умолчанию — 2.0
`Positive`	`DropShadow`	Если `true` (значение по умолчанию), тень создается из прозрачных пикселов элемента страницы, если `false` — из непрозрачных ("негативная" тень)

Таблица 15.2 *(окончание)*

Свойство	Фильтр	Описание
`Rotation`	`BasicImage`	Угол поворота элемента страницы. Доступны значения: 0 (значение по умолчанию) — нет поворота, 1 — на 90º, 2 — на 180º и 3 — на 270º
`ShadowOpacity`	`Blur`	Прозрачность тени. Может быть от 0.0 (полностью прозрачная) до 1.0 (полностью непрозрачная). Значение по умолчанию — 0.75
`sizingMethod`	`AlphaImageLoader`	Способ размещения изображения в границах элемента страницы. Доступны значения: "crop" (обрезание изображения), "image" (уменьшение или увеличение самого элемента страницы; значение по умолчанию) и "scale" (уменьшение или увеличение изображения)
`src`	`AlphaImageLoader`	Интернет-адрес файла изображения
`StartColor`	`Gradient`	Начальный цвет градиентной закраски. Задается в RGB-формате как число (например, 0xFFFFFF). Значение по умолчанию отсутствует
`StartColorStr`	`Gradient`	Начальный цвет градиентной закраски. Задается в RGB-формате как строка (например, #FFFFFF). Значение по умолчанию отсутствует
`StartX`, `StartY`	`Alpha`	Горизонтальная и вертикальная координаты позиции, в которой начинается область градиентной прозрачности, в пикселах. Значение по умолчанию — 0
`Strength`	`MotionBlur`, `Glow`, `Wave`	Дистанция в пикселах, на которой выполняется преобразование. Значение по умолчанию зависит от фильтра
`Style`	`Alpha`	Форма градиентной прозрачности. Доступны значения: 0 (нет градиента; значение по умолчанию), 1 (линейный градиент), 2 (круговой градиент) и 3 (прямоугольный градиент)
`XRay`	`BasicImage`	Если 1, то элемент страницы будет отображен "просвеченным" "рентгеновскими лучами", если 0 (значение по умолчанию) — как обычно

Таблица 15.3. *Доступные значения свойства* `Function` *фильтра* `Compositor`

Значение	Описание
0	Никакая операция не выполняется. Значение по умолчанию
1	Сравнивает накладывающиеся пикселы обоих изображений и выводит пиксел с меньшей яркостью
2	Сравнивает накладывающиеся пикселы обоих изображений и выводит пиксел с большей яркостью
3	Выводит только первое изображение
4	Выводит первое изображение поверх второго. Второе изображение будет видно сквозь прозрачные участки первого
5	Выводит те пикселы первого изображения, которым соответствуют непрозрачные пикселы второго
6	Выводит те пикселы первого изображения, которым соответствуют прозрачные пикселы второго
7	Выводит первое изображение, чьи пикселы имеют такую же прозрачность, что и соответствующие пикселы второго изображения
8	Сначала вычитает цвет каждого пиксела второго изображения из цвета соответствующего пиксела первого. После этого задает для каждого пиксела результирующего изображения прозрачность соответствующих пикселов второго изображения. Выводит результирующее изображение
9	Сначала складывает цвет каждого пиксела первого изображения с цветом соответствующего пиксела второго. После этого задает для каждого пиксела результирующего изображения прозрачность соответствующих пикселов второго изображения. Выводит результирующее изображение
10	Выполняет операцию "исключающее ИЛИ" (XOR) с пикселами обоих изображений и выводит результат
19	Выводит только второе изображение
20	Выводит второе изображение поверх первого. Первое изображение будет видно сквозь прозрачные участки второго
21	Выводит те пикселы второго изображения, которым соответствуют непрозрачные пикселы первого
22	Выводит те пикселы второго изображения, которым соответствуют прозрачные пикселы первого
23	Выводит второе изображение, чьи пикселы имеют такую же прозрачность, что и соответствующие пикселы первого изображения

Таблица 15.3 *(окончание)*

Значение	Описание
24	Сначала вычитает цвет каждого пиксела первого изображения из цвета соответствующего пиксела второго. После этого задает для каждого пиксела результирующего изображения прозрачность соответствующих пикселов первого изображения. Выводит результирующее изображение
25	Сначала складывает цвет каждого пиксела второго изображения с цветом соответствующего пиксела первого. После этого задает для каждого пиксела результирующего изображения прозрачность соответствующих пикселов первого изображения. Выводит результирующее изображение

Для примера давайте создадим страницу с абзацем и применим к этому абзацу сразу два фильтра: полупрозрачность (`Alpha`) и тень (`Shadow`). HTML-код этой страницы приведен далее.

```
<HTML>
  <HEAD>
    <TITLE>Два фильтра</TITLE>
    <STYLE>
      #para { width: 100%;
              filter: progid:DXImageTransform.Microsoft.Shadow
              ↳(Color=#CCCCCC, Direction=135)
              progid:DXImageTransform.Microsoft.Alpha
              ↳(Opacity=100, FinishOpacity=0, Style=1)}
    </STYLE>
  </HEAD>
  <BODY>
    <P ID="para">К этому абзацу применены сразу два фильтра:
    полупрозрачность (Alpha) и тень (Shadow).</P>
  </BODY>
</HTML>
```

Здесь мы создали стиль-селектор (см. *главу 3*), в котором и задали определения обоих фильтров, и привязали его к абзацу. Кроме того, мы явно указали размеры абзаца, в данном случае ширину — этого достаточно. Дело в том, что абзац требует явного указания хотя бы одного размера, чтобы к нему можно было применить фильтр.

Для фильтра `Alpha` мы задали следующие свойства:

- начальный уровень прозрачности (`Opacity`) — 100;
- конечный уровень прозрачности (`FinishOpacity`) — 0;

- форма градиента (`Style`) — линейный (значение 1).

Свойства для фильтра `Shadow` таковы:

- цвет тени (`Color`) — серый (значение #CCCCCC; обратим внимание, что в случае задания значения цвета прямо в стиле оно указывается без кавычек);
- направление тени (`Direction`) — 135º.

Если мы загрузим эту страницу в Internet Explorer, увидим то, что показано на рис. 15.1.

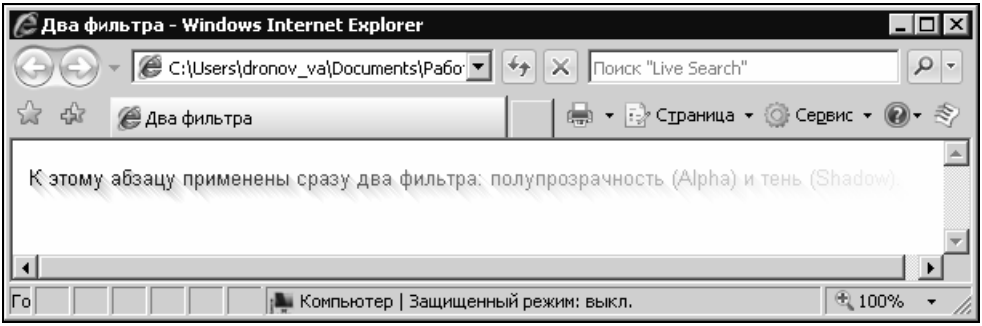

Рис. 15.1. Результат применения к абзацу фильтров `Alpha` и `Shadow`

Программное управление фильтрами

Для нас как Web-программистов полезно будет узнать, какие средства Internet Explorer предоставляет для управления фильтрами из сценариев. А их немало.

Самый простой способ применить фильтр к элементу страницы — присвоить строку, содержащую определение фильтра, свойству `filter` объекта `CSSRule`. Доступ к экземпляру этого объекта, представляющего текущий стиль какого-либо элемента страницы, можно получить через свойство `style` — об этом говорилось еще в *главе 8*.

```
<P ID="para">Сейчас мы применим к этому абзацу фильтр.</P>
. . .
<SCRIPT>
  var paraObj = document.getElementById("para");
```

```
  paraObj.style.filter = "progid:DXImageTransform.Microsoft.Shadow
  ↳(Color=#CCCCCC, Direction=45)";
</SCRIPT>
```

Этот сценарий применяет к абзацу `para` фильтр `Shadow`.

Но что делать, если нужно получить доступ к фильтрам, уже примененным к элементу страницы? Воспользоваться свойством `filters` объекта `HTMLElement`. Это свойство возвращает одноименную коллекцию, содержащую все примененные к элементу страницы фильтры. К сожалению, автору не удалось выяснить имена объектов, представляющих коллекцию фильтров и единичный фильтр; назовем их `Filters` и `Filter` соответственно.

Для доступа к нужному фильтру через коллекцию `filters` мы можем использовать как числовой, так и строковый индекс; в качестве строкового индекса выступает имя фильтра с добавленными в начале символами "DXImageTransform.Microsoft." (точка в конце обязательна).

```
paraObj.filters[0].Enabled = false;
paraObj.filters["DXImageTransform.Microsoft.Shadow"].Enabled = false;
```

Оба этих выражения отключают единственный фильтр, примененный к абзацу `para` (это фильтр `Shadow`). Как видим, для этого достаточно присвоить свойству `Enabled` фильтра значение `false`.

Да, мы можем использовать перечисленные в табл. 15.2 свойства и в сценариях! Так что параметры фильтров полностью в нашей власти.

```
paraObj.filters["DXImageTransform.Microsoft.Shadow"].Color = "#000000";
```

Например, это выражение задает для абзаца `para` черный цвет тени.

```
para2Obj.filters["DXImageTransform.Microsoft.Gradient"].StartColor =
↳0xCCCCCC;
```

А это выражение задает для фильтра `Gradient`, примененного к абзацу `para2`, в качестве начального цвета градиента серый цвет. Обратим внимание, что мы для этого используем свойство `StartColor`, принимающее числовое значение цвета. Если бы мы использовали свойство `StartColorStr`, принимающее строковое значение цвета, то должны будем написать это выражение таким образом:

```
para2Obj.filters["DXImageTransform.Microsoft.Gradient"].StartColorStr =
↳"#CCCCCC";
```

Кроме того, фильтр `Light` поддерживают ряд методов, описанных в табл. 15.4. Эти методы позволяют добавлять источники освещения, изменять их параметры и удалять их.

Таблица 15.4. *Методы фильтра* `Light`

Метод	Описание
`AddAmbient(`*`<красный>`*`, `*`<зеленый>`*`, `*`<синий>`*`, `*`<интенсивность>`*`)`	Добавляет источник рассеянного света с заданными цветовыми параметрами. Все параметры задаются в виде чисел: цвета — от 0 до 255, `интенсивность` — от 0 до 100
`AddCone(`*`<X1>`*`, `*`<Y1>`*`, `*`<Z1>`*`, `*`<X2>`*`, `*`<Y2>`*`, `*`<красный>`*`, `*`<зеленый>`*`, `*`<синий>`*`, `*`<интенсивность>`*`, `*`<угол>`*`)`	Добавляет источник направленного света с заданными цветовыми параметрами. `X1`, `Y1` и `Z1` задают координаты источника, а `X2` и `Y2` — точки, куда падает свет. Все параметры задаются в виде чисел, цвета — от 0 до 255, `интенсивность` — от 0 до 100. Координаты задаются в пикселах
`AddPoint(`*`<X>`*`, `*`<Y>`*`, `*`<Z>`*`, `*`<красный>`*`, `*`<зеленый>`*`, `*`<синий>`*`, `*`<интенсивность>`*`)`	Добавляет источник направленного света с заданными цветовыми параметрами. `X`, `Y` и `Z` задают координаты источника. Все параметры задаются в виде чисел, цвета — от 0 до 255, `интенсивность` — от 0 до 100. Координаты задаются в пикселах
`ChangeColor(`*`<номер>`*`, `*`<красный>`*`, `*`<зеленый>`*`, `*`<синий>`*`, `*`<тип значения>`*`)`	Изменяет цвет источника света с заданным в виде числа `номером`. Цвета задаются в виде чисел от 0 до 255. Последний параметр задает абсолютное (1 или любое ненулевое значение) или относительное (0) изменение цвета
`ChangeStrength(`*`<номер>`*`, `*`<интенсивность>`*`, `*`<тип значения>`*`)`	Изменяет интенсивность источника света с заданным в виде числа `номером`. `Интенсивность` задается в виде числа от 0 до 100. Последний параметр задает абсолютное (1 или любое ненулевое значение) или относительное (0) изменение интенсивности
`Clear()`	Удаляет все источники света
`MoveLight(`*`<номер>`*`, `*`<X>`*`, `*`<Y>`*`, `*`<Z>`*`, `*`<тип значения>`*`)`	Перемещает источник света с заданным `номером`. Координаты `X`, `Y` и `Z` задаются в виде чисел в пикселах. Последний параметр задает абсолютное (1 или любое ненулевое значение) или относительное (0) перемещение

К сожалению, мы не можем программно добавлять и удалять фильтры так же просто, как, скажем, элементы страницы или пункты списка. Объект коллекции `Filters` не предоставляет для этого никаких средств. Очередная недоработка Internet Explorer, в общем-то, хорошей программы….

Если же мы хотим программно изменить фильтр, примененный к элементу страницы, то можем пойти двумя путями. Во-первых, мы можем сформиро-

вать новую строку определения фильтра и присвоить ее свойству `filter` объекта `CSSRule`. Во-вторых, мы можем определить все нужные фильтры прямо в стиле, а потом просто включать и выключать нужные. Последний подход мы как раз рассмотрим в предлагаемом далее примере.

Давайте создадим страницу с абзацем и применим к этому абзацу сразу два фильтра: полупрозрачность (`Alpha`) и тень (`Shadow`). Все как в предыдущем примере, но изначально оба этих фильтра будут запрещены. Также мы предусмотрим две гиперссылки, каждая из которых будет включать "свой" фильтр. HTML-код этой страницы приведен далее.

```
<HTML>
  <HEAD>
    <TITLE>Два фильтра</TITLE>
    <STYLE>
      #para { width: 100%;
              filter: progid:DXImageTransform.Microsoft.Shadow
              ↳(Color=#CCCCCC, Direction=135, Enabled=false)
              progid:DXImageTransform.Microsoft.Alpha
              ↳(Opacity=100, FinishOpacity=0, Style=1, Enabled=false)}
    </STYLE>
    <SCRIPT>
      function applyAlpha() {
        var paraObj = document.getElementById("para");
        paraObj.filters[0].Enabled = false;
        paraObj.filters[1].Enabled = true;
      }

      function applyShadow() {
        var paraObj = document.getElementById("para");
        paraObj.filters[0].Enabled = true;
        paraObj.filters[1].Enabled = false;
      }
    </SCRIPT>
  </HEAD>
  <BODY>
    <P ID="para">К этому абзацу мы можем применить один из двух фильтров:
    полупрозрачность (Alpha) или тень (Shadow).</P>
    <P><A HREF="#" ONCLICK="applyAlpha();">Применить полупрозрачность
    (Alpha)</A></P>
```

```
    <P><A HREF="#" ONCLICK="applyShadow();">Применить тень
    (Shadow)</A></P>
  </BODY>
</HTML>
```

Комментарии, как говорится, излишни.

Преобразования

Преобразования выглядят несравнимо эффектнее, но при их создании не обойтись без программирования. Но нам к программированию не привыкать, не так ли?

Создание преобразований

Если фильтр — эффект статический, то *преобразование* — динамический, выполняющийся, что называется, прямо на глазах. Так, одно графическое изображение может плавно заменяться другим с использованием эффекта "шторы" или ирисовой диафрагмы — это и есть преобразование.

Преобразования создаются с помощью атрибута стиля `filter`, как и фильтры. В этом они схожи.

```
<IMG SRC="image.gif" STYLE="height: 100; width: 100; filter:
progid:DXImageTransform.Microsoft.Fade(Duration=2)">
```

Этот HTML-код создает на странице графическое изображение и применяет к нему преобразование "наплыв" (плавное исчезновение старого содержимого элемента страницы и такое же плавное появление нового содержимого). Это преобразование имеет имя `Fade` и поддерживает свойство `Duration`, задающее продолжительность преобразования (длительность "наплыва") и имеющее значение 2 секунды.

Только это преобразование не будет работать. И вот почему.

Фактически преобразование задает эффект, применяемый при смене элемента страницы другим. Элемент страницы может менять какие-то параметры — размеры, цвет, параметры рамки и др., — а также может менять свое содержимое. Но мы-то еще не изменили ни параметров, ни содержимого нашего абзаца `para`! Значит, созданное нами преобразование пока что будет ждать своего часа.

Чтобы запустить преобразование, нам понадобится написать небольшой сценарий, вызывающий два метода объекта `Filter`. Да-да, преобразование тоже

представляется экземпляром этого объекта! А для доступа к преобразованию мы можем воспользоваться коллекцией `filters`.

1. Сначала мы вызываем метод `Apply`, не принимающий параметров и не возвращающий результата. Этот метод фиксирует текущее состояние элемента страницы, к которому мы применили преобразование, и подготавливает это преобразование к запуску.
2. Далее мы изменяем состояние элемента страницы. Мы можем изменить какие-то его параметры (скажем, размеры, цвет, параметры рамки) или само содержимое.
3. Напоследок мы вызываем метод `Play`, также не принимающий параметров и не возвращающий результата. Этот метод собственно запускает преобразование.

```
<IMG SRC="image.gif" ID="img" STYLE="height: 100; width: 100;
visibility: hidden; filter:
progid:DXImageTransform.Microsoft.Fade(Duration=2)">
<SCRIPT>
  var imgObj = document.getElementById("img");
  imgObj.filters[0].Apply();
  imgObj.style.visibility = "visible";
  imgObj.filters[0].Play();
</SCRIPT>
```

Здесь мы сделали графическое изображение `img` изначально скрытым, а потом в процессе преобразования `Fade` постепенно делаем его видимым. Вот теперь наше преобразование будет работать.

Все доступные в Internet Explorer преобразования перечислены в табл. 15.5, а все их свойства — в табл. 15.6.

Таблица 15.5. *Преобразования, поддерживаемые Internet Explorer*

Преобразование	Описание
`Barn`	Создает эффект "открывающейся" и "закрывающейся" "двери"
`BlendTrans`	Плавно заменяет старое содержимое элемента страницы на новое
`Blinds`	Создает эффект "открывающихся" и "закрывающихся" в заданном направлении "жалюзи"
`CheckerBoard`	Создает эффект "шахматной доски", движущейся в заданном направлении

Таблица 15.5 *(окончание)*

Преобразование	Описание
`Fade`	Создает эффект наплыва (старое содержимое элемента страницы плавно пропадает, а новое одновременно так же плавно появляется)
`GradientWipe`	Новое содержимое элемента страницы наползает на старое, причем граница между ними выглядит как градиентная цветная полоса
`Inset`	Новое содержимое элемента страницы диагонально наползает на старое
`Iris`	Создает эффект ирисовой диафрагмы
`Pixelate`	Старое содержимое элемента страницы "рассыпается" на отдельные пикселы и пропадает, а новое "собирается" из отдельных пикселов
`RadialWipe`	Новое содержимое элемента страницы радиально наползает на старое
`RandomBars`	Старое содержимое элемента страницы "рассыпается" на отдельные линии и пропадает, а новое "собирается" из отдельных линий
`RandomDissolve`	Новое содержимое элемента страницы попиксельно "проявляется" на месте старого
`RevealTrans`	Плавно заменяет старое содержимое элемента управления на новое, используя заданный эффект
`Slide`	Старое содержимое элемента страницы сдвигается в сторону, открывая под собой новое
`Spiral`	Новое содержимое элемента страницы спирально закрашивает старое
`Stretch`	Новое содержимое элемента страницы растягивается, заменяя собой старое
`Strips`	Новое содержимое элемента страницы диагонально отдельными полосками наползает на старое
`Wheel`	Новое содержимое элемента страницы посекторно наползает на старое
`Zigzag`	Новое содержимое элемента страницы зигзагообразно, отдельными полосками наползает на старое

Таблица 15.6. *Свойства преобразований*

Свойство	Преобразование	Описание
`bands`	`Blinds`, `Slide`	Количество полосок. Значение по умолчанию зависит от преобразования
`Direction`	`Blinds`, `CheckerBoard`	Направление движения. Может быть одним из четырех строковых значений: "up" (вверх), "down" (вниз), "right" (вправо) и "left" (влево). Значение по умолчанию зависит от преобразования
`Duration`	Все преобразования	Продолжительность преобразования в секундах. Значение по умолчанию отсутствует, поэтому данное свойство всегда следует задавать явно
`Dx`, `Dy`	`Matrix`	Значения `fDx` и `fDy` матричных преобразований. Значение по умолчанию — 1.0
`Enabled`	Все преобразования	Если `true` (значение по умолчанию), то преобразование применяется к элементу страницы, если `false` — не применяется
`FilterType`	`Matrix`	Задает тип пикселов нового содержимого: "bilinear" (значение по умолчанию) или "nearest neighbor"
`GradientSize`	`GradientWipe`	Часть площади элемента страницы, покрываемой градиентной полосой. Может быть от 0.0 до 1.0. Значение по умолчанию — 0.25
`GridSizeX`, `GridSizeY`	`Spiral`, `Zigzag`	Количество полосок по горизонтали или вертикали соответственно. Может быть от 1 до 100. Значение по умолчанию — 16
`IrisStyle`	`Iris`	Форма апертуры "ирисовой диафрагмы". Доступны значения: "DIAMOND" (бриллиант), "CIRCLE" (круг), "CROSS" (X-образная), "PLUS" (знак "плюс"; значение по умолчанию), "SQUARE" (квадратная) и "STAR" (звездообразная)
`M11`, `M22`	`Matrix`	Значения `fM11` и `fM22` матричных преобразований. Значение по умолчанию — 1.0
`M12`, `M21`	`Matrix`	Значения `fM12` и `fM21` матричных преобразований. Значение по умолчанию — 0.0
`MaxSquare`	`Pixelate`	Максимальный размер "пиксела" в пикселах. Значение по умолчанию — 50

Таблица 15.6 *(продолжение)*

Свойство	Преобразование	Описание
`Motion`	`Barn`, `Iris`	Если "out" (значение по умолчанию), движение будет происходить от центра элемента страницы к его границам, если "in" — от границ к центру
`Motion`	`Strips`	Угол наклона полосок и направление движения нового содержимого. Доступны значения: "leftdown" (полоски наклонены влево, движение выполняется вниз; значение по умолчанию), "leftup" (влево и вверх), "rightdown" (вправо и вниз) и "rightup" (вправо и вверх)
`Motion`	`GradientWipe`	Если "forward" (значение по умолчанию), движение производится согласно значению свойства `WipeStyle`, если "reverse" — в обратном направлении
`Orientation`	`Barn`, `RandomBars`	Если "horizontal", преобразование происходит по горизонтали, если "vertical" — по вертикали. Значение по умолчанию зависит от преобразования
`Overlap`	`Fade`	Время, относительно общей продолжительности преобразования, когда и старое, и новое содержимое элемента страницы отображаются одновременно. Может быть от 0.0. до 1.0 (значение по умолчанию)
`SizingMethod`	`Matrix`	Способ размещения нового содержимого в границах элемента страницы. Доступны значения: "clip to original" (обрезание содержимого; значение по умолчанию) и "auto expand" (уменьшение или увеличение содержимого)
`SlideStyle`	`Slide`	Способ замещения содержимого элемента страницы. Доступны значения: "HIDE" (скрытие; значение по умолчанию), "PUSH" ("выталкивание") и "SWAP" (замена)
`spokes`	`Wheel`	Количество секторов. Может быть от 2 до 20. Значение по умолчанию — 4
`SquaresX`, `SquaresY`	`CheckerBoard`	Количество рядов по горизонтали и вертикали. Значения по умолчанию — 12 и 10 соответственно

Таблица 15.6 *(окончание)*

Свойство	Преобразование	Описание
`StretchStyle`	`Stretch`	Способ замещения содержимого элемента страницы. Доступны значения: "HIDE" (скрытие), "PUSH" ("выталкивание") и "SPIN" (замена; значение по умолчанию)
`Transition`	`RevealTrans`	Номер применяемого эффекта. Список всех возможных значений приведен в табл. 15.7. Значение по умолчанию отсутствует
`WipeStyle`	`RadialWipe`	Способ замещения содержимого элемента страницы. Доступны значения: "CLOCK" (вращение вокруг центра элемента страницы по часовой стрелке; значение по умолчанию), "WEDGE" (вращение сразу в обе стороны) и "RADIAL" (радиальное вращение)
`WipeStyle`	`GradientWipe`	Если 0 (значение по умолчанию), движение происходит по горизонтали, если 1 — по вертикали

Таблица 15.7. *Доступные значения свойства* `Transition` *преобразования* `RevealTrans`

Значение	Описание
0	Увеличивающийся прямоугольник
1	Уменьшающийся прямоугольник
2	Увеличивающийся круг
3	Уменьшающийся круг
4	Подъем снизу
5	Соскальзывание сверху
6	Сдвиг справа
7	Сдвиг слева
8	Вертикальные "жалюзи"
9	Горизонтальные "жалюзи"
10	"Шахматная доска" с увеличивающимися клетками
11	"Шахматная доска" с уменьшающимися клетками

Таблица 15.7 *(окончание)*

Значение	Описание
12	Размытие и появление
13	"Раскрывающаяся дверь" по вертикали
14	"Закрывающаяся дверь" по вертикали
15	"Раскрывающаяся дверь" по горизонтали
16	"Закрывающаяся дверь" по горизонтали
17	"Наезд" из левого нижнего угла
18	"Наезд" из левого верхнего угла
19	"Наезд" из правого нижнего угла
20	"Наезд" из правого верхнего угла
21	Горизонтальные полосы
22	Вертикальные полосы
23	Случайно выбранный из приведенных ранее эффект

В качестве примера давайте создадим страницу с абзацем, меняющим свое содержимое после загрузки страницы. При этом к нему будет применено преобразование "ирисовая диафрагма" (`Iris`).

```
<HTML>
  <HEAD>
    <TITLE>Преобразование</TITLE>
    <STYLE>
      #para { width: 100%;
              font-size: 40pt;
              filter: progid:DXImageTransform.Microsoft.Iris
              (IrisStyle=CIRCLE, Duration=2)}
    </STYLE>
  </HEAD>
  <BODY>
    <P ID="para">Страница загружается...</P>
    <SCRIPT>
      var paraObj = document.getElementById("para");
      paraObj.filters[0].Apply();
      paraObj.innerText = "...Загрузилась!";
```

```
      paraObj.filters[0].Play();
    </SCRIPT>
  </BODY>
</HTML>
```

Для абзаца `para`, к которому будет применено преобразование, мы задали размер шрифта 40 пунктов, чтобы он был крупнее и примененное к нему преобразование выглядело эффектнее. И не забудем задать для него хотя бы один размер (в нашем случае — ширину), иначе Internet Explorer не сможет применить к нему преобразование.

А для преобразования `Iris` мы задали такие свойства:

- форма апертуры (`IrisStyle`) — круг (значение "CIRCLE");
- продолжительность преобразования (`Duration`) — 2 секунды.

Вот и все.

Программное управление преобразованиями

Преобразования предоставляют несколько больше возможностей по управлению ими из сценариев. Сейчас мы их рассмотрим.

Прежде всего, объект `Filter` в случае преобразований поддерживает два полезных свойства. Они доступны только в сценариях и позволяют узнать, на какой стадии находится текущее преобразование.

Свойство `Percent` задает или возвращает процент выполнения преобразования в виде строки, содержащей число от 0 до 100. Отметим — именно строки, а не числа!

```
<IMG SRC="image.gif" ID="img" STYLE="height: 100; width: 100;
visibility: hidden; filter:
progid:DXImageTransform.Microsoft.Fade(Duration=2)">
<SCRIPT>
  var imgObj = document.getElementById("img");
  imgObj.filters[0].Apply();
  imgObj.style.visibility = "visible";
  imgObj.filters[0].Percent = "20";
  imgObj.filters[0].Play();
</SCRIPT>
```

Теперь преобразование, примененное к графическому изображению `img`, начнется не с самого начала, а с отметки 20%.

Свойство `status` возвращает число, обозначающее стадию, на которой находится преобразование. Таких чисел (и состояний) может быть три:

- 0 — преобразование завершилось или было остановлено вызовом метода `Stop` (будет описан далее);
- 1 — был вызван метод `Apply`, но преобразование еще не началось;
- 2 — преобразование выполняется.

Кроме того, все преобразования поддерживают метод `Stop`. Этот метод немедленно останавливает преобразование. Он не принимает параметров и не возвращает результата.

Еще для нас представляет интерес событие `onFilterChange` объекта `HTMLElement`. Это событие возникает, когда преобразование завершается. Оно не всплывает, поведение по умолчанию у него как таковое отсутствует. Мы можем использовать это событие, например, для запуска нового преобразования после завершения старого.

Если мы будем обрабатывать событие, нам пригодится свойство `srcFilter` объекта `Event`, то есть самого события. Оно возвращает фильтр (экземпляр объекта `Filter`), который вызвал возникновение данного события.

Давайте изменим предыдущий пример таким образом, чтобы в абзаце `para` последовательно, одно за другим, менялись два содержимых. Преобразование мы оставим старое — `Iris`.

```
<HTML>
  <HEAD>
    <TITLE>Преобразование</TITLE>
    <STYLE>
      #para { width: 100%;
              font-size: 40pt;
              filter: progid:DXImageTransform.Microsoft.Iris
              ↳(IrisStyle=CIRCLE, Duration=2)}
    </STYLE>
    <SCRIPT>
      var flag = false;

      function paraFilterChange() {
        var paraObj = document.getElementById("para");
        paraObj.filters[0].Apply();
        paraObj.innerText = (flag) ? "Страница загружается..." :
        "...Загрузилась!";
```

```
        paraObj.filters[0].Play();
        flag = !flag;
      }
    </SCRIPT>
  </HEAD>
  <BODY>
    <P ID="para" ONFILTERCHANGE="paraFilterChange();">Страница
    загружается...</P>
    <SCRIPT>
      paraFilterChange();
    </SCRIPT>
  </BODY>
</HTML>
```

Здесь мы использовали переменную `flag`, чтобы хранить признак того, должна ли в абзаце `para` присутствовать первая или вторая строка. Остальное пояснений не требует.

> ***Внимание!***
>
> Описанный в этой главе синтаксис определения фильтров и преобразований поддерживается Internet Explorer, начиная с версии 5.5. Но, вообще-то, фильтры и преобразования начали поддерживаться Internet Explorer 4.0, где использовался другой синтаксис, не совместимый с рассмотренным нами.

> ***На заметку***
>
> Во всех подробностях фильтры и преобразования описаны в посвященном им разделе MSDN (**http://msdn.microsoft.com/workshop/author/filter/filters_transitions_entry.asp**). Там же описывается старый синтаксис определения фильтров и преобразований, использовавшийся в версиях 4.0—5.5 Internet Explorer.

Применение преобразований к странице

Internet Explorer позволяет применять преобразования не только к отдельным элементам страницы, но и к странице целиком. При этом преобразование будет выполняться при загрузке данной страницы и при уходе с нее в результате загрузки другой страницы.

Для привязки преобразований к страницам используется знакомый нам по *главе 2* одинарный тег `<META>`. В этом случае его формат таков:

```
<META HTTP-EQUIV="Page-Enter|Page-Exit"
↳CONTENT="<определение преобразования>"
```

Если мы хотим, чтобы данное преобразование применялось при загрузке текущей страницы, то должны использовать в качестве значения атрибута `HTTP-EQUIV` тега `<META>` значение "Page-Enter". А значение "Page-Exit" этого же атрибута применит преобразование при уходе со страницы.

Тег `<META>` должен помещаться в секции заголовка страницы (в теге `<HEAD>`).

```
<HEAD>
  . . .
  <META HTTP-EQUIV="Page-Enter"
  CONTENT="progid:DXImageTransform.Microsoft.Fade(Duration=2)"
  <META HTTP-EQUIV="Page-Exit"
  CONTENT="progid:DXImageTransform.Microsoft.Fade(Duration=2)"
</HEAD>
```

Если приведенный ранее HTML-код поместить в секцию заголовка страницы, то при ее загрузке и при уходе с нее будет выполняться преобразование `Fade` продолжительностью в 2 секунды.

На этом о фильтрах и преобразованиях все.

Что дальше?

В данной главе мы рассмотрели особые эффекты, поддерживаемые Internet Explorer и применяемые к элементам страницы, — фильтры и преобразования. Они помогут нам без особых затрат украсить наши будущие Web-творения.

Но всему на свете приходит конец. Вот и пришел конец богатым возможностям Internet Explorer в плане Web-дизайна и Web-программирования. В следующей главе мы рассмотрим еще одну возможность, предлагаемую Internet Explorer и весьма полезную Web-программистам, — поведения. И распрощаемся с этим Web-обозревателем.

Глава 16

Поведения и HTML-компоненты (Internet Explorer)

Разработчики Internet Explorer — странные люди. С одной стороны, они допускают в своем творении досаднейшие ошибки и недоделки (мы уже в этом не раз убеждались…). С другой, предусматривают возможности, которыми больше ни один Web-обозреватель похвастаться не может. В том числе и возможности, облегчающие труд Web-программиста.

Предположим такую, весьма часто встречающуюся ситуацию. Мы создаем на странице несколько элементов, которые должны вести себя одинаковым образом в ответ на действия посетителя. (Это могут быть те же самые горячие изображения, которые мы создавали в *главе 9*.) Мы пишем HTML-, CSS- и JavaScript-код для одного такого элемента страницы, потом делаем то же самое для второго, третьего, четвертого… Сколько работы, и это притом, что сам код почти не меняется! Неужели нельзя написать все один-единственный раз, оформить все это в виде отдельного файла и потом только подставлять в код страницы ссылки на этот файл?

Можно! Именно для этого в Internet Explorer предусмотрена поддержка так называемых поведений и HTML-компонентов, которыми мы займемся в этой главе.

Поведения

Поведение — это аналог стилей CSS, но применительно к сценариям. То есть мы пишем сценарий, управляющий поведением элемента страницы в ответ на действия посетителя, помещаем его в отдельный файл и подключаем к нужному элементу страницы с помощью особого атрибута стиля. И все — элемент страницы будет работать так, как нам нужно, без каких-либо дополнительных действий!

Создание простых поведений

Давайте выясним, как создаются самые простые поведения. И выберем в качестве "подопытного кролика" горячие изображения.

Как мы уже знаем, практически всегда Web-программирование заключается в обработке определенных событий элемента страницы. Мы пишем функции-обработчики и привязываем их к нужным событиям данного элемента. Если мы рассмотрим приведенные в этой книге примеры, то абсолютное большинство из них будут включать в себя обработчики событий.

Значит, при создании поведений нам нужно вынести в отдельный файл именно обработчики событий. Скажем больше — все поведения по большему счету суть набор обработчиков событий, которые при подключении поведения к элементу страницы будут автоматически к нему привязаны.

Поведения хранятся в текстовых файлах с расширением htc. Эти файлы содержат, в основном, JavaScript-код сценариев плюс некоторые служебные HTML-теги, используемые для объявления событий, к которым будут привязаны сценарии-обработчики, и некоторых служебных нужд.

Формат содержимого файла, в котором хранится простейшее поведение, таков:

```
<PUBLIC:COMPONENT>
  <объявление событий и их обработчиков>
  <SCRIPT>
    <код обработчиков событий>
  </SCRIPT>
</PUBLIC:COMPONENT>
```

Как видим, все содержимое кода поведения помещается внутри особого парного тега <PUBLIC:COMPONENT>. Собственно, этот тег и создает поведение.

Внутри тега <PUBLIC:COMPONENT> находятся объявления событий элемента страницы, к которому будет подключено поведение, и обработчиков этих событий. Эти объявления записываются с использованием одинарного тега <PUBLIC:ATTACH>.

```
<PUBLIC:ATTACH EVENT="<имя события>"
↳[FOR="<элемент страницы, событие которого обрабатывается>"]
↳ONEVENT="<обработчик события>"/>
```

А теперь — один важный момент! Мы привыкли, что в языке HTML одинарные теги не содержат закрывающей пары (собственно, поэтому они и называются одинарными). Но поведения — другое дело, и даже одинарный тег

(например, <PUBLIC:ATTACH>) должен содержать закрывающий символ / перед символом >. Вот так (символ / выделен полужирным шрифтом):

<PUBLIC:ATTACH *<атрибуты>*/>

Обязательный атрибут EVENT этого тега задает само событие, которое должно быть обработано. В качестве его значения указывается имя нужного события.

Необязательный атрибут FOR указывает элемент страницы, событие которого должно быть обработано. Его значение может быть одной из следующих строк:

- "element" — элемент страницы, к которому подключено поведение. Это значение по умолчанию, применяемое, если атрибут FOR не указан;
- "document" — секция тела страницы;
- "window" — текущее окно Web-обозревателя.

То есть мы можем с помощью данного поведения обрабатывать события не только элемента страницы, к которому подключено поведение, но и тела страницы и окна Web-обозревателя, в котором она загружена. Иногда это бывает полезно.

Наконец, обязательный атрибут ONEVENT содержит сам код обработчика указанного события.

Код обработчиков событий и, возможно, загрузочные сценарии помещаются в уже знакомый нам парный тег <SCRIPT>, который также должен находиться внутри тега <PUBLIC:COMPONENT> и следовать за тегами <PUBLIC:ATTACH>, объявляющими события и их обработчики. Этот код пишется точно так же, как код обычных обработчиков, которые помещаются прямо в странице и которых мы написали уже бог знает сколько.

В коде обработчиков событий, присутствующих в поведении, можно использовать две очень полезных переменных. И не просто очень полезных — без них зачастую никуда. Эти переменные объявляет сам Internet Explorer. Переменная element хранит ссылку на элемент страницы (экземпляр объекта HTMLElement), к которому подключено данное поведение. А переменная document хранит ссылку на страницу (экземпляр объекта HTMLDocument), на которой находится этот элемент.

Всех этих знаний нам будет вполне достаточно, чтобы написать простейшее поведение. Давайте реализуем в нем функциональность горячего изображения. В качестве заменяющего изображения, которое будет подставлено на место изначального при наведении на него курсора мыши, используем изображение hover.jpg. (Впоследствии мы научимся создавать свои атрибуты для поведений и сможем задавать любое заменяющее изображение.)

HTML-код нашего первого поведения представлен далее.

```
<PUBLIC:COMPONENT>
  <PUBLIC:ATTACH EVENT="onmouseover" ONEVENT="elementMouseOver();"/>
  <PUBLIC:ATTACH EVENT="onmouseout" ONEVENT="elementMouseOut();"/>
  <SCRIPT>
    var primaryImg = "";

    function elementMouseOver() {
      primaryImg = element.src;
      element.src = "hover.jpg";
    }

    function elementMouseOut() {
      element.src = primaryImg;
    }
  </SCRIPT>
</PUBLIC:COMPONENT>
```

Давайте поговорим о нем.

Когда посетитель наведет на наше горячее изображение курсор мыши, мы должны сменить изначальное изображение на заменяющее, а когда уведет курсор прочь — вернуть изначальное изображение на место. Это значит, что мы должны сохранить интернет-адрес изначального изображения перед тем, как заменить его заменяющим, — ведь иначе мы его выяснить не сможем. Для этого мы в небольшом загрузочном сценарии объявили переменную `primaryImg`. Эта переменная будет доступна только в данном экземпляре поведения, так что мы можем использовать ее без боязни.

Далее мы объявляем два события, которые нам следует обработать, чтобы горячее изображение работало нормально: `onMouseOver` и `onMouseOut`. И, разумеется, объявляем функции-обработчики этих событий: `elementMouseOver` и `elementMouseOut`. Первая функция сохраняет в переменной `primaryImg` интернет-адрес текущего изображения — а это будет изначальное изображение — и подставляет на его место заменяющее изображение. Вторая функция загружает изображение, интернет-адрес которого хранится в переменной `primaryImg`, то есть изначальное изображение. Все.

Сохраним данное поведение в файл с именем 16.1.htc. Впоследствии мы его неоднократно используем и неоднократно модифицируем.

Вот и все! Теперь наше первое, пока еще совсем простое поведение готово к подключению к элементу страницы — графическому изображению, помещенному в гиперссылку (изображению-гиперссылке).

Подключение поведений к элементам страницы

Подключить поведение к элементу страницы можно с помощью атрибута стиля `behavior`. Его значение записывается в таком формате:

```
url(<интернет-адрес файла поведения>)
```

То есть `интернет-адрес нужного нам файла поведения` помещается в скобки, а перед ними ставятся символы "url", причем пробелы между этими символами и открывающей скобкой недопустимы.

Что ж, сразу же рассмотрим пример страницы с горячим изображением, созданным с помощью поведения 16.1.htc. Далее представлен ее HTML-код.

```
<HTML>
  <HEAD>
    <TITLE>Простейшее поведение</TITLE>
    <STYLE TYPE="text/css">
      .hotimage { behavior: url(16.1.htc); }
    </STYLE>
  </HEAD>
  <BODY>
    <P><A HREF="#"><IMG CLASS="hotimage" SRC="default.jpg"></A></P>
  </BODY>
</HTML>
```

И все! Если бы мы создавали горячее изображение традиционным способом, рассмотренным в *главе 9*, код был бы раза в два больше.

> ***На заметку***
>
> Вообще-то обольщаться насчет того, что поведения делают код страниц компактнее, не стоит. Код зачастую остается таким же объемным, просто из файла страницы переносится в файл поведения.
>
> Исходя их этого можно заметить, что поведения стоит создавать только в том случае, если мы хотим использовать их многократно, подключая к множеству элементов на разных страницах. Если же нам нужно "оживить" каким-то образом только один элемент на одной странице, лучше это сделать традиционным способом.

Поведение 16.1.htc мы подключили в стилевом классе `hotimage`, который привязали к тегу `<IMG>`, создающему изображение. В принципе, мы могли бы не городить огород со стилевым классом, а записать проще:

```
<IMG STYLE="behavior: url(16.1.htc);" SRC="default.jpg">
```

Но горячие изображения, как правило, не присутствуют на странице поодиночке. А стилевой класс — лучший способ привязать один и тот же стиль сразу к нескольким элементам страницы.

Специфические события поведений и их обработка

Как мы уже знаем, в поведении можно помещать обработчики не только событий самого элемента страницы, к которому это поведение подключено, а также страницы, где присутствует этот элемент, и текущего окна Web-обозревателя. Но часто бывает необходимо в поведении отследить момент окончания загрузки содержимого элемента страницы и самой страницы. Как это сделать?

Легко! С помощью особых событий, доступных только в поведении. Таких событий четыре, и сейчас с ними разберемся.

Первое событие, о котором нам следует знать, — `onContentReady`. Оно возникает после окончания загрузки и обработки HTML-кода, формирующего элемент страницы, к которому подключено данное поведение.

Часто нам приходится изменять содержимое или параметры элемента страницы, к которому подключено поведение, прямо из поведения. Перед тем как изменять их, нам нужно будет отследить момент окончания его загрузки, иначе возможны всяческие неприятности. Событие `onContentReady` как раз позволяет это сделать.

Второе нужное нам событие — `onDocumentReady`. Оно возникает после окончания загрузки и обработки HTML-кода всей страницы.

Если мы хотим изменять из поведения содержимое или параметры других элементов страницы, то должны отслеживать момент окончания загрузки всей страницы, иначе результаты могут быть совсем не те, на которые мы рассчитываем. В этом случае нам поможет событие `onDocumentReady`.

Событие `onDetach` возникает при отключении поведения от элемента страницы (как отключить поведение, будет описано далее). Оно обрабатывается не столь часто, как два описанные ранее события, и может использоваться для выполнения какого-либо завершающего кода.

Еще одно событие — `onContentSave` — возникает при попытке скопировать содержимое элемента страницы в Буфер обмена или сохранить страницу в файле. Зачем оно может понадобиться, автор даже не способен себе представить.

Привязка обработчиков к этим событиям выполняется с помощью уже знакомого нам тега `<PUBLIC:ATTACH>`.

```
<PUBLIC:ATTACH EVENT="<имя события>"
↳ONEVENT="<обработчик события>"/>
```

Атрибут `FOR` здесь указывать не нужно.

В качестве примера давайте перепишем наше первое поведение таким образом, чтобы смена изображений выполнялась только после загрузки всей страницы. Далее приведен его HTML-код.

```
<PUBLIC:COMPONENT>
  <PUBLIC:ATTACH EVENT="onmouseover" ONEVENT="elementMouseOver();"/>
  <PUBLIC:ATTACH EVENT="onmouseout" ONEVENT="elementMouseOut();"/>
  <PUBLIC:ATTACH EVENT="ondocumentready" ONEVENT="documentReady();"/>
  <SCRIPT>
    var primaryImg = "";
    var isDocumentLoaded = false;

    function elementMouseOver() {
      if (isDocumentLoaded) {
        primaryImg = element.src;
        element.src = "hover.jpg";
      }
    }

    function elementMouseOut() {
      if (isDocumentLoaded)
        element.src = primaryImg;
    }

    function documentReady() {
      isDocumentLoaded = true;
    }
  </SCRIPT>
</PUBLIC:COMPONENT>
```

Здесь мы объявили переменную `isDocumentLoaded`, хранящую признак того, загружена страница полностью (значение `true`) или нет (значение `false`). Изначально она будет хранить значение `false`, так как страница еще не загружена.

Далее мы добавили обработчик события `onDocumentReady` — функцию `documentReady`. Она выполняет единственную задачу — присваивает переменной `isDocumentLoaded` значение `true`, то есть отмечает, что страница уже загружена.

И еще мы немного изменили функции — обработчики событий `onMouseOver` и `onMouseOut`. Теперь они выполняют смену изображений только в том случае,

если страница полностью загружена (переменная `isDocumentLoaded` хранит значение `true`).

Мы можем сохранить это поведение в файле 16.2.htc и проверить его в действии. Для этого достаточно взять HTML-код проверочной страницы для первого поведения 16.1.htc и заменить в нем имя файла поведения на 16.2.htc.

Создание свойств поведения

Поведения Internet Explorer имеют очень полезную особенность — они позволяют добавлять к элементу страницы, к которому подключены, новые свойства (их можно использовать и как атрибуты тега), методы и события. Если мы хотим использовать всю мощь поведений, то обязательно должны научиться это делать.

Тем более что наше простейшее поведение так и просит нового свойства. Это свойство и соответствующий ему атрибут тега будет задавать заменяющее изображение.

Новое свойство поведения создается с помощью одинарного тега `PUBLIC:PROPERTY`.

```
<PUBLIC:PROPERTY NAME="<имя свойства>" [VALUE="<значение по умолчанию>"]
[INTERNALNAME="<имя переменной, соответствующей свойству>"]
[GET="<вызов get-функции>"] [PUT="<вызов put-функции>"]/>
```

Отметим, что этот тег также должен содержать закрывающий символ `/` перед символом `>`. Как и все одинарные теги, используемые в HTML-коде поведений.

Единственный обязательный атрибут тега `<PUBLIC:PROPERTY>` — `NAME` — задает имя свойства (атрибута тега). Это имя мы можем использовать в коде обработчиков событий поведения, чтобы получить или задать его значение.

Что касается значения нового свойства, то здесь нужно иметь в виду следующее.

- Если значение свойства задано через одноименный атрибут тега, оно всегда имеет строковый тип.
- Если значение свойства задано в сценарии, оно имеет тот тип, что, собственно, и задан (строковый, числовой или логический).

Но вернемся к тегу `<PUBLIC:PROPERTY>`. Необязательный атрибут `VALUE` задает для свойства значение по умолчанию. Если он не указан, значение этого свойства обязательно должно быть задано; то есть мы фактически создадим обязательный атрибут.

Может случиться так, что заданное нами в атрибуте NAME имя свойства совпадает с именем какой-либо переменной, функции или объекта, что присутствует в коде сценариев поведения. В таком случае мы не сможем обратиться к созданному нами свойству по этому имени, иначе получим ошибку.

Для таких случаев предназначен необязательный атрибут INTERNALNAME, задающий имя переменной, которая будет использоваться для доступа к значению этого свойства в сценариях поведения. Эту переменную мы сами должны объявить в загрузочном сценарии и при этом задать для нее значение свойства по умолчанию. Атрибут VALUE, задающий для свойства значение по умолчанию, в этом случае не указывается.

```
<PUBLIC:PROPERTY NAME="enabled" INTERNALNAME="elementEnabled"/>
. . .
  <SCRIPT>
    var elementEnabled = true;
    . . .
  </SCRIPT>
```

Этот тег создает свойство enabled и задает для доступа к его значению переменную elementEnabled. Эта переменная объявляется в загрузочном сценарии, где ей присваивается значение по умолчанию true.

Необязательный атрибут GET задает имя функции, которая будет вызвана при запросе значения этого свойства (так называемой *get-функции*). То есть если в сценарии страницы, на которой расположен данный элемент, выполняется обращение к данному свойству с целью получить его значение, фактически при этом вызывается заданная нами get-функция.

Зачем это нужно? Во-первых, мы сами можем отвести для хранения значения данного свойства какую-то переменную, и тогда нам придется создавать get-функцию для извлечения из нее значения. Во-вторых, при чтении значения свойства из такой переменной мы можем выполнять какие-то дополнительные вычисления, например, преобразовывать тип этого значения в другой. В-третьих, значение свойства у нас может вообще нигде не храниться, а получаться в результате вычислений, которые будут выполнены в теле get-функции.

Get-функция должна удовлетворять двум условиям: она не должна принимать параметров и должна возвращать значение, которое и будет значением данного свойства. В теле get-функции могут выполняться любые вычисления; в самом простом случае это может быть обращение к переменной, хранящей значение свойства.

Необязательный атрибут PUT задает имя функции, которая будет вызвана при присвоении нового значения этому свойству (*set-функции*; термин "put-функция"

не прижился, так как в большинстве языков программирования для задания такой функции используется ключевое слово `set`). Он, так сказать, "пара" атрибуту `GET`.

Set-функция может выполнять множество задач. Во-первых, если мы храним значение свойства в объявленной нами самим переменной, то должны будем сами заносить его в эту переменную. Во-вторых, значение нашего свойства может присваиваться непосредственно какому-либо свойству элемента страницы. И, в-третьих, при присвоении нового значения свойству мы можем выполнить какие-то дополнительные вычисления.

Set-функция должна удовлетворять трем условиям.

1. Она должна принимать единственный параметр, в который будет занесено новое значение свойства.
2. Она не должна возвращать результата.
3. Она должна вызвать метод `fireChange` объекта, соответствующего тегу `<PUBLIC:PROPERTY>`. Этот метод не принимает параметров, не возвращает результата и уведомляет Internet Explorer об изменении значения одного из свойств данного элемента страницы. Чтобы получить доступ к тегу `<PUBLIC:PROPERTY>` из сценария, мы должны будем дать ему имя с помощью атрибута `ID`.

Как уже говорилось, в теле set-функции мы можем занести новое значение свойства в какую-либо переменную или свойство элемента страницы. При этом мы должны учитывать, что тип этого свойства может быть любым: строковым, числовым или логическим. Поэтому нам придется проверить тип полученного значения и при необходимости преобразовать его в нужный нам тип.

И еще кое-что, касающееся get- и set-функций…

- Если мы не укажем ни get-, ни set-функции, для доступа к значению свойства можем использовать либо имя свойства, либо переменную, заданную атрибутом `INTERNALNAME`.
- Если мы укажем только get-функцию, свойство будет доступно только для чтения; записать в него новое значение мы не сможем.
- Если мы укажем только set-функцию, свойство будет доступно только для записи; получить его значение мы не сможем.
- Если мы укажем и get-, и set-функцию, свойство будет доступно и на чтение, и на запись.
- Если указана get- и (или) set-функция, значение атрибута `INTERNALNAME` игнорируется.

Для примера давайте добавим в наше поведение свойство `hover` (и, соответственно, атрибут `HOVER`), задающее заменяющее изображение. В качестве основы возьмем поведение 16.1.htc — его код проще и компактнее, чем код 16.2.htc.

HTML-код нового поведения приведен далее.

```
<PUBLIC:COMPONENT>
  <PUBLIC:ATTACH EVENT="onmouseover" ONEVENT="elementMouseOver();"/>
  <PUBLIC:ATTACH EVENT="onmouseout" ONEVENT="elementMouseOut();"/>
  <PUBLIC:PROPERTY NAME="hover"/>
  <SCRIPT>
    var primaryImg = "";

    function elementMouseOver() {
      primaryImg = element.src;
      element.src = element.hover;
    }

    function elementMouseOut() {
      element.src = primaryImg;
    }
  </SCRIPT>
</PUBLIC:COMPONENT>
```

Как видим, код практически не изменился по сравнению с 16.1.htc. Мы только объявили новое свойство и использовали его значение для задания заменяющего изображения.

Сохраним новое поведение в файле 16.3.htc и исправим тестовую страницу, чей код был приведен ранее. Ее обновленный HTML-код приведен далее.

```
<HTML>
  <HEAD>
    <TITLE>Простейшее поведение</TITLE>
    <STYLE TYPE="text/css">
      .hotimage { behavior: url(16.3.htc); }
    </STYLE>
  </HEAD>
  <BODY>
    <P><A HREF="#"><IMG CLASS="hotimage" SRC="default.jpg"
    HOVER="hover.jpg"></A></P>
  </BODY>
</HTML>
```

Здесь мы только изменили имя файла поведения и задали заменяющее изображение с помощью атрибута `HOVER`, соответствующего созданному нами свойству.

Как мы уже знаем, к свойству, созданному в поведении тегом `<PUBLIC:PROPERTY>`, можно обращаться и из сценариев, расположенных на самой странице. Это, конечно, полезно, но не для нашего поведения. И вот почему…

Давайте рассмотрим такой случай. Посетитель навел курсор мыши на наше горячее изображение, заменяющее изображение сменило изначальное, и в этот момент какой-то сценарий присваивает новое значение свойству `hover`. Что произойдет? А ничего. Заменяющее изображение останется старым.

Теперь предположим, что другой сценарий присвоит новое значение свойству `src`, пытаясь задать новое изначальное изображение, в тот момент, когда в теге `<IMG>` присутствует изображение заменяющее. Посетитель уводит курсор мыши с горячего изображения и видит… старое изначальное изображение. Неприятно…

Давайте исправим поведение 16.3.htc таким образом, чтобы при программном изменении интернет-адресов изначального и заменяющего изображения соответствующие изображения тут же появлялись на странице. HTML-код исправленного поведения приведен далее.

```
<PUBLIC:COMPONENT>
  <PUBLIC:ATTACH EVENT="onmouseover" ONEVENT="elementMouseOver();"/>
  <PUBLIC:ATTACH EVENT="onmouseout" ONEVENT="elementMouseOut();"/>
  <PUBLIC:PROPERTY ID="primaryTag" NAME="primary" GET="getPrimary"
  PUT="setPrimary"/>
  <PUBLIC:PROPERTY ID="hoverTag" NAME="hover" GET="getHover"
  PUT="setHover"/>
  <SCRIPT>
    var isHovered = false;
    var iPrimary = "";
    var iHover = "";

    function elementMouseOver() {
      element.src = iHover;
      isHovered = true;
    }

    function elementMouseOut() {
      element.src = element.primary;
```

```
    isHovered = false;
  }

  function getPrimary() {
    return iPrimary;
  }

  function setPrimary(pValue) {
    iPrimary = pValue;
    if (!(isHovered)) element.src = iPrimary;
    primaryTag.fireChange();
  }

  function getHover() {
    return iHover;
  }

  function setHover(pValue) {
    iHover = pValue;
    if (isHovered) element.src = iHover;
    hoverTag.fireChange();
  }
</SCRIPT>
</PUBLIC:COMPONENT>
```

Прежде всего, для хранения интернет-адреса изначального изображения лучше всего не использовать свойство `src`. Специально для этого мы объявили новое свойство — `primary`.

Для хранения значений свойств `primary` и `hover` мы объявили переменные `iPrimary` и `iHover` соответственно (буква "i" в начале имени переменной означает "internal" — внутренний). Кроме того, мы объявили переменную `isHovered` для хранения признака того, изначальное (значение `false`) или заменяющее (значение `true`) изображение отображается в данный момент. При объявлении мы задали этой переменной значение `false`, поскольку после загрузки страницы всегда отображается изначальное изображение.

Далее мы объявили get- и set-функции для свойств `primary` и `hover` и указали их в соответствующих тегах `<PUBLIC:PROPERTY>`. Также мы задали для этих тегов имена, чтобы можно было вызвать для них метод `fireChange`.

- `getPrimary` — get-функция свойства `primary` — возвращает значение переменной `iPrimary`.

- `setPrimary` — set-функция свойства `primary` — заносит новое значение свойства в переменную `iPrimary` и, если в данный момент отображается изначальное изображение (переменная `isHovered` хранит значение `false`), выводит новое изображение и вызывает метод `fireChange` для тега `<PUBLIC:PROPERTY>`, объявляющего это свойство.
- `getHover` — get-функция свойства `hover` — возвращает значение переменной `iHover`.
- `setHover` — set-функция свойства `hover` — заносит новое значение свойства в переменную `iHover` и, если в данный момент отображается заменяющее изображение (переменная `isHovered` хранит значение `true`), выводит новое изображение и вызывает метод `fireChange` для тега `<PUBLIC:PROPERTY>`, объявляющего это свойство.

Отметим, что для доступа к тегам, объявляющим свойства `primary` и `hover`, мы использовали прямой доступ по имени тега, заданному атрибутом `ID` (подробнее см. в *главе 5*). Это самый простой способ, вдобавок, поскольку мы пишем именно под Internet Explorer, то не должны заботиться о совместимости с другими Web-обозревателями.

Отметим также, что мы удалили небольшой загрузочный сценарий, выводящий изначальное изображение. Дело в том, что в любой момент, когда свойству `primary` или одноименному атрибуту присваивается новое значение, вызывается set-функция этого свойства `setPrimary`, которая и выведет изначальное изображение на страницу. Никаких дополнительных действий для этого нам предпринимать не нужно.

Готовое поведение можно сохранить в файле 16.4.htc и проверить в действии с помощью страницы, чей HTML-код приведен далее.

```
<HTML>
  <HEAD>
    <TITLE>Простейшее поведение</TITLE>
    <STYLE TYPE="text/css">
      .hotimage { behavior: url(16.4.htc); }
    </STYLE>
  </HEAD>
  <BODY>
    <P><A HREF="#"><IMG CLASS="hotimage" PRIMARY="default.jpg"
    HOVER="hover.jpg"></A></P>
  </BODY>
</HTML>
```

Комментировать здесь совершенно нечего, кроме того, что для задания изначального изображения мы использовали атрибут `PRIMARY`, соответствующий созданному нами одноименному свойству.

Мы научились работать со свойствами поведений, принимающими строковые значения. Давайте попробуем создать свойство, значение которого имеет другой тип. Пусть это будет свойство `enabled`, позволяющее разрешить или запретить смену изображений в горячем изображении и принимающее значение логического типа. HTML-код обновленного поведения приведен далее.

```
<PUBLIC:COMPONENT>
  <PUBLIC:ATTACH EVENT="onmouseover" ONEVENT="elementMouseOver();"/>
  <PUBLIC:ATTACH EVENT="onmouseout" ONEVENT="elementMouseOut();"/>
  <PUBLIC:PROPERTY ID="primaryTag" NAME="primary" GET="getPrimary"
  PUT="setPrimary"/>
  <PUBLIC:PROPERTY ID="hoverTag" NAME="hover" GET="getHover"
  PUT="setHover"/>
  <PUBLIC:PROPERTY ID="enabledTag" NAME="enabled" GET="getEnabled"
  PUT="setEnabled"/>
  <SCRIPT>
    var isHovered = false;
    var iPrimary = "";
    var iHover = "";
    var iEnabled = true;

    function elementMouseOver() {
      if (iEnabled) {
        element.src = iHover;
        isHovered = true;
      }
    }

    function elementMouseOut() {
      element.src = element.primary;
      isHovered = false;
    }

    function getPrimary() {
      return iPrimary;
    }

    function setPrimary(pValue) {
```

```
    iPrimary = pValue;
    if (!(isHovered)) element.src = iPrimary;
    primaryTag.fireChange();
  }

  function getHover() {
    return iHover;
  }

  function setHover(pValue) {
    iHover = pValue;
    if (isHovered) element.src = iHover;
    hoverTag.fireChange();
  }

  function getEnabled() {
    return iEnabled;
  }

  function setEnabled(pValue) {
    var newEnabled = true;
    switch (typeof(pValue)) {
      case "string":
        newEnabled = (pValue == "true");
        break;
      case "number":
        newEnabled = (pValue > 0);
        break;
      case "boolean":
        newEnabled = pValue;
        break;
      case "object":
        newEnabled = (pValue != null);
        break;
    }
    iEnabled = newEnabled;
    enabledTag.fireChange();
  }
</SCRIPT>
</PUBLIC:COMPONENT>
```

Мы объявили, помимо самого свойства `enabled`, get-функцию `getEnabled` и set-функцию `setEnabled` и переменную `iEnabled`. Переменная будет хранить значение свойства, get-функция — возвращать его, а set-функция — заносить новое значение, предварительно преобразовав его тип, если это нужно, и вызывать метод `fireChange` тега `<PUBLIC:PROPERTY>`, объявляющего свойство. Здесь все нам знакомо.

В функцию `elementMouseOver`, обрабатывающую событие `onMouseOver` и реализующую смену изначального изображения на заменяющее, мы добавили проверку значения переменной `iEnabled`. Если это значение равно `true`, смена изображений выполняется, если `false` — не выполняется. Отметим, что для функции `elementMouseOut`, обрабатывающей событие `onMouseOut` и реализующей смену заменяющего изображения на изначальное, мы такой проверки не делаем — это позволит вернуть изначальное изображение на место, даже если смена изображений уже запрещена.

Сохраним очередное поведение (его уже с трудом можно назвать простейшим) в файле 16.5.htc. И можем проверить его в действии, воспользовавшись приведенным ранее HTML-кодом тестовой страницы.

Создание методов поведения

Параграф, посвященный созданию свойств поведений, был весьма велик, и это неудивительно — свойства поведений таят много "подводных камней". С методами же все много проще, в чем мы сейчас и убедимся.

Новый метод поведения создается с помощью одинарного тега `PUBLIC:METHOD`.

```
<PUBLIC:METHOD NAME="<имя метода>"
↳[INTERNALNAME="<имя функции, реализующей метод>"]/>
```

И этот тег должен содержать закрывающий символ / перед символом >.

Единственный обязательный атрибут тега `<PUBLIC:METHOD>` — `NAME` — задает имя метода. Это имя будет использоваться в сценариях страницы для вызова метода.

Задав имя метода, мы также должны создать в поведении функцию, которая будет реализовывать этот метод.

- Эта функция должна иметь то же имя, что задано для метода.
- Эта функция должна принимать столько же параметров, сколько принимает их созданный нами метод.
- Эта функция должна возвращать такой же результат, как и созданный нами метод.

Фактически формат вызова этой функции определяет формат вызова создаваемого нами метода.

Необязательный атрибут `INTERNALNAME` позволяет отдельно задать имя функции, реализующей метод. Это может быть полезно, если мы по какой-то причине не можем объявить функцию с тем же именем, что выбрали для метода.

Давайте в качестве примера создадим в нашем поведении 16.5.htc три метода.

- Метод `setHovered` будет принудительно выводить заменяющее изображение. Он не будет принимать параметров и возвращать значения.
- Метод `reset` будет принудительно возвращать на место изначальное изображение. Он не будет принимать параметров и возвращать значения.
- Метод `getState` будет возвращать `true`, если в данный момент выведено заменяющее изображение, и `false` в противном случае. Он не будет принимать параметров.

Собственно, изменений в коде поведения будет крайне мало. Все они представлены далее — это три добавленных тега `PUBLIC:METHOD` и одна добавленная функция.

```
<PUBLIC:COMPONENT>
  . . .
  <PUBLIC:METHOD NAME="setHovered" INTERNALNAME="elementMouseOver"/>
  <PUBLIC:METHOD NAME="reset" INTERNALNAME="elementMouseOut"/>
  <PUBLIC:METHOD NAME="getState"/>
  <SCRIPT>
    . . .
    function getState() {
      return isHovered;
    }
  </SCRIPT>
</PUBLIC:COMPONENT>
```

Для реализации метода `getState` мы объявляем одноименную функцию, которая будет возвращать значение служебной переменной `isHovered`. А для реализации методов `setHovered` и `reset` мы используем давно имеющиеся в коде поведения функции `elementMouseOver` и `elementMouseOut`, обрабатывающие события `onMouseOver` и `onMouseOut`.

Сохраним новое поведение в файле 16.6.htc и проверим его в действии.

Создание событий поведения

Да-да, мы можем добавить в поведение не только свойства и методы, но и события, которые мы впоследствии можем обрабатывать в сценариях страницы обычным способом, рассмотренным в *главе 6*. Сейчас мы узнаем, как это делается.

Событие в поведение добавляется с помощью одинарного тега `PUBLIC:EVENT`.

```
<PUBLIC:EVENT NAME="<имя события>"/>
```

Не забываем, что этот тег также должен содержать закрывающий символ `/` перед символом `>`.

Обязательный атрибут `NAME` задает имя события. Это имя будет использоваться в сценариях страницы для привязки обработчиков к этому событию.

Но объявить событие с помощью тега `PUBLIC:EVENT` — это только первый шаг. Мы должны инициировать возникновение этого события. Именно так — раз мы сами объявили это событие, то сами должны вызвать его возникновение.

Прежде всего, нам нужно найти место в сценариях поведения, где это событие должно возникать. Это может быть какая-то из функций-обработчиков события, изменяющая параметры и (или) содержимое элемента страницы. Обычно такие функции и генерируют события.

Найдя такое место, мы должны вставить в него несколько выражений, которые будут:

- создавать экземпляр объекта `Event`, несущий информацию о событии;
- задавать для этого экземпляра объекта специфические свойства данного события;
- собственно инициировать возникновение события.

Экземпляр объекта `Event` создается вызовом функции `createEventObject`. Эта функция не принимает параметров и возвращает как раз то, что нам нужно, — экземпляр объекта `Event`.

```
var evtObj = createEventObj();
```

Это выражение создает экземпляр объекта, несущий сведения о событии `Event`, и присваивает его переменной `evtObj`.

Созданный таким образом экземпляр объекта `Event` хранит только самые основные сведения о событии в виде значений свойств, описанных в табл. 6.1. Нам может понадобиться создать какие-то свои свойства этого экземпляра и задать для них нужные значения. Для этого достаточно просто присвоить

этим свойствам нужные значения; соответствующие свойства будут тут же созданы.

```
evtObj.state = 1;
```

Это выражение создает в полученном ранее экземпляре объекта `Event` свойство `state` и присваивает ему значение 1.

Для инициирования возникновения события нам остается вызвать метод `fire` тега `PUBLIC:EVENT` (для которого нам придется с помощью атрибута `ID` задать имя).

```
fire(<событие>)
```

Единственным параметром передается событие в виде экземпляра объекта `Event`. Значения этот метод не возвращает.

```
eventTag.fire(evtObj);
```

Это выражение инициирует возникновение созданного ранее события `evtObj`. (`eventTag` — имя тега `PUBLIC:EVENT`, объявляющего событие.)

Давайте для примера еще раз усовершенствуем наше многострадальное поведение. А именно добавим в него событие `onHover`, возникающее при смене изображения. В этом событии мы создадим новое свойство `state`; значение 0 этого свойства будет сигнализировать о том, что отображается изначальное изображение, значение 1 — заменяющее изображение.

Мы не будем приводить HTML-код обновленного поведения целиком — он слишком велик. Приведем только добавленные и исправленные фрагменты.

```
<PUBLIC:COMPONENT>
  . . .
  <PUBLIC:EVENT ID="onHoverTag" NAME="onhover"/>
  <SCRIPT>
    . . .
    function elementMouseOver() {
      if (iEnabled) {
        element.src = iHover;
        isHovered = true;
        var evtObj = createEventObject();
        evtObj.state = 1;
        onHoverTag.fire(evtObj);
      }
    }

    function elementMouseOut() {
```

```
      element.src = element.primary;
      isHovered = false;
      var evtObj = createEventObject();
      evtObj.state = 0;
      onHoverTag.fire(evtObj);
    }
    . . .
  </SCRIPT>
</PUBLIC:COMPONENT>
```

Здесь мы добавили тег `PUBLIC:EVENT` и изменили функции `elementMouseOver` и `elementMouseOut`, выполняющие смену изображений. Эти изменения не требуют комментариев.

Сохраним готовое поведение в файле 16.7.htc. И напишем тестовую страницу. Ее HTML-код приведен далее.

```
<HTML>
  <HEAD>
    <TITLE>Поведение</TITLE>
    <STYLE TYPE="text/css">
      .hotimage { behavior: url(16.7.htc); }
    </STYLE>
    <SCRIPT>
      function imgHover() {
        var outputObj = document.getElementById("output");
        if (event.state == 0)
          outputObj.innerText = "Изначальное изображение"
        else
          outputObj.innerText = "Заменяющее изображение";
      }
    </SCRIPT>
  </HEAD>
  <BODY>
    <P><A HREF="#"><IMG CLASS="hotimage" PRIMARY="default.jpg"
    HOVER="hover.jpg" ONHOVER="imgHover();"></A></P>
    <P ID="output"> </P>
  </BODY>
</HTML>
```

Здесь тоже нечего комментировать.

Программное управление поведениями

Теперь рассмотрим программное управление поведениями — их подключение и отключение.

Для программного подключения поведения к элементу страницы проще всего использовать свойство `behavior` объекта `CSSRule`, соответствующее одноименному атрибуту стиля.

```
<IMG ID="img" PRIMARY="default.jpg" HOVER="hover.jpg">
. . .
<SCRIPT>
  var imgObj = document.getElementById("img");
  img.style.behavior = "url(16.7.htc)";
</SCRIPT>
```

Этот сценарий подключает к графическому изображению `img` поведение, хранящееся в файле 16.7.htc. Отметим, что формат значения свойства `behavior` такой же, как и у значения одноименного атрибута стиля.

Чтобы отключить поведение от элемента страницы, достаточно присвоить свойству `behavior` пустую строку.

```
img.style.behavior = "";
```

Кроме того, объект `HTMLElement` в Internet Explorer поддерживает два метода, позволяющие подключить и отключить поведение. Возможно, в некоторых случаях использовать их будет удобнее.

Метод `addBehavior` подключает поведение к элементу страницы.

```
addBehavior(<интернет-адрес файла поведения>)
```

Единственный параметр этого метода задает интернет-адрес файла, в котором хранится поведение, в виде строки. Метод `addBehavior` возвращает *идентификатор поведения* — целое число, обозначающее подключенное поведение; его можно использовать потом для отключения этого поведения.

```
var bhID = imgObj.addBehavior("16.7.htc");
```

Метод `removeBehavior` отключает подключенное ранее поведение.

```
removeBehavior(<идентификатор поведения>)
```

Единственным параметром этого метода передается идентификатор поведения, возвращенный методом `addBehavior`. Метод `removeBehavior` возвращает `true`, если поведение было благополучно отключено, и `false` в противном случае.

```
imgObj.removeBehavior(bhID);
```

Стандартные поведения Internet Explorer

Internet Explorer поддерживает некоторое количество *стандартных поведений*, определенных в нем самом. Работа с ними выполняется так же, как с поведениями, созданными Web-программистом.

Список стандартных поведений не очень велик, но интересен.

- `anchorClick` — работа с Web-папками. Не используется на обычных Web-страницах.
- `anim` — отображение анимации. Для описания анимации требуется применение технологии HTML+TIME, также поддерживаемой Internet Explorer.
- `clientCaps` — получение сведений о клиентском компьютере. Фактически расширенный аналог объекта `Screen`, описанного в *главе 7*.
- `download` — выполняет загрузку произвольного текстового файла и делает его содержимое доступным из сценариев.
- `homePage` — работа с "домашней" страницей.
- `httpFolder` — работа с Web-папками. Не используется на обычных Web-страницах.
- `mediaBar` — позволяет воспроизводить мультимедийные файлы средствами Проигрывателя Windows Media прямо в окне Internet Explorer.
- `saveFavorite` — позволяет сохранять произвольные данные в ярлыке страницы, помещенном в Избранное.
- `saveHistory` — позволяет сохранять произвольные данные в истории Web-обозревателя.
- `saveSnapshot` — позволяет сохранять произвольные данные в файле страницы на клиентском компьютере.
- `userData` — позволяет сохранять произвольные данные в особом независимом хранилище.

При подключении стандартного поведения к элементу страницы вместо интернет-адреса файла должно использоваться значение вида

```
#default#<имя стандартного поведения>
```

Например:

```
behavior: url(#default#download);
imgObj.style.behavior = "url(#default#download)";
imgObj.addbehavior("#default#download");
```

Все эти три выражения подключают к графическому изображению `imgObj` стандартное поведение `download`.

Полностью рассмотреть стандартные поведения Internet Explorer не получится, так как объем книги ограничен (только рассказ о стандартном поведении `anim` и сопутствующих технологиях потребует целой главы). Их описание можно найти в соответствующем разделе сайта MSDN (**http://msdn2.microsoft.com/en-us/library/ms531081(VS.85).aspx**). Мы же рассмотрим только использование стандартного поведения `download` как представляющего некоторый интерес.

Ранее было сказано, что стандартное поведение `download` выполняет загрузку любого текстового файла. Этот файл может содержать как обычный текст, так и HTML-код, который можно вывести на страницу.

Поведение `download` поддерживает единственный метод `startDownload`, запускающий загрузку файла.

```
startDownload(<интернет-адрес загружаемого файла>,
↳<функция, которая будет вызвана после окончания загрузки>)
```

Первый параметр задает интернет-адрес загружаемого файла в виде строки. Второй параметр задает функцию, которая будет вызвана после загрузки файла; этот параметр имеет тип функции, то есть в нем указывается сама функция.

Функция, вызываемая после загрузки файла, должна удовлетворять двум условиям.

1. Она должна принимать один параметр, значением которого станет содержимое загруженного файла в виде строки.
2. Она не должна возвращать значения.

В качестве примера давайте напишем страницу с двумя гиперссылками; при щелчке на одной из этих гиперссылок будет загружаться соответствующий файл. Один из этих файлов будет содержать текст, другой — фрагмент HTML-кода.

Текстовый файл 16.1.txt будет содержать строку:

```
Это фрагмент страницы, который будет загружен с использованием
стандартного поведения download.
```

Файл с HTML-кодом 16.2.txt будет содержать вот что:

```
Это фрагмент страницы, который будет загружен с использованием
<EM>стандартного поведения</EM> <STRONG>download</STRONG>.
```

Код самой страницы приведен далее.

```
<HTML>
  <HEAD>
    <TITLE>Стандартное поведение download</TITLE>
```

```
    <STYLE TYPE="text/css">
      #hrfText, #hrfHtml { behavior: url(#default#download); }
    </STYLE>
    <SCRIPT>
      function hrfTextClick() {
        var hrfTextObj = document.getElementById("hrfText");
        hrfTextObj.startDownload("16.1.txt", hrfTextDownloadComplete);
      }

      function hrfHtmlClick() {
        var hrfHtmlObj = document.getElementById("hrfHtml");
        hrfHtmlObj.startDownload("16.2.txt", hrfHtmlDownloadComplete);
      }

      function hrfTextDownloadComplete(pContent) {
        var outputObj = document.getElementById("output");
        outputObj.innerText = pContent;
      }

      function hrfHtmlDownloadComplete(pContent) {
        var outputObj = document.getElementById("output");
        outputObj.innerHTML = pContent;
      }
    </SCRIPT>
  </HEAD>
  <BODY>
    <P><A ID="hrfText" HREF="#" ONCLICK="hrfTextClick();">Загрузить
    текст</A></P>
    <P><A ID="hrfHtml" HREF="#" ONCLICK="hrfHtmlClick();">Загрузить
    HTML-код</A></P>
    <P ID="output"></P>
  </BODY>
</HTML>
```

Вряд ли здесь потребуются какие-то комментарии.

Кстати, стандартное поведение `download` Internet Explorer — весьма простой и действенный способ загрузки и отображения фрагментов содержимого страницы "на лету", в ответ на действия посетителя, и неплохая альтернатива технологии AJAX (будет описана в *части IV*). К сожалению, другие Web-обозреватели не могут предложить ничего подобного…

HTML-компоненты

Поведения Internet Explorer очень помогут нам, если мы собираемся создать множество элементов страницы, ведущих себя одинаково в "диалоге" с посетителем. Но Internet Explorer предлагает и более мощную технологию. Это HTML-компоненты, или просто *компоненты*, — дальнейшее развитие поведений, которые, кроме сценариев, содержат еще и код HTML.

Что это может дать? Давайте возьмем за пример наше поведение, обеспечивающее функциональность горячего изображения. Ранее мы создавали на странице гиперссылку, в нее помещали графическое изображение, а к этому графическому изображению подключали поведение. Это поведение содержало сценарии, обрабатывающие события графического изображения и добавляющие поддержку новых свойств, методов и событий. Довольно много было работы…

С переходом на использование HTML-компонентов большая часть этой работы исключается. Мы прямо в поведении создаем необходимый HTML-код, формирующий гиперссылку и графическое изображение, все нужные сценарии, а также объявляем тег, который будет представлять этот HTML-компонент. В самой странице мы сначала выполняем подключение этого HTML-компонента и просто вставляем в нужные места кода страницы определенный для него тег. Все!

HTML-компоненты имеют еще одно преимущество перед поведениями. Весь присутствующий в них HTML-код суть вещь в себе, он не становится частью DOM самой страницы. Это значит, что мы можем в HTML-коде компонента использовать имена тегов без опасений, что они будут совпадать с именами элементов самой страницы. Web-обозреватель не сможет "забраться" внутрь HTML-кода компонента, для него компонент — это "черный ящик", из которого "торчат" только объявленные разработчиком свойства, методы и события.

Как и поведения, HTML-компоненты сохраняются в отдельных текстовых файлах с расширением htc. И создаются они примерно по таким же правилам, как поведения.

Создание HTML-компонентов

Сказанное ранее означает, что HTML-компонент создается с помощью уже знакомого нам парного тега `<PUBLIC:COMPONENT>`. Только формат его записи несколько иной.

```
<PUBLIC:COMPONENT TAGNAME="<имя тега HTML-компонента>">
  <объявление событий и их обработчиков>
```

```
  [<объявление свойств, методов и событий HTML-компонента>]
  <PUBLIC:DEFAULTS VIEWLINKCONTENT="true"
  ↳[<другие дополнительные параметры HTML-компонента>]/>
  <SCRIPT>
    <код обработчиков событий и прочих сценариев>
  </SCRIPT>
  <BODY>
    <HTML-код компонента>
  </BODY>
</PUBLIC:COMPONENT>
```

Первое, что бросается в глаза, — это наличие парного тега `<BODY>`. Внутри этого тега помещается HTML-код, формирующий компонент. Правила здесь такие же, как и в случае секции тела страницы: пишем теги, задаем атрибуты и следим, чтобы не было ошибок.

Идем далее и видим, что здесь появился один новый атрибут и один новый тег. Давайте рассмотрим их.

Обязательный в случае компонента атрибут `TAGNAME` тега `<PUBLIC:COMPONENT>` задает имя тега, который будет представлять в коде страницы наш компонент. Это имя не должно совпадать с именами уже существующих тегов, что, впрочем, понятно.

Одинарный тег `<PUBLIC:DEFAULTS>` является обязательным в случае компонента и задает его дополнительные параметры. Он должен присутствовать в теге `<PUBLIC:COMPONENT>` только в одном экземпляре. Отметим, что этот тег должен содержать закрывающий символ `/` перед символом `>`.

В теге `<PUBLIC:DEFAULTS>` нас интересует только обязательный в случае компонента атрибут `VIEWLINKCONTENT`. Значение `true` этого атрибута указывает Internet Explorer, что данный код определяет именно HTML-компонент как "черный ящик".

> ***НА ЗАМЕТКУ***
>
> Если задать для атрибута `VIEWLINKCONTENT` тега `<PUBLIC:DEFAULTS>` значение `false` или вообще не указать этот атрибут, HTML-код компонента будет включен Internet Explorer в DOM страницы и перестанет быть "черным ящиком". Это далеко не всегда удобно.

Тег `<PUBLIC:DEFAULTS>` поддерживает еще довольно много атрибутов, но мы пока их не будем рассматривать, а отложим на потом. Давайте лучше уясним правила написания HTML-компонентов. Их немного, но они очень важны.

- Для обращения к тегам HTML-кода компонента используются средства, описанные в *главе 5*. Лучше всего использовать прямой доступ по име-

нам — поскольку мы пишем под Internet Explorer, необязательно заботиться о совместимости с другими Web-обозревателями.

- Обработчики событий привязываются непосредственно к тегам HTML-кода компонента. Тег `<PUBLIC:ATTACH>` для этого использовать не допускается.
- Скорее всего, придется объявить свойства компонента, соответствующие свойствам отдельных тегов, его составляющих. (Например, для нашего горячего изображения это будут свойства `href` и `target`, соответствующие одноименным свойствам тега `<A>`.) Для их реализации придется использовать get- и set-функции — другого способа получить их значения и дать им новые просто нет.

Собственно, больше рассказывать нечего. Давайте лучше попрактикуемся и перепишем поведение 16.7.htc в HTML-компонент. Зададим для него имя тега `HOTIMG`.

```
<PUBLIC:COMPONENT TAGNAME="HOTIMG">
  <PUBLIC:PROPERTY ID="primaryTag" NAME="primary" GET="getPrimary"
  PUT="setPrimary"/>
  <PUBLIC:PROPERTY ID="hoverTag" NAME="hover" GET="getHover"
  PUT="setHover"/>
  <PUBLIC:PROPERTY ID="enabledTag" NAME="enabled" GET="getEnabled"
  PUT="setEnabled"/>
  <PUBLIC:PROPERTY ID="hrefTag" NAME="href" GET="getHref" PUT="setHref"/>
  <PUBLIC:PROPERTY ID="targetTag" NAME="target" GET="getTarget"
  PUT="setTarget"/>
  <PUBLIC:METHOD NAME="setHovered" INTERNALNAME="imgMouseOver"/>
  <PUBLIC:METHOD NAME="reset" INTERNALNAME="imgMouseOut"/>
  <PUBLIC:METHOD NAME="getState"/>
  <PUBLIC:EVENT ID="onHoverTag" NAME="onhover"/>
  <PUBLIC:DEFAULTS VIEWLINKCONTENT="true"/>
  <SCRIPT>
    var isHovered = false;
    var iPrimary = "";
    var iHover = "";
    var iEnabled = true;

    function imgMouseOver() {
      if (iEnabled) {
        img.src = iHover;
```

```
    isHovered = true;
    var evtObj = createEventObject();
    evtObj.state = 1;
    onHoverTag.fire(evtObj);
  }
}

function imgMouseOut() {
  img.src = iPrimary;
  isHovered = false;
  var evtObj = createEventObject();
  evtObj.state = 0;
  onHoverTag.fire(evtObj);
}

function getPrimary() {
  return iPrimary;
}

function setPrimary(pValue) {
  iPrimary = pValue;
  if (!(isHovered)) img.src = iPrimary;
  primaryTag.fireChange();
}

function getHover() {
  return iHover;
}

function setHover(pValue) {
  iHover = pValue;
  if (isHovered) img.src = iHover;
  hoverTag.fireChange();
}

function getEnabled() {
  return iEnabled;
}
```

```
function setEnabled(pValue) {
  var newEnabled = true;
  switch (typeof(pValue)) {
    case "string":
      newEnabled = (pValue == "true");
      break;
    case "number":
      newEnabled = (pValue > 0);
      break;
    case "boolean":
      newEnabled = pValue;
      break;
    case "object":
      newEnabled = (pValue != null);
      break;
  }
  iEnabled = newEnabled;
  enabledTag.fireChange();
}

function getHref() {
  return hrf.href;
}

function setHref(pValue) {
  hrf.href = pValue;
  hrefTag.fireChange();
}

function getTarget() {
  return hrf.target;
}

function setTarget(pValue) {
  hrf.target = pValue;
  targetTag.fireChange();
}

function getState() {
```

```
      return isHovered;
    }
  </SCRIPT>
  <BODY>
    <A ID="hrf"><IMG ID="img" ONMOUSEOVER="imgMouseOver();"
    ONMOUSEOUT="imgMouseOut();"></A>
  </BODY>
</PUBLIC:COMPONENT>
```

Наш компонент будет иметь, помимо свойств, присутствовавших еще в поведении 16.7.htc, свойства `href` и `target`, соответствующие одноименным свойствам тега `<A>`. Для новых свойств мы объявили get- и set-функции, которые просто возвращают значения соответствующих свойств тега `<A>` и заносят в них новые значения.

Остальной код комментариев не требует. Поэтому сохраним его в файле под именем 16.8.htc.

Можно было, конечно, объявить в компоненте и другие свойства, например, свойство `alt`, соответствующее свойству `alt` тега `<IMG>`. Но автор не стал этого делать, чтобы не перегружать и без того немалый код. Думается, читатели без особого труда смогут сделать это самостоятельно.

Использование HTML-компонентов

Использовать созданные компоненты в страницах очень просто. Но для этого придется выполнить два действия, которые сейчас мы рассмотрим.

Первое действие — объявление пространства имен, в котором будет присутствовать компонент. *Пространство имен* задает своего рода сверхимя для имен тегов, соответствующих компонентам. Оно предусмотрено на тот случай, если на странице используются сразу несколько разных компонентов, имена тегов которых совпадают. Такие компоненты должны быть помещены в разные пространства имен — в этом случае конфликта имен тегов не возникнет. Но даже если мы используем на странице один-единственный компонент, пространство имен все равно должно быть задано.

Пространство имен задается обязательным в случае использованием компонентов атрибутом `XMLNS` тега `<HTML>`. Да-да, того самого "глобального" тега, объединяющего весь HTML-код страницы! В качестве его значения указывается имя создаваемого пространства имен. Имена эти лучше всего задавать большими буквами.

```
<HTML XMLNS="SAMPLE">
```

Здесь мы создали пространство имен `SAMPLE`, в которое поместим наш единственный компонент `HOTIMG`.

Если нужно задать сразу несколько пространств имен, атрибут указывается нужное число раз XMLNS.

```
<HTML XMLNS="SAMPLE1" XMLNS="SAMPLE2" XMLNS="TESTING">
```

Второе действие — подключение к странице файла с HTML-компонентом и помещение его в одно из созданных пространств имен. Это выполняется с помощью одинарного тега <?IMPORT> (символ вопросительного знака после символа < обязателен). Данный тег должен присутствовать в секции заголовка страницы (в теге <HEAD>).

```
<?IMPORT NAMESPACE="<имя пространства имен, соответствующее компоненту>"
↳IMPLEMENTATION="<интернет-адрес файла HTML-компонента>">
```

Обязательный атрибут NAMESPACE задает пространство имен, в которое помещается компонент. А обязательный атрибут IMPLEMENTATION задает интернет-адрес файла, где хранится компонент.

```
<?IMPORT NAMESPACE="SAMPLE" IMPLEMENTATION="16.8.htc">
```

Этот тег подключает к странице созданный нами ранее компонент 16.8.htc (тег HOTIMG) и помещает его в пространство имен SAMPLE.

После всего этого можно использовать созданный нами компонент, как и любой другой тег, чья поддержка встроена прямо в Internet Explorer. Для этого достаточно вставить в нужное место HTML-кода страницы заданный для компонента тег. Имя этого тега должно записываться в таком формате:

```
<<имя пространства имен>:<имя тега HTML-компонента> [<атрибуты>]>
</<имя пространства имен>:<имя тега HTML-компонента>>
```

если тег парный, или в формате

```
<<имя пространства имен>:<имя тега HTML-компонента> [<атрибуты>]/>
```

если тег одинарный. Отметим, что в последнем случае перед закрывающим символом > обязательно должен присутствовать символ /.

```
<SAMPLE:HOTIMG HREF="16.1.htm" PRIMARY="default.jpg" HOVER="hover.jpg"/>
```

Этот HTML-код помещает на страницу компонент 16.8.htc. Обратим внимание на имя тега компонента — оно должно соответствовать приведенному ранее формату — и на то, что перед закрывающим символом > должен присутствовать символ /, так как тег HOTIMG одинарный.

Далее полностью приведен HTML-код тестовой страницы для компонента 16.8.htc. Он не требует никаких комментариев.

```
<HTML XMLNS:SAMPLE>
  <HEAD>
    <TITLE>HTML-компонент</TITLE>
    <?IMPORT NAMESPACE="SAMPLE" IMPLEMENTATION="16.8.htc">
```

```
  </HEAD>
  <BODY>
    <P><SAMPLE:HOTIMG HREF="otherpage.htm" PRIMARY="default.jpg"
    HOVER="hover.jpg"/></P>
  </BODY>
</HTML>
```

HTML-компоненты — замечательная вещь. Но вдвойне замечательной она станет, когда ее возьмут на вооружение разработчики Opera и Firefox.

Дополнительные параметры HTML-компонента

Мы собирались более подробно рассмотреть тег `<PUBLIC:DEFAULTS>`, задающий дополнительные параметры компонента. Давайте это сделаем.

Формат написания этого тега приведен далее.

```
<PUBLIC:DEFAULTS VIEWLINKCONTENT="true" [STYLE="<стиль HTML-компонента>"]
[TABSTOP="true|false"] [VIEWINHERITSTYLE="true|false"]
[VIEWMASTERTAB="true|false"]/>
```

Рассмотрим атрибуты этого тега, которые нам пока еще не знакомы.

Необязательный атрибут `STYLE` задает стиль для компонента. Понятно, что это встроенный стиль.

Необязательный атрибут `TABSTOP` позволяет указать, будет ли данный компонент активизироваться при использовании клавиш <Tab> и <Shift>+<Tab>. Если дать этому атрибуту значение "true", компонент будет активизироваться, а если дать значение "false" или вообще не указывать этот атрибут, то не будет.

Необязательный атрибут `VIEWINHERITSTYLE` задает, будет ли компонент, помещенный в страницу, наследовать от родителя значения некоторых атрибутов стиля. Если дать ему значение "true" или вообще не указать этот атрибут, компонент должен наследовать значения атрибутов стиля; значение "false" отменяет наследование.

Необязательный атрибут `VIEWMASTERTAB` будет полезен, если компонент содержит элементы управления. Если дать ему значение "true" или вообще не указать его, элементы управления, содержащиеся в компоненте, будут включены в порядок обхода страницы (о порядке обхода говорилось в *главе 12*). Значение "false" предписывает Internet Explorer создать для данного компонента свой собственный, независимый от страницы порядок обхода.

Тег `<PUBLIC:DEFAULTS>` поддерживает еще несколько атрибутов, но их полезность довольно сомнительна.

Программное управление HTML-компонентами

Internet Explorer позволяет добавлять на страницу HTML-компоненты программно. Но, к сожалению, для этого можно использовать только свойства `innerHTML` и `outerHTML` — при использовании для этой цели методов DOM происходит что-то странное…

Далее приведен HTML-код страницы, на которую программно добавляется компонент 16.8.htc.

```
<HTML XMLNS:SAMPLE>
  <HEAD>
    <TITLE>HTML-компонент</TITLE>
    <?IMPORT NAMESPACE="SAMPLE" IMPLEMENTATION="16.8.htc">
  </HEAD>
  <BODY>
    <SCRIPT>
      var s = "<P><SAMPLE:HOTIMG HREF='16.1.htm' PRIMARY='default.jpg'
      ↳HOVER='hover.jpg'></P>"
      document.body.innerHTML = s;
    </SCRIPT>
  </BODY>
</HTML>
```

Больше об HTML-компонентах рассказывать нечего.

Что дальше?

Поведения и HTML-компоненты Internet Explorer позволят нам заметно облегчить работу. Мы можем написать весьма сложный HTML- и JavaScript-код один раз, а потом использовать его на разных страницах сколько угодно раз. Думается, Web-программисты это оценят, ведь они ленивые люди…

На этом мы прощаемся с Internet Explorer и приветствуем Firefox. Этот весьма примечательный Web-обозреватель поддерживает особый элемент страницы под названием "канва", который позволит нам рисовать прямо на странице. Так что художественно настроенным натурам — приготовиться!

Глава 17

Рисование на Web-странице (Firefox)

Что ж, настала пора обратиться к другому Web-обозревателю, также предлагающему интересные специфические возможности, — Firefox. И хоть он в этом смысле заметно беднее своего "старшего коллеги" Internet Explorer, поговорить о нем стоит.

Еще в *главе 2* мы узнали, как поместить на страницу графическое изображение, а в последующих главах активно в этом практиковались. Мы выяснили, что HTML позволяет проделать с изображением довольно многое: задать его параметры с помощью стилей CSS, текст замены, создать на его основе изображение-гиперссылку и карту-изображение. А если применить немного JavaScript, то получится горячее изображение, с которыми мы также познакомились.

Единственное, чего не позволяет сделать HTML, — это нарисовать что-либо прямо на странице. Текущий стандарт данного языка не предусматривает никаких средств для этого. И JavaScript нам в этом деле не помощник.

Разработчики Firefox возмутились подобным положением вещей и решили-таки помочь Web-дизайнерам и Web-программистам, чувствующим в себе талант художника. Они добавили в свое детище поддержку особого тега, создающего элемент страницы, как раз и предназначенный для программного создания произвольных рисунков. Этот тег и соответствующий ему объект поддерживает обширный набор методов, позволяющих рисовать линии и простейшие фигуры и закрашивать их любым цветом.

Конечно, картины мы с его помощью создать не сможем. Но для вывода простых рисунков, схем и графиков, генерируемых программно, этого вполне достаточно. Большего обычно и не требуется.

В этой главе мы изучим "художественный" тег Firefox и все его возможности. И, разумеется, попрактикуемся.

Канва

Для программного создания произвольных рисунков в Firefox используется особый элемент страницы, называемый *канвой*. Мы можем рассматривать канву как своего рода холст, на котором мы в качестве художника и будем творить свои произведения.

Канва создается с помощью парного тега `<CANVAS>`. Это тот самый "художественный" тег, о котором мы упоминали.

```
<CANVAS ID="<имя>" [WIDTH="<ширина>"] [HEIGHT="<высота>"]</CANVAS>
```

Тег `<CANVAS>` может иметь любое содержимое. Web-обозреватели, поддерживающие канву, проигнорируют содержимое этого тега и выведут на страницу только саму канву в виде пустого пространства, не занятого ничем. Web-обозреватели, не поддерживающие канву, наоборот, проигнорируют тег `<CANVAS>` как неизвестный им и выведут на страницу его содержимое. Этим можно воспользоваться, чтобы предупредить пользователей таких Web-обозревателей о том, что они не поддерживают канву.

Мы обязательно должны задать имя тега `<CANVAS>` с помощью атрибута `ID`. Поскольку рисовать в нем можно только программно, нам придется получить к нему доступ, и проще всего это сделать через имя тега.

Необязательные атрибуты `WIDTH` и `HEIGHT` задают, соответственно, ширину и высоту канвы в пикселах. По умолчанию канва имеет размеры 300×150 пикселов.

> ***Внимание!***
>
> Разработчики Firefox не рекомендуют задавать размеры канвы с помощью стилей CSS.

```
<CANVAS ID="cnv" WIDTH="400" HEIGHT="300">
  <P>Извините, но ваш Web-обозреватель не поддерживает канву.</P>
</CANVAS>
```

Этот HTML-код создаст на странице канву `cnv` размером 400×300 пикселов. Те Web-обозреватели, что не поддерживают канву, выведут на страницу абзац с предупреждающим текстом.

Канва представляется как экземпляр объекта `HTMLCanvasElement`. Для нас будут полезны только свойства `width` и `height` этого объекта, соответствующие одноименным атрибутам, и единственный его метод, 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, применяемых к данной канве для вывода на ней графики.

Это значит, что перед тем как начать рисование, нам придется как-то получить экземпляр объекта `CanvasRenderingContext2D` для данной канвы. Это выполняется вызовом единственного метода `getContext` объекта `HTMLCanvasElement`.

```
getContext("2d")
```

Мы видим, что этот метод принимает единственный параметр — строку "2d". Возвращает он то, что нам нужно, — экземпляр объекта `CanvasRenderingContext2D`, представляющий контекст рисования данной канвы.

```
var cnvObj = document.getElementById("cnv");
var ctxCnvObj = cnvObj.getContext("2d");
```

Этот небольшой сценарий помещает в переменную `ctxCnvObj` контекст рисования для ранее созданной канвы `cnv`. Впоследствии мы будем пользоваться этим контекстом рисования для наших примеров.

Вот теперь, вооружившись контекстом рисования канвы, мы можем начать рисовать на ней. Это выполняется с помощью весьма многочисленных свойств и методов объекта `CanvasRenderingContext2D`, рассмотрению которых будет посвящена вся эта глава.

При выполнении операций рисования нам будет нужно задавать координаты точек, в которых будет начинаться и заканчиваться рисование фигур и пр. Эти 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. Это рисование различных прямоугольников, с заливкой и без нее.

Для рисования прямоугольника без заливки (то есть одного лишь контура прямоугольника) используется метод `strokeRect` объекта `CanvasRenderingContext2D`.

(В дальнейшем все рассматриваемые свойства и методы будут относиться к этому объекту, если в тексте не сказано иное.)

```
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 значения.

```
ctxCnvObj.strokeRect(20, 20, 360, 260);
```

Метод `fillRect` рисует прямоугольник с заливкой.

```
fillRect(<горизонтальная координата>, <вертикальная координата>,
↳<ширина>, <высота>)
```

Как видим, формат его вызова такой же, как у метода `strokeRect`.

```
ctxCnvObj.fillRect(40, 40, 320, 220);
```

Весьма полезный для создания сложных фигур метод `clearRect` очищает заданную прямоугольную область от любой присутствовавшей там графики.

```
clearRect(<горизонтальная координата>, <вертикальная координата>,
↳<ширина>, <высота>)
```

И его формат вызова схож с форматом вызова метода `strokeRect`.

```
ctxCnvObj.fillRect(0, 0, 400, 300);
ctxCnvObj.clearRect(100, 100, 200, 100);
```

Этот сценарий рисует большой прямоугольник с заливкой, занимающий всю канву `cnv`, после чего создает в ее середине прямоугольную "прореху".

```
ctxCnvObj.clearRect(0, 0, 400, 300);
```

А этот сценарий очищает канву от всей графики, что на ней присутствует.

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. Цвет этот задается в виде строки 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 задаются в виде 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 задаются в виде десятичных чисел от 0 до 255. Уровень прозрачности задается в виде числа от 0.0 (полностью прозрачный) до 1.0 (полностью непрозрачный).

```
ctxCnvObj.strokeStyle = "#FF0000";
ctxCnvObj.strokeStyle = "rgb(255, 0, 0)";
ctxCnvObj.strokeStyle = "rgb(255, 0, 0, 1)";
```

Все эти три выражения задают непрозрачный красный цвет линий контура.

```
ctxCnvObj.strokeStyle = "rgb(0, 0, 0, 0.5)";
```

А это выражение задает для линий контура полупрозрачный черный цвет.

Отметим одну важную вещь. Если мы зададим цвет контура, то все фигуры, которые мы впоследствии нарисуем, будут иметь контур данного цвета. Чтобы впоследствии нарисовать фигуру с контуром другого цвета, мы должны будем задать нужный цвет контура перед их рисованием.

Изначально, сразу после загрузки и вывода канвы на страницу, линии контура будут иметь черный цвет.

Свойство `fillStyle` задает 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.

```
ctxCnvObj.fillStyle = "rgb(0, 127, 0)";
```

Это выражение задает тускло-зеленый непрозрачный цвет заливки.

```
with (ctxCnvObj) {
  strokeStyle = "rgba(255, 0, 0, 1)";
  fillStyle = "rgba(255, 0, 0, 1)";
  fillRect(0, 100, 400, 100);
  strokeStyle = "rgba(0, 255, 0, 0.5)";
  fillStyle = "rgba(0, 255, 0, 0.5)";
```

```
  fillRect(100, 0, 200, 300);
}
```

Этот сценарий рисует прямоугольник с заливкой, используя и для контура, и для заливки непрозрачный красный цвет, после чего поверх него рисует прямоугольник с заливкой, но уже используя для контура и заливки полупрозрачный зеленый цвет. 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 сценарии.

Свойство `globalAlpha`, возможно, также нам пригодится. Оно позволяет задать уровень прозрачности для любой графики, что мы впоследствии нарисуем. Уровень прозрачности также задается в виде числа от 0.0 (полностью прозрачный) до 1.0 (полностью непрозрачный).

```
ctxCnvObj.globalAlpha = 0.1;
```

Это выражение задаст для всей графики, которую мы потом нарисуем на канве, уровень прозрачности 10%.

Свойство `lineWidth` задает толщину линий в пикселах в виде числа.

```
ctxCnvObj.lineWidth = 20;
ctxCnvObj.strokeRect(20, 20, 360, 260);
```

Этот сценарий рисует прямоугольник без заливки линиями толщиной в 20 пикселов.

Рисование сложных фигур

Firefox также поддерживает рисование более сложных, чем прямоугольники, фигур, чьи контуры состоят из множества прямых и кривых линий. Сейчас мы выясним, как это делается.

Как рисуются сложные контуры

Контуры сложных фигур рисуются в три этапа.

1. Firefox ставится в известность, что сейчас начнется рисование контура сложной фигуры.
2. Рисуются отдельные линии, прямые и кривые, составляющие сложный контур.
3. Firefox ставится в известность, что рисование контура закончено, и теперь фигура должны быть выведена на канву, возможно, с созданием заливки. Также можно указать Firefox, что следует замкнуть нарисованный контур.

Начало рисования сложного контура обозначается вызовом метода `beginPath`. Этот метод не принимает параметров и не возвращает результата.

Собственно рисование линий, составляющих сложный контур, выполняется вызовом особых методов того же объекта `CanvasRenderingContext2D`. Мы рассмотрим эти методы потом.

После окончания рисования сложного контура мы можем захотеть, чтобы Firefox его замкнул. Это выполняется вызовом метода `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 этих метода не принимают параметров и не возвращают результата.

А теперь рассмотрим методы, используемые для рисования разнообразных линий, составляющих сложный контур.

Перо. Перемещение пера

Для рисования сложного контура используется особый инструмент под названием *перо*. 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 страницы и вывода канвы, перо находится в точке с координатами [0,0], то есть в верхнем левом углу канвы. Чтобы переместить перо в другую точку канвы, где мы собираемся начать рисование контура, мы используем особый метод объекта `CanvasRenderingContext2D` — `moveTo`.

```
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 значения.

```
ctxCnxObj.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`.

`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`. Этот 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.

```
with (ctxCnvObj) {
  beginPath();
  moveTo(20, 20);
  lineTo(380, 20);
  lineTo(200, 280);
  closePath();
  stroke();
}
```

Этот сценарий рисует треугольник без заливки. Давайте его рассмотрим.

1. Сообщаем Firefox, что собираемся рисовать контур сложной фигуры, вызовом метода `beginPath`.
2. Устанавливаем перо в точку, где начнется рисование. Для этого используется метод `moveTo`.
3. Рисуем две линии, которые станут сторонами треугольника, с помощью метода `lineTo`.
4. Третью сторону мы рисовать не будем, а лучше вызовем метод `closePath`, чтобы Firefox сам нарисовал ее, закрыв тем самым нарисованный нами контур.
5. Вызываем метод `stroke`, чтобы закончить рисование треугольника без заливки.

Дуги

Дуги рисуются тоже довольно просто. Для этого используется метод `arc`.

`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 задает 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 значения.

Здесь нам нужно отметить две вещи. Во-первых, начальный и конечный углы рисуемой дуги задаются в радианах, а не в градусах. Для пересчета величины угла из градусов в радианы нам придется использовать 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.

Во-вторых, при рисовании дуги мы явно задаем точку, где будет находиться ее центр. Перо в этом случае не используется. Более того, автору не удалось установить, куда переместится перо после рисования дуги. Так что включать дуги в сложные контуры следует с осторожностью — кто знает, что мы получим в результате…

```
with (ctxCnvObj) {
  beginPath();
  arc(200, 150, 100, 0, Math.PI * 2, false);
  stroke();
}
```

Этот сценарий рисует окружность без заливки. Отметим, какие параметры метода `arc`, в частности, значения начального и конечного угла, мы задавали в этом случае.

Вообще, инструменты Firefox для рисования дуг не настолько удобны, как хотелось бы. В частности, было бы желательно получить возможность рисовать секторы окружности, но с помощью этих инструментов сделать их не получится, так как неизвестно, в какой точке после рисования дуги окажется перо, а значит, мы не сможем нарисовать прямые, ограничивающие сектор. Придется использовать кривые Безье, о которых мы сейчас поговорим.

Кривые Безье

Кривые Безье — это кривые 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.

Посмотрим на рис. 17.1. Там мы видим кривую Безье (толстая линия) и ее начальную и конечную точки (обозначены кружками). Кроме того, там мы видим контрольные точки (обозначены квадратиками). Через каждую контрольную точку и начальную и конечную точку 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.

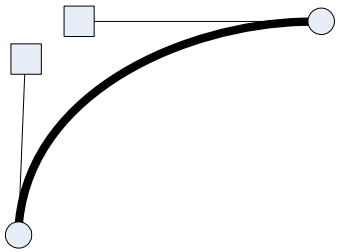

Рис. 17.1. Кривая Безье с двумя контрольными точками

На рис. 17.1 представлена обычная кривая Безье с двумя контрольными точками. Такие кривые применяются чаще всего.

Но зачастую предпочтительнее использовать другую, "вырожденную", форму кривых Безье — с одной контрольной точкой. Такая кривая представлена на рис. 17.2.

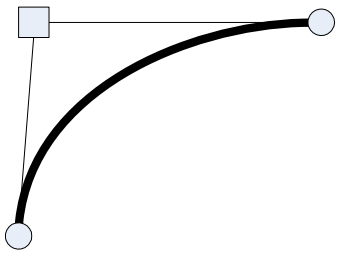

Рис. 17.2. Кривая Безье с одной контрольной точкой

Кривые Безье с одной контрольной точкой могут использоваться для создания дуг. С их помощью легко рисовать секторы, в чем мы вскоре убедимся.

Для рисования кривых Безье с двумя контрольными точками используется метод `bezierCurveTo`.

```
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. Результата этот метод не возвращает.

Рисование кривой Безье начинается в той точке, где в данный момент установлено перо. После рисования кривой перо устанавливается в ее конечную точку.

```
with (ctxCnvObj) {
  beginPath();
  moveTo(100, 100);
  bezierCurveTo(120, 80, 160, 20, 100, 200);
  stroke();
}
```

Этот сценарий рисует кривую Безье с двумя контрольными точками.

Для рисования кривых Безье с одной контрольной точкой используется метод `quadraticCurveTo`.

```
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. Результата этот метод не возвращает.

Рисование такой кривой Безье также начинается в той точке, где в данный момент установлено перо. После рисования кривой перо устанавливается в ее конечную точку.

```
with (ctxCnvObj) {
  beginPath();
  moveTo(100, 100);
  quadraticCurveTo(200, 100, 200, 200);
```

```
  stroke();
}
```

Этот сценарий рисует кривую Безье с одной контрольной точкой. Получившаяся кривая будет иметь вид дуги.

```
with (ctxCnvObj) {
  beginPath();
  strokeStyle = "rgb(255, 0, 0)";
  fillStyle = "rgb(255, 0, 0)";  moveTo(100, 100);
  quadraticCurveTo(200, 100, 200, 200);
  lineTo(100, 200);
  lineTo(100, 100);
  fill();
}
```

А этот сценарий 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.

```
with (ctxCnvObj) {
  beginPath();
  moveTo(20, 0);
  lineTo(180, 0);
  quadraticCurveTo(200, 0, 200, 20);
  lineTo(200, 80);
  quadraticCurveTo(200, 100, 180, 100);
  lineTo(20, 100);
  quadraticCurveTo(0, 100, 0, 80);
  lineTo(0, 20);
  quadraticCurveTo(0, 0, 20, 0);
  stroke();
}
```

Этот сценарий 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 использовать метод `rect`.

```
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 также в виде чисел. Метод `rect` не возвращает значения.

После рисования прямоугольника методом `rect` перо устанавливается в точку с координатами [0,0], то есть в верхний левый угол канвы.

```
with (ctxCnvObj) {
  beginPath();
  rect(50, 50, 50, 50);
  rect(75, 75, 50, 50);
  rect(100, 100, 50, 50);
  fill();
}
```

Этот сценарий рисует сложную фигуру, состоящую их трех накладывающихся друг на друга квадратов, и создает для нее заливку.

Задание стиля линий

Firefox 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.

```
with (ctxCnvObj) {
  beginPath();
  lineWidth = 10;
  lineCap = "round";
  moveTo(20, 20);
```

```
  lineTo(180, 20);
  stroke();
}
```

Этот сценарий рисует толстую прямую линию, начальная и конечная точки которой имеют вид кружков.

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.

```
with (ctxCnvObj) {
  beginPath();
  lineWidth = 10;
  lineJoin = "bevel";
  moveTo(20, 20);
  lineTo(380, 20);
  lineTo(200, 280);
  closePath();
  stroke();
}
```

Этот сценарий рисует контур треугольника толстыми линиями, причем точки соединения этих линий, образующие острые углы, будут срезаться.

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 этого свойства задается в пикселах в виде числа. Каково его значение по умолчанию, автору выяснить не удалось.

```
with (ctxCnvObj) {
  beginPath();
  lineJoin = "miter";
  lineWidth = 10;
  miterLimit = 1;
  moveTo(20, 20);
  lineTo(380, 20);
```

```
  lineTo(200, 280);
  closePath();
  stroke();
}
```

Это исправленный вариант сценария, приведенного ранее. Дистанция, на которую могут выступать острые углы, образованные соединением линий контура, не должна превышать одного пиксела. Углы, выступающие на бóльшую дистанцию, будут срезаны.

Использование сложных цветов

Ранее мы использовали для линий и заливок только *простые*, однотонные, цвета. Настала пора познакомиться со средствами Firefox для создания и использования *сложных* цветов: градиентных и графических.

Линейный градиентный цвет

В *линейном градиентном* цвете (или просто *линейном градиенте*) один простой цвет перетекает 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.

Создание линейного градиентного цвета выполняется в три этапа. Сейчас мы их рассмотрим.

Первый этап — собственно создание линейного градиентного цвета. Это выполняется вызовом метода `createLinearGradient`.

```
createLinearGradient(<горизонтальная координата начальной точки>,
↳<вертикальная координата начальной точки>,
↳<горизонтальная координата конечной точки>,
↳<вертикальная координата конечной точки>)
```

Параметры этого метода задают 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 обязательно должны сохранить его в какой-либо переменной, иначе не сможем задать для него нужные параметры.

```
var lgObj = ctxCnvObj.createLinearGradient(0, 0, 100, 50);
```

Это выражение создает линейный градиент, который будет "распространяться" по прямой с координатами начальной и конечной точек [0,0] и [100,50], и помещает его в переменную `lgObj`.

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.

Ключевая точка ставится вызовом метода `addColorStop` объекта `CanvasGradient`.

```
addColorStop(<положение ключевой точки>, <цвет>)
```

Первый параметр задает 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 можно использовать все форматы описания цвета, упомянутые в начале этой главы.

Метод `addColorStop` не возвращает значения.

```
lgObj.addColorStop(0, "#000000");
lgObj.addColorStop(0.4, "rgba(0, 0, 255, 0.5)");
lgObj.addColorStop(1, "#FF0000");
```

Этот 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.

```
ctvCnvObj.fillStyle = lgObj;
```

Вообще, как правило, градиенты, и линейные, и радиальные (их мы рассмотрим позже), применяются к заливкам. Так они выглядят эффектнее.

А теперь нам нужно рассмотреть один очень важный вопрос. И рассмотрим мы его на примере созданного ранее градиента.

Давайте 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). То есть 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.

То есть созданный нами градиент не "втиснулся" в каждый из нарисованных прямоугольников целиком, а как бы зафиксировался на самой канве, а прямоугольники только "проявили" его фрагменты, соответствующие размерам этих прямоугольников. Можно сказать, что градиент задается для целой канвы, а фигуры, к которым он применен, окрашиваются соответствующими его фрагментами.

Так что, если мы хотим, чтобы какая-то фигура была окрашена градиентом полностью, нам придется задать для этого градиента такие координаты воображаемой прямой, чтобы он "покрыл" всю фигуру. Иначе фигура будет содержать только часть градиента.

```
var lgObj = ctxCnvObj.createLinearGradient(0, 0, 100, 50);
lgObj.addColorStop(0, "#000000");
lgObj.addColorStop(0.4, "rgba(0, 0, 255, 0.5)");
lgObj.addColorStop(1, "#FF0000");
ctxCnvObj.fillStyle = lgObj;
ctxCnvObj.fillRect(0, 0, 200, 100);
```

Этот сценарий рисует прямоугольник и заполняет его линейным градиентом, аналогичным тому, что мы создали ранее в этом параграфе. 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 окружностями: внутренней, меньшего радиуса, и внешней, большего радиуса.

При движении от границы внутренней окружности к границе внешней один простой цвет перетекает в другой.

Радиальный градиентный цвет также создается в три этапа, которые мы сейчас рассмотрим.

Первый этап — создание радиального градиентного цвета. Это выполняется вызовом метода `createRadialGradient`.

```
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 обоих окружностей, описывающих радиальный градиент. Они задаются в пикселах в виде чисел.

Метод `createRadialGradient` возвращает экземпляр объекта `CanvasGradient`, представляющий созданный нами радиальный градиент. Мы обязательно должны сохранить его в какой-либо переменной, иначе не сможем задать для него нужные параметры.

```
var rgObj = ctxCnvObj.createRadialGradient(100, 100, 10, 150, 100, 120);
```

Это выражение создает радиальный градиент и помещает его в переменную `rgObj`. Созданный им 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 в 120 пикселов.

Второй этап — расстановка ключевых точек — выполняется с помощью уже знакомого нам метода `addColorStop` объекта `CanvasGradient`.

```
addColorStop(<положение ключевой точки>, <цвет>)
```

Первый параметр задает относительное положение создаваемой ключевой точки на промежутке между внутренней окружностью и окружностью внешней. 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 можно использовать все форматы описания цвета, упомянутые в начале этой главы. Метод `addColorStop` не возвращает значения.

Как и линейный градиент, радиальный может содержать сколько угодно ключевых точек.

```
rgObj.addColorStop(0, "#CCCCCC");
rgObj.addColorStop(0.8, "#000000");
rgObj.addColorStop(1, "#00FF00");
```

Этот сценарий создает на полученном нами ранее радиальном градиенте три ключевые точки:

- расположенную в начальной точке воображаемого промежутка между окружностями 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 и задающую зеленый цвет.

Третий этап — использование готового радиального градиента — выполняется так же, как использование линейного градиента, то есть присваиванием его свойству `strokeStyle` или `fillStyle`.

```
ctvCnvObj.fillStyle = rgObj;
```

Радиальный градиент ведет себя точно так же, как линейный, то есть фиксируется на канве и частично "проявляется" на фигурах, к которым применен.

```
var rgObj = ctxCnvObj.createRadialGradient(100, 100, 10, 150, 100, 120);
rgObj.addColorStop(0, "#CCCCCC");
rgObj.addColorStop(0.8, "#000000");
rgObj.addColorStop(1, "#00FF00");
ctxCnvObj.fillStyle = rgObj;
ctxCnvObj.fillRect(0, 0, 200, 200);
```

Этот сценарий рисует прямоугольник и заполняет его радиальным градиентом, аналогичным тому, что мы создали ранее в этом параграфе. 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, которое используется для закраски линии или заливки. Таким графическим изображением может быть как обычный графический файл, так и содержимое другой канвы.

Графический цвет создается в два этапа. Сейчас мы их рассмотрим.

Первый этап — собственно создание графического цвета. Это выполняется с помощью метода `createPattern`.

`createPattent(`*<графическое изображение или канва>*`, `*<режим повторения>*`)`

Первый параметр задает графическое изображение в виде экземпляра объекта `Image` (мы рассматривали этот объект в *главе 9*; он используется для представления графического изображения, загруженного программно) или канву в виде экземпляра объекта `HTMLCanvasElement`.

Часто бывает так, что заданное графическое изображение имеет размеры меньшие, чем фигура, 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 (значение по умолчанию);
- "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, представляющий созданный нами графический цвет. Автору не удалось выяснить имя этого объекта, поэтому назовем его `Pattern`.

```
var imgObj = new Image();
imgObj.src = "someimage.jpg";
var gObj = ctxCnvObj.createPattern(imgObj, "repeat");
```

Это выражение создает графический цвет, используя для него графическое изображение, хранящееся в файле someimage.jpg. Отметим, что мы загружаем нужное изображение с использованием экземпляра объекта `Image`, как в случае с предзагрузкой изображений, описанной в *главе 9*.

Второй этап — использование готового графического цвета — выполняется так же, как использование градиентов, то есть присваиванием его свойству `strokeStyle` или `fillStyle`.

```
ctvCnvObj.fillStyle = imgObj;
```

В отличие от градиентов, графический цвет не фиксируется на канве, а полностью применяется к рисуемой фигуре. В этом его принципиальное отличие от градиентов.

```
var imgObj = new Image();
imgObj.src = "someimage.jpg";
var gObj = ctxCnvObj.createPattern(imgObj, "repeat");
ctxCnvObj.fillStyle = gObj;
ctxCnvObj.fillRect(0, 0, 200, 100);
```

Этот сценарий рисует прямоугольник с заливкой, которую создает с использованием графического цвета. А графический цвет создается из изображения, хранящегося в файле someimage.jpg.

> ***НА ЗАМЕТКУ***
>
> В приведенном ранее примере мы предположили, что файл someimage.jpg имеет небольшие размеры или уже присутствует в кэше Web-обозревателя и поэтому загрузится очень быстро. Но если он, так сказать, задержится в пути, сценарий не выполнится. Поэтому для вывода изображений, хранящихся в других файлах, используется иная методика, которую мы скоро рассмотрим.

Вывод внешних изображений

Внешним по отношению к канве называется графическое изображение, хранящееся в отдельном файле, или содержимое другой канвы. Firefox предоставляет довольно мощные средства для вывода таких изображений на канву; при этом мы имеем возможность изменить размеры изображения и даже вывести только его часть.

Если нам нужно просто вывести внешнее изображение на канву, мы используем метод `drawImage`, точнее, его сокращенный формат.

```
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 задают координаты точки канвы, где должен находиться верхний левый угол выводимого изображения; они задаются в пикселах в виде чисел. Метод `drawImage` не возвращает результата.

```
var imgObj = new Image();
imgObj.src = "someimage.jpg";
ctxCnvObj.drawImage(imgObj, 0, 0);
```

Этот сценарий загружает изображение из файла someimage.jpg и выводит его на канву так, чтобы его верхний левый угол находился в точке [0,0], то есть в верхнем левом углу канвы.

Если нам нужно при выводе внешнего изображения изменить его размеры, к нашим услугам расширенный формат метода `drawImage`.

```
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.

```
var imgObj = new Image();
imgObj.src = "someimage.jpg";
ctxCnvObj.drawImage(imgObj, 0, 0, 400, 300);
```

Этот сценарий загружает изображение из файла someimage.jpg и выводит его на канву, растягивая так, чтобы занять канву целиком.

Рассмотрим теперь самый сложный случай — вырезание из внешнего изображения фрагмента и вывод его на канву с изменением размеров. Для этого случая Firefox припас нам уже третий по счету формат метода `drawImage`.

```
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 задают координаты верхнего левого угла вырезаемого из внешнего изображения фрагмента. Они задаются относительно внешнего изображения в пикселах в виде чисел.

Четвертый и пятый параметры задают ширину и высоту вырезаемого из внешнего изображения фрагмента. Они также задаются относительно внешнего изображения в пикселах в виде чисел.

Шестой и седьмой параметры задают 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 задают ширину и высоту выводимого фрагмента внешнего изображения в пикселах в виде чисел.

```
var imgObj = new Image();
imgObj.src = "someimage.jpg";
ctxCnvObj.drawImage(imgObj, 20, 40, 40, 20, 0, 0, 400, 300);
```

Этот сценарий загружает внешнее изображение из файла someimage.jpg, вырезает из него фрагмент с верхним левым углом в точке [20,40], шириной в 40 и высотой в 20 пикселов и выводит этот фрагмент на канву, растягивая его так, чтобы занять канву целиком.

Все приведенные ранее примеры подразумевают, что файл, где хранится внешнее изображение, загружается очень быстро. (Это может быть в случае, 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 сценария. Как избежать этого?

Очень просто. Объект `Image`, как и знакомый нам по *главе 9* объект `HTMLImageElement`, поддерживает событие `onLoad`, возникающее после окончания загрузки изображения. Нам достаточно поместить код, выводящий внешнее изображение на канву, в функцию и привязать ее к событию `onLoad` в качестве его обработчика.

```
var imgObj = new Image();
function imgOnLoad() {
  ctxCnvObj.drawImage(imgObj, 20, 40, 40, 20, 0, 0, 400, 300);
}
imgObj.src = "someimage.jpg";
imgObj.onload = imgOnLoad;
```

Этот сценарий является примером использования только что описанного приема.

Далее приведен HTML-код страницы, многократно выводящей на канву содержимое графического файла someimage.jpg, чтобы получить своего рода мозаику. В этой странице также применен описанный ранее прием.

```
<HTML>
  <HEAD>
    <TITLE>Мозаика</TITLE>
  </HEAD>
  <BODY>
    <CANVAS ID="cnv" WIDTH="400" HEIGHT="300"></CANVAS>
    <SCRIPT>
      var cnvObj = document.getElementById("cnv");
      var ctxCnvObj = cnvObj.getContext("2d");

      function imgOnLoad() {
        for (var i = 0; i * 100 < cnvObj.width; i++) {
```

```
          for (var j = 0; j * 100 < cnvObj.height; j++) {
            ctxCnvObj.drawImage(imgObj, i * 100, j * 100, 100, 100);
          }
        }
      }

      var imgObj = new Image();
      imgObj.src = "default.jpg";
      imgObj.onload = imgOnLoad;
    </SCRIPT>
  </BODY>
</HTML>
```

Вряд ли здесь нужны комментарии. Можно отметить только код, применяемый нами для вывода "мозаики", но он также довольно прост.

Преобразования системы координат

Преобразования в терминологии канвы Firefox — это различные действия, которые мы можем выполнить над системой координат канвы. К преобразованиям относятся изменение масштаба, поворот и перемещение точки начала координат.

При выполнении преобразования изменяется только система координат канвы. Рисуемая после этого графика будет создаваться уже в измененной системе координат; а уже 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.

Сохранение состояния канвы выполняется вызовом метода `save`. Он не принимает параметров и не возвращает значения.

Состояние канвы сохраняется в памяти компьютера и впоследствии может быть восстановлено. Более того, сохранять состояние канвы можно несколько раз; при этом все предыдущие сохранения остаются в памяти и также могут быть восстановлены.

При вызове этого метода 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).

Восстановить сохраненное ранее состояние можно вызовом метода `restore`. Он также не принимает параметров и не возвращает значения.

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.

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`.

`translate(`*<горизонтальная координата>*`, `*<вертикальная координата>*`)`

Параметры этого метода задают 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 `translate` не возвращает результата.

При перемещении начала координат канвы будут учитываться все трансформации, примененные к канве ранее. То есть 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).

```
with (ctxCnvObj) {
  save();
  translate(100, 100);
  fillRect(0, 0, 50, 50);
  translate(100, 100);
  fillRect(0, 0, 50, 50);
  restore();
  fillRect(0, 0, 50, 50);
}
```

Этот сценарий 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, то есть 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, расположенных на воображаемой диагонали, тянущейся от верхнего левого угла канвы вниз и вправо.

> ***Внимание!***
>
> К сожалению, Firefox не предоставляет простого способа вернуться к системе координат по умолчанию. Единственный способ сделать это — сохранить состояние системы координат перед любыми трансформациями и потом восстановить его.

Поворот системы координат

Мы можем повернуть систему координат, точнее, ее оси, на произвольный угол вокруг точки начала координат. Для этого достаточно вызвать метод `rotate`.

```
rotate(<угол поворота>)
```

Единственный параметр этого метода задает угол поворота системы координат в радианах в виде числа; этот угол отсчитывается от горизонтальной оси. Метод `rotate` не возвращает значения.

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.

```
with (ctxCnvObj) {
  translate(200, 150);
```

```
  for (var i = 0; i < 3; i++) {
    rotate(Math.PI / 6);
    strokeRect(-50, -50, 100, 100);
  }
}
```

Этот сценарий сдвигает начало координат в центр канвы (точку [200,150]), после чего трижды поворачивает систему координат на $\pi/6$ 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

Мы также можем изменить масштаб системы координат канвы, сделав его крупнее или мельче. Для этого предназначен метод `scale`.

`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 использовать значение 1.0 соответствующего параметра. Метод `scale` не возвращает значения.

При изменении масштаба координат канвы будут учитываться все трансформации, примененные к канве ранее: перемещения начала координат, повороты и изменения масштаба.

```
with (ctxCnvObj) {
  save();
  strokeRect(0, 0, 50, 50);
  scale(3, 1);
  strokeRect(0, 0, 50, 50);
  restore();
  save();
  scale(1, 3);
  strokeRect(0, 0, 50, 50);
  restore();
  save();
  scale(3, 3);
  strokeRect(0, 0, 50, 50);
  restore();
}
```

Этот сценарий 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 Firefox по умолчанию, которое мы можем изменить.

Для управления наложением графики используется свойство `globalCompositeOperation`. Его значение должно быть одной из строк, перечисленных далее.

- "source-over" — новая фигура накладывается на старую, перекрывая ее (значение по умолчанию).
- "destination-over" — новая фигура перекрывается старой.
- "source-in" — выводится только та часть новой фигуры, которая накладывается на старую. Остальные части новой и старой фигур не выводятся.

- "destination-in" — выводится только та часть старой фигуры, на которую накладывается новая. Остальные части новой и старой фигур не выводятся.
- "source-out" — выводится только та часть новой фигуры, которая не накладывается на старую. Остальные части новой и старой фигур не выводятся.
- "destination-out" — выводится только та часть старой фигуры, на которую не накладывается новая. Остальные части новой и старой фигур не выводятся.
- "source-atop" — выводится 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" — выводится 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, в полученный цвет окрашиваются накладывающиеся части фигур.
- "darker" — цвета накладывающихся частей старой и новой фигур вычитаются, в полученный цвет окрашиваются накладывающиеся части фигур.
- "xor" — накладывающиеся части старой и новой фигур делаются прозрачными.
- "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.

```
with (ctxCnvObj) {
  fillStyle = "#0000FF";
  fillRect(0, 50, 400, 200);
  fillStyle = "#FF0000";
  globalCompositeOperation = "source-over";
  fillRect(100, 0, 200, 300);
}
```

Этот небольшой сценарий рисует два накладывающихся прямоугольника разных цветов и позволит изучить поведение канвы при разных значениях свойства `globalCompositeOperation`. Изменяем значение этого свойства, перезагружаем страницу в Firefox и смотрим, что получится.

Маски

Маска — это особая фигура, задающая своего рода "окно", сквозь которое будет видна часть графики, нарисованной на канве. Вся графика, не попадающая в это "окно", будет скрыта.

В качестве маски может быть использован только сложный контур, рисование которого было описано ранее. И создается она примерно так же.

1. Рисуем сложный контур, который станет маской.
2. Обязательно делаем его закрытым.
3. Вместо вызова методов `stroke` или `fill` вызываем метод `clip`. Этот метод не принимает параметров и не возвращает результата.
4. Рисуем графику, которая будет находиться под маской.

После этого нарисованная нами на шаге 4 графика будет частично видна сквозь маску. Требуемый результат достигнут.

```
with (ctxCnvObj) {
  beginPath();
  moveTo(100, 150);
  lineTo(200, 0);
  lineTo(200, 300);
  closePath();
  clip();
  fillRect(0, 100, 400, 100);
}
```

Этот сценарий рисует маску в виде треугольника, а потом — прямоугольник. Часть этого прямоугольника будет видна через маску.

На этом о рисовании на Web-странице средствами канвы все.

Что дальше?

В данной главе мы познакомились со средствами Firefox для рисования на странице произвольной графики. Правда, они не столь мощны, как средства, например, Adobe Flash, но Web-обозреватель все-таки не графическая программа.

На этом мы закончим рассмотрение специфических особенностей Web-обозревателей, которые могут быть полезны Web-дизайнерам и Web-программистам. В следующей части книги мы познакомимся с новомодной технологией AJAX, позволяющей подгружать данные в уже загруженную в Web-обозреватель страницу и выводить их на экран. Это будет, можно сказать, итог всех полученных нами знаний. И последнее, что мы рассмотрим в этой книге.

Часть IV

Начала технологии AJAX

Глава 18

Работа с данными XML

В предыдущих частях книги мы познакомились с языками HTML, CSS и JavaScript, изучили принципы написания Web-сценариев и обработки событий, работу со страницами и их элементами, способы обработки данных и взаимодействия с посетителем и мн. др. Теперь мы умеем создавать страницы, реагирующие на действия посетителя, причем весьма сложным образом.

Настала пора объединить все эти знания в нечто большее, чем просто умение заставить элемент страницы двигаться туда-сюда, Web-форму — проверять введенные данные на правильность, а Internet Explorer — создать тень для заголовка. В нечто всеобъемлющее, что потребовало бы владения всеми изученными нами интернет-технологиями. В нечто новое, чего еще никто не видел.

Этим большим, всеобъемлющим и новым "нечто" мы и займемся в последней части книги. AJAX — вот имя этого "нечто". Высший класс Web-программирования, принципиально новые способы работы с данными и, как результат, впечатляющие возможности, от которых посетители ахнут, а другие Web-программисты воспылают черной завистью. То, что способно радикально изменить давно знакомую нам Всемирную Паутину. Пик интернет-технологий. Пик интернет-моды.

Если коротко, то технология *AJAX* — это гремучая смесь JavaScript, XML и особых возможностей современных Web-обозревателей, позволяющих программно подгружать с Web-сервера данные и тут же их обрабатывать и выводить на экран. Кстати, аббревиатура "AJAX" и расшифровывается как "Asynchronous JavaScript and XML" — "асинхронный JavaScript и XML". Web-обозреватель подгружает данные в формате XML, а сценарий на JavaScript их обрабатывает и преобразует в HTML и CSS.

JavaScript нам уже знаком. А вот XML и способы программной подгрузки данных — нет. Их мы изучим в этой части. И начнем с XML.

Язык XML

XML ("eXtensible Markup Language" — "расширяемый язык разметки") — это универсальных язык описания данных. С его помощью часто описываются данные, предназначенные для передачи по сетям, и различные служебные данные, используемые различными программами. Разработанный еще в 90-х годах прошлого столетия, он только сейчас становится популярным.

Основные принципы, на которых построен язык XML, описаны далее.

- Данные хранятся в текстовом виде.
- Расширение файлов, хранящих данные XML, не стандартизовано. Чаще всего используется "говорящее" расширение xml, но иногда применяются и другие расширения.
- Данные форматируются с использованием тегов.
- Теги для описания различных данных могут быть любыми. Главное — чтобы эти данные были удобны для разбора соответствующими программами.
- Теги XML имеют тот же вид, что теги HTML: название тега записывается между символами < и >, после символа < закрывающего тега ставится символ /. Теги XML могут иметь атрибуты.

Если короче, то язык XML можно описать так. Мы берем данные, описываем их в текстовом виде, разбиваем на части и форматируем с использованием тегов и атрибутов, которые сами придумываем. Весьма удобно!

А теперь уясним требования языка XML к описанию данных. Они очень важны.

- Язык XML чувствителен к регистру символов, которыми набраны названия тегов. Так, `<SOMEDATA>` и `<somedata>` — совершенно разные теги.
- В языке XML допускаются только парные теги. Если нам понадобится создать подобие одинарного тега HTML, то придется заменить его пустым парным тегом (например, `<emptydata></emptydata>`). Но обычно для написания пустых тегов пользуются сокращенным синтаксисом `<<имя тега>/>` (например, `<emptydata/>`). Обратим внимание на символ / перед символом > — он в этом случае обязателен.
- Язык XML крайне чувствителен к ошибкам. Так, если мы, например, не закроем парный тег, программа даже не сможет обработать данные.

Традиционно названия тегов XML и их атрибутов записываются маленькими буквами. Вероятно, это делается для того, чтобы сразу отличить данные XML от HTML-кода.

А теперь давайте рассмотрим пару примеров данных, описанных языком XML. В нашем случае это будет список языков программирования.

```
<langlist>
  <lang>
    <name>JavaScript</name>
    <type>Интерпретируемый</type>
  </lang>
  <lang>
    <name>VBScript</name>
    <type>Интерпретируемый</type>
  </lang>
  <lang>
    <name>Java</name>
    <type>Компилируемый</type>
  </lang>
  <lang>
    <name>C++</name>
    <type>Компилируемый</type>
  </lang>
  <lang>
    <name>C#</name>
    <type>Компилируемый</type>
  </lang>
  <lang>
    <name>Delphi</name>
    <type>Компилируемый</type>
  </lang>
</langlist>
```

В первом примере мы использовали четыре тега, которые придумали сами. В этом вся прелесть XML — мы можем придумать любые теги и атрибуты, какие захотим.

- `<langlist>` — содержит сам список языков программирования.
- `<lang>` — содержит сведения об одном языке программирования.
- `<name>` — содержит название языка программирования.
- `<type>` — содержит категорию языка программирования — интерпретируемый или компилируемый.

В тег <langlist> вложен набор тегов <lang>. В тег <lang> вложены теги <name> и <type>. А теги <name> и <type> содержат текстовые элементы, представляющие сведения о языке программирования.

Все данные, представленные в формате XML, помещаются в один "глобальный" тег; в нашем случае это тег <langlist>. Это требование стандарта XML.

```
<langlist>
  <lang type="0">JavaScript</lang>
  <lang type="0">VBScript</lang>
  <lang type="1">Java</lang>
  <lang type="1">C++</lang>
  <lang type="1">C#</lang>
  <lang type="1">Delphi</lang>
</langlist>
```

Второй пример представляет собой сокращенный вариант первого. В нем мы отказались от тегов <name> и <type>, поместили название языка программирования прямо в тег <lang>, а для обозначения категории использовали атрибут type этого тега. Значение "0" атрибута type обозначает интерпретируемый язык, значение "1" — компилируемый.

Собственно, этих начальных сведений о языке XML нам пока должно хватить. Желающие узнать о XML больше могут обратиться к соответствующим книгам.

XML DOM

Когда какая-либо программа (тот же Web-обозреватель) загружает данные XML в память, она формирует совокупность экземпляров различных объектов, представляющих эти данные, — *XML DOM*. Она аналогична DOM, представляющей Web-страницу и рассмотренной нами в *главе 5*.

Давайте рассмотрим такие данные XML:

```
<langlist>
  <lang>
    <name>JavaScript</name>
    <type>Интерпретируемый</type>
  </lang>
</langlist>
```

Это фрагмент приведенного ранее списка языков программирования. Созданная на его основе XML DOM будет иметь такую схему:

```
Тег <langlist>
  Тег <lang>
    Тег <name>
      Текст "JavaScript"
    Тег <type>
      Текст "Интерпретируемый"
```

Здесь нам все знакомо — мы уже видели нечто подобное, когда рассматривали DOM Web-страницы в *главе 5*. Языки HTML и DOM по сути "родственники", и программное представление Web-страницы и документа XML выглядит одинаково.

К сожалению, здесь также наблюдается картина, описанная еще в *главе 5*. Internet Explorer, с одной стороны, и Opera и Firefox, с другой, формируют XML DOM по-разному. Как и обычную DOM Web-страницы.

Представленная ранее схема XML DOM будет сформирована Internet Explorer. Схема XML DOM, сформированная Opera и Firefox, будет выглядеть так:

```
Тег <XML>
  "Пустой" текст
  Тег <langlist>
    "Пустой" текст
    Тег <lang>
      "Пустой" текст
      Тег <name>
        Текст "JavaScript"
      "Пустой" текст
      Тег <type>
        Текст "Интерпретируемый"
```

Мы видим, что в этом случае DOM включает в себя текстовые элементы, представляющие промежутки между тегами: последовательности символов пробела, возврата каретки и перевода строки. Эту особенность обязательно нужно учитывать при написании сценариев, разбирающих данные XML; подробнее мы рассмотрим этот вопрос далее.

Любой тег XML представляется экземпляром объекта `Element`. Этот объект поддерживает свойства `attributes`, `childNodes`, `firstChild`, `lastChild`, `nextSibling`, `nodeName`, `nodeType`, `nodeValue`, `parentNode`, `previousSibling` и `tagName` и методы `appendChild`, `getAttributeNode`, `getElementsByTagName`, `hasAttribute` и `hasChildNodes`.

Свойство `attributes` возвращает ссылку на одноименную коллекцию — экземпляр объекта `NamedNodeList`, — содержащую все атрибуты данного тега. Этот объект поддерживает свойство `length` и метод `getNamedItem`.

Атрибут тега XML представляется экземпляром объекта `Attr`. Этот объект поддерживает свойства `name` и `value`.

Текстовый элемент XML представляется экземпляром объекта `Text`. Этот объект поддерживает свойства `nodeName`, `nodeType` и `nodeValue`.

Сам же фрагмент данных XML, хранящийся в отдельном файле или являющийся частью Web-страницы, представляется экземпляром объекта `Document`. Чтобы, имея его, получить доступ к самому внешнему тегу (тегу нулевого уровня вложенности; подробнее см. *главу 2*), достаточно использовать свойство `firstChild` объекта `Document`. Кроме того, этот объект поддерживает метод `getElementsByTagName`, использовать который будет явно удобнее.

Все эти свойства и методы уже знакомы нам и описаны в *главе 5*. Знания их нам будет вполне достаточно для того, чтобы разобрать данные XML — извлечь содержимое тегов и значения атрибутов и использовать их каким-то образом, например, вывести на страницу. Ведь, как правило, Web-сценарии занимаются разбором уже созданных данных, а не их созданием.

Вставка данных XML в Web-страницу

Стандарт HTML не предлагает средств для вставки данных XML прямо в HTML-код страницы. Однако мы можем использовать один фокус, чтобы все-таки это сделать.

Еще в *главе 9* говорилось, Web-обозреватель, столкнувшийся с тегом, который он не поддерживает (неизвестным тегом), должен его проигнорировать и вывести на страницу содержимое данного тега (если оно присутствует). Это требование все того же стандарта HTML. Но самое интересное — неизвестный тег становится частью DOM страницы! Это значит, что мы можем использовать знакомые нам по *главам 5* и *8* методы и свойства DOM для доступа к нему и его дочерним тегам и текстовым элементам.

Здесь нужно иметь в виду вот что. Ранее было сказано, что данные XML представляются обрабатывающими их программами в виде XML DOM. Но если мы вставим данные XML в Web-страницу, они станут частью DOM Web-страницы. Со всеми вытекающими отсюда особенностями, которые мы впоследствии опишем.

Давайте вставим в страницу данные XML, приведенные ранее и содержащие список языков программирования.

```
<BODY>
  . . .
  <XML>
    <langlist>
      <lang>
        <name>JavaScript</name>
        <type>Интерпретируемый</type>
      </lang>
      <lang>
        <name>VBScript</name>
        <type>Интерпретируемый</type>
      </lang>
      <lang>
        <name>Java</name>
        <type>Компилируемый</type>
      </lang>
      <lang>
        <name>C++</name>
        <type>Компилируемый</type>
      </lang>
      <lang>
        <name>C#</name>
        <type>Компилируемый</type>
      </lang>
      <lang>
        <name>Delphi</name>
        <type>Компилируемый</type>
      </lang>
    </langlist>
  </XML>
  . . .
</BODY>
```

Здесь мы использовали парный тег `<XML>`, внутри которого поместили наши данные XML. Этот тег не поддерживается ни одним Web-обозревателем и, стало быть, является неизвестным.

Но здесь мы сталкиваемся с тремя проблемами. Давайте их рассмотрим и сразу же наметим пути решения.

Первая проблема — нам нужно как-то получить доступ к неизвестному тегу, содержащему данные XML, из сценария. Решается она просто — достаточно использовать давно нам знакомый атрибут `ID` для задания имени этого тега. Стандарт HTML требует, чтобы все теги, даже неизвестные, поддерживали этот атрибут, и они его поддерживают.

Вторая проблема посерьезнее. Как мы уже знаем, Web-обозреватель, встретивший в HTML-коде неизвестный тег, должен вывести на страницу его содержимое, если оно есть. И он выведет. В нашем случае мы увидим на странице примерно следующее:

```
JavaScript Интерпретируемый VBScript Интерпретируемый Java Компилируемый
↳C++ Компилируемый C# Компилируемый Delphi Компилируемый
```

И зачем нам это нужно? Ведь мы не хотим, чтобы данные XML присутствовали на странице в таком виде. Ведь мы хотим их обработать в сценариях перед выводом на экран, а то и вообще использовать только "внутри" страницы, вообще никуда не выводя. Как нам указать Web-обозревателю, чтобы он не выводил содержимое тега `<XML>`?

Здесь нам помогут стили CSS, поддерживать которые, согласно стандарту HTML, должны даже неизвестные теги. А именно атрибут стиля `display`, управляющий отображением элемента страницы и описанный в *главе 3*. Достаточно привязать к тегу `<XML>` встроенный стиль, поместить в него атрибут `display`, дать этому атрибуту значение "none" — и содержимое тега `<XML>` не будет выводиться на страницу ни в каком случае.

Исходя из всего этого, нам необходимо написать открывающий тег `<XML>` таким образом:

```
<XML ID="data" STYLE="display: none;">
```

Здесь мы выбрали для тега `<XML>` имя `data`. Хотя, разумеется, можем выбрать и другое.

Третья проблема самая серьезная. Она связана с тем, что разные Web-обозреватели по-разному формируют XML DOM. Мы уже рассматривали этот момент, но он настолько важен, что достоин повторного рассмотрения.

Возьмем такой фрагмент данных XML:

```
<XML ID="data" STYLE="display: none;">
  <langlist>
    <lang>
      <name>JavaScript</name>
```

```
      <type>Интерпретируемый</type>
    </lang>
  </langlist>
</XML>
```

Он помещен в HTML-код некой страницы с помощью тега `<XML>` только что рассмотренным нами способом. Загрузим эту страницу в Internet Explorer. Он сформирует такую DOM для этого фрагмента:

```
Тег <XML>
  Тег <langlist>
    Тег <lang>
      Тег <name>
        Текст "JavaScript"
      Тег <type>
        Текст "Интерпретируемый"
```

В этой DOM мы можем пользоваться любыми методами и свойствами DOM для доступа к различным тегам, описывающим данные XML. Так, мы можем получить доступ к тегу `<langlist>` и "пройти" все теги `<lang>`, используя свойство `childNodes`. Как мы помним, это свойство возвращает коллекцию, которая содержит все элементы страницы, являющиеся дочерними по отношению к данному тегу.

```
var dataObj = document.getElementById("data");
var langListObj = dataObj.childNodes[0];
var langObj = langListObj.childNodes[0];
```

После выполнения этого сценария в переменной `dataObj` окажется сам тег `<XML>`, в переменной `langListObj` — тег `<langlist>` (единственный дочерний элемент тега `<XML>`), а в переменной `langObj` — тег `<lang>` (единственный дочерний элемент тега `<langlist>`).

Используя переменную `langObj` и свойство `childNodes`, мы сможем без труда добраться до тегов `<name>` и `<type>` и содержащихся в них текстовых элементов:

```
var langName = langObj.childNodes[0].firstChild.nodeValue;
var langType = langObj.childNodes[1].firstChild.nodeValue;
```

Все очень просто и наглядно. Но это только в случае Internet Explorer.

Теперь загрузим эту страницу в Opera или Firefox. Мы уже знаем, что они включают в DOM текстовые элементы, представляющие промежутки между тегами. И они сформируют такую DOM:

```
Тег <XML>
  "Пустой" текст
  Тег <langlist>
```

```
"Пустой" текст
Тег <lang>
  "Пустой" текст
  Тег <name>
    Текст "JavaScript"
  "Пустой" текст
  Тег <type>
    Текст "Интерпретируемый"
```

Посмотрим, как выполнится уже знакомый нам сценарий

```
var dataObj = document.getElementById("data");
var langListObj = dataObj.childNodes[0];
var langObj = langListObj.childNodes[0];
```

В переменной `dataObj` окажется тег `<XML>`. А переменная `langListObj` будет в данном случае содержать текстовый элемент, представляющий промежуток между открывающим тегом `<XML>` и открывающим тегом `<langlist>` (первый дочерний элемент тега `<XML>`). Третье же выражение вызовет ошибку, так как текстовые элементы не имеют потомков.

Поэтому при разборе данных XML, вставленных в HTML-код, следует использовать другие способы. Удобнее всего применять метод `getElementsByTagName`, подробно описанный в *главе 5*. Он позволит нам добраться до конкретного тега.

```
var dataObj = document.getElementById("data");
var langsObj = dataObj.getElementsByTagName("lang");
var langName = langsObj.getElementsByTagName("name")[0].
↳firstChild.nodeValue;
var langType = langsObj.getElementsByTagName("type")[0].
↳firstChild.nodeValue;
```

После выполнения этого сценария:

- в переменной `dataObj` окажется тег `<XML>`;
- в переменной `langsObj` окажется коллекция, содержащая все теги `<lang>`;
- в переменных `langName` и `langType` окажутся название и категория языка программирования.

Здесь мы даже не стали получать доступ к тегу `<langlist>`, а сразу через его "голову" добрались до всех тегов `<lang>`. В этом еще одно преимущество метода `getElementsByTagName` — он позволяет получить доступ к любому дочернему тегу.

А самое главное для нас — приведенный ранее код будет работать и в Internet Explorer, и в Opera, и в Firefox.

Описанный здесь способ позволит нам без проблем поместить нужные нам данные XML в любую страницу для дальнейшей обработки и, возможно, вывода на экран. А для того чтобы закрепить полученные знания, мы рассмотрим три примера страниц, работающих с данными XML.

НА ЗАМЕТКУ

Internet Explorer официально поддерживает тег <XML>, предназначенный для помещения на страницу данных XML; по крайней мере, так написано в MSDN. Но никаких реальных преимуществ эта поддержка нам не дает.

Простейшая страница, обрабатывающая данные XML

Начнем с самой простой страницы, содержащей список языков программирования в формате XML и выводящей его в виде таблицы. HTML-код этой страницы приведен далее.

```
<HTML>
  <HEAD>
    <TITLE>XML</TITLE>
  </HEAD>
  <BODY>
    <XML ID="data" STYLE="display: none;">
      <langlist>
        <lang>
          <name>JavaScript</name>
          <type>Интерпретируемый</type>
        </lang>
        <lang>
          <name>VBScript</name>
          <type>Интерпретируемый</type>
        </lang>
        <lang>
          <name>Java</name>
          <type>Компилируемый</type>
        </lang>
        <lang>
          <name>C++</name>
          <type>Компилируемый</type>
        </lang>
```

```
    <lang>
      <name>C#</name>
      <type>Компилируемый</type>
    </lang>
    <lang>
      <name>Delphi</name>
      <type>Компилируемый</type>
    </lang>
  </langlist>
</XML>
<SCRIPT>
  var dataObj = document.getElementById("data");
  var tableObj = document.createElement("TABLE");
  var tHeadObj = document.createElement("THEAD");
  var trObj = document.createElement("TR");
  var cellObj = document.createElement("TH");
  var textObj = document.createTextNode("Название");
  cellObj.appendChild(textObj);
  trObj.appendChild(cellObj);
  cellObj = document.createElement("TH");
  textObj = document.createTextNode("Категория");
  cellObj.appendChild(textObj);
  trObj.appendChild(cellObj);
  tHeadObj.appendChild(trObj);
  tableObj.appendChild(tHeadObj);
  var tBodyObj = document.createElement("TBODY");
  var langsObj = dataObj.getElementsByTagName("lang");
  for (var i = 0; i < langsObj.length; i++) {
    trObj = document.createElement("TR");
    cellObj = document.createElement("TD");
    textObj = document.createTextNode(langsObj[i].
    ↳getElementsByTagName("name")[0].firstChild.nodeValue);
    cellObj.appendChild(textObj);
    trObj.appendChild(cellObj);
    cellObj = document.createElement("TD");
    textObj = document.createTextNode(langsObj[i].
    ↳getElementsByTagName("type")[0].firstChild.nodeValue);
    cellObj.appendChild(textObj);
```

```
        trObj.appendChild(cellObj);
        tBodyObj.appendChild(trObj);
      }
      tableObj.appendChild(tBodyObj);
      document.body.appendChild(tableObj);
    </SCRIPT>
  </BODY>
</HTML>
```

Здесь мы формируем таблицу, используя методы и свойства DOM, и заполняем ее данными, извлеченными из тега `<XML>`, где эти данные хранятся в формате XML. Больше комментировать здесь нечего.

Более сложная страница, обрабатывающая данные XML

Аппетит приходит во время еды. Давайте усложним нашу первую станицу, работающую с данными XML. В качестве собственно данных используем второй список языков программирования, где категория языка обозначается номером, заданным в атрибуте `type` тега `<lang>`. Кроме этого, создадим второй фрагмент данных XML, представляющий собой список категорий языков программирования и соответствующих им номеров. И, разумеется, при формировании таблицы будем подставлять вместо номера категории ее название, извлекая его из второго списка.

Далее представлен HTML-код новой страницы. Он не очень сильно изменился по сравнению с предыдущей страницей.

```
<HTML>
  <HEAD>
    <TITLE>XML</TITLE>
  </HEAD>
  <BODY>
    <XML ID="data" STYLE="display: none;">
      <langlist>
        <lang type="0">JavaScript</lang>
        <lang type="0">VBScript</lang>
        <lang type="1">Java</lang>
        <lang type="1">C++</lang>
        <lang type="1">C#</lang>
        <lang type="1">Delphi</lang>
```

```
  </langlist>
</XML>
<XML ID="data2" STYLE="display: none;">
  <typelist>
    <type tid="0">Интерпретируемый</type>
    <type tid="1">Компилируемый</type>
  </typelist>
</XML>
<SCRIPT>
  var data2Obj = document.getElementById("data2");
  var typesObj = data2Obj.getElementsByTagName("type");
  var types = [];
  for (var i = 0; i < typesObj.length; i++) {
    types[typesObj[i].getAttributeNode("tid").value] =
    typesObj[i].firstChild.nodeValue;
  }
  var dataObj = document.getElementById("data");
  var tableObj = document.createElement("TABLE");
  var tHeadObj = document.createElement("THEAD");
  var trObj = document.createElement("TR");
  var cellObj = document.createElement("TH");
  var textObj = document.createTextNode("Название");
  cellObj.appendChild(textObj);
  trObj.appendChild(cellObj);
  cellObj = document.createElement("TH");
  textObj = document.createTextNode("Категория");
  cellObj.appendChild(textObj);
  trObj.appendChild(cellObj);
  tHeadObj.appendChild(trObj);
  tableObj.appendChild(tHeadObj);
  var tBodyObj = document.createElement("TBODY");
  var langsObj = dataObj.getElementsByTagName("lang");
  for (var i = 0; i < langsObj.length; i++) {
    trObj = document.createElement("TR");
    cellObj = document.createElement("TD");
    textObj = document.createTextNode(types[langsObj[i].
    ↳getAttributeNode("type").value]);
    cellObj.appendChild(textObj);
```

```
        trObj.appendChild(cellObj);
        cellObj = document.createElement("TD");
        textObj = document.createTextNode(langsObj[i].
        ↳firstChild.nodeValue);
        cellObj.appendChild(textObj);
        trObj.appendChild(cellObj);
        tBodyObj.appendChild(trObj);
      }
      tableObj.appendChild(tBodyObj);
      document.body.appendChild(tableObj);
    </SCRIPT>
  </BODY>
</HTML>
```

Поскольку у нас уже два фрагмента данных XML, мы использовали два тега `<XML>` с разными именами, заданными атрибутами `ID`. Первый из этих фрагментов содержит список языков программирования, второй — список категорий.

> ***Внимание!***
>
> В принципе, можно поместить оба фрагмента данных XML в один тег `<XML>`, но тогда Internet Explorer не сможет их обработать. Так что нам пришлось использовать два тега `<XML>`. И в дальнейшем следует использовать принцип: один фрагмент данных XML — один тег для его размещения.
>
> Можно, правда, поместить оба фрагмента данных XML в один тег `<XML>`, объединив их дополнительно еще в один тег, который можно назвать, скажем, `<langdata>`. Вот так:

```
<XML ID="data" STYLE="display: none;">
  <langdata>
    <langlist>
      <lang type="0">JavaScript</lang>
      <lang type="0">VBScript</lang>
      . . .
    </langlist>
    <typelist>
      <type tid="0">Интерпретируемый</type>
      <type tid="1">Компилируемый</type>
    </typelist>
  </langdata>
</XML>
```

> Возможно, так даже будет нагляднее. Но, по мнению автора, это дело вкуса.

Первым делом мы получаем доступ к списку категорий и помещаем все содержащиеся в нем категории в ассоциативный массив `types`. Номер категории становится индексом элемента этого массива, а название категории — значением элемента массива.

При формировании таблицы мы считываем номер категории из атрибута `type` тега `<lang>` и используем его в качестве индекса для извлечения из массива `types` соответствующего элемента. Мы уже проделывали нечто подобное в предыдущих главах этой книги.

Остальной код не требует комментариев.

Страница, выводящая данные XML по частям с возможностью листания

Последний пример, что мы рассмотрим в этой главе, будет самым сложным. Мы возьмем все ту же страницу, выводящую на экран список языков программирования в виде таблицы, и снова переделаем ее. Пусть теперь она выводит одновременно только три позиции списка и предоставляет возможность его "листания". То есть этот пример будет аналогичным одному из тех, что мы рассмотрели в *главе 14*.

HTML-код этой страницы приведен далее.

```
<HTML>
  <HEAD>
    <TITLE>XML</TITLE>
  </HEAD>
  <BODY>
    <XML ID="data" STYLE="display: none;">
      <langlist>
        <lang type="0">JavaScript</lang>
        <lang type="0">VBScript</lang>
        <lang type="1">Java</lang>
        <lang type="1">C++</lang>
        <lang type="1">C#</lang>
        <lang type="1">Delphi</lang>
      </langlist>
    </XML>
    <XML ID="data2" STYLE="display: none;">
      <typelist>
        <type tid="0">Интерпретируемый</type>
        <type tid="1">Компилируемый</type>
```

```
  </typelist>
</XML>
<DIV ID="cont"> </DIV>
<SCRIPT>
  var data2Obj = document.getElementById("data2");
  var typesObj = data2Obj.getElementsByTagName("type");
  var types = [];
  for (var i = 0; i < typesObj.length; i++) {
    types[typesObj[i].getAttributeNode("tid").value] =
    typesObj[i].firstChild.nodeValue;
  }

  var pageSize = 3;

  var currPage = 0;
  var firstItem = 0;
  var lastItem = 0;

  function showTable() {
    var contObj = document.getElementById("cont");
    while (contObj.childNodes.length > 0)
      contObj.removeChild(contObj.firstChild);
    var tableObj = document.createElement("TABLE");
    var tHeadObj = document.createElement("THEAD");
    var trObj = document.createElement("TR");
    var cellObj = document.createElement("TH");
    var textObj = document.createTextNode("Название");
    cellObj.appendChild(textObj);
    trObj.appendChild(cellObj);
    cellObj = document.createElement("TH");
    textObj = document.createTextNode("Категория");
    cellObj.appendChild(textObj);
    trObj.appendChild(cellObj);
    tHeadObj.appendChild(trObj);
    tableObj.appendChild(tHeadObj);
    var tBodyObj = document.createElement("TBODY");
    var dataObj = document.getElementById("data");
    var langsObj = dataObj.getElementsByTagName("lang");
    if (currPage < 0) currPage = 0;
    firstItem = currPage * pageSize;
```

```
        lastItem = firstItem + pageSize - 1;
        if (lastItem > langsObj.length - 1)
          lastItem = langsObj.length - 1;
        for (var i = firstItem; i <= lastItem; i++) {
          trObj = document.createElement("TR");
          cellObj = document.createElement("TD");
          textObj = document.createTextNode(types[langsObj[i].
          ↳getAttributeNode("type").value]);
          cellObj.appendChild(textObj);
          trObj.appendChild(cellObj);
          cellObj = document.createElement("TD");
          textObj = document.createTextNode(langsObj[i].
          ↳firstChild.nodeValue);
          cellObj.appendChild(textObj);
          trObj.appendChild(cellObj);
          tBodyObj.appendChild(trObj);
        }
        tableObj.appendChild(tBodyObj);
        contObj.appendChild(tableObj);
      }

      showTable();
    </SCRIPT>
    <P><A HREF="#" ONCLICK="currPage--; showTable();">Предыдущая
      страница</A>,
      <A HREF="#" ONCLICK="currPage++; showTable();">следующая
      страница</A></P>
  </BODY>
</HTML>
```

Здесь мы объявляем четыре переменные. Переменная `pageSize` хранит размер страницы (количество одновременно выводимых позиций списка языков), переменная `currPage` — номер выводимой в данный момент страницы списка (нумерация страниц, по старой доброй программистской традиции, начинается с нуля), а переменные `firstItem` и `lastItem` — номера первой и последней выводимых позиций списка (также нумеруются с нуля).

Сама таблица, содержащая текущую страницу, выводится в контейнере `cont`. Благодаря этому мы сможем поместить таблицу в нужное место Web-страницы и, главное, удалить ее перед выводом новой страницы списка вызовом метода `removeChild`.

Вывод текущей страницы списка выполняет функция `showTable`. Эта функция удаляет таблицу, содержащую ранее выведенную страницу списка, вычисляет номера первой и последней выводимых позиций списка и выводит таблицу с новой страницей. Впервые эта функция вызывается при загрузке Web-страницы и выводит на нее первую страницу списка (имеющую номер 0).

Отметим, как мы вычисляем номера выводимых позиций списка. Номер первой позиции вычисляется как произведение номера текущей страницы на ее размер. Номер последней позиции есть сумма номера первой позиции и размера страницы за вычетом единицы — это необходимо, так как позиции списка у нас нумеруются с нуля. После этого мы проверяем, не превышает ли номер последней выводимой позиции списка номера последнего элемента этого списка (он равен размеру списка за вычетом единицы — не забываем, что позиции списка нумеруются с нуля), и, если превышает, приравниваем его номеру последней позиции списка.

Может случиться так, что посетитель, наблюдая на экране первую страницу списка, попытается перейти к предыдущей. На этот случай мы проверяем, не стал ли номер текущей страницы меньше нуля, и, если стал, приравниваем его нулю.

Для "листания" мы используем две пустые гиперссылки, к событиям `onClick` которых привязаны обработчики. Эти обработчики просто уменьшают или увеличивают номер текущей страницы на единицу и вызывают функцию `showTable`.

Остальной код не требует комментариев. Вы сами сможете с ним разобраться и, при желании, даже усовершенствовать.

На этом мы закончим рассмотрение языка XML, способов описания данных с его помощью и вставку их в HTML-код Web-страниц. Желающие узнать о XML больше, могут обратиться к соответствующей литературе.

Что дальше?

В самом начале этой главы было сказано, что технология AJAX "стоит" на трех "китах": JavaScript, XML и программной подгрузке данных. С первыми двумя "китами" мы уже знакомы: XML, способы описания с его помощью данных и работы с ними были рассмотрены в этой главе, а JavaScript мы занимаемся на протяжении всей книги. Настала пора познакомиться с программной подгрузкой данных.

Программную подгрузку данных реализуют все современные Web-обозреватели. Выполняется она с помощью особого компонента Web-обозревателя, с которым мы начнем работать совсем скоро — в следующей главе.

Глава 19

Асинхронный обмен данными

В предыдущей главе мы кратко рассмотрели язык XML, описание данных с его помощью и способы программной обработки таких данных. Фактически мы сделали первый шаг в технологию AJAX.

В этой главе мы рассмотрим способы программной подгрузки данных — сделаем второй шаг. После этого мы можем считать, что технология AJAX нам подвластна, по крайней мере, на клиентском уровне. (Третий, последний, шаг в технологию AJAX требует знания серверного программирования, но это тема других книг.)

Программная подгрузка данных в произвольный момент времени, иначе говоря, *асинхронный обмен данными*, — замечательное изобретение. Благодаря ей мы можем загрузить любые нужные нам данные с Web-сервера или получить их от серверной программы, не перезагружая саму страницу. Эти данные, как правило, представляются в XML, с которым мы уже знакомы.

Но что это мы все: AJAX, AJAX… А что это такое? В предыдущей главе мы дали только очень краткое определение, из которого мало что понятно. Не пора ли рассмотреть эту технологию подробнее?

Введение в технологию AJAX

Еще в *главе 12*, рассматривая Web-формы, мы говорили о том, где и как будут обрабатываться введенные в них данные. Конечно, мы можем написать сценарий, который будет выполняться Web-обозревателем, обрабатывать эти данные и выводить результат их обработки на страницу. (В *главе 12* мы этим и занимались.) Но в абсолютном большинстве случаев введенные в Web-форму данные обрабатываются не Web-сервером и уж тем более не Web-обозревателем.

А кем же?

Особой программой, которая работает на серверном компьютере совместно с Web-сервером (серверной программой). Она получает введенные в Web-форму данные через Web-сервер, обрабатывает их определенным образом и возвращает результат их обработки в виде — внимание! — обычной Web-страницы. Эта страница, опять же, через Web-сервер отправляется Web-обозревателю, и Web-обозреватель выводит ее посетителю.

Серверные программы обеспечивают функциональность очень многих сайтов, таких как почтовые системы, работающие через Web-обозреватель, поисковые системы, интернет-магазины, гостевые книги, форумы, блоги и пр. Именно серверная программа является "сердцем" таких сайтов; Web-обозреватель же просто принимает от посетителя данные (имя и пароль посетителя, текст письма или статьи для блога, ключевое слово для поиска, почтовый адрес для отправки товара и др.) и выводит то, что выработала серверная программа.

Серверные программы в свое время дали немалый толчок развитию Интернета. Можно сказать, что это была вторая интернет-революция, если первой интернет-революцией считать появление WWW.

Но все ли так безоблачно сейчас, в "послереволюционное" время? Отнюдь! Безоблачных времен вообще никогда не бывает.

Давайте рассмотрим, как происходит обработка данных серверной программой. Сначала Web-обозреватель выводит страницу с Web-формой, в которую нужно ввести какие-то данные, скажем, ключевое слово для поиска. Посетитель вводит эти данные и нажимает кнопку отправки данных (за подробностями о формах и кнопках — в *главу 12*). Web-сервер получает отправленные данные, запускает нужную серверную программу и передает данные ей. Серверная программа обрабатывает данные, формирует новую страницу и через Web-сервер отправляет Web-обозревателю. Web-обозреватель загружает сгенерированную серверной программой страницу и выводит на экран. Посетитель смотрит список сайтов, удовлетворяющих введенному им ключевому слову.

Теперь предположим, что посетитель не нашел то, что искал, и ввел другое ключевое слово. Серверная программа получит его, сформирует другую Web-страницу и отправит ее Web-обозревателю; Web-обозреватель загрузит ее и выведет на экран. Вроде бы, все замечательно…

Но давайте посмотрим на полезные данные, которые формируются серверной программой, — собственно список сайтов, удовлетворяющих ключевому слову. Зачастую он занимает меньшую часть страницы; бóльшую же часть

занимают элементы оформления (заголовок, украшения и пр.), набор гиперссылок на другие страницы, список рубрик каталога сайтов, погода, курсы валют, заголовки новостей и т. д. И все это загружается каждый раз заново.

Вот если бы существовала возможность не загружать страницу целиком, а только подгружать сам список найденных сайтов, а потом просто вставлять его на нужное место страницы… Тогда поисковая система работала бы существенно быстрее (по крайней мере, на взгляд посетителя), ведь чем меньше данные, тем быстрее они передаются по Сети.

Именно такую возможность предлагает технология AJAX. Серверная программа отправляет Web-обозревателю только полезные данные, которые обрабатываются и выводятся на страницу особыми Web-сценариями. Сама страница при этом не перезагружается.

Давайте рассмотрим, как работает технология AJAX, на примере поисковой системы.

1. Web-обозреватель загружает страницу с формой, в которую нужно ввести ключевое слово.
2. Посетитель вводит ключевое слово в форму и нажимает кнопку отправки данных.
3. Особый Web-сценарий инициирует асинхронный обмен данными и с его помощью отправляет данные серверной программе.
4. Серверная программа получает ключевое слово, формирует список найденных сайтов в виде фрагмента данных XML и отправляет его Web-обозревателю.
5. Web-обозреватель получает фрагмент данных XML, содержащий список найденных сайтов, через все тот же асинхронный обмен данными.
6. Web-обозреватель запускает Web-сценарий, который разбирает полученные данные и выводит их на страницу.

Вот здесь-то и пригодится наше знание XML! Технология AJAX манипулирует данными, представленными именно на этом языке.

НА ЗАМЕТКУ

Вообще, серверная программа, реализующая технологию AJAX, может отправлять Web-обозревателю данные в любом виде: и текстовом, и даже в виде HTML-кода. Но традиционно для этого используются фрагменты данных XML.

ВНИМАНИЕ!

Данные XML, которые формирует серверная программа, реализующая технологию AJAX, должны быть представлены в кодировке UTF-8. Дело в том, что Web-обозреватели для разбора данных XML используют по умолчанию именно

эту кодировку. Конечно, можно использовать и другие кодировки, но это требует от Web-программиста дополнительных телодвижений и вряд ли дает какие-то преимущества.

Чтобы попытаться реализовать некое подобие технологии AJAX на своих страницах, нам не хватает знания асинхронного обмена данными. В этой главе мы как раз им и займемся.

Внимание!

Мы не будем писать серверные программы — эта книга посвящена только клиентскому Web-программированию, то есть созданию сценариев, работающих на Web-страницах. Желающие заняться серверным Web-программированием могут обратиться к соответствующей литературе, которой сейчас довольно много.

Реализация асинхронного обмена данными

За асинхронный обмен данными "отвечает" особый объект `XMLHttpRequest`. Он поддерживает свойства, методы и события, с помощью которых мы можем осуществить отправку данных серверной программе и получение от нее результата.

В Internet Explorer 7, Opera и Firefox объект `XMLHttpRequest` является неотъемлемой частью Web-обозревателя. Его экземпляр создается уже привычным для нас способом — с помощью оператора `new`, описанного в *главе 4*. Никаких параметров конструктору этого объекта не передается.

```
var xhrObj = new XMLHttpRequest();
```

Это выражение создает экземпляр объекта `XMLHttpRequest` и помещает его в переменную `xhrObj`.

Internet Explorer версий 5.* и 6.0 (более ранние версии этой программы не поддерживают асинхронный обмен данными) не включают в себя объект `XMLHttpRequest`, а используют его аналог, реализуемый операционной системой. Поэтому для создания экземпляра данного объекта используется такой синтаксис:

```
new ActiveXObject("Microsoft.XMLHTTP")
```

Например:

```
var xhrObj = new ActiveXObject("Microsoft.XMLHTTP");
```

Мы взяли приведенное ранее выражение и переписали его для Internet Explorer 5.* и 6.0.

Далее приведен код функции `createXMLHttpRequest`, создающей экземпляр объекта `XMLHttpRequest`, возвращающей его в качестве результата и рабо-

тающей во всех перечисленных ранее Web-обозревателях. Мы будем использовать ее для дальнейшей работы.

```
function createXMLHttpRequest() {
  if (typeof(XMLHttpRequest) == "undefined")
    return new ActiveXObject("Microsoft.XMLHTTP")
  else
    return new XMLHttpRequest();
}
```

Здесь мы проверяем тип объекта `XMLHttpRequest` и, если он равен "undefined" (то есть данный объект не поддерживается Web-обозревателем), применяем "новый" способ создания его экземпляра; в противном случае используем "старый" способ.

Создав экземпляр объекта `XMLHttpRequest`, мы должны задать параметры обмена данными. Это выполняется с помощью метода `open` данного объекта.

```
open(<метод отправки данных>,
↳<интернет-адрес серверной программы или файла XML>
↳[, true|false[, <имя пользователя>[, <пароль>]]])
```

Первый параметр этого метода задает метод отправки данных в виде строки "GET" или "POST". Оба метода отправки данных были описаны в *главе 12*.

В текущей реализации технологии AJAX серверной программе практически всегда передаются данные небольшого объема. Для этого используется метод GET. Метод POST редко применяется в AJAX; для передачи больших данных удобнее использовать традиционные способы, рассмотренные в *главе 12*.

Второй параметр задает интернет-адрес серверной программы, от которой будут получаться данные XML, или файла XML, если данные планируется получить из него. Интернет-адрес серверной программы может включать строку поиска, которую можно сформировать программно (см. *главы 11* и *12*); в этом случае следует выбрать метод передачи данных GET.

Третий — необязательный — параметр задает режим обмена данными. Он заслуживает самого обстоятельного разговора.

Если значение этого параметра равно `true`, или этот параметр не указан, используется асинхронный обмен данными. В этом случае после запроса данных у серверной программы или запуска загрузки файла XML (что выполняется вызовом метода `send`, который мы рассмотрим далее) Web-обозреватель не дожидается получения этих данных, а начинает выполнять следующий код. Когда данные будут получены, в экземпляре объекта `XMLHttpRequest`, получившего данные, возникнет событие `onReadyStateChange`, в обработчике

которого мы сможем их разобрать. Это обычная практика при реализации технологии AJAX.

Если значение третьего параметра метода `open` равно `false`, используется *синхронный обмен данными*. В таком случае после запроса данных у серверной программы или запуска загрузки файла XML Web-обозреватель будет ждать, пока данные не будут получены, и только после этого начнет выполнять следующий код сценария. Вероятно, это может быть иногда полезно, например, если получаемые данные гарантированно невелики и будут загружены достаточно быстро.

Необязательные четвертый и пятый параметры метода `open` задают имя пользователя и пароль. Они необходимы только в редких случаях, когда серверная программа, с которой мы собираемся работать, требует обязательной регистрации и не хочет иметь дела с "незнакомцами". На практике эти параметры практически никогда не используются.

Итак, параметры обмена данными мы задали. Теперь нам нужно собственно запросить данные у серверной программы или запустить загрузку файла XML. Это выполняется вызовом метода `send` объекта `XMLHttpRequest`. Он не возвращает значения, а в качестве параметра ему следует передать пустую строку.

> ***НА ЗАМЕТКУ***
>
> Вообще-то, в качестве единственного параметра методу `send` передается строка с данными, которые будут переданы серверной программой с использованием метода передачи данных POST. Однако нам самим придется формировать эту строку — Web-обозреватель за нас ничего не сделает. Поэтому, если все-таки возникнет потребность передать серверной программе данные методом POST, лучше пользоваться библиотеками, описанными в *приложении 4*.

```
var xhrObj = createXMLHttpRequest();
xhrObj.open("GET", "http://www.somesite.ru/program.exe");
xhrObj.send("");
```

Этот сценарий запускает загрузку данных, являющихся результатом работы серверной программы program.exe сайта **http://www.somesite.ru**.

```
var xhrObj = createXMLHttpRequest();
xhrObj.open("GET", "http://www.othersite.ru/data.xml");
xhrObj.send("");
```

А этот сценарий загружает файл data.xml с сайта **http://www.othersite.ru**.

После того как данные будут получены, нам нужно как-то получить к ним доступ. Для этого мы можем воспользоваться одним из специально предназначенных для этого свойств объекта `XMLHttpRequest`. Свойство `responseText`

возвращает строку, содержащую принятые данные. А более полезное для нас свойство `responseXML` возвращает ссылку на экземпляр объекта `Document`, представляющий фрагмент данных XML. Этот объект был описан в *главе 18*.

```
var xmlObj = xhrObj.responseXML;
var childTags = xmlObj.getElementsByTagName("child");
```

Данный сценарий помещает в переменную `childTags` коллекцию, содержащую все теги `<child>`, присутствующие в полученных данных XML. Использовать для этого метод `getElementsByTagName` удобнее всего.

Но стоп! Ведь ясно, что мы сможем получить доступ к принятым данным только после того, как они будут загружены. Но когда они будут загружены? Это зависит от того, синхронный или асинхронный обмен данными мы выбрали. (О синхронном и асинхронном обмене данными было сказано ранее.)

В случае если мы выбрали синхронный обмен данными, все просто. Web-обозреватель остановит выполнение сценария на вызове метода `send` на тот момент, когда будут загружаться данные, и продолжит только после завершения из загрузки. Значит, мы можем поместить код, выполняющий разбор принятых данных, сразу после вызова метода `send` и больше ни о чем не беспокоиться.

Но если мы выбрали асинхронный обмен данными, Web-обозреватель выполнит вызов метода `send` и сразу же, не дожидаясь загрузки данных, начнет выполнять следующий код. Если этот следующий код выполняет разбор принятых данных, он породит ошибку — ведь данные еще не приняты. Что делать в этом случае?

Во-первых, нам пригодится свойство `readyState` объекта `XMLHttpRequest`. Это свойство возвращает число, обозначающее состояние экземпляра данного объекта. Таких состояний и соответствующих им чисел пять:

- 0 — экземпляр объекта `XMLHttpRequest` создан, но параметры его еще не были заданы (не был вызван метод `open`);
- 1 — экземпляр объекта `XMLHttpRequest` был создан, и параметры его уже заданы, но получение данных еще не инициировано (не был вызван метод `send`);
- 2 — получение данных уже инициировано, но реально еще не началось;
- 3 — идет получение данных;
- 4 — данные получены.

Во-вторых, нам не обойтись без обработки события `onReadyStateChange` объекта `XMLHttpRequest`. Это событие возникает при любом изменении состояния экземпляра данного объекта, то есть значения свойства `readyState`.

Итак, дальнейший ход наших действий ясен. Мы создаем обработчик события `onReadyStateChange`, в его теле проверяем, не стало ли значение свойства `readyState` равным 4 (то есть загружены ли данные), и в случае успешности проверки эти данные разбираем.

```
function xhrDataLoaded() {
  if (xhrObj.readyState == 4) {
    . . .
  }
}
var xhrObj = createXMLHttpRequest();
xhrObj.open("GET", "http://www.othersite.ru/data.xml");
xhrObj.onreadystatechange = xhrDataLoaded;
xhrObj.send("");
```

Этот сценарий так и делает.

Кроме перечисленных ранее, нам могут пригодиться еще два свойства и один метод объекта `XMLHttpRequest`. Давайте их рассмотрим.

Свойство `status` возвращает *код ответа Web-сервера* — особое число, обозначающее состояние выполненного Web-обозревателем клиентского запроса. (О клиентских запросах было рассказано в *главе 1*.) Например, код 200 означает, что запрошенный файл успешно отправлен Web-обозревателю, а 404 — что запрошенный файл не найден. В последнем случае свойство `status` фактически вернет код ошибки.

```
function xhrDataLoaded() {
  if ((xhrObj.readyState == 4) $$ (xhrObj.status == 200)) {
    . . .
  }
}
var xhrObj = createXMLHttpRequest();
xhrObj.open("GET", "http://www.othersite.ru/data.xml");
xhrObj.onreadystatechange = xhrDataLoaded;
xhrObj.send("");
```

Это немного измененный вариант приведенного ранее сценария, в котором также проверяется код ответа Web-сервера — он должен быть равен 200 (запрос успешно выполнен). Хотя обычно так не поступают — достаточно проверить значение свойства `readyState`, как мы делали ранее.

Свойство `statusText` возвращает строку, описывающую состояние выполненного Web-обозревателем клиентского запроса. Так, при успешном выполнении запроса возвращается "OK", а в случае возникновении ошибки — описание этой ошибки.

Метод `abort` прерывает выполнение асинхронного обмена данными. Он не принимает параметров и не возвращает результата.

Больше об объекте `XMLHttpRequest` рассказывать нечего. Поэтому сразу перейдем к примерам.

Простая страница, реализующая технологию AJAX

Давайте создадим вот такой файл XML:

```
<langlist>
  <lang>
    <name>JavaScript</name>
    <type>Интерпретируемый</type>
  </lang>
  <lang>
    <name>VBScript</name>
    <type>Интерпретируемый</type>
  </lang>
  <lang>
    <name>Java</name>
    <type>Компилируемый</type>
  </lang>
  <lang>
    <name>C++</name>
    <type>Компилируемый</type>
  </lang>
  <lang>
    <name>C#</name>
    <type>Компилируемый</type>
  </lang>
  <lang>
    <name>Delphi</name>
    <type>Компилируемый</type>
  </lang>
</langlist>
```

Да-да, это все тот же знакомый нам по *главе 18* список языков программирования. Сохраним его под именем 19.1.xml в кодировке UTF-8. (Это можно сделать в Блокноте, поставляемом в составе Windows.) И напишем страницу,

которая будет загружать этот файл и выводить его содержимое на экран. Ее HTML-код приведен далее.

```
<HTML>
  <HEAD>
    <TITLE>AJAX</TITLE>
  </HEAD>
  <BODY>
    <SCRIPT>
      //Здесь необходимо поместить код, объявляющий функцию
      //createXMLHttpRequest

      function xhrDataLoaded() {
        if (xhrObj.readyState == 4) {
          var tableObj = document.createElement("TABLE");
          var tHeadObj = document.createElement("THEAD");
          var trObj = document.createElement("TR");
          var cellObj = document.createElement("TH");
          var textObj = document.createTextNode("Название");
          cellObj.appendChild(textObj);
          trObj.appendChild(cellObj);
          cellObj = document.createElement("TH");
          textObj = document.createTextNode("Категория");
          cellObj.appendChild(textObj);
          trObj.appendChild(cellObj);
          tHeadObj.appendChild(trObj);
          tableObj.appendChild(tHeadObj);
          var tBodyObj = document.createElement("TBODY");
          var langsObj = xhrObj.responseXML.getElementsByTagName("lang");
          for (var i = 0; i < langsObj.length; i++) {
            trObj = document.createElement("TR");
            cellObj = document.createElement("TD");
            textObj = document.createTextNode(langsObj[i].
            ↳getElementsByTagName("name")[0].firstChild.nodeValue);
            cellObj.appendChild(textObj);
            trObj.appendChild(cellObj);
            cellObj = document.createElement("TD");
            textObj = document.createTextNode(langsObj[i].
            ↳getElementsByTagName("type")[0].firstChild.nodeValue);
```

```
            cellObj.appendChild(textObj);
            trObj.appendChild(cellObj);
            tBodyObj.appendChild(trObj);
          }
          tableObj.appendChild(tBodyObj);
          document.body.appendChild(tableObj);
        }
      }

      var xhrObj = createXMLHttpRequest();
      xhrObj.open("GET", "http://localhost/19.1.xml");
      xhrObj.onreadystatechange = xhrDataLoaded;
      xhrObj.send("");
    </SCRIPT>
  </BODY>
</HTML>
```

Ничего особо нового по сравнению с первым примером из *главы 18* тут нет. Единственное — нам обязательно нужно поместить в этот код объявление функции `createXMLHttpRequest`, приведенное ранее. Данное объявление нужно вставить в то место кода, где находится предписывающий сделать это комментарий.

Сохраним эту страницу в файле 19.1.htm. И испытаем в действии.

Но тут мы столкнемся с одной сложностью. Дело в том, что различные Web-обозреватели имеют разные настройки безопасности по умолчанию. И эти настройки в некоторых случаях могут запрещать загрузку файлов XML. Так что наша страница, возможно, не будет работать в некоторых Web-обозревателях.

Чтобы обойти это ограничение, мы должны установить на своем компьютере Web-сервер, поместить файл страницы и файл XML в его корневую папку и открыть, обращаясь к этому Web-серверу. Для чего в нашем случае следует набрать в поле ввода интернет-адреса Web-обозревателя **http://localhost/19.1.htm**. При загрузке файлов XML с Web-сервера ограничения, вызванные настройками безопасности Web-обозревателей, действовать уже не будут, и наша страница 19.1.htm будет успешно загружена.

Кстати, если вы посмотрите на представленный ранее код, то увидите, что вторым параметром метода `open` объекта `XMLHttpRequest` указан как раз интернет-адрес, подобный тому, что мы набирали для загрузки страницы 19.1.htm с Web-сервера. То есть мы создали страницы уже с прицелом на то, что она будет загружаться с Web-сервера.

Загрузка данных в ответ на действия посетителя

Второй пример реализации технологии AJAX будет заметно сложнее. Мы реализуем загрузку данных XML в ответ на действия посетителя. Пусть наша страница изначально загружает только список языков программирования, а категорию и краткое описание подгружает и выводит на экран только после наведения курсора мыши на название языка.

Список языков программирования будет иметь такой вид:

```
<langlist>
  <lang id="0">JavaScript</lang>
  <lang id="1">VBScript</lang>
  <lang id="2">Java</lang>
  <lang id="3">C++</lang>
  <lang id="4">C#</lang>
  <lang id="5">Delphi</lang>
</langlist>
```

Здесь мы использовали атрибут `id` тега `<lang>` для указания номера языка программирования. Этот номер будет указывать на файл с подробными сведениями о данном языке.

Сохраним список в файле под именем 19.2.xml в кодировке UTF-8.

Следующий шаг — создание шести файлов с подробными сведениями о представленных в списке языках. Чтобы упростить код, разбирающий данные, используем в качестве имен этих файлов номера языков программирования, заданные атрибутом `id` тега `<lang>` в приведенном ранее коде: 0.xml, 1.xml и т. д.

Далее приведено содержимое файла 0.xml, описывающего язык JavaScript.

```
<langdata>
  <type>Интерпретируемый</type>
  <desc>Используется для написания Web-сценариев.</desc>
</langdata>
```

Остальные файлы с описаниями языков программирования имеют аналогичное содержимое.

HTML-код самой страницы приведен далее.

```
<HTML>
  <HEAD>
    <TITLE>AJAX</TITLE>
```

```
</HEAD>
<BODY>
  <SCRIPT>
    //Здесь необходимо поместить код, объявляющий функцию
    //createXMLHttpRequest

    //Здесь необходимо поместить код, объявляющий функции
    //getBrowser и getBrowserStrict. Этот код приведен в главе 7

    function trMouseOverIEOpera() {
      xhr2Obj = createXMLHttpRequest();
      xhr2Obj.open("GET", "http://localhost/" + event.srcElement.id +
      ".xml");
      xhr2Obj.onreadystatechange = xhr2DataLoaded;
      xhr2Obj.send("");
    }

    function trMouseOverFF(evt) {
      xhr2Obj = createXMLHttpRequest();
      xhr2Obj.open("GET", "http://localhost/" + evt.target.id +
      ".xml");
      xhr2Obj.onreadystatechange = xhr2DataLoaded;
      xhr2Obj.send("");
    }

    function trMouseOut() {
      var outputObj = document.getElementById("output");
      while (outputObj.childNodes.length > 0)
        outputObj.removeChild(outputObj.firstChild);
    }

    function xhrDataLoaded() {
      if (xhrObj.readyState == 4) {
        var tableObj = document.createElement("TABLE");
        var tBodyObj = document.createElement("TBODY");
        var langsObj = xhrObj.responseXML.getElementsByTagName("lang");
        for (var i = 0; i < langsObj.length; i++) {
          trObj = document.createElement("TR");
          cellObj = document.createElement("TD");
```

```
      cellObj.id = langsObj[i].getAttributeNode("id").value;
      if (getBrowserStrict() == "FF")
        cellObj.onmouseover = trMouseOverFF
      else
        cellObj.onmouseover = trMouseOverIEOpera;
      cellObj.onmouseout = trMouseOut;
      textObj = document.createTextNode(langsObj[i].firstChild.
      ↳nodeValue);
      cellObj.appendChild(textObj);
      trObj.appendChild(cellObj);
      tBodyObj.appendChild(trObj);
    }
    tableObj.appendChild(tBodyObj);
    var contObj = document.getElementById("cont");
    contObj.appendChild(tableObj);
  }
}

function xhr2DataLoaded() {
  if (xhr2Obj.readyState == 4) {
    var outputObj = document.getElementById("output");
    while (outputObj.childNodes.length > 0)
      outputObj.removeChild(outputObj.firstChild);
    var langDataObj = xhr2Obj.responseXML.
    ↳getElementsByTagName("langdata")[0];
    var text1Obj = document.createTextNode(langDataObj.
    ↳getElementsByTagName("type")[0].firstChild.nodeValue);
    outputObj.appendChild(text1Obj);
    var brObj = document.createElement("BR");
    outputObj.appendChild(brObj);
    var text2Obj = document.createTextNode(langDataObj.
    ↳getElementsByTagName("desc")[0].firstChild.nodeValue);
    outputObj.appendChild(text2Obj);
  }
}

var xhrObj = createXMLHttpRequest();
xhrObj.open("GET", "http://localhost/19.2.xml");
```

```
      xhrObj.onreadystatechange = xhrDataLoaded;
      xhrObj.send("");
      var xhr2Obj = null;
    </SCRIPT>
    <DIV ID="cont"></DIV>
    <DIV ID="output"> </DIV>
  </BODY>
</HTML>
```

Все приемы формирования содержимого таблицы и обработки событий, использованные в этом коде, нам давно знакомы. Остановимся только на некоторых деталях.

Прежде всего, нам обязательно нужно поместить в этот код объявление функции `createXMLHttpRequest`, приведенное ранее, и объявления функций `getBrowser` и `getBrowserStrict`, приведенные в *главе 7*. Эти объявления нужно вставить в те места кода, где находятся предписывающие сделать это комментарии.

При формировании таблицы со списком языков программирования мы, во-первых, выводим в ней только названия языков. Во-вторых, мы присваиваем всем тегам `<TD>` имена с помощью атрибутов `ID`, а в качестве имен задаем номера языков. В-третьих, мы привязываем к событиям `onMouseOver` и `onMouseOut` тегов `<TD>` обработчики событий; первый обработчик будет при наведении курсора мыши на ячейку таблицы выводить в контейнер `output` сведения о языке, чье имя находится в этой ячейке, второй — при уводе курсора мыши очищать этот контейнер. Саму таблицу мы выводим в другой контейнер — `cont`.

Вы спросите, а почему бы не привязать обработчики событий к тегам `<TR>`, и этим же тегам дать имена? Дело в том, что эти события возникают сначала в ячейке таблицы (теге `<TD>`), а уже потом всплывают в строку (тег `<TR>`). Если мы будем обрабатывать события в тегах `<TR>`, то не сможем получить их имена — при обращении к свойству `srcElement` или `target` объекта `Event` мы получим ячейку, в которой возникло событие, и никак не сможем получить имя строки, в которой находится эта ячейка. Так что лучше обрабатывать все эти события именно в ячейках, а не в строках таблицы.

Обработчик события `onMouseOver` сначала получает имя ячейки (тега `<TD>`), в которой возникло событие, на его основе формирует интернет-адрес файла XML, где содержатся сведения о языке программирования, чье название находится в этой ячейке, и инициирует его загрузку. Когда файл загрузится, Web-обозреватель вызовет функцию `xhr2DataLoaded`, которая выполнит разбор полученных данных и выведет их в контейнер `output`, предварительно

удалив предыдущее содержимое этого контейнера. При этом сначала будет выведена категория языка, после нее будет поставлен тег разрыва строк `<BR>`, а за ним — описание языка.

А обработчик события `onMouseOut` просто очищает содержимое контейнера `output`.

Сохраним эту страницу под именем 19.2.htm, поместим ее файлы 19.2.xml и 0.xml-5.xml в корневую папку нашего Web-сервера и наберем в поле ввода интернет-адреса в Web-обозревателе **http://localhost/19.2.htm**. Когда страница будет загружена, наведем курсор мыши на любой язык программирования — и ниже появятся его категория и описание.

Вот и все о технологии AJAX. По крайней мере, о ее клиентской части. Серверные программы мы в этой книге разрабатывать не будем — она посвящена исключительно клиентскому программированию. Желающие узнать больше могут обратиться к книгам, посвященным серверному программированию, и тем самым сделать третий — последний — шаг в технологию AJAX.

А эта книга закончилась!

Заключение

Вот и закончилась эта книга.

Мы узнали многое. Мы познакомились с языками HTML и CSS и научились создавать простейшие, "неживые", Web-странички. Мы изучили язык JavaScript и принципы написания Web-сценариев и смогли добавить своим страничкам "жизни". Мы рассмотрели возможности, предлагаемые современными Web-обозревателями программистам, и использовали их для создания все более и более сложных страниц. Их уже и страничками называть как-то несерьезно!..

В общем, мы стали Web-программистами. Вполне серьезными, знающими, знакомыми со многими хитростями и тонкостями, необходимыми в работе. И готовыми взяться за Web-программирование по-настоящему, а возможно, и профессионально.

Но все ли мы знаем о Web-дизайне и Web-программировании? Отнюдь. Никому еще не удавалось объять необъятное. И даже авторы самых серьезных, невероятно толстых книг что-то да упускают из виду.

Прежде всего, мы довольно поверхностно рассмотрели сам Web-дизайн. Мы изучили HTML и CSS только в том объеме, чтобы начать создавать странички, но уж никак не в том, чтобы стать грамотными Web-дизайнерами. Так что в этом плане желающие посвятить себя Web-дизайну должны продолжить свое обучение.

Мы не рассмотрели многие возможности, предлагаемые современными Web-обозревателями. Internet Explorer, возможностям которого мы посвятили несколько глав, — весьма развитая программа, в которой стоит покопаться. Так, мы не рассмотрели технологию HTML+TIME, позволяющую создавать на страницах простую анимацию практически без программирования. Мы не познакомились вплотную со стандартными поведениями — это слишком

большая тема, которой впору посвятить отдельную книгу. Да и другие "вкусности" этой программы остались нами не рассмотренными.

А уж в технологию AJAX мы сделали только два первых шага. AJAX — это ведь не только клиентская часть (Web-сценарии, выполняющие отправку, прием и вывод данных), но и серверная (серверная программа, обрабатывающая данные и возвращающая результат). А серверное программирование — тема очень большая, которой посвящены отдельные книги, не менее толстые, чем эта.

К тому же, все течет, все изменяется. Скоро выйдут новые версии Web-обозревателей: Internet Explorer 8 и Firefox 3.0. Они будут поддерживать дополнительные возможности, о которых на данный момент известно только в общих чертах.

Так что нам придется продолжать свое образование. В этом нам помогут книги и сайты соответствующей тематики. Книги можно найти в книжных магазинах, а интернет-адреса сайтов мы приведем далее.

http://msdn.microsoft.com/ie/

"Домашний" сайт Internet Explorer. Исчерпывающее описание Web-обозревателя фирмы Microsoft, особенностей поддержки им HTML, CSS, JavaScript и многого другого. К несчастью, читать эту документацию можно только с сайта.

http://www.opera.com/

"Домашний" сайт Opera. Все то же самое: описание Web-обозревателя, особенности поддержки HTML, CSS, JavaScript и мн. др.

http://www.mozilla.org/

"Домашний" сайт Mozilla и Firefox.

http://www.w3c.org/

Сайт самого великого и ужасного комитета W^3C. Все интернет-стандарты в удобном для чтения виде.

http://webreview.com/

Огромное количество документации по HTML, CSS, JavaScript, Web-обозревателям.

http://www.webmascon.com/

Электронный интернет-журнал, посвященный интернет-технологиям. Все статьи на русском языке.

http://www.citforum.ru/

"Город" электронной документации по компьютерным и интернет-технологиям. Все по-русски. В числе прочего, есть и переводы стандартов с сайта комитета W^3C.

http://html.krsk.ru/

Этот сайт называется "Рецепты HTML". Здесь можно найти различные описания, статьи и полезные советы.

Вот и все. Настала пора автору попрощаться. До свидания!

Владимир Дронов.

mail@bhv.ru

http://vgi.volsu.ru:8000/~Vladimir.Dronov/

ПРИЛОЖЕНИЯ

Приложение 1

Часто используемые теги и атрибуты HTML, объявленные стандартами как устаревшие

В этом приложении перечислены и описаны теги и атрибуты HTML, все еще часто применяющиеся на практике, но объявленные стандартами как устаревшие и не рекомендованные к употреблению.

> ***Внимание!***
>
> Некоторые из этих тегов и атрибутов могут не поддерживаться отдельными Web-обозревателями.

Современные интернет-стандарты рекомендуют применять вместо этих тегов и атрибутов стили CSS, выполняющие аналогичные задачи.

Устаревшие теги

Парный тег `<B>` выводит текст, являющийся его содержимым, полужирным шрифтом.

```
<P>Это <B>полужирный</B> текст.</P>
```

Никакого особого значения при этом, в отличие от тега `<STRONG>`, тексту не дается.

Одинарный тег `<BASEFONT>` задает параметры шрифта по умолчанию для страницы. Весь текст, для которого не были заданы параметры шрифта, будет выводиться этим шрифтом.

```
<BASEFONT [FACE="<список имен шрифтов, разделенных запятыми>"]
↳[SIZE="<размер шрифта>"]>
```

Этот тег может присутствовать либо в секции заголовка страницы, либо в секции ее тела; в последнем случае он должен находиться в самом начале секции тела, сразу после открывающего тега <BODY>.

Необязательный атрибут FACE задает список имен шрифтов. Сначала на клиентском компьютере ищется первый шрифт из списка, в случае неудачного поиска — второй, потом третий и т. д. Кроме имен конкретных шрифтов можно использовать имена семейств шрифтов (см. *главу 3*).

Необязательный атрибут SIZE задает размер шрифта в виде числа от 1 (минимальный размер) до 7 (максимальный размер). Эти числа соответствуют значениям атрибута стиля font-size, задающим фиксированные размеры шрифта (см. *главу 3*).

```
<BODY>
  <BASEFONT FONT="Verdana" SIZE="5">
  <P>Этот текст набран шрифтом Verdana размером 5.</P>
  . . .
```

Парный тег <BIG> выделяет текст, являющийся его содержимым, шрифтом увеличенного на одну ступень размера. Так, если размер шрифта родительского элемента страницы равен 4, содержимое тега <BIG> будет выведено шрифтом, имеющим размер 5.

```
<P>Это <BIG>большой</BIG> текст.</P>
```

Парный тег <CENTER> выводит свое содержимое как отдельный абзац, выровненный по центру.

```
<CENTER>Этот абзац выровнен по центру</CENTER>
```

Парный тег <DIR> создает маркированный список, как и тег <UL> (см. *главу 2*).

```
<DIR>
  <LI>HTML;</LI>
  <LI>CSS;<LI>
  <LI>JavaScript.</LI>
</DIR>
```

Парный тег <FONT> выводит свое содержимое шрифтом с заданными параметрами.

```
<FONT [FACE="<список имен шрифтов, разделенных запятыми>"]
↳[SIZE="<размер шрифта>"] [COLOR="<цвет текста>"]>
  <текст, выводимый заданным шрифтом>
</FONT>
```

Необязательные атрибуты `FACE` и `SIZE` знакомы нам по тегу `<BASEFONT>`. Необязательный атрибут `COLOR` задает цвет текста.

```
<P>Это <FONT SIZE="2" COLOR="#0000FF">маленький синий</COLOR> текст.<P>
```

Парный тег `<I>` выводит текст, являющийся его содержимым, курсивом.

```
<P>Это <I>курсив</I>.</P>
```

Никакого особого значения при этом, в отличие от тега `<EM>`, тексту не дается.

Парный тег `<LISTING>` аналогичен тегу `<PRE>` (см. *главу 2*) — он выводит свое содержимое моноширинным шрифтом с сохранением форматирования как отдельный абзац.

```
<LISTING>
Этот текст будет выведен моноширинным шрифтом с сохранением
форматирования.
</LISTING>
```

Парный тег `<MENU>` создает маркированный список, как и тег `<UL>` (см. *главу 2*).

```
<MENU>
  <LI>HTML;</LI>
  <LI>CSS;<LI>
  <LI>JavaScript.</LI>
</MENU>
```

Парный тег `<PLAINTEXT>` аналогичен тегу `<PRE>` (см. *главу 2*) — он выводит свое содержимое моноширинным шрифтом с сохранением форматирования как отдельный абзац.

```
<PLAINTEXT>
Этот текст будет выведен моноширинным шрифтом с сохранением
форматирования.
</PLAINTEXT>
```

Парный тег `<S>` выводит свое содержимое зачеркнутым.

```
<P>Это <S>зачеркнутый</S> текст.</P>
```

Парный тег `<SMALL>` выделяет текст, являющийся его содержимым, шрифтом уменьшенного на одну ступень размера. Так, если размер шрифта родительского элемента страницы равен 4, содержимое тега `<SMALL>` будет выведено шрифтом, имеющим размер 3.

```
<P>Это <SMALL>маленький</SMALL> текст.</P>
```

Парный тег `<STRIKE>` выводит свое содержимое зачеркнутым.

```
<P>Это <STRIKE>зачеркнутый</STRIKE> текст.</P>
```

Парный тег <TT> аналогичен тегам <CODE>, <KBD> и <SAMP> (см. *главу 2*) — он выводит свое содержимое моноширинным шрифтом.

```
<P>Это <TT>моноширинный</TT> текст.</P>
```

Никакого особого значения при этом тексту не дается.

Парный тег <U> выводит свое содержимое подчеркнутым.

```
<P>Это <U>подчеркнутый</U> текст.</P>
```

Парный тег <XMP> аналогичен тегу <PRE> (см. *главу 2*) — выводит свое содержимое моноширинным шрифтом с сохранением форматирования как отдельный абзац.

```
<XMP>
Этот текст будет выведен моноширинным шрифтом с сохранением
форматирования.
</XMP>
```

Устаревшие атрибуты тегов

Необязательный атрибут ALIGN позволяет задать выравнивание элемента страницы или его содержимого.

Для графических изображений, модулей расширения, элементов ActiveX и графических кнопок этот атрибут поддерживает следующие значения:

- "absbottom" — выравнивание низа элемента страницы по самому низу текста родителя;
- "absmiddle" — выравнивание центра элемента страницы по центру текста родителя;
- "baseline" — выравнивание низа элемента страницы по базовой линии текста родителя;
- "bottom" — выравнивание низа элемента страницы по низу текста родителя;
- "left" — элемент страницы сдвигается к левому краю родителя, весь окружающий его текст обтекает изображение справа (значение по умолчанию);
- "middle" — выравнивание центра элемента страницы по центру родителя;
- "right" — элемент страницы сдвигается к правому краю родителя, весь окружающий его текст обтекает изображение слева;
- "texttop" — выравнивание верха элемента страницы по самому верху текста родителя;
- "top" — выравнивание верха элемента страницы по верху текста родителя.

```
<IMG SRC="someimage.gif" ALIGN="middle">
```

Центр этого изображения будет выровнен по центру родителя.

Для блочного контейнера, горизонтальной линии, заголовка, абзаца, строки и ячейки таблицы этот атрибут поддерживает следующие значения:

- "center" — выравнивание содержимого элемента страницы по центру родителя;
- "justify" — выравнивание содержимого элемента страницы по ширине родителя;
- "left" — выравнивание содержимого элемента страницы по левому краю родителя (значение по умолчанию);
- "right" — выравнивание содержимого элемента страницы по правому краю родителя.

```
<H1 ALIGN="center">Этот заголовок будет находиться в центре родителя<H1>
```

Для таблицы (тега `<TABLE>`) этот атрибут поддерживает следующие значения:

- "center" — таблица сдвигается в центр родителя;
- "left" — таблица сдвигается к левому краю родителя;
- "right" — таблица сдвигается к правому краю родителя.

```
<TABLE ALIGN="right">
```

Эта таблица будет сдвинута к правому краю родителя.

Необязательный атрибут `ALINK` тега `<BODY>` задает цвет активной гиперссылки (той, на которой в данный момент щелкают мышью).

```
<BODY ALINK="#FF0000">
  <P><A HREF="somepage.htm">Эта гиперссылка при щелчке на ней станет
  красной>/A></P>
```

Необязательный атрибут `BACKGROUND`, поддерживаемый секцией тела страницы, таблицей и ячейкой таблицы, задает интернет-адрес фонового изображения, соответственно, для самой страницы, таблицы и ее ячейки.

```
<BODY BACKGROUND="somebackground.jpg">
  <TABLE BACKGROUND="tablebackground.jpg">
    <TR>
      <TD BACKGROUND="cellbackground.jpg">
```

Необязательный атрибут `BGCOLOR`, поддерживаемый секцией тела страницы, таблицей, секцией, строкой и ячейкой таблицы, задает цвет фона, соответственно, для самой страницы, таблицы, ее секции, строки и ячейки.

```
<BODY BGCOLOR="#FF0000">
  <TABLE BGCOLOR="#00FF00">
    <TBODY BGCOLOR="#0000FF">
```

```
<TR BGCOLOR="#CCCCCC">
  <TD BGCOLOR="#FFFFFF">
```

Если для какого-либо из перечисленных ранее элементов страницы с помощью рассмотренного ранее атрибута BACKGROUND задано фоновое изображение, атрибут BGCOLOR игнорируется.

Атрибут без значения BGPROPERTIES тега <BODY> управляет прокруткой фона. Если он указан, фон не будет прокручиваться вместе с содержимым страницы, если не указан — будет.

Необязательный атрибут BORDER поддерживается графическими изображениями, таблицами (тегом <TABLE>) и элементами ActiveX и задает толщину рамки вокруг элемента страницы в пикселах.

```
<TABLE BORDER="2">
```

Эта таблица будет иметь рамку толщиной в 2 пиксела.

```
<IMG SRC="someimage.gif" BORDER="0">
```

Это изображение не будет иметь рамки.

Необязательный атрибут COLOR поддерживается горизонтальной линией и задает ее цвет.

Атрибут без значения COMPACT поддерживается всеми тегами списков и позволяет убрать дополнительное пространство между маркером или нумерацией и текстом пункта списка. Если этот атрибут указан, дополнительное пространство будет убрано, если не указан — не будет.

```
<UL COMPACT>
  <LI>HTML;</LI>
  <LI>CSS;<LI>
  <LI>JavaScript.</LI>
</UL>
```

Необязательный атрибут HEIGHT, поддерживаемый графическими изображениями, модулями расширения, элементами ActiveX, таблицами (тегом <TABLE>), строками и ячейками таблицы, задает высоту элемента страницы в пикселах или процентах от высоты родителя.

```
<IMG SRC="someimage.gif" HEIGHT="400">
```

Это изображение будет иметь высоту в 400 пикселов.

```
<TABLE HEIGHT="80%">
```

Эта таблица будет иметь высоту 80% от высоты родителя.

```
<TD HEIGHT="25">Эта ячейка будет иметь высоту в 25 пикселов</TD>
```

Атрибут без значения HIDDEN поддерживается модулями расширения и позволяет скрыть данный модуль. Если этот атрибут указан, модуль расширения будет скрыт, если не указан — будет присутствовать на странице.

Необязательный атрибут HSPACE, поддерживаемый графическими изображениями, графическими кнопками и элементами ActiveX, задает величину отступа по горизонтали между вертикальными границами элемента страницы и окружающим его содержимым в пикселах.

```
<IMG SRC="someimage.gif" HSPACE="2">
```

Между границами этого изображения и окружающим его содержимым будет сделан горизонтальный отступ в 2 пиксела.

Необязательный атрибут LINK тега <BODY> задает цвет непосещенных гиперссылок.

```
<BODY LINK="#0000FF">
  <P><A HREF="somepage.htm">Эта гиперссылка изначально будет
  синей>/A></P>
```

Атрибут без значения NOWRAP, поддерживаемый секцией тела страницы, ячейками таблицы и пунктами списка определений (см. *главу 2*), позволяет запретить перенос строк. Если он указан, перенос строк не будет выполняться в любом случае, если не указан — будет выполняться.

```
<TH NOWRAP>Дли-и-и-и-и-и-и-и-и-и-и-и-и-и-и-и-и-и-и-инный заголовок</TH>
```

Содержимое этой ячейки не будет переноситься на другую строку.

Необязательный атрибут RULES таблицы (тега <TABLE>) позволяет задать, какие разделительные линии или рамки будут присутствовать в данной таблице. Он может иметь следующие значения:

- "all" — вокруг всех ячеек будут нарисованы рамки;
- "cols" — между столбцами будут присутствовать вертикальные разделительные линии;
- "groups" — между секциями будут присутствовать горизонтальные разделительные линии;
- "none" — никакие рамки или разделительные линии не будут присутствовать;
- "rows" — между строками будут присутствовать горизонтальные разделительные линии.

```
<TABLE RULES="cols">
```

Между столбцами этой таблицы будут присутствовать вертикальные разделительные линии.

Необязательный атрибут SCROLL секции тела страницы (тега <BODY>) управляет отображением полос прокрутки. Значение "auto" задает их отображение лишь в том случае, если содержимое страницы не помещается в окне Web-обозревателя, значение "yes" задает отображение полос прокрутки в любом

случае, даже если они и не нужны (это значение по умолчанию), а значение "no" убирает их совсем.

Необязательный атрибут SIZE, поддерживаемый горизонтальной линией, задает ее ширину в пикселах.

Необязательный атрибут TEXT тега <BODY> задает цвет текста страницы.

```
<BODY TEXT="#FFFFFF" BGCOLOR="#000000">
  <P>Белый текст на черном фоне...</P>
```

Необязательный атрибут TYPE тегов маркированного и нумерованного списка и пункта списка задает вид маркера или нумерации пунктов. Его доступные значения таковы:

- "1" — арабские цифры (значение по умолчанию для нумерованного списка);
- "a" — маленькие латинские буквы;
- "A" — большие латинские буквы;
- "i" — римские цифры, набранные символами нижнего регистра;
- "I" — римские цифры, набранные символами верхнего регистра;
- "disc" — кружок с заливкой (значение по умолчанию для маркированного списка);
- "circle" — кружок без заливки;
- "square" — квадратик с заливкой.

```
<UL TYPE="circle">
  <LI>HTML;</LI>
  <LI>CSS;<LI>
  <LI TYPE="square">JavaScript.</LI>
</UL>
```

Первые два пункта этого списка будут помечены кружком без заливки, а третий — квадратиком с заливкой.

Необязательный атрибут VALIGN, поддерживаемый секциями, строками и ячейками таблицы, задает вертикальное выравнивание текста в ячейках таблицы. Он может принимать следующие значения:

- "baseline" — базовая линия первой строки текста всех ячеек данной строки таблицы будет находиться на одной прямой;
- "bottom" — выравнивание по низу ячейки;
- "middle" — выравнивание по центру ячейки (значение по умолчанию);
- "top" — выравнивание по верху ячейки.

```
<TD VALIGN="bottom">Этот текст будет выровнен по низу ячейки</TD>
```

Необязательный атрибут `VLINK` тега `<BODY>` задает цвет посещенных гиперссылок.

```
<BODY VLINK="#00FF00">
  <P><A HREF="somepage.htm">Эта гиперссылка после щелчка на ней станет
  зеленой>/A></P>
```

Необязательный атрибут `VSPACE`, поддерживаемый графическими изображениями, графическими кнопками и элементами ActiveX, задает величину отступа по вертикали между горизонтальными границами элемента страницы и окружающим его содержимым в пикселах.

```
<IMG SRC="someimage.gif" VSPACE="2">
```

Между границами этого изображения и окружающим его содержимым будет сделан вертикальный отступ в 2 пиксела.

Необязательный атрибут `WIDTH`, поддерживаемый графическими изображениями, модулями расширения, элементами ActiveX, таблицами (тегом `<TABLE>`), строками и ячейками таблицы, задает ширину элемента страницы в пикселах или процентах от ширины родителя.

```
<IMG SRC="someimage.gif" WIDTH="300">
```

Это изображение будет иметь ширину в 300 пикселов.

```
<TABLE WIDTH="50%">
```

Эта таблица будет иметь ширину 50% от ширины родителя.

```
<TD WIDTH="40">Эта ячейка будет иметь ширину в 40 пикселов</TD>
```

Приложение 2

Специальные символы HTML

В табл. П2.1 представлены коды и литералы специальных символов, определенных в стандарте HTML.

Таблица П2.1. *Специальные символы HTML*

Символ	Десятичный код	Литерал	Описание
"	&#34;	&quot;	Кавычка
&	&#38;	&amp;	Амперсанд
<	&#60;	&lt;	Знак "меньше"
>	&#62;	&gt;	Знак "больше"
			Неразрывный пробел (по нему никогда не производится перенос строки)
¡	&#161;	&iexcl;	Перевернутый восклицательный знак
¢	&#162;	&cent;	Цент
£	&#163;	&pound;	Фунт стерлингов
¤	&#164;	&curren;	Общий символ валюты
¥	&#165;	&yen;	Йена
¦	&#166;	¦ или &brkbar;	Разорванная вертикальная черта
§	&#167;	&sect;	Параграф
¨	&#168;	¨ или ¨	Умляут

Таблица П2.1 *(продолжение)*

Символ	Десятичный код	Литерал	Описание
©	&#169;	&copy;	Знак авторского права
ª	&#170;	&ordf;	Знак женского рода
«	&#171;	&laquo;	Открывающая угловая кавычка
¬	&#172;	&not	Знак логической инверсии
	&#173;	&shy;	"Мягкий" перенос
®	&#174;	&reg;	Зарегистрированная торговая марка
¯	&#175;	¯ или &hibar;	Знак долготы над гласным
°	&#176;	&deg;	Градус
±	&#177;	&plusmn;	Плюс-минус
²	&#178;	&sup2;	Квадрат (вторая степень)
³	&#179;	&sup3;	Куб (третья степень)
´	&#180;	&acute;	Ударение
µ	&#181;	&micro;	Микро (мю)
¶	&#182;	&para;	Конец абзаца
·	&#183;	&middot;	Точка сверху
¸	&#184;	&cedil;	Седиль
¹	&#185;	&sup1;	Первая степень
º	&#186;	&ordm;	Знак мужского рода
»	&#187;	&raquo;	Закрывающая угловая кавычка
¼	&#188;	&frac14;	Одна четвертая
½	&#189;	&frac12;	Одна вторая
¾	&#190;	&frac34;	Три четвертых
¿	&#191;	&iquest;	Перевернутый вопросительный знак
À	&#192;	&Agrave;	Большая А глухая
Á	&#193;	&Aacute;	Большая А звонкая
Â	&#194;	&Acirc;	Большая А с циркумфлексом
Ã	&#195;	&Atilde;	Большая А с тильдой
Ä	&#196;	&Auml;	Большая А с умляутом

Таблица П2.1 *(продолжение)*

Символ	Десятичный код	Литерал	Описание
Å	&#197;	&Aring;	Большая A с кружком
Æ	&#198;	&AElig;	Большая лигатура AE
Ç	&#199;	&Ccedil;	Большая C с седилью
È	&#200;	&Egrave;	Большая E глухая
É	&#201;	&Eacute;	Большая E звонкая
Ê	&#202;	&Ecirc;	Большая E с циркумфлексом
Ë	&#203;	&Euml;	Большая E с умляутом
Ì	&#204;	&Igrave;	Большая I глухая
Í	&#205;	&Iacute;	Большая I звонкая
Î	&#206;	&Icirc;	Большая I с циркумфлексом
Ï	&#207;	&Iuml;	Большая I с умляутом
Ð	&#208;	&ETH;	Большая исландская "Эт"
Ñ	&#209;	&Ntilde;	Большая N с тильдой
Ò	&#210;	&Ograve;	Большая O глухая
Ó	&#211;	&Oacute;	Большая O звонкая
Ô	&#212;	&Ocirc;	Большая O с циркумфлексом
Õ	&#213;	&Otilde;	Большая O с тильдой
Ö	&#214;	&Ouml;	Большая O с умляутом
×	&#215;	&times;	Знак умножения
Ø	&#216;	&Oslash;	Большая O зачеркнутая
Ù	&#217;	&Ugrave;	Большая U глухая
Ú	&#218;	&Uacute;	Большая U звонкая
Û	&#219;	&Ucirc;	Большая U с циркумфлексом
Ü	&#220;	&Uuml;	Большая U с умляутом
Ý	&#221;	&Yacute;	Большая Y звонкая
Þ	&#222;	&THORN;	Большая исландская "Торн"
ß	&#223;	&szlig;	Лигатура ss
à	&#224;	&agrave;	Малая A глухая
á	&#225;	&aacute;	Малая A звонкая
â	&#226;	&acirc;	Малая A с циркумфлексом

Таблица П2.1 *(окончание)*

Символ	Десятичный код	Литерал	Описание
ã	&#227;	&atilde;	Малая A с тильдой
ä	&#228;	&auml;	Малая A с умляутом
å	&#229;	&aring;	Малая A с кружком
æ	&#230;	&aelig;	Малая лигатура AE
ç	&#231;	&ccedil;	Малая C с седилью
è	&#232;	&egrave;	Малая E глухая
é	&#233;	&eacute;	Малая E звонкая
ê	&#234;	&ecirc;	Малая E с циркумфлексом
ë	&#235;	&euml;	Малая E с умляутом
ì	&#236;	&igrave;	Малая I глухая
í	&#237;	&iacute;	Малая I звонкая
î	&#238;	&icirc;	Малая I с циркумфлексом
ï	&#239;	&iuml;	Малая I с умляутом
ð	&#240;	&eth;	Малая исландская "Эт"
ñ	&#241;	&ntilde;	Малая N с тильдой
ò	&#242;	&ograve;	Малая O глухая
ó	&#243;	&oacute;	Малая O звонкая
ô	&#244;	&ocirc;	Малая O с циркумфлексом
õ	&#245;	&otilde;	Малая O с тильдой
ö	&#246;	&ouml;	Малая O с умляутом
÷	&#247;	&divide;	Знак деления
ø	&#248;	&oslash;	Малая O зачеркнутая
ù	&#249;	&ugrave;	Малая U глухая
ú	&#250;	&uacute;	Малая U звонкая
û	&#251;	&ucirc;	Малая U с циркумфлексом
ü	&#252;	&uuml;	Малая U с умляутом
ý	&#253;	&yacute;	Малая Y звонкая
þ	&#254;	&thorn;	Малая исландская "Торн"
ÿ	&#255;	&yuml;	Малая Y с умляутом

Приложение 3

Коды и обозначения цветов

В табл. П3.1 перечислены обозначения и RGB-коды цветов, определенные в стандартах HTML и CSS.

Таблица П3.1. *Коды и обозначения цветов*

Обозначение	RGB-код	Цвет
aliceBlue	F0F8FF	Блекло-голубой
antuqueWhite	FAEBD7	Античный белый
aqua	00FFFF	Синий
aquamarine	7FFFD4	Аквамарин
azure	F0FFFF	Лазурь
beige	F5F5DC	Бежевый
bisque	FFE4C4	Бисквитный
black	000000	Черный
blanchedalmond	FFEBCD	Светло-кремовый
blue	0000FF	Голубой
blueviolet	8A2BE2	Светло-фиолетовый
brown	A52A2A	Коричневый
burlywood	DEB887	Старого дерева
cadetblue	5F9EA0	Блеклый серо-голубой
chartreuse	7FFF00	Фисташковый
chocolate	D2691E	Шоколадный

Таблица П3.1 *(продолжение)*

Обозначение	RGB-код	Цвет
coral	FF7F50	Коралловый
cornflowerblue	6495ED	Васильковый
cornsilk	FFF8DC	Темно-зеленый
crimson	DC143C	Малиновый
cyan	00FFFF	Циан
darkblue	00008B	Темно-голубой
darkcyan	008B8B	Темный циан
darkgoldenrod	B8860B	Темный красно-золотой
darkgray	A9A9A9	Темно-серый
darkgreen	006400	Темно-зеленый
darkkhaki	BDB76B	Темный хаки
darkmagenta	8B008B	Темный фуксин
darkolivegreen	556B2F	Темно-оливковый
darkorange	FF8C00	Темно-оранжевый
darkorchid	9932CC	Темно-орхидейный
darkred	8B0000	Темно-красный
darksalmon	E9967A	Темный оранжево-розовый
darkseagreen	8FBC8F	Темный морской волны
darkslateblue	483D8B	Темный серовато-синий
darkslategray	2F4F4F	Темный синевато-серый
darkturquoise	00CED1	Темно-бирюзовый
darkviolet	9400D3	Темно-фиолетовый
deeppink	FF1493	Темно-розовый
deepskyblue	00BFFF	Темный небесно-синий
dimgray	696969	Тускло-серый
dodgerblue	1E90FF	Тускло-васильковый
firebrick	B22222	Огнеупорного кирпича
floralwhite	FFFAF0	Цветочно-белый
forestgreen	228B22	Лесной зеленый
fuchsia	FF00FF	Фуксия

Таблица П3.1 *(продолжение)*

Обозначение	RGB-код	Цвет
fainsboro	DCDCDC	Светлый серо-фиолетовый
ghostwhite	F8F8FF	Туманно-белый
gold	FFD700	Золотой
goldenrod	DAA520	Красного золота
gray	808080	Серый
green	008000	Зеленый
greenyellow	ADFF2F	Желто-зеленый
honeydew	F0FFF0	Свежего меда
hotpink	FF69B4	Ярко-розовый
indianred	CD5C5C	Ярко-красный
indigo	4B0082	Индиго
ivory	FFFFF0	Слоновой кости
khaki	F0E68C	Хаки
lavender	E6E6FA	Бледно-лиловый
lavenderblush	FFF0F5	Бледный розово-лиловый
lawngreen	7CFC00	Зеленой травы
lemonchiffon	FFFACD	Лимонный
lightblue	ADD8E6	Светло-голубой
lightcoral	F08080	Светло-коралловый
lightcyan	E0FFFF	Светлый циан
lightgray	D3D3D3	Светло-серый
lightgreen	9CEE90	Светло-зеленый
lightpink	FFB6C1	Светло-розовый
lightsalmon	FFA07A	Светлый оранжево-розовый
lightseagreen	20B2AA	Светлый морской волны
lightskyblue	87CEFA	Светлый небесно-синий
lightslategray	778899	Светлый синевато-серый
lightsteelblue	B0C4DE	Светло-стальной
lightyellow	FFFFE0	Светло-желтый

Таблица П3.1 *(продолжение)*

Обозначение	RGB-код	Цвет
lime	00FF00	Известковый
limegreen	32CD32	Зеленовато-известковый
linen	FAF0E6	Льняной
magenta	FF00FF	Фуксин
maroon	800000	Оранжево-розовый
mediumaquamarine	66CDAA	Умеренно-аквамариновый
mediumblue	0000CD	Умеренно-голубой
mediumorchid	BA55D3	Умеренно-орхидейный
mediumpurple	9370DB	Умеренно-пурпурный
mediumseagreen	3CB371	Умеренный морской волны
mediumslateblue	7B68EE	Умеренный серовато-синий
mediumspringgreen	00FA9A	Умеренный синевато-серый
mediumturquoise	48D1CC	Умеренно-бирюзовый
mediumvioletred	C71585	Умеренный красно-фиолетовый
midnightblue	191970	Ночной синий
mintcream	F5FFFA	Мятно-кремовый
mistyrose	FFE4E1	Туманно-розовый
moccasin	FFE4B5	Болотный
navajowhite	FFDEAD	Грязно-серый
navy	000080	Темно-синий
oldlace	FDF5E6	Старого коньяка
olive	808000	Оливковый
olivedrab	6B8E23	Тускло-коричневый
orange	FFA500	Оранжевый
orangered	FF4500	Красно-оранжевый
orchid	DA70D6	Орхидейный
palegoldenrod	EEE8AA	Бледно-золотой
palegreen	98FB98	Бледно-зеленый

Таблица П3.1 *(продолжение)*

Обозначение	RGB-код	Цвет
paleturquoise	AFEEEE	Бледно-бирюзовый
plaevioletred	DB7093	Бледный красно-фиолетовый
papayawhip	FFEFD5	Дыни
peachpuff	FFDAB9	Персиковый
peru	CD853F	Коричневый
pink	FFC0CB	Розовый
plum	DDA0DD	Сливовый
powderblue	B0E0E6	Туманно-голубой
purple	800080	Пурпурный
rosybrown	BC8F8F	Розово-коричневый
royalblue	4169E1	Королевский голубой
saddlebrown	8B4513	Старой кожи
salmon	FA8072	Оранжево-розовый
sandybrown	F4A460	Рыже-коричневый
seagreen	2E8B57	Морской волны
seashell	FFF5EE	Морской пены
sienna	A0522D	Охра
silver	C0C0C0	Серебристый
skyblue	87CEEB	Небесно-голубой
slateblue	6A5ACD	Серовато-синий
slategray	708090	Синевато-серый
snow	FFFAFA	Снежный
springgreen	00FF7F	Весенний зеленый
steelblue	4682B4	Голубовато-стальной
tan	D2B48C	Желтовато-коричневый
teal	008080	Чайный
thistle	D8BFD8	Чертополоха
tomato	FF6347	Томатный
turquoise	40E0D0	Бирюзовый

Таблица П3.1 *(окончание)*

Обозначение	RGB-код	Цвет
violet	EE82EE	Фиолетовый
wheat	F5DEB3	Пшеничный
white	FFFFFF	Белый
whitesmoke	F5F5F5	Белый дымчатый
yellow	FFFF00	Желтый
yellowgreen	9ACD32	Желто-зеленый

Приложение 4

Свободно распространяемые библиотеки для JavaScript-программистов

В этом приложении кратко описаны свободно распространяемые библиотеки, которые могут пригодиться JavaScript-программистам в работе. Все эти библиотеки распространяются бесплатно и не требуют никаких отчислений.

JsHttpRequest

Небольшая библиотека JsHttpRequest заметно облегчает создание приложений, реализующих технологию AJAX. Ее возможности перечислены далее.

- Работает во всех современных Web-обозревателях.
- Поддерживает все кодировки.
- Позволяет отправлять все данные, введенные в Web-форму.
- Предоставляет возможность отправки файлов из Web-формы.
- Предоставляет возможность отправки данных, сформированных самими сценариями, в том числе таких типов, как массивы и экземпляры объектов.
- Модульная структура (программист может включить в страницы только тот код, который он реально использует).
- Полная совместимость с PHP — популярной технологией создания серверных программ. Так, в состав JsHttpRequest входит модуль, предназначенный для использования в PHP-программах и реализующий взаимодействие с клиентской частью этой библиотеки.
- Возможность совместной работы с известной библиотекой Prototype (будет описана далее).

Версия, доступная на момент написания книги, — 5.

"Домашний" сайт — **http://en.dklab.ru/lib/JsHttpRequest/**.

Prototype

Библиотека Prototype сейчас исключительно популярна. Почему — станет ясно, если рассмотреть ее возможности, которые перечислены далее.

- Расширенные средства для работы со строками, числами, массивами, функциями, объектами и экземплярами объектов, то есть практически со всеми типами данных, поддерживаемыми JavaScript.
- Мощные средства для форматирования строк.
- Создание принципиально новых типов данных, не поддерживаемых JavaScript напрямую, но доступных в других языках программирования, например, перечислений.
- Упрощенный доступ к элементам страницы. Prototype предлагает особый язык для описания "пути" к элементу страницы, к которому нужно получить доступ.
- Удобные средства для изменения содержимого страницы, в том числе и для управления местоположением ее элементов.
- Простые средства для работы с событиями, в частности, получения информации о событии.
- Средства, расширяющие функциональность стандартного таймера JavaScript.
- Исключительно богатые возможности для работы с Web-формами и элементами управления. Можно получить доступ к нужному элементу управления, сделать его доступным или недоступным для ввода и даже преобразовать введенные в форму данные в вид, пригодный для отправки серверной программе, реализующей технологию AJAX.
- Простые средства для реализации технологии AJAX, в частности, отправки, приема и вывода данных.

Версия, доступная на момент написания книги, — 1.5.1 RC1.

"Домашний" сайт — **http://www.prototypejs.org/**.

DOJO

Библиотека DOJO будет полезна тем Web-программистам, которые разрабатывают Web-приложения. Создателям обычных Web-страниц она вряд ли пригодится.

Прежде всего, DOJO предоставляет программисту огромный набор элементов управления, заменяющих элементы управления, которые определены

в стандарте HTML. Это всевозможные поля ввода, кнопки, флажки, переключатели и списки, которые нам уже знакомы по *главе 12*. Также DOJO предоставляет свою собственную Web-форму, заменяющую стандартную. Все эти элементы имеют дополнительные возможности, позволяющие легко настраивать не только их поведение, но и внешний вид.

Кроме того, в предоставляемый данной библиотекой набор элементов управления входят и те, что не определены в стандарте HTML. Это:

- поле ввода с фильтром (в него можно ввести только строго определенные значения: числа, телефонные номера, почтовые индексы, значения даты и времени и пр., причем поле ввода не позволит посетителю ввести некорректные данные);
- поле ввода со счетчиком;
- кнопка с всплывающим меню;
- регулятор (движок);
- список для выбора цвета;
- иерархический список;
- таблица;
- полоса прогресса;
- всплывающая подсказка;
- панель с заголовком;
- меню;
- панель инструментов;
- специальные элементы управления для позиционирования других элементов относительно друг друга
- и мн. др.

Так же, как и многие современные библиотеки для JavaScript-программистов, DOJO позволяет реализовывать технологию AJAX.

Версия, доступная на момент написания книги, — 1.1.0.

"Домашний" сайт — **http://dojotoolkit.org/**.

Предметный указатель

(окончание рубрики см. на стр. 700)

Л

М

(продолжение рубрики см. на стр. 704)

(окончание рубрики см. на стр. 706)

П

P

C

(продолжение рубрики см. на стр. 710)

T

(окончание рубрики см. на стр. 714)