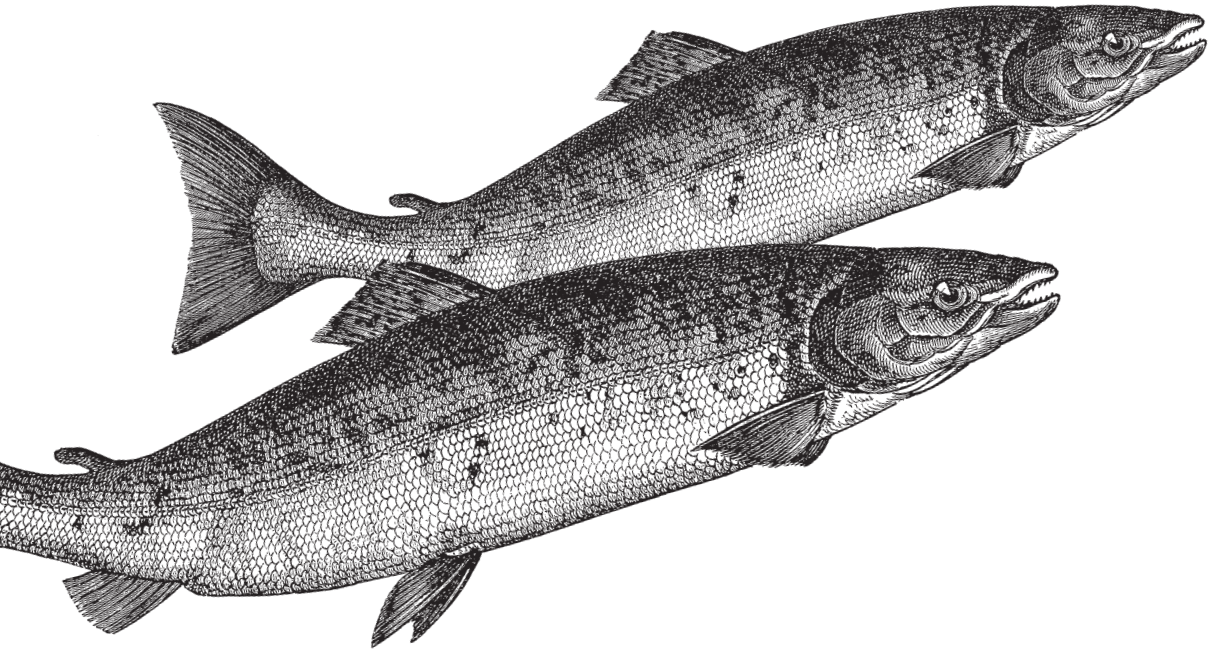
Визуальное представление веб-документов

3-е издание

CSS

каскадные таблицы стилей

Подробное руководство



Символ®

O'REILLY®

Эрик А. Мейер

По договору между издательством «Символ-Плюс» и Интернет-магазином «Books.Ru – Книги России» единственный легальный способ получения данного файла с книгой ISBN 5-93286-107-X, название «CSS – каскадные таблицы стилей. Подробное руководство» – покупка в Интернет-магазине «Books.Ru – Книги России». Если Вы получили данный файл каким-либо другим образом, Вы нарушили международное законодательство и законодательство Российской Федерации об охране авторского права.

Вам необходимо удалить данный файл, а также сообщить издательству «Символ-Плюс» (piracy@symbol.ru), где именно Вы получили данный файл.

Cascading Style Sheets

The Definitive Guide

Third Edition

Eric A. Meyer

O'REILLY®

CSS

каскадные таблицы стилей

Подробное руководство

Третье издание

Эрик Мейер

Символ®

Санкт-Петербург — Москва
2008

Эрик Мейер

CSS – каскадные таблицы стилей. Подробное руководство, 3-е издание

Перевод Н. Шатохиной

Главный редактор	*А. Галунов*
Зав. редакцией	*Н. Макарова*
Научный редактор	*Б. Попов*
Редактор	*Ю. Бочина*
Корректор	*Л. Минина*
Верстка	*Д. Орлова*

Мейер Э.
CSS – каскадные таблицы стилей. Подробное руководство, 3-е издание. – Пер. с англ. – СПб: Символ-Плюс, 2008. – 576 с., ил.

ISBN-13: 978-5-93286-107-3
ISBN-10: 5-93286-107-X

Третье издание «CSS – каскадные таблицы стилей. Подробное руководство» показывает, как реализовать на практике все возможности каскадных таблиц стилей для стандартов CSS2 и CSS2.1. Множество примеров позволит научиться быстро и без усилий разрабатывать стилевое оформление веб-страниц, отвечающее современным требованиям.

Эрик Мейер, признанный эксперт по CSS, HTML и веб-стандартам, опираясь на свой богатейший опыт, рассматривает все свойства CSS и их взаимодействие, теги, атрибуты, реализации, поддержку различными броузерами, дает рекомендации разработчикам. Вы узнаете о сложном стилевом оформлении документов, пользовательском интерфейсе, верстке таблиц, о списках и генерируемом содержимом, о свободном перемещении и позиционировании, о семействах шрифтов и механизмах резервирования, о том, как работает модель блоков, о новых селекторах CSS3, поддерживаемых IE7, Firefox и другими броузерами. Книга поможет избежать распространенных ошибок, она является полным справочником по CSS и будет полезна как опытному веб-разработчику, так и новичку. От читателя потребуется только знание HTML 4.0.

ISBN-13: 978-5-93286-107-3
ISBN-10: 5-93286-107-X
ISBN 0-596-52733-0 (англ)

© Издательство Символ-Плюс, 2008
Authorized translation of the English edition © 2006 O'Reilly Media, Inc. This translation is published and sold by permission of O'Reilly Media, Inc., the owner of all rights to publish and sell the same.

Все права на данное издание защищены Законодательством РФ, включая право на полное или частичное воспроизведение в любой форме. Все товарные знаки или зарегистрированные товарные знаки, упоминаемые в настоящем издании, являются собственностью соответствующих фирм.

Издательство «Символ-Плюс». 199034, Санкт-Петербург, 16 линия, 7,
тел. (812) 324-5353, www.symbol.ru. Лицензия ЛП N 000054 от 25.12.98.
Налоговая льгота – общероссийский классификатор продукции
ОК 005-93, том 2; 953000 – книги и брошюры.

Подписано в печать 29.02.2008. Формат 70×100¹/16. Печать офсетная.
Объем 36 печ. л. Тираж 2000 экз. Заказ №
Отпечатано с готовых диапозитивов в ГУП «Типография «Наука»
199034, Санкт-Петербург, 9 линия, 12.

Посвящаю жене и дочери за все то счастье,
которое они дарят мне.

Оглавление

Введение

Если вы веб-дизайнер, разработчик веб-страниц и вас интересуют сложные стили оформления страниц, улучшение их восприятия и экономия времени и усилий, эта книга – для вас. Все, что надо для начала, – это неплохое знание HTML 4.0. Чем лучше вы знаете HTML, тем лучше вы подготовлены к чтению книги. Что касается остальных знаний и умений, для работы с этой книгой достаточно базового уровня.

Третье издание книги «CSS – каскадные таблицы стилей. Подробное руководство» посвящено стандартам CSS2 и CSS2.1 (вплоть до рабочего проекта, вышедшего 11 апреля 2006 года), последний из которых во многом представляет собой дополненную версию первого. Несмотря на то что некоторые части CSS3 получили статус предварительной рекомендации, в этом издании я решил их не рассматривать (за исключением некоторых селекторов CSS3), потому что реализация соответствующих модулей до сих пор не завершена или ее попросту нет. Я считаю, что сейчас важно сосредоточиться на поддерживаемых в настоящий момент и хорошо понятных уровнях CSS, а все грядущие возможности лучше оставить для последующих изданий.

Принятые обозначения

В этой книге действуют следующие соглашения о шрифтовом оформлении:

Курсив

Применяется для выделения новых терминов, URL, переменных в тексте, имен файлов и каталогов, команд, расширений файлов и путей доступа UNC.

`Моноширинный шрифт`

Предназначен для данных, выводимых в окне командной строки, примеров кода, ключей реестра.

`Моноширинный жирный`

Обозначает в примерах данные, вводимые пользователем.

`Моноширинный курсив`

Показывает переменные в примерах и ключах реестра. Также используется для выделения переменных или определяемых пользо-

вателем элементов в тексте, написанном курсивом (например, в пути или имени файла). Например, в пути *\Windows\имя_пользователя* замените текст имя_пользователя на реальное имя зарегистрированного в системе пользователя.

Так отмечаются подсказки, советы и примечания.

А это предупреждение и предостережение.

Соглашения по представлению свойств

В этой книге встречаются блоки разбора рассматриваемых свойств CSS. Они практически дословно воспроизведены из спецификаций CSS, поэтому необходимы некоторые разъяснения их синтаксиса.

По всей книге допустимые значения каждого свойства приводятся в соответствии со следующим синтаксисом:

Значение: **[** <длина> | thick | thin **]{1,4}**

Значение: **[**<имя_семейства> **,]*** <имя_семейства>

Значение: <url>**?** <цвет> **[** / <цвет> **]?**

Значение: <url> || <цвет>

Любые слова между символами «<» и «>» определяют тип значения или ссылку на другое свойство. Например, свойство font может принимать значения, которые на самом деле относятся к свойству font-family. На это указывает выражение <font-family>. Любые слова, представленные моноширинным шрифтом, представляют собой ключевые слова, которые должны применяться буквально, без кавычек. Прямая наклонная черта (/) и запятая (,) также должны использоваться буквально.

Выстраивание нескольких ключевых слов в некоторой последовательности также означает, что все они должны выполняться в данном порядке. Например, help me означает, что в свойстве эти ключевые слова должны быть расположены именно в таком порядке.

Если варианты разделены вертикальной чертой (X | Y), то следует выбрать какой-то один из них. Двойная вертикальная черта (X || Y) означает, что можно выбрать X, Y или оба элемента, и появляться они могут в любом порядке. Скобки [...] предназначены для группировки. Размещение рядом имеет больший приоритет, чем двойная вертикальная черта, которая в свою очередь имеет больший приоритет, чем одиночная вертикальная черта. Таким образом, запись «V W | X || Y Z» эквивалентна «[V W] | [X || [Y Z]]».

За каждым словом или заключенной в скобки группой может располагаться один из следующих модификаторов:

- Звездочка (*) указывает на то, что предшествующее ей значение или заключенная в скобки группа повторяются нуль или более раз. То есть запись `bucket*` означает, что слово `bucket` может повторяться любое количество раз, а может вообще отсутствовать. Верхней границы для возможного количества его применений нет.
- Плюс (+) указывает на то, что предшествующее значение или заключенная в скобки группа повторяются один или более раз, то есть запись `mop+` означает, что слово `mop` должно быть использовано хотя бы единожды или, возможно, несколько раз.
- Знак вопроса (?) указывает на то, что предшествующее значение или заключенная в скобки группа являются необязательными. Например, `[pine tree]?` означает, что слова `pine tree` могут отсутствовать (хотя в случае употребления они должны появляться строго в указанном порядке).
- Пара чисел в фигурных скобках {M,N} указывает на то, что предшествующее значение или заключенная в скобки группа повторяются не менее `M` и не более `N` раз. Например, `ha{1,3}` означает, что слово `ha` можно повторить один, два или три раза.

Вот некоторые примеры:

`give || me || liberty`

Должно присутствовать хотя бы одно из этих трех слов, и они могут располагаться в любом порядке. Например, `give liberty`, `give me`, `liberty me give` и `give me liberty` – все это действительные варианты.

`[ I | am ]? the || walrus`

Может присутствовать любое из слов `I` и `am`, но не оба сразу. Кроме того, можно вообще обойтись без этих слов. Также должны присутствовать слова `the` или `walrus` или оба в произвольном порядке. Таким образом, вы могли бы составить фразы `I the walrus`, `am walrus the`, `am the`, `I walrus`, `walrus the` и т. д.

`koo+ ka-choo`

Один или более экземпляров слова `koo` должны быть продолжены словом `ka-choo`. Следовательно, выражения `koo koo ka-choo`, `koo koo koo ka-choo` и `koo ka-choo` являются допустимыми. Несмотря на существующие ограничения, определяемые конкретной реализацией, число экземпляров `koo` стандартом никак не ограничено.

`I really{1,4}? [love | hate] [Microsoft | Netscape | Opera | Safari]`

Это универсальное средство выражения мнений веб-разработчика. Этот пример можно интерпретировать как `I love Netscape`, `I really love Microsoft` и аналогичные выражения. Слово `really` может отсутствовать или применяться от одного до четырех раз. Также можно

выбирать между `love` и `hate`, хотя здесь показан только пример со словом `love`.

`[[Alpha || Baker || Cray], ]{2,3} and Delphi`

Это более длинное и сложное выражение. Одним из возможных результатов может быть `Alpha, Cray, and Delphi`. Запятая здесь вставляется потому, что она указана в ограниченной скобками вложенной группе.

Использование примеров кода

Данная книга призвана помочь вам в вашей работе. В общем, вы можете использовать код из этой книги в своих программах и документации. Не надо обращаться в O'Reilly за разрешением на использование небольших частей кода, например при написании программы, в которой используется несколько блоков кода из этой книги. А вот продажа или распространение CD-ROM с примерами из книг O'Reilly требует специального разрешения. Вы можете свободно ссылаться на книгу и цитировать примеры кода, но для включения больших частей кода из этой книги в документацию вашего продукта требуется наше согласие.

Будем признательны, но не настаиваем на указании авторства. Обычно ссылка на источник включает название, автора, издателя и ISBN. Например: «CSS: Подробное руководство, Эрик А. Мейер. Copyright 2007 O'Reilly Media, Inc., 978-0-596-52733-4».

Если вам кажется, что использование вами примеров кода выходит за рамки законного использования или разрешений, оговоренных выше, не стесняйтесь, обращайтесь к нам по адресу *permissions@oreilly.com*.

Контактная информация

Сотрудники издательства O'Reilly тщательно проверили корректность информации, приведенной в данной книге, но не исключено, что некоторые возможности изменились (или даже остались ошибки!). Пожалуйста, сообщайте нам о любых найденных неточностях, а также присылайте ваши предложения для будущих изданий по адресу:

O'Reilly Media, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472
800-998-9938 (в США или Канаде)
707-829-0515 (международный или местный)
707-829-0104 (факс)

Для этой книги создана веб-страница, на которой представлены список опечаток, примеры и другая дополнительная информация. Страница расположена по адресу:

http://www.oreilly.com/catalog/csstdg3

Вопросы и комментарии по этой книге присылайте по электронной почте:

bookquestions@oreilly.com

Для получения более подробной информации о книгах, конференциях, ресурсах и сети O'Reilly посетите веб-сайт издательства:

http://www.oreilly.com

Safari® Enabled

Если на обложке книги есть пиктограмма «Safari® Enabled», это означает, что книга доступна в Сети через O'Reilly Network Safari Bookshelf.

Safari предлагает намного лучшее решение, чем электронные книги. Это виртуальная библиотека, позволяющая без труда находить тысячи лучших технических книг, копировать и использовать примеры кода, скачивать главы и быстро находить ответы, когда требуется самая точная и свежая информация. Она свободно доступна по адресу *http://safari. oreilly.com*.

Благодарности

Я бы хотел воспользоваться случаем и выразить благодарность всем, кто поддерживал меня во время долгого путешествия этой книги до полок магазинов.

Прежде всего хотелось бы поблагодарить всех сотрудников O'Reilly за все, что они делали в течение этих лет, периодически предоставляя мне возможность отдыхать, чтобы я мог написать достойную книгу. В этом третьем издании я выражаю благодарность Татьяне Апанди (Tatiana Apandi) за ее хорошее чувство юмора, терпение и понимание в случаях, когда я запаздывал со сроками.

Я также хотел бы выразить глубокую признательность моим техническим рецензентам. В первом издании это были Дэвид Бэрон (David Baron) и Ян Хиксон (Ian Hickson) с участием Берта Боса (Bert Bos) и Хекон Ли (Hekon Lie). Вторым изданием занимались Тантек Келик (Tantek Celik) и Ян Хиксон (Ian Hickson). Третье издание, которое вы держите в руках, редактировали замечательные люди – Даррелл Остин (Darrell Austin), Лайза Дейли (Liza Daly) и Нейл Ли (Neil Lee). Все они вложили немалый опыт и понимание, благодаря чему я смог учесть самые последние изменения, внесенные в CSS, а также избежать небрежных описаний и туманных пояснений. Ни одно из изданий этой книги, тем более данное, не получилось бы таким качественным, если бы не их коллективный вклад, но любые ошибки, которые вы найдете в тексте, – это, безусловно, моя вина, а не их. Это избитая фраза, но так оно и есть.

Также я хотел бы сказать спасибо всем, кто нашел опечатки и сообщил о них. Возможно, я не всегда оперативно отвечал на ваши письма, но читал все вопросы и замечания и в случае необходимости вносил коррективы. Продолжайте присылать свои отзывы, обратная связь и конструктивная критика лишь помогут сделать книгу лучше, как это было всегда.

Я также хочу выразить несколько личных благодарностей.

Коллективу WRUW, 91.1 FM Cleveland, спасибо вам за девять лет поддержки, замечательную музыку и бесшабашный юмор. Может быть, когда-нибудь я верну в ваш эфир звуки биг-бенда, а может, и нет, но в любом случае оставайтесь с нами.

Джеффри Зельдману (Jeffrey Zeldman) спасибо за то, что он замечательный коллега и партнер. И всему семейству Зельдманов спасибо за дружбу.

«Тетушке» Молли спасибо за то, что она всегда остается самой собой.

«Дядю» Джима благодарю за все – и в профессиональном, и в личном плане. Не будет преувеличением сказать, что я не достиг бы того, чего достиг, без вашего влияния, да и жизнь была бы намного более пресной, не будь вас рядом.

Всей команде кулинаров – Джиму, Женевьев, Джиму, Джини, Ферретт, Джен, Дженн и Молли – спасибо за великолепную стряпню и приятную беседу.

Всем, кого я должен был бы поблагодарить, но не сделал этого, мои извинения. И благодарности.

Моим жене и дочери безграничная благодарность за то, что они делают мои дни богаче, чем я заслуживаю, и за то, что они окружают меня такой любовью, которую я даже не надеюсь им вернуть. Хотя, конечно, я буду стараться.

– Эрик А. Мейер (Eric A. Meyer)
г. Кливленд-Хайтс, Огайо
1 августа 2006

1

CSS и документы

Каскадные таблицы стилей (CSS – Cascading Style Sheets) – мощный механизм управления представлением отдельных документов или их наборов. Очевидно, собственно каскадные таблицы стилей при отсутствии документа бесполезны, поскольку в них нет содержимого, которое надо представлять. Конечно, термин «документ» понимается здесь крайне широко. Например, Mozilla и родственные броузеры используют CSS, чтобы воздействовать на представление деталей интерфейса самого броузера. Но и в этом случае без «декораций» – кнопок, полей ввода адреса, диалоговых и обычных окон и т. д. – нет необходимости в CSS (или любой другой информации о представлении).

Веб спускается с Олимпа

В смутно припоминаемые (1990–1993) ранние годы Всемирной паутины HTML был довольно бедным языком. Он почти целиком состоял из структурных элементов, полезных для описания абзацев, гиперссылок, списков и заголовков. В нем не было ничего, даже отдаленно напоминавшего таблицы, фреймы или сложную разметку, – того, что считается абсолютно необходимым для создания веб-страниц. HTML изначально задумывался как структурный язык разметки, применяемый для описания различных частей документа. О том, как должны отображаться эти части, говорилось совсем немного. Язык не затрагивал описание внешнего вида – он был лишь небольшой схемой разметки.

Затем пришел Mosaic.

Мощь Всемирной паутины вдруг стала очевидной практически каждому, кто проводил в ней более 10 минут. Для перехода от одного документа к другому было достаточно указать курсором на выделенный специальным цветом фрагмент текста или даже изображение и щелкнуть кнопкой мыши. Кроме того, текст и графику можно было отобра-

жать на экране вместе, а для создания страницы нужен был только простой текстовый редактор. Все это находилось в свободном доступе, было открыто, и это здорово.

Веб-сайты начали возникать повсеместно. Это были личные дневники, сайты университетов, корпоративные сайты и т. д. Поскольку росло количество сайтов, росла и потребность в новых HTML-элементах, которые должны были выполнять определенные функции. Авторам потребовались средства, которые позволяли выделять фрагменты текста полужирным шрифтом или курсивом.

На тот момент HTML не имел возможности реализовать эти пожелания. Вы могли указать необходимость выделения фрагмента текста, но это не обязательно означало, что текст выделялся курсивом; вместо этого он мог быть выделен полужирным шрифтом или даже обычным шрифтом другого цвета в зависимости от броузера и его установок. Не было средств, которые бы гарантировали, что пользователь увидит то, что создал автор.

В результате этого давления в язык начали просачиваться такие элементы разметки, как теги `<FONT>` и `<BIG>`. Так структурный язык начал превращаться в язык представления.

Полная неразбериха

Прошли годы, и мы унаследовали все извивы этого беспорядочного процесса. Значительная часть средств HTML 3.2 и HTML 4.0 оказалась посвящена вопросам представления. Возможность изменять цвет и размер текста с помощью элемента `font`, назначать фоновые цвета и изображения для документов и таблиц, задавать атрибуты тега `table` (такие как `cellspacing`) и делать текст мигающим – все это наследие произнесенных в то время просьб «побольше управления!».

Чтобы увидеть результаты возникшей неразберихи, взгляните на разметку практически любого корпоративного веб-сайта. Поразительно, насколько объем разметки превышает объем полезной информации. Что еще хуже, разметка большинства сайтов практически полностью состоит из таблиц и элементов `font`, ни один из которых не помогает понять, что именно он представляет. С точки зрения структурированности эти страницы ненамного лучше, чем случайный набор строк или символов.

Давайте, например, предположим, что для заголовков страницы автор вместо тегов заголовка, таких как `h1`, применяет элементы `font`:

```
<font size="+3" face="Helvetica" color="red">Заголовок страницы</font>
```

Тег `font` никак не влияет на структуру документа. Это делает документ гораздо менее полезным. Какой, например, толк в теге `font` для броузера с возможностью синтеза речи? Если автор применяет заголовки, а не элементы `font`, броузер при воспроизведении текста может изменить

громкость или интонацию. Тег `font` не дает броузеру возможности узнать, что данный фрагмент чем-то отличается от остального текста.

Почему же авторы применяют для управления структурой и содержанием такие жесткие методы? Потому что они хотят, чтобы читатели видели страницу такой, какой она была создана. Использовать структурированную разметку HTML для них означает передать броузеру контроль за внешним видом страницы, что кажется недопустимым для плотно укомплектованных страниц, которые обрели популярность за эти годы. Но давайте взглянем на проблемы, возникающие при таком подходе:

- Неструктурированные страницы значительно усложняют индексирование содержимого. Мощный поисковый механизм позволяет вести поиск только по заголовкам страниц, или только по заголовкам разделов страницы, или только по тексту абзацев, или, возможно, только по специально отмеченным абзацам. Однако для реализации таких возможностей содержимое страниц должно соответствовать некому стандарту структурированной разметки. Именно этого не хватает большинству страниц. Обратите внимание, что Google при индексации страниц учитывает структуру разметки, поэтому структурированная страница повысит ваш ранг в Google.

- Недостаточная структурированность снижает удобство восприятия. Представьте, что вы слепы и при поиске полагаетесь на броузер с модулем синтеза речи. Что вы предпочтете: структурированную страницу, которая позволит вашему броузеру читать только заголовки разделов, и вы сможете выбрать раздел, о котором хотите услышать подробнее, или страницу, в которой структура отсутствует настолько, что ваш броузер вынужден читать весь текст подряд, не понимая, является ли он заголовком, текстом абзаца или помечен как важный? Давайте снова вспомним о Google – поисковой системе, которая сегодня является наиболее активным скрытым пользователем, имеющим миллионы друзей, которые принимают каждое из его предложений типа «где искать и что купить».

- Улучшенное представление страницы возможно только при наличии некоторой структуры документа. Представьте себе страницу, на которой показаны только заголовки разделов со стрелкой рядом с каждым из них. Пользователь сам выбирает, какой заголовок его интересует, и, щелкнув по нему, получает доступ к тексту раздела.

- Структурированная разметка упрощает обслуживание сайта. Сколько раз вы охотились за незначительными ошибками в чужом (или даже в собственном) HTML, которые портили вид страницы в том или ином броузере? Сколько времени вы потратили, создавая вложенные таблицы или элементы `font`, чтобы просто получить врезку с белыми ссылками? Сколько переходов строки вы вставили, пытаясь добиться необходимого расстояния между заголовком и текстом?

Применение структурированной разметки позволяет очистить код и упростить поиск в нем.

Тем не менее приходится признать, что полностью структурированный документ несколько некрасив. Из-за одного этого факта и сотни аргументов в пользу структурированной разметки не хватит, чтобы отговорить отдел маркетинга от использования того типа HTML, который был распространен в конце XX столетия и существует до сих пор. Поэтому нужен метод, позволяющий сочетать структурированную разметку с привлекательным представлением страницы.

CSS спешит на помощь

Конечно, проблема загрязнения HTML разметкой представления не осталась незамеченной консорциумом W3C (World Wide Web Consortium), который приступил к поиску быстрого решения. В 1995 году консорциум начал публикацию рабочего варианта стандарта, названного CSS. К 1996 году он получил статус рекомендации, такой же значимой, как и сам HTML. Вот причины этого.

Богатство стилей

Прежде всего, CSS обеспечивают более богатое представление документа, чем когда-либо, даже на пике своего репрезентативного пыла, позволял HTML. CSS позволяют задавать цвета текста и фона любых элементов, создавать рамки и увеличивать или уменьшать отступы вокруг элементов. Благодаря им можно сделать так, чтобы текст отображался прописными буквами, и добавить дополнительные элементы оформления (например, подчеркивание), разбивки и даже управлять тем, будет ли он отображаться вообще, а также они дают возможность реализовать многие другие эффекты.

Возьмем для примера первый (и основной) заголовок на странице, который обычно является заголовком самой страницы. Пример правильной разметки:

```
<h1>Паря над водой</h1>
```

Теперь предположим, что вы хотите, чтобы этот заголовок стал темно-красным, был набран определенным шрифтом курсивного начертания, подчеркнут и располагался на желтом фоне. Чтобы сделать все это с помощью HTML, понадобилось бы поместить тег `h1` в таблицу и нагрузить его массой других элементов, таких как `font` и `U`. CSS позволяет обойтись одним правилом:

```
h1 {color: maroon; font: italic 2em Times, serif; text-decoration: underline;
      background: yellow;}
```

Вот и все. Как видите, все, что вы делали в HTML, можно сделать в CSS. Однако не надо ограничивать себя только тем, что умеет HTML:

```
h1 {color: maroon; font: italic 2em Times, serif; text-decoration: underline;
      background: yellow url(titlebg.png) repeat-x;
      border: 1px solid red; margin-bottom: 0; padding: 5px;}
```

Теперь в качестве фона элемента h1 выступает изображение, которое повторяется в горизонтальном направлении, а вокруг элемента добавлена рамка, отстоящая от текста минимум на пять пикселов. Вы также удалили поле (пустое пространство) в нижней части элемента. Это те возможности, к которым HTML даже и близко не мог подобраться, и это всего лишь малая часть того, что могут CSS.

Простота применения

Если сила CSS не убедила вас, примите во внимание следующее: таблицы стилей могут существенно сократить объем работы разработчика веб-страниц.

Во-первых, таблицы стилей концентрируют команды, реализующие визуальные эффекты, в одном легкодоступном месте, а не разбрасывают их по всему документу. В качестве примера предположим, что в документе все заголовки h2 должны быть фиолетовыми. Работая с HTML, можно поместить тег font в каждый заголовок:

```
<h2><font color="purple">Это фиолетовый!</font></h2>
```

Это должно быть сделано для каждого заголовка второго уровня. Если в документе 40 заголовков, то надо вставить 40 элементов font, по одному для каждого заголовка! Слишком много труда требует один небольшой эффект.

Предположим, что вы вставили все эти элементы font. Вы закончили, вы счастливы, а затем решаете (или начальник решает за вас), что заголовки h2 должны быть темно-зелеными, а не фиолетовыми. Надо возвращаться назад и вновь заменять каждый из элементов font. Конечно, можно обратиться к командам поиска и замены, поскольку заголовки являются единственным текстом фиолетового цвета в вашем документе. Но если вы ввели и другие элементы font фиолетового цвета, вы *не можете* применить автоматический поиск и замену, потому что тогда будут изменены и эти элементы.

Вместо этого намного лучше было бы иметь одно правило:

```
h2 {color: purple;}
```

Его не только намного быстрее набрать на клавиатуре, но и намного проще изменять. Для перехода от фиолетового цвета к темно-зеленому достаточно изменить одно-единственное правило.

Вернемся к нагруженному стилями элементу h1 из предыдущего раздела:

```
h1 {color: maroon; font: italic 2em Times, serif; text-decoration: underline;
      background: yellow;}
```

Может показаться, что написать это правило намного сложнее, чем HTML, но представьте, что на странице расположена дюжина элементов h2, которые должны выглядеть так же, как h1. Сколько разметки потребуется для этих 12 элементов h2? Много. А применяя CSS, достаточно сделать следующее:

```
h1, h2 {color: maroon; font: italic 2em Times, serif;
          text-decoration: underline; background: yellow;}
```

Теперь стили применены и к элементам h1, и к элементам h2, а понадобилось для этого всего лишь три дополнительных нажатия на клавиши.

Если вы хотите изменить вид элементов h1 и h2, преимущества CSS даже еще более ощутимы. Представьте, сколько времени понадобилось бы, чтобы изменить разметку HTML для элементов h1 и h2, по сравнению с внесением следующих изменений в предыдущие стили:

```
h1, h2 {color: navy; font: bold 2em Helvetica, sans-serif;
          text-decoration: underline overline; background: silver;}
```

Если бы продолжительность каждого из этих двух подходов фиксировалась с помощью секундомера, держу пари, что разбирающийся в CSS автор без труда обогнал бы приверженца HTML.

Кроме того, большинство правил CSS сосредоточиваются в документе в одном месте. Можно распределить их по всему документу, группируя в ассоциированные стили или отдельные элементы, но как правило, намного эффективнее поместить все стили в одну таблицу стилей. Это позволяет разрабатывать (или изменять) внешний вид всего документа в одном месте.

Применение стилей к нескольким страницам

Но есть и еще кое-что! Можно не только централизовать всю информацию о стилях страницы в одном месте, но и создать таблицу стилей, которая может применяться ко многим страницам. Это реализуется путем сохранения таблицы стилей в отдельном документе, который затем импортируется любой использующей его страницей документа. Эта возможность позволит быстро создавать единообразный внешний вид всего веб-сайта. Для этого достаточно привязать одну таблицу стилей ко всем документам веб-сайта. Затем, если вам когда-нибудь захочется изменить внешний вид страниц сайта, надо будет лишь отредактировать один файл, и внесенные изменения распространятся на весь сервер автоматически!

Рассмотрим сайт, в котором все заголовки выделены серым шрифтом на белом фоне. Этот цвет они получают из таблицы стилей, которая гласит:

```
h1, h2, h3, h4, h5, h6 {color: gray; background: white;}
```

Предположим, этот сайт состоит из 700 страниц, каждая из которых использует таблицу стилей, оговаривающую, что заголовки должны

быть серыми. В некоторый момент разработчик веб-сайта принимает решение, что заголовки должны быть белыми на сером фоне. Итак, он редактирует таблицу стилей следующим образом:

```
h1, h2, h3, h4, h5, h6 {color: white; background: gray;}
```

затем таблица стилей сохраняется на диск и изменение выполнено. Это, конечно же, лучше, чем редактировать 700 страниц, чтобы просмотреть и изменить каждый заголовок таблицы и тег `font`, не так ли?

Каскадирование

И это еще не все! CSS также поддерживает средства разрешения конфликтов правил, называемые *каскадным включением* (*cascade*). Возьмем для примера предыдущий сценарий, в котором одна таблица стилей импортировалась в несколько веб-страниц. Теперь добавим набор страниц, которые ряд стилей используют совместно, но в то же время включают специализированные правила, применяемые только к конкретным страницам. В дополнение к уже существующей таблице стилей можно создать еще одну таблицу, импортируемую в эти страницы, или просто поместить специальные стили в страницы, которые в них нуждаются.

Например, требуется, чтобы на одной из 700 страниц заголовки были выделены желтым цветом на темно-синем фоне вместо желтого на сером. Тогда в этот отдельный документ можно ввести такое правило:

```
h1, h2, h3, h4, h5, h6 {color: yellow; background: blue;}
```

Благодаря каскадному включению это правило переопределит импортированное правило, реализующее желтые заголовки на сером фоне. Понимая и разумно применяя правила каскадирования, можно создавать сложные таблицы стилей, без труда изменяемые и объединяемые для обеспечения профессионального представления страниц.

От мощи каскадного объединения выигрывает не только автор. Вебсерферы (или *читатели*) могут в некоторых броузерах создавать собственные таблицы стилей (названные *таблицами стилей читателя* (*reader style sheets*)), которые будут каскадироваться со стилями автора, а также со стилями, используемыми броузером. Благодаря этому читатель-дальтоник может создать стиль, который выделяет гиперссылки:

```
a:link, a:visited {color: white; background: black;}
```

Таблица стилей читателя может включать все, что угодно: директиву увеличения размера текста для пользователя с ослабленным зрением, правила для удаления изображений, чтобы увеличить скорость чтения и просмотра, и даже стили для размещения любимой картинки пользователя в качестве фона каждого документа. (Это, конечно, не рекомендуется, но возможно.) Благодаря этому пользователи могут настраивать представление веб-документа, не отключая все стили автора.

CSS – замечательный инструмент для любого автора или читателя, поскольку обладает возможностями импортирования, каскадирования и реализации разнообразных эффектов.

Небольшой размер файла

Кроме визуальной мощи CSS и его способности расширять возможности как автора, так и читателя, в нем есть еще кое-что, что понравится вашим читателям. CSS способствует уменьшению размеров документов, сокращая таким образом время загрузки. Как? Я уже говорил, что для реализации визуальных эффектов многие страницы использовали таблицы и элементы `font`. К сожалению, оба эти метода создают дополнительную HTML-разметку, увеличивающую размеры файлов. Группируя информацию о визуальных стилях в одном месте и представляя эти правила посредством компактного синтаксиса, можно удалить элементы `font` и другие фрагменты обычной мешанины тегов. Таким образом, CSS уменьшит время загрузки к вящему удовольствию читателей.

Подготовка к будущему

Как я уже упоминал, HTML представляет собою структурный язык, тогда как CSS – это дополнение, стилистический язык. Осознавая это, W3C – орган, обсуждающий и утверждающий стандарты для Всемирной паутины, – начинает удалять стилистические элементы из HTML. Для создания эффектов, сейчас обеспечиваемых определенными HTML-элементами, могут применяться таблицы стилей. Так зачем тогда нужны эти HTML-элементы?

Таким образом, спецификация XHTML содержит ряд элементов, применять которые не рекомендуется (*deprecated*), поскольку со временем они будут полностью удалены из языка. В конце концов, они будут отмечены как устаревшие (*obsolete*), то есть от броузеров не будет требоваться и не будет поощряться их поддержка. К нерекомендуемым к применению элементам относятся: `<font>`, `<basefont>`, `<u>`, `<strike>`, `<s>` и `<center>`. С появлением таблиц стилей необходимость во всех этих элементах отпала. А со временем, возможно, количество нерекомендуемых элементов увеличится.

Мало того, существует вероятность того, что постепенно HTML будет вытеснен *расширяемым языком разметки* (eXtensible Markup Language – XML). XML намного сложнее HTML, но при этом он обладает значительно большими возможностями и гибкостью. Несмотря на это XML не предоставляет способа объявить стилевые элементы, такие как `<i>` или `<center>`. Вместо них, скорее всего, для определения внешнего вида XML-документов будут применяться таблицы стилей. Хотя таблицы стилей, применяемые с XML, могут и не быть таблицами CSS, по всей вероятности, они будут производными от CSS или чем-то очень близким к ним. Поэтому изучение CSS сейчас обеспечит авторам

большое преимущество, когда придет время сделать переход к Всемирной паутине на базе XML.

Итак, для начала очень важно понимать, как связаны друг с другом CSS и структура документа. CSS позволяют полностью изменить облик документа, но их возможности не безграничны. Начнем с изучения базовой терминологии.

Элементы

Элементы (*elements*) – это основа структуры документа. Нетрудно понять, какие элементы более всего используются в HTML: `p`, `table`, `span`, `a` и `div`. Каждый элемент документа играет определенную роль в его представлении. В терминах CSS (по крайней мере, для CSS2.1) это означает, что каждый элемент генерирует блок, в котором находится содержимое элемента.

Замещаемые и незамещаемые элементы

CSS определяется элементами, но не все элементы создаются одинаково. Например, изображения и абзацы – это элементы разных типов, так же как `span` и `div`. В CSS элементы разделяются на замещаемые и незамещаемые. Эти два типа элементов подробно обсуждаются в главе 7, где структуры блоков рассматриваются детально, а здесь я лишь вскользь коснусь их.

Замещаемые элементы

Замещаемыми (*replaced*) называются те элементы, содержимое которых замещается чем-то, что не содержится непосредственно в документе. Наиболее очевидный пример из XHTML – элемент `img`, замещаемый файлом изображения, который является внешним по отношению к документу. Кстати, `img` фактически не имеет содержимого, как видно из простого примера:

```
<img src="howdy.gif" />
```

Этот фрагмент разметки не имеет реального содержимого, а только имя элемента и атрибут. Данный элемент ничего не представляет, пока вы не укажете на внешнее содержимое (в данном случае изображение, заданное атрибутом `src`). Элемент `input` замещается кнопкой, переключателем или полем ввода текста в зависимости от его типа. Замещаемые элементы также генерируют блоки в своем визуальном представлении.

Незамещаемые элементы

Основная масса элементов HTML и XHTML – *незамещаемые* (*nonreplaceable*). Это означает, что их отображаемое агентом пользователя (обычно броузером) содержимое находится внутри генерируемого эле-

ментом блока. Например, элемент `<span>эй там</span>` – незамещаемый элемент, и агент пользователя (user agent) будет отображать текст «эй там». Это выполняется для абзацев, заголовков, ячеек таблиц, списков и почти всех остальных элементов XHTML.

Роль элементов в формировании представления

Кроме замещаемых и незамещаемых, в CSS2.1 специфицированы еще два базовых типа элементов: *блочные* (*block-level*) и *строковые* (*inline-level*) *элементы*. Эти типы хорошо знакомы тем авторам, которые имеют опыт работы с разметкой HTML или XHTML и ее отображением в веб-броузерах (рис. 1.1).

Блочные элементы

Блочные элементы (*block-level elements*) генерируют блок, который (по умолчанию) полностью заполняет область содержимого своего родительского элемента; расположение других элементов, помимо этого блока, недопустимо. Тем самым он генерирует «разрывы» до и после блока элемента. Самые известные блочные элементы HTML: `p` и `div`. Замещаемые элементы могут быть блочными, но обычно они таковыми не являются.

Элементы списка – особый случай блочных элементов. Помимо того, что их поведение аналогично поведению других блочных элементов, они генерируют маркер – как правило, буллет для ненумерованных списков и число для нумерованных – который «прикрепляется» к блоку элемента. Наличие маркера – это единственное отличие элементов списка от остальных блочных элементов.

Строковые элементы

Строковые элементы (*inline-level elements*) генерируют блок элемента в строке текста и не разрывают ее. Лучший пример строкового элемента – тег `a` в XHTML. Также можно упомянуть элементы `strong` и `em`. Они не создают разрывов текста, поэтому могут находиться внутри содержимого другого элемента, не нарушая его внешний вид.

Обратите внимание, что хотя термины «блок» и «строка» в XHTML имеют много общего, между блочными и строковыми элементами существует одно принципиальное различие. В HTML и XHTML блочные

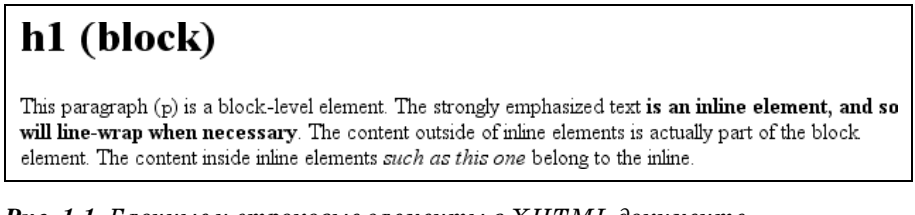


Рис. 1.1. *Блочные и строковые элементы в XHTML-документе*

элементы не могут происходить от строковых. В CSS нет ограничения на вложение ролей.

Посмотрим, как это работает, обратившись к CSS-свойству `display`.

Вы, вероятно, заметили, что оно имеет много значений, но здесь мы рассмотрим только три: `block`, `inline` и `list-item`. Все остальные значения подробно описаны в главе 2 и главе 7.

Сначала сосредоточимся на значениях `block` и `inline`. Рассмотрим следующую разметку:

```
<body>
<p>Это абзац со <em>строковым элементом</em> в нем.</p>
</body>
```

Здесь мы имеем два блочных элемента (`body` и `p`) и строковый элемент (`em`). Согласно спецификации XHTML `em` может происходить от `p`, но не наоборот. Обычно иерархия XHTML составляется таким образом, что строковые элементы могут происходить от элементов уровня блока, но не наоборот.

Напротив, в CSS нет подобных ограничений. Вы можете оставить существующую разметку, но изменить роли формирования представления этих двух элементов следующим образом:

```
p {display: inline;}
em {display: block;}
```

Этим вы заставите элементы генерировать контейнер блока внутри контейнера строки. Это совершенно законно и не нарушает ни одной спецификации. Проблема может возникнуть, если вы попытаетесь изменить порядок вложения элементов на обратный:

```
<em><p>Это абзац, ошибочно включенный в строковый элемент.</p></em>
```

Независимо от того, что вы делаете с ролями формирования представления посредством CSS, в XHTML это недопустимо.

display

Значения:	`none` | `inline` | `block` | `inline-block` | `list-item` | `run-in` | `table` | `inline-table` | `table-row-group` | `table-header-group` | `table-footer-group` | `table-row` | `table-column-group` | `table-column` | `table-cell` | `table-caption` | `inherit`
Исходное значение:	`inline`
Область применения:	все элементы
Наследование:	нет
Вычисляемое значение:	меняется для плавающих, абсолютно позиционированных и корневых элементов (см. CSS2.1, раздел 9.7); в противном случае – как задано

Изменение ролей формирования представления элементов может быть полезным в XHTML-документах и абсолютно необходимо в XML-документах. XML-документ не поддерживает каких-либо предопределенных ролей формирования представления, так что их определение остается в руках автора. Например, вы могли бы задаться вопросом, как будет представлен следующий фрагмент XML:

```
<book>
 <maintitle>Cascading Style Sheets: The Definitive Guide</maintitle>
 <subtitle>Second Edition</subtitle>
 <author>Eric A. Meyer</author>
 <publisher>O'Reilly and Associates</publisher>
 <pubdate>2004</pubdate>
 <isbn>blahblahblah</isbn>
</book>
<book>
 <maintitle>CSS2 Pocket Reference</maintitle>
 <author>Eric A. Meyer</author>
 <publisher>O'Reilly and Associates</publisher>
 <pubdate>2004</pubdate>
 <isbn>blahblahblah</isbn>
</book>
```

Поскольку по умолчанию установлено значение `display` — `inline`, содержимое будет отображаться как обычный текст строки, что показано на рис. 1.2. Такое представление практически бесполезно.

Свойство `display` позволяет определить базовую разметку:

```
book, maintitle, subtitle, author, isbn {display: block;}
publisher, pubdate {display: inline;}
```

Здесь определены пять блочных элементов и два строковых. Это означает, что каждый из блочных элементов будет интерпретироваться как элемент `div` в XHTML, а два строковых элемента будут трактоваться аналогично элементу `span`.

Эта фундаментальная возможность влиять на роли формирования представления делает CSS очень полезными в различных ситуациях. Вы можете в качестве отправной точки взять предыдущие правила, добавить несколько других стилей и получить результат, представленный на рис. 1.3.

Далее в этой книге мы исследуем различные свойства и значения, которые обеспечивают такое представление. Однако сначала нам надо рассмотреть способы связи CSS с документом. Несмотря на всю свою мощь, без установленных связей CSS не сможет воздействовать на до-

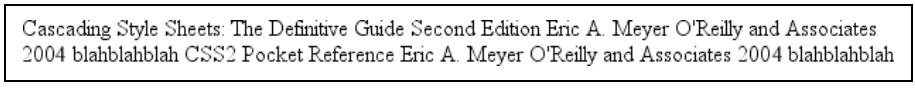


***Рис. 1.2.** Стандартное представление XML-документа*

кумент. Мы рассмотрим привязку в XHTML, поскольку она наиболее привычна.

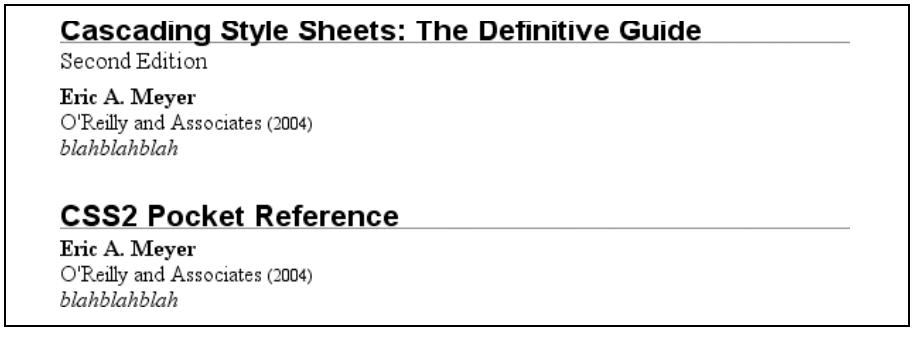


Рис. 1.3. *Представление XML-документа с использованием стилей*

Объединение CSS и XHTML

Я упомянул, что документы HTML и XHTML имеют схожую структуру, и этот момент следует повторить. Кстати, именно в этом состоит проблема со старыми веб-страницами: слишком часто мы забываем, что документы имеют внутреннюю структуру, которая совершенно отличается от визуальной. В погоне за созданием веб-страниц, выглядящих наилучшим образом, мы забываем, что страницы должны содержать информацию об их структуре.

Эта структура является неотъемлемой частью взаимоотношений между XHTML и CSS; без структуры подобная взаимосвязь вообще невозможна. Чтобы лучше понять это, давайте рассмотрим пример XHTML-документа и разберем его по частям:

```
<html>
<head>
<title>Eric's World of Waffles</title>
<link rel="stylesheet" type="text/css" href="sheet1.css" media="all" />
<style type="text/css">
@import url(sheet2.css);
h1 {color: maroon;}
body {background: yellow;}
/* Вот мои стили! Вау! */
</style>
</head>
<body>
<h1>Waffles!</h1>
<p style="color: gray;">The most wonderful of all breakfast foods is
the waffle--a ridged and cratered slab of home-cooked, fluffy goodness
that makes every child's heart soar with joy. And they're so easy to make!
Just a simple waffle-maker and some batter, and you're ready for a morning
of aromatic ecstasy!
```

```
</p>
</body>
</html>
```

Приведенная выше разметка показана на рис. 1.4.

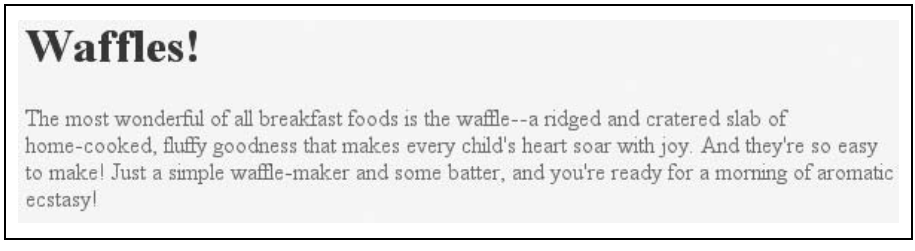


***Рис. 1.4.** Простой документ*

Теперь проанализируем различные способы, которыми этот документ связывается с CSS.

Тег link

Сначала рассмотрим применение тега `link`:

```
<link rel="stylesheet" type="text/css" href="sheet1.css" media="all" />
```

Тег `link` – это малозаметный, но совершенно полноценный тег, который годами маялся в спецификации HTML, ожидая достойного применения. Его основное назначение – предоставить авторам HTML возможность устанавливать связь между документом, содержащим тег `link`, и другими документами. В случае CSS он связывает таблицы стилей с документом; на рис. 1.5 таблица стилей *sheet1.css* связана с документом.

Таблицы стилей, не являющиеся частью HTML-документа, но используемые им, называют *внешними таблицами стилей* (*external style sheets*). Это название дано потому, что такие таблицы стилей являются внешними по отношению к HTML-документу. (Надо же!)

Чтобы внешняя таблица стилей была успешно загружена, тег `link` должен быть помещен внутрь элемента `head` и не может находиться внутри какого-либо другого элемента, так же как `title`. В результате веб-броузер отыщет и загрузит таблицу стилей, после чего будет использовать любые содержащиеся в ней стили для формирования представления HTML-документа, как показано на рис. 1.5.

А каков формат внешней таблицы стилей? Это просто список таких же правил, какие мы видели в предыдущем разделе и в примере XHTML-документа, но в данном случае правила сохраняются в отдельном файле. Просто помните, что в таблицу стилей не может быть включен ни XHTML, ни любой другой язык разметки, а только стилевые правила. Вот как может выглядеть содержимое внешней таблицы стилей:

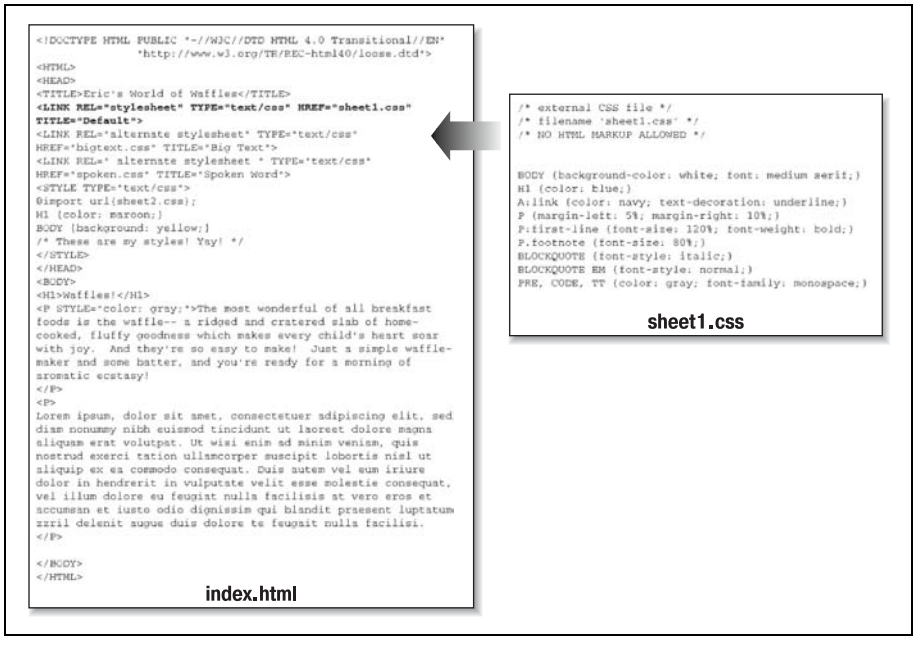


Рис. 1.5. *Применение внешней таблицы стилей к документу*

```
h1 {color: red;}
h2 {color: maroon; background: white;}
h3 {color: white; background: black;
 font: medium Helvetica;}
```

Вот и все: вообще никакой HTML-разметки или комментариев, только простые и понятные объявления стилей. Их сохраняют в простом текстовом файле и обычно дают расширение *.css*, как в *sheet1.css*.

Внешняя таблица стилей не может содержать какой-либо разметки документа, только правила и комментарии CSS (и те, и другие рассматриваются в этой главе позже). Наличие во внешней таблице стилей разметки может привести к тому, что некоторые ее части или вообще вся таблица будут проигнорированы.

Расширение имени файла не является обязательным, но некоторые более старые броузеры не смогут опознать файл как содержащий таблицу стилей, если он не заканчивается на *.css*, даже если вы *включили* верный тип элемента `text/css` в элемент `link`. Кстати, некоторые вебсерверы не будут передавать файл как `text/css`, если его имя не заканчивается на *.css*, хотя обычно с этим можно справиться, изменив файлы конфигурации сервера.

Атрибуты

Все остальные атрибуты и значения тега `link` довольно просты. Атрибут `rel` отвечает за установку взаимосвязи и в данном случае имеет значение `stylesheet`. Атрибуту `type` всегда присваивается значение `text/css`. Оно описывает тип данных, которые будут загружены с помощью тега `link`. Таким образом, веб-броузер знает, что таблица стилей – это таблица стилей CSS. Это определяет, как броузер будет обрабатывать импортируемые им данные. Однако в будущем возможно использование других языков описания стилей, поэтому важно объявлять, какой именно язык используется.

Далее расположен атрибут `href`. Значением этого атрибута является URL таблицы стилей. Он может быть как абсолютным, так и относительным в зависимости от ваших потребностей. В нашем примере URL относительный. Но он мог бы быть и абсолютным: `http://www.meyerweb.com/sheet1.css`.

И наконец, у нас есть атрибут `media`. В нашем случае он имеет значение `all`, т. е. эта таблица стилей должна использоваться для всех средств визуального представления. CSS2 определяет несколько допустимых значений для этого атрибута:

`all`

Указывается для всех устройств.

`aural`

Задают для синтезаторов речи, средств чтения с экрана и других аудио-представлений документа.

`braille`

Задают при выводе документа на устройствах отображения азбуки Брайля.

`embossed`

Указывается при печати документа азбукой Брайля.

`handheld`

Задается для переносных устройств, таких как карманные компьютеры или смартфоны.

`print`

Задается при распечатке документа на обычном принтере, а также при выводе в окне просмотра документа перед печатью.

`projection`

Указывают для проекционных устройств, таких как цифровой проектор, используемый для демонстрации слайдов, сопровождающей чтение доклада.

`screen`

Указывают при представлении документа в экранном устройстве, таком как монитор компьютера. Все веб-броузеры, выполняющие-

ся на подобных системах, являются экранными агентами пользователя.

`tty`

Применяется для устройств, использующих набор символов с фиксированной шириной, таких как телетайп.

`tv`

Задают при отображении документа на телевизоре.

Основная масса этих устройств не поддерживается ни одним из современных веб-броузеров. Три из наиболее широко распространенных: `all`, `screen` и `print`. Opera также поддерживает тип `projection`, что позволяет представлять документ в виде последовательности слайдов.

Таблицу стилей можно использовать не только для одного устройства, но и для нескольких, указав разделенный запятыми список этих устройств. Таким образом, например, вы можете применить связанную таблицу стилей как для экранного, так и для проекционного устройства:

```
<link rel="stylesheet" type="text/css" href="visual-sheet.css"
   media="screen, projection" />
```

Обратите внимание, что с документом может быть ассоциирована не одна связанная таблица стилей. В таких случаях в исходном представлении документа будут применяться только те теги `link`, атрибут `rel` которых имеет значение `stylesheet`. Таким образом, если бы вы захотели связать две таблицы стилей, *basic.css* и *splash.css*, это выглядело бы следующим образом:

```
<link rel="stylesheet" type="text/css" href="basic.css" />
<link rel="stylesheet" type="text/css" href="splash.css" />
```

При этом броузер будет загружать обе таблицы стилей, комбинируя правила обеих, и применять их все к документу. (Как именно происходит комбинирование таблиц, мы увидим в главе 3, но на данный момент давайте просто примем, что они комбинируются.) Например:

```
<link rel="stylesheet" type="text/css" href="basic.css" />
<link rel="stylesheet" type="text/css" href="splash.css" />

<p class="a1">Этот абзац будет серым, только если
применены стили из таблицы стилей 'basic.css'.</p>
<p class="b1">Этот абзац будет серым, только если
применены стили из таблицы стилей 'splash.css'.</p>
```

Единственный атрибут, который не приведен в примере разметки, но мог бы там находиться, – это `title`. Он редко используется, но в будущем может обрести важное значение, а его неправильное применение может иметь неожиданные последствия. Почему? Мы рассмотрим это в следующем разделе.

Альтернативные таблицы стилей

Существует возможность определения *альтернативных таблиц стилей* (*alternate style sheets*). Для этого атрибуту `rel` присваивается значение `alternate stylesheet`, и тогда таблицы стилей задействуются в представлении документа только в том случае, если их выбирает пользователь.

Для организации работы с альтернативными таблицами стилей броузер возьмет значения атрибутов `title` элементов `link` и создаст из них список альтернативных стилей. Так что вы можете написать следующее:

```
<link rel="stylesheet" type="text/css"
   href="sheet1.css" title="По умолчанию" />
<link rel="alternate stylesheet" type="text/css"
   href="bigtext.css" title="Крупный текст" />
<link rel="alternate stylesheet" type="text/css"
   href="zany.css" title="Сумасшедшие цвета!" />
```

В результате пользователи могут выбирать стили, и броузер переключится с первого (в данном случае помечен словами «По умолчанию») на любой другой выбранный пользователем стиль. На рис. 1.6 показан один из вариантов реализации этого механизма выбора.

Альтернативные таблицы стилей поддерживаются большинством броузеров, созданных компанией Netscape Communications, такими как Mozilla и Netscape 6+, и в Opera 7. Броузеры Internet Explorer не поддерживают альтернативные таблицы стилей, но в них того же эффекта можно достичь с помощью JavaScript.

Альтернативные таблицы стилей можно группировать, присваивая их атрибуту `title` одинаковые значения. Благодаря этому пользователь может выбирать разные представления сайта, как на экране, так и при печати. Например:

```
<link rel="stylesheet" type="text/css"
   href="sheet1.css" title="По умолчанию" media="screen" />
```

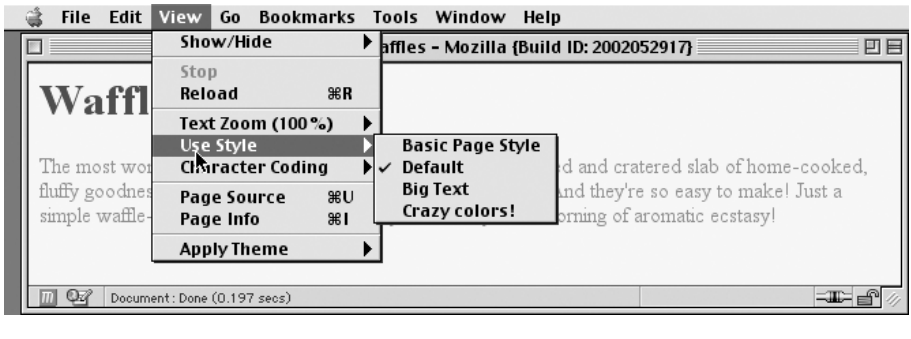


***Рис. 1.6.** Броузер, предлагающий выбрать альтернативные таблицы стилей*

```
<link rel="stylesheet" type="text/css"
   href="print-sheet1.css" title="По умолчанию" media="print" />
<link rel="alternate stylesheet" type="text/css"
   href="bigtext.css" title="Крупный текст" media="screen" />
<link rel="alternate stylesheet" type="text/css"
   href="print-bigtext.css" title="Крупный текст" media="print" />
```

Если в соответствующем агенте пользователя из списка альтернативных таблиц стилей выбран вариант «Крупный текст», то стиль документа при отображении на экране будет формироваться по таблице *bigtext.css*, а при выводе на печать – по *print-bigtext.css*. Ни в одном из устройств не будут использованы ни *sheet1.css*, ни *print-sheet1.css*.

Почему? Потому что если в теге `link` с атрибутом `rel`, имеющим значение `stylesheet`, указан заголовок, данная таблица стилей помечается как *предпочтительная таблица стилей* (*preferred style sheet*). Это значит, что из всех альтернативных таблиц стилей ее применение предпочтительнее, и именно она будет использоваться при первом выводе документа на экран. Однако если выбрать альтернативную таблицу стилей, предпочтительная таблица использоваться *не будет*.

Более того, если обозначить как предпочтительные несколько таблиц стилей, все они, кроме одной, будут проигнорированы. Рассмотрим:

```
<link rel="stylesheet" type="text/css"
   href="sheet1.css" title="Раскладка по умолчанию" />
<link rel="stylesheet" type="text/css"
   href="sheet2.css" title="Шрифт по умолчанию" />
<link rel="stylesheet" type="text/css"
   href="sheet3.css" title="Цвета по умолчанию" />
```

Все три элемента `link` теперь обозначены как предпочтительные таблицы стилей, благодаря наличию во всех трех атрибута `title`, но на самом деле только один из них выступает в этом качестве. Остальные два будут полностью проигнорированы. Какие именно из них? Точно ответить на этот вопрос невозможно, поскольку ни HTML, ни XHTML не предоставляют метода, позволяющего определить, какая из предпочтительных таблиц стилей должна быть проигнорирована, а какая нет.

Если просто не указать заголовок таблицы стилей, то она становится *постоянной таблицей стилей* (*persistent style sheet*) и всегда используется в представлении документа. Зачастую именно этого и хочет автор.

Элемент style

Элемент `style` – это единственный способ включения таблиц стилей и располагается он в самом документе:

```
<style type="text/css">
```

В элементе `style` всегда должен присутствовать атрибут `type`; если документ использует CSS, его значение должно быть `"text/css"`, так же как и для элемента `link`.

Элемент style должен всегда начинаться с <style type="text/css">, как показано в предыдущем примере. Далее следуют один или несколько стилей, и все завершается закрывающим тегом </ style>. Также элементу style можно присвоить атрибут media, допустимые значения которого не отличаются от тех, которые обсуждались ранее при рассмотрении связанных таблиц стилей.

Стили, включаемые между открывающим и закрывающим тегами style, называют *таблицей стилей документа* (*document style sheet*), или *вложенной таблицей стилей* (*embedded style sheet*), поскольку эта таблица стилей вложена в документ. В ней будут содержаться применяемые к документу стили, но она также может включать ссылки на внешние таблицы стилей с помощью директивы @import.

Директива @import

Теперь мы займемся тем, что находится внутри тега style. Для начала у нас есть аналог элемента link – директива @import:

```
@import url(sheet2.css);
```

Так же как и link, @import может указывать веб-броузеру на необходимость загрузки внешней таблицы стилей и использования ее стилей при формировании представления HTML-документа. Единственное основное отличие заключается в синтаксисе и размещении команды. Как видите, @import находится в контейнере style. Она должна располагаться перед всеми остальными правилами CSS, иначе она вообще не будет работать. Рассмотрим такой пример:

```
<style type="text/css">
@import url(styles.css); /* @import идет самой первой */
h1 {color: gray;}
</style>
```

В документе может быть несколько директив выражения @import, как и тегов link. Однако, в отличие от link, все указанные в директивах @import таблицы стилей будут загружены и использованы; с помощью @import невозможно назначить альтернативные таблицы стилей. Итак, исходя из следующей разметки:

```
@import url(sheet2.css);
@import url(blueworld.css);
@import url(zany.css);
```

все три внешние таблицы стилей будут загружены, и все их правила будут использоваться в визуальном представлении документа.

Многие старые броузеры не могут обрабатывать разнообразные формы директивы @import. Фактически это может применяться специально для «сокрытия» стилей от этих броузеров. Более подробную информацию можно найти по адресу *http://w3development.de/css/hide_css_from_browsers*.

Как и в случае элемента link, можно ограничить применение импортируемой таблицы стилей одним или несколькими устройствами, перечислив их после URL таблицы стилей:

```
@import url(sheet2.css) all;
@import url(blueworld.css) screen;
@import url(zany.css) projection, print;
```

Директива @import крайне полезна для переноса стилей между внешними таблицами стилей. Поскольку внешние таблицы стилей не могут содержать никакой разметки документа, элемент link применить нельзя, а директиву @import можно. Следовательно, можно себе представить внешнюю таблицу стилей такого содержания:

```
@import url(http://example.org/library/layout.css);
@import url(basic-text.css);
@import url(printer.css) print;
body {color: red;}
h1 {color: blue;}
```

Ну, может быть, не именно эти стили, но идея ясна. Обратите внимание на использование в примере как абсолютного, так и относительных URL. Можно применять любую форму URL, так же как и в link.

Также заметьте, что директивы @import находятся в начале таблицы стилей, как и в примере нашего документа. CSS требует, чтобы директива @import располагалась в таблице стилей раньше всех остальных правил. Директива @import, расположенная после правил (например, после body {color: red;}), будет проигнорирована соответствующими агентами пользователя.

Internet Explorer для Windows не игнорирует директивы @import, даже если они следуют после остальных правил. Поскольку другие броузеры игнорируют неверно размещенные директивы @import, очень легко ошибиться – поставить директивы @import не туда, куда надо, и таким образом изменить представление в других броузерах.

Действительные правила стилей

После инструкции @import в нашем примере мы находим несколько обычных правил стилей. На данный момент совершенно не важно, что они реально означают, хотя вы, вероятно, можете догадаться, что они задают красно-коричневый цвет для элементов h1 и желтый фон для элементов body:

```
h1 {color: maroon;}
body {background: yellow;}
```

Аналогичные правила встречаются в большинстве встроенных таблиц стилей – простых и сложных, коротких и длинных. Документ, в котором элемент style не содержит никаких правил, – редкость.

Обратная совместимость

Необходимо предупредить тех, кто беспокоится о том, чтобы сделать свои документы доступными для более старых броузеров. Вероятно, вы знаете, что броузеры игнорируют теги, которые не могут распознать; например, если веб-страница содержит тег `blooper`, броузеры полностью проигнорируют этот тег, потому что они его не знают.

То же касается и таблиц стилей. Если броузер не знает элементов `<style>` и `</style>`, он проигнорирует их. Однако это не означает, что будут проигнорированы и объявления, содержащиеся между этими тегами, потому что броузер интерпретирует их как обычный текст. Поэтому объявления стилей появятся в верхней части вашей страницы! (Конечно, броузер должен игнорировать текст, потому что он не является частью элемента `body`, но это происходит не всегда.)

Для решения этой проблемы рекомендуется заключать ваши объявления в тег комментария. В приведенном здесь примере начало тега комментария находится сразу после открывающего тега `style`, а конец комментария – сразу перед закрывающим тегом `style`:

```
<style type="text/css"><!--
@import url(sheet2.css);
h1 {color: maroon;}
body {background: yellow;}
--></style>
```

В результате старые броузеры проигнорируют объявления, так же как и теги `style`, потому что комментарии HTML не отображаются. Но те броузеры, которые понимают CSS, смогут прочитать таблицу стилей.

Комментарии CSS

В CSS также предусмотрены комментарии. Они очень похожи на комментарии в C/C++, поскольку ограничиваются символами `/*` и `*/`:

```
/* Это комментарий CSS1 */
```

Комментарии могут распространяться на несколько строк, как и в C++:

```
/* Это комментарий CSS1, и он может занимать
 несколько строк без всяких проблем. */
```

Важно помнить, что комментарии CSS не могут быть вложенными. Поэтому следующий пример не будет правильным:

```
/* Это комментарий, в котором мы находим
 другой комментарий, что НЕПРАВИЛЬНО
   /* Другой комментарий */
 и вновь первый комментарий */
```

Однако вряд ли вложенные комментарии кому-нибудь когда-нибудь понадобятся, поэтому данное ограничение не имеет особого значения.

Можно случайно получить «вложенные» комментарии, если временно закомментировать большой фрагмент таблицы стилей, уже содержащий комментарий. Поскольку в CSS вложенные комментарии не допускаются, «внешний» комментарий будет заканчиваться в точке окончания «внутреннего».

Если вы хотите поместить комментарии на одной строке с правилом, следует быть аккуратным. Например, здесь все правильно:

```
h1 {color: gray;}   /* Этот комментарий CSS растянулся на несколько строк, */
h2 {color: silver;} /* но поскольку он размещается параллельно со */
p {color: white;}   /* стилями, каждая строка должна быть */
pre {color: gray;}  /* разграничена знаками комментариев. */
```

Если в этом примере не ограничивать комментарий на каждой строке, большая часть таблицы стилей станет частью комментария и поэтому не будет работать:

```
h1 {color: gray;}   /* Этот комментарий CSS растянулся на несколько
h2 {color: silver;}  строк, но поскольку он не разграничен
p {color: white;}    знаками комментариев, последние три
pre {color: gray;}   стиля стали частью комментария. */
```

В этом примере только первое правило (`h1 {color: gray;}`) будет применено к документу. Остальные правила как часть комментария игнорируются механизмом визуализации броузера.

Продолжим разбор нашего примера, чтобы увидеть, как можно поместить информацию CSS в теги XHTML!

Подставляемые в строку стили

В тех случаях, когда вы хотите просто назначить несколько стилей отдельному элементу, не применяя встроенные или внешние таблицы стилей, задайте *подставляемый в строку стиль* (*inline style*) посредством HTML-атрибута `style`:

```
<p style="color: gray;">Самый прекрасный завтрак -
это вафля, рифленый кусочек домашнего воздушного совершенства...
</p>
```

Атрибут `style` может быть ассоциирован с любым тегом HTML, за исключением тех, которые располагаются вне элемента `body` (`head` или `title`, например).

Синтаксис атрибута `style` достаточно прост. Кстати, он очень похож на объявления, размещающиеся в контейнере `style`, за исключением того, что фигурные скобки заменяются двойными кавычками. Таким образом, согласно выражению `<p style="color: maroon; background: yellow;">` цвет текста станет красно-коричневым, а фон – желтым, но *только для этого абзаца*. Ни на какую другую часть документа это объявление не повлияет.

Заметьте, что в подставляемый в строку атрибут `style` можно ввести только блок описания, а не всю таблицу стилей. Следовательно, нельзя вставить в атрибут `style` ни директиву `@import`, ни включить какие-либо полные правила. Вы можете помещать в значение атрибута `style` только то, что может находиться в правиле между фигурными скобками.

Применение атрибута `style` в общем нежелательно. Кроме того, спецификацией XHTML 1.1 он отмечен как нерекомендуемый и вряд ли появится в других языках XML, кроме XHTML. При размещении стилей в атрибуте `style` некоторые из основных преимуществ CSS – способность организовывать централизованные стили, управляющие представлением всего документа или всех документов веб-сервера, – оказываются утраченными. Подставляемые в строку стили не намного лучше, чем тег `font`, хотя они обладают существенно большей гибкостью.

Заключение

Применяя CSS, можно полностью изменить способ представления элементов агентом пользователя. Это может быть сделано просто с помощью свойства `display` или по-другому – путем связывания таблиц стилей с документом. Пользователь никогда не будет знать, сделано ли это с помощью внешних или встроенных таблиц стилей или даже через подставляемый в строку стиль. В случае применения внешних таблиц стилей действительно важно, каким образом они делают возможным для авторов размещение всей информации о представлении сайта в одном месте и как они направляют туда все документы. Это не только облегчает обновление и эксплуатацию сайта, но помогает сохранить пропускную способность канала передачи данных, поскольку из документов удаляется вся информация о представлении.

Чтобы использовать CSS на полную мощность, авторам надо знать, как связать набор стилей с элементами документа. Чтобы полностью понять, как CSS со всем этим справляется, необходимо глубоко понимать, каким образом CSS выбирает части документа для стилевого оформления, что и обсуждается в следующей главе.

2

Селекторы

Одно из основных преимуществ CSS – особенно для разработчиков – это возможность легко применять набор стилей ко всем однотипным элементам. Не впечатляет? Представьте, отредактировав всего одну строку CSS, можно изменить цвета всех заголовков в документе. Вам не нравится этот синий цвет? Изменяете одну строку кода, и все заголовки станут фиолетовыми, желтыми, красно-коричневыми или любого другого цвета. Это позволяет разработчику сосредоточиться на дизайне, а не на рутинной работе. Если кому-то захочется увидеть заголовки в другом оттенке зеленого, просто подправьте стиль и нажмите Reload. *Вуаля!* Через несколько секунд результат будет представлен на всеобщее обозрение.

Конечно, CSS не избавит вас от всех трудностей, например, с его помощью нельзя изменить цвета GIF-изображений, но можно намного упростить внесение некоторых глобальных изменений. Итак, начнем с селекторов и структуры.

Основные правила

Как я уже упоминал, главная особенность CSS – возможность применять определенные правила ко всему набору типов элементов документа. Предположим, вы хотите установить серый цвет для всех элементов `h2`. В старом добром HTML это придется делать путем добавления тегов `<FONT COLOR="gray">...</FONT>` во все элементы `h2`:

```
<h2><font color="gray">Это текст элемента h2</font></h2>
```

Очевидно, что если в документе много элементов `h2`, это утомительный процесс. Еще хуже, если позднее вы решите, что все элементы `h2` должны быть зелеными, а не серыми, и придется вновь повсюду вручную править теги.

CSS предоставляет вам возможность создавать правила, которые легко менять, редактировать и применять ко всем определяемым вами элементам текста (следующий раздел объяснит механизм действия этих правил). Например, чтобы сделать все элементы `h2` серыми, достаточно один раз написать правило:

```
h2 {color: gray;}
```

Для того чтобы изменить цвет текста всех элементов `h2`, например на серебристый, просто поменяйте правило:

```
h2 {color: silver;}
```

Структура правила

Чтобы лучше понять, что же такое правила, давайте разберем их структуру.

Каждое правило имеет две основные части: *селектор* (*selector*) и *блок объявлений* (*declaration block*). Блок объявлений состоит из одного или более *объявлений* (*declarations*), а каждое объявление представляет собой сочетание *свойства* (*property*) и *значения* (*value*). Каждая таблица стилей образуется из наборов правил. Части правила показаны на рис. 2.1.

Селектор, расположенный в левой части правила, определяет, на какие части документа распространяется правило. На рис. 2.1 выбраны элементы `h1`. Если бы селектором был `p`, тогда выбирались бы все элементы `p` (абзацы).

В правой части правила находится блок объявлений, образованный одним или несколькими объявлениями. Каждое объявление представляет собой сочетание свойства CSS и значения этого свойства. На рис. 2.1 представлен блок, содержащий два объявления. Первое определяет, что согласно правилу цвет (`color`) указанных элементов документа будет красным (`red`), а второе – что фон (`background`) этих элементов будет желтым (`yellow`). Таким образом, все элементы `h1` (определенные селектором) этого документа будут выводиться красным текстом на желтом фоне.

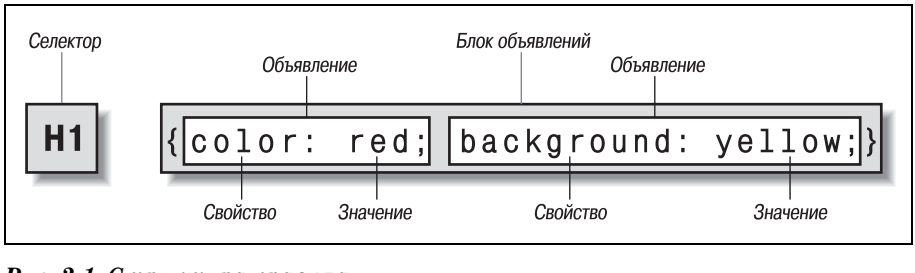


***Рис. 2.1.** Структура правила*

Селекторы элементов

Чаще всего, но не всегда, селектор – это элемент HTML. Например, если CSS-файл содержит стили для XML-документа, селектор может выглядеть примерно так:

```
QUOTE  {color: gray;}
BIB  {color: red;}
BOOKTITLE  {color: purple;}
MYElement  {color: red;}
```

Иначе говоря, элементы документа играют роль базовых селекторов. В XML селектором может быть все что угодно, поскольку XML разрешает создание новых языков разметки, в которых в качестве имени элемента может выступать практически любая последовательность символов. С другой стороны, если вы создаете стили для HTML-документа, селектором обычно будет один из элементов HTML, например `p`, `h3`, `em`, `a` или даже `html`. Например:

```
html  {color: black;}
h1  {color: gray;}
h2  {color: silver;}
```

Результаты применения этой таблицы стилей показаны на рис. 2.2.

Применив глобальные правила к элементам, можно переносить стили с одного элемента на другой. Допустим, вы решили, что на рис. 2.2 серым должен быть текст абзаца, а не заголовок `h1`. Без проблем. Просто меняем селектор `h1` на `p`:

```
html  {color: black;}
p  {color: gray;}
h2  {color: silver;}
```

Результаты показаны на рис. 2.3.

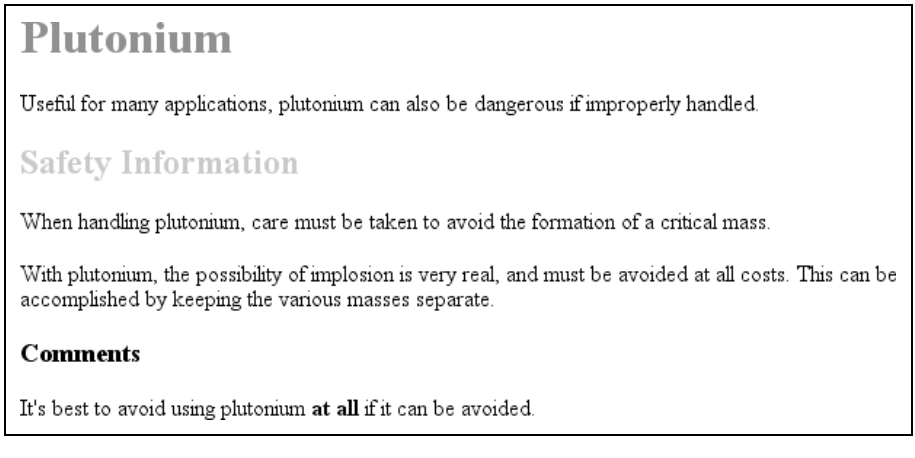


Рис. 2.2. *Простое стилевое оформление простого документа*

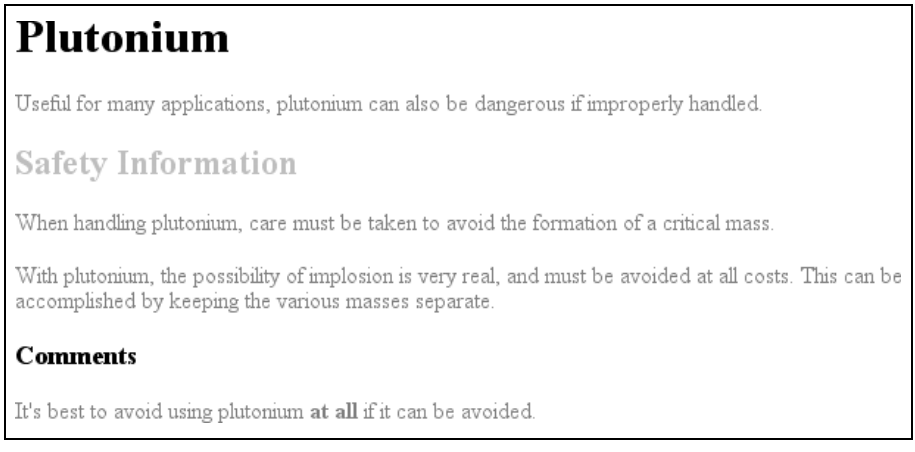


Рис. 2.3. *Перенос стиля одного элемента на другой*

Объявления и ключевые слова

В блок объявлений входит одно или несколько объявлений. Формат объявлений обычно такой: *имя свойства*, за которым следует двоеточие, затем *значение* и точка с запятой. После двоеточия и точки с запятой может быть произвольное количество пробелов (в частности, возможно отсутствие пробела). Практически во всех случаях значение – это или отдельное ключевое слово, или список из нескольких допустимых для данного свойства ключевых слов, разделенных пробелами. Если указать неверное свойство или значение, будет проигнорировано все объявление целиком. Поэтому следующие два объявления не будут выполнены:

```
brain-size: 2cm;     /* неизвестное свойство */
color: ultraviolet;  /* неизвестное значение */
```

В случаях, когда допускается указывать в качестве значения свойства более одного ключевого слова, ключевые слова обычно разделяются пробелами. Не все свойства могут принять несколько ключевых слов, но многие, такие как свойство `font`, могут. Скажем, текст абзаца должен быть набран шрифтом Helvetica среднего размера, как показано на рис. 2.4.

Правило должно выглядеть так:

```
p {font: medium Helvetica;}
```

Обратите внимание на пробел между значениями `medium` и `Helvetica`, каждое из которых является ключевым словом (первое – размер шрифта, а второе – фактическое имя шрифта). Благодаря пробелу агент пользователя различает эти два ключевых слова и применяет их правильно. Точка с запятой указывает на завершение объявления.

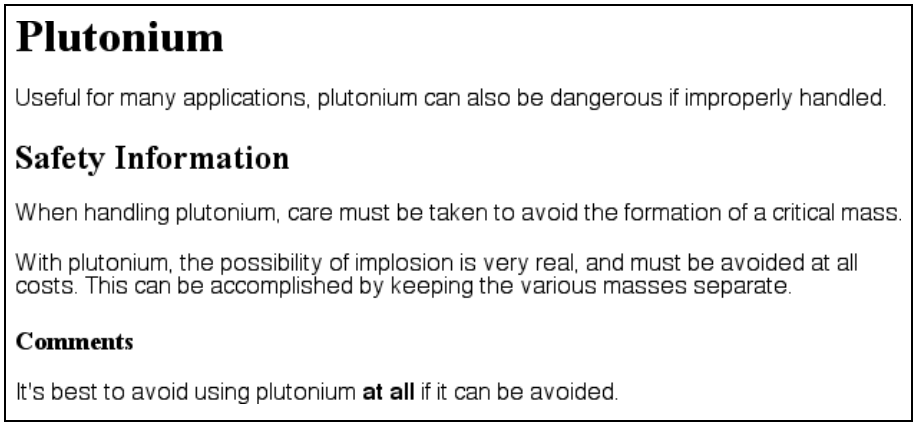


***Рис. 2.4.** Результаты применения значения свойства с несколькими ключевыми словами*

Эти разделенные пробелом слова называются ключевыми, потому что все вместе они образуют значение свойства. Рассмотрим следующее вымышленное правило:

```
rainbow: red orange yellow green blue indigo violet;
```

Конечно же, нет такого свойства – `rainbow`, и два из указанных названий цветов тоже недействительны, но этот пример полезен в пояснительных целях. Значение элемента `rainbow` – это `red orange yellow green blue indigo violet`, и семь ключевых слов образуют одно уникальное значение. Мы можем переопределить значение `rainbow` следующим образом:

```
rainbow: infrared red orange yellow green blue indigo violet ultraviolet;
```

Теперь мы получили новое значение для `rainbow`, составленное из девяти ключевых слов вместо семи. Хотя имя двух значений одно и то же, они уникальны и отличаются друг от друга, как нуль и единица.

Как видите, ключевые слова CSS разделяются пробелами за одним исключением. CSS-свойство `font` – это единственное место, где два ключевых слова могут быть разделены прямым слэшем (/). Например:

```
h2 {font: large/150% sans-serif;}
```

Слэш разделяет ключевые слова, которые задают размер шрифта и высоту строки элемента. Это единственное место, где допускается применение слэша в описании элемента `font`. Все остальные разрешенные для применения в элементе `font` ключевые слова разделяются пробелами.

Все, что вы видели до сих пор, – основы простых объявлений, но они могут быть намного более сложными. В следующем разделе начнется демонстрация потенциальной мощи CSS.

Группировка

На данный момент мы изучили довольно простые методики применения одного стиля к одному селектору. Но что если вы хотите один и тот же стиль применить к нескольким элементам? Тогда понадобятся несколько селекторов или потребуется применить несколько стилей к элементу или группе элементов.

Группировка селекторов

Предположим, вы хотите, чтобы текст элементов `h2` и абзацев был серого цвета. Проще всего это достигается посредством следующего объявления:

```
h2, p {color: gray;}
```

Поместив в левой части правила разделенные запятой селекторы `h2` и `p`, вы определяете правило, в котором находящийся в правой части стиль (`color: gray;`) применяется к элементам, обозначенным обоими селекторами. Запятая сообщает броузеру о том, что в правило включены два разных селектора. Если запятую опустить, правило приобретет совершенно другое значение, о чем мы поговорим позже в разделе «Селекторы потомков».

Нет никаких ограничений на количество объединяемых в группу селекторов. Например, если необходимо отобразить несколько элементов в сером цвете, можно применить приблизительно такое правило:

```
body, table, th, td, h1, h2, h3, h4, p, pre, strong, em, b, i {color: gray;}
```

Группировка позволяет автору сделать определенные типы назначений стилей более компактными, что способствует сокращению таблицы стилей. Приведенная ниже альтернативная запись формирует абсолютно аналогичный результат, но совершенно очевидно, какую из этих записей легче набрать на клавиатуре:

```
h1 {color: purple;}
h2 {color: purple;}
h3 {color: purple;}
h4 {color: purple;}
h5 {color: purple;}
h6 {color: purple;}

h1, h2, h3, h4, h5, h6 {color: purple;}
```

Группировка делает возможными некоторые интересные варианты. Например, все группы правил в следующем примере эквивалентны: каждая просто демонстрирует свой способ группировки как селекторов, так и описаний:

```
/* группа 1 */
h1 {color: silver; background: white;}
```

```
h2 {color: silver; background: gray;}
h3 {color: white; background: gray;}
h4 {color: silver; background: white;}
b {color: gray; background: white;}

/* группа 2 */
h1, h2, h4 {color: silver;}
h2, h3 {background: gray;}
h1, h4, b {background: white;}
h3 {color: white;}
b {color: gray;}

/* группа 3 */
h1, h4 {color: silver; background: white;}
h2 {color: silver;}
h3 {color: white;}
h2, h3 {background: gray;}
b {color: gray; background: white;}
```

Любая из этих групп сформирует результат, показанный на рис. 2.5. (Эти стили используют сгруппированные объявления, которые рассматриваются в следующем разделе.)

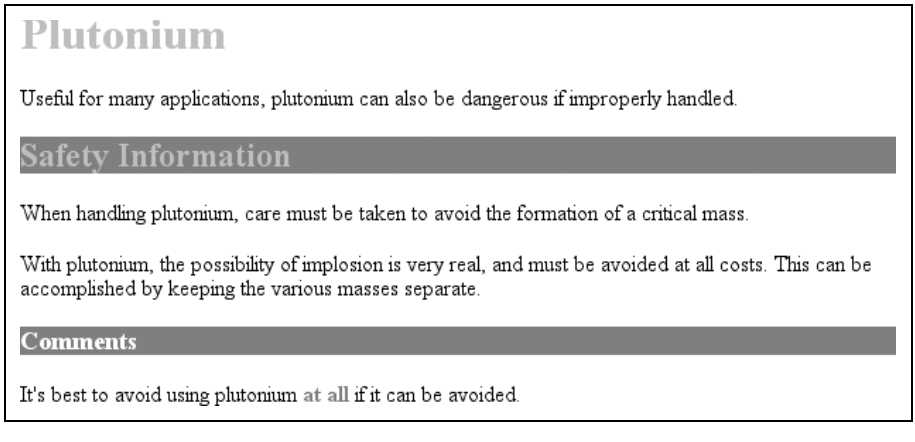


Рис. 2.5. *Результат применения эквивалентных таблиц стилей*

Универсальный селектор

В CSS2 появился новый простой селектор, названный *универсальным селектором* (*universal selector*) и представляемый символом звездочка (*). Этот селектор соответствует любому элементу почти так же, как подстановочный символ в маске имени файла. Например, чтобы сделать все элементы документа красными, можно написать:

```
* {color: red;}
```

Это описание эквивалентно групповому селектору, в котором перечислен каждый содержащийся в документе элемент. Универсальный се-

лектор позволяет присваивать атрибуту color каждого элемента документа значение red одним нажатием клавиши. Однако будьте внимательны: хотя универсальный селектор удобен, его применение может иметь некоторые неожиданные последствия, о чем мы поговорим в следующей главе.

Группировка объявлений

Селекторы можно группировать в одно правило, а следовательно, также можно группировать и объявления. Предположим, вы хотите, чтобы текст всех элементов h1 был представлен шрифтом Helvetica фиолетового цвета высотой в 18 пикселов на фоне цвета морской волны (и вы не прочь сбить с толку своих читателей). Вы могли бы написать свои стили так:

```
h1 {font: 18px Helvetica;}
h1 {color: purple;}
h1 {background: aqua;}
```

Но этот метод неэффективен. Представьте, что подобный список создается для элемента, у которого будет 10 или 15 стилей! В таком случае объявления можно сгруппировать:

```
h1 {font: 18px Helvetica; color: purple; background: aqua;}
```

При этом результат будет абсолютно аналогичным тому, который получается и в случае применения показанной выше таблицы стилей, состоящей из трех строк.

Обратите внимание, что точка с запятой в конце каждого объявления очень важна при группировке. Броузеры игнорируют пробелы в таблицах стилей, и при их анализе агент пользователя полностью зависит от правильности синтаксиса. Стили можно смело форматировать следующим образом:

```
h1 {
  font: 18px Helvetica;
  color: purple;
  background: aqua;
}
```

Однако если опустить вторую точку с запятой, агент пользователя интерпретирует эту таблицу стилей так:

```
h1 {
  font: 18px Helvetica;
  color: purple  background: aqua;
}
```

Поскольку background: не является действительным значением для атрибута color, а также потому, что атрибуту color может быть присвоено только одно ключевое слово, агент пользователя полностью проигнорирует объявление color (включая часть background: aqua). Возможно, цвет

текста в элементах h1 и будет фиолетовым без фона цвета морской волны, но скорее всего, вы не получите и фиолетовых элементов h1. Всем им будет назначен применяемый по умолчанию цвет (обычно черный) вообще без фона. (Объявление font: 18px Helvetica останется в силе, поскольку оно совершенно правильно завершается точкой с запятой.)

Хотя с технической точки зрения завершать последнее объявление правила точкой с запятой вовсе не обязательно, лучше принять это за правило. Во-первых, это выработает у вас привычку завершать объявления точкой с запятой, отсутствие которой является одной из самых распространенных причин ошибок. Во-вторых, если вы решите добавить в правило еще одно объявление, вам не надо беспокоиться о том, что вы забыли вставить дополнительную точку с запятой. И наконец, некоторые старые броузеры, такие как Internet Explorer 3.x, имеют склонность приходить в замешательство, если последнее объявление правила не завершается точкой с запятой. Всех этих неприятностей можно избежать заранее: всегда завершайте объявления точкой с запятой, в каком бы месте правила они ни находились.

Как и группировка селекторов, группировка объявлений – удобный способ, помогающий обеспечить небольшой размер, выразительность и простоту обслуживания таблиц стилей.

Группируем все

Сейчас вы знаете, что можете группировать селекторы и объявления. Сочетая в одном правиле оба типа группировки, можно определять очень сложные стили посредством всего лишь нескольких выражений. Что если вы решили назначить для всех заголовков документа несколько сложных стилей и вам хочется, чтобы одни и те же стили применялись ко всем заголовкам? Это делается так:

```
h1, h2, h3, h4, h5, h6 {color: gray; background: white; padding: 0.5em;
  border: 1px solid black; font-family: Charcoal, sans-serif;}
```

Селекторы сгруппированы, поэтому стили, находящиеся в правой части правила, будут применены ко всем перечисленным заголовкам, а группировка объявлений означает, что все перечисленные стили будут применены к селекторам, приведенным в левой части правила. Результат работы этого правила показан на рис. 2.6.

Этот способ предпочтительнее альтернативного ему развернутого, который выглядел бы примерно так:

```
h1 {color: gray;}
h2 {color: gray;}
h3 {color: gray;}
h4 {color: gray;}
h5 {color: gray;}
h6 {color: gray;}
```

```
h1 {background: white;}
h2 {background: white;}
h3 {background: white;}
```

и растянулся бы на множество строк. Вы *можете* выбрать более долгий путь для написания своих стилей, но я бы не советовал. Редактирование стилей будет настолько же утомительным, как и необходимость повсеместной расстановки тегов `font`!

Можно обеспечить селекторам даже еще больше выразительности и распределить стили по элементам согласно типам информации. Конечно, придется немного потрудиться, но результат, определенно, стоит того.

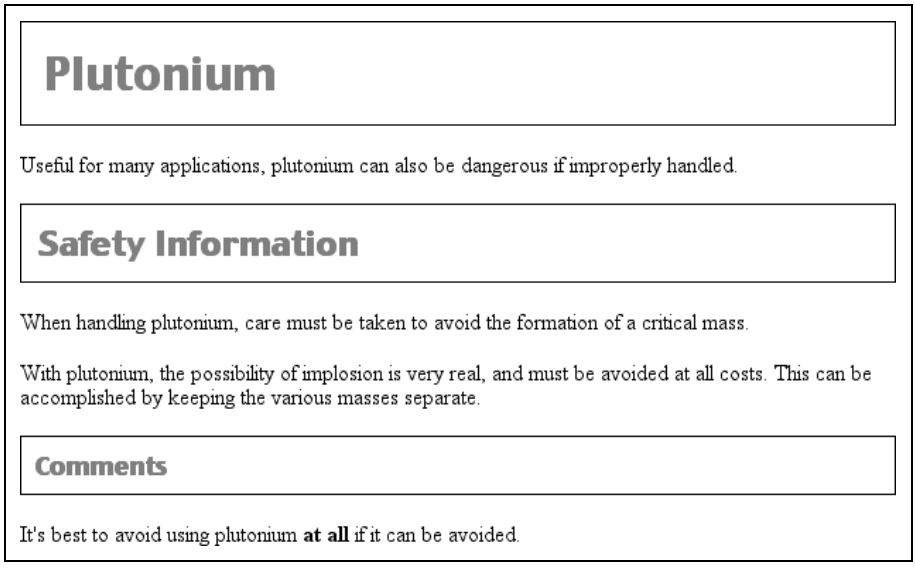


Рис. 2.6. *Группировка селекторов и правил*

Селекторы классов и идентификаторов

До сих пор мы различными способами группировали селекторы и объявления, но используемые нами селекторы были простыми. Они обращались только к элементам документа, а это хорошо лишь до поры до времени: ведь бывают моменты, когда требуется что-то немного более специализированное.

Кроме простых элементов документа есть еще два типа селекторов: *селекторы классов* (*class selectors*) и *селекторы идентификаторов* (*ID selectors*), позволяющие назначать стили независимо от элементов документа. Эти селекторы могут применяться самостоятельно или в сочетании с селекторами элементов. Однако работают они только в том случае, если документ размечен соответствующим образом, поэтому

их применение подразумевает некоторое предварительное планирование и проработку.

Предположим, вы создаете документ, в котором обсуждаются способы обработки плутония. Этот документ содержит различные предупреждения, касающиеся техники безопасности при работе с таким опасным веществом. Каждое предупреждение требуется выделить полужирным шрифтом. Однако вы не знаете, какие элементы будут использоваться для предупреждений. Некоторые из предупреждений могут занимать целые абзацы, тогда как другие могут быть отдельным пунктом длинного списка или небольшим фрагментом текста. В результате невозможно определить правило с помощью простых селекторов. Допустим, вы попробовали сделать это так:

```
p {font-weight: bold;}
```

Полужирным шрифтом будут выделены *все* абзацы, а не только те, в которых содержится предупреждение. Нужен способ, позволяющий выбрать только тот текст, в котором содержатся предупреждения, или, говоря строже, способ выбрать только те элементы, которые являются предупреждениями. Как это сделать? С помощью селекторов класса вы применяете стили только к тем частям документа, которые были отмечены определенным образом, независимо от входящих в них элементов.

Селекторы классов

Самый распространенный способ применения стилей без учета элементов состоит в том, чтобы обратиться к селекторам классов. Однако сначала придется изменить разметку документа таким образом, чтобы обеспечить возможность работы селекторов класса. Введите атрибут `class`:

```
<p class="warning">При работе с плутонием необходимо предпринять меры по
недопущению достижения критической массы.</p>
<p>При работе с плутонием <span class="warning">существует реальная
возможность взрыва, которую надо избежать любой ценой</span>.
Это может быть достигнуто путем разделения масс.</p>
```

Чтобы связать стили селектора класса с элементом, необходимо присвоить соответствующее значение атрибуту `class` данного элемента. В предыдущем коде значение `warning` было присвоено двум элементам: первому абзацу и элементу `span` во втором абзаце.

Вам всего лишь нужен способ применения стилей к этим элементам заданного класса. В HTML-документах возможна очень краткая запись, в которой имени класса предшествует точка (`.`), и оно может быть объединено с простым селектором:

```
*.warning {font-weight: bold;}
```

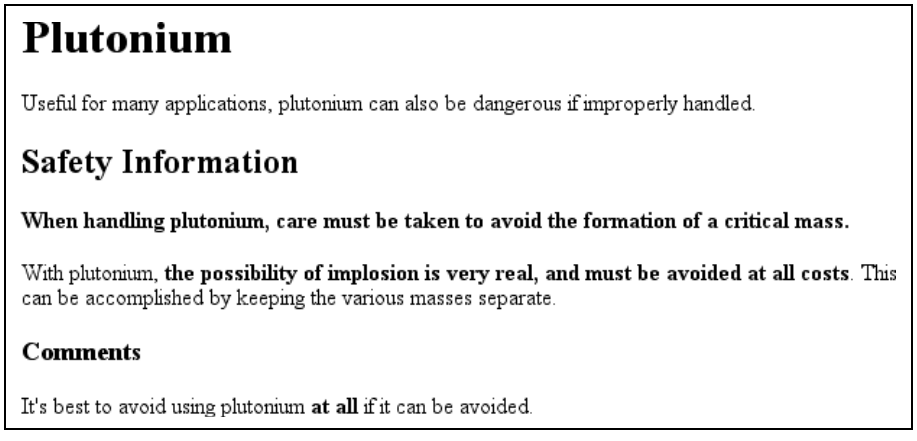


Рис. 2.7. *Применение селектора класса*

В сочетании с показанным ранее примером разметки применение этого простого правила приводит к результату, показанному на рис. 2.7. Другими словами, стиль `font-weight: bold` будет применен ко всем элементам (благодаря наличию универсального селектора), атрибут `class` которых имеет значение `warning`.

Как видите, селектор класса напрямую ссылается на значение, которое будет найдено в атрибуте `class` элемента. Перед этой ссылкой *всегда* ставится точка (`.`), которая отмечает ее как селектор класса. Точка помогает отделить селектор класса от всего, с чем он может быть объединен, например от селектора элемента. Скажем, вам захотелось выделять полужирным шрифтом текст только в том случае, если предупреждением является весь абзац:

```
p.warning {font-weight: bold;}
```

Теперь селектор выбирает только элементы `p`, у которых значение атрибута `class` равно `warning`, и игнорирует все другие элементы независимо от их класса. Селектор `p.warning` расшифровывается так: «Любой абзац, у которого значение атрибута `class` равно `warning`». Поскольку элемент `span` не является абзацем, селектор правила ему не соответствует, и текст не будет выделен полужирным шрифтом.

Если бы вы действительно захотели назначить элементу `span` другие стили, вы могли бы использовать селектор `span.warning`:

```
p.warning {font-weight: bold;}
span.warning {font-style: italic;}
```

В этом случае абзац с предупреждением выделяется полужирным шрифтом, тогда как предупреждения-фрагменты выделяются курсивом. Каждое правило применяется только к определенному типу сочетания элемент/класс, и поэтому не распространяется на другие элементы.

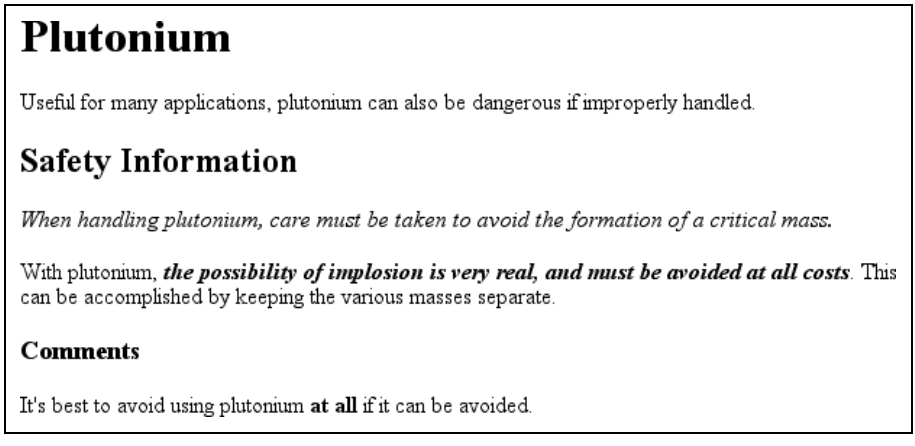


Рис. 2.8. *Применение универсальных и специальных селекторов для объединения стилей*

Другой вариант – комбинирование общего селектора классов и селектора классов, предназначенного для конкретного элемента, чтобы сделать стили еще более полезными, как в следующей разметке:

```
.warning {font-style: italic;}
span.warning {font-weight: bold;}
```

Результаты показаны на рис. 2.8.

В данном случае любой текст предупреждения будет выделен курсивом, но только текст элементов `span`, значение атрибута `class` которых равно `warning`, будет выделен полужирным шрифтом и курсивом.

Обратите внимание на формат универсального селектора классов в предыдущем примере: это просто имя класса, предваренное точкой, без указания какого-либо имени элемента. Если же требуется выбрать все элементы с одинаковым именем класса, то можно без вреда опускать в селекторе класса универсальный селектор.

Множественные классы

В предыдущем разделе мы рассматривали значения атрибута `class`, состоящие из одного слова. В HTML значением `class` может быть и разделенный пробелами список слов. Например, если вы хотите обозначить конкретный элемент и как важное сообщение, и как предупреждение, можно было бы написать:

```
<p class="urgent warning">При работе с плутонием необходимо предпринять меры
по недопущению достижения критической массы.</p>
<p>При работе с плутонием <span class="warning">существует реальная
возможность взрыва, которую надо избежать любой ценой</span>.
Это может быть достигнуто путем разделения масс.</p>
```

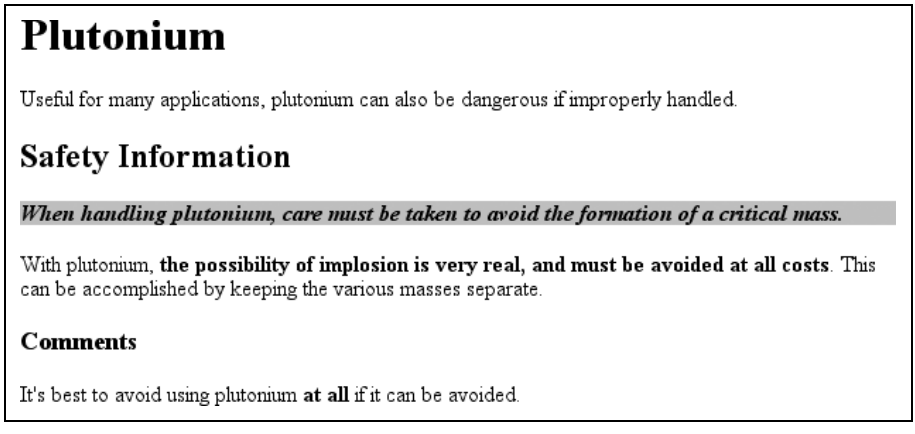


Рис. 2.9. *Выбор элементов с множественными именами классов*

Порядок слов на самом деле не имеет значения; подошло бы и `warning urgent`.

Теперь, скажем, вы хотите, чтобы все элементы, атрибут `class` которых имеет значение `warning`, были выделены полужирным шрифтом, те элементы, атрибут `class` которых имеет значение `urgent`, были выделены курсивом, а элементы, имеющие оба значения, получили серебряный фон. Это могло бы быть написано следующим образом:

```
.warning {font-weight: bold;}
.urgent {font-style: italic;}
.warning.urgent {background: silver;}
```

Объединяя два селектора класса, можно выбрать только те элементы, которые имеют оба имени класса, стоящие в любом порядке. Как видите, исходный код HTML содержит `class="urgent warning"`, но CSS-селектор записан так: `.warning.urgent`. Несмотря на это, согласно правилу абзац «При работе с плутонием...» будет расположен на серебряном фоне, как показано на рис. 2.9.

Если множественный селектор класса содержит имя, не входящее в разделенный пробелами список, то сопоставления не произойдет. Рассмотрим следующее правило:

```
p.warning.help {background: red;}
```

Как и можно было ожидать, селектор будет выбирать только те элементы `p`, атрибут `class` которых содержит слова `warning` и `help`. Следовательно, он не будет работать с элементами `p`, в атрибут `class` которых входят только слова `warning` или `urgent`. Однако он выберет такой элемент:

```
<p class="urgent warning help">Помогите!</p>
```

Internet Explorer до версии IE7 для обеих платформ не всегда правильно обрабатывает множественные селекторы классов. В более старых версиях вполне возможно выбирать из списка одно имя класса, но выбор на основании нескольких имен выполняется неправильно. Таким образом, `p.warning` работает, как и ожидается, а вот `p.warning.help` выбирает только те элементы `p`, в атрибуте `class` которых имеется слово `help`, потому что оно идет последним в селекторе. Если бы вы написали `p.help.warning`, в более старой версии Internet Explorer выбирались бы все элементы `p` с атрибутом `class`, имеющим значение `warning`, независимо от присутствия `help` в этом значении.

Селекторы идентификаторов

В некотором смысле селекторы идентификаторов аналогичны селекторам классов, но есть несколько существенных отличий. Во-первых, перед селекторами идентификаторов вместо точки ставится «решетка» (`#`), называемая также знаком фунта, диезом и даже полем для игры в крестики-нолики. Таким образом, возможно и такое правило:

```
*#first-para {font-weight: bold;}
```

Оно устанавливает полужирный шрифт для любого элемента, атрибут `id` которого имеет значение `first-para`.

Второе отличие – вместо значений атрибута `class` в селекторах идентификаторов, что и не удивительно, используются значения атрибутов `id`. Вот пример селектора идентификатора в действии:

```
*#lead-para {font-weight: bold;}

<p id="lead-para">Этот параграф будет выделен полужирным шрифтом.</p>
<p> Этот параграф НЕ будет выделен полужирным шрифтом.</p>
```

Обратите внимание, что значение `lead-para` можно было бы присвоить любому элементу документа. В этом конкретном случае оно присваивается первому абзацу, но его можно легко применить и ко второму, и к третьему абзацу.

Как и в селекторах класса, в селекторе идентификатора можно опустить универсальный селектор. В предыдущем примере можно было бы написать:

```
#lead-para {font-weight: bold;}
```

Результат не изменился бы.

Выбор между селектором классов и селектором идентификаторов

Как было показано ранее, назначать классы можно любому количеству элементов; имя класса `warning` было применено и к элементу `p`, и к элементу `span` и могло бы применяться к намного большему количеству элементов. С другой стороны, идентификаторы в HTML-документе ис-

пользуются один и только один раз. Поэтому если в документе есть элемент, значение атрибута `id` которого равно `lead-para`, ни один другой элемент этого документа не может иметь `id` со значением `lead-para`.

Броузеры не всегда проверяют уникальность идентификаторов в HTML-документе, а это значит, что если поместить в него несколько элементов с одинаковым значением атрибутов `id`, то, по всей вероятности, к ним всем будут применены одинаковые стили. Это неправильно, но такое иногда случается. Наличие одинаковых идентификаторов в документе усложняет написание DOM-сценариев, поскольку такие функции, как `getElementById()`, полагают, что в документе существует единственный элемент с заданным значением идентификатора.

В отличие от селекторов класса, селекторы идентификаторов не могут объединяться, поскольку в атрибуты `id` нельзя помещать разделенные пробелами списки.

На синтаксическом уровне работоспособность точечной нотации (например, `.warning`) в XML-документах не гарантируется. Точку можно применять в HTML, SVG и MathML; возможно, ее применение будет разрешено в будущих языках, но это решается на уровне спецификации каждого языка. Нотация идентификатора со знаком фунта (например, `#lead`) будет работать в любом языке документов, где предусмотрен атрибут, от которого требуется уникальность значений в рамках документа. Уникальность значений может требоваться для атрибута с именем `id` или с каким-нибудь другим – главное, чтобы в языке для значений этого атрибута требовалась уникальность в рамках документа.

Еще одно отличие между именами `class` и `id` состоит в том, что идентификаторы имеют больший вес, когда определяется, какие стили должны применяться к данному элементу. Более подробно это рассмотрено в следующей главе.

Подобно классам, идентификаторы могут быть выбраны независимо от элемента. Возможны такие ситуации, когда заранее известно, что в документе появится определенное значение идентификатора, но в каком элементе это произойдет, неизвестно (как в предупреждениях по обработке плутония), поэтому надо объявить независимый селектор идентификатора. Например, известно, что в любом данном документе будет элемент, значение идентификатора которого равно `mostImportant`. Но будет ли эта часть абзацем, короткой фразой, пунктом списка или заголовком раздела, неизвестно. Вы только знаете, что она будет появляться в произвольном элементе каждого документа и встречаться не чаще одного раза. Правило должно быть таким:

```
#mostImportant {color: red; background: yellow;}
```

Оно выбирает любой из следующих элементов (которые, как я отмечал ранее, *не должны* появляться вместе в одном и том же документе, потому что у них одинаковые идентификаторы):

```
<h1 id="mostImportant">Это важно!</h1>
<em id="mostImportant">Это важно!</em>
<ul id="mostImportant">Это важно!</ul>
```

Также заметьте, что в зависимости от языка документа селекторы идентификаторов могут быть чувствительными к регистру. Языки HTML и XHTML определяют имена классов и идентификаторы как чувствительные к регистру, поэтому использование в значениях класса и идентификаторах заглавных букв должно соответствовать тому, что находится в ваших документах. Таким образом, в следующем объединении CSS и HTML элемент не будет выделен полужирным шрифтом:

```
p.criticalInfo {font-weight: bold;}

<p class="criticalinfo">Не смотрите вниз.</p>
```

Из-за несовпадения регистра буквы «I» селектор не выберет данный элемент.

Некоторые старые броузеры рассматривают имена классов и идентификаторы без анализа регистра написания, но во всех современных броузерах чувствительность к регистру реализована.

Селекторы атрибутов

И в селекторах классов, и в селекторах идентификаторов речь на самом деле идет о выборе значений атрибутов. Синтаксис двух предыдущих разделов подходит (на момент написания данной книги) для документов HTML, SVG и MathML. В других языках разметки эти селекторы классов и идентификаторов могут отсутствовать. Для разрешения этой ситуации в CSS2 были введены *селекторы атрибутов* (*attribute selectors*), которые могут применяться для выбора элементов на основании их атрибутов и значений этих атрибутов. Существует четыре типа селекторов атрибутов.

Селекторы атрибутов поддерживаются броузерами Safari, Opera и всеми Gecko-броузерами, но не поддерживаются Internet Explorer вплоть до IE5/Mac и IE6/Windows. IE7 полностью поддерживает все селекторы атрибутов CSS2.1, а также некоторые селекторы атрибутов CSS3, рассматриваемые в данном разделе.

Простой выбор атрибутов

Для того чтобы выбрать элементы с определенным атрибутом независимо от значения этого атрибута, можно обратиться к простому селектору атрибутов. Например, чтобы выбрать все элементы `h1`, имеющие атрибут `class` с любым значением, и сделать их текст серебряным, напишите:

```
h1[class] {color: silver;}
```

Итак, следующая разметка:

```
<h1 class="hoopla">Hello</h1>
<h1 class="severe">Serenity</h1>
<h1 class="fancy">Fooling</h1>
```

даст результат, показанный на рис. 2.10.

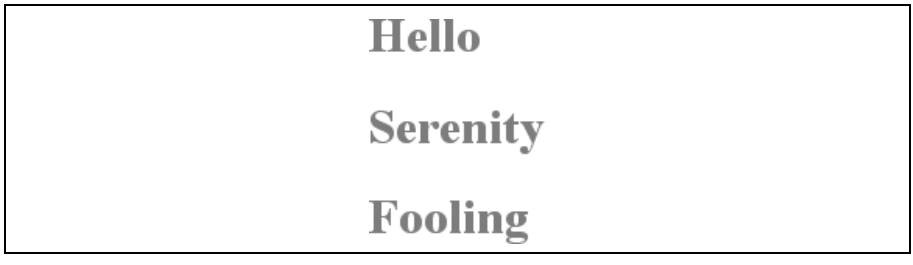


***Рис. 2.10.** Выбор элементов по атрибутам*

Эта тактика очень полезна в XML-документах, поскольку в языках XML имеется тенденция присваивать элементам и атрибутам имена, соответствующие их назначению. Рассмотрим язык XML, применяемый для описания планет Солнечной системы (назовем его PlanetML). Для того чтобы выбрать все элементы `planet` с атрибутом `moons` (луны) и выделить их полужирным шрифтом, обращая внимание на любую планету, имеющую естественные спутники, можно написать:

```
planet[moons] {font-weight: bold;}
```

Это приведет к тому, что в следующем фрагменте разметки текст второго и третьего элементов, но не первого, будет выделен полужирным шрифтом:

```
<planet>Venus</planet>
<planet moons="1">Earth</planet>
<planet moons="2">Mars</planet>
```

В HTML-документах этот подход может быть реализован творчески. Например, можно выделить все изображения, имеющие атрибут `alt`, отметив таким образом изображения, оформленные правильно:

```
img[alt] {border: 3px solid red;}
```

Этот конкретный пример полезен больше для диагностики, т. е. для определения правильности оформления изображений, чем для целей разработки.

Для того чтобы выделить полужирным шрифтом элементы, содержащие какую-либо информацию в атрибуте `title`, которую большинство броузеров отображает как «всплывающую подсказку» при прохождении курсора над элементом, можно написать:

```
*[title] {font-weight: bold;}
```

Аналогичным образом можно оформить только те элементы a, в которых есть атрибут href.

Также можно осуществлять выбор на основании наличия нескольких атрибутов. Это делается путем простого объединения селекторов атрибутов. Например, чтобы выделить полужирным шрифтом текст любой гиперссылки HTML, которая имеет и атрибут href, и атрибут title, можно написать:

```
a[href][title] {font-weight: bold;}
```

При этом полужирным шрифтом будет выделена первая ссылка следующей разметки, но не вторая или третья:

```
<a href="http://www.w3.org/" title="W3C Home">W3C</a><br />
<a href="http://www.webstandards.org">Standards Info</a><br />
<a title="Not a link">dead.letter</a>
```

Выбор на основании конкретного значения атрибута

В дополнение к выбору элементов по атрибутам можно еще более сузить выбор, чтобы охватить только те элементы, атрибуты которых имеют определенное значение. Например, пусть мы хотим выделить полужирным шрифтом гиперссылку, указывающую на определенный документ веб-сервера. Это могло бы выглядеть примерно так:

```
a[href="http://www.css-discuss.org/about.html"] {font-weight: bold;}
```

Для любого элемента может быть определено любое сочетание атрибута и значения. Однако если эта конкретная комбинация не встречается в документе, селектор не выберет ничего. И опять же языки XML могут выиграть от применения этого подхода для оформления. Вернемся к нашему примеру PlanetML. Предположим, требуется выбрать только те элементы planet, значение атрибута moons которых равно 1:

```
planet[moons="1"] {font-weight: bold;}
```

При этом полужирным шрифтом будет выделен текст второго элемента следующего фрагмента разметки, но не первого или третьего:

```
<planet>Venus</planet>
<planet moons="1">Earth</planet>
<planet moons="2">Mars</planet>
```

Как и в случае с выбором атрибутов, можно объединить несколько селекторов, чтобы выбрать один документ. Например, чтобы удвоить размер текста любой гиперссылки HTML, у которой атрибут href имеет значение http://www.w3.org/ и атрибут title имеет значение W3C Home, напишите:

```
a[href="http://www.w3.org/"][title="W3C Home"] {font-size: 200%;}
```

При этом в приведенной ниже разметке будет увеличен размер текста первой ссылки, но не второй или третьей:

```
<a href="http://www.w3.org/" title="W3C Home">W3C</a><br />
<a href="http://www.webstandards.org"
  title="Web Standards Organization">Standards Info</a><br />
<a href="http://www.example.org/" title="W3C Home">dead.link</a>
```

Результат показан на рис. 2.11.

***Рис. 2.11.** Выбор элементов на основании атрибутов и их значений*

Обратите внимание, что этот формат требует точного совпадения значения атрибута. А это требование становится источником затруднений, когда неожиданно встречаются значения, которые могут, в свою очередь, содержать разделенный пробелами список значений (как атрибут `class` в HTML). Например, рассмотрим следующий фрагмент разметки:

```
<planet type="barren rocky">Mercury</planet>
```

Единственный способ выбрать этот элемент по точному значению его атрибута – записать:

```
planet[type="barren rocky"] {font-weight: bold;}
```

Если бы мы написали `planet[type="barren"]`, правило не нашло бы соответствия с примером разметки и поэтому дало бы сбой. Это справедливо и для атрибута `class` в HTML. Рассмотрим следующее:

```
<p class="urgent warning">При работе с плутонием необходимо предпринять меры
по недопущению достижения критической массы.</p>
```

Чтобы выбрать этот элемент по конкретному значению его атрибута, надо было бы написать:

```
p[class="urgent warning"] {font-weight: bold;}
```

Это *не* эквивалент точечной нотации классов, как будет рассмотрено в следующем разделе. Данное выражение выбирает любой элемент `p`, значение атрибута `class` которого *именно* `urgent warning` – со строгим соблюдением порядка слов и одним пробелом между ними. По сути, это строгое сравнение строк.

Также помните, что селекторы идентификаторов и селекторы атрибутов, нацеленные на атрибут `id`, – это не совсем одно и то же. Иначе говоря, существует едва различимое, но важное отличие между `h1#page-title` и `h1[id="page-title"]`. Это отличие обсуждается в следующей главе.

Выбор по частичному значению атрибута

Любой атрибут, допускающий наличие разделенного пробелами списка слов, допускает и выбор на основании любого из этих слов. Классический пример в HTML – атрибут `class`, способный принимать в качестве значения несколько слов. Рассмотрим наш обычный пример текста:

```
<p class="urgent warning">При работе с плутонием необходимо предпринять меры
по недопущению достижения критической массы.</p>
```

Скажем, требуется выбрать элементы, атрибут `class` которых содержит слово `warning`. Это можно сделать посредством селектора атрибутов:

```
p[class~="warning"] {font-weight: bold;}
```

Обратите внимание на наличие в селекторе тильды (~). Это ключ для осуществления выбора на основании наличия в значении атрибута отделенного пробелами слова. Если пропустить тильду, то получится требование точного соответствия конкретному значению, рассмотренное в предыдущем разделе.

Эта конструкция селектора эквивалентна рассмотренному ранее условному обозначению классов с помощью предшествующей точки. Таким образом, записи `p.warning` и `p[class~="warning"]` эквивалентны в случае применения к HTML-документам. Вот пример, который является HTML-версией представленной ранее разметки «PlanetML»:

```
<span class="barren rocky">Mercury</span >
<span class=" cloudy barren">Venus</span >
<span class="life-bearing cloudy">Earth</span >>
```

Чтобы выделить курсивом все элементы, содержащие слово `barren` в атрибуте `class`, напишите:

```
span[class~="barren"] {font-style: italic;}
```

Селектор этого правила выберет первые два элемента примера разметки и соответственно выделит их текст курсивом, как показано на рис. 2.12. Такой же результат был бы получен при записи `span.barren {font-style: italic;}`.

Так зачем же возиться с этой формой селектора атрибутов (тильда-равно) в HTML? Потому что она может использоваться для любого атрибута, не только `class`. Пусть, например, есть документ, содержащий ряд изображений, причем лишь некоторые из них являются рисунками. В этом случае для выбора только этих рисунков можно применить селектор атрибута по частичному значению, настроенный на текст атрибута `title`:

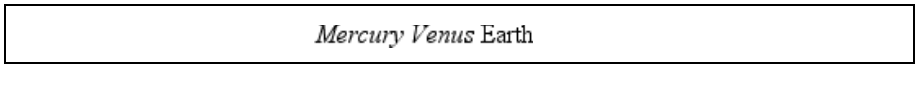


***Рис. 2.12.** Выбор элементов на основании частей значений атрибутов*

```
img[title~="Рисунок"] {border: 1px solid gray;}
```

Это правило будет соответствовать любому изображению, текст атрибута `title` которого содержит слово `Рисунок`. Следовательно, поскольку текст атрибута `title` всех рисунков выглядит примерно так: «Рисунок 4. Лысый старейшина», правило будет соответствовать этим изображениям. Уж если на то пошло, селектор `img[title~="Рисунок"]` также будет выбирать атрибут `title`, имеющий значение «Рисунок для художника – то же, что движение для оратора...». Любое изображение, не имеющее атрибута `title`, или значение атрибута `title` которого не содержит слова «Рисунок», не будет выбрано.

Более расширенный модуль CSS Selectors, который был выпущен значительно позже CSS2, содержит еще несколько селекторов атрибутов по частичному значению (или, как они названы в спецификации, «селекторов атрибутов по подстроке»). Поскольку они поддерживаются многими современными броузерами, включая IE7, в табл. 2.1 приводим их краткое описание.

Таблица 2.1. Селекторы атрибутов по подстроке

Тип	Описание
`[foo^="bar"]`	Выбирает любой элемент, значение атрибута `foo` которого начинается с `"bar"`.
`[foo$="bar"]`	Выбирает любой элемент, значение атрибута `foo` которого заканчивается `"bar"`.
`[foo*="bar"]`	Выбирает любой элемент, значение атрибута `foo` которого содержит подстроку `"bar"`.

Таким образом, исходя из следующих правил и разметки получаем результаты, представленные на рис. 2.13.

```
span[class*="cloud"] {font-style: italic;}
span[class^="bar"] {background: silver;}
span[class$="y"] {font-weight: bold;}
```

```
<span class="barren rocky">Mercury</span>
<span class="cloudy barren">Venus</span>
<span class="life-bearing cloudy">Earth</span>
```

Первое из трех правил соответствует любому элементу `span`, атрибут `class` которого содержит подстроку `cloud`; таким образом выбираются обе планеты, в описании которых содержится слово «cloudy». По второму правилу выбирается любой элемент `span`, атрибут `class` которого начинается с подстроки `bar`. Сюда подходит только Меркурий (Mer-

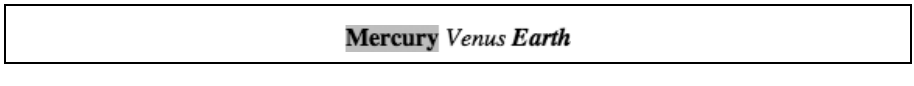


***Рис. 2.13.** Выбор элементов по подстрокам значений атрибутов*

cury), атрибут `class` которого имеет значение `barren rocky`. Венера (Venus) не включена, потому что подстрока `bar`, входящая в слово `barren`, располагается не в начале значения атрибута `class`. Наконец, третье правило соответствует любому элементу `span`, атрибут `class` которого заканчивается подстрокой `y`, поэтому выбираются Меркурий (Mercury) и Земля (Earth). Венера (Venus) опять осталась в стороне, поскольку значение ее атрибута `class` не заканчивается на `y`.

Как и следовало ожидать, существует множество вариантов полезного применения таких селекторов. В качестве примера предположим, что требуется применить особый стиль ко всем ссылкам на веб-сайт O'Reilly Media. Вместо того чтобы описывать для них всех атрибут `class` и создавать стили для этого класса, можно просто записать следующее правило:

```
a[href*="oreilly.com"] {font-weight: bold;}
```

И конечно же, мы не ограничены лишь атрибутами `class` и `href`. Можно работать с любыми атрибутами: `title`, `alt`, `src`, `id`... все, что угодно. Стили могут применяться на основании значения атрибута или частей значения. Следующее правило описывает любой рисунок, используемый для формирования таблицы (плюс любое другое изображение, в URL которого есть строка «space» (пробел)):

```
img[src*="space"] {border: 5px solid red;}
```

На момент написания данной книги селекторы по подстроке поддерживаются броузерами Safari, Gecko-броузерами, Opera и IE7/Win.

Выбор конкретного атрибута

Последний тип селекторов атрибутов, *селектор конкретного атрибута* (*particular attribute selector*), проще показать, чем описать. Рассмотрим следующее правило:

```
*[lang|="en"] {color: white;}
```

Это правило будет выбирать любой элемент, чей атрибут `lang` эквивалентен `en` или начинается с `en-`. Поэтому первые три элемента следующего примера разметки будут выбраны, а последние два – нет:

```
<h1 lang="en">Hello!</h1>
<p lang="en-us">Greetings!</p>
<div lang="en-au">G'day!</div>
<p lang="fr">Bonjour!</p>
<h4 lang="cy-en">Jrooana!</h4>
```

В общем, форма `[атрибут|="значение"]` может применяться для любого атрибута и его значений. Скажем, в HTML-документе есть наборы рисунков, каждый из которых хранится в файле с именем, например *figure-1.gif* или *figure-3.jpg*. Все эти изображения можно выбрать посредством следующего селектора:

```
img[src|="figure"] {border: 1px solid gray;}
```

Чаще всего этот тип селекторов атрибутов применяется для сопоставления значений языков, как будет показано в этой главе позже.

Использование структуры документа

Как я упоминал ранее, мощь CSS обусловлена тем, что для определения соответствующих стилей и их применения они используют структуру HTML-документов. Но это еще не все, так как подразумевается, что подобные определения – единственный вариант использования структуры документа в CSS. Структура играет намного большую роль в механизме применения стилей к документу. Итак, прежде чем перейти к более мощным формам выбора, немного остановимся на структуре.

Отношения родитель–потомок

Чтобы понять взаимоотношения между селекторами и документами, надо еще раз вспомнить, как структурирован документ. Рассмотрим очень простой HTML-документ:

```
<html>
<head>
 <base href="http://www.meerkat.web/">
 <title>Meerkat Central</title>
</head>
<body>
 <h1>Meerkat <em>Central</em></h1>
 <p>
 Welcome to Meerkat <em>Central</em>, the <strong>best meerkat web site
 on <a href="inet.html">the <em>entire</em> Internet</a></strong>!</p>
 <ul>
  <li>We offer:
   <ul>
    <li><strong>Detailed information</strong> on how to adopt a meerkat</li>
    <li>Tips for living with a meerkat</li>
    <li><em>Fun</em> things to do with a meerkat, including:
     <ol>
      <li>Playing fetch</li>
      <li>Digging for food</li>
      <li>Hide and seek</li>
     </ol>
    </li>
   </ul>
  </li>
  <li>...and so much more!</li>
 </ul>
 <p>
 Questions? <a href="mailto:suricate@meerkat.web">Contact us!</a>
 </p>
```

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

Мощь CSS в основном базируется на *родительско-дочерних отношениях* (*parent–child relationship*) элементов. HTML-документы (фактически самые структурированные документы из всех) строятся на основании иерархии элементов, которую можно показать в форме древовидного представления документа (рис. 2.14). В этой иерархии каждый элемент тем или иным образом вписывается в общую структуру документа. Каждый элемент является или *родительским* (*parent*), или *дочерним* (*child*) элементом другого элемента, а зачастую выполняет обе эти роли.

Говорят, что элемент является родителем другого элемента, если в иерархии документа он находится прямо над этим элементом. Например, на рис. 2.14 первый элемент `p` является родителем для элементов `em` и `strong`, тогда как `strong` – родитель элемента `a`, который в свою очередь является родителем другого элемента `em`. И наоборот, элемент является дочерним элементом другого элемента, если он находится прямо под этим элементом. Таким образом, элемент `a` – это дочерний элемент элемента `strong`, который в свою очередь является потомком элемента `p`, и т. д. (рис. 2.14).

Термины родительский и дочерний элементы – это частные случаи терминов *предок* (*ancestor*) и *потомок* (*descendant*). Между ними существует разница: если в представлении в виде древовидного списка элемент находится ровно на один уровень выше другого, между ними существуют родительско-дочерние отношения. Если путь от одного элемента к другому пересекает два или более уровней, между элементами существуют отношения предок-потомок, но не родительско-дочерние. (Конечно, дочерний элемент также является потомком, а родитель – предком.) На рис. 2.14 первый элемент `ul` – это родитель двух

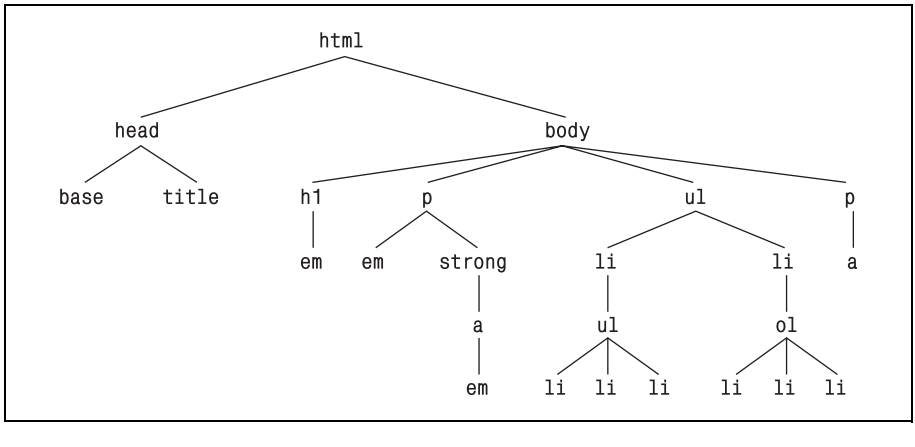


Рис. 2.14. *Древовидная структура документа*

элементов li, но первый ul также является предком всех элементов, происходящих от его элемента li, вплоть до самых глубоко вложенных элементов li.

Также на рис. 2.14 показан элемент a, который является дочерним по отношению к strong, но также и потомком абзаца, элементов body и html. Элемент body – предок всего, что броузер будет отображать по умолчанию, а элемент html – предок всего документа. Поэтому элемент html также называют *корневым элементом* (*root element*).

Селекторы потомков

Первое преимущество понимания этой модели – возможность определять *селекторы потомков* (*descendant selectors*), также известные как *контекстные селекторы* (*contextual selectors*). Определение селекторов потомков – это создание правил, действующих лишь в рамках заданной структуры. Например, требуется задать стили только для тех элементов em, которые происходят от элементов h1. Можно задать атрибут class для каждого элемента em, находящегося в h1, но эта процедура практически настолько же длительная, как и применение тега font. Очевидно, намного эффективнее объявить правила, которые соответствуют только элементам em, находящимся внутри элементов h1.

Для этого напишите следующее:

```
h1 em {color: gray;}
```

Это правило сделает серым любой текст элемента em, который является потомком элемента h1. Текст других элементов em, например находящихся в абзаце или блоке, не будет выбран этим правилом. Это видно из рис. 2.15.

В селекторе потомков часть правила, соответствующая селектору, состоит из двух или более разделенных пробелами селекторов. Пробел между селекторами – это пример *комбинатора* (*combinator*). Каждый комбинатор-пробел, если читать его справа налево, может быть истолкован как «находящийся в», «который представляет собой часть» или «являющийся потомком». Таким образом, h1 em можно перевести как: «любой элемент em, который является потомком элемента h1». (Если прочитать селектор слева направо, может получиться примерно следующее: «для любого h1, содержащего em, к em будут применены следующие стили».)

Конечно, два селектора не предел. Например:

```
ul ol ul em {color: gray;}
```

***Рис. 2.15**. Выбор элемента на основании его контекста*

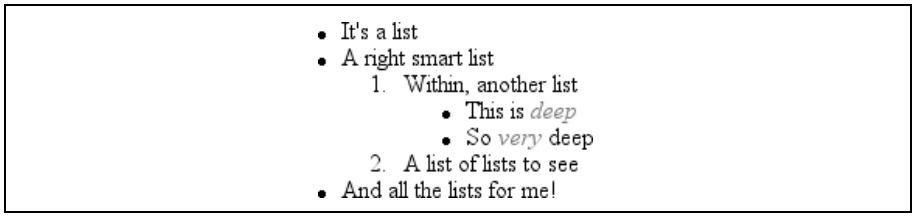


Рис. 2.16. *Очень специфичный селектор потомков*

В этом случае, как показывает рис. 2.16, серым будет любой выделенный текст, представляющий собою часть ненумерованного списка, который является частью нумерованного списка, который, в свою очередь, есть часть ненумерованного списка (да, это не ошибка). Очевидно, что это очень специфичный критерий выбора.

Селекторы потомков могут обладать исключительной силой. Они делают возможным то, что никогда не могло бы быть осуществлено в HTML, по крайней мере, без громадного количества тегов `font`. Рассмотрим простой пример. Предположим, имеется документ, содержащий врезку и основную область. У врезки – голубой фон, а у основной области – белый, и обе области включают списки ссылок. Нельзя сделать все ссылки синими, потому что тогда невозможно будет прочитать их во врезке.

Селекторы потомков позволяют обойти это препятствие. В данном случае ячейке таблицы, содержащей вашу врезку, назначается класс `sidebar`, а основной области – класс `main`. Затем стили записываются так:

```
td.sidebar {background: blue;}
td.main {background: white;}
td.sidebar a:link {color: white;}
td.main a:link {color: blue;}
```

На рис. 2.17 показан результат.

`:link` обозначает ссылки, указывающие на те ресурсы, которые не посещались. Подробнее мы поговорим об этом в данной главе несколько позже.

Рис. 2.17. *Селекторы потомков позволяют применить разные стили к элементам одного типа*

Вот другой пример: необходимо, чтобы текст любого элемента b (полужирное начертание), который является частью blockquote, был серым, и также был серым любой выделенный полужирным шрифтом текст, находящийся в обычном абзаце:

```
blockquote b, p b {color: gray;}
```

В результате текст элементов b, являющихся потомками абзацев или элементов blockquote, будет серым.

Еще одно не упоминавшееся свойство селекторов потомков – степень удаления двух элементов может быть практически бесконечной. Например, если написать ul em, то будет выбран любой элемент em, происходящий от элемента ul, и насколько глубоко вложен em, значения не имеет. Таким образом, в следующей разметке ul em выбирает элемент em:

```
<ul>
<li>Элемент списка 1
<ol>
<li>Элемент списка 1-1</li>
<li>Элемент списка 1-2</li>
<li>Элемент списка 1-3
<ol>
<li>Элемент списка 1-3-1</li>
<li>Элемент списка <em>1-3-2</em></li>
<li>Элемент списка 1-3-3</li>
</ol></li>
<li>Элемент списка 1-4</li>
</ol></li>
</ul>
```

Выбор дочерних элементов

В некоторых случаях не требуется выбирать любой элемент-потомок; напротив, необходимо сузить диапазон выбора до дочернего элемента другого элемента. Предположим, надо выбрать элемент strong, только если он является дочерним элементом (а не просто потомком) элемента h1. Для этого используется символ-комбинатор селектора дочерних элементов, которым является символ «больше» (>):

```
h1 > strong {color: red;}
```

Это правило сделает красным элемент strong для первого из следующих далее h1 и не сделает для второго:

```
<h1>Это <strong>очень</strong> важно.</h1>
<h1> Это <em>действительно <strong>очень</strong></em> важно.</h1>
```

Прочитанный справа налево селектор h1 > strong переводится как «выбираем любой элемент strong, являющийся дочерним элементом h1». Комбинатор селектора дочерних элементов может быть окружен пробелами. Таким образом, селекторы h1 > strong, h1> strong и h1>strong эк-

вивалентны. Добавлять или опускать пробелы можно по собственному усмотрению.

При просмотре документа в виде древовидной структуры легко заметить, что селектор дочерних элементов ограничивает свои сопоставления только непосредственно соединенными друг с другом элементами. На рис. 2.18 показана часть дерева документа.

На этом фрагменте дерева можно без труда выделить родительско-дочерние отношения. Например, элемент `a` – родитель элемента `strong`, он же и дочерний элемент элемента `p`. Можно было бы сопоставить элементы этого фрагмента с селекторами `p > a` и `a > strong`, но не с селектором `p > strong`, поскольку `strong` является потомком `p`, а не дочерним элементом.

Можно также составлять в одном и том же селекторе комбинации потомков и дочерних элементов. Так, селектор `table.summary td > p` будет выбирать любой элемент `p`, являющийся дочерним элементом `td`, происходящего от элемента `table`, с атрибутом `class`, значение которого равно `summary`.

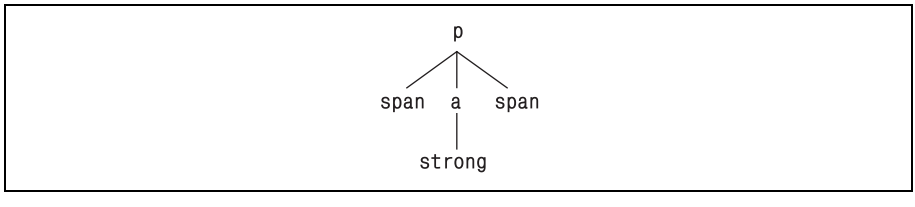


***Рис. 2.18.** Фрагмент дерева документа*

Выбор сестринских элементов

Скажем, требуется оформить абзац, находящийся в дереве документа на одном уровне с заголовком, или создать специальное обрамление для списка, стоящего на одном уровне с абзацем. Чтобы выбрать элемент, расположенный на одном уровне с другим элементом и имеющий того же родителя, применяется *комбинатор селектора сестринских элементов* (*adjacent-sibling combinator*), представляемый в виде знака плюс (+). Как и комбинатор селектора дочерних элементов, этот символ может быть окружен пробелами по усмотрению автора.

Чтобы удалить верхний отступ абзаца, непосредственно следующего за элементом `h1`, запишем:

```
h1 + p {margin-top: 0;}
```

Этот селектор означает следующее: «выбираем любой абзац, расположенный непосредственно за элементом `h1`, имеющим общих родителей с элементом `p`».

Наглядно представить себе, как работает этот селектор, проще всего, еще раз рассмотрев фрагмент дерева документа (рис. 2.19).

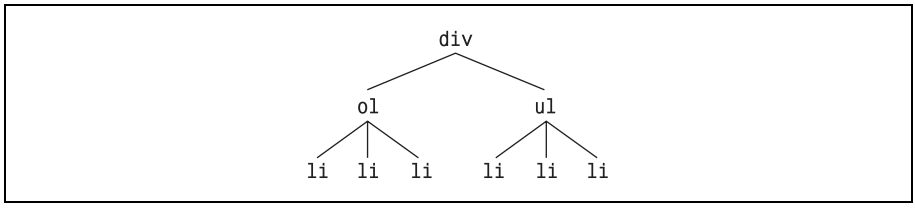


Рис. 2.19. *Еще один фрагмент дерева документа*

В этом фрагменте от элемента `div` происходит пара списков, один нумерованный, а другой нет, каждый из которых содержит по три элемента списка. Списки представляют собой сестринские элементы, и сами элементы списков – тоже сестринские элементы. Однако элементы первого списка не являются сестринскими для элементов второго списка, поскольку их родительские элементы разные. (В лучшем случае они являются кузинами.)

Помните, что из двух сестринских элементов одним комбинатором выбирается только второй элемент. Поэтому если вы записываете `li + li {font-weight: bold;}`, полужирным шрифтом будут выделены только второй и третий элементы каждого списка. Первые элементы списков останутся нетронутыми, как показано на рис. 2.20.

Для обеспечения правильной работы CSS требует, чтобы два элемента были приведены в «исходном порядке». В нашем примере за элементом `ol` непосредственно следует элемент `ul`. Это позволило бы выбрать второй элемент с помощью селектора `ol + ul`, но первый элемент посредством аналогичного синтаксиса выбрать не удастся. Чтобы можно было выбрать `ul + ol`, нумерованный список должен следовать непосредственно за ненумерованным списком.

Следует добавить, что текст между двумя элементами не мешает работать комбинатору селектора сестринских элементов. Рассмотрим следующий фрагмент разметки, представление которого в виде древовидного списка совпадает с приведенным на рис. 2.19:

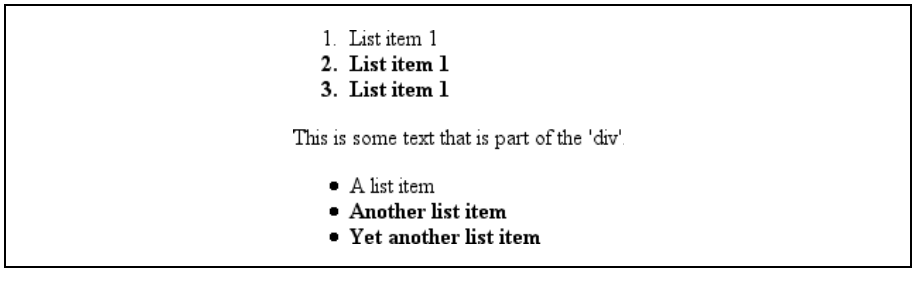


Рис. 2.20. *Выбор сестринских элементов*

```
<div>
<ol>
<li>Элемент списка 1</li>
<li>Элемент списка 1</li>
<li>Элемент списка 1</li>
</ol>
Это текст, который является частью элемента 'div'.
<ul>
<li>Элемент списка</li>
<li>Другой элемент списка</li>
<li>Еще один элемент списка</li>
</ul>
</div>
```

Даже несмотря на то, что между двумя списками вставлен текст, попрежнему можно сопоставлять второй список с селектором `ol + ul`. Дело в том, что вклинившийся текст не стоит в одном блоке с сестринским элементом, а является частью родительского элемента `div`. Если бы вы включили этот текст в абзац, это бы помешало сопоставлению селектора `ol + ul` со вторым списком. Вместо этого пришлось бы написать приблизительно так: `ol + p + ul`.

Как показывает следующий пример, комбинатор селектора сестринских элементов может применяться в сочетании с другими комбинаторами:

```
html > body table + ul{margin-top: 1.5em;}
```

Этот селектор надо понимать так: «выбираем любой элемент `ul`, расположенный непосредственно за сестринским элементом `table`, являющимся потомком элемента `body`, который сам представляет собою дочерний элемент элемента `html`».

Internet Explorer для Windows вплоть до версии IE6 не поддерживает селекторы дочерних и сестринских элементов. IE7 поддерживает оба типа селекторов.

Псевдоклассы и псевдоэлементы

Но действительно интересным все делают *селекторы псевдоклассов* (*pseudo-class selectors*) и *селекторы псевдоэлементов* (*pseudo-element selectors*). Они позволяют назначать стили структурам, существование которых в документе необязательно, или фантомным классам, наличие которых зависит от состояния элемента или даже от состояния документа в целом. Иначе говоря, стили применяются к частям документа не на основании информации, содержащейся в структуре документа, и способом, который не может быть определен путем простого изучения разметки документа.

Может показаться, будто я применяю стили случайным образом, но это не так. Напротив, я применяю стили в соответствии с выполнением некоторых условий, возникновение которых я не могу предсказать заранее. Однако обстоятельства, при которых будут появляться стили, в действительности строго определены. Проведем аналогию: во время соревнований всякий раз, когда команда-хозяин выигрывает очко, толпа приветствует это одобрительными возгласами. Неизвестно точно, когда в ходе игры это произойдет, но если это случится, толпа будет восклицать, как и предполагалось. Тот факт, что нельзя предсказать момент, когда это случится, не делает эффект менее ожидаемым.

Селекторы псевдоклассов

Давайте начнем с изучения селекторов псевдоклассов, поскольку они лучше поддерживаются броузерами и поэтому применяются шире.

Возьмем элемент `a`, который в HTML и XHTML устанавливает связь одного документа с другим. Конечно, эти элементы остаются сами собой, но некоторые из них ссылаются на страницы, которые уже посещались, тогда как другие указывают на страницы, на которые еще надо заглянуть. Вы не увидите разницу, просто взглянув на разметку HTML, потому что в ней все эти элементы выглядят одинаково. Единственный способ указать на то, какие ссылки посещались, – сравнить ссылки документа с хронологией броузера пользователя. Итак, на самом деле существует два основных типа ссылок: посещенные и непосещенные. Эти типы известны как *псевдоклассы* (*pseudo-classes*), и использующие их селекторы называют селекторами псевдоклассов.

Чтобы лучше понять эти классы и селекторы, посмотрим, как ведут себя броузеры по отношению к ссылкам. Согласно условным обозначениям, принятым в Mosaic, ссылки на непосещенные страницы выделены голубым цветом, а на уже посещенные – красным (в последующих броузерах, например Internet Explorer, красный цвет сменился на фиолетовый). Таким образом, если бы можно было ввести в ссылки классы, чтобы любая уже посещенная ссылка относилась бы, скажем, к классу «visited», то можно было бы написать стиль, позволяющий выделять такие ссылки красным цветом:

```
a.visited {color: red;}

<a href="http://www.w3.org/" class="visited">W3C Web site</a>
```

Однако подобный подход требует, чтобы классы ссылок менялись при каждом посещении новой страницы, что несколько глупо. Вместо этого CSS определяет псевдоклассы, заставляющие ссылки посещенных страниц вести себя так, как будто имя их класса – «visited»:

```
a:visited {color: red;}
```

Теперь любая ссылка, указывающая на посещенную страницу, будет красной, и не надо добавлять в них атрибуты `class`. Обратите внимание

на двоеточие (:), присутствующее в правиле. Двоеточие, разделяющее a и visited, – это визитная карточка псевдокласса или псевдоэлемента. Все ключевые слова псевдоклассов и псевдоэлементов начинаются с двоеточия.

Псевдоклассы ссылок

CSS2.1 определяет два псевдокласса (табл. 2.2), которые применяются только к гиперссылкам. В HTML и XHTML 1.0 и 1.1 это любые элементы a, у которых есть атрибут href; в языках XML это любые элементы, которые действуют как ссылки на другие ресурсы.

Таблица 2.2. Псевдоклассы ссылок

Имя	Описание
:link	Ссылается на любую гиперссылку (т. е. имеющую атрибут href) и указывает на адрес, который не был посещен. Заметьте, что некоторые броузеры могут неверно интерпретировать :link и ссылаться на любую гиперссылку, посещенную или непосещенную.
:visited	Ссылается на любую гиперссылку, указывающую на уже посещенный адрес.

Первый из псевдоклассов в табл. 2.1 может показаться несколько избыточным. В конце концов, ссылка либо была посещена, либо нет, правильно? В этом случае все, что должно нам понадобиться, это:

```
a {color: blue;}
a:visited {color: red;}
```

Хотя этот формат кажется приемлемым, на самом деле его не вполне достаточно. Первое из приведенных здесь правил применяется не только к непосещенным ссылкам, но и к целевым связям, таким как:

```
<a name="section4">4. Жизнеописание мангустов</a>
```

Результирующий текст будет синим, потому что элемент a будет соответствовать правилу a {color: blue;}, как показано выше. Поэтому, чтобы избежать применения стилей вашей ссылки к целевым связям, используйте псевдокласс :link:

```
a:link {color: blue;}     /* непосещенные ссылки выделены голубым */
a:visited {color: red;}   /* посещенные ссылки выделены красным */
```

Как вы, возможно, уже поняли, селекторы псевдоклассов :link и :visited в функциональном плане эквивалентны двум атрибутам body: link и vlink. Предположим, автор хочет, чтобы все ссылки на непосещенные страницы были выделены фиолетовым цветом, а ссылки на посещенные страницы были серебряными. В HTML 3.2 это можно было определить следующим образом:

```
<body link="purple" vlink="silver">
```

Рис. 2.21. *Применение нескольких стилей к посещенной ссылке*

В CSS тот же эффект достигается так:

```
a:link {color: purple;}
a:visited {color: silver;}
```

В случае с псевдоклассами CSS, конечно же, можно применить не только цвета. Скажем, вы хотите, чтобы посещенные ссылки выделялись перечеркнутым курсивом серебряного цвета, как показано на рис. 2.21.

Это делается с помощью следующих стилей:

```
a:visited {color: silver; text-decoration: line-through; font-style: italic;}
```

Наступил подходящий момент вспомнить о селекторах классов и показать, как их можно сочетать с псевдоклассами. Например, требуется изменить цвет ссылок, указывающих на страницы, расположенные вне вашего сайта. Если каждой из таких ссылок присвоить класс, реализация не вызовет затруднений:

```
<a href="http://www.mysite.net/">Моя начальная страница</a>
<a href="http://www.site.net/" class="external">Другая начальная страница</a>
```

Чтобы применить к внешней ссылке ее собственные стили, достаточно такого правила:

```
a.external:link, a.external:visited {color: maroon;}
```

Это правило сделает вторую ссылку предыдущей разметки красно-коричневой, тогда как первая ссылка будет сохранять стандартный для гиперссылок цвет (обычно синий).

Аналогичный синтаксис применяется и для селекторов идентификаторов:

```
a#footer-copyright:link{font-weight: bold;}
a#footer-copyright:visited {font-weight: normal;}
```

Хотя псевдоклассы `:link` и `:visited` очень полезны, они также статичны: обычно они не меняют стилевого оформления документа во время его просмотра. В CSS2.1 есть другие псевдоклассы, которые не настолько статичны; мы рассмотрим их следующими.

Динамические псевдоклассы

CSS2.1 определяет три псевдокласса, которые могут изменять внешний вид документа в результате действий пользователя. Эти динамические псевдоклассы традиционно применяются для оформления гиперссылок, но их возможности намного шире. Эти псевдоклассы описаны в табл. 2.3.

Таблица 2.3. Динамические псевдоклассы

Имя	Описание
:focus	Ссылается на любой элемент, которому в настоящий момент принадлежит фокус ввода, т. е. который готов принимать ввод с клавиатуры или быть активированным некоторым образом.
:hover	Ссылается на любой элемент, над которым размещен указатель некоторого устройства, например гиперссылка, по которой проводят курсором мыши.
:active	Ссылается на любой элемент, который был активирован пользователем, например гиперссылка, по которой щелкает пользователь в течение того времени, когда удерживается кнопка мыши.

Как и :link, и :visited, эти псевдоклассы лучше всего известны в контексте гиперссылок. Стили многих веб-страниц выглядят так:

```
a:link {color: navy;}
a:visited {color: gray;}
a:hover {color: red;}
a:active {color: yellow;}
```

В первых двух правилах применяются статические псевдоклассы, а в двух последних – динамические псевдоклассы. Псевдокласс :active представляет собой аналог атрибута alink в HTML 3.2, хотя, как и раньше, к активным ссылкам можно применять любые цвета и стили.

Порядок расположения псевдоклассов в селекторе важнее, чем это может показаться на первый взгляд. Обычная рекомендация – «link-visited-focus-hover-active», хотя она была изменена на «link-visited-focus-hover-active». В следующей главе объясняется, почему этот порядок важен, и обсуждается ряд причин, по которым можно принять решение изменить или даже проигнорировать рекомендуемый порядок расположения.

Обратите внимание, что динамические псевдоклассы могут применяться к любым элементам, что весьма полезно, поскольку часто бывает необходимо применить динамические стили к элементам, которые не являются ссылками. Например, такая разметка:

```
input:focus {background: silver; font-weight: bold;}
```

позволила бы выделить элемент формы, который получает вводимые с клавиатуры данные, как показано на рис. 2.22.

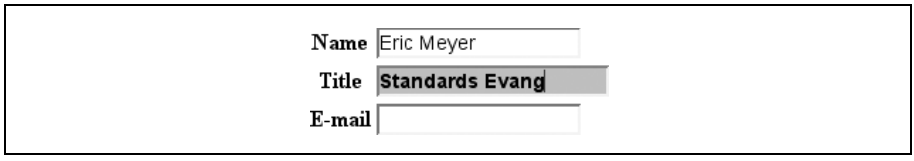


Рис. 2.22. *Выделение элемента формы, получившего фокус ввода*

Применяя динамические псевдоклассы к обычным элементам, можно осуществить и некоторые довольно необычные операции. Скажем, реализовать эффект подсветки:

```
body *:hover {background: yellow;}
```

Согласно этому правилу любой элемент, происходящий от элемента `body`, при помещении над ним указателя мыши будет приобретать желтый фон. Заголовки, абзацы, списки, таблицы, изображения и любые другие элементы, находящиеся в элементе `body`, будут изменять свой фон на желтый. Можно также изменить шрифт, заключить такой элемент в рамку или изменить все, что позволит броузер.

Internet Explorer для Windows вплоть до версии IE6 не позволяет псевдоклассам выбирать какие-либо элементы, кроме гиперссылок. В IE7 добавлена поддержка `:hover` для всех элементов, но нет стилей `:focus` для элементов формы.

Проблемы динамического применения стилей

Работая с динамическими псевдоклассами, следует учитывать ряд особенностей. Например, можно задать посещенным и непосещенным ссылкам шрифт одного размера и сделать так, чтобы при зависании над ними указателя мыши ссылки увеличивались в размере, как показано на рис. 2.23:

```
a:link, a:visited {font-size: 13px;}
a:hover {font-size: 20px;}
```

Как видите, агент пользователя увеличивает размер текста ссылки на тот промежуток времени, пока указатель не покинет ее. Агент пользова-

Рис. 2.23. *Изменение компоновки при использовании динамических псевдоклассов*

теля, который обеспечивает это требование, должен перерисовать документ, когда указатель мыши зависает над ссылкой, что может привести к переформатированию всего содержимого, следующего за ссылкой.

Однако спецификации CSS определяют, что агенты пользователя не обязаны перерисовывать документ после его первичного формирования, так что нельзя быть абсолютно уверенными, что желаемый эффект будет реализован. Я настойчиво рекомендую избегать конструкций, которые основаны на таком эффекте.

Выбор первого дочернего элемента

Еще один статический псевдокласс, `:first-child`, применяется для выбора элементов, представляющих собой первые дочерние элементы других элементов. Назначение этого псевдокласса очень легко понять неправильно, поэтому требуется развернутый пример. Рассмотрим следующую разметку:

```
<div>
<p>These are the necessary steps:</p>
<ul>
<li>Insert key</li>
<li>Turn key <strong>clockwise</strong></li>
<li>Push accelerator</li>
</ul>
<p>
Do <em>not</em> push the brake at the same time as the accelerator.
</p>
</div>
```

В данном примере первые дочерние элементы – это первый `p`, первый `li` и элементы `strong` и `em`. По следующим двум правилам:

```
p:first-child {font-weight: bold;}
li:first-child {text-transform: uppercase;}
```

получается результат, показанный на рис. 2.24.

Первое правило выделяет полужирным шрифтом любой элемент `p`, который является первым дочерним элементом другого элемента. Второе правило переводит в верхний регистр текст любого элемента `li`, являющегося первым дочерним элементом другого элемента (которыми в HTML должны быть элементы `ol` или `ul`).

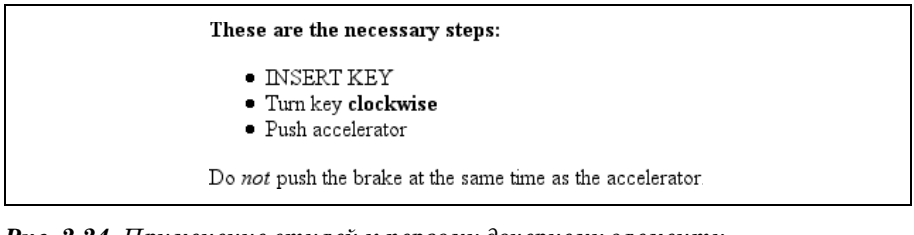


Рис. 2.24. *Применение стилей к первому дочернему элементу*

Самая распространенная ошибка – полагать, что такой селектор, как `p:first-child`, выберет первый дочерний элемент элемента `p`. Однако вспомните природу псевдоклассов, которая состоит в подключении к ассоциированному с псевдоклассом элементу фантомного класса некоторого типа. Если бы было необходимо добавить в разметку реальные классы, это выглядело бы так:

```
<div>
<p class="first-child">These are the necessary steps:</p>
<ul>
<li class="first-child">Insert key</li>
<li>Turn key <strong class="first-child">clockwise</strong></li>
<li>Push accelerator</li>
</ul>
<p>
Do <em class="first-child">not</em> push the brake at the same time
as the accelerator.
</p>
</div>
```

Следовательно, для того чтобы выбрать элементы `em`, являющиеся первыми дочерними элементами другого элемента, нужна запись `em:first-child`. Этот селектор позволяет, например, оформлять первый элемент списка, первый абзац элемента `div` или первый элемент `td` в строке таблицы.

Internet Explorer для Windows вплоть до версии IE6 не поддерживает `:first-child`, а IE7 поддерживает `:first-child`.

Выбор по языку

Если требуется выбрать элемент на основании его языка, то можно обратиться к псевдоклассу `:lang()`. С точки зрения шаблонов соответствия псевдокласс `:lang()` аналогичен селектору атрибутов `|=`. Например, чтобы выделить курсивом любой элемент на французском языке, можно написать:

```
*:lang(fr) {font-style: italic;}
```

Основное отличие между псевдоселектором и селектором атрибутов в том, что информация о языке может быть получена из нескольких источников, причем некоторые из них находятся вне самого элемента. Вот правило CSS2.1:

> В HTML язык определяется сочетанием атрибута «lang», элемента META и, возможно, информацией из протокола (такой как HTTP-заголовки). В XML применяется атрибут с именем `xml:lang`, и кроме того возможно существование других методов определения языка, зависящих от языка разметки документа.

Следовательно, псевдокласс надежнее, чем селектор атрибутов, и, вероятно, лучше применить именно его в тех случаях, когда необходимо ориентированное на конкретный язык применение специальных стилей.

Комбинирование псевдоклассов

В CSS2.1 можно комбинировать псевдоклассы в одном селекторе. Например, можно сделать так, чтобы при зависании над ними указателя мыши непосещенные ссылки становились красными, а посещенные ссылки – красно-коричневыми:

```
a:link:hover {color: red;}
a:visited:hover {color: maroon;}
```

Порядок, в котором они задаются, фактически неважен; вы могли бы написать `a:hover:link` и получить такой же результат. Можно назначить отдельные стили оформления при зависании указателя для непосещенных и посещенных ссылок, написанных на другом языке, например немецком:

```
a:link:hover:lang(de) {color: gray;}
a:visited:hover:lang(de) {color: silver;}
```

Будьте осторожны, чтобы не объединить взаимоисключающие псевдоклассы. Например, ссылка не может быть одновременно и посещенной, и непосещенной, так что `a:link:visited` не имеет никакого смысла. Агенты пользователя, скорее всего, проигнорируют такой селектор и таким образом проигнорируют все правило. Однако такое поведение не является гарантированным, потому что разные броузеры обрабатывают ошибки по-разному.

Internet Explorer для Windows вплоть до версии IE6 не распознает правильно комбинированные псевдоклассы. Как и в случае с сочетаниями класс-значение, он обратит внимание только на последний из комбинированных псевдоклассов. Таким образом, если задан `a:link:hover`, более старые версии IE/Win заметят часть `:hover`, но не заметят часть селектора `:link`. В IE7 такого ограничения нет; он правильно обрабатывает комбинированные псевдоклассы.

Селекторы псевдоэлементов

Почти так же, как псевдоклассы назначают фантомные классы для ссылок, псевдоэлементы вводят фиктивные элементы в документ, чтобы достигнуть определенных эффектов. В CSS2.1 определены четыре псевдоэлемента: первая буква, первая строка и применение специальных стилей до и после элемента.

Применение специальных стилей к первой букве

Первый псевдоэлемент участвует в стилевом оформлении первой буквы блочного элемента:

```
p:first-letter {color: red;}
```

Согласно этому правилу первая буква каждого абзаца будет окрашена в красный цвет. Вместо этого можно было бы, например, сделать первую букву каждого элемента `h2` в два раза больше, чем остальной текст:

```
h2:first-letter {font-size: 200%;}
```

Результат применения этого правила показан на рис. 2.25.

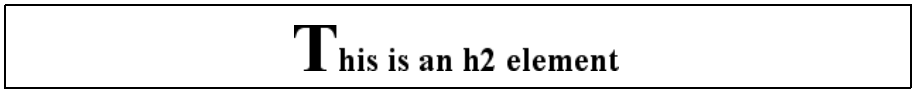


Рис. 2.25. *Псевдоэлемент :first-letter в действии*

Как я говорил, это правило заставляет агент пользователя реагировать на фиктивный элемент, содержащий первую букву каждого элемента `h2`. Это могло бы выглядеть примерно так:

```
<h2><h2:first-letter>T</h2:first-letter>his is an h2 element</h2>
```

Стили `:first-letter` применяются только к содержимому фиктивного элемента, показанного в примере. Элемента `<h2:first-letter>` *нет* в исходной разметке документа. Он создается агентом пользователя на лету и нужен для того, чтобы применить стиль `:first-letter` к соответствующему блоку текста. Иначе говоря, `<h2:first-letter>` – это псевдоэлемент. Помните: вам не надо добавлять никаких новых тегов. Агент пользователя все сделает за вас.

Применение специальных стилей к первой строке

Аналогичным образом `:first-line` может применяться для оформления первой строки текста элемента. Например, можно сделать первую строку каждого абзаца документа фиолетовой:

```
p:first-line {color: purple;}
```

На рис. 2.26 стиль применяется к первой отображаемой строке текста каждого абзаца. Здесь не имеет значения ширина области отображения. Если первая строка содержит только первые пять слов абзаца, то лишь эти пять слов будут закрашены фиолетовым. Если первая строка состоит из 30 слов элемента, тогда фиолетовыми будут все 30.

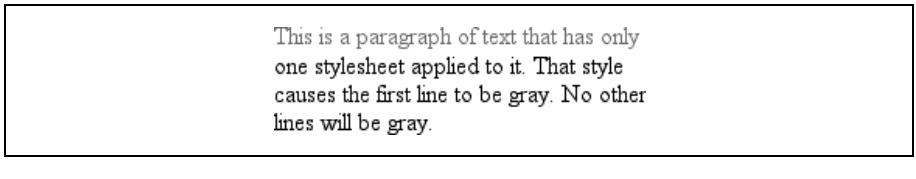


Рис. 2.26. *Псевдоэлемент :first-line в действии*

Поскольку фиолетовым должен быть текст от слова «This» до слова «only», агент пользователя применяет фиктивную разметку, которая выглядит примерно так:

```
<p><p:first-line>This is a paragraph of text that has only</p:first-line>
one stylesheet applied to it. That style
causes the first line to be purple. No other ...
```

Если в результате редактирования первая строка будет включать первые семь слов абзаца, фиктивный тег `</p:first-line>` переместится и окажется сразу за словом «that».

Ограничения :first-letter и :first-line

В CSS2 псевдоэлементы `:first-letter` и `:first-line` могут применяться только к блочным элементам, таким как заголовки или абзацы, но не могут быть применены к строчным элементам, таким как гиперссылки. В CSS2.1 `:first-letter` применяется ко всем элементам. Существуют также ограничения для свойств CSS, которые могут применяться в псевдоэлементах `:first-line` и `:first-letter`. Ограничения показаны в табл. 2.4.

Таблица 2.4. Свойства, применение которых допускается в псевдоэлементах

:first-letter	:first-line
Все свойства `font`	Все свойства `font`
`color`	`color`
Все свойства `background`	Все свойства `background`
Все свойства `margin`	`word-spacing`
Все свойства `padding`	`letter-spacing`
Все свойства `border`	`text-decoration`
`text-decoration`	`vertical-align`
`vertical-align` (если `float` присвоено значение `none`)	`text-transform`
`text-transform`	`line-height`
`line-height`	`clear` (только CSS2; в CSS2.1 удалено)
`float`	`text-shadow` (только CSS2)
`letter-spacing` (добавлено в CSS2.1)	
`word-spacing` (добавлено в CSS2.1)	
`clear` (только CSS2; в CSS2.1 удалено)	
`text-shadow` (только CSS2)	

Кроме того, все псевдоэлементы должны размещаться в самом конце селектора. Следовательно, запись `p:first-line em` неверна, поскольку псевдоэлемент идет раньше селектора объекта (объект – последний из

перечисленных элементов). Такое же правило действует и по отношению к остальным двум псевдоэлементам CSS2.1.

Применение специальных стилей до и после элементов

Предположим, требуется предварить каждый элемент `h2` парой серебряных квадратных скобок:

```
h2:before {content: "]]"; color: silver;}
```

CSS2.1 позволяет вставлять *генерируемое содержимое* (*generated content*), а затем применять к нему специальные стили, используя псевдоэлементы `:before` и `:after`. Пример показан на рис. 2.27.

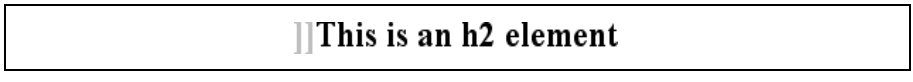


Рис. 2.27. *Вставка содержимого перед элементом*

Псевдоэлемент предназначен для добавления генерируемого содержимого и для применения к нему специальных стилей. Чтобы поместить содержимое после элемента, применяйте псевдоэлемент `:after`. Вы могли бы использовать следующее правило для завершения документов:

```
body:after {content: "  The End.";}
```

Генерируемое содержимое – это отдельный вопрос, и вся эта тема (включая более детальное рассмотрение псевдоэлементов `:before` и `:after`) подробно обсуждается в главе 12.

Заключение

Посредством селекторов, базирующихся на языке документа, авторы могут создавать CSS-правила, применяемые к огромному количеству аналогичных элементов, так же просто, как и создавать правила, имеющие очень ограниченную область применения. Возможность группировать селекторы и правила обеспечивает компактность и гибкость таблиц стилей, что, между прочим, ведет к уменьшению размеров файлов и времени загрузки.

Селекторы – это то, что должно передаваться агентам пользователя без ошибок, потому что невозможность правильно интерпретировать селекторы в большой степени является причиной того, что агенты пользователя вообще отказываются от применения CSS. С другой стороны, для авторов крайне важно правильно записывать селекторы, потому что ошибки могут помешать агенту пользователя применить стили так, как это предполагалось. Основная составляющая правильного понимания селекторов и их сочетаний – понимание взаимодействия селекторов со структурой документа и действия таких механизмов, как наследование и каскад, при определении оформления элемента. Это и есть предмет рассмотрения следующей главы.

3

Структура и каскад

В главе 2 было показано, как структура документа и селекторы CSS позволяют применять различные стили к элементам. Зная, что каждый действительный (valid) документ генерирует структурное дерево, вы можете создавать селекторы, которые определяют целевой элемент через предков, атрибуты, сестринские элементы и т. д. Структурное дерево – это то, благодар селекторы выполняют свои функции; кроме того, это центральное понятие не менее важного аспекта CSS – *наследования* (*inheritance*).

Наследование – это механизм, с помощью которого некоторые значения свойств передаются от элемента к его потомкам. Определяя, какие значения должны быть применены к элементу, агент пользователя должен учитывать не только наследование, но и специфичность объявлений, а также их происхождение. Этот процесс анализа и скрывается за термином *каскад* (*cascade*). В данной главе мы изучим взаимосвязи между этими тремя механизмами: спецификой, наследованием и каскадом.

Хотя все эти понятия могут казаться совершенно абстрактными, не останавливайтесь! Ваша настойчивость будет вознаграждена.

Специфичность

Из главы 2 вы знаете, что существует множество способов выбора необходимых элементов. Фактически один и тот же элемент может быть выбран двумя и более правилами, каждое из которых имеет собственный селектор. Рассмотрим следующие три пары правил. Предположим, что каждая пара будет сопоставляться с одним и тем же элементом:

```
h1 {color: red;}
body h1 {color: green;}

h2.grape {color: purple;}
```

```
h2 {color: silver;}

html > body table tr[id="totals"] td ul > li {color: maroon;}
li#answer {color: navy;}
```

Очевидно, что применено будет только одно из двух правил каждой пары, поскольку сопоставляемый элемент может быть только одного цвета. А как узнать, какое из правил одержит верх?

Ответ кроется в *специфичности* (*specificity*) каждого селектора. Для каждого правила агент пользователя вычисляет специфичность селектора и прикрепляет ее к каждому объявлению правила. Если элемент имеет несколько конфликтующих объявлений свойства, выигрывает то, которое имеет наибольшую специфичность.

Это еще не все, что касается разрешения конфликтов. На самом деле конфликты всех стилей разрешаются каскадом, которому в этой главе посвящен отдельный раздел.

Специфичность селектора определяется компонентами самого селектора. Значение специфичности состоит из четырех частей: `0,0,0,0`. Реальная специфичность селектора определяется следующим образом:

- Для каждого указанного в селекторе значения идентификатора к специфичности добавляется `0,1,0,0`.
- Для каждого указанного в селекторе имени класса, псевдокласса или атрибута к специфичности добавляется `0,0,1,0`.
- Для каждого заданного в селекторе элемента и псевдоэлемента к специфичности добавляется `0,0,0,1`. Внутреннее противоречие CSS2 состояло в том, что не было определено, обладают ли псевдоэлементы какой-либо специфичностью, но в CSS2.1 их специфичность явно определена и учитывается данным правилом.
- Комбинаторы и универсальный селектор не учитываются (более подробно об этих значениях позже).

Таким образом, вычисляя специфичность следующих селекторов правил, получаем:

```
h1 {color: red;}  /* специфичность = 0,0,0,1 */
p em {color: purple;}  /* специфичность = 0,0,0,2 */
.grape {color: purple;}  /* специфичность = 0,0,1,0 */
*.bright {color: yellow;}  /* специфичность = 0,0,1,0 */
p.bright em.dark {color: maroon;}  /* специфичность = 0,0,2,2 */
#id216 {color: blue;}  /* специфичность = 0,1,0,0 */
div#sidebar *[href] {color: silver;}  /* специфичность = 0,1,1,1 */
```

В приведенном выше примере элементу `em` сопоставляются второе и пятое правила, при этом цвет элемента будет красно-коричневым, потому что специфичность пятого правила больше.

Чтобы поупражняться, давайте вернемся к парам правил, приведенным в начале раздела, и вычислим специфичности:

```
h1 {color: red;}  /* 0,0,0,1 */
body h1 {color: green;}  /* 0,0,0,2  (победитель)*/

h2.grape {color: purple;}  /* 0,0,1,1 (победитель) */
h2 {color: silver;}  /* 0,0,0,1 */

html > body table tr[id="totals"] td ul > li {color: maroon;}  /* 0,0,1,7 */
li#answer {color: navy;}  /* 0,1,0,1  (победитель) */
```

Правило-победитель в каждой паре определяется по более высокому показателю специфичности. Обратите внимание, как они сортируются. Во второй паре выигрывает селектор `h2.grape`, потому что у него на одну единицу больше: `0,0,1,1` побеждает `0,0,0,1`. В третьей паре выигрышным является второе правило, потому что `0,1,0,1` берет верх над `0,0,1,7`. Фактически значение специфичности `0,0,1,0` выиграет и у значения `0,0,0,13`.

Дело в том, что вес значения растет слева направо. Специфичность `1,0,0,0` возьмет верх над любой специфичностью, которая начинается с `0`, независимо от того, какими будут остальные числа. Таким образом, `0,1,0,1` выигрывает у `0,0,1,7`, потому что `1`, стоящая на втором месте первого значения, побеждает `0` на этом месте во втором значении.

Объявления и специфичность

Как только специфичность селектора вычислена, ее значение передается всем ассоциированным с селектором объявлениям. Рассмотрим следующее правило:

```
h1 {color: silver; background: black;}
```

Чтобы определить специфичность, агент пользователя должен рассматривать правило так, как будто бы оно разделено на отдельные «разгруппированные». правила. Таким образом, предыдущий пример принимает вид:

```
h1 {color: silver;}
h1 {background: black;}
```

Оба правила имеют специфичность `0,0,0,1`, и именно это значение присваивается каждому объявлению. Аналогичный процесс разделения имеет место и с группированными селекторами. Данное правило:

```
h1, h2.section {color: silver; background: black;}
```

агент пользователя интерпретирует следующим образом:

```
h1 {color: silver;}  /* 0,0,0,1 */
h1 {background: black;}  /* 0,0,0,1 */
h2.section {color: silver;}  /* 0,0,1,1 */
h2.section {background: black;}  /* 0,0,1,1 */
```

Это важно в ситуациях, когда одному элементу сопоставляются несколько правил и возникает конфликт нескольких объявлений. Рассмотрим правило:

```
h1 + p {color: black; font-style: italic;}  /* 0,0,0,2 */
p {color: gray; background: white; font-style: normal;}  /* 0,0,0,1 */
*.aside {color: black; background: silver;}  /* 0,0,1,0 */
```

Если его применить к следующей разметке, будет сформировано содержимое, показанное на рис. 3.1:

```
<h1>Greetings!</h1>
<p class="aside">
It's a fine way to start a day, don't you think?
</p>
<p>
There are many ways to greet a person, but the words are not as important as
the act of greeting itself.
</p>
<h1>Salutations!</h1>
<p>
There is nothing finer than a hearty welcome from one's fellow man.
</p>
<p class="aside">
Although a thick and juicy hamburger with bacon and mushrooms runs
a close second.
</p>
```

В любом случае агент пользователя находит, какие правила сопоставляются элементу, вычисляет все ассоциированные объявления и их специфичности, устанавливает победителей и затем применяет их к элементу для получения результата, оформленного в соответствии с указанными стилями. Такие действия должны быть осуществлены для каждого элемента, селектора и объявления. К счастью, агент пользователя делает все автоматически. Это поведение представляет собою важный компонент каскада, рассматриваемый далее в данной главе.

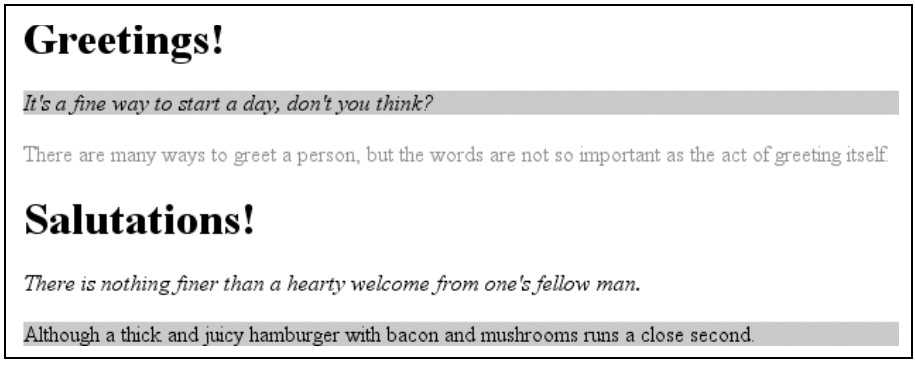


***Рис. 3.1.** Влияние различных правил на документ*

Специфичность универсального селектора

Как упоминалось ранее, универсальный селектор не принимает участия в вычислении специфичности селектора. Иначе говоря, он имеет специфичность 0,0,0,0, а это не то же самое, что совсем не иметь специфичности (этот вопрос будет обсуждаться в разделе «Наследование»). Поэтому исходя из следующих двух правил цвет текста в абзаце, являющемся потомком div, будет черным, а во всех остальных элементах – серым:

```
div p {color: black;}  /* 0,0,0,2 */
* {color: gray;}  /* 0,0,0,0 */
```

Как вы, должно быть, и ожидали, это означает, что специфичность селектора не меняется от факта присутствия или отсутствия в нем универсального селектора. Следующие два селектора имеют совершенно одинаковую специфичность:

```
div p  /* 0,0,0,2 */
body * strong  /* 0,0,0,2 */
```

Комбинаторы, для сравнения, вообще не имеют никакой специфичности, даже нулевой. Таким образом, они вообще не оказывают никакого влияния на общую специфичность селектора.

Специфичность селекторов идентификаторов и атрибутов

Важно отметить разницу в специфичности селектора идентификатора (ID selector) и селектора атрибутов (attribute selector), в котором указан атрибут id. Возвращаясь к третьей паре правил в примере кода, мы находим:

```
html > body table tr[id="totals"] td ul > li {color: maroon;}  /* 0,0,1,7 */
li#answer {color: navy;}  /* 0,1,0,1  (победитель) */
```

К общей специфичности селектор идентификатора (#answer) из второго правила добавляет 0,1,0,0. В то же время в первом правиле селектор атрибута ([id="totals"]) добавляет к значению специфичности 0,0,1,0. Таким образом, приведенные ниже правила определяют, что цвет элемента, значение атрибута id которого равно meadow, будет зеленым:

```
#meadow {color: green;}  /* 0,1,0,0 */
*[id="meadow"] {color: red;}  /* 0,0,1,0 */
```

Специфичность подставляемых в строку стилей

Все рассмотренные до сих пор значения специфичности начинались с нуля, и вы, наверное, хотите знать, зачем он вообще там нужен. Дело в том, что этот первый нуль зарезервирован для встроенных объявлений стилей, специфичность которых превосходит специфичность всех

остальных объявлений. Рассмотрим следующее правило и фрагмент разметки:

```
h1 {color: red;}

<h1 style="color: green;">Вечеринка на лугу</h1>
```

Исходя из того, что это правило применяется к элементу `h1`, можно ожидать, что текст `h1` будет зеленым. В CSS2.1 так и происходит, и объясняется это тем, что специфичность данного объявления равна `1,0,0,0`.

Это значит, что даже элементы с атрибутами `id`, которые сопоставляются с правилом, подчиняются встроенным в тег стилям. Изменим предыдущий пример и включим в него атрибут `id`:

```
h1#meadow {color: red;}

<h1 id="meadow" style="color: green;">Вечеринка на лугу</h1>
```

В соответствии со специфичностью встроенных объявлений текст элемента `h1` по-прежнему будет зеленым.

Приоритетность подставляемых в строку объявлений стилей является новшеством для CSS2.1 и была введена при написании CSS2.1 для фиксации поведения веб-броузера. В CSS2 специфичность подставляемого в строку объявления стиля была `1,0,0` (специфичности CSS2 составлялись из трех значений, а не из четырех). Иначе говоря, его специфичность была такой же, как и у селектора идентификатора, который при этом перекрывал встраиваемые объявления стилей.

Важность

Иногда важность объявления настолько велика, что перевешивает все остальные факторы. CSS2.1 называет их (по вполне понятным причинам) *важными объявлениями* (*important declarations*) и предоставляет вам возможность отмечать их путем введения в объявление ключевого слова `!important` прямо перед завершающей точкой с запятой:

```
p.dark {color: #333 !important; background: white;}
```

Здесь значение цвета `#333` отмечено как `!important`, тогда как цвет фона `white` – нет. Если вы хотите сделать важными оба объявления, каждому из них понадобится собственный маркер `!important`:

```
p.dark {color: #333 !important; background: white !important;}
```

Необходимо правильно размещать `!important`, иначе объявление будет признано недействительным. Ключевое слово `!important` всегда располагается в конце объявления, прямо перед точкой с запятой. Это особенно важно, когда дело доходит до свойств (например, `font`), значения которых могут состоять из нескольких ключевых слов:

```
p.light {color: yellow; font: smaller Times, serif !important;}
```

Если бы `!important` было расположено где-либо в другом месте объявления `font`, то все объявление было бы признано недействительным и ни один из его стилей не был бы применен.

Объявления, отмеченные как `!important`, не имеют особого значения специфичности, они рассматриваются отдельно от остальных. Фактически все объявления `!important` группируются вместе, и тогда уже их конфликты специфичностей разрешаются относительно друг друга. Аналогично группируются все остальные объявления, и конфликты свойств разрешаются с помощью специфичностей. В любом случае, когда имеет место конфликт важного и неважного объявления, всегда побеждает важное.

На рис. 3.2 показан результат применения следующих правил и фрагмента разметки:

```
h1 {font-style: italic; color: gray !important;}
.title {color: black; background: silver;}
* {background: black !important;}

<h1 class="title">NightWing</h1>
```

Рис. 3.2. *Важные объявления всегда побеждают*

Важные объявления и их обработка более подробно обсуждаются в разделе «Каскад» данной главы.

Наследование

Для понимания механизма применения объявлений к документу не менее важное значение, чем специфичность, имеет еще одно ключевое понятие – наследование. Наследование – это механизм, с помощью которого стили применяются не только к указанным элементам, но также к их потомкам. Например, если цвет применен к элементу `h1`, то этот цвет будет применен ко всему тексту в `h1` и даже к тому, который заключен в дочерние элементы этого `h1`.

```
h1 {color: gray;}

<h1>Meerkat <em>Central</em></h1>
```

И обычный текст `h1`, и текст `em` окрашены в серый цвет, потому что элемент `em` наследует значение `color`. Если бы значения свойств не наследовались элементами-потомками, текст `em` был бы черным, а не серым, и пришлось бы окрашивать этот элемент отдельно.

Наследование так же хорошо работает с ненумерованными списками. Скажем, стиль `color: gray;` применяется к элементам `ul`:

```
ul {color: gray;}
```

Можно ожидать, что стиль, примененный к `ul`, также будет применен и к элементам списка, и к любому их содержимому. Благодаря наследованию именно это и происходит, как видно на рис. 3.3.

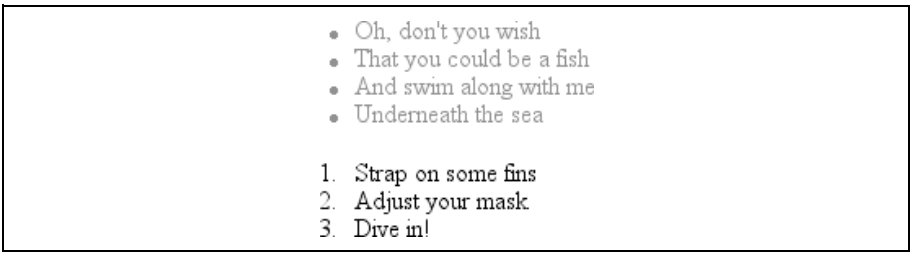


Рис. 3.3. *Наследование стилей*

Лучше всего принцип работы наследования иллюстрирует древовидное представление документа. На рис. 3.4 показана древовидная схема простого документа, содержащего два списка – ненумерованный и нумерованный.

Когда к элементу `ul` применяется объявление `color: gray;`, он принимает это объявление. Затем это значение передается вниз по дереву элементов-потомков до тех пор, пока не останется потомков, которые могли бы наследовать это значение. Значения *никогда* не передаются вверх по иерархии, т. е. элемент никогда не передает значение своим предкам.

В HTML есть исключение из правила восходящего распространения: стили фона, применяемые к элементу `body`, могут передаваться элементу `html`, представляющему собою корневой элемент документа и, следовательно, определяющему его полотно (canvas).

Наследование является настолько фундаментальной концепцией CSS, что о ней практически никогда не задумываются, пока не сталкиваются с ней непосредственно. Однако кое-что следует помнить.

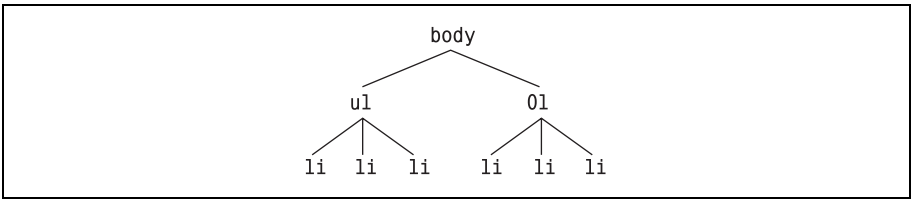


Рис. 3.4. *Простая древовидная схема*

Во-первых, обратите внимание, что некоторые свойства не наследуются, обычно просто из соображений здравого смысла. Например, не наследуется свойство `border` (предназначенное для обрамления элемента). Беглого взгляда на рис. 3.5 хватит, чтобы понять почему. Если бы рамки наследовались, документы стали бы намного более загроможденными, и автору приходилось бы прилагать дополнительные усилия, чтобы отключить унаследованные рамки.

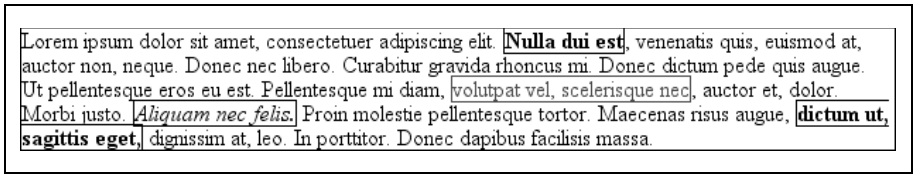


Рис. 3.5. *Почему рамки не наследуются*

Кстати, большинство свойств блочной модели, в том числе отступы, выравнивание, фон и рамки, не наследуются по той же причине. В конце концов, вы же не хотите, чтобы все ссылки внутри абзаца наследовали от родительского элемента отступ слева в 30 пикселов!

С точки зрения специфичности унаследованные значения вообще не имеют специфичности, даже нулевой. Это кажется формальным отличием до тех пор, пока вы не проработаете последствия отсутствия унаследованной специфичности. Рассмотрим следующие правила и фрагмент разметки и сравним их с результатом, приведенным на рис. 3.6:

```
* {color: gray;}
h1#page-title {color: black;}

<h1 id="page-title">Meerkat <em>Central</em></h1>
<p>
Welcome to the best place on the web for meerkat information!
</p>
```

Поскольку универсальный селектор применяется ко всем элементам и имеет нулевую специфичность, заданный в нем серый цвет одерживает верх над унаследованным значением `black`, которое вообще не имеет никакой специфичности. Следовательно, цвет элемента `em` будет серым, а не черным.

Рис. 3.6. *Нулевая специфичность одерживает победу над полным отсутствием таковой*

Наследование ошибок

Если автор хочет гарантировать работу кода в любых обстоятельствах, то ему нельзя полагаться на наследование из-за проблем в различных реализациях броузеров. Например, в Navigator 4 (и в меньшей степени Explorer 4 и 5) таблицы не наследуют стили. Таким образом, результатом применения следующего правила станет уменьшение размера всего текста, находящегося вне таблиц:

```
body {font-size: 0.8em;}
```

Такой код нарушает требования CSS, но встречается, поэтому исторически сложилось так, что авторы обращаются за помощью к таким трюкам:

```
body, table, th, td {font-size: 0.8em;}
```

Это более определенно, хотя по-прежнему не гарантирует достижения желаемого эффекта в недоработанных броузерах.

К сожалению, приведенная выше «поправка» ведет к еще более страшным проблемам в броузерах, правильно реализующих наследование, таких как IE6/Win, IE5/Mac, Netscape 6+ и др. В этих броузерах размер текста внутри ячейки таблицы будет составлять 41% от заданного пользователем стандартного размера шрифта. Часто попытки обойти ошибки наследования в старых броузерах опаснее, чем написание правильного CSS для обновленных броузеров.

Этот пример ярко иллюстрирует одну из потенциальных трудностей, порождаемых необдуманным применением универсального селектора. Поскольку такой селектор соответствует любому элементу, он часто вызывает эффект наследования с силой короткого замыкания. С этим можно справиться, но обычно благоразумней вообще избежать неприятностей, не применяя универсальный селектор произвольно.

Полное отсутствие специфичности у унаследованных значений отнюдь не мелочь. Предположим, таблица стилей написана так, что весь текст панели инструментов должен быть белым на черном фоне:

```
#toolbar {color: white; background: black;}
```

Это правило будет выполняться, только если элемент, атрибут `id` которого имеет значение `toolbar`, не содержит ничего, кроме обычного текста. Если, однако, весь текст этого элемента представляет собой гиперссылку (элемент `a`), тогда верх возьмут стили агента пользователя для гиперссылок. Это значит, что в веб-броузере они, скорее всего, будут окрашены синим цветом, поскольку таблица стилей броузера, вероятно, содержит подобную запись:

```
a:link {color: blue;}
```

Для решения этой проблемы должно быть объявлено:

```
#toolbar {color: white; background: black;}
#toolbar a:link {color: white;}
```

Ориентируя правила непосредственно на элементы a в составе панели инструментов, получим результат, показанный на рис. 3.7.

Рис. 3.7. *Прямое назначение стилей соответствующим элементам*

Каскад

До сих пор в данной главе мы обходили один весьма важный вопрос: что происходит, когда к одному элементу применяются два правила с равной специфичностью? Как броузер разрешает этот конфликт? Например, имеются следующие правила:

```
h1 {color: red;}
h1 {color: blue;}
```

Какое из них победит? Специфичность обоих равна 0, 0, 0, 1, у них равные шансы, и они оба должны быть применены. Это просто невозможно, потому что элемент не может быть и красным, и синим одновременно. Но каким он будет?

Наконец-то название «Каскадные таблицы стилей» обретает некоторый смысл. CSS основывается на методе каскадирования стилей, реализация которого стала возможной благодаря сочетанию наследования и специфичности. Правила каскадирования для CSS2.1 предельно просты:

1. Найти все правила, содержащие селектор, сопоставляемый с данным элементом.
2. Провести сортировку согласно явной приоритетности всех применяемых к элементу объявлений. Правилам, отмеченным как !important, присваивается более высокий приоритет, чем остальным. Также все применяемые к данному элементу объявления сортируются согласно их источнику. Существует три возможных источника правил: автор, читатель и агент пользователя. В общем случае стили автора приоритетней стилей читателя. Но стили читателя, отмеченные как !important, сильнее, чем все остальные стили, включая и те стили автора, которые отмечены как !important. И стили автора, и стили читателя замещают применяемые по умолчанию стили агента пользователя.
3. Провести сортировку всех объявлений, применяемых к элементу, согласно их специфичности. Элементы с более высокой специфич-

ностью имеют больший приоритет по сравнению с теми, специфичность которых ниже.

4. Провести сортировку всех объявлений, применяемых к элементу, в соответствии с очередностью расположения. Чем позже объявление появляется в таблице стилей или документе, тем больший приоритет ему присваивается. Считается, что объявления, находящиеся в импортированных таблицах стилей, располагаются перед всеми объявлениями импортировавшей их таблицы стилей.

Чтобы полностью разобраться в том, как все это работает, рассмотрим три примера, иллюстрирующих последние три из четырех правил каскадирования.

Сортировка по приоритетности и источнику

Согласно второму правилу, если к элементу применяются два правила и одно из них отмечено как `!important`, то оно побеждает:

```
p {color: gray !important;}

<p style="color: black;">Well,<em> hello</em> there!</p>
```

Несмотря на то что цвет задан в атрибуте `style` абзаца, побеждает правило с пометкой `!important`, и текст абзаца становится серым. Этот серый цвет также наследуется элементом `em`.

Более того, учитывается источник правила. Если с элементом сопоставляются стили с обычной приоритетностью из таблицы стилей автора и таблицы стилей читателя, применяются стили автора. Предположим, что следующие стили происходят из указанных источников:

```
p em {color: black;}    /* таблица стилей автора */

p em {color: yellow;}   /* таблица стилей читателя */
```

В данном случае выделенный текст параграфа закрашивается черным цветом, а не желтым, потому что стили автора с обычной приоритетностью побеждают обладающие обычным приоритетом стили читателя. Однако если оба правила отмечены как `!important`, ситуация меняется:

```
p em {color: black !important;}    /* таблица стилей автора */

p em {color: yellow !important;}   /* таблица стилей читателя */
```

Теперь выделенный текст параграфа будет желтым, а не черным.

Так сложилось, что применяемые по умолчанию стили агента пользователя, которые обычно отражают предпочтения пользователя, учитываются именно так. Применяемые по умолчанию объявления стилей вообще относятся к категории правил, обладающей наименьшим влиянием. Поэтому если заданное автором правило применяется к тегам `<a>` (например, объявляет, что они будут белыми), то оно замещает стандартные настройки агента пользователя.

Подводя итог, скажем, что с точки зрения приоритетности объявлений выделены пять уровней. В порядке уменьшения приоритетности это:

1. Важные объявления читателя.
2. Важные объявления автора.
3. Обычные объявления автора.
4. Обычные объявления читателя.
5. Объявления агента пользователя.

Авторам обычно надо беспокоиться только о первых четырех уровнях приоритетности, поскольку приоритет всего, что объявляется, будет выше, чем приоритет стилей агента пользователя.

Сортировка по специфичности

Согласно третьему правилу, если к элементу применяются конфликтующие объявления и все они имеют одинаковую приоритетность, они должны сортироваться в соответствии со специфичностью. Побеждает объявление, обладающее наибольшей специфичностью. Например:

```
p#bright {color: silver;}
p {color: black;}

<p id="bright">Well, hello there!</p>
```

Исходя из приведенных правил текст параграфа будет окрашен в серебристый цвет, как показано на рис. 3.8. Почему? Потому что специфичность `p#bright` (`0, 1, 0, 1`) превышает специфичность `p` (`0, 0, 0, 1`), даже несмотря на то, что последнее правило расположено в таблице стилей позже.

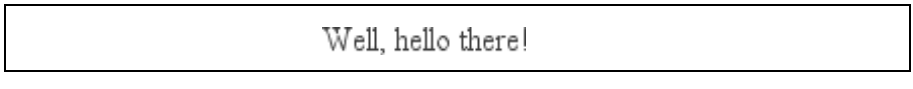


***Рис. 3.8.** Большая специфичность побеждает меньшую*

Сортировка по порядку расположения

И наконец, согласно четвертому правилу, если два правила имеют совершенно одинаковые приоритетность, источник и специфичность, тогда побеждает то из них, которое расположено в таблице стилей позже. Вернемся к нашему предыдущему примеру, в котором мы нашли следующие два правила таблицы стилей документа:

```
h1 {color: red;}
h1 {color: blue;}
```

Значение `color` для всех элементов `h1` документа будет `blue`, а не `red`, поскольку именно это правило стоит в таблице стилей ниже. Любое правило, содержащееся в документе и имеющее более высокий приоритет, чем импортированное правило, побеждает. Это выполняется, да-

же если правило является частью таблицы стилей документа, а не частью атрибута style элемента. Рассмотрим следующее:

```
p em {color: purple;}  /* из импортированной таблицы стилей */

p em {color: gray;}    /* правило, содержащееся в документе */
```

В этом случае второе правило одержит верх над импортированным, потому что оно является частью таблицы стилей документа.

С позиций этого правила принимается, что стили, определенные в атрибуте style элемента, находятся в самом конце таблицы стилей документа, т. е. размещаются после всех остальных правил. Однако эти рассуждения часто имеют только теоретическое значение, поскольку в CSS2.1 подставляемые в строку объявления стилей имеют более высокую специфичность, чем любой селектор таблицы стилей.

Помните, что в CSS2 подставляемые в строку объявления стилей имеют специфичность, эквивалентную селекторам идентификаторов. В CSS2 (но не в CSS2.1) считается, что объявления атрибута style находятся в конце таблицы стилей документа и их сортировка происходит согласно приоритету, источнику, специфичности и т. д., как для любого другого объявления таблицы стилей.

Принцип сортировки по порядку расположения лежит в основе рекомендуемого порядка расположения стилей ссылок. Рекомендуется выстраивать ваши стили ссылок в порядке link-visited-hover-active или LVHA, например:

```
:link {color: blue;}
:visited {color: purple;}
:hover {color: red;}
:active {color: orange;}
```

Теперь вы знаете, что специфичность всех этих селекторов одинакова: 0,0,1,0. Поскольку все они имеют одинаковые приоритет, источник и специфичность, верх одерживает тот из них, который сопоставляется с элементом последним. При щелчке по непосещенной ссылке последняя сопоставляется с тремя правилами – :link, :hover и :active, в результате побеждает то из них, которое объявлено последним. Исходя из порядка LVHA это будет :active, что, вероятно, и предполагалось автором.

Представим на секунду, что вы решили пренебречь обычным порядком расположения и вместо этого поместили стили ваших ссылок по алфавиту. При этом получилось бы следующее:

```
:active {color: orange;}
:hover {color: red;}
:link {color: blue;}
:visited {color: purple;}
```

При таком расположении ни одна из ссылок никогда не продемонстрирует стили :hover или :active, потому что правила :link и :visited следуют после них. Каждая ссылка должна быть или посещенной, или непосещенной, так что эти стили всегда будут перекрывать правило :hover.

Давайте посмотрим, что получится, если автор решит использовать порядок LVHA. При таком порядке расположения непосещенные ссылки будут получать стиль :hover, а посещенные ссылки – нет. Стиль :active будет применяться и к посещенным, и к непосещенным ссылкам:

```
:link {color: blue;}
:hover {color: red;}
:visited {color: purple;}
:active {color: orange;}
```

Конечно, подобные конфликты возникают, когда все выражения пытаются задавать одно и то же свойство. Если все выражения задают разные свойства, то порядок их расположения не имеет значения. В следующем случае стили ссылок могли бы быть заданы в любом порядке и при этом не потеряли бы своей функциональности:

```
:link {font-weight: bold;}
:visited {font-style: italic;}
:hover {color: red;}
:active {background: yellow;}
```

Вы, возможно, уже также поняли, что порядок стилей :link и :visited не важен. Без всякого вреда можно было бы расположить стили в порядке LVHA или VLHA. Однако LVHA предпочтительнее, потому что этот порядок был рекомендован в спецификации CSS2 и еще потому, что мнемоническая схема «LoVe–HA!» получила довольно широкое распространение. (Правда, в этом есть что-то грустное.)

Возможность объединения псевдоклассов устраняет эти проблемы. Следующие стили могут быть расположены в любом порядке без каких-либо негативных последствий:

```
:link {color: blue;}
:visited {color: purple;}
:link:hover {color: red;}
:visited:hover {color: gray;}
```

Поскольку каждое правило применяется к уникальному набору состояний ссылки, конфликта между ними не возникает. Следовательно, изменение порядка их расположения не повлечет за собой изменения оформления документа. Специфичность двух последних правил одинакова, но это не имеет значения. Непосещенная ссылка, над которой расположен указатель мыши, не будет сопоставляться с правилом, относящимся к посещенной ссылке, над которой расположен указатель,

и наоборот. Если бы мы добавили стили для активного состояния, порядок опять имел бы значение. Рассмотрим следующий вариант:

```
:link {color: blue;}
:visited {color: purple;}
:link:hover {color: red;}
:visited:hover {color: gray;}
:link:active {color: orange;}
:visited:active {color: silver;}
```

Если поместить стили для активного состояния перед стилями `:hover`, они будут проигнорированы. Это опять происходит из-за конфликтов специфичности. Их можно было бы избежать путем введения в последовательности большего количества псевдоклассов, например, так:

```
:link:hover:active {color: orange;}
:visited:hover:active {color: silver;}
```

Объединенные в последовательности псевдоклассы, позволяющие меньше заботиться о специфичности и порядке расположения, могли бы применяться намного чаще, если бы поддерживались Internet Explorer. (Более подробно об этом см. в главе 2.)

Инструкции по оформлению, не принадлежащие CSS

Документ может содержать инструкции по оформлению, не принадлежащие CSS, например элемент `font`. Не принадлежащие CSS инструкции интерпретируются так, как будто их специфичность равна 0 и они находятся в *начале* таблицы стилей автора. Подобные инструкции по оформлению будут переопределяться стилями автора или читателя, но не стилями агента пользователя.

Заключение

Возможно, самым фундаментальным понятием каскадных таблиц стилей является сам каскад – процесс, с помощью которого упорядочиваются конфликтующие объявления и исходя из которого определяется окончательное представление документа. Важная составляющая часть этого процесса – специфичность селекторов и ассоциированных с ними объявлений и механизм наследования.

В следующей главе мы обратимся к различным типам единиц измерения, предназначенным для придания смысла значениям свойств. Этим мы закончим обсуждение основных принципов, и вы будете готовы изучать свойства, применяемые для оформления документов.

4

Значения и единицы измерения

В этой главе мы коснемся элементов, составляющих основу практически всего, что позволяют делать CSS, – *единиц измерений* (*units*), применяемых во всех свойствах для задания цвета, расстояний и размеров. Без единиц измерения нельзя объявить о том, что текст абзаца должен быть фиолетовым, или что вокруг изображения должно быть пустое пространство в 10 пикселов, или что текст заголовка должен иметь определенный размер. Поняв основные принципы, изложенные в этой главе, вы намного быстрее изучите и начнете использовать все остальное в CSS.

Числа

В CSS существует два типа чисел: *целые* (*integer*) и *вещественные* (*real*). Эти типы чисел в большинстве случаев служат базой для всех остальных типов значений, но иногда в качестве значений свойств могут выступать числа других форматов.

В CSS2.1 вещественное число определяется как целое, за которым могут следовать десятичная точка и дробная часть. Таким образом, все эти числа являются действительными числовыми значениями: `15.5`, `-270.00004` и `5`. И целые, и вещественные числа могут быть как положительными, так и отрицательными, хотя свойства могут (и часто так оно и есть) ограничивать диапазон принимаемых ими чисел.

Процентные значения

Процентное значение (*percentage value*) – это вычисляемое вещественное число, за которым следует знак процента (`%`). Процентные значения практически всегда выражены относительно другого значения, которым может быть все что угодно, включая значение другого свойст-

ва того же элемента, значение, унаследованное от родительского элемента, или значение элемента-предка. Любое свойство, принимающее значения, задаваемые в процентах, определяет свои ограничения на допустимый диапазон процентных значений и точность представления относительных процентных значений.

Цвет

Один из первых вопросов, задаваемых каждым начинающим автором веб-страниц: «Как устанавливать цвета на странице?». HTML предоставляет на выбор два варианта: взять один из немногочисленных именованных цветов, такой как `red` или `purple`, либо оперировать таинственными шестнадцатеричными кодами. Оба этих способа описания цветов сохранились и в CSS, но появились и некоторые другие, как я полагаю, более понятные методы.

Именованные цвета

Тем, кого удовлетворяет небольшой базовый набор цветов, проще всего указать имя цвета. CSS достаточно логично называет все эти предоставляемые на выбор цвета *именованными цветами* (*named colors*).

Вопреки тому, во что нас пытаются заставить поверить некоторые создатели броузеров, имеется очень небольшое количество действительных ключевых слов для обозначения именованных цветов. Например, нельзя выбрать «перламутровый» цвет, поскольку для него ключевое слово отсутствует. Что касается CSS2.1, спецификация CSS определяет 17 цветов. Это 16 цветов, описанных в HTML 4.01, плюс оранжевый (`orange`):

aqua	fuchsia	lime	olive	red	white
black	gray	maroon	orange	silver	yellow
blue	green	navy	purple	teal	

Итак, скажем, вы хотите, чтобы все заголовки первого уровня были красно-коричневыми. Лучше всего объявить их так:

```
h1 {color: maroon;}
```

Просто и понятно, не так ли? На рис. 4.1 показаны еще несколько примеров:

```
h1 {color: gray;}
h2 {color: silver;}
h3 {color: black;}
```

Конечно, встречаются (а может быть, вы даже сами использовали) названия цветов, отсутствующие в приведенном выше списке. Например, если задать:

```
h1 {color: lightgreen;}
```

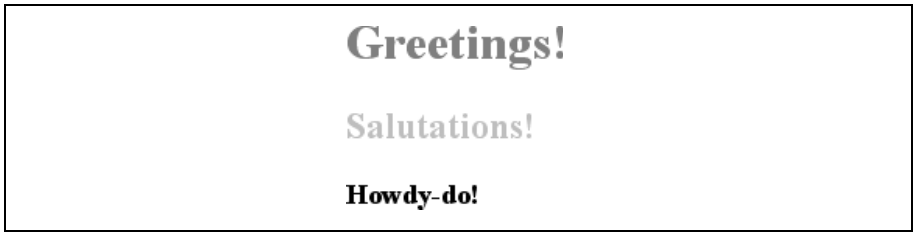


Рис. 4.1. *Именованные цвета*

Скорее всего, все элементы `h1` и в самом деле станут светло-зелеными, несмотря на то, что ключевого слова `lightgreen` нет в списке именованных цветов CSS2.1. Дело в том, что большинство веб-броузеров распознают до 140 цветов, включая стандартные 17. Эти дополнительные цвета определены в цветовой спецификации CSS3, которая в данной книге не рассматривается. Эти стандартные цвета (на момент написания книги), вероятно, надежнее, чем более длинный список из 140 или около этого цветов, потому что коды цветов для этих 17 определены в CSS2.1. Расширенный список из 140 цветов, появившийся в CSS3, базируется на стандартных значениях X11 RGB, которые используются десятилетиями, поэтому они, вероятно, будут поддерживаться очень хорошо.

К счастью, в CSS существуют более точные способы задания цветов. Их преимущество в том, что они позволяют определить любой цвет цветового спектра, а не только 17 (или 140) именованных цветов.

Цветовая модель RGB

Компьютеры создают цвета путем комбинирования различных уровней красного, зеленого и синего. Такую комбинацию часто называют *цветовой моделью RGB*. Кстати, если открыть старый ЭЛТ-монитор компьютера и залезть в проекционный кинескоп поглубже, можно обнаружить три электронных пушки. (Однако вы их лучше не ищите, если не хотите лишиться гарантии на монитор.) Эти пушки испускают в каждую точку экрана пучки электронов с различной интенсивностью. В этих точках яркости каждого из лучей комбинируются, образуя все цвета, которые вы видите. Каждая точка называется *пикселом* (*pixel*). К данному термину мы вернемся в этой главе несколько позже. Даже несмотря на то, что в большинстве современных мониторов электронные пушки уже не используются, их цветовой вывод все равно основан на комбинации трех цветов.

Исходя из того, как происходит формирование цвета на мониторе, необходим прямой доступ к этим цветам, чтобы была возможность определять собственные сочетания трех цветов. Это непросто, но реально, и результат стоит того, потому что в этом случае ресурсы создания цветов практически не ограничены. Существует четыре способа формирования цвета.

Функциональные RGB-цвета

Существует два типа кодов цветов, основанных на *функциональном формате записи RGB* (*functional RGB notation*), в противоположность шестнадцатеричной нотации. Обобщенный синтаксис этого типа кодировки цвета – `rgb(color)`, где `color` представляет собой комбинацию трех процентных значений или целых чисел. Допустимый диапазон процентных значений – от `0%` до `100%`, а диапазон целых значений – от `0` до `255`.

Следовательно, код, задающий белый и черный цвета с помощью процентных значений, будет таким:

```
rgb(100%, 100%, 100%)
rgb(0%, 0%, 0%)
```

А вот те же цвета, представленные записью из целых чисел (integer-triplet notation):

```
rgb(255, 255, 255)
rgb(0, 0, 0)
```

Предположим, требуется, чтобы элементы `h1` были закрашены произвольным оттенком красного – чем-то средним между красным и красно-коричневым. Цвет `red` эквивалентен записи `rgb(100%, 0%, 0%)`, тогда как `maroon` равен `(50%, 0%, 0%)`. Можно попробовать получить цвет, промежуточный между этими двумя, посредством такой записи:

```
h1 {color: rgb(75%, 0%, 0%);}
```

При этом красный компонент цвета становится светлее, чем `maroon`, но темнее, чем `red`. Если же надо получить бледно-красный цвет, то можно повысить уровни зеленого и синего:

```
h1 {color: rgb(75%, 50%, 50%);}
```

Самый близкий к этому эквивалент цвета при записи с помощью целых чисел:

```
h1 {color: rgb(191, 127, 127);}
```

Самый простой способ визуализировать соответствия цвет-код состоит в том, чтобы создать таблицу кодов для оттенков серого. Кроме того, на иллюстрации в книге можно изобразить только шкалу оттенков серого (рис. 4.2):

```
p.one {color: rgb(0%, 0%, 0%);}
p.two {color: rgb(20%, 20%, 20%);}
p.three {color: rgb(40%, 40%, 40%);}
p.four {color: rgb(60%, 60%, 60%);}
p.five {color: rgb(80%, 80%, 80%);}
p.six {color: rgb(0, 0, 0);}
p.seven {color: rgb(51, 51, 51);}
p.eight {color: rgb(102, 102, 102);}
p.nine {color: rgb(153, 153, 153);}
p.ten {color: rgb(204, 204, 204);}
```

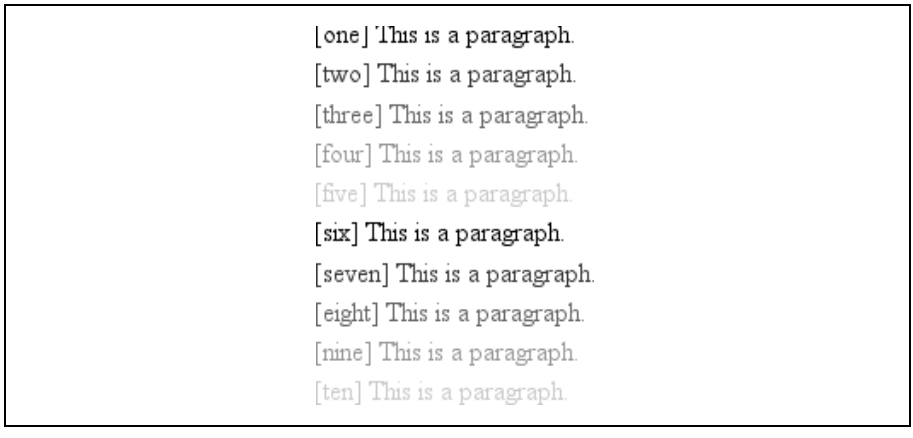


***Рис. 4.2.** Текст, представленный в оттенках серого*

Конечно, поскольку мы работаем с оттенками серого, все три компонента RGB-кода в каждом выражении одинаковые. Если бы любое из них отличалось от других, начал бы проявляться цвет. Если, например, `rgb(50%, 50%, 50%)` превратить в `rgb(50%, 50%, 60%)`, получился бы серый цвет с небольшой синевой.

Процентные значения можно записывать дробными числами. Например, по каким-либо причинам вы хотите определить цвет, содержащий точно 25,5% красного, 40% зеленого и 98,6% синего:

```
h2 {color: rgb(25.5%,40%,98.6%);}
```

Если агент пользователя игнорирует десятичные точки (а некоторые так и поступают), он округлит значение до ближайшего целого, и в результате будет объявлено `rgb(26%, 40%, 99%)`. Запись в виде трех целых ограничивает нас, естественно, целыми значениями.

Значения, которые выходят за рамки допустимого диапазона для каждого из форматов записи, «урезаются» до ближайшей границы диапазона. Это значит, что значение, которое больше `100%` или меньше `0%`, будет по умолчанию приведено к этим допустимым границам. Таким образом, следующие объявления были бы интерпретированы так, как показано в комментариях:

```
P.one {color: rgb(300%,4200%,110%);}   /* 100%,100%,100% */
P.two {color: rgb(0%,-40%,-5000%);}    /* 0%,0%,0% */
p.three {color: rgb(42,444,-13);}      /* 42,255,0 */
```

Может показаться, что преобразования между процентными значениями и целыми числами выполняются произвольным образом, но на самом деле для их вычисления существует простая формула. Если известно процентное значение каждого из уровней RGB, то для того чтобы получить окончательные значения, достаточно составить пропорцию с числом 255. Скажем, есть цвет, содержащий 25% красного,

37,5% зеленого и 60% синего. Умножьте каждое из этих процентных значений на 255 и разделите результат на 100, после чего получите 63,75; 95,625 и 153. Затем округлите числа до ближайших целых: `rgb(64, 96, 153)`.

Конечно, если процентные значения известны, то не составит труда преобразовать их в целые. Запись в виде целых удобнее тем, кто работает с такими программами, как Photoshop, отображающими в диалоговом окне «Info» целые значения, или тем, кто настолько близко знаком с деталями формирования цветов, что привык «мыслить» числами 0–255. В последнем случае, вероятно, удобнее оперировать шестнадцатеричными числами, что мы и обсудим в следующем разделе.

Шестнадцатеричные RGB-цвета

CSS позволяет определять цвет с помощью шестнадцатеричной записи, такой близкой авторам веб-страниц, работающим с HTML:

```
h1 {color: #FF0000;}   /* делаем заголовки H1 красными */
h2 {color: #903BC0;}   /* делаем заголовки H2 темно-фиолетовыми */
h3 {color: #000000;}   /* делаем заголовки H3 черными */
h4 {color: #808080;}   /* делаем заголовки H4 умеренно-серыми */
```

Шестнадцатеричная форма записи («hex notation») применяется в компьютерном мире уже довольно давно, и программисты обычно знают ее. Такая распространенность шестнадцатеричной системы счисления, возможно, и послужила причиной ее использования при задании цветов в HTML старой школы. Эта практика была просто перенесена в CSS.

Вот как это делается: цвет задается посредством объединения трех шестнадцатеричных чисел в диапазоне от `00` до `FF`. Обобщенный синтаксис для этой формы записи – `#RRGGBB`. Обратите внимание, что здесь между числами нет ни пробелов, ни запятых, ни каких-либо других разделителей.

Шестнадцатеричная форма записи с математической точки зрения эквивалентна записи целыми числами, обсуждаемой в предыдущем разделе. Например, запись `rgb(255, 255, 255)` эквивалентна `#FFFFFF`, а `rgb(51, 102, 128)` – это то же самое, что и `#336680`. Выбирайте любую форму записи, в большинстве агентов пользователя они интерпретируются одинаково. При наличии калькулятора, умеющего осуществлять преобразования между десятичными и шестнадцатеричными числами, переход от одних к другим не составит труда.

Для шестнадцатеричных чисел, составленных из трех согласованных пар символов, CSS допускает более короткую запись. Обобщенный синтаксис для этой формы записи – `#RGB`.

```
h1 {color: #000;}   /* делаем заголовки H1 черными */
h2 {color: #666;}   /* делаем заголовки H2 темно-серыми */
h3 {color: #FFF;}   /* делаем заголовки H3 белыми */
```

Как видите, в каждом значении цвета присутствует только три символа. Однако для записи шестнадцатеричных чисел в диапазоне от `00` до `FF` необходимо по два символа для каждого, а у вас всего три символа. Как же работает этот метод?

Секрет в том, что броузер удваивает каждый символ. Следовательно, запись `#F00` эквивалентна `#FF0000`, `#6FA` будет аналогична `#66FFAA`, а `#FFF` будет преобразована в `#FFFFFF`, что соответствует `white`. Очевидно, что не каждый цвет может быть представлен таким образом. Умеренно-серый, например, надо записывать в стандартной шестнадцатеричной форме как `#808080`. Это значение не может быть представлено в краткой форме, ближайший эквивалент – `#888`, что аналогично `#888888`.

Сводим вместе все формы записи цветов

В табл. 4.1 представлен обзор некоторых из рассмотренных нами цветов. Возможно, ключевые слова, обозначающие цвета, не будут распознаваться броузерами, и поэтому цвета должны задаваться с помощью RGB или шестнадцатеричных кодов (просто для надежности). Кроме того, для некоторых цветов здесь вообще не представлены укороченные шестнадцатеричные коды. В таких случаях длинные (шестиразрядные) коды не могут быть представлены в краткой форме, потому что символы в них не дублируются. Например, значение `#880` разворачивается в `#888800`, а не в `#808000` (в другом представлении – `olive`). Поэтому для кода `#808000` краткой версии не существует, и соответствующая ей запись таблицы пуста.

Таблица 4.1. Эквивалентные записи цветов

Цвет	Процентные соотношения	Числовая	Шестнадцатеричная	Краткая шестнадцатеричная
red	rgb(100%,0%,0%)	rgb(255,0,0)	#FF0000	#F00
orange	rgb(100%,40%,0%)	rgb(255,102,0)	#FF6600	#F60
yellow	rgb(100%,100%,0%)	rgb(255,255,0)	#FFFF00	#FF0
green	rgb(0%,50%,0%)	rgb(0,128,0)	#008000	
blue	rgb(0%,0%,100%)	rgb(0,0,255)	#0000FF	#00F
aqua	rgb(0%,100%,100%)	rgb(0,255,255)	#00FFFF	#0FF
black	rgb(0%,0%,0%)	rgb(0,0,0)	#000000	#000
fuchsia	rgb(100%,0%,100%)	rgb(255,0,255)	#FF00FF	#F0F
gray	rgb(50%,50%,50%)	rgb(128,128,128)	#808080	
lime	rgb(0%,100%,0%)	rgb(0,255,0)	#00FF00	#0F0
maroon	rgb(50%,0%,0%)	rgb(128,0,0)	#800000	
navy	rgb(0%,0%,50%)	rgb(0,0,128)	#000080	
olive	rgb(50%,50%,0%)	rgb(128,128,0)	#808000	

Таблица 4.1 (продолжение)

Цвет	Процентные соотношения	Числовая	Шестнадцатеричная	Краткая шестнадцатеричная
purple	rgb(50%, 0%, 50%)	rgb(128, 0, 128)	#800080	
silver	rgb(75%, 75%, 75%)	rgb(192, 192, 192)	#C0C0C0	
teal	rgb(0%, 50%, 50%)	rgb(0, 128, 128)	#008080	
white	rgb(100%, 100%, 100%)	rgb(255, 255, 255)	#FFFFFF	#FFF

Цвета безопасной веб-палитры

Цвета безопасной веб-палитры («web-safe» colors) – это такие цвета, которые обычно не подлежат смешиванию в 256-цветных компьютерных системах. Цвета безопасной веб-палитры могут быть выражены RGB-кодами, кратными 20%, или 51, или шестнадцатеричному коду 33. К безопасным кодам также относятся 0% или 0. Таким образом, если RGB-цвета задаются в виде процентных соотношений, то все коды должны или содержать 0%, или быть числом, кратным 20, например rgb(40%, 100%, 80%) или rgb(60%, 0%, 0%). Если используются RGB-коды по шкале 0–255, то значения должны быть или равны 0, или кратны 51, как в rgb(0, 204, 153) или rgb(255, 0, 102).

При шестнадцатеричной форме записи любой триплет с кодами 00, 33, 66, 99, CC и FF считается принадлежащим к безопасной веб-палитре, например #669933, #00CC66 и #FF00FF. Это значит, что безопасными укороченными шестнадцатеричными кодами являются 0, 3, 6, 9, C и F; поэтому #693, #0C6 и #F0F – примеры цветов из безопасной веб-палитры.

Единицы измерения длины

Многие свойства CSS, такие как отступы, зависят от мер длины для обеспечения правильного представления различных элементов страниц. Поэтому неудивительно, что в CSS для измерения длины существует множество способов.

Все единицы измерения длины могут быть представлены как положительные или отрицательные числа (хотя некоторые свойства будут принимать только положительные значения), непосредственно за которыми следует обозначение единицы измерения. Числа могут быть и вещественными, т. е. содержать дробную часть, например 10,5 или 4,561. Все значения длины сопровождаются двухбуквенной аббревиатурой, представляющей единицы измерения, например in (дюймы) или pt (пункты). Единственное исключение из этого правила – нулевая длина (0), для которой не надо указывать единицы измерения.

Различают два типа единиц измерения длины: *абсолютные единицы измерения* (*absolute length units*) и *относительные* (*relative length units*).

Абсолютные единицы измерения длины

Начнем с абсолютных единиц измерения, потому что они проще для понимания, несмотря на тот факт, что они практически не применяются в разработке веб-страниц. Вот пять типов абсолютных единиц измерения:

Дюймы (`in`)

Как вы, возможно, поняли, это условное обозначение дюймов на линейках, с которыми имеют дело жители США. (Тот факт, что эти единицы измерения попали в спецификацию, несмотря на то, что практически весь мир пользуется метрической системой, может рассматриваться как свидетельство распространения интересов США на Интернет. Но не будем углубляться в виртуальную социально-политическую теорию.)

Сантиметры (`cm`)

Обозначает сантиметры, которые можно найти на линейках по всему миру. В одном дюйме 2,54 сантиметра, а один сантиметр равен 0,394 дюйма.

Миллиметры (`mm`)

Для тех американцев, которым тяжело привыкнуть к метрической системе, сообщаю, что в сантиметре 10 миллиметров, следовательно, один дюйм равен 25,4 миллиметра, а миллиметр равен 0,0394 дюйма.

Пункты (`pt`)

Пункты – это стандартная типографская единица измерения, которая десятилетиями используется в принтерах, наборных машинах и в программах обработки текстов. Традиционно дюйму соответствуют 72 пункта (пункты появились еще до широкого распространения метрической системы). Следовательно, 12 пунктов соответствуют одной шестой дюйма. Например, запись `p {font-size: 18pt;}` эквивалентна `p {font-size: 0.25in;}`.

Пики (`pc`)

Пики – это еще один типографский термин. Пика эквивалентна 12 пунктам, т. е. в дюйме 6 пик. А одна пика равна одной шестой доле дюйма. Например, выражение `p {font-size: 1.5pc;}` задает такую же высоту текста, как и объявления, приведенные выше.

Конечно, с этими единицами измерения удобно работать, только если броузер знает все характеристики монитора, на котором отображается страница, принтера или любого другого агента пользователя, который может быть применен. В веб-броузере представление зависит от размера монитора и разрешения, установленного на нем. И вы как автор не можете влиять на эти факторы. Вы можете только надеяться, что размеры будут хотя бы согласованными, т. е. что длина в `1.0in` будет в два раза больше `0.5in`, как показано на рис. 4.3.

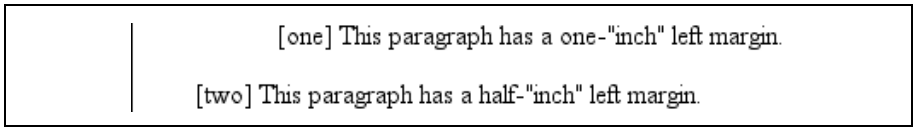


Рис. 4.3. *Задание левого отступа в абсолютных единицах длины*

Работа с абсолютными длинами

Если разрешение экрана монитора составляет 1024 пиксела в ширину и 768 пикселов в высоту, размеры экрана составляют 14,22 дюйма в ширину и 10,67 дюйма в высоту, а область отображения занимает весь экран монитора, то размеры каждого пиксела составят $^{1}/_{72}$ дюйма в ширину и высоту. Как вы, возможно, догадались, вероятность реализации этого сценария очень и очень мала (вы когда-нибудь видели монитор с такими размерами?). Таким образом, в большинстве мониторов реальное количество пикселов на дюйм (ppi – pixels per inch) превышает 72, причем иногда существенно, достигая 120 ppi и более.

В Windows драйвер видеоадаптера можно настроить таким образом, чтобы отображение элементов соответствовало реальным размерам. Для этого нажмите Пуск (Start)→Настройки (Settings)→Панель управления (Control Panel). В панели управления дважды щелкните по значку Экран (Display). Выберите вкладку Параметры (Settings) и нажмите Дополнительно (Advanced), чтобы открыть диалоговое окно (на разных ПК оно может быть разным). В разделе Размер шрифта (Font Size) выберите Другие (Other) и затем, приложив линейку к экрану, перемещайте бегунок до тех пор, пока деления на экранной линейке не совпадут с делениями на физической. Щелкайте по кнопке OK до тех пор, пока не закроете все диалоговые окна. Вот вы и произвели настройку.

В операционной системе, под управлением которой работают Макинтоши, нет возможности произвести такую настройку. В Mac Classic OS (т. е. любой версии до OS X) задано соотношение между экранными пикселами и абсолютными размерами: предполагается, что монитор имеет разрешающую способность 72 пиксела на дюйм. Это предположение совершенно неверно, но оно встроено в операционную систему и поэтому практически неустранимо. В результате во многих веб-броузерах, работающих на Classic Mac, любое заданное в пунктах значение будет эквивалентно такой же длине в пикселах: текст размером в `24pt` будет в высоту составлять 24 пиксела, и текст размером в `8pt` будет 8 пикселов в высоту. К сожалению, это слишком мало, и текст такого размера разобрать невозможно (рис. 4.4).

В OS X принято более близкое к Windows значение ppi: 96ppi. Это ничуть не более правильно, но по крайней мере приемлемо для компьютеров, работающих под Windows.

Classic Mac – замечательная иллюстрация необходимости избегать применения пунктов при разработке для Всемирной паутины. «Эмы» (em),

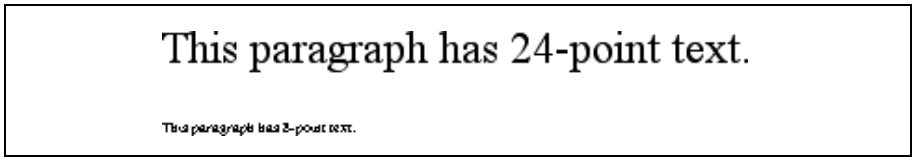


Рис. 4.4. *Очень мелкий текст очень трудно читать*

процентные соотношения и даже пикселы – все эти единицы более предпочтительны, чем пункты, когда дело касается отображения в броузере.

Начиная с Internet Explorer 5 для Macintosh и таких, основанных на технологии Gecko, броузерах, как Netscape 6+, сам броузер имеет настройки для установки значения ppi. Можно выбрать стандартное для Макинтоша соотношение 72ppi, обычное соотношение для Windows в 96ppi или значение, которое соответствует соотношению ppi вашего монитора. Последний вариант аналогичен описанной выше схеме настройки в Windows, где для сравнения с линейкой применяется подвижная шкала, благодаря чему удается получить точное соответствие размеров, отображаемых вашим монитором, и реальных физических размеров.

Теперь абстрагируемся от практических реализаций и сделаем весьма сомнительное предположение о том, что ваш компьютер знает достаточно о своей системе отображения, чтобы точно воспроизводить реальные размеры. В таком случае, объявив `p {margin-top: 0.5in;}`, можно было бы гарантировать, что верхний отступ каждого абзаца будет составлять половину дюйма. Независимо от размера шрифта или любых других обстоятельств отступ в верхней части абзаца будет равен половине дюйма.

Абсолютные единицы измерения намного удобнее при определении таблиц стилей для печатных документов, где измерения в дюймах, пунктах и пиках – обычное дело. Как вы увидели, пытаться применять абсолютные измерения в веб-разработке в лучшем случае рискованно, так что давайте вернемся к более полезным единицам измерения.

Относительные единицы измерения длины

Относительные единицы измерения получили такое название потому, что они измеряются относительно других единиц измерения. Измеряемое ими фактическое (или абсолютное) расстояние может меняться под действием внешних факторов, таких как разрешение экрана, ширина области просмотра, предпочтительные настройки пользователя и масса других параметров. Кроме того, для некоторых относительных единиц измерения их размер практически всегда измеряется относительно использующего их элемента и соответственно будет меняться от элемента к элементу.

Существует три относительных единицы измерения длины: `em`, `ex` и `px`. Первые две обозначают «em-высоту» и «x-высоту» и представляют со-

бой обычные типографские меры длины, но те, кто знаком с типографией, заметят, что в CSS эти единицы обрели неожиданный смысл. Последний тип единиц измерения длины – px, что обозначает «пикселы». Пиксел – это одна из точек, которые можно увидеть на мониторе компьютера, если внимательно присмотреться. Это значение определено как относительное, потому что зависит от разрешения устройства отображения. Данного вопроса мы скоро коснемся.

Единицы измерения em и ex

Сначала, однако, рассмотрим em и ex. В CSS один «em» – это значение свойства font-size заданного шрифта. Если для элемента font-size равен 14 пикселам, тогда для него же 1em равен 14 пикселам.

Очевидно, что это значение может меняться от элемента к элементу. Например, возьмем h1, размер шрифта которого составляет 24 пиксела, элемент h2, размер шрифта которого составляет 18 пикселов, и абзац со шрифтом в 12 пикселов. Если задать левый отступ в 1em для всех трех элементов, то отступ слева для них будет 24, 18 и 12 пикселов соответственно:

```
h1 {font-size: 24px;}
h2 {font-size: 18px;}
p {font-size: 12px;}
h1, h2, p {margin-left: 1em;}
small {font-size: 0.8em;}

<h1>Left margin = <small>24 pixels</small></h1>
<h2>Left margin = <small>18 pixels</small></h2>
<p>Left margin = <small>12 pixels</small></p>
```

При задании размера шрифта значение em вычисляется относительно размера шрифта родительского элемента, как показано на рис. 4.5.

С другой стороны, величина ex опирается на высоту буквы x нижнего регистра выбранного шрифта. Поэтому, если в двух абзацах размер текста составляет 24 пункта, но для каждого абзаца выбран свой шрифт, то значение ex для каждого из них может быть различным. Дело в том, что высота буквы «x» в разных шрифтах разная, что показано на рис. 4.6. В примерах текст имеет высоту 24 пункта, и, следовательно, значение em каждого примера составляет 24 пункта, но x-высота (высота глифов строчных букв) каждого из них разная.

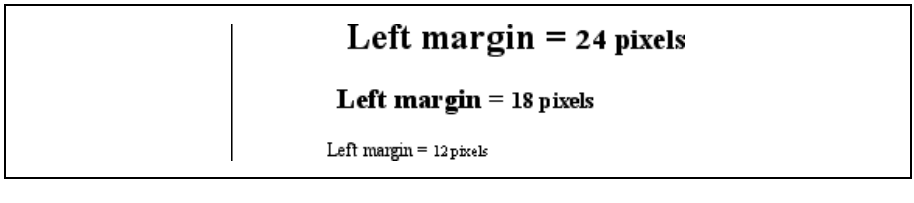


Рис. 4.5. *Применение единицы измерения длины* em *для определения размеров отступов и шрифтов*

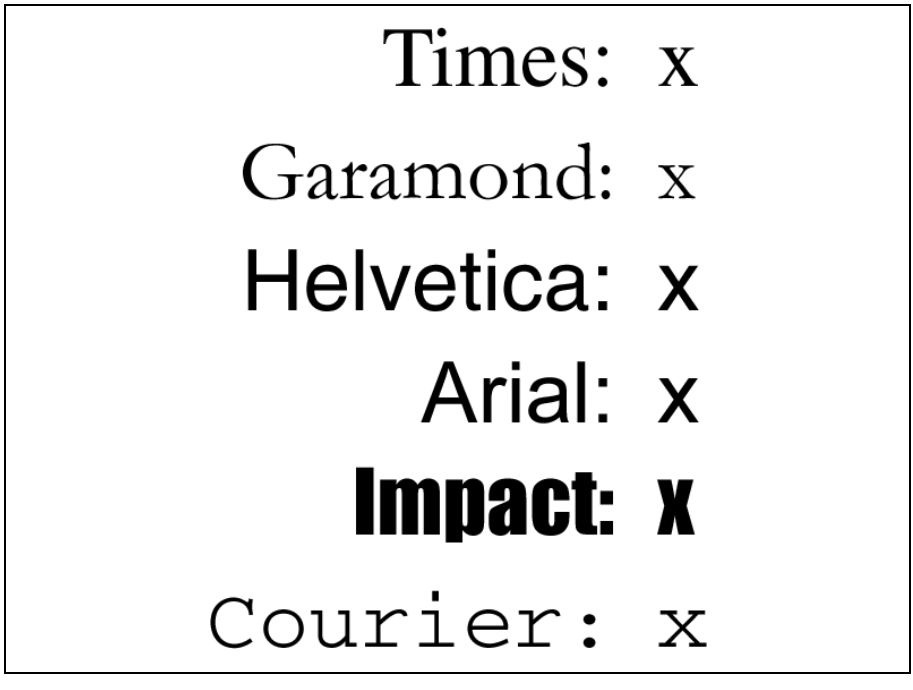


Рис. 4.6. *Разные x-высоты*

Практические трудности с применением em и ex

Конечно, все, о чем я только что говорил, сплошная теория. Я показал, что *должно* происходить, но на практике многие агенты пользователя вычисляют значение `ex` путем деления значения `em` на два. Почему? Большинство шрифтов не имеют встроенного значения высоты `ex`, и вычислить его очень сложно. Поскольку строчные буквы многих шрифтов примерно вполовину ниже прописных, удобно считать, что `1ex` эквивалентен `0.5em`.

Некоторые броузеры, включая Internet Explorer 5 для Mac OS, на самом деле пытаются определять x-высоту заданного шрифта, генерируя внутри строчную букву «x» и определяя ее высоту в пикселах, чтобы сравнить со значением свойства `font-size`, используемым для создания этого символа. Не самый идеальный метод, но это намного лучше, чем просто приравнивать `1ex` к `0.5em`. Мы, специалисты-практики CSS, можем надеяться, что со временем количество агентов пользователя, работающих с реальными значениями `ex`, возрастет и упомянутое выше упрощение с половиной `em` канет в лету.

Измерение длин в пикселах

На первый взгляд с пикселами все довольно просто. Если присмотреться к монитору достаточно пристально, то можно увидеть, что его

изображение представляет собой сетку из крошечных точек. Каждая точка – это пиксел. Определим элемент, состоящий из некоторого количества пикселов в ширину и высоту, как в следующей разметке:

```
<p>
The following image is 20 pixels tall and wide: <img src="test.gif"
  style="width: 20px; height: 20px;" alt="" />
</p>
```

Он будет занимать по высоте и ширине именно это количество точек на экране монитора (рис 4.7).

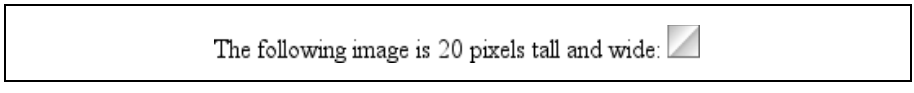


***Рис. 4.7.** Задаем длину в пикселах*

К сожалению, в применении пикселов есть потенциальный недостаток. Если размер шрифта задается в пикселах, то пользователи Internet Explorer для Windows до версии IE7 не смогут изменять размер текста с помощью команды меню Text Size своего броузера. Это неудобство может стать существенным, если текст слишком мелкий и его неудобно читать. Если выбрать более гибкие единицы измерения, например `em`, то пользователь сможет менять размер текста. (Те, кто очень трепетно относятся к своему дизайну, могут, конечно, назвать *это* недостатком.)

С другой стороны, измерения в пикселах идеальны для задания размеров изображений, которые уже представлены определенным количеством пикселов в ширину и высоту. Кстати, единственный случай, при котором окажется неудобным выражать размер изображения в пикселах, это если вам требуется масштабировать пикселы относительно некоторого текста. Этот превосходный и подчас очень полезный подход сможет предоставить реальные преимущества при работе с векторными, а не с растровыми изображениями. (С переходом к масштабируемой векторной графике Scalable Vector Graphics он получит большее распространение в будущем.)

Теория пикселов

Так почему же все-таки пикселы определены как относительные единицы измерения длины? Я говорил, что пикселы – это крошечные цветные точки на экране монитора. Но сколько таких точек в дюйме? Это может показаться нелогичным, но немного терпения.

При рассмотрении пикселов спецификация CSS рекомендует агентам пользователя в случаях, когда разрешающая способность устройства отображения значительно отличается от 96 ppi, изменить масштаб пиксела на эталонную величину. В CSS2 эталонная величина пиксела равна 90 ppi, а в CSS2.1 предлагается 96 ppi – величина, свойственная

компьютерам под управлением Windows и принятая современными броузерами для Макинтоша, такими как Safari.

В общем, если объявлено что-то вроде `font-size: 18px`, веб-броузер практически наверняка возьмет размер пикселов на мониторе (в конце концов, они уже там), но для других устройств отображения, таких как принтеры, агенту пользователя придется менять масштаб длин, выраженных в пикселах, на что-то более удобное. Иначе говоря, программа, отвечающая за вывод на печать, должна определить, сколько точек содержится в пикселе, и для этого она может взять в качестве эталона величину 96 ppi.

Один из примеров трудностей, связанных с определением размера пиксела, можно найти в ранней реализации CSS1. Internet Explorer 3.x при распечатке документа предполагал, что `18px` – это то же самое, что и 18 точек, что на принтере с разрешением 600dpi составляет 18/600 или 3/100 дюйма, или, если вам так больше нравится, `0.03in`. Для текста это совсем маленький размер!

Из-за этой потенциальной возможности масштабирования пикселы определены как относительные единицы измерения, даже несмотря на то, что в веб-разработке они ведут себя аналогично абсолютным единицам.

Что делать?

Имея в виду все связанные с этим вопросом трудности, лучше всего, вероятно, использовать относительные единицы измерения, в первую очередь, `em` и при необходимости – `px`. Поскольку `ex` в большинстве применяемых сейчас броузеров, как правило, вычисляется как часть `em`, ее достоинства в настоящий момент обсуждать не будем. Если реальное измерение x-высоты будут поддерживать большее количество агентов пользователя, возможно, `ex` возьмет свое. В общем, `em` является более гибкой единицей измерения, потому что соотнесена с размером шрифта, так что относительное расположение элементов будет оставаться постоянным.

В ряде случаев при работе с элементами удобнее иметь дело с пикселами, например, для задания ширины рамки или позиционирования элементов. Все зависит от ситуации. Например, в проектах, где по традиции фрагменты дизайна друг от друга отделяются при помощи GIF-изображений, применение отступов, длина которых выражена в пикселах, имеет аналогичный результат. При преобразовании этих величин в `em` они бы увеличивались или уменьшались в соответствии с изменениями размера текста, что иногда хорошо, а иногда – нет.

URL

Те, у кого есть опыт создания веб-страниц, наверняка хорошо знакомы с URL (или в CSS2.1 – URI). Общий формат обращения к нужному объ-

екту, например, в операторе `@import`, применяемом при импортировании внешней таблицы стилей, таков:

```
url(protocol://server/pathname)
```

Приведенный выше пример определяет то, что известно как *абсолютный URL* (*absolute URL*). Под абсолютным я подразумеваю URL, который будет работать везде, независимо от того, где (или вернее, на какой странице) он находится, потому что он определяет абсолютное месторасположение в веб-пространстве. Скажем, есть сервер *www.waffles.org*. На этом сервере есть каталог под названием `pix`, и в этом каталоге есть файл изображения *waffle22.gif*. В этом случае абсолютный URL этого файла будет записан так:

```
http://www.waffles.org/pix/waffle22.gif
```

Этот URL будет действительным независимо от того, располагается ли содержащая данный URL страница на сервере *www.waffles.org* или *web.pancakes.com*.

Другой тип URL – *относительный* URL, названный так потому, что он определяет местоположение относительно содержащего этот URL документа. Если вы ссылаетесь на относительное местоположение, например файл, находящийся в том же каталоге, что и ваша веб-страница, тогда общий формат такой:

```
url(pathname)
```

Эта форма записи годится, только если файл находится на том же сервере, что и страница, содержащая URL. Для примера представим себе, что веб-страница расположена по адресу *http://www.waffles.org/syrup.html*, и на этой странице необходимо показать изображение *waffle22.gif*. Для этого случая URL будет таким:

```
pix/waffle22.gif
```

Этот путь действителен, потому что веб-броузер знает, что он должен начать поиск с места, где находится веб-документ, и затем добавить относительный URL. В этом случае путь *pix/waffle22.gif* добавляется к имени сервера *http://www.waffles.org/*, что аналогично *http://www.waffles.org/pix/waffle22.gif*. Практически во всех ситуациях можно применять абсолютный URL вместо относительного. Не имеет значения, какой из URL выбран, если он правильно определяет местоположение.

В CSS относительные URL определяются относительно положения таблицы стилей, а не HTML-документа, который ее использует. Например, у вас есть внешняя таблица стилей, которая импортирует другую таблицу стилей. Если для импорта второй таблицы стилей вы используете относительный URL, он должен быть задан относительно первой таблицы стилей. В качестве примера рассмотрим HTML-документ, находящийся по адресу *http://www.waffles.org/toppings/tips.html*, в котором есть ссылка на таблицу стилей *http://www.waffles.org/styles/basic.css*:

```
<link rel="stylesheet" type="text/css"
    href="http://www.waffles.org/styles/basic.css">
```

В файле *basic.css* находится инструкция `@import`, ссылающаяся на другую таблицу стилей:

```
@import url(special/toppings.css);
```

Эта инструкция `@import` заставит броузер искать таблицу стилей по адресу *http://www.waffles.org/styles/special/toppings.css*, а не *http://www.waffles.org/toppings/special/toppings.css*. Если таблица стилей расположена по второму адресу, то директива `@import` в файле *basic.css* должна выглядеть так:

```
@import url(http://www.waffles.org/toppings/special/toppings.css);
```

Netscape Navigator 4 интерпретирует относительные URL относительно HTML-документа, а не таблицы стилей. Если на вашу веб-страницу заходит много посетителей с NN4.x или вы хотите гарантировать, что NN4.x сможет найти все ваши таблицы стилей и фоновые изображения, проще сделать все URL абсолютными, поскольку их Navigator обрабатывает правильно.

Обратите внимание, что между `url` и открывающей круглой скобкой не должно быть пробела:

```
body {background: url(http://www.pix.web/picture1.jpg);}   /* верно */
body {background: url  (images/picture2.jpg);}           /* НЕВЕРНО */
```

Если вставить здесь пробел, все объявление будет признано недействительным и, следовательно, проигнорировано.

Ключевые слова

Ключевые слова существуют с тех пор, как в качестве значений используются некоторые определенные слова. Очень распространенный пример – ключевое слово `none`, отличающееся от `0` (нуля). Так, чтобы удалить подчеркивание ссылок в HTML-документе, можно написать:

```
a:link, a:visited {text-decoration: none;}
```

Аналогично, если бы потребовалось подчеркнуть ссылки, можно было бы указать ключевое слово `underline`.

Если свойство допускает применение ключевых слов, то его ключевые слова определены только для этого свойства. Если одно и то же слово задано как ключевое для двух свойств, то действие ключевого слова для одного свойства никак не будет связано с его действием в рамках другого свойства. В качестве примера возьмем слово `normal`. Определенное для свойства `letter-spacing`, оно означает нечто совершенно отличное от того, что задает `normal` для свойства `font-style`.

Ключевое слово inherit

Существует одно ключевое слово, которое используется всеми свойствами CSS2.1: `inherit`. Оно делает значение свойства таким же, как и у его родительского элемента. В большинстве случаев не надо задавать наследование, потому что большая часть свойств реализует его по умолчанию; однако `inherit` может быть крайне полезным.

Например, рассмотрим следующие стили и разметку:

```
#toolbar {background: blue; color: white;}

<div id="toolbar">
<a href="one.html">Один</a> | <a href="two.html">Два</a> |
<a href="three.html">Три</a>
</div>
```

Сам элемент `div` будет белым на синем фоне, а вот ссылки будут оформлены согласно предпочтительным настройкам броузера. Скорее всего, это будет синий текст на синем фоне с белыми вертикальными полосками между ними.

Можно было бы написать правило, явно задающее белый цвет текста ссылок панели инструментов, но можно обратиться к более надежному способу и применить `inherit`. Для этого достаточно добавить в таблицу стилей следующее правило:

```
#toolbar a {color: inherit;}
```

Это заставит ссылки руководствоваться унаследованным значением `color` вместо применяемых по умолчанию стилей агента пользователя. Обычно непосредственно назначенные стили замещают унаследованные стили, но `inherit` может изменить это поведение на обратное.

Единицы измерения CSS2

В добавление к рассмотренным в рамках CSS2.1, рассмотрим несколько дополнительных единиц измерений, которые содержит CSS2. Все они имеют отношение к звуковым таблицам стилей (применяемым теми броузерами, которые могут воспроизводить речевой сигнал). Эти единицы измерения не были включены в CSS2.1, но поскольку они могут стать частью будущих версий CSS, мы кратко обсудим их здесь:

Угловые величины

Предназначены для определения местоположения, из которого должен исходить данный звук. Существует три типа единиц измерения углов: градусы (`deg`), грады (`grad`) и радианы (`rad`). Например, прямой угол может быть задан как `90deg`, `100grad` или `1.57rad`; в любом случае значения переводятся в градусы в диапазоне от 0 до 360. Это также выполняется и для отрицательных значений, которые допускаются. Величина `-90deg` эквивалентна `270deg`.

Значения времени

Определяют задержки между элементами речи. Они могут быть выражены или в миллисекундах (`ms`), или в секундах (`s`). Таким образом, значения `100ms` и `0.1s` эквивалентны. Значения времени не могут быть отрицательными, поскольку CSS не предполагает создания парадоксов.

Значения частот

Служат для задания частоты звуков, генерируемых броузером с возможностью синтеза речи. Значения частот могут быть выражены в герцах (`Hz`) или мегагерцах (`MHz`) и не могут быть отрицательными. Сокращения единиц измерения частот нечувствительны к регистру, поэтому записи `10MHz` и `10mhz` эквивалентны.

На момент написания данной книги единственным известным агентом пользователя, поддерживающим эти значения, является реализация звуковых таблиц стилей *Emacspeak*. Звуковые стили подробно рассмотрены в главе 14.

Кроме этих значений существуют также старые добрые друзья под новыми именами. *URI* (*Uniform Resource Identifier – универсальный идентификатор ресурса*) является разновидностью другого имени – URL (Uniform Resource Locator – универсальный адрес ресурса). Спецификации CSS2 и CSS2.1 требуют, чтобы URI объявлялись в форме `url(...)`, так что в этом практически ничего не изменилось.

Заключение

Единицы измерения и значения охватывают очень широкий спектр понятий: от единиц измерения длин до специальных ключевых слов, которые описывают эффекты (такие как `underline` (подчеркивать)), от названий цветов до местоположений файлов (таких как изображения). Большей частью, единицы измерения – это та область, которую агенты пользователя обрабатывают практически полностью правильно, за исключением нескольких небольших ошибок и индивидуальных особенностей. Например, неспособность Navigator 4.x правильно интерпретировать относительные URL измучила многих авторов и привела к повсеместному переходу к абсолютным URL. Цвета – это еще одна сфера, где агенты пользователя практически всегда ведут себя хорошо, кроме нескольких небольших случайностей, происходящих то тут, то там. Однако причуды единиц измерения длин далеки от того, чтобы считать их простыми ошибками, это интересная проблема, над которой должен потрудиться каждый автор.

Все эти единицы измерения имеют свои преимущества и недостатки в зависимости от обстоятельств, в которых они применяются. Мы уже видели некоторые из этих обстоятельств, а нюансы будут обсуждаться далее, начиная с CSS-свойств, которые определяют способы изменения отображения текста.

5

Шрифт

Как точно заметили авторы спецификации CSS, выбор шрифта является самой популярной (и значимой) возможностью. В конце концов, сколько страниц забито десятками или даже сотнями тегов <FONT FACE="...">? Кстати, раздел спецификации «Свойства шрифтов» начинается со слов: «Таблицы стилей наиболее широко будут использоваться для настройки свойств шрифтов».

Несмотря на исключительную важность, на настоящий момент возможности обеспечения единообразного использования шрифтов в Веб не существует, потому что нет унифицированного способа описания шрифтов и их разновидностей. Например, шрифты Times, Times New Roman и TimesNR, возможно, подобны или даже одинаковы, но как об этом может знать агент пользователя? Автор мог бы задать в документе «TimesNR», но что если пользователь просматривает документ, не установив этот конкретный шрифт? Даже если установлен Times New Roman, агент пользователя никак не может знать, что эти два шрифта взаимозаменяемы. И если вы надеетесь поразить читателя определенным шрифтом, забудьте об этом.

Хотя в CSS2 определены средства для загрузки шрифтов, они не очень хорошо реализованы веб-броузерами, и читатель всегда может отказаться от загрузки шрифтов по соображениям производительности. CSS *не* предоставляет реального контроля над шрифтами, во всяком случае не более, чем текстовый редактор, – когда кто-то загружает документ Microsoft Office, его представление зависит от шрифтов, установленных на компьютере. Если на нем нет таких же шрифтов, как у автора документа, документ будет выглядеть иначе. Это происходит и с документами, разработанными с применением CSS.

Проблема присваивания имен шрифтам становится особенно острой, когда дело доходит до начертаний шрифтов (например, полужирный или курсив). Большинство людей знает, как выглядит курсив, но

лишь некоторые могут описать, чем он отличается от наклонного (slanted) текста, даже несмотря на то, что отличия есть. «Slanted» – это не просто одно из названий выделенного курсивом текста, который также может обозначаться как *oblique*, *incline* (или *inclined*), *cursive* и *kursiv*. Таким образом, один шрифт может иметь вариант, названный, например, TimesItalic, тогда как кто-то другой выберет что-то вроде GeorgiaOblique. Хотя эти два шрифта могут быть полностью эквивалентными как «курсивная форма» этих шрифтов, названы они совершенно по-разному. Аналогично названия вариантов шрифтов *bold*, *black* и *heavy* могут означать одно и то же, а могут и нет.

CSS делает попытки предоставить некоторые механизмы разрешения всех этих вопросов со шрифтами, хотя и не может полностью решить проблему. Самые сложные этапы обработки шрифтов в CSS – это сопоставление семейств шрифтов и сопоставление толщины линий шрифтов, а также вычисление размера шрифтов. Спецификация CSS обращается к таким аспектам шрифтов, как их начертание, например курсив, и видоизменения, например капитель. Различные аспекты стилевого оформления шрифтов сведены воедино в свойстве `font`, которое мы будем обсуждать в этой главе позже. Сначала рассмотрим семейства шрифтов, поскольку это понятие является базовым при правильном выборе шрифта для документа.

Семейства шрифтов

Как мы уже говорили, существует несколько способов определить, что же на самом деле есть один и тот же шрифт. CSS делает смелую попытку помочь агентам пользователя справиться с этой неразберихой. В конце концов, то, что мы подразумеваем под словом «шрифт», может быть образовано множеством его вариантов для описания полужирного, курсивного начертания текста и т. д. Например, вы, вероятно, хорошо знакомы со шрифтами Times. Однако Times на самом деле является сочетанием многих вариантов, включая TimesRegular, TimesBold, TimesItalic, TimesOblique, TimesBoldItalic, TimesBoldOblique и т. д. Каждый из этих вариантов шрифта Times является *гарнитурой шрифта* (*font face*), а Times, как мы обычно полагаем, есть сочетание всех этих гарнитур. Иначе говоря, Times на самом деле представляет собой *семейство шрифтов* (*font family*), а не отдельный шрифт, даже несмотря на то, что большинство из нас рассматривает шрифты как отдельные сущности.

В дополнение к каждому отдельному семейству шрифтов, например Times, Verdana, Helvetica или Arial, CSS определяет пять базовых семейств шрифтов:

Шрифты с засечками, или антиква (Serif)

Это пропорциональные шрифты, имеющие засечки. Шрифт считается пропорциональным, если все его символы имеют различную

ширину из-за разницы в их размерах (ширина строчных букв `i` и `m` различна). Например, шрифт абзаца этой книги является пропорциональным. Засечки – это украшения на концах линий каждого символа, например черточки сверху и снизу строчной буквы *l* или в основании каждой из «ножек» прописной *A*. К шрифтам с засечками относятся, в частности, Times, Georgia и New Century Schoolbook.

Рубленые шрифты, или гротески (Sans-serif)

Это пропорциональные шрифты без засечек. Примеры – Helvetica, Geneva, Verdana, Arial и Univers.

Моноширинные (Monospace) шрифты

Моноширинные шрифты – непропорциональны. Они обычно применяются для имитации машинописного текста, распечаток на старых матричных принтерах и древних видеотерминалах. В этих шрифтах абсолютно все символы имеют одинаковую ширину, поэтому ширина строчной буквы *i* такая же, как и у строчной *m*. Эти шрифты могут иметь, а могут и не иметь засечки. Если ширина символов шрифта постоянна, он классифицируется как моноширинный независимо от наличия засечек. Примеры моноширинных шрифтов: Courier, Courier New и Andale Mono.

Рукописные (Cursive) шрифты

Эти шрифты представляют собой попытку имитации человеческого почерка. Обычно они состоят преимущественно из кривых и украшены штрихами больше, чем шрифты антиква. Например, у прописной `A` мог бы присутствовать небольшой завиток в нижней части левой «ножки» или вся она могла бы состоять из наклонных линий и завитков. Примеры рукописных шрифтов: Zapf Chancery, Author и Comic Sans.

Аллегорические шрифты

Подобные шрифты нельзя охарактеризовать как-либо более или менее определенно, можно лишь констатировать невозможность отнести их к одному из известных семейств шрифтов. Это, например, Western, Woodblock и Klingon.

Теоретически любое семейство шрифтов, которое мог бы установить пользователь, будет принадлежать одному из этих базовых семейств. На самом деле это не обязательно, но исключения (если таковые возникнут), скорее всего, будут немногочисленны.

Работа с базовыми семействами шрифтов

Шрифты любого из упомянутых семейств могут быть использованы в документе посредством свойства `font-family`.

Если требуется использовать в документе любой рубленый шрифт, подойдет следующее объявление:

```
body {font-family: sans-serif;}
```

font-family	
Значения:	[[<имя-семейства> | <базовое-семейство>],]* [<имя-семейства> | <базовое-семейство>] | inherit
Начальное значение:	зависит от агента пользователя
Область применения:	все элементы
Наследование:	да
Вычисляемое значение:	как задано

Это заставит агент пользователя выбрать семейство рубленых шрифтов (например, Helvetica) и применить его к элементу `body`. Благодаря наследованию такой же шрифт будет применяться ко всем элементам, происходящим от `body`, если только не будет переопределен более конкретным селектором.

Применяя только базовые семейства, автор может создавать достаточно изощренные таблицы стилей. Следующий набор правил проиллюстрирован на рис. 5.1:

```
body {font-family: serif;}
h1, h2, h3, h4 {font-family: sans-serif;}
code, pre, tt, span.input {font-family: monospace;}
p.signature {font-family: cursive;}
```

Таким образом, большая часть документа будет оформлена шрифтом с засечками, например Times, в том числе все абзацы, кроме тех, для которых задан класс `signature`: они будут отображаться рукописным

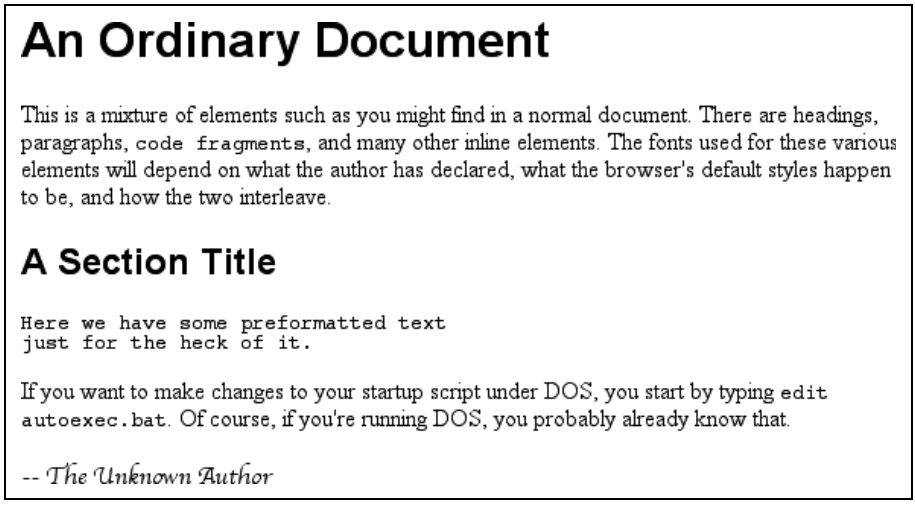


***Рис. 5.1.** Различные семейства шрифтов*

шрифтом, например Author. Заголовки h1–h4 будут набраны рубленым шрифтом, например Helvetica, тогда как элементы code, pre, tt и span.input будут отформатированы моноширинным шрифтом, таким как Courier, с помощью которого, кстати, большинство подобных элементов и представлено в данной книге.

Задание семейства шрифтов

У автора могут быть и более определенные предпочтения по поводу шрифтов в представлении документа или элемента. Да и пользователь может захотеть создать таблицу стилей, определяющую конкретные шрифты, применяемые в представлении всех документов. В любом случае все это также осуществляется с помощью свойства font-family.

Представим на мгновение, что все элементы h1 должны быть отформатированы шрифтом Georgia. Самое простое правило для этого могло бы быть таким:

```
h1 {font-family: Georgia;}
```

В результате агент пользователя будет отображать все элементы h1 данного документа шрифтом Georgia, как показано на рис. 5.2.

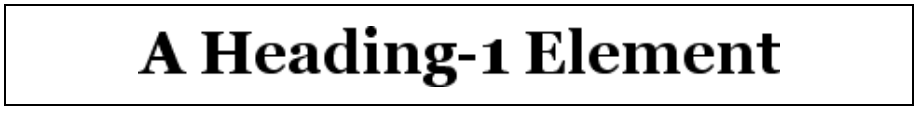


***Рис. 5.2.** Элемент h1, набранный шрифтом Georgia*

Конечно, это правило предполагает, что шрифт Georgia доступен агенту пользователя. Если нет, агент пользователя вообще не сможет применить правило. Он не проигнорирует его, но если не сможет найти шрифт под названием Georgia, то отобразит элементы h1 применяемым по умолчанию шрифтом агента пользователя.

Однако все не так плохо. Комбинируя конкретные шрифты с базовыми семействами, можно создавать документы, которые, если не в точности, то весьма близко соответствуют вашим намерениям. Продолжим предыдущий пример. Следующая разметка указывает, что агент пользователя должен применять шрифт Georgia, если он доступен, и другой шрифт с засечками, если Georgia недоступен.

```
h1 {font-family: Georgia, serif;}
```

Если у читателя не установлен шрифт Georgia, но есть шрифт Times, агент пользователя может применить Times ко всем элементам h1. Хотя шрифт Times не является точным аналогом шрифта Georgia, он достаточно близок.

По этой причине я убедительно рекомендую вам всегда указывать базовое семейство в любом правиле font-family. Этим вы реализуете механизм резервирования, обеспечивающий агентам пользователя воз-

можность выбора, когда точное соответствие шрифтов недостижимо. Такая защитная мера очень полезна, поскольку в межплатформенной среде невозможно знать, у кого какие шрифты установлены. Несомненно, на любой машине, работающей под Windows, установлены шрифты Arial и Times New Roman, но на некоторых компьютерах Макинтош (особенно старых) их нет, то же можно сказать и о системах Unix. И наоборот, тогда как шрифты MarkerFelt и Charcoal обычны для всех последних Макинтошей, вряд ли пользователи Windows и Unix имеют в своем распоряжении хотя бы один из них, не говоря уже об обоих. Вот еще несколько примеров:

```
h1 {font-family: Arial, sans-serif;}
h2 {font-family: Charcoal, sans-serif;}
p {font-family: TimesNR, serif;}
address {font-family: Chicago, sans-serif;}
```

Те, кто хорошо знаком со шрифтами, возможно, знают несколько аналогичных шрифтов, которые можно применить для отображения данного элемента. Скажем, все абзацы документа требуется представить шрифтом Times, но допускаются также шрифты TimesNR, Georgia, New Century Schoolbook и New York (все с засечками). Во-первых, определим порядок предпочтений для этих шрифтов, а затем выстроим их в список, разделяя запятыми:

```
p {font-family: Times, TimesNR, 'New Century Schoolbook', Georgia, 'New
York', serif;}
```

На основании этого списка агент пользователя будет искать шрифты в порядке их перечисления. Если ни один из приведенных шрифтов не найден, будет выбран один из доступных шрифтов антиква.

Использование кавычек

В предыдущем примере вы могли заметить одиночные кавычки, которые не встречались ранее. Кавычки нужны в объявлении свойства `font-family`, только если в имени шрифта имеются пробелы, как, например, в New York, или если имя шрифта включает такие символы, как # или $. В обоих случаях все имя шрифта должно быть взято в кавычки, чтобы агент пользователя мог правильно определить, где начинается и где кончается имя. (Может показаться, что достаточно было бы и запятых, но это не так.) Таким образом, шрифт под названием Karrank% должен быть взят в кавычки:

```
h2 {font-family: Wedgie, 'Karrank%', Klingon, fantasy;}
```

Если опустить кавычки, то агенты пользователя могут проигнорировать это имя шрифта, хотя все остальное правило будет обработано. Обратите внимание, что в спецификации CSS2.1 нет требования заключать в кавычки имя шрифта, содержащего символ. Но есть рекомендация, близкая к описанию «лучших практик». Точно так же рекомендуется брать в кавычки имя шрифта, содержащее пробелы. Как

выясняется, единственное обязательное требование состоит в том, что надо брать в кавычки имя шрифта, если оно совпадает с зарезервированными ключевыми словами. Таким образом, желая предусмотреть шрифт с именем «cursive», надо взять его в кавычки.

Очевидно, что имена шрифтов, состоящие из одного слова (которое не конфликтует ни с одним из ключевых слов `font-family`), не надо брать в кавычки, также никогда не берутся в кавычки базовые имена шрифтов (`serif`, `monospace` и т. д.), если они ссылаются на реальные базовые семейства. Если заключить в кавычки базовое имя, то агент пользователя сочтет, что указан конкретный шрифт с таким именем (например, «serif»), а не базовое семейство.

Что касается вида кавычек, то допускаются и одиночные, и двойные. Помните, что если правило `font-family` помещается в атрибут `style`, то кавычки должны отличаться от тех, в которые взято имя самого атрибута. Поэтому если правило `font-family` заключено в двойные кавычки, то в самом правиле кавычки должны быть одиночными, как в следующей разметке:

```
p {font-family: sans-serif;}  /* в абзацах по умолчанию применяется
                                 рубленый шрифт */

<!-- следующий пример правильный (одиночные кавычки) -->
<p style="font-family: 'New Century Schoolbook', Times, serif;">...</p>

<!-- следующий пример НЕПРАВИЛЬНЫЙ (двойные кавычки) -->
<p style="font-family: "New Century Schoolbook", Times, serif;">...</p>
```

Двойные кавычки здесь конфликтуют с синтаксисом атрибута, как видно на рис. 5.3.

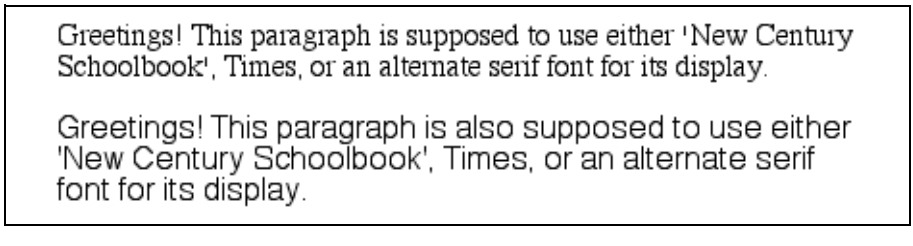


***Рис. 5.3.** К чему приводит применение не тех кавычек*

Насыщенность шрифта

Возможно, вы уже хорошо знакомы с насыщенностью шрифта (хотя и не осознаете этого); полужирный шрифт – самый обычный пример повышенной насыщенности. CSS посредством свойства `font-weight` предоставляет, по крайней мере теоретически, значительный контроль за насыщенностью.

font-weight	
Значения:	normal | bold | bolder | lighter | 100 | 200 | 300 | 400 | 500 | 600 | 700 | 800 | 900 | inherit
Начальное значение:	normal
Область применения:	все элементы
Наследование:	да
Вычисляемое значение:	одно из числовых значений (100 и пр.) или одно из числовых значений плюс одно из относительных значений (bolder или lighter)

Вообще говоря, чем выше насыщенность шрифта, тем он темнее и выглядит «более жирным». Существует огромное количество возможностей обозначения насыщенной гарнитуры шрифта. Например, семейство шрифтов Zurich имеет массу вариантов, таких как Zurich Bold, Zurich Black, Zurich UltraBlack, Zurich Light и Zurich Regular. Каждый из этих шрифтов основан на одном и том же базовым шрифте, а различаются они насыщенностью начертания.

Итак, скажем, вы желаете применить в документе шрифт Zurich, но хотели бы задействовать все возможные уровни насыщенности. К ним можно обратиться непосредственно через свойство font-family, но на самом деле совсем не обязательно делать это именно так. Кроме того, нет особого удовольствия писать такую таблицу стилей:

```
h1 {font-family: 'Zurich UltraBlack', sans-serif;}
h2 {font-family: 'Zurich Black', sans-serif;}
h3 {font-family: 'Zurich Bold', sans-serif;}
h4, p {font-family: Zurich, sans-serif;}
small {font-family: 'Zurich Light', sans-serif;}
```

Мало того, что очевидна утомительность создания подобной таблицы стилей, вдобавок таблица действует, только если данные шрифты установлены у всех, а это маловероятно. Логичнее было бы определить одно семейство шрифтов для всего документа, а затем назначить различную насыщенность разных элементов. Согласно теории это можно сделать, задавая различные значения свойства font-weight. Вот самое обычное объявление font-weight:

```
b {font-weight: bold;}
```

Здесь лишь сказано, что элемент b должен быть представлен полужирным шрифтом; или, иначе говоря, шрифтом, более насыщенным, чем обычный шрифт документа. Все выглядит привычно, т. к. тег b выделяет текст полужирным шрифтом.

Однако в реальности при отображении элемента b применяется другой, более насыщенный, вариант шрифта. Так, если абзац представлен

шрифтом Times, и часть его выделена полужирным шрифтом, то на самом деле используются два варианта одного и того же шрифта – Times и TimesBold. Обычный текст отображается с помощью шрифта Times, а полужирный – с помощью TimesBold.

Как работает свойство насыщенности

Чтобы понять, как агент пользователя устанавливает толщину, или насыщенность, данного варианта шрифта, не упоминая о том, как происходит ее наследование, проще всего начать с рассказа о ключевых словах от 100 до 900. Эти ключевые слова-числа были определены для обеспечения соответствия общепринятой практике разработки шрифтов, когда шрифту присваивается девять уровней насыщенности. Например, OpenType основывается на числовой шкале, состоящей из девяти значений. Если в шрифт встроены эти уровни насыщенности, числа непосредственно сопоставляются с предопределенными уровнями, где 100 – самый легкий вариант начертания шрифта, а 900 – самый плотный.

Кстати, эти числа не определяют конкретной плотности. Спецификация CSS говорит только о том, что каждое число соответствует плотности, по крайней мере не меньшей, чем плотность предшествующего ему числа. Таким образом, 100, 200, 300 и 400 могут соответствовать одному и тому же слабо насыщенному варианту начертания шрифта; 500 и 600 – одному и тому же варианту средней насыщенности; а 700, 800 и 900 могут формировать одинаковое очень насыщенное начертание. Поскольку ключевого слова, соответствующего варианту менее плотному, чем обозначенный предыдущим ключевым словом, нет, все будет в порядке.

Кстати, числа не задают собственных значений свойства `font-weight`, а соответствуют определенным общепринятым вариантам шрифтов. Значение 400 определено эквивалентом `normal`, а 700 соответствует `bold`. Остальные числа не сопоставляются с другими значениями `font-weight`, но они могут соответствовать общепринятым именам вариантов шрифтов. Если существует вариант, обозначенный, например, «Normal», «Regular», «Roman» или «Book», то ему присваивается значение 400, а любому варианту с меткой «Medium» ставится в соответствие 500. Однако если «Medium» – единственный доступный вариант шрифта, он сопоставляется с числом 400, а *не* с числом 500.

Еще больше придется потрудиться агенту пользователя, если в данном семействе шрифтов определено меньше девяти уровней насыщенности. В этом случае он должен заполнять пробелы по следующей заранее определенной схеме:

- Если значение 500 не поставлено в соответствие, ему назначается такая же насыщенность, как и для 400.
- Если значение 300 не поставлено в соответствие, ему присваивается вариант, менее насыщенный, чем 400. Если нет доступных менее

насыщенных вариантов, 300 назначается тот же вариант, что и для 400. В этом случае обычно он будет «Normal» или «Medium». Этот метод также применяется к 200 и 100.

- Если значение 600 не поставлено в соответствие, ему присваивается вариант, более насыщенный, чем 400. Если нет доступных более насыщенных вариантов, 600 назначается тот же вариант, что и для 500. Этот метод также применяется к 700, 800 и 900.

Чтобы проиллюстрировать эту схему распределения плотностей, рассмотрим три примера назначения плотности шрифта. В первом примере допустим, что семейство шрифтов Karrank% объединяет шрифты OpenType и поэтому имеет девять уровней насыщенности. В таком случае каждому уровню назначается число и ключевым словам normal и bold сопоставляются числа 400 и 700 соответственно.

Во втором примере рассмотрим семейство шрифтов Zurich, о котором мы говорили в начале этого раздела. Гипотетически его варианты могут быть сопоставлены с числовыми значениями font-weight, как показано в табл. 5.1.

Таблица 5.1. Гипотетическое распределение насыщенностей конкретного семейства шрифтов

Гарнитура шрифта	Назначенное ключевое слово	Назначенное число
Zurich Light		100, 200, 300
Zurich Regular	normal	400
Zurich Medium		500
Zurich Bold	bold	600, 700
Zurich Black		800
Zurich UltraBlack		900

Первые три числовых значения соответствуют наименее насыщенному начертанию. Гарнитура «Regular», как и ожидалось, получает ключевое слово normal и числовое значение насыщенности 400. Поскольку имеется шрифт «Medium», ему присваивается число 500. Здесь нет варианта, которому можно было бы присвоить 600, поэтому данное число сопоставляется с гарнитурой «Bold», которой также соответствуют значения 700 и bold. И наконец, числа 800 и 900 присваиваются вариантам «Black» и «UltraBlack» соответственно. Обратите внимание, что последнее назначение имеет место, только если для гарнитур в шрифте явно заданы два последних уровня насыщенности. В противном случае агент пользователя может проигнорировать их и назначить 800 и 900 гарнитуре «Bold» или присвоить оба этих значения тому или иному варианту «Black».

Наконец, рассмотрим сокращенную версию шрифта Times. В табл. 5.2 представлено только два варианта плотности: «TimesRegular» и «TimesBold».

Таблица 5.2. Гипотетическое распределение насыщенностей шрифта «Times»

Гарнитура шрифта	Назначенное ключевое слово	Назначенное число
TimesRegular	normal	100, 200, 300, 400, 500
TimesBold	bold	600, 700, 800, 900

Конечно, назначение ключевых слов normal и bold вполне понятно. А вот числа от 100 до 300 сопоставлены гарнитуре «Regular», потому что нет доступных гарнитур с менее насыщенным начертанием. Как и следовало ожидать, 400 отходит к «Regular», а как насчет 500? Это значение присваивается гарнитуре «Regular» (или normal), потому что нет доступных гарнитур «Medium»; таким образом, оно присваивается той же гарнитуре, что и значение 400. Что касается всех остальных числовых значений, 700, как обычно, отходит к bold, тогда как 800 и 900 по причине отсутствия более насыщенной гарнитуры присваиваются ближайшей менее насыщенной гарнитуре, каковой является «Bold». И наконец, 600 назначается следующей более насыщенной гарнитуре, которой, конечно же, является «Bold».

Свойство font-weight – наследуемое, поэтому если для абзаца должно быть задано значение bold:

```
p.one {font-weight: bold;}
```

то все его дочерние элементы унаследуют это начертание, как показано на рис. 5.4.

Здесь нет ничего необычного, но ситуация станет более интересной, если задать два еще не обсуждаемых нами последних значения: bolder и lighter. В общих чертах эти ключевые слова оказывают вполне ожидаемый эффект: они делают текст более или менее жирным по сравнению с насыщенностью шрифта его родителя. Рассмотрим сначала значение bolder.

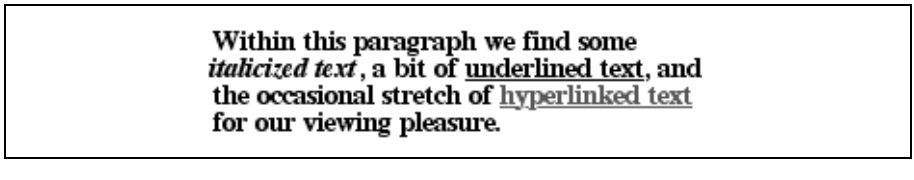


***Рис. 5.4.** Унаследованное свойство font-weight*

Увеличение насыщенности

Если для элемента задана насыщенность bolder, то агент пользователя прежде всего должен определить, какое значение свойства font-weight было унаследовано от родительского элемента. Затем он выбирает наименьшее числовое значение, соответствующее насыщенности шрифта,

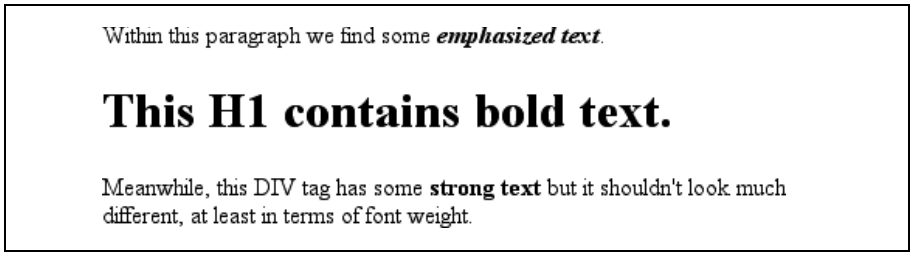


Рис. 5.5. *Попытки сделать текст более плотным*

превышающей унаследованную. Если ни одно из этих числовых значений не доступно, то агент пользователя присваивает плотности шрифта следующее числовое значение, кроме случая, когда значение уже равно `900`. Тогда значение насыщенности остается равным `900`. Таким образом, можно столкнуться со следующими ситуациями (рис. 5.5):

```
p {font-weight: normal;}
p em {font-weight: bolder;}  /* в результате получаем полужирный текст,
                                вычисленное значение - '700' */

h1 {font-weight: bold;}
h1 b {font-weight: bolder;}  /* если не существует более насыщенной
                                гарнитуры, вычисленное значение - '800' */

div {font-weight: 100;}      /* принимаем, что гарнитура 'Light'
                                существует; см. пояснение */
div strong {font-weight: bolder;}   /* в результате получаем текст
                                       стандартной насыщенности, '400' */
```

В первом примере агент пользователя повышает насыщенность с `normal` до `bold`; в числовом представлении это переход от `400` к `700`. Во втором примере для текст `h1` уже задано значение `bold`. Если нет доступной более насыщенной гарнитуры, агент пользователя присваивает насыщенности текста `b` элемента `h1` значение `800`, т. к. это следующий после `700` (числовой эквивалент `bold`) шаг. Поскольку значение `800` присвоено той же гарнитуре шрифта, что и `700`, между обычным и полужирным текстом `h1` нет видимой разницы, но тем не менее их насыщенность разная.

В последнем примере текст абзацев должен иметь наименьшую насыщенность шрифта, которую мы приняли как вариант Light. Кроме того, должны существовать другие гарнитуры этого семейства шрифтов – Regular и Bold. Насыщенность любого текста элемента `em`, расположенного внутри абзаца, будет приведена к значению `normal`, поскольку это следующая в порядке повышения насыщенности гарнитура семейства шрифтов. Но что если единственными гарнитурами шрифта являются Regular и Bold? В этом случае объявления преобразовывались бы так:

```
/* предполагаем для данного примера только две гарнитуры: 'Regular' и 'Bold' */
p {font-weight: 100;}            /* выглядит аналогично тексту 'normal' */
p span {font-weight: bolder;}    /* преобразуется в '700' */
```

Как видите, значение плотности 100 присваивается гарнитуре шрифта normal, но значение свойства font-weight все еще остается равным 100. Таким образом, любой текст элемента span, происходящего от элемента p, унаследует значение 100, а затем перейдет к следующей по насыщенности гарнитуре, каковой является Bold, а в числовом выражении – 700.

Сделаем еще шаг вперед и добавим два правила плюс некоторую разметку, чтобы показать, как все это работает (результат представлен на рис. 5.6):

```
/*  предполагаем для этого примера только две гарнитуры: 'Regular' и 'Bold'  */
p {font-weight: 100;}             /* выглядит аналогично тексту 'normal' */
p span {font-weight: 400;}        /* аналогично предыдущему */
strong {font-weight: bolder;}     /* более насыщенный, чем его родитель */
strong b {font-weight: bolder;}   /* по-прежнему более насыщенный */
```

```
<p>
This paragraph contains elements of increasing weight: there is a
<span>span element that contains a <strong>strongly emphasized
element and a <b>boldface element</b></strong></span>.
</p>
```

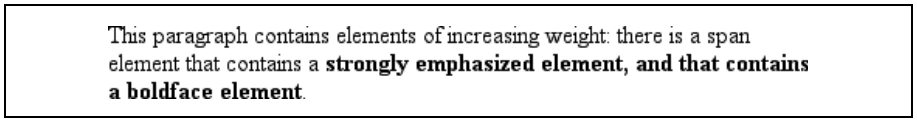


***Рис. 5.6.** Увеличение насыщенности*

В двух последних вложенных элементах вычисляемое значение font-weight увеличивается по причине нестрогого применения ключевого слова bolder. Если текст параграфа был заменен числами, представляющими значения font-weight каждого элемента, то получилось бы следующее:

```
<p>
100 <span> 400 <strong> 700 <b> 800 </b></strong></span>.
</p>
```

В первых двух случаях видно увеличение насыщенности шрифта, поскольку происходит переход от 100 к 400 и от 400 к bold (700). После 700 более насыщенную гарнитуру сопоставить не удается, поэтому агент пользователя просто переходит к значению font-weight, следующему по числовой шкале (800). Более того, если в элемент b вставить элемент strong, то получится следующее:

```
<p>
100 <span> 400 <strong> 700 <b> 800 <strong> 900
```

```
</strong></b></strong></span>.
</p>
```

Если вставить еще один элемент `b` во внутренний элемент `strong`, его насыщенность также была бы равна `900`, поскольку значение `font-weight` не может превышать `900`. Если предполагается, что доступны только две гарнитуры шрифта, то для текста выбирается или Regular, или Bold, как видно на рис. 5.7:

```
<p>
regular <span> regular <strong> bold <b> bold
<strong> bold </strong></b></strong></span>.
</p>
```

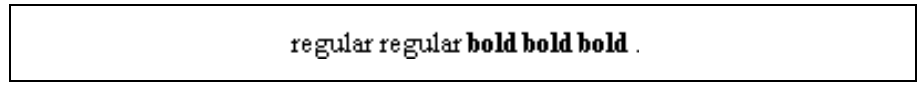


***Рис. 5.7.** Визуальное представление насыщенности с указанием дескрипторов*

Уменьшение насыщенности

Принцип действия `lighter` совершенно аналогичен за исключением того, что оно заставляет агента пользователя опускаться вниз по шкале плотностей, а не подниматься. Изменим предыдущий пример:

```
/*  предполагаем для этого примера только две гарнитуры: 'Regular' и 'Bold'  */
p {font-weight: 900;}            /* максимально возможное жирное начертание,
                                    которое будет выглядеть 'bold' */
p span {font-weight: 700;}       /* это также будет жирным */
strong {font-weight: lighter;}   /* менее насыщенный, чем его родитель */
b {font-weight: lighter;}        /* по-прежнему менее насыщенный */
```

```
<p>
900 <span> 700 <strong> 400 <b> 300 <strong> 200
</strong></b></strong></span>.
</p>
<!-- ...или иначе... -->
<p>
bold <span> bold <strong> regular <b> regular
<strong> regular </strong></b></strong></span>.
</p>
```

Абстрагируемся от того, что этот пример совершенно нелогичен, и посмотрим на рис. 5.8: насыщенность основного текста абзаца равна `900`. Если указывается, что для текста `strong` должно быть задано значение `lighter`, то вычисляется следующая по порядку менее насыщенная гарнитура, которой является Regular или `400` (аналогично `normal`) на цифровой шкале. Следующий шаг вниз – `300`, что аналогично `normal`, поскольку еще менее насыщенных гарнитур не существует. С этого момента агент пользователя будет уменьшать насыщенность по одно-

му числовому значению за раз до тех пор, пока не достигнет значения 100 (этого нет в примере). Второй абзац показывает, какой текст будет жирным, а какой обычным.

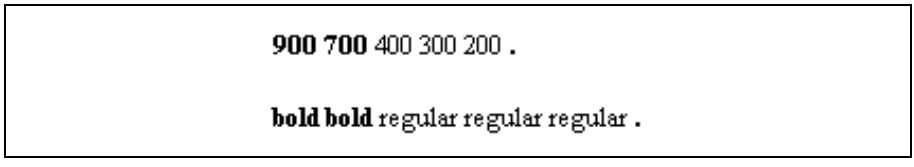


***Рис. 5.8.** Уменьшение насыщенности*

Размер шрифта

Способ задания размера шрифта выглядит очень похоже и в то же время сильно отличается от задания насыщенности.

font-size

Значения:	xx-small | x-small | small | medium | large | x-large | xx-large | smaller | larger | <размер> | <процент> | inherit
Начальное значение:	medium
Область применения:	все элементы
Наследование:	да
Процентное задание:	вычисляется относительно размера шрифта родительского элемента
Вычисляемое значение:	абсолютный размер

Для свойства font-size определены ключевые слова larger и smaller, аналогичные ключевым словам bolder и lighter свойства font-weight. Во многом так же, как это было и с относительными насыщенностями шрифтов, согласно этим ключевым словам вычисляемое значение font-size перемещается вверх и вниз по шкале значений размеров, с которой надо познакомиться до того, как приступать к изучению ключевых слов larger и smaller. Сначала поговорим о том, как измеряются размеры шрифтов.

Следует учитывать, что связь свойства font-size с тем, что отображается на экране, в действительности определяется разработчиком шрифта. Размер шрифта характеризуется кегельной *площадкой* (*em square*) (некоторые называют ее кегельным квадратом) шрифта. Кегельная площадка (и соответственно размер шрифта) не задается границами каких-либо символов шрифта. Эта величина определяется расстоянием между базовыми линиями, если шрифт задан без дополнительных межстрочных интервалов (line-height (высота строки) в CSS). Шрифты

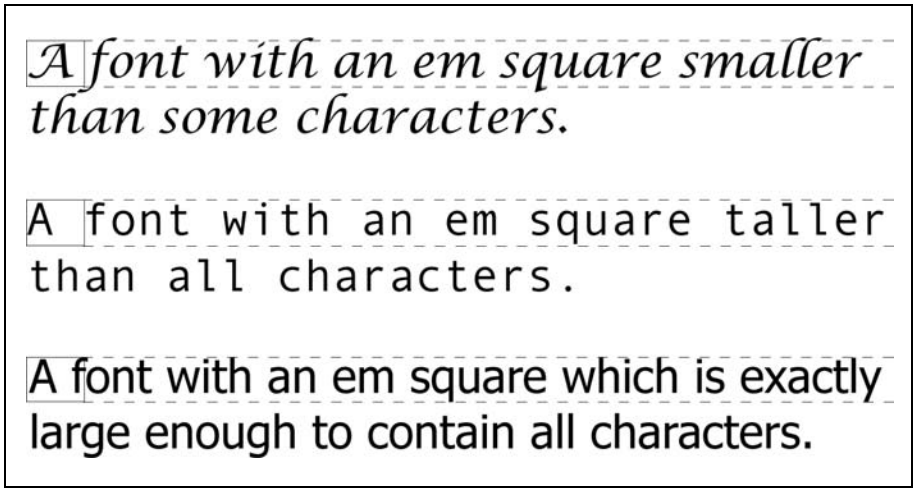


Рис. 5.9. *Символы шрифта и кегельные площадки*

могут иметь символы, размер которых превышает стандартное расстояние между базовыми линиями. Впрочем, шрифт может быть определен так, что все его символы будут меньше, чем его кегельная площадка, что и наблюдается для многих шрифтов. Некоторые гипотетические примеры приведены на рис. 5.9.

Таким образом, действие `font-size` состоит в задании размера кегельной площадки используемого шрифта. Это не гарантирует, что любой из фактически отображаемых символов будет иметь такой размер.

Абсолютные размеры

Разобравшись с этим, обратимся к ключевым словам для задания абсолютных размеров. Существует семь значений абсолютного размера для свойства `font-size`: `xx-small`, `x-small`, `small`, `medium`, `large`, `x-large` и `xx-large`. Они определены не строго, а относительно друг друга, как показано на рис. 5.10:

```
p.one {font-size: xx-small;}
p.two {font-size: x-small;}
p.three {font-size: small;}
p.four {font-size: medium;}
p.five {font-size: large;}
p.six {font-size: x-large;}
p.seven {font-size: xx-large;}
```

Согласно спецификации CSS1 разница (или коэффициент *масштабирования* (*scaling factor*)) между двумя соседними абсолютными размерами должна составлять примерно 1,5 при переходе от меньшего к большему и 0,66 при переходе от большего к меньшему. Таким образом, если размер `medium` соответствует `10px`, то `large` должен быть `15px`.

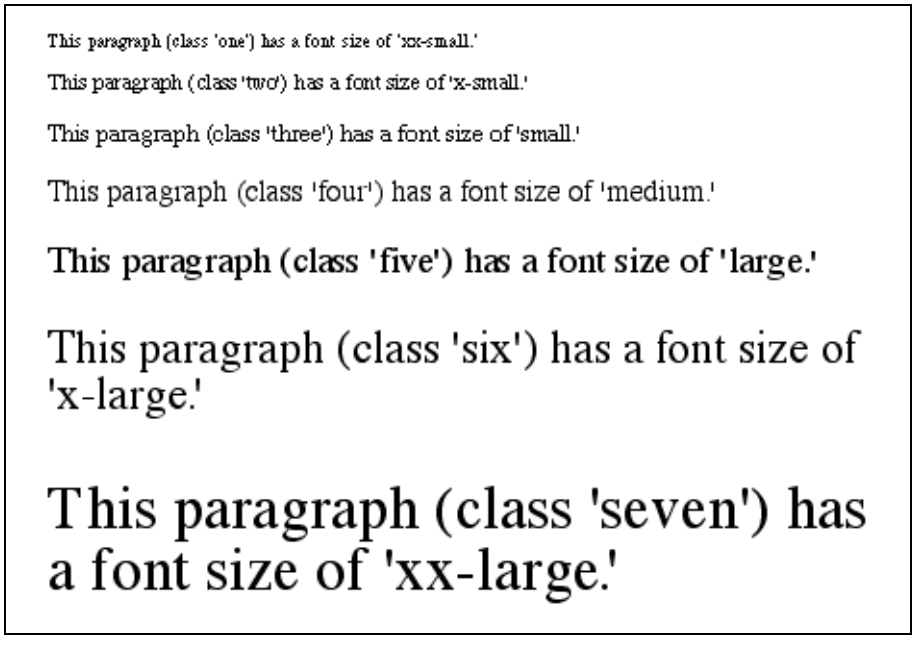


Рис. 5.10. *Абсолютные размеры шрифтов*

С другой стороны, коэффициент масштабирования не обязательно должен быть равен 1,5; он не только может быть различным в разных агентах пользователя, но в CSS2 его значение было изменено и теперь находится где-то между 1,0 и 1,2.

Исходя из предположения, что размер `medium` равен `16px`, для разных коэффициентов масштабирования мы получаем абсолютные размеры, приведенные в табл. 5.3. (Указанные размеры, конечно, являются приближенными.)

Таблица 5.3. Преобразование коэффициентов масштабирования в пикселы

Ключевое слово	Масштабирование: 1.5	Масштабирование: 1.2
`xx-small`	5px	9px
`x-small`	7px	11px
`small`	11px	13px
`medium`	16px	16px
`large`	24px	19px
`x-large`	36px	23px
`xx-large`	54px	28px

Еще более усложняет ситуацию тот факт, что в различных агентах пользователя разные ключевые слова, задающие абсолютные размеры, принимаются за «стандартный» размер шрифта. В качестве примера возьмем версии 4 броузеров: Navigator 4 размером `medium` считает размер текста, к которому не применены никакие стили, тогда как Internet Explorer 4 принимает, что размеру нестилизованного текста соответствует `small`. Несмотря на тот факт, что применяемым по умолчанию значением свойства `font-style` является `medium`, поведение IE4 может показаться нелогичным, но оно не настолько ошибочно, как это может показаться на первый взгляд.[1] К счастью, в IE6 эта проблема решена, по крайней мере в стандартном режиме, и в качестве стандартного размера принимается `medium`.

Относительные размеры

По сравнению со всем вышесказанным, с ключевыми словами `larger` и `smaller` все проще: они обусловливают смещение размера элемента вверх или вниз по шкале абсолютных размеров – относительно размера их родительского элемента – с применением того же коэффициента масштабирования, который фигурировал в вычислениях абсолютных размеров. Иначе говоря, если броузер, вычисляя абсолютные размеры, принимал коэффициент масштабирования равным 1,2, то и в случае ключевых слов задания относительных размеров он должен брать такой же коэффициент.

```
p {font-size: medium;}
strong, em {font-size: larger;}
```

```
<p>This paragraph element contains <strong>a strong-emphasis element which
itself contains <em>an emphasis element that also contains
<strong>a strong element.</strong></em></strong></p>
```

```
<p> medium <strong>large <em> x-large
<strong>xx-large</strong></em></strong></p>
```

В отличие от относительных значений насыщенности, для относительных значений размера не должно выполняться ограничение диапазоном абсолютных размеров. Таким образом, размер шрифта может выйти за рамки размеров, предполагаемых для `xx-small` и `xx-large`. Например:

```
h1 {font-size: xx-large;}
em {font-size: larger;}
```

[1] Примите во внимание, что существует семь ключевых слов задания абсолютных размеров, так же как существует семь размеров `font` (например, `<font size="5">`). Поскольку исторически сложилось, что применяемым по умолчанию размером шрифта был `3`, имело бы смысл применять для обозначения стандартного размера шрифта третье значение из списка ключевых слов абсолютных значений CSS. Поскольку третьим значением оказалось `small`, становится понятным решение Explorer.

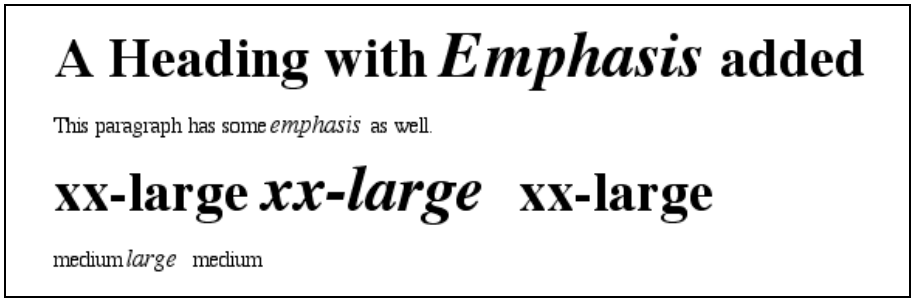


Рис. 5.11. Установка относительных размеров шрифта, выходящих за границы абсолютных размеров

```
<h1>A Heading with <em>Emphasis</em> added</h1>
<p>This paragraph has some <em>emphasis</em> as well.</p>
```

Как видно из рис. 5.11, выделенный текст элемента `h1` немного больше, чем `xx-large`. Выбор масштаба остается за агентом пользователя, хотя предпочтительным является коэффициент масштабирования 1,2. Размер текста элемента `em` в абзаце, очевидно, на одну позицию превышает максимальный размер шкалы абсолютных размеров (`large`).

От агентов пользователя не требуется увеличивать или уменьшать шрифт так, чтобы его размер выходил за границы, определяемые ключевыми словами задания абсолютных размеров.

Процентные значения и размеры

В некотором роде процентные значения подобны ключевым словам, задающим относительные размеры. Процентное значение всегда вычисляется на основании любого размера, унаследованного от родительского элемента. В отличие от задающих относительные размеры ключевых слов, процентные значения обеспечивают более точную настройку вычисляемого размера шрифта. Рассмотрим следующий пример, проиллюстрированный на рис. 5.12:

```
body {font-size: 15px;}
p {font-size: 12px;}
em {font-size: 120%;}
strong {font-size: 135%;}
small, .fnote {font-size: 75%;}
```

```
<body>
<p>This paragraph contains both <em>emphasis</em> and <strong>strong
emphasis</strong>, both of which are larger than their parent element.
The <small>small text</small>, on the other hand, is smaller by a quarter.</p>
<p class="fnote">This is a 'footnote' and is smaller than regular text.</p>
```

```
<p> 12px <em> 14.4px </em> 12px <strong> 16.2px </strong> 12px
```

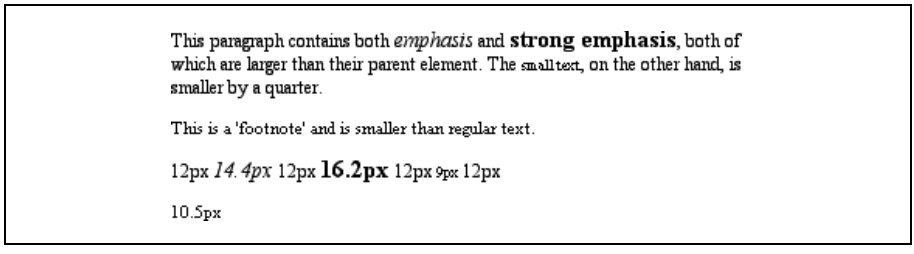


Рис. 5.12. *Результат задания размеров в процентах*

```
<small> 9px </small> 12px </p>
<p class="fnote"> 10.5px </p>
</body>
```

В этом примере показаны точно заданные в пикселах размеры. На практике веб-броузер, скорее всего, округлит значения до ближайшего целого, например до `14px`, хотя улучшенные агенты пользователя могут аппроксимировать дробные значения с помощью механизма сглаживания или при распечатке документа. Для других значений свойства `font-size` веб-броузер, возможно (а возможно, и нет), сохранит дробные части.

Кстати, CSS определяет значение длины `em` как процентное в том смысле, что при задании размеров шрифтов значение `1em` эквивалентно `100%`. Таким образом, следующие правила формируют идентичные результаты (при условии, что родительский элемент обоих абзацев один и тот же):

```
p.one {font-size: 166%;}
p.two {font-size: 1.6em;}
```

Для единиц измерения длины `em` действуют те же принципы, что и при работе с процентными значениями, такие как наследование вычисленных размеров и т. д.

Размер шрифта и наследование

На рис. 5.12 также показано, что хотя свойство `font-size` является в CSS наследуемым, наследуются вычисленные значения, а не процентные. Таким образом, элемент `strong` наследует значение `font-size`, равное `12px`, и в соответствии с объявленным значением `135%` размер шрифта составляет `16.2px` (что, вероятно, будет округлено до `16px`). Для абзаца сноски процентное значение вычисляется относительно значения `font-size`, унаследованного от элемента `body` и равного `15px`. Вычислив `75%` от этого значения, получаем `11.25px`.

Как и ключевые слова, задающие относительные размеры, процентные значения относятся к накопительным. Таким образом, следующая разметка отображается, как показано на рис. 5.13:

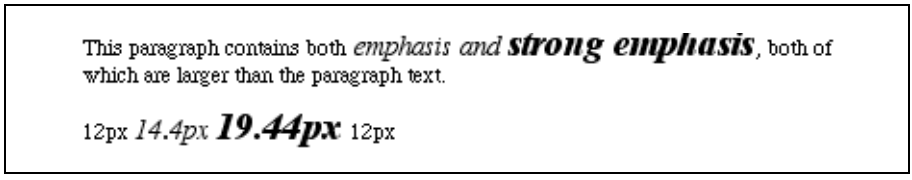


Рис. 5.13. *Вопросы наследования*

```
p {font-size: 12px;}
em {font-size: 120%;}
strong {font-size: 135%;}

<p>This paragraph contains both<em>emphasis and <strong>strong
emphasis</strong></em>, both of which are larger than the paragraph text. </p>

<p> 12px <em>14.4px <strong> 19.44px </strong></em> 12px  </p>
```

Значение размера элемента `strong`, показанного на рис. 5.13, вычисляется следующим образом:

```
12 px x 120% = 14.4px
14.4px x 135% = 19.44px (возможно округление до 19px)
```

Однако существует альтернативный сценарий, в котором окончательное значение несколько иное. В этом сценарии агент пользователя округляет размер, выраженный в пикселах, и затем дочерние элементы наследуют эти округленные значения. Хотя это поведение с точки зрения спецификации было бы некорректным, давайте предположим, что рабочий агент это делает. Итак:

```
12px × 120% = 14.4px [14.4px ≈ 14px]
14px × 135% = 18.9px [18.9px ≈ 19px]
```

Даже если предположить, что агент пользователя выполняет округление на каждом этапе, конечный результат этих двух вычислений и предыдущих будет одинаковым: 19 пикселов. Однако при вычислении процентных значений погрешность округления накапливается и по мере увеличения количества этих операций растет.

Проблема неконтролируемого масштабирования может также иметь и другие последствия. Представим на мгновение документ, состоящий из ненумерованных списков, многие из которых вложены друг в друга. Некоторые из этих списков имеют четыре уровня вложенности. Представьте результат применения к этому документу следующего правила:

```
ul {font-size: 80%;}
```

На четвертом уровне вложенности вычисленное значение `font-size` ненумерованного списка составляло бы 40,96% размера шрифта предка самого верхнего уровня. Размер шрифта каждого вложенного списка составлял бы 80% размера шрифта его родительского списка, что все

более и более затрудняло бы чтение каждого последующего уровня. Аналогичная проблема может возникнуть, если для верстки документа применяются вложенные таблицы. Тогда можно было бы написать такое правило:

```
td {font-size: 0.8em;}
```

В любом случае, скорее всего, получится страница, практически непригодная для чтения.

Использование единиц измерения длины

Свойство `font-size` может быть задано в любых единицах измерения длины, которые обсуждались в главе 4. Все приведенные ниже объявления `font-size` эквивалентны:

```
p.one {font-size: 36pt;}
p.two {font-size: 3pc;}
p.three {font-size: 0.5in;}
p.four {font-size: 1.27cm;}
p.five {font-size: 12.7mm;}
```

Представление на рис. 5.14 предполагает, что агент пользователя знает разрешение, установленное для устройства отображения. Разные агенты пользователя делают разные предположения: некоторые на основании операционной системы, некоторые на основании предпочтительных настроек, а некоторые на основании предположений программиста, написавшего агент пользователя. Однако размер этих пяти строк должен быть всегда одинаковым. Итак, хотя результат может не

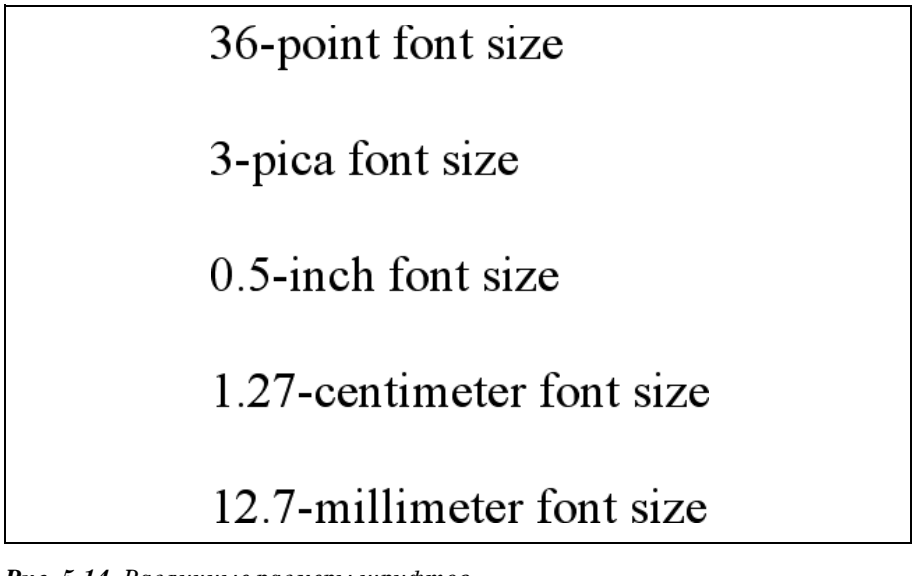


Рис. 5.14. *Различные размеры шрифтов*

соответствовать реальности (например, реальный размер `p.three`, возможно, не будет равен половине дюйма), все размеры должны быть согласованными.

Существует еще одно значение размера, потенциально такое же, как показанные на рис. 5.14, – `36px`, что физически составляло бы такое же расстояние, если бы разрешение устройства отображения было 72 пиксела на дюйм (ppi). Однако сейчас таких мониторов очень мало. Разрешение большинства намного выше и находится в диапазоне от 96ppi до 120ppi. Многие старые веб-броузеры, работающие на Макинтошах, обрабатывают пункты и дюймы так, как будто они эквивалентны, поэтому значения `14pt` и `14px` в их «исполнении» могут выглядеть одинаково. Это, однако, не касается Windows и других платформ, включая MacOS X, и это одна из основных причин, по которым пункты могут быть очень неудобной единицей измерения при разработке документов.

Из-за этих отличий операционных систем многие авторы для измерения размеров шрифтов прибегают к пикселам. Этот подход особенно привлекателен, когда на веб-странице есть и текст, и изображения, поскольку для текста можно (теоретически) задать высоту – такую же, как и высота графических элементов на странице, объявляя `font-size: 11px;` или что-то подобное, как показано на рис. 5.15.

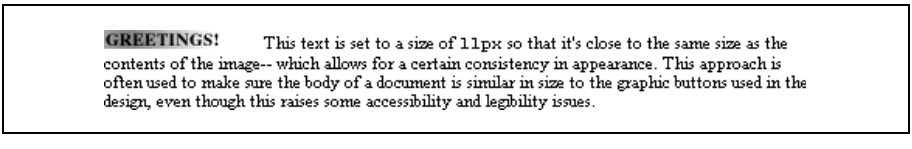


***Рис. 5.15.** Сохранение масштабов текста и графических элементов за счет определения размеров в пикселах*

Конечно, обращение к пикселам для определения `font-size` – один из способов получения «согласованных» результатов в случае применения `font-size` (и вообще для любой длины), но здесь есть существенный недостаток. Internet Explorer для Windows вплоть до версии 6.0 не дает пользователям возможности изменять размер текста, заданый в пикселах. Другие броузеры, включая Mozilla, Netscape 6+, IE5+ для Mac OS, Opera и даже IE7, разрешают пользователю менять размер текста независимо от того, как он был задан. Таким образом, применение пикселов для задания размера текста (как и применение в других случаях) не гарантирует, что результат везде будет представлен одинаково. Другие подходы, рассмотренные в этой главе, такие как ключевые слова и процентные соотношения, представляют собой намного более надежный (и удобный для пользователя) способ, поскольку могут применяться для масштабирования текста на основании стандартного размера шрифта пользователя.

Стили и варианты

По сравнению со всем, о чем шла речь ранее, понимание данного раздела практически не требует умственного напряжения. Обсуждаемые здесь свойства настолько просты, а сложности так мизерны, что все это похоже на сказку. Сначала поговорим о свойстве `font-style`, а затем, прежде чем завершить обсуждение свойств шрифтов, перейдем к `font-variant`.

Стиль шрифта

Свойство `font-style` очень простое: оно применяется для выбора обычного текста (`normal`), курсива (`italic`) и наклонного текста (`oblique`). Вот и все! Единственная сложность состоит в том, чтобы понять разницу между курсивом и наклонным текстом и осознать, почему броузеры не всегда предоставляют выбор.

font-style

Значения:	`italic | oblique | normal | inherit`
Начальное значение:	`normal`
Область применения:	все элементы
Наследование:	да
Вычисляемое значение:	как задано

Как видите, по умолчанию для `font-style` устанавливается значение `normal`. Подразумевается «прямой» текст, который, вероятно, лучше всего описывается как «текст, не являющийся курсивным или как-либо наклоненным». Например, текст этой книги, за редкими исключениями, набран прямым шрифтом. Поэтому остается только объяснить разницу между начертаниями `italic` и `oblique`. Для этого проще всего обратиться к рис. 5.16, который это иллюстрирует.

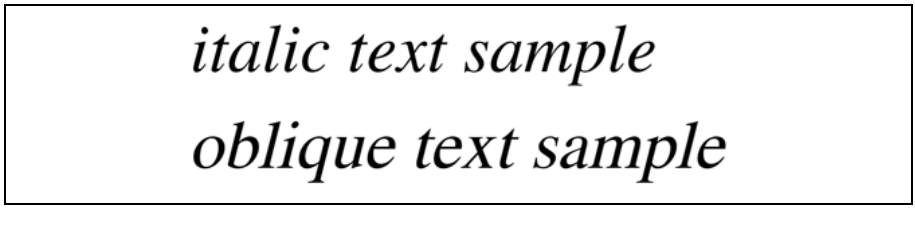


Рис. 5.16. *Курсивное и наклонное начертания*

По существу, курсив – это отдельная гарнитура шрифта с небольшими изменениями, внесенными в структуру каждой буквы для варьирования внешнего вида. Это особенно касается шрифтов антиква, в которых, кроме того, что символы этого текста «наклоняются», гарнитура антик-

ва может быть превращена в гарнитуру «курсив». Наклонный текст, с другой стороны, представляет собой просто наклонную версию обычного прямого текста. Гарнитуры шрифтов, обозначенные как Italic, Cursive и Kursiv, обычно соответствуют ключевому слову `italic`, тогда как `oblique` часто присваивается гарнитурам Oblique, Slanted и Incline.

Для того чтобы гарантировать единообразное представление курсива в документе, можно написать такую таблицу стилей:

```
p {font-style: normal;}
em, i {font-style: italic;}
```

Согласно этим стилям в абзацах будет использоваться прямой шрифт, а элементы `em` и `i` будут выделены курсивом. С другой стороны, вы можете принять решение о том, что элементы `em` и `i` должны немного отличаться:

```
p {font-style: normal;}
em {font-style: oblique;}
i {font-style: italic;}
```

Посмотрев внимательно на рис. 5.17, можно не заметить особой разницы между элементами `em` и `i`. На практике не всякий шрифт настолько сложен, чтобы иметь и курсивную, и наклонную гарнитуры, и еще меньшее количество веб-броузеров настолько совершенны, чтобы различать эти гарнитуры.

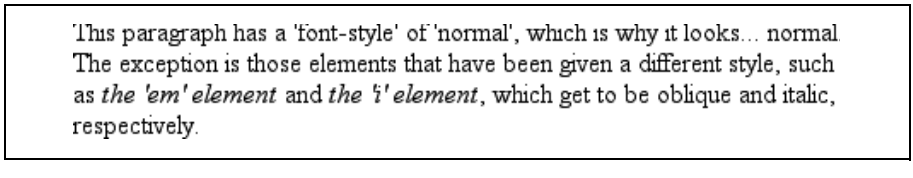


***Рис. 5.17.** Стили шрифтов*

В любом случае возможны несколько вариантов развития событий. Если нет гарнитуры Italic, но есть гарнитура Oblique, то вместо первой может использоваться вторая. Если ситуация обратная – существует гарнитура Italic, но нет определенной гарнитуры Oblique, – то агент пользователя, возможно, *не* заменит вторую гарнитуру первой согласно спецификации. И наконец, агент пользователя может просто генерировать наклонную гарнитуру, самостоятельно формируя наклонную версию прямого шрифта. Кстати, именно это чаще всего и происходит в компьютерах: ведь с помощью несложных вычислений очень просто «наклонить» шрифт.

Более того, в некоторых операционных системах шрифт, объявленный как `italic`, может превратиться из курсивного в наклонный в зависимости от его фактического размера. На рис. 5.18 приведено представление шрифта Times на Макинтоше, работающем под Classic OS (Mac OS 9), и вся разница – один пиксел в размере.

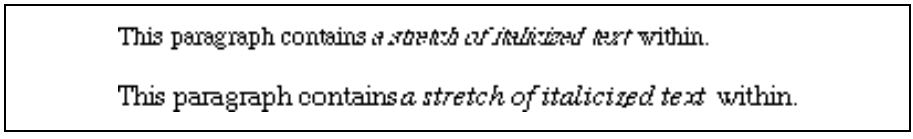


Рис. 5.18. *Один шрифт, один стиль, разные размеры*

К сожалению, с этим практически ничего нельзя сделать. Лучше оставьте обработку шрифтов операционным системам, как это делается в Mac OS X и Windows XP. Обычно курсивные и наклонные шрифты выглядят в веб-броузерах совершенно одинаковыми.

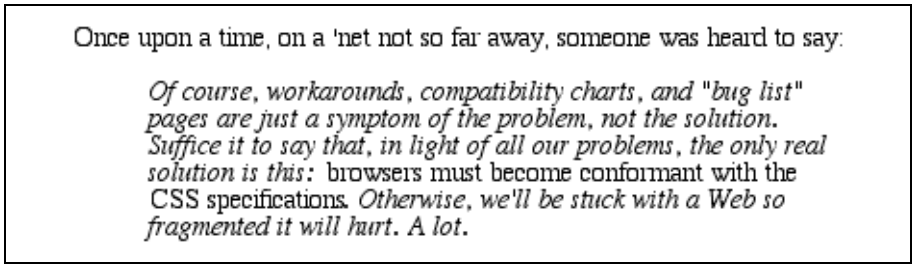


Рис. 5.19. *Общепринятые типографские соглашения в CSS*

Но несмотря на все это, свойство `font-style` может быть полезным. Например, по общепринятому типографскому соглашению цитаты должны набираться курсивом, но любой специально выделенный текст в них должен быть прямым. Чтобы реализовать этот эффект, продемонстрированный на рис. 5.19, можно применить такие стили:

```
blockquote {font-style: italic;}
blockquote em, blockquote i {font-style: normal;}
```

Варианты шрифтов

Кроме размеров и стилей, шрифты также могут иметь варианты. CSS предлагает способ реализации одного очень распространенного варианта.

font-variant

Значения:	`small-caps | normal | inherit`
Начальное значение:	`normal`
Область применения:	все элементы
Наследование:	да
Вычисляемое значение:	как задано

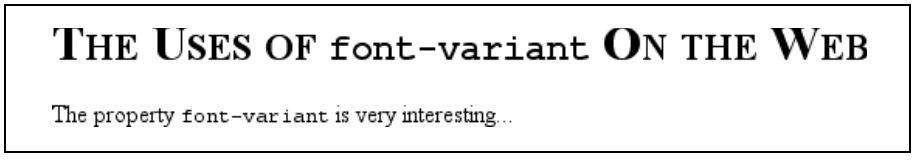


***Рис. 5.20.** Применение капители*

Что касается свойства `font-variant`, у него только два ненаследуемых значения: применяемое по умолчанию `normal`, которое описывает обычный текст, и `small-caps` (капители, или малые прописные). Для тех, кто незнаком с этим эффектом: Это Выглядит Примерно Так. Вместо прописных и строчных букв в капители прописные буквы имеют разный размер. Таким образом, можно увидеть примерно следующее (рис. 5.20):

```
h1 {font-variant: small-caps;}
h1 code, p {font-variant: normal;}

<h1>The Uses of <code>font-variant</code> On the Web</h1>
<p>
The property <code>font-variant</code> is very interesting...
</p>
```

Как можно было заметить, в представлении элемента `h1` место прописных букв исходного текста занимают большие прописные буквы, а строчные буквы исходного текста замещаются маленькими прописными буквами. Это очень похоже на `text-transform: uppercase`, единственное реальное отличие в том, что здесь прописные буквы имеют разные размеры. Однако `small-caps` объявляется с помощью свойства шрифта потому, что некоторые шрифты имеют особую гарнитуру капители, которая определяется свойством `font`.

Что происходит, если такой гарнитуры нет? В спецификации предлагается два варианта. Первый – агент пользователя создает гарнитуру капители путем масштабирования прописных букв. Второй – все буквы приводятся к верхнему регистру и одному размеру, точно как если бы использовалось объявление `text-transform: uppercase;`. Конечно, это не идеальное решение, но оно допускается.

В Internet Explorer для Windows до версии 6 применялся вариант перевода всех букв в верхний регистр. Остальные броузеры в большинстве своем отображают текст, набранный капителью.

Растяжение и корректировка шрифтов

Есть два свойства шрифтов, появившихся в CSS2, но отсутствующих в CSS2.1. В CSS2.1 они были опущены, потому что, несмотря на их многолетнее присутствие в спецификации, они не были реализованы

font-stretch	
Значения:	`normal | wider | narrower | ultra-condensed | extra-condensed | condensed | semi-condensed | semi-expanded | expanded | extra-expanded | ultra-expanded | inherit`
Начальное значение:	`normal`
Область применения:	все элементы
Наследование:	да

в броузерах. Первое обеспечивает растяжение шрифтов в горизонтальном направлении, а второе – интеллектуальное масштабирование подставляемых шрифтов, когда недоступна первая возможность. Начнем с растяжения.

Как можно ожидать из его значений, это свойство предназначено для того, чтобы делать символы шрифта шире или уже. Его поведение очень схоже с ключевыми словами задания абсолютных значений (например, `xx-large`) свойства `font-size`, оно имеет диапазон абсолютных значений и два значения, которые позволяют автору увеличивать или уменьшать разряжение шрифта. Например, автор решил воздействовать на текст элемента, выделяемого жирным шрифтом, растягивая его символы так, чтобы они были шире, чем символы шрифта родительского элемента, как показано на рис. 5.21:

```
strong {font-stretch: wider;}
```

If there's one thing I can't **stress enough**, it's the value of Photoshop in producing a book like this one.

Рис. 5.21. *Растяжение символов шрифта*

Изображение на рис. 5.21 было обработано в программе Photoshop, поскольку на момент написания данной книги веб-броузеры не поддерживают свойство `font-stretch`.

Точно так же нереализованный процесс корректировки размера шрифта является немного более сложным.

Цель этого свойства – сохранить удобочитаемость в том случае, когда применяется не тот шрифт, который изначально был выбран автором. Поскольку все шрифты выглядят по-разному, один шрифт, имея определенный размер, может читаться хорошо, тогда как другой шрифт при таком же размере – с трудом или вообще никак.

Факторы, оказывающие влияние на удобочитаемость шрифтов, – размер и x-высота (высота глифов строчных букв). Результат деления x-высоты на `font-size` называют *аспектом шрифта* (*aspect value*).

font-size-adjust	
Значения:	<число> | none | inherit
Начальное значение:	none
Область применения:	все элементы
Наследование:	да

Шрифты, имеющие большее значение аспекта, дольше сохраняют разборчивость по мере уменьшения размера, и соответственно шрифты с малым аспектом быстрее перестают читаться.

Сказанное хорошо иллюстрирует сравнение обычных шрифтов Verdana и Times. Рассмотрим рис. 5.22 и следующую разметку, где размер обоих шрифтов равен 10px:

```
p {font-size: 10px;}
p.cl1 {font-family: Verdana, sans-serif;}
p.cl2 {font-family: Times, serif; }
```

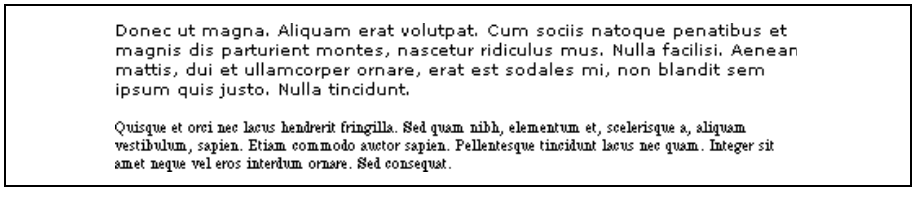


Рис. 5.22. *Сравнение шрифтов Verdana и Times*

Текст, написанный шрифтом Times, читать намного сложнее, чем текст Verdana. Частично дело в ограничениях растрового отображения, но еще и в том, что шрифт Times просто становится нечитаемым при малых размерах.

Как выясняется, отношение x-высоты к размеру символа в шрифте Verdana составляет 0,58, тогда как для Times эта величина равна 0,46. В этом случае можно объявить аспект Verdana, и агент пользователя скорректирует размер фактически используемого текста. Вот формула:

> Объявленный `font-size` × (значение `font-size-adjust` ÷ аспект доступного шрифта) = скорректированный `font-size`

Таким образом, если шрифт Times применяется вместо Verdana, корректировка такова:

```
10px × (0.58 ÷ 0.46) = 12.6px
```

Результат показан на рис. 5.23:

```
p {font: 10px Verdana, sans-serif; font-size-adjust: 0.58;}
p.cl1 {font-family: Times, serif; }
```

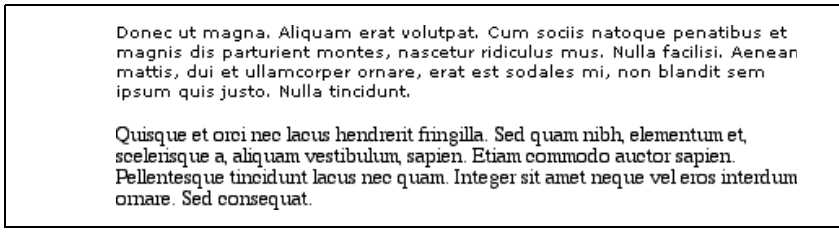


Рис. 5.23. *Корректировка шрифта Times*

Изображение на рис. 5.23 было обработано в Photoshop, поскольку на момент написания данной книги свойство `font-size-adjust` поддерживается очень небольшим количеством веб-браузеров.

Конечно, чтобы обеспечить агенту пользователя возможность выполнения разумной корректировки размеров, необходимо знать аспект первоначально выбранного шрифта. В CSS2 нельзя просто получить это значение из шрифта, а многие шрифты, возможно, вообще не предоставляют подобной информации.

Свойство font

Все эти свойства очень замысловаты, а применение их в совокупности может сделать объявления несколько громоздкими:

```
h1 {font-family: Verdana, Helvetica, Arial, sans-serif; font-size: 30px;
font-weight: 900; font-style: italic; font-variant: small-caps;}
h2 {font-family: Verdana, Helvetica, Arial, sans-serif; font-size: 24px;
font-weight: bold; font-style: italic; font-variant: normal;}
```

font

Значения:	[[<стиль-шрифта> || <вариант-шрифта> || <плотность-шрифта>]? <размер-шрифта> [/ <высота-строки>]? <семейство-шрифтов>] | caption | icon | menu | message-box |small-caption | status-bar | inherit
Начальное значение:	обратитесь к отдельным свойствам
Область применения:	все элементы
Наследование:	да
Процентные соотношения:	вычисляются относительно родительского элемента для <размер-шрифта> и относительно <размер-шрифта> элемента для <высота-строки>
Вычисляемое значение:	см. отдельные свойства (`font-style` и др.)

Эту проблему можно частично решить посредством группировки селекторов, но не проще ли объединить все в одно свойство? Введем свойство `font`, представляющее собой короткую форму записи всех остальных свойств шрифта (плюс еще кое-что).

Вообще говоря, в объявлении `font` могут задаваться любые значения каждого из перечисленных свойств шрифта или значение «системный шрифт» (описанное в разделе «Применение системных шрифтов»). Таким образом, предыдущий пример можно было бы сократить следующим образом:

```
h1 {font: italic 900 small-caps 30px Verdana, Helvetica, Arial, sans-serif;}
h2 {font: bold normal italic 24px Verdana, Helvetica, Arial, sans-serif;}
```

и получить точно такой же результат (рис. 5.24).

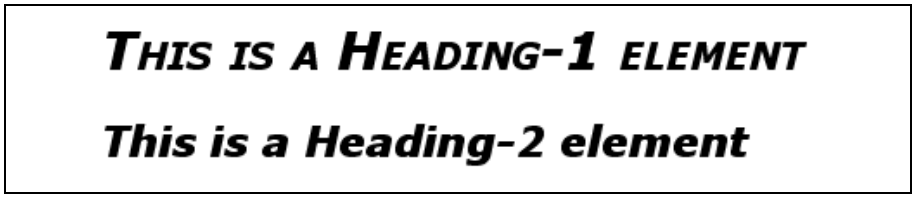


***Рис. 5.24.** Обычные правила оформления шрифта*

Я говорю, что запись стилей «можно было бы» сократить таким образом, потому что существуют еще некоторые другие возможности, обусловленные относительной свободой записи свойства `font`. Внимательно рассмотрев предыдущий пример, можно увидеть, что порядок задания первых трех значений различен. В правиле для `h1` первые три значения – `font-style`, `font-weight` и `font-variant` – размещены в указанной последовательности, тогда как во втором правиле они упорядочены так: `font-weight`, `font-variant` и `font-style`. Здесь нет ошибки, потому что эти три свойства могут быть записаны в произвольном порядке. Более того, если значение любого из них равно `normal`, оно может быть совсем опущено. Поэтому следующие правила эквивалентны предыдущему примеру:

```
h1 {font: italic 900 small-caps 30px Verdana, Helvetica, Arial, sans-serif;}
h2 {font: bold italic 24px Verdana, Helvetica, Arial, sans-serif;}
```

В этом примере значение `normal` в правиле для `h2` было пропущено, но результат совершенно аналогичен предыдущему примеру.

Однако важно понимать, что такая свобода касается только первых трех значений свойства `font`. Поведение двух последних подчиняется намного более строгим правилам. Значения `font-size` и `font-family` не только должны быть расположены в установленном порядке, но и обязаны всегда присутствовать в объявлении `font`. И точка, никаких вариантов. Если какое-либо из них будет пропущено, все правило будет признано недействительным, и, скорее всего, агент пользователя его

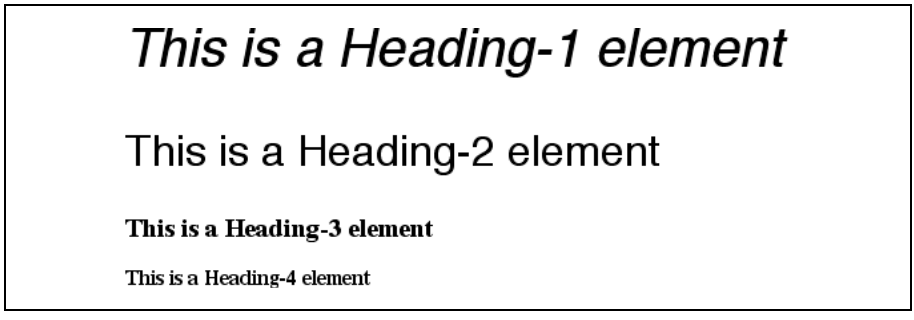


Рис. 5.25. *Необходимость наличия и размера, и семейства*

проигнорирует. Таким образом, следующие правила обеспечат результат, приведенный на рис. 5.25:

```
h1 {font: normal normal italic 30px sans-serif;}   /* здесь нет проблем */
h2 {font: 1.5em sans-serif;}   /* также все в порядке;
                                  опущены значения 'normal' */
h3 {font: sans-serif;}         /* НЕВЕРНО--отсутствует 'font-size' */
h4 {font: lighter 14px;}         /* НЕВЕРНО--отсутствует 'font-family' */
```

Установка высоты строки

До настоящего времени мы рассматривали свойство `font` так, как будто оно имеет всего пять значений, что не совсем верно. Существует еще возможность задания высоты строки с помощью свойства `font`, несмотря на то, что `line-height` – это свойство текста, а не шрифта. Высота строки задается как дополнение к значению `font-size`, отделенное от него прямым слэшем (/):

```
body {font-size: 12px;}
h2 {font: bold italic 200%/1.2 Verdana, Helvetica, Arial, sans-serif;}
```

Эти правила, проиллюстрированные на рис. 5.26, задают для всех элементов `h2` полужирное курсивное начертание (применяя гарнитуру одного из семейства рубленых шрифтов), свойству `font-size` – значение `24px` (вдвое больше размера элемента `body`), а свойству `line-height` – значение `30px`.

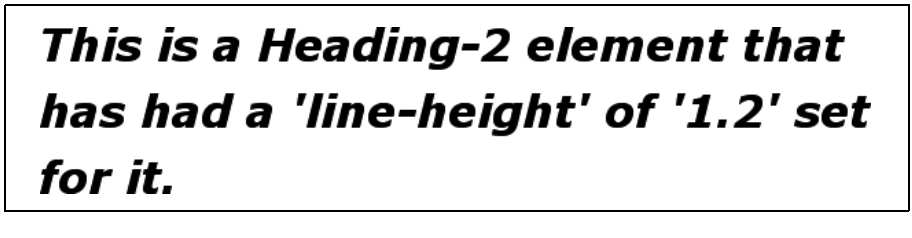


Рис. 5.26. *Введение высоты строки*

Значение `line-height` можно не указывать, как и первые три значения свойства `font`. Включив все-таки `line-height`, помните, что значение `font-size` всегда предшествует `line-height`, и эти два значения всегда разделяются наклонной чертой.

Возможно, я повторюсь, но это одна из самых распространенных ошибок авторов, использующих CSS, поэтому говорю еще раз: значения `font-size` и `font-family` свойства `font` являются обязательными, указываться они должны именно в этом порядке. Все остальные значения абсолютно необязательны.

Свойство `line-height` обсуждается в следующей главе.

Как правильно использовать сокращенную запись свойств

Важно помнить, что свойство `font`, поддерживая возможность записи в сокращенном виде, может демонстрировать неожиданные результаты в случае небрежного его применения. Рассмотрим следующие правила, которые проиллюстрированы на рис. 5.27:

```
h1, h2, h3 {font: italic small-caps 250% sans-serif;}
h2 {font: 200% sans-serif;}
h3 {font-size: 150%;}
```

```
<h1>This is an h1 element</h1>
<h2>This is an h2 element</h2>
<h3>This is an h3 element</h3>
```

Текст элемента `h2` не стал ни курсивом, ни капителью, и ни один из элементов не выделен полужирным шрифтом? И это правильно. При использовании сокращенной записи свойства `font` любым опущенным величинам присваиваются значения по умолчанию. Таким образом, предыдущий пример мог бы быть записан следующим образом, что абсолютно эквивалентно:

```
h1, h2, h3 {font: italic normal small-caps 250% sans-serif;}
h2 {font: normal normal normal 200% sans-serif;}
h3 {font-size: 150%;}
```

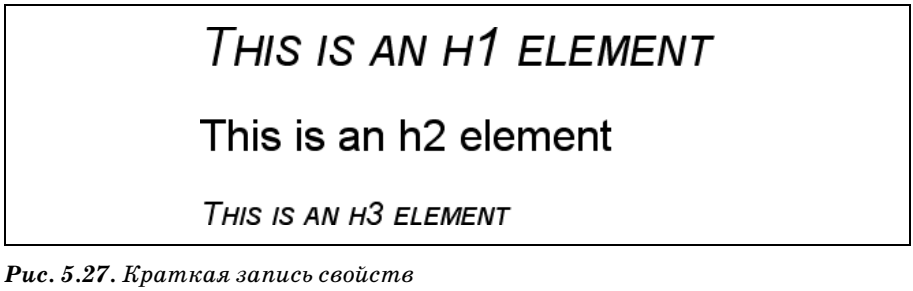


***Рис. 5.27.** Краткая запись свойств*

В соответствии с этими правилами стилю и варианту шрифта элемента h2 и свойству font-weight всех трех элементов присваивается значение normal. Это ожидаемое поведение при сокращенной записи свойств. Элемент h3 не постигла та же участь, потому что задано свойство font-size, которое не является сокращенной формой и, следовательно, оказывает влияние только на указанное значение.

Применение системных шрифтов

Если требуется создать веб-страницу, гармонирующую с операционной системой пользователя, значения системных шрифтов свойства font оказываются очень кстати. Они применяются, чтобы взять размер, семейство, насыщенность, стиль и вариант шрифта элементов операционной системы и применить их к элементу. Значения следующие:

caption

Применяется для элементов управления, имеющих надпись, таких как кнопки.

icon

Применяется для подписи значков.

menu

Используется в меню, т. е. выпадающих меню и списках меню.

message-box

Используется в диалоговых окнах.

small-caption

Используется для небольших элементов управления.

status-bar

Применяется в строках состояния окон.

Пусть требуется сделать шрифт кнопки таким же, как и у кнопок операционной системы:

```
button {font: caption;}
```

Имея в распоряжении эти значения, можно создавать веб-приложения, очень похожие на приложения операционной системы пользователя.

Обратите внимание, что системные шрифты можно задавать только как единое целое, т. е. семейство шрифта, размер, насыщенность, стиль и т. д. – все это задается вместе. Поэтому текст кнопки из нашего предыдущего примера будет выглядеть совершенно так же, как и текст кнопки операционной системы, независимо от того, соответствует или нет этот размер содержимому, располагающемуся вокруг кнопки. Однако, даже задав системный шрифт, отдельные значения можно менять. Так, применение следующего правила сделает размер шрифта кнопки таким же, как у родительского элемента:

```
button {font: caption; font-size: 1em;}
```

Если задан определенный системный шрифт, а на машине пользователя такого шрифта нет, агент пользователя может попытаться выполнить некоторое приближение, например, уменьшить размер шрифта `caption`, чтобы получить шрифт `small-caption`. Если такая аппроксимация невозможна, агент пользователя должен взять собственный применяемый по умолчанию шрифт. Если он может найти системный шрифт, но не может считать все его значения, тогда он должен взять его стандартное значение. Например, агент пользователя нашел шрифт `status-bar`, но не может получить информацию о том, является ли этот шрифт капителью. В этом случае агент пользователя присвоит свойству `small-caps` значение `normal`.

Стили пользовательского интерфейса более подробно обсуждаются в главе 13.

Сопоставление шрифтов

Мы уже видели, что CSS дает возможность сопоставлять семейства шрифтов, их насыщенность и варианты. Это выполняется с помощью весьма замысловатой процедуры сопоставления шрифтов. Ее понимание важно авторам, которые хотят помочь агентам пользователя осуществлять правильный выбор шрифтов при представлении их документов. Я отложил рассмотрение этого вопроса на конец главы, потому что понимание принципа работы свойств шрифтов на самом деле не имеет особой важности, и некоторые читатели, вероятно, пропустят эту часть и перейдут к следующей теме. Итак, для тех, чей интерес еще не угас, приводится описание сопоставления шрифтов:

1. Агент пользователя получает доступ к базе данных свойств шрифтов. В ней перечислены различные CSS-свойства всех шрифтов, к которым имеет доступ агент пользователя. Как правило, это шрифты, установленные на компьютере, хотя там могут находиться и другие (например, агент пользователя может иметь собственные встроенные шрифты). Если агент пользователя находит два идентичных шрифта, он просто проигнорирует один из них.
2. Агент пользователя анализирует элемент, к которому применены свойства шрифта, и создает список свойств, необходимых для представления этого элемента. На основании этого списка агент пользователя осуществляет начальный выбор семейства шрифтов, применяемых для отображения документа. Если существует полностью соответствующий этим параметрам шрифт, то его и выбирает агент пользователя. В противном случае ему надо сделать еще кое-что:
   a. Шрифт сначала сопоставляется по свойству `font-style`. Ключевому слову `italic` соответствует любой шрифт, отмеченный как «italic» или «oblique». Если среди доступных нет ни одного из таких шрифтов, сопоставление завершается неудачей.

   b. Следующая попытка подбора соответствия проводится для свойства `font-variant`. Любой шрифт, не отмеченный как «small-caps», считается `normal`. Любой шрифт, обозначенный как «small-caps», или который допускает синтез стиля капитель, или в котором строчные буквы заменяются прописными, может быть поставлен в соответствие по свойству `small-caps`.

   c. Далее подбираем соответствие по свойству `font-weight`. Благодаря способу обработки `font-weight` в CSS (рассмотренному ранее в этой главе), эта операция никогда не заканчивается неудачей.

   d. Затем беремся за свойство `font-size`. Сопоставление в данном случае должно проводиться с учетом определенного допускаемого отклонения, но устанавливает эту погрешность агент пользователя. Таким образом, один агент пользователя может допускать сопоставление с 20% диапазоном отклонения, тогда как другой может разрешать лишь 10% разницу между заданным и фактическим размерами.

3. Если на шаге 2 не найдено соответствие, агент пользователя ищет альтернативные шрифты в рамках того же семейства шрифтов. Если он находит какой-то адекватный шрифт, для него повторяются этапы шага 2.
4. Предположим, что общее соответствие найдено, но шрифт не обладает всем необходимым для отображения данного элемента, например ему не хватает символа обозначения авторского права, тогда агент пользователя возвращается к шагу 3, что означает поиск другой альтернативы и еще одно путешествие по шагу 2.
5. Наконец, если соответствие не найдено и все альтернативные шрифты проверены, агент пользователя выбирает применяемый по умолчанию шрифт для данного базового семейства шрифтов и делает все возможное, чтобы представить элемент правильно.

Весь этот процесс долог и утомителен, но он помогает понять, как агенты пользователя выбирают шрифты. Например, вы решили применить в документе Times или какой-нибудь другой шрифт антиква:

```
body {font-family: Times, serif;}
```

Агент пользователя должен проверить символы каждого из элементов и определить, может ли Times предоставить соответствующие символы. В большинстве случаев он может сделать это без проблем. Однако предположим, что в параграфе встретился китайский иероглиф. Шрифт Times ничего не может сопоставить с этим символом, поэтому агенту пользователя надо как-то обработать его или найти другой шрифт, который может представить этот элемент. Конечно, любой западный шрифт вряд ли содержит иероглифы, но наверняка существует шрифт (назовем его AsiaTimes), который агент пользователя мог бы применить для отображения этого элемента или просто одного символа. Таким образом, либо весь параграф может быть представлен с по-

мощью AsiaTimes, либо его текст может быть оформлен шрифтом Times, за исключением одного иероглифа, который отображается шрифтом AsiaTimes.

Правила гарнитуры шрифта

CSS2 предоставляет способ контроля за сопоставлением шрифтов посредством правила `@font-face`. Поскольку к весне 2003 года ни один веб-броузер полностью не реализовал его, `@font-face` было удалено из CSS2.1. Я не буду тратить на него много времени, поскольку детали этого правила очень сложны и их рассмотрению можно было бы, вероятно, посвятить целую главу (или даже книгу!).

Существуют четыре способа прийти к шрифту, который должен использоваться в документе. Кратко рассмотрим каждый из них, поскольку будущие версии CSS, возможно, будут применять эти механизмы и большинство средств визуализации SVG, по крайней мере частично, будут поддерживать сопоставление гарнитуры шрифтов, описанное в CSS2. Если вам надо реализовать `@font-face`, пожалуйста, обратитесь к спецификации CSS2 или любой самой свежей версии CSS (например, раздел CSS3 Web Fonts); приведенные далее описания в лучшем случае неполные.

Сопоставление шрифтов по имени

При сопоставлении шрифтов по имени агент пользователя выбирает доступный шрифт, имя семейства которого такое же, как и у запрашиваемого шрифта. Внешний вид и параметры этого шрифта могут отличаться. Этот метод описан в данном разделе выше.

Интеллектуальное сопоставление шрифтов

В этом случае агент пользователя останавливается на доступном шрифте, внешний вид которого наиболее полно соответствует запрашиваемому шрифту. Эти два шрифта могут не сопоставляться точно, но они должны быть максимально близки.

Информация, по которой сопоставляются два шрифта, включает тип шрифта (текст или символ), характер засечек, насыщенность, высоту прописных букв, x-высоту, вынос вверх, вынос вниз, наклон и т. д. Например, автор мог бы потребовать максимального соответствия определенного шрифта заданному наклону, написав:

```
@font-face {font-style: normal; font-family: "Times"; slope: -5;}
```

В этом случае, если Times не отвечает всем требованиям, агент пользователя будет искать обычный (прямой) шрифт антиква, наклон вправо которого был бы максимально близок к 5°. В CSS2 описано огромное количество различных параметров шрифтов, которые могут использоваться для проведения процесса сопоставления в поддерживающем их агенте пользователя.

Синтез шрифтов

Агент пользователя также может принять решение на лету генерировать шрифт, внешний вид и параметры которого соответствуют описанию, приведенному в правиле `@font-face`. Вот что говорит CSS2 об этом процессе:

> В этом случае агент пользователя создает шрифт, который практически полностью соответствует запрашиваемому шрифту не только по внешнему виду, но также и по параметрам. Информация синтеза включает информацию сопоставления и обычно требует более точных значений параметров, чем те, которые применяются для некоторых схем сопоставления. В частности, синтез требует точного задания значений ширины, символов для подстановки глифов и информации о позиционировании, если все особенности расположения указанного шрифта должны быть сохранены.

Если это понятно, тогда, скорее всего, мои разъяснения ни к чему. Если нет, вероятно, вам никогда не придется обращать на это внимания.

Загрузка шрифтов

В этом случае агент пользователя может загрузить удаленный шрифт, чтобы использовать его в документе. Чтобы объявить шрифт, предназначенный для загрузки, вы могли бы написать примерно следующее:

```
@font-face {font-family: "Scarborough Fair";
   src: url(http://www.example.com/fonts/ps/scarborough.ps);}
```

Затем вы могли бы применить этот шрифт по всему документу.

Даже в агенте пользователя, который допускает загрузку шрифтов, получение файла шрифта может занять некоторое время (такие файлы могут быть довольно большими), что замедлит процесс визуализации документа или, по крайней мере, задержит окончательное формирование визуального представления.

Заключение

Хотя авторы не могут быть уверены, что будут использованы конкретные шрифты, заданные в документе, они могут определить базовые семейства используемых шрифтов. Именно это поведение очень хорошо поддерживается, поскольку любой агент пользователя, не разрешающий авторам (или даже читателям) назначать шрифты, очень быстро оказался бы не у дел.

Что касается остальных возможностей работы со шрифтами, они поддерживаются по-разному. Изменение размера обычно работает хорошо, но то, что было сделано в этой области в XX веке, варьируется от удручающей простоты до почти точного соответствия. Разочаровывает авторов обычно не то, как поддерживается изменение размеров шрифтов, а то, как единицы измерения, которые они хотят использовать

(пункты), могут приводить к совершенно разным результатам на разных устройствах или даже разных операционных системах и агентах пользователя. Опасностей при применении пунктов масса; выбор единиц измерения длины при разработке веб-страниц вообще не очень хорошая идея. Процентные соотношения, единицы `em` и `ex` гораздо лучше подходят для обеспечения возможности изменения размеров шрифтов, поскольку они отлично масштабируются во всех распространенных средствах отображения.

Еще одним разочарованием можно считать отсутствие реализации механизма задания шрифтов, предназначенных для загрузки и использования в документе. Это значит, что авторы все еще зависят от шрифтов, имеющихся в распоряжении пользователя, и поэтому не могут предсказать, как будет выглядеть тот или иной текст.

Если говорить о стилевом оформлении текста, то его можно выполнить без привлечения шрифтов, о чем пойдет речь в следующей главе.

6

Свойства текста

Уверен, многие разработчики веб-страниц занимаются подбором подходящих цветов и стараются сделать свои страницы самыми стильными, но когда доходит до дела, львиная доля времени наверняка поглощается заботами о том, где будет располагаться текст и как он будет выглядеть. Эти вопросы обусловили появление таких тегов HTML, как <FONT> и <CENTER>, обеспечивающих определенную степень контроля над представлением и размещением текста.

По причине такой важности текста существует большое количество CSS-свойств, которые тем или иным образом касаются работы с ним. В чем разница между текстом и шрифтами? Текст – это содержимое, а шрифты нужны для отображения этого содержимого. Изменяя свойства текста, можно влиять на его положение, располагать символы выше позиции строки текста, подчеркивать текст и изменять регистр букв. В какой-то степени можно даже имитировать нажатие клавиши табуляции пишущей машинки.

Отступы и горизонтальное выравнивание

Для начала обсудим возможности управления горизонтальным размещением текста в строке. Будем считать эти основные действия шагами, которые можно предпринять для создания информационного бюллетеня или написания отчета.

Отступ

Отступ в первой строке абзаца веб-страницы – один из самых распространенных после форматирования текста эффектов. (На втором месте устранение пустой строки между абзацами, обсуждаемое в главе 7.) Некоторые сайты создают иллюзию структурированности текста, помещая перед первой буквой абзаца небольшие прозрачные изображе-

text-indent	
Значения:	<длина> | <процентное соотношение> | inherit
Начальное значение:	0
Область применения:	блочные элементы
Наследование:	да
Процентные соотношения:	относительно ширины содержащего блока
Вычисляемое значение:	для процентных соотношений – как задано; для длин – абсолютная длина

ния, которые сдвигают текст. На других сайтах применяют совершенно не соответствующий установленным стандартам[1] тег SPACER. Благодаря CSS появился более удачный способ структурирования текста – свойство text-indent.

В случае применения text-indent первая строка любого элемента может быть смещена на заданную величину, даже если эта величина отрицательная. Чаще всего это свойство применяется, конечно, для создания абзацного отступа в первой строке:

```
p {text-indent: 3em;}
```

Согласно этому правилу первая строка любого абзаца будет сдвинута на 3em, как показано на рис. 6.1.

В общем случае свойство text-indent можно применять к любому блочному элементу. Его нельзя применять к строковым элементам или к замещаемым элементам, таким как изображения. Но если в первой строке блочного элемента, например, абзаца, есть изображение, то оно будет сдвинуто вместе со всем остальным текстом строки.

Первую строку строчного элемента можно сдвинуть с помощью отступа слева или поля.

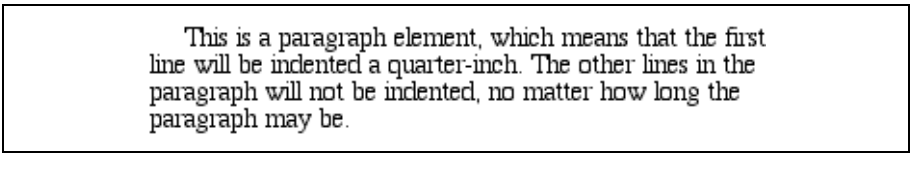


Рис. 6.1. *Абзац*

[1] Имеются в виду рекомендации консорциума W3C (*www.w3c.org*), признаваемые профессиональными разработчиками во всем мире как спецификации и стандарты. – *Примеч. науч. ред.*

Для свойства text-indent можно также задавать отрицательные значения, получая довольно интересные результаты. Чаще всего таким способом создают «выступы» – при этом первая строка выдвигается влево относительно края элемента:

```
p {text-indent: -4em;}
```

Задавая отрицательное значение text-indent, будьте аккуратны: первые три слова («This is a») могут быть отсечены левым краем окна броузера. Во избежание неприятностей для создания отрицательного абзацного отступа я рекомендую применять поля:

```
p {text-indent: -4em; padding-left: 4em;}
```

Однако отрицательные отступы могут быть полезны. Рассмотрим следующий пример, показанный на рис. 6.2, в котором к тексту добавляется «обтекаемое» изображение:

```
p.hang {text-indent: -25px;}

<img src="star.gif" style="width: 60px; height: 60px;
float: left;" alt="An image of a five-pointed star."/>
<p class="hang"> This paragraph has a negatively indented first line, which
overlaps the floated image that precedes the text.  Subsequent lines do not
overlap the image, since they are not indented in any way.</p>
```

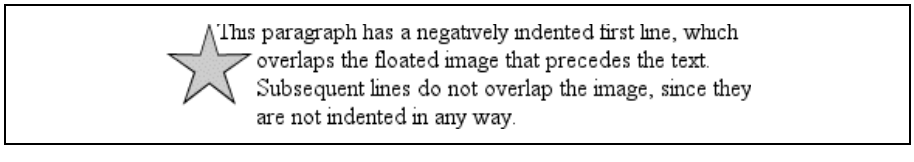


Рис. 6.2. *Плавающее изображение и отрицательный отступ*

Эта простая методика позволяет реализовывать некоторые интересные конструкции.

С text-indent могут применяться любые единицы измерения длины, в том числе процентные значения. В следующем случае процентное значение вычисляется относительно ширины родительского элемента. Иначе говоря, если значение отступа задается равным 10%, то первая строка элемента будет сдвинута на 10% ширины его родительского элемента, как показано на рис. 6.3:

```
div {width: 400px;}
p {text-indent: 10%;}

<div>
```

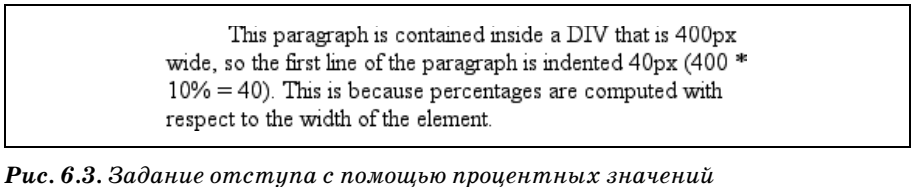


Рис. 6.3. *Задание отступа с помощью процентных значений*

```
<p>This paragraph is contained inside a DIV, which is 400px wide, so the
first line of the paragraph is indented 40px (400 * 10% = 40).  This is
because percentages are computed with respect to the width of the element.</p>
</div>
```

Обратите внимание, что такая организация абзаца применяется только к первой строке элемента, даже если вставить разрывы строки. Самое интересное, что свойство `text-indent` является наследуемым, и это может приводить к неожиданным эффектам. Например, рассмотрим разметку, проиллюстрированную на рис. 6.4:

```
div#outer {width: 500px;}
div#inner {text-indent: 10%;}
p {width: 200px;}
```

```
<div id="outer">
<div id="inner">
This first line of the DIV is indented by 50 pixels.
<p>
This paragraph is 200px wide, and the first line of the paragraph
is indented 50px.  This is because computed values for 'text-indent'
are inherited, instead of the declared values.
</p>
</div>
</div>
```

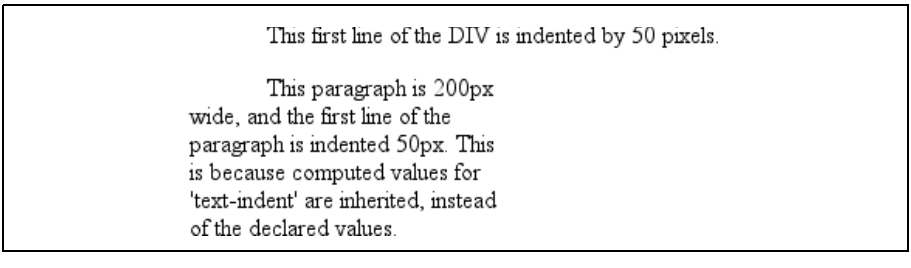


Рис. 6.4. Унаследованный отступ

В версиях CSS, предшествующих 2.1, свойство `text-indent` всегда наследует вычисляемое значение, а не объявляемое.

Горизонтальное выравнивание

Еще более фундаментальным, чем `text-indent`, является свойство `text-align`, определяющее, как выравниваются строки текста элемента относительно границ блока. Первые три значения совершенно просты, но с четвертым и пятым связаны некоторые сложности.

text-align	
Значения CSS2.1:	`left | center | right | justify | inherit`
Значения CSS2:	`left | center | right | justify |` <строка> `| inherit`
Начальное значение:	зависит от агента пользователя; также может зависеть от направления написания
Область применения:	блочные элементы
Наследование:	да
Вычисляемое значение:	как задано
Примечание:	CSS2 включало значение <строка>, которое было выброшено из CSS2.1 из-за отсутствия реализации

Быстрее всего можно понять суть этих значений, если посмотреть на рис. 6.5.

Очевидно, что значения `left`, `right` и `center` выравнивают текст элементов так, как следует из их названий. Поскольку `text-align` применяется только к блочным элементам, таким как абзацы, невозможно выровнять ссылку по центру строки без выравнивания всей строки (да это и нежелательно, т. к., скорее всего, приведет к наложению текста).

Для западных языков, которые читаются слева направо, по умолчанию устанавливается значение `left`. Текст выравнивается по левой границе и имеет неровный правый край (иначе известен как «левосторонний» текст). Для таких языков, как иврит и арабский, по умолча-

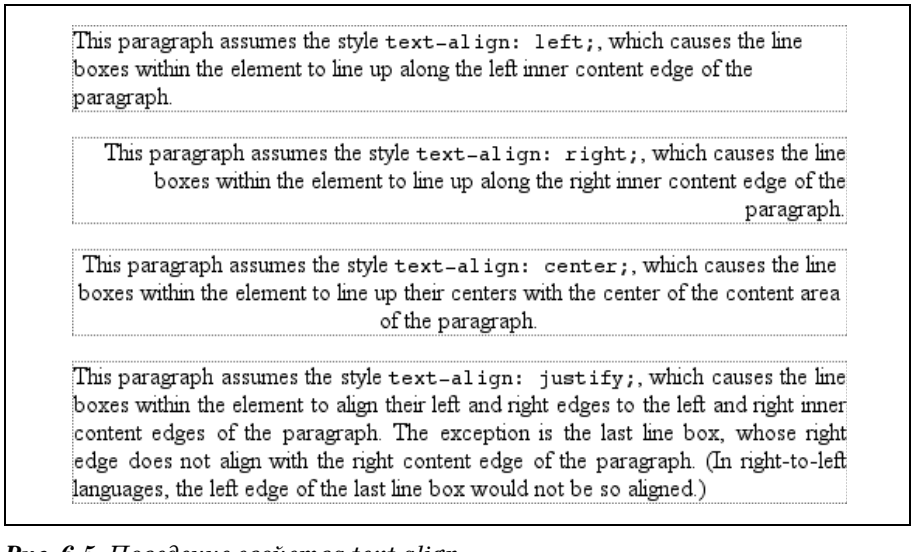


Рис. 6.5. *Поведение свойства text-align*

нию задается значение `right`, поскольку они читаются справа налево. Значение `center`, что не удивляет, центрирует каждую строку текста элемента.

Центрирование блочных или табличных элементов осуществляется путем задания правых и левых полей этих элементов. За более подробной информацией обращайтесь к главе 7.

Не исключено, что выражение `text-align: center` на первый взгляд может показаться аналогичным элементу `<CENTER>`, но на самом деле они сильно отличаются. `<CENTER>` оказывает влияние не только на текст, но центрирует элементы в целом, например таблицы. Свойство `text-align` управляет выравниванием содержимого, а не самих элементов. Это ясно видно на рис. 6.5. Сами элементы не сдвинуты в сторону, затронут только их текст.

Одна из самых пагубных ошибок в IE/Win вплоть до версии IE6 в том, что он интерпретирует `text-align: center` как элемент `<CENTER>` и центрирует и элементы, и текст. Этого не происходит в обычном режиме печати в IE6 и более поздних версиях, но сохраняется в IE5.x и более ранних версиях.

Последний вариант горизонтального выравнивания – `justify`, и с ним связаны некоторые особые вопросы. В выровненном по ширине тексте оба конца строки размещаются вплотную к внутренним краям родительского элемента, как показано на рис. 6.6. Затем пробелы между словами и буквами корректируются так, чтобы длина всех строк была строго одинакова. Выравнивание текста по ширине применяется часто (например, в этой книге), но с появлением CSS начали действовать некоторые дополнительные соображения.

Агент пользователя – а не CSS – определяет, каким образом будет растянут выровненный по ширине текст, чтобы заполнить пространство меж-

This is a paragraph of justified text. Notice that the spacing between words, or even between individual letters, depends greatly on the number of words in each line. Intraword and intracharacter spacing is adjusted to create the justification effect, so it can effectively override values for properties such as **`word-spacing`** and **`letter-spacing`**.

Рис. 6.6. *Текст, выровненный по ширине*

ду левым и правым краями родительского элемента. Некоторые броузеры увеличивают только отступы между словами, тогда как другие могут увеличивать интервал между буквами (хотя спецификация CSS особо подчеркивает, что «агенты пользователя не могут увеличивать или уменьшать расстояние между символами», если задано значение свойства `letter-spacing`). Некоторые агенты пользователя могут сужать строки, сжимая текст немного больше обычного. Все эти варианты влияют на представление элемента и даже могут изменить его высоту в зависимости от того, сколько строк текста получается в результате применения того или иного варианта выравнивания агентом пользователя.

CSS также не определяет, как должны расставляться переносы.[1] Как правило, в текстах, выровненных по ширине, переносы расставляются для разбиения длинных слов, благодаря чему уменьшается расстояние между словами и улучшается внешний вид строки. Однако, поскольку CSS не определяет правила расстановки переносов, агенты пользователя вряд ли будут делать это автоматически. В результате выровненный по ширине текст в случае применения CSS выглядит намного менее привлекательным, чем при печати, особенно когда элементы становятся настолько узкими, что в каждую строку может вместиться лишь несколько слов. Конечно, применение узких элементов не запрещается, но следует помнить об этих недостатках.

Вертикальное выравнивание

Итак, мы рассмотрели горизонтальное выравнивание и переходим к вертикальному. Строки намного подробнее обсуждаются в главе 7, поэтому здесь приводится лишь краткий обзор.

Высота строк

Свойство `line-height` задает расстояние между базовыми линиями строк текста, а не размер шрифта, и определяет величину, на которую увеличивается или уменьшается высота блока каждого элемента. В самых простых случаях задание `line-height` позволяет увеличить (или уменьшить) расстояние по вертикали между строками текста, но не надо обольщаться: в работе `line-height` все не так просто. Свойство `line-height` управляет *межстрочным интервалом* (*leading*), или *интерлиньяжем*, – дополнительным расстоянием между строками текста над и под шрифтом. Иначе говоря, межстрочный интервал – это разница между значением `line-height` и размером шрифта.

В случае применения к блочным элементам свойство `line-height` определяет минимальное расстояние между базовыми линиями текста это-

[1] Расстановка переносов не описана в CSS, потому что эти правила зависят от языка. Не пытаясь выдумывать набор правил, которые, скорее всего, были бы несовершенными, спецификация пока оставила проблему без решения.

line-height

Значения:	<длина> | <процентное значение> | <число> | `normal` | `inherit`
Начальное значение:	`normal`
Область применения:	все элементы (смотрите текст, касающийся замещаемых и блочных элементов)
Наследование:	да
Процентные значения:	относительно размера шрифта элемента
Вычисляемое значение:	для значений длины и процентных соотношений – абсолютное значение; в других случаях – как задано

го элемента. Обратите внимание, что свойство определяет минимум, а не абсолютное значение, и расстояние между базовыми линиями текста может намного превышать значение `line-height`. Свойство `line-height` не оказывает влияния на компоновку замещаемых элементов, но тем не менее применяется к ним. (Разгадка этой хитрой загадки дается в главе 7.)

Конструирование строк

Каждый элемент строки текста генерирует *область содержимого* (*content area*), которая определяется размером шрифта. Эта область содержимого, в свою очередь, генерирует *строковый блок* (*inline box*), который в отсутствие каких-либо других факторов эквивалентен области содержимого. Один из факторов, увеличивающих или уменьшающих высоту каждого строкового блока, – межстрочный интервал, генерируемый `line-height`.

Чтобы определить межстрочный интервал для заданного элемента, достаточно найти разность между вычисляемым значением `line-height` и `font-size`. Это значение и есть общая величина межстрочного интервала. И помните, это может быть и отрицательное число. Межстрочный интервал обычно делится на два, и каждая половина добавляется к области содержимого сверху и снизу. В результате получается строковый блок данного элемента.

В качестве примера, скажем, `font-size` (и, следовательно, область содержимого) составляет 14 пикселов в высоту, а вычисленное значение `line-height` – 18 пикселов. Разница (четыре пиксела) делится на два, и каждая половина добавляется к области содержимого снизу и сверху. В результате создается строковый блок высотой 18 пикселов с двумя дополнительными пикселами над и под областью содержимого. Описание принципа работы `line-height` выглядит очень многословным, но на это есть веские причины.

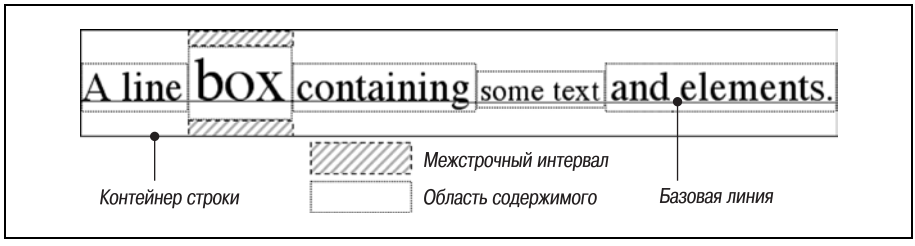


Рис. 6.7. *Схема контейнера строки*

Все сгенерированные строковые блоки данной строки содержимого рассматриваются в составе контейнера строки. Высота контейнера строки как раз такова, чтобы охватить верх самого высокого строкового блока и низ расположенного ниже всех строкового блока. На рис. 6.7 приведена схема.

Задаем значение line-height

Теперь рассмотрим возможные значения свойства `line-height`. Если используется значение по умолчанию `normal`, то агент пользователя должен вычислить вертикальное расстояние между строками. Значение может варьироваться в зависимости от агента пользователя (обычно оно в 1,2 раза больше размера шрифта); оно определяет превышение высоты контейнеров строк над значением `font-size` для данного элемента.

Большинство значений задается в простых единицах измерения длины (например, `18px` или `2em`). Помните, что даже если указаны правильные значения длины, такие как `4cm`, броузер (или операционная система) может искажать реальные размеры, и высота строки на мониторе может оказаться не равной 4 сантиметрам. Более подробно об этом написано в главе 4.

Процентные значения `em` и `ex` вычисляются относительно значения свойства `font-size` элемента. Следующая разметка относительно проста, ее результаты приведены на рис. 6.8:

```
body {line-height: 14px; font-size: 13px;}
p.cl1 {line-height: 1.5em;}
p.cl2 {font-size: 10px; line-height: 150%;}
p.cl3 {line-height: 0.33in;}
```

```
<p>This paragraph inherits a 'line-height' of 14px from the body, as well as
a 'font-size' of 13px.</p>
<p class="cl1">This paragraph has a 'line-height' of 19,5px(13 * 1.5), so it
will have slightly more line-height than usual.</p>
<p class="cl2">This paragraph has a 'line-height' of 15px (10 * 150%), so it
will have slightly more line-height than usual.</p>
<p class="cl3">This paragraph has a 'line-height' of 0.33in, so it will have
slightly more line-height than usual.</p>
```

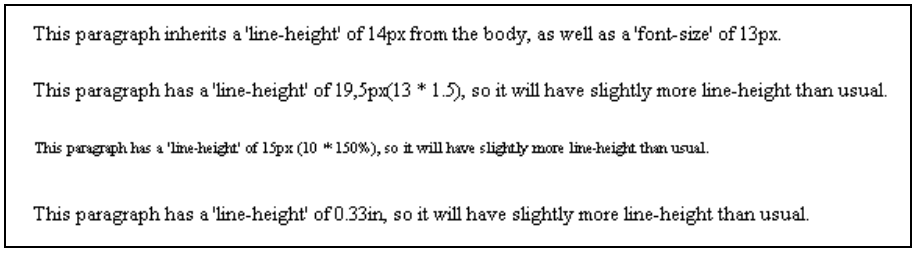


Рис. 6.8. *Простые вычисления с использованием свойства line-height*

Высота строки и наследование

Если свойство `line-height` наследуется одним блочным элементом от другого, все становится еще запутаннее. Свойство `line-height` наследует вычисленные значения родительского элемента, а не дочернего. Результаты следующей разметки показаны на рис. 6.9. Скорее всего, это не то, что имел в виду автор:

```
body {font-size: 10px;}
div {line-height: 1em;}  /* получается '10px' */
p {font-size: 18px;}

<div>
<p>This paragraph's 'font-size' is 18px, but the inherited 'line-height'
value is only 10px. This may cause the lines of text to overlap each other by
a small amount.</p>
</div>
```

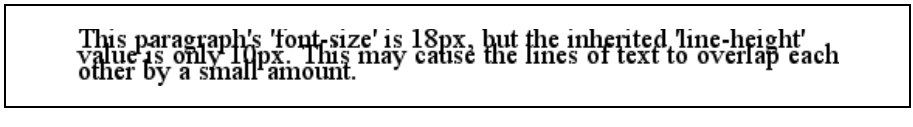


Рис. 6.9. *Небольшое значение line-height, большое font-size, некоторые трудности*

Почему строки расположены так близко друг к другу? Потому что значение `line-height`, равное `10px`, было унаследовано абзацем от его родителя `div`. Одно из решений, позволяющее справиться с трудностями, порожденными `line-height`, состоит в том, чтобы явно задать `line-height` для каждого элемента, но это непрактично. Лучше определять число, которое задает коэффициент масштабирования:

```
body {font-size: 10px;}
div {line-height: 1;}
p {font-size: 18px;}
```

Если при задании числа не указать единицы измерения, то вместо вычисляемого значения наследуется коэффициент масштабирования. Этот коэффициент будет применяться к элементу и ко всем его потом-

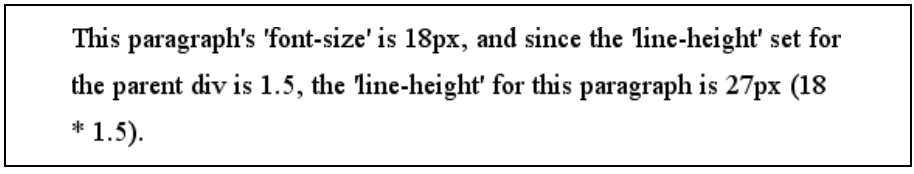


Рис. 6.10. *Применение коэффициентов line-height для решения проблем наследования*

кам, поэтому `line-height` каждого элемента будет вычислено относительно его собственного `font-size` (рис. 6.10):

```
div {line-height: 1.5;}
p {font-size: 18px;}
```

```
<div>
<p>This paragraph's 'font-size' is 18px, and since the 'line-height' set for
the parent div is 1.5, the 'line-height' for this paragraph is 27px (18 *
1.5).</p>
</div>
```

Кажется, что `line-height` выделяет дополнительное пространство над и под каждой строкой текста, но на самом деле он добавляет (или вычитает) определенную величину над и под областью содержимого строковых элементов для создания строкового блока. Предположим, что по умолчанию задано значение `font-size`, равное `12pt`, и рассмотрим следующее правило:

```
p {line-height: 16pt;}
```

Поскольку унаследованная высота строки 12-пунктного текста составляет 12 пунктов, предыдущее правило добавит к каждой строке текста в абзаце поле в 4 пункта. Эта дополнительная величина разбивается пополам, и одна половина располагается над каждой строкой, а другая половина – под строкой. Теперь расстояние между базовыми линиями составляет 16 пунктов, что является косвенным результатом распределения дополнительного пространства.

Если задать значение `inherit`, то элемент унаследует вычисляемое значение своего родительского элемента. Это не слишком отличается от обычного наследования значения, за исключением того, что касается специфичности и каскада. Эти вопросы подробно рассмотрены в главе 3.

Итак, вы овладели азами построения строк, и пора поговорить о вертикальном выравнивании элементов относительно контейнера строки.

Вертикальное выравнивание текста

Если вам приходилось иметь дело с элементами `sup` и `sub` (элементами верхнего и нижнего индексов) или с изображением с подобной разметкой: `<img src="foo.gif" align="middle">`, то вы уже производили некоторое простейшее вертикальное выравнивание. В CSS свойство `vertical-`

vertical-align	
Значения:	baseline | sub | super | top | text-top | middle | bottom | text-bottom | <процентное значение> | <длина> | inherit
Начальное значение:	baseline
Область применения:	строковые элементы и ячейки таблиц
Наследование:	нет
Процентные соотношения:	относительно значения line-height элемента
Вычисляемое значение:	для процентных значений и длин – абсолютная длина; в остальных случаях – как задано
Примечание:	если применяется к ячейкам таблицы, распознаются только значения baseline, top, middle и bottom

align применяется только к строковым и замещаемым элементам, таким как изображения и поля ввода формы. Свойство vertical-align не является наследуемым свойством.

Свойство vertical-align принимает любое из восьми ключевых слов, процентные значения или значения длин. В vertical-align применяются как уже знакомые ключевые слова, так и новые: baseline (значение по умолчанию), sub, super, bottom, text-bottom, middle, top и text-top. Посмотрим, какое действие оказывает каждое из этих ключевых слов на строковые элементы.

Запомните: vertical-align *не* влияет на выравнивание содержимого блочного элемента. Тем не менее вы можете применять его, чтобы менять вертикальное выравнивание элементов, находящихся в ячейках таблиц. Более подробно об этом рассказано в главе 11.

Выравнивание по базовой линии

Объявление vertical-align: baseline выравнивает базовую линию элемента по базовой линии его родителя. Броузеры по большей части делают это в любом случае, поскольку скорее всего ожидается, что низ всех текстовых элементов строки будет выровнен.

Если у выровненного по вертикали элемента нет базовой линии, т. е. если это изображение, поле ввода формы или другой замещаемый элемент, тогда по базовой линии родителя выравнивается низ элемента, как показано на рис. 6.11:

```
img {vertical-align: baseline;}

<p>The image found in this paragraph <img src="dot.gif" alt="A dot" /> has
its bottom edge aligned with the baseline of the text in the paragraph.</p>
```

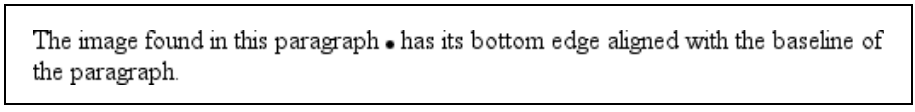


Рис. 6.11. *Выравнивание изображения по базовой линии*

Правило выравнивания важно учитывать, потому что оно заставляет некоторые броузеры всегда помещать нижний край замещаемых элементов на базовой линии, даже если в строке нет другого текста. Допустим, что в ячейке таблицы имеется только изображение. Оно может располагаться на базовой линии, но в некоторых броузерах это приводит к появлению зазора под изображением. Другие броузеры «пакуют» изображение в ячейку таблицы без всяких зазоров. Зазор предусмотрен требованиями CSS Working Group, хотя это и не нравится большинству авторов.

Более подробно зазоры и способы, позволяющие избежать их появления, обсуждаются в моей статье «Images, Tables, and Mysterious Gaps», находящейся по адресу *http://developer.mozilla.org/en/docs/Images,_Tables,_and_Mysterious_Gaps*. Вопросы компоновки строк также рассматриваются в главе 7.

Надстрочные и подстрочные элементы

Объявление `vertical-align: sub` превращает элемент в подстрочный. Это означает, что его базовая линия (или низ, если это замещаемый элемент) опущена относительно базовой линии его родителя. Спецификация не определяет величину понижения элемента, поэтому она может меняться в зависимости от агента пользователя.

Значение `super` противоположно `sub`; оно поднимает базовую линию элемента (или низ замещаемого элемента) относительно базовой линии родителя. Опять же расстояние, на которое поднимается текст, зависит от агента пользователя.

Заметьте, что значения `sub` и `super` *не меняют* размер шрифта элемента, так что подстрочный или надстрочный текст не станет меньше (или больше). Напротив, любой текст под- или надстрочного элемента должен по умолчанию иметь такой же размер, что и текст родительского элемента, как показано на рис. 6.12:

```
span.raise {vertical-align: super;}
span.lower {vertical-align: sub;}
```

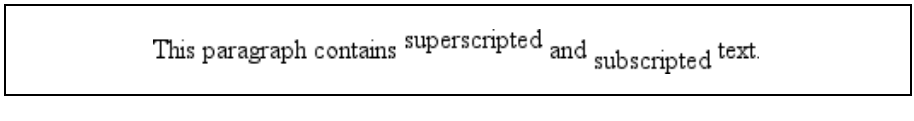


Рис. 6.12. *Подстрочное и надстрочное выравнивание*

```
<p>This paragraph contains <span class="raise">superscripted</span>
and <span class="lower">subscripted</span> text.</P>
```

Для того чтобы сделать над- или подстрочный текст меньшим, чем текст его родительского элемента, надо обратиться к свойству `font-size`, рассмотренному в главе 5.

Выравнивание по низу

Объявление `vertical-align: bottom` выравнивает низ строкового блока элемента по низу контейнера строки. Например, результат применения следующей разметки показан на рис. 6.13:

```
.feeder {vertical-align: bottom;}

<p>This paragraph, as you can see quite clearly, contains a <img src="
tall.gif" alt="tall" class="feeder" /> image and a <img src="short.gif" alt="
short" class="feeder" /> image, and then some text that is not tall.</p>
```

Вторая строка абзаца на рис. 6.13 содержит два строковых элемента, нижние края которых выровнены относительно друг друга. Они также находятся ниже базовой линии текста.

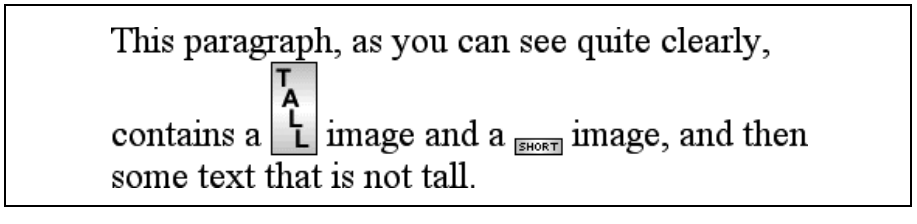


Рис. 6.13. *Выравнивание по низу*

Значение `vertical-align: text-bottom` измеряется относительно нижней линии текста строки. При этом замещаемые или любые другие нетекстовые элементы игнорируются. Вместо этого учитывается «применяемый по умолчанию» текстовый блок. Этот стандартный блок происходит от `font-size` родительского элемента, и низ строкового блока выровненного элемента выравнивается относительно низа стандартного текстового блока. Таким образом, применив показанную ниже разметку, мы получаем результат, представленный на рис. 6.14:

```
img.tbot {vertical-align: text-bottom;}
```

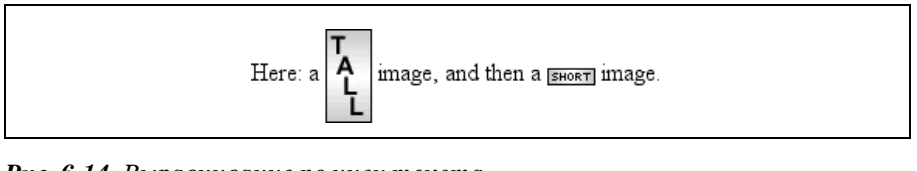


Рис. 6.14. *Выравнивание по низу текста*

```
<p>Here: a <img src="tall.gif" style="vertical-align: middle;" alt="tall" />
image, and then a <img src="short.gif" class="tbot" alt="short" /> image.</p>
```

Выравнивание по верхнему краю

Применение выражения `vertical-align: top` имеет эффект, противоположный значению `bottom`. Аналогично `vertical-align: text-top` – противоположность `text-bottom`. На рис. 6.15 показано, как могла бы быть представлена следующая разметка:

```
.up {vertical-align: top;}
.textup {vertical-align: text-top;}

<p>Here: a <img src="tall.gif" alt="tall image"> tall image, and then <span
class="up">some text</span> that's been vertically aligned.</p>
<p>Here: a <img src="tall.gif" class="textup" alt="tall"> image that's been
vertically aligned, and then a <img src="short.gif" class="textup" alt="
short" /> image that's similarly aligned.</p>
```

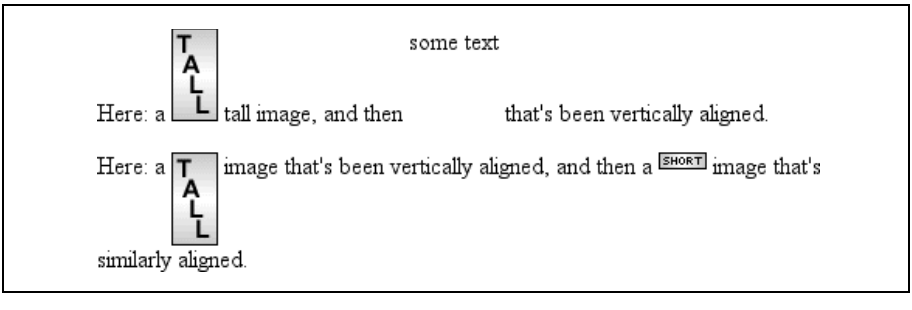


***Рис. 6.15.** Выравнивание по верху и верху текста строки*

Точное размещение при таком выравнивании, естественно, будет зависеть от того, какие элементы находятся в строке, насколько они высоки, и от размера шрифта родительского элемента.

В середине

Значение `middle` обычно (но не всегда) применяется к изображениям. Оказываемый им эффект не совсем такой, какого можно ожидать, если судить по его имени. Значение `middle` выравнивает середину блока строкового элемента по точке, которая находится на расстоянии `0.5ex` над базовой линией родительского элемента, где `1ex` определяется относительно `font-size` родительского элемента. Более подробно это показано на рис. 6.16.

Большинство агентов пользователя интерпретируют `1ex` как половину `em`, поэтому применение `middle` обычно выравнивает среднюю точку элемента с точкой, находящейся на четверть `em` выше базовой линии родителя. Однако не полагайтесь на это, поскольку некоторые агенты

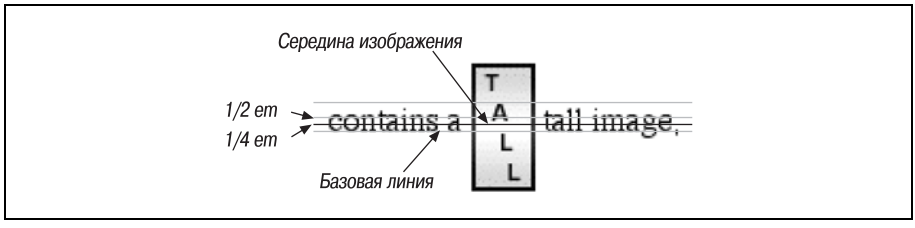


Рис. 6.16. *Детали выравнивания по середине*

пользователя на самом деле вычисляют точную x-высоту для каждого элемента. (Подробнее об x-высоте рассказывается в главе 5.)

Процентные значения

Процентные значения не позволяют моделировать `align="middle"` для изображений. Вместо этого задание процентного значения в `vertical-align` поднимает или опускает базовую линию элемента (или нижний край замещаемого элемента) относительно базовой линии родительского элемента на объявленную величину. (Задаваемое процентное значение вычисляется как часть `line-height` элемента, *а не его родителя*.) Если процентное значение положительное, то базовая линия элемента поднимается, а если отрицательное, то опускается. В зависимости от того, на какую величину поднимается или опускается текст, он может оказаться на соседней строке, как показано на рис. 6.17, поэтому будьте осторожны, задавая процентные значения.

```
sub {vertical-align: -100%;}
sup {vertical-align: 100%;}

<p>We can either <sup>soar to new heights</sup> or, instead, <sub>sink into
despair...</sub></p>
```

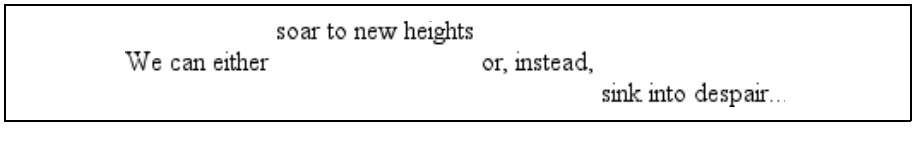


Рис. 6.17. *Забавные эффекты при использовании процентных значений*

Рассмотрим процентные значения подробнее. Предположим следующее:

```
<div style="font-size: 14px; line-height: 18px;">
Я чувствовал, что, по крайней мере, заслужил <span style="vertical-align:
50%;">повышения</span> за мои труды.
</div>
```

Базовая линия элемента `span`, значение `vertical-align` которого равно `50%`, поднимается на девять пикселов, что составляет половину унаследованного элементом значения `line-height: 18px`, а *не семь пикселов*.

Выравнивание по заданному расстоянию

Рассмотрим вертикальное выравнивание на заданное расстояние. Свойство `vertical-align` очень простое: оно перемещает элемент вверх или вниз на указанное расстояние. Таким образом, выражение `vertical-align: 5px;` сдвинет элемент вверх на пять пикселов относительно исходного положения. Отрицательные значения длины сдвигают элемент вниз. Этой простой формы выравнивания не было в CSS1, но она добавлена в CSS2.

Важно понимать, что выровненный по вертикали текст не становится частью другой строки и не перекрывает текст других строк. Взгляните на рис. 6.18, где часть выровненного по вертикали текста находится в середине абзаца.

Как видите, любой вертикально выровненный элемент может влиять на высоту строки. Вспомним описание контейнера строки, высота которого именно такова, чтобы вместить верх самого высокого строкового блока и низ самого низкого строкового блока. Это касается и строковых блоков, которые были сдвинуты вверх или вниз при вертикальном выравнивании.

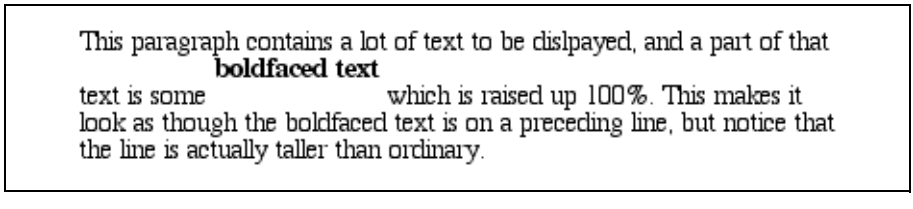


Рис. 6.18. *Вертикальное выравнивание может привести к увеличению высоты строк*

Расстояние между буквами и словами

Покончив с выравниванием, обратимся к управлению расстояниями между словами и буквами. У этих свойств тоже есть своя специфика.

Расстояния между словами

Свойство `word-spacing` допускает положительные или отрицательные значения длины. Это значение *добавляется* к стандартному расстоянию между словами. В сущности, `word-spacing` служит для *изменения* расстояния между словами. Поэтому применяемое по умолчанию значение `normal` аналогично нулевому значению (`0`).

Если задать положительное значение длины, расстояние между словами увеличится. А если задать отрицательное значение для `word-spacing`, то слова сдвинутся ближе друг к другу:

word-spacing

Значения:	<длина> | normal | inherit
Начальное значение:	normal
Область применения:	все элементы
Наследование:	да
Вычисляемое значение:	для normal – абсолютная длина 0; в противном случае – абсолютная длина

```
p.spread {word-spacing: 0.5em;}
p.tight {word-spacing: -0.5em;}
p.base {word-spacing: normal;}
p.norm {word-spacing: 0;}
```

```
<p class="spread">The spaces between words in this paragraph will be
   increased by 0.5em.</p>
<p class="tight">The spaces between words in this paragraph will be decreased
   by 0.5em.</p>
<p class="base">The spaces between words in this paragraph will be normal.</p>
<p class="norm">The spaces between words in this paragraph will be normal.</p>
```

Эффект, оказываемый этими настройками, показан на рис. 6.19.

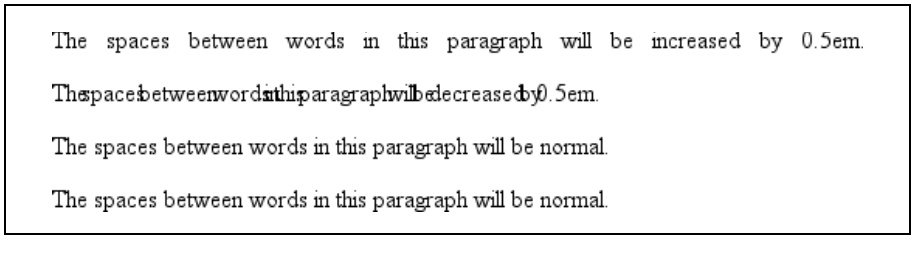


Рис. 6.19. *Изменение расстояния между словами*

До сих пор я не дал точного определения термину «слово». В терминологии CSS «слово» – это любая строка, не содержащая символов-разделителей и ограниченная такими символами с обеих сторон. Это определение не относится к семантике, оно просто предполагает, что документ содержит слова, окруженные одним или несколькими символами-разделителями. Нельзя ожидать от поддерживающего CSS агента пользователя, что он сможет во всех случаях отличить слово от не-слова в данном языке. Это определение, каким бы оно ни было, означает, что word-spacing вряд ли будет работать в языках, основанных на пиктографии или на нероманских стилях написания. Это свойство позволяет создавать совершенно нечитаемые документы, что ясно видно на рис. 6.20. Следует с осторожностью применять word-spacing.

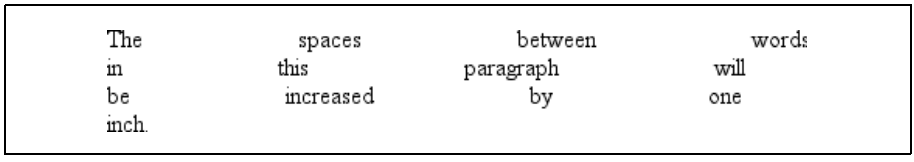


Рис. 6.20. *Действительно большие расстояния между словами*

Расстояние между буквами

Многие проблемы, встречающиеся при работе с `word-spacing`, имеют место в случае применения `letter-spacing`. Единственное отличие между ними состоит в том, что свойство `letter-spacing` изменяет расстояния между символами или буквами.

letter-spacing

Значения:	<длина> | `normal` | `inherit`
Начальное значение:	`normal`
Область применения:	все элементы
Наследование:	да
Вычисляемое значение:	для значений длины – абсолютная длина; в противном случае – `normal`

Как и для свойства `word-spacing`, допустимым значением `letter-spacing` является любая длина. По умолчанию указывается ключевое слово `normal` (что аналогично `letter-spacing: 0`). Любое вводимое значение длины будет увеличивать или уменьшать расстояние между буквами на эту величину. На рис. 6.21 показаны результаты применения следующей разметки:

```
p {letter-spacing: 0;}    /* идентично 'normal' */
p.spacious {letter-spacing: 0.25em;}
p.tight {letter-spacing: -0.25em;}

<p>The letters in this paragraph are spaced as normal.</p>
<p class="spacious">The letters in this paragraph are spread out a bit.</p>
<p class="tight">The letters in this paragraph are a bit smashed together.</p>
```

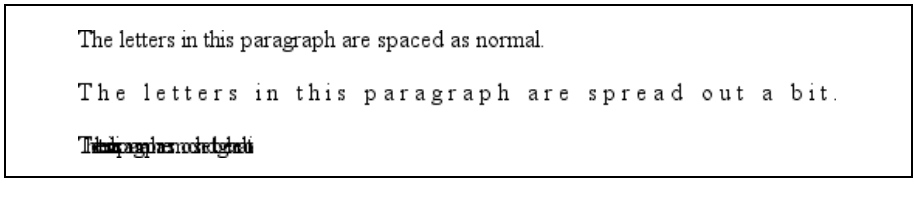


Рис. 6.21. *Разные расстояния между буквами*

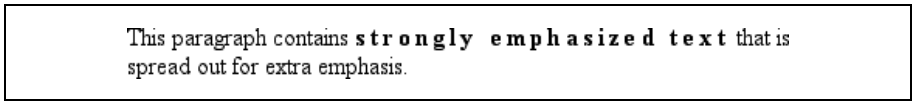


Рис. 6.22. *Применение свойства letter-spacing для увеличения выразительности выделенного текста*

Применение `letter-spacing` для повышения выразительности выделения – это методика, освященная временем. Можно было бы написать следующее объявление и получить эффект, показанный на рис. 6.22:

```
strong {letter-spacing: 0.2em;}

<p>This paragraph contains <strong>strongly emphasized text</strong> that is
spread out for extra emphasis.</p>
```

Расстояние между словами и буквами и выравнивание

На значение `word-spacing` может оказывать влияние значение свойства `text-align`. Если элемент выровнен по ширине, пробелы между буквами и словами корректируются так, чтобы текст занимал всю строку. Это может в свою очередь изменить параметры, определяющие расстояние между словами и буквами, объявленные автором в свойствах `word-spacing`. Если задано значение `letter-spacing`, оно не может быть изменено `text-align`, но если `letter-spacing` имеет значение `normal`, расстояние между символами может меняться для обеспечения выравнивания текста по ширине. CSS не указывает, как должны вычисляться пробелы, поэтому агент пользователя просто создает их.

Дочерний элемент, как обычно, наследует вычисленное значение своего родителя. Нельзя сделать так, чтобы вместо вычисленного значения свойств `word-spacing` или `letter-spacing` наследовался коэффициент масштабирования (как это происходит в случае с `line-height`). В результате можно столкнуться с проблемой, показанной на рис. 6.23:

```
p {letter-spacing: 0.25em; font-size: 20px;}
small {font-size: 50%;}

<p>This spacious paragraph features <small>tiny text that is just as
spacious</small>, even though the author probably wanted the spacing to be in
proportion to the size of the text.</p>
```

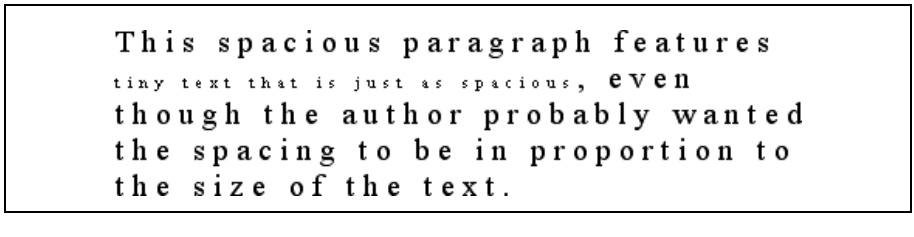


Рис. 6.23. *Унаследованный пробел между буквами*

Единственная возможность добиться, чтобы пробел между буквами был пропорциональным размеру текста, – задавать его явно:

```
p {letter-spacing: 0.25em;}
small {font-size: 50%; letter-spacing: 0.25em;}
```

Преобразование текста

Теперь обратимся к способам изменения регистра букв текста с помощью свойства `text-transform`.

text-transform	
Значения:	`uppercase | lowercase | capitalize | none | inherit`
Начальное значение:	`none`
Область применения:	все элементы
Наследование:	да
Вычисляемое значение:	как задано

Устанавливаемое по умолчанию значение `none` не оказывает на текст никакого воздействия: регистр, пришедший из исходного документа, не изменяется. Как следует из их имен, значения `uppercase` и `lowercase` переводят все символы текста в верхний и нижний регистр соответственно. И наконец, если указано значение `capitalize`, то прописной становится только первая буква каждого слова. Рисунок 6.24 иллюстрирует каждый из этих вариантов:

```
h1 {text-transform: capitalize;}
strong {text-transform: uppercase;}
p.cummings {text-transform: lowercase;}
p.raw {text-transform: none;}

<h1>The heading-one at the beginninG</h1>
```

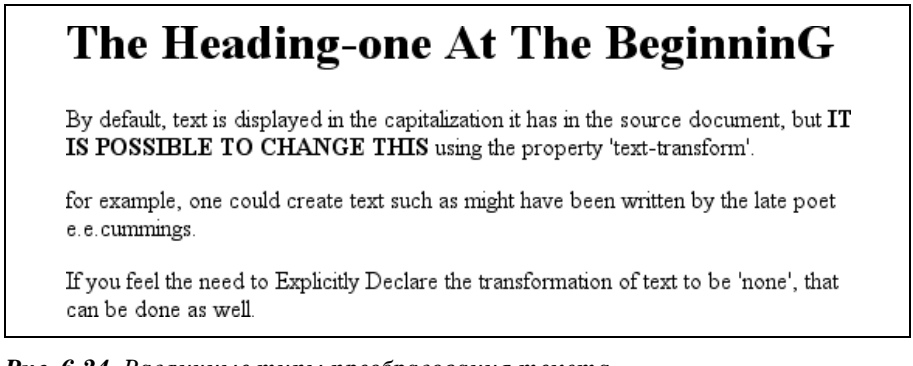


Рис. 6.24. *Различные типы преобразования текста*

```
<p>
By default, text is displayed in the capitalization it has in the source
document, but <strong>it is possible to change this</strong> using the
property 'text-transform'.
</p>
<p class="cummings">
For example, one could Create TEXT such as might have been Written by the
late Poet e.e.cummings.
</p>
<p class="raw">
If you feel the need to Explicitly Declare the transformation of text to be
'none', that can be done as well.
</p>
```

Разные агенты пользователя могут применять разные методики обнаружения начала слова и в результате по-разному определять, какую букву переводить в верхний регистр. Например, визуальное представление текста «heading-one» элемента `h1`, показанного на рис. 6.24, могло бы быть сгенерировано одним из следующих двух способов: «Heading-one» или «Heading-One». CSS не определяет, какой из них правильный, так что возможны оба.

Наверное, вы также заметили, что последняя буква элемента `h1` на рис. 6.24 остается строчной. Это правильно, поскольку если для свойства `text-transform` задано значение `capitalize`, CSS требует от агентов пользователя лишь убедиться, что в верхний регистр переведена первая буква каждого слова. Все остальные символы слова игнорируются.

Само по себе свойство `text-transform` может показаться незначительным, но оно очень полезно, если вдруг потребуется перевести в верхний регистр символы всех элементов `h1`. Вместо того чтобы изменять содержимое каждого элемента `h1`, достаточно применить `text-transform`:

```
h1 {text-transform: uppercase;}

<h1>This is an H1 element</h1>
```

Применяя `text-transform`, вы получаете двойное преимущество. Во-первых, для осуществления этого изменения достаточно написать одно правило, а не менять содержимое элементов `h1`. Во-вторых, если позже потребуется от всех прописных вернуться к первым заглавным, выполнить это изменение будет еще проще (рис. 6.25):

```
h1 {text-transform: capitalize;}

<h1>This is an H1 element</h1>
```

This Is An H1 Element

Рис. 6.25. *Преобразование элемента H1*

Оформление текста

Далее мы переходим к свойству `text-decoration`, которое является замечательным свойством, предлагающим целый ворох интересных возможностей.

text-decoration	
Значения:	`none | [ underline || overline || line-through || blink ] | inherit`
Начальное значение:	`none`
Область применения:	все элементы
Наследование:	нет
Вычисляемое значение:	как задано

Как и следовало ожидать, если выбрано значение `underline`, происходит подчеркивание элемента; точно так же действует тег `U` в HTML. Значение `overline` приводит к противоположному эффекту: линия отрисовывается над текстом. Значение `line-through` проводит линию посередине текста, получается *перечеркнутый текст* (*strikethrough text*); это аналогично элементам `S` и `strike` в HTML. При выборе значения `blink` текст начинает мерцать аналогично эффекту, получаемому с помощью довольно опасного тега `blink`, поддерживаемого только Netscape. Рисунок 6.26 иллюстрирует примеры применения каждого из этих значений:

```
p.emph {text-decoration: underline;}
p.topper {text-decoration: overline;}
```

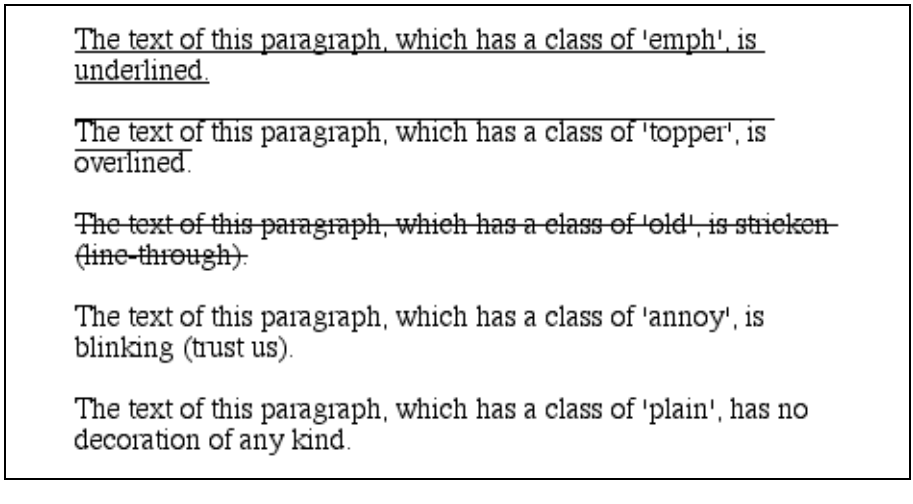


Рис. 6.26. *Различные виды оформления текста*

```
p.old {text-decoration: line-through;}
p.annoy {text-decoration: blink;}
p.plain {text-decoration: none;}
```

Конечно, в книге невозможно показать эффект применения `blink`, но его довольно легко представить (возможно, даже слишком легко). Кстати, от агентов пользователя не требуется поддерживать `blink`, и на момент написания данной книги Internet Explorer этого и не делал.

Значение `none` отключает любое дополнительное оформление, которое в противном случае, возможно, применялось бы к элементу. Обычно по умолчанию текст никак не украшается, но не всегда. Например, ссылки обычно подчеркиваются. Для того чтобы запретить подчеркивание гиперссылок, можно использовать такое CSS-правило:

```
a {text-decoration: none;}
```

Если подчеркивание ссылок явно отключается таким правилом, то единственным визуальным отличием между ссылками и обычным текстом будет их цвет (по крайней мере, таково поведение по умолчанию, хотя твердой гарантии этому нет).

Хотя лично у меня не было проблем с этим, многие пользователи раздражаются, когда понимают, что подчеркивание ссылок отключено. Это спорный вопрос, так что руководствуйтесь собственным вкусом, но помните: если ссылки сливаются с обычным текстом, пользователям может быть трудно находить гиперссылки в документах.

Также можно комбинировать виды оформления в одно правило. Если требуется, чтобы все гиперссылки были и подчеркнуты, и надчеркнуты, правило будет таким:

```
a:link, a:visited {text-decoration: underline overline;}
```

Однако будьте осторожны: если с одним элементом сопоставляются два разных вида оформления, значение более приоритетного правила полностью вытеснит значение проигравшего. Рассмотрим следующий пример:

```
h2.stricken {text-decoration: line-through;}
h2 {text-decoration: underline overline;}
```

В соответствии с этими правилами любой элемент `h2` класса `stricken` будет только зачеркнут. Подчеркивание и надчеркивание утеряны, т. к. значения при краткой записи замещают друг друга, а не суммируются.

Странности оформления

Теперь поговорим о необычных особенностях `text-decoration`. Первая заключается в том, что свойство `text-decoration` *не наследуется*. От-

сутствие наследования означает, что любые линии оформления, отрисовываемые с текстом (под, над или через него), будут иметь тот же цвет, что и родительский элемент. Это так, даже если цвет элементов-потомков другой, как показано на рис. 6.27:

```
p {text-decoration: underline; color: black;}
strong {color: gray;}

<p>This paragraph, which is black and has a black underline, also contains
<strong>strongly emphasized text</strong> which has the black underline
beneath it as well.</p>
```

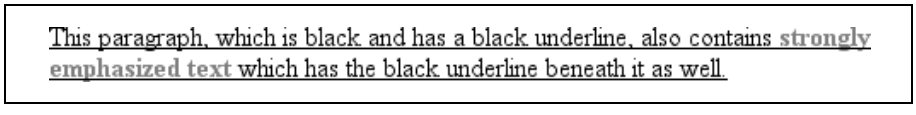


Рис. 6.27. *Постоянство цвета линий подчеркивания*

Почему так? Поскольку значение `text-decoration` не наследуется, элемент `strong` получает установленное по умолчанию значение `none`. Поэтому у элемента `strong` *нет* подчеркивания. Но мы ясно видим линию под элементом `strong`, так что странно на первый взгляд было бы говорить, что у него ее нет. Тем не менее так и есть. То, что вы видите под элементом `strong`, – это подчеркивание абзаца, которое охватывает и элемент `strong`. Более ясно это можно увидеть, если изменить стили выделенного полужирным шрифтом элемента, как показано ниже:

```
p {text-decoration: underline; color: black;}
strong {color: gray; text-decoration: none;}

<p>This paragraph, which is black and has a black underline, also contains
<strong>strongly emphasized text</strong> which has the black underline
beneath it as well.</p>
```

Результат аналогичен показанному на рис. 6.27, поскольку вы всего лишь «в лоб» объявили то, что делалось по умолчанию. Иначе говоря, нельзя отключить подчеркивание (или надчеркивание, или перечеркивание), сгенерированное родительским элементом.

Когда свойство `text-decoration` комбинируется с `vertical-align`, происходят еще более странные вещи. На рис. 6.28 показан один из этих курьезов. Поскольку элемент `sup` не имеет собственного оформления, но смещается вверх в рамках надчеркнутого элемента, верхняя черта перечеркивает элемент `sup`:

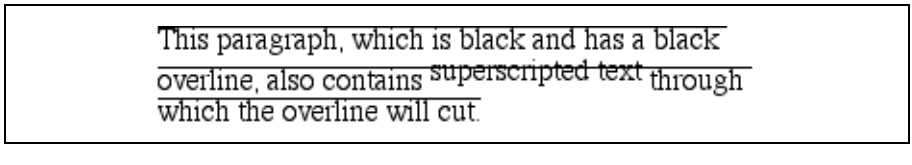


Рис. 6.28. *Верное, хотя и странное поведение при декорировании текста*

```
p {text-decoration: overline; font-size: 12pt;}
sup {vertical-align: 50%; font-size: 12pt;}
```

Сейчас вы, наверное, клянетесь никогда не прибегать к оформлению текста из-за всех этих неприятностей, к которым оно может привести. Кстати, я привел лишь самые простые из возможных последствий, т. к. говорил только о том, как все это *должно* работать согласно спецификации. В действительности некоторые веб-броузеры отключают подчеркивание в дочерних элементах, даже несмотря на то, что они не должны этого делать. Причина, по которой броузеры нарушают спецификацию, достаточно проста: предпочтения авторов броузера. Рассмотрим такую разметку:

```
p {text-decoration: underline; color: black;}
strong {color: silver; text-decoration: none;}
```

```
<p>This paragraph, which is black and has a black underline, also contains
<strong>boldfaced text</strong> which does not have black underline beneath
it.</p>
```

На рис. 6.29 показано представление в веб-броузере, который отключает подчеркивание для элемента `strong`.

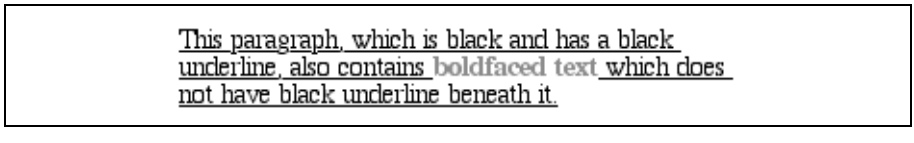


Рис. 6.29. *Как некоторые броузеры ведут себя на самом деле*

Опасность здесь в том, что многие броузеры действительно *следуют* спецификации, а будущие версии существующих броузеров (или любых других агентов пользователя) когда-нибудь все-таки будут точно ее выполнять. Применяя `none` для отмены оформления, важно понимать, что это может аукнуться в будущем или даже стать причиной проблем уже сейчас. Но в будущих версиях CSS, возможно, будут реализованы средства, позволяющие отключать оформление, не прибегая к некорректному применению `none`, так что, может быть, есть надежда.

Существует способ изменить цвет оформления без нарушения спецификации. Как вы помните, применение оформления текста элемента означает, что оформление всего элемента будет одноцветным, даже если дочерние элементы выделены другим цветом. Чтобы цвет оформления соответствовал цвету элемента, надо прямо объявить оформление, как это сделано в следующем примере:

```
p {text-decoration: underline; color: black;}
strong {color: silver; text-decoration: underline;}
```

```
<p>This paragraph, which is black and has a black underline, also contains
<strong>strongly emphasized text</strong> which has the black underline
beneath it as well, but whose gray underline overlays the black underline of
its parent.</p>
```

На рис. 6.30 для элемента strong задан серый цвет и подчеркивание. Серая подчеркивающая черта налагается поверх черной черты родительского элемента, поэтому цвет оформления соответствует цвету элемента strong.

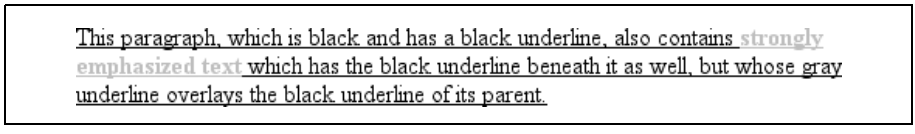


Рис. 6.30. *Преодоление стандартного поведения при подчеркивании*

Затенение текста

CSS2 включает свойство для создания отбрасываемой текстом тени, но в CSS2.1 это свойство не перешло, потому что ни один броузер не реализовал полной его поддержки на момент завершения CSS2.1. Отсутствие реализации затенения в спецификации станет более понятным, когда вы поймете, что необходимо сделать, чтобы заставить веб-броузер определять контуры текста элемента и затем вычислять одну или несколько теней, которые должны накладываться друг на друга, не перекрывая сам текст.

Очевидно, что по умолчанию затенение текста отсутствует. В противном случае теоретически возможно определить одну или несколько теней. Для каждой тени задаются цвет и три длины. Параметр color, конечно же, задает цвет тени, так что можно сделать тени зелеными, фиолетовыми или даже белыми.

Первые два значения длины определяют смещение тени по отношению к тексту, а необязательное третье значение – радиус размытия тени. Чтобы создать зеленую тень, смещенную относительно текста на пять пикселов вправо и на 0,5 em вниз без размытия, следует написать:

```
text-shadow: green 5px 0.5em;
```

Если значения длин отрицательные, тень смещается влево и вверх от текста.

text-shadow

Значения:	none | [<цвет> || <длина> <длина> <длина>? ,]* [<цвет> || <длина> <длина> <длина>?] | inherit
Начальное значение:	none
Область применения:	все элементы
Наследование:	нет

white-space	
Значения:	normal | nowrap | pre | pre-wrap | pre-line | inherit
Начальное значение:	normal
Область применения:	все элементы (CSS2.1); блочные элементы (CSS1 иCSS2)
Наследование:	нет
Вычисляемое значение:	как задано

Радиус размытия определяется как расстояние от контура тени до края эффекта размытия. Радиус в два пиксела в результате дал бы размытие, заполняющее пространство между контуром тени и краем размытия. Способ размытия точно не определен, поэтому разные агенты пользователя могут применять различные эффекты. Например, визуальное представление следующих стилей могло бы быть сгенерировано примерно так, как представлено на рис. 6.31:

```
p.cl1 {color: black; text-shadow: silver 2px 2px 2px;}
p.cl2 {color: white; text-shadow: 0 0 4px black;}
p.cl3 {color: black; text-shadow: 1em 1em 5px gray, -1em -1em silver;}
```

Рисунок 6.31 был сгенерирован с помощью программы Photoshop, поскольку веб-броузеры на момент написания данной книги не поддерживали свойство text-shadow.

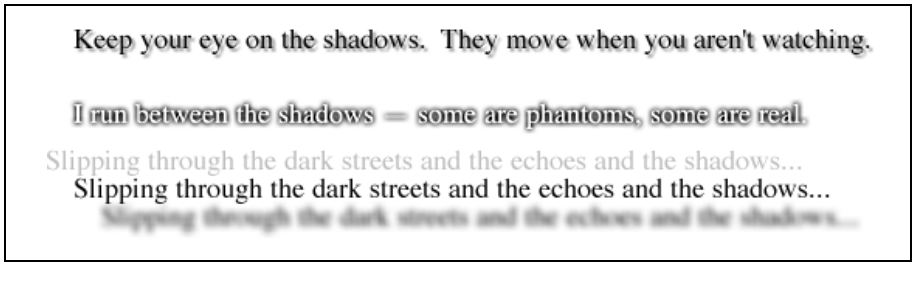


***Рис. 6.31.** Отбрасывание теней во все стороны*

Обработка пробелов

Рассмотрев различные способы стилевого оформления текста, поговорим о свойстве white-space, которое влияет на то, как агент пользователя обрабатывает пробелы, символы перевода строки и табуляции в документе.

Применяя это свойство, можно управлять тем, как броузер интерпретирует разделители между словами и строками текста. В какой-то ме-

ре в XHTML это уже является поведением по умолчанию: любое количество символов-разделителей заменяется одним пробелом. Поэтому, реализуя следующую разметку, веб-броузер вставил бы между словами только по одному пробелу и проигнорировал бы перенос строки.

```
<p>This    paragraph   has     many
  spaces        in it.</p>
```

Это стандартное поведение можно задать прямо:

```
p {white-space: normal;}
```

Это правило указывает броузеру поступать так, как он всегда делал: отбрасывать дополнительные пробелы. Согласно этому значению символы переноса строки (возврата каретки) преобразовываются в пробелы, и любая последовательность, состоящая из более чем одного пробела подряд, заменяется одним пробелом.

Однако если присвоить свойству `white-space` значение `pre`, пробелы в элементе будут интерпретироваться так, как будто бы он является XHTML-элементом `pre`; т. е. пробелы *не игнорируются*, как показано на рис. 6.32:

```
p {white-space: pre;}

<p>This    paragraph   has     many
  spaces        in it.</p>
```

Если значение свойства `white-space` равно `pre`, броузер будет отображать дополнительные пробелы и переносы строки. В этом отношении любой элемент можно заставить работать как элемент `pre`.

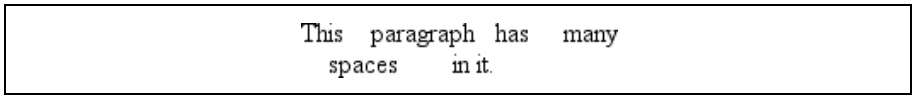


***Рис. 6.32.** Обработка пробелов разметки*

Противоположное значение – `nowrap`, которое запрещает перенос текста в элементе, за исключением случаев применения элементов `<br/>`. Применение `nowrap` в CSS во многом похоже на объявление `<td nowrap>` для ячеек таблицы в HTML 4, за исключением того, что значение `white-space` может применяться к любому элементу. Результат применения следующей разметки показан на рис. 6.33:

```
<p style="white-space: nowrap;">This paragraph is not allowed to wrap, which
means that the only way to end a line is to insert a line-break element. If
no such element is inserted, then the line will go forever, forcing the user
to scroll horizontally to read whatever can't be initially displayed <br/>in
the browser window.</p>
```

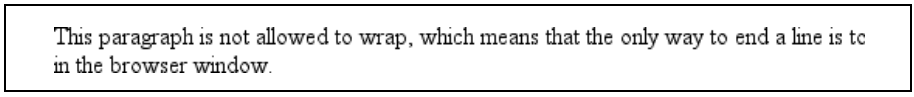


Рис. 6.33. *Как с помощью свойства white-space запретить автоматический перенос по словам*

Теперь можно применять `white-space` вместо атрибута `nowrap` ячеек таблицы:

```
td {white-space: nowrap;}

<table><tr>
<td>The contents of this cell are not wrapped.</td>
<td>Neither are the contents of this cell.</td>
<td>Nor this one, or any after it, or any other cell in this table.</td>
<td>CSS prevents any wrapping from happening.</td>
</tr></table>
```

CSS2.1 предоставляет значения `pre-wrap` и `pre-line`, которых не было в более ранних версиях CSS. Задача этих значений – предоставить более широкие возможности обработки разделителей.

Если для элемента задано `pre-wrap`, пробелы в тексте элемента сохраняются, но строки текста прерываются где необходимо. Если задано это значение, то учитываются не только изначально заданные разрывы строк, но и сгенерированные. Значение `pre-line` является противоположностью `pre-wrap` и приводит к удалению лишних пробелов, как в обычном тексте, но учитывает разделители строк. Рассмотрим следующую разметку, которая проиллюстрирована на рис. 6.34:

```
<p style="white-space: pre-wrap;">
This  paragraph     has  a  great   many   s p a c e s   within  its textual
  content,   but their   preservation    will   not    prevent   line
    wrapping or line breaking.
</p>
<p style="white-space: pre-line;">
This  paragraph     has  a  great   many   s p a c e s   within  its textual
  content,   but their collapse  will  not    prevent  line
    wrapping or line breaking.
</p>
```

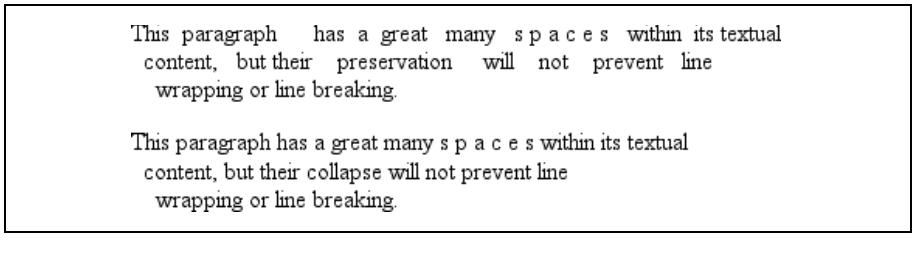


Рис. 6.34. *Два разных способа обработки пробелов*

В табл. 6.1 описано поведение свойства white-space.

Таблица 6.1. Свойство white-space

Значение	Пробелы	Перевод строки	Автоматический перенос строк
pre-line	Сворачиваются	Обрабатывается	Разрешен
normal	Сворачиваются	Игнорируется	Разрешен
nowrap	Сворачиваются	Игнорируется	Запрещен
pre	Сохраняются	Обрабатывается	Запрещен
pre-wrap	Сохраняются	Обрабатывается	Разрешен

Направление написания текста

Текст этой книги следует читать слева направо и сверху вниз, таково направление письма в русском языке. Однако не все языки читаются так. Существует множество языков с написанием справа налево, таких как иврит или арабский, и CSS2 представляет свойство для описания их направленности.

Свойство direction влияет на направление написания текста блочного элемента, направление размещения столбца таблицы, направление, в котором содержимое заполняет блок своего элемента в горизонтальной плоскости, и на положение последней строки выровненного по ширине элемента. Для строковых элементов направление применяется, только если свойству unicode-bidi задано или значение embed, или bidi-override. (Описание unicode-bidi приведено ниже.)

До CSS3 спецификация CSS не включала обеспечения для языков, читаемых сверху вниз. На момент написания данной книги CSS3 Text Module является предварительной рекомендацией и посвящает этому вопросу новое свойство, названное writing-mode.

Значение ltr устанавливается по умолчанию, но предполагается, что если броузер отображает текст, написанный справа налево, значение изменится на rtl. Таким образом, броузер мог бы иметь внутреннее правило, утверждающее примерно следующее:

```
*:lang(ar), *:lang(he) {direction: rtl;}
```

Реальное правило могло бы быть более длинным и включать все языки с написанием справа налево, а не только арабский и иврит, но пример приведен лишь для иллюстрации. Пока CSS лишь делает попытку учитывать направление написания, в Unicode уже есть намного более надежный метод обработки направленности. Имея в распоряжении свойство unicode-bidi, CSS-авторы могут использовать преимущество некоторых возможностей Unicode.

direction

Значения:	`ltr | rtl | inherit`
Начальное значение:	`ltr`
Область применения:	все элементы
Наследование:	да
Вычисляемое значение:	как задано

Здесь мы лишь приведем описания значений из спецификации CSS2.1, которые хорошо отражают суть каждого из них:

`normal`

Элемент не открывает дополнительного уровня вложенности согласно двунаправленному алгоритму. Для строковых элементов неявное переупорядочивание выполняется в пределах их границ.

`embed`

Если элемент строковый, это значение открывает дополнительный уровень вложенности согласно двунаправленному алгоритму. Направление этого уровня вложенности задается свойством `direction`. Внутри элемента переупорядочивание осуществляется неявно. Это аналогично добавлению LRE (U+202A; для `direction: ltr`) или RLE (U+202B; для `direction: rtl`) в начало элемента и PDF (U+202C) в конец.

`bidi-override`

Этим создается замещение для строкового элемента. Для элементов уровня блока создается замещение для строковых потомков, находящихся только в этом блоке. Это означает, что внутри элемента переупорядочивание проводится в последовательности, строго соответствующей свойству `direction`; неявная часть двунаправленного алгоритма игнорируется. Это аналогично добавлению LRO (U+202D; для `direction: ltr`) или RLO (U+202E; для `direction: rtl`) в начало элемента и PDF (U+202C) в конец элемента.

unicode-bidi

Значения:	`normal | embed | bidi-override | inherit`
Начальное значение:	`normal`
Область применения:	все элементы
Наследование:	нет
Вычисляемое значение:	как задано

Заключение

Есть множество способов изменить представление текста, даже не изменяя шрифт. Конечно, существуют классические эффекты, такие как подчеркивание, но CSS также дает возможность отрисовывать линии над текстом или поверх него, менять расстояния между словами и буквами, задавать отступ первой строки абзаца (или другого блочного элемента), выравнивать текст по правой или по левой границе и многое другое. Можно даже изменить расстояние между строками текста, хотя с этой операцией связаны неожиданные сложности, более подробно рассматриваемые в главе 7.

Все эти возможности или относительно хорошо поддерживаются, или не поддерживаются вообще. Выравнивание текста по ширине – одно из свойств, которое реализовано недостаточно хорошо, и большинство агентов пользователя, разработанных в XX веке, демонстрируют ошибки при оформлении текста и вертикальном выравнивании, а также в вычислении высоты строки. С другой стороны, расстояния между словами и буквами практически всегда обрабатываются правильно (если они обрабатываются), и в реализации структурирования текста замечено лишь несколько маленьких ошибок. То же самое можно сказать и об изменении регистра символов, которое обычно корректно реализовано.

Несколько раз в этой главе я упоминал, что компоновка строк – более сложный процесс, чем может показаться на первый взгляд. Ее детали и ряд других тем я рассматриваю в следующей главе.

7

Основы модели визуального форматирования

В предыдущих главах я привел огромное количество практической информации о том, как CSS управляет текстом и шрифтами документа. В этой главе мы обратимся к теоретическим аспектам формирования визуального представления, которые отвечают на многие вопросы, пропущенные нами ранее ради того, чтобы сосредоточить внимание на реализации CSS.

Почему необходимо посвятить целую главу теоретическим основам формирования визуального представления в CSS? Дело в том, что CSS – настолько мощная, открытая модель, что ни одна книга не могла бы претендовать на охват всех возможных способов сочетания свойств и эффектов. Несомненно, вы продолжите открывать новые возможности применения CSS для создания собственных эффектов в документах.

В ходе изучения CSS вы можете столкнуться со странным на вид поведением агентов пользователя. Полностью понимая принцип работы модели формирования визуального представления CSS, можно определить, является ли это поведение правильным (хотя и неожиданным) результатом действия механизма формирования визуального представления CSS, или вы столкнулись с ошибкой, о которой надо сообщить.

Основные блоки

CSS предполагает, что каждый элемент генерирует один или несколько прямоугольных блоков, называемых *контейнерами элемента* (*element boxes*). (В будущих версиях спецификации, возможно, появятся непрямоугольные блоки, но на данный момент все блоки прямоугольные.) Основной частью каждого блока элемента является *область содержимого* (*content area*). Область содержимого окружена произвольным ко-

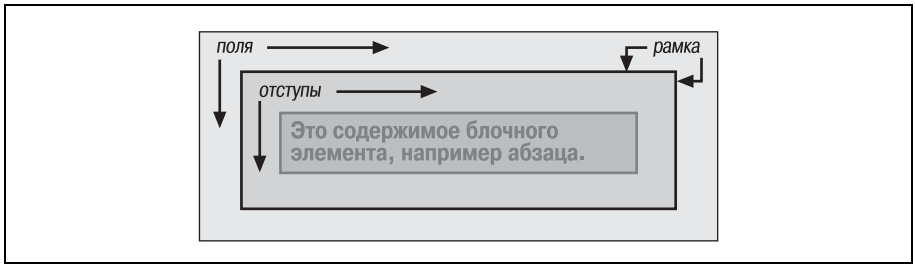


***Рис. 7.1.** Область содержимого и ее окружение*

личеством отступов, рамок и полей. Эти элементы считаются необязательными, потому что всем им может быть присвоена нулевая ширина, что удалит их из блока. Пример области содержимого с указанием окружающих его зон отступов, рамок и полей приведен на рис. 7.1.

Поля, рамки и отступы могут быть заданы посредством различных свойств, таких как `margin-left` или `border-bottom`. Фон содержимого, например цвет или повторяющееся изображение, также применяется к отступам. Поля всегда остаются прозрачными, сквозь них виден фон любых родительских элементов. Отступ не может иметь отрицательное значение, а вот поля могут. Результат применения отрицательных полей мы рассмотрим в этой главе позже.

Рамки генерируются с применением определенных стилей, таких как `solid` или `inset`, а их цвета задаются свойством `border-color`. Если цвет не задан, рамка окрашивается в основной цвет содержимого элемента. Например, если текст абзаца белый, то все рамки вокруг этого абзаца будут белыми, если только автор не задал явно другой цвет рамки. Если в рамке есть разрывы, сквозь них виден фон элемента. Иначе говоря, рамка имеет такой же фон, что и содержимое, и отступ. И наконец, ширина рамки никогда не может быть отрицательной.

На различные компоненты контейнера элемента влияет множество свойств, таких как `width` или `border-right`. Многие из этих свойств будут фигурировать в этой главе, даже несмотря на то, что мы еще их не обсуждали. Фактические объявления свойств приведены в главе 8, в которой более развернуто рассматриваются понятия, изложенные в этой главе.

Однако можно найти отличия в форматировании различных типов элементов. Блочные и строковые элементы обрабатываются по-разному, и, кроме того, свободно перемещаемые элементы и элементы с абсолютным позиционированием имеют собственные модели поведения.

Блок-контейнер

Каждый элемент располагается относительно его блока-контейнера. С высокой степенью достоверности можно утверждать, что блок-кон-

тейнер – это «контекст расположения» элемента. CSS2.1 устанавливает ряд правил для объявления блока-контейнера элемента. Я рассмотрю только те из них, которые имеют отношение к концепции, излагаемой в данной главе, а все остальные оставлю для будущих глав.

Для элемента при нормальном (западном) течении текста блок-контейнер формируется *краем содержимого* (*content edge*) ближайшего блочного элемента, ячейки таблицы или строкового блока, выступающего в качестве предка данного элемента. Рассмотрим следующую разметку:

```
<body>
 <div>
  <p>Это параграф.</p>
 </div>
</body>
```

В этом очень простом примере блок-контейнер элемента `p` – элемент `div`, поскольку из всех блочных элементов, ячеек таблиц или строковых блоков это ближайший элемент-предок (в данном случае он представляет собою блочный контейнер). Аналогичным образом в качестве контейнера элемента `div` выступает `body`. Таким образом, компоновка `p` зависит от компоновки `div`, которая, в свою очередь, находится в зависимости от компоновки `body`.

Сейчас мы не будем говорить о строковых элементах, поскольку их расположение не зависит непосредственно от блоков-контейнеров. Они рассмотрены в этой главе позже.

Краткий обзор терминов

Охарактеризуем коротко типы элементов, обсуждаемых в данной главе, а также вспомним некоторые важные термины, необходимые нам для понимания материала:

Нормальный поток

Это воспроизведение текста, написанного на западных языках, слева направо и сверху вниз и обычная схема компоновки текста традиционных HTML-документов. Обратите внимание, что направление потока может меняться для восточных языков. Большинство элементов размещается в нормальном потоке: элемент может покинуть его, лишь став свободно перемещаемым или абсолютно позиционированным (рассматриваются в главе 10). Помните, в этой главе обсуждаются только элементы, размещаемые в нормальном потоке.

Незамещаемый элемент

Элемент, содержимое которого включено в документ. Например, абзац представляет собою незамещаемый элемент, потому что его текстовое содержимое находится в самом документе.

Замещаемый элемент

Элемент, который служит полем подстановки для чего-то другого. Классический пример замещаемого элемента – элемент `img`, указывающий на файл изображения, подставляемого затем в поток документа в том месте, где находится сам элемент `img`. Большинство элементов формы также относятся к замещаемым (например, `<input type="radio">`).

Блочный элемент

Элементы, такие как абзацы, заголовки или `div`. Эти элементы в нормальном потоке генерируют «разделители строк» как перед, так и после своих блоков, поэтому блочные элементы в нормальном потоке размещаются по вертикали. Элемент можно принудительно сделать блочным, объявив `display: block`.

Строковый элемент

Элементы, такие как `strong` или `span`. Они не генерируют «разрывов строки» перед или после себя и являются потомками блочных элементов. Такой элемент можно использовать для формирования блока строкового уровня, объявив `display: inline`.

Корневой элемент

Элемент, находящийся в самой вершине дерева документа. В HTML-документах это элемент `html`. В XML-документах это может быть любой допускаемый языком элемент.

Блочные элементы

Поведение блочных элементов иногда бывает непредсказуемым. Например, может отличаться размещение элементов относительно горизонтальной и вертикальной осей. Чтобы полностью понимать, как обрабатываются блочные элементы, необходимо иметь четкое представление о некотором наборе границ и областей. Они представлены на рис. 7.2.

В общем случае ширина элемента определяется как расстояние от левой внутренней границы до правой внутренней границы, а высота – как расстояние от верхней внутренней границы до нижней внутренней границы. К элементу могут применяться оба этих свойства.

Для определения планировки документа производится комбинирование различных ширин, высот, отступов и полей. В большинстве случаев броузер автоматически определяет высоту и ширину документа на основании доступных сведений об области отображения и других факторов. CSS, конечно, предоставляет больше возможностей прямого управления изменением размеров и отображением элементов. Можно выбирать различные эффекты для осуществления горизонтальной и вертикальной компоновки, поэтому мы рассмотрим их отдельно.

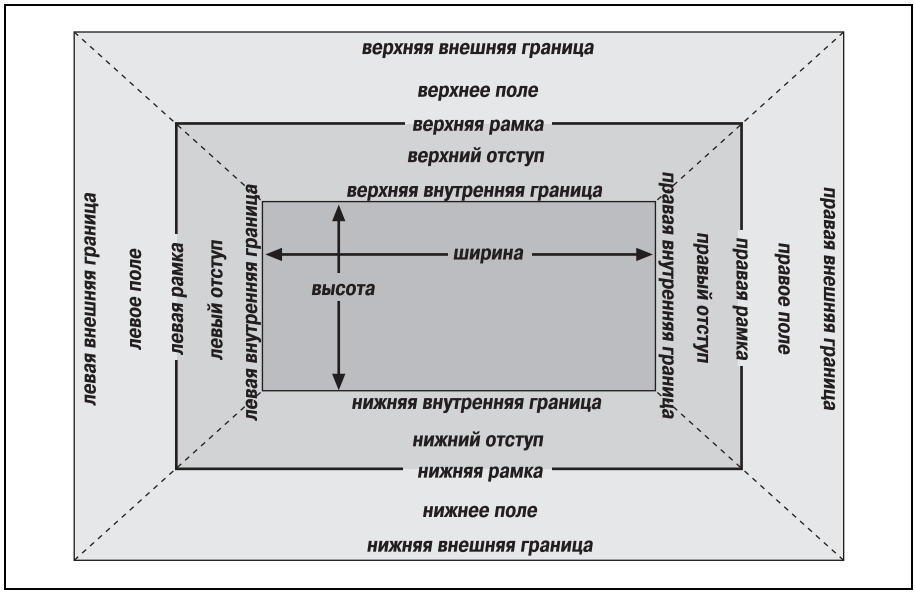


***Рис. 7.2.** Модель блока*

Горизонтальное форматирование

Горизонтальное форматирование обычно сложнее, чем может показаться на первый взгляд. Частично сложность заключается в том, чтобы понять, что свойство `width` определяет ширину области содержимого, а *не* всего видимого блока элемента. Рассмотрим следующий пример:

```
<p style="width: 200px;">ширина?</p>
```

Эта строка кода сделает ширину содержимого абзаца равной 200 пикселам. Задав для элемента фон, это можно было бы увидеть явно. Однако любые определяемые отступы, рамки или поля *добавляются* к значению ширины. Предположим, есть такая разметка:

```
<p style="width: 200px; padding: 10px; margin: 20px;">ширина?</p>
```

Видимый блок элемента теперь составляет 220 пикселов в ширину, поскольку справа и слева от содержимого добавлены отступы по 10 пикселов. Затем поля добавят еще по 20 пикселов с каждой стороны, и общая ширина блока элемента составит 260 пикселов.

Понимать скрытые дополнения к значению свойства `width` крайне важно. Большинство пользователей думают, что `width` определяет ширину видимого блока элемента и что если они объявляют в элементе отступы, рамку и ширину, задаваемое ими значение ширины будет расстоянием от внешнего левого края рамки до внешнего правого края рамки. *В CSS это не так.* Твердо запомните это, чтобы избежать неприятностей в будущем.

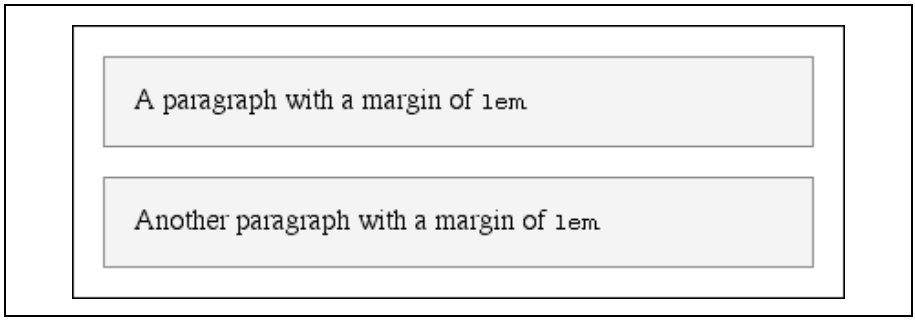


***Рис. 7.3.** Ширина блоков элементов равна ширине их родительского элемента*

На момент написания данной книги модуль Box Model спецификации CSS3 содержит предложения, рекомендующие предоставить авторам выбирать, будет ли `width` задавать ширину содержимого или ширину видимого блока.

Следующее простое правило гласит, что сумма размеров горизонтальных компонентов контейнера блочного элемента в нормальном потоке всегда равна значению `width` его родителя. Возьмем два абзаца, для которых заданы поля в `1em`, находящихся в элементе `div`. Ширина содержимого (значение `width`) абзаца плюс его левый и правый отступы, рамки и поля всегда наращиваются до значения свойства `width` области содержимого элемента `div`.

Пусть, скажем, `width` элемента `div` равна `30em`. Это значит, что общая сумма ширины содержимого, отступов, рамок и полей каждого параграфа составляет `30em`. На рис. 7.3 «пустое» пространство вокруг абзацев – это на самом деле их поля. Если бы у элемента `div` были отступы, пустого пространства было бы еще больше, но здесь их нет. Скоро мы обсудим и отступы.

Свойства горизонтального форматирования

Перечислим семь свойств горизонтального форматирования: `margin-left`, `border-left`, `padding-left`, `width`, `padding-right`, `border-right` и `margin-right`. Они относятся к горизонтальной компоновке контейнеров блочных элементов и представлены схематически на рис. 7.4.

Значения этих семи свойств должны соответствовать ширине элемента, содержащего данный блок, которая обычно равна значению свойства `width` родителя блочного элемента (поскольку в качестве родителя блочных элементов практически всегда выступают также блочные элементы).

Лишь для трех из этих семи свойств может быть задано значение `auto`: для ширины содержимого элемента, для левого и правого полей. Всем остальным свойствам должно быть присвоено или конкретное, или

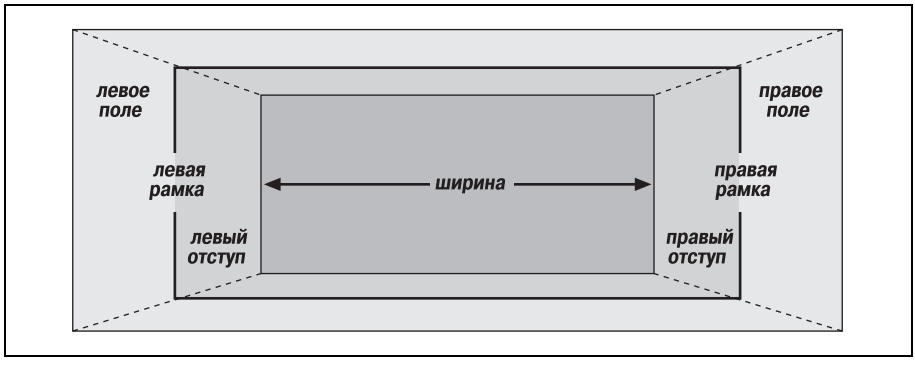


Рис. 7.4. *Семь свойств горизонтального форматирования*

применяемое по умолчанию (нуль) значение. На рис. 7.5 показано, какие части блока могут принимать значение `auto`, а какие – нет.

Свойству `width` должно быть присвоено либо значение `auto`, либо неотрицательное значение некоторого типа. Применение `auto` при горизонтальном форматировании может порождать разнообразные эффекты.

CSS разрешает броузерам задавать минимальное значение для `width`; это величина, ниже которой ширина блочного элемента опуститься не может. Значение этого минимума в разных броузерах может меняться, поскольку в спецификации оно не определено.

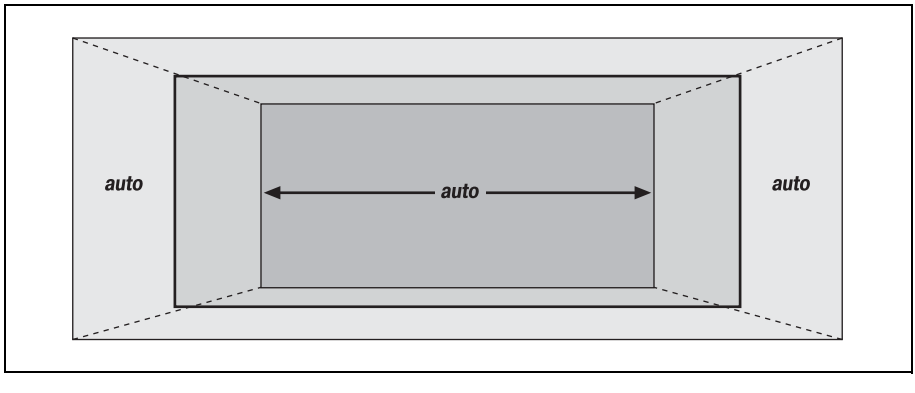


Рис. 7.5. *Свойства горизонтального форматирования, которым может быть присвоено значение auto*

Применение значения auto

Если одному из свойств `width`, `margin-left` или `margin-right` задано значение `auto`, а для оставшихся двух – определенные значения, то свойство, заданное как `auto`, определяет длину, необходимую, чтобы сделать

ширину блока равной ширине родительского элемента. Иначе говоря, пусть сумма значений семи свойств должна равняться 400 пикселам, не заданы ни отступы, ни рамка, поле справа и ширина составляют 100px, а поле слева определено как auto. Ширина поля слева составит 200 пикселов:

```
p {margin-left: auto; margin-right: 100px;
  width: 100px;}   /* поле слева - 'auto' - приравнивается 200px */
```

В некотором смысле auto может компенсировать разницу между суммой всех остальных и требуемым общим значением. Но что если для всех этих трех свойств задано значение 100px и ни одному из них – auto?

Если всем трем свойствам задано значение, отличное от auto, – или в терминах CSS: когда эти свойства форматирования сверхограничены (overconstrained), – свойству margin-right *всегда* принудительно присваивается значение auto. Это означает, что если для обоих полей ширины задано значение 100px, то агент пользователя переопределит правое поле и присвоит ему значение auto. Тогда ширина поля справа будет определена в соответствии с правилом о том, что значение auto дополняет общую ширину элемента до ширины его блока-контейнера. На рис. 7.6 показан результат применения следующей разметки:

```
p {margin-left: 100px; margin-right: 100px;
  width: 100px;}   /* правому полю принудительно присваивается
значение 200px */
```

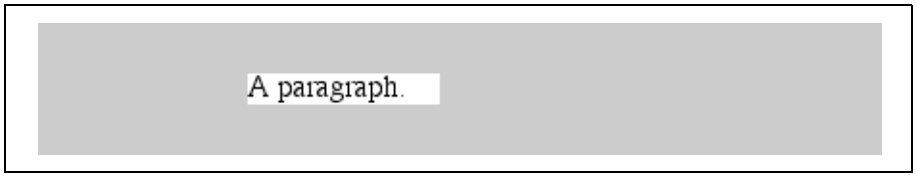


Рис. 7.6. *Переопределение установочных параметров margin-right*

Принудительное присваивание значения auto свойству margin-right осуществляется только для языков с написанием слева направо, таких как русский. В языках с написанием справа налево происходит прямо противоположное: значение auto присваивается свойству margin-left, а не margin-right.

Если оба поля задаются явно, а значение auto определено для width, то свойству width будет присвоено значение, необходимое для достижения необходимой общей ширины (каковой является ширина содержимого родительского элемента). Результаты применения следующей разметки показаны на рис. 7.7:

```
p {margin-left: 100px; margin-right: 100px; width: auto;}
```

Вариант, показанный на рис. 7.7, – самый распространенный, он эквивалентен заданию полей без объявления конкретной ширины. Ре-

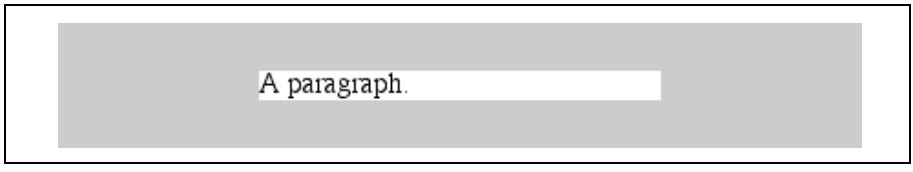


Рис. 7.7. *Автоматическое определение ширины*

зультат применения следующей разметки полностью аналогичен показанному на рис. 7.7:

```
p {margin-left: 100px; margin-right: 100px;} /* аналогичен предыдущему */
```

Значение auto для нескольких свойств

Теперь давайте посмотрим, что происходит, когда значение `auto` задано для двух из трех свойств (`width`, `margin-left` или `margin-right`). Если оба поля имеют значение `auto`, как показано в приведенном ниже коде, для них устанавливается одинаковая длина. Таким образом, происходит центрирование элемента в рамках его родительского элемента, что можно увидеть на рис. 7.8:

```
p {width: 100px; margin-left: auto; margin-right: auto;}
```

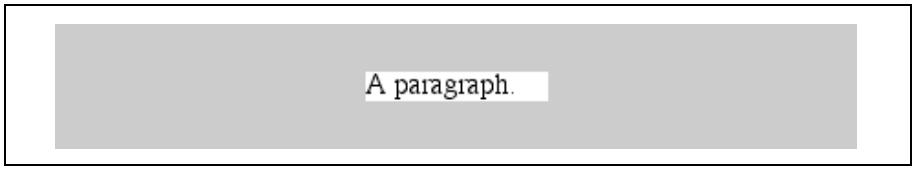


Рис. 7.8. *Явное задание ширины*

Назначение для обоих полей одинаковой длины – надежный способ центрирования элемента, чего нельзя сказать о применении свойства `text-align`. (Свойство `text-align` применяется только к строковому содержимому блочного элемента, поэтому если определить для элемента свойство `text-align` со значением `center`, он не будет центрирован.)

На практике только броузеры, выпущенные после февраля 1999 года, правильно обрабатывают центрирование поля с шириной `auto`, и не все они делают это абсолютно верно. Поведение тех броузеров, которые неправильно обрабатывают поля `auto`, непредсказуемо, и самый безопасный вариант – если они будут задавать для обоих полей нулевое значение.

Теперь посмотрим, как будут вычислены размеры элементов, если присвоить значение `auto` одному из полей и ширине. Поле со значением `auto` обращается в нуль:

```
p {margin-left: auto; margin-right: 100px;
 width: auto;}  /* поле слева обращается в 0 */
```

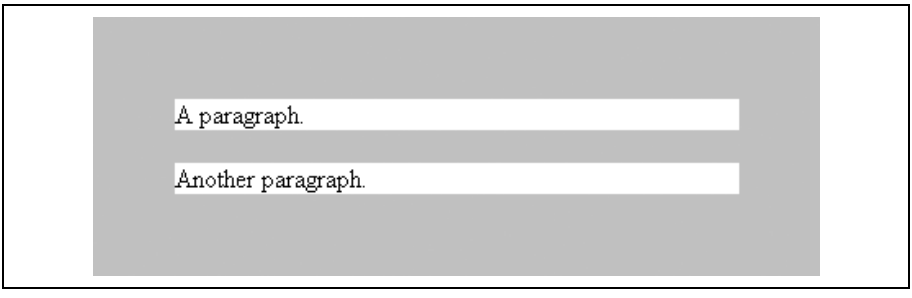


***Рис. 7.9.** Неявное смещение в полях и отступах родителя*

Затем `width` присваивается значение, необходимое, чтобы элемент заполнил свой контейнер.

Наконец, что происходит, если значение `auto` присвоено всем трем свойствам? Ответ прост: оба поля обращаются в нуль, а ширина становится максимальной. Это аналогично стандартной ситуации, когда значения полей или ширины явно не заданы. В этом случае по умолчанию поля становятся нулевыми, а ширине присваивается значение `auto`.

Заметьте, что поскольку горизонтальные поля не сворачиваются, отступы, рамки и поля родительского элемента могут оказывать влияние на дочерние элементы. Это влияние косвенное и проявляется в том, что поля (и т. д.) элемента могут вызывать смещение дочерних элементов. Результаты применения следующей разметки показаны на рис. 7.9:

```
div {padding: 30px; background: silver;}
p {margin: 20px; padding: 0; background: white;}
```

Отрицательные поля

До сих пор все кажется довольно простым, и вы, наверное, уже удивляетесь, почему я говорил, что здесь могут быть какие-то сложности. У полей есть и другая сторона – отрицательная. Все верно, существует возможность задания отрицательных значений полей. В результате возникают некоторые интересные эффекты, если предположить, что агент пользователя вообще поддерживает отрицательные поля.

Согласно спецификации CSS от агентов пользователя не требуется полностью поддерживать отрицательные поля. Там сказано: «Отрицательные значения для свойств полей допускаются, но возможны зависящие от реализации ограничения». На момент написания данной книги в броузерах имеется несколько подобных ограничений.

Помните, что сумма значений семи свойств горизонтального форматирования всегда равна значению свойства `width` родительского элемента. Пока все свойства имеют нулевое или положительное значение, элемент никогда не может быть шире области содержимого его родителя. Однако рассмотрим следующую разметку, изображенную на рис. 7.10:

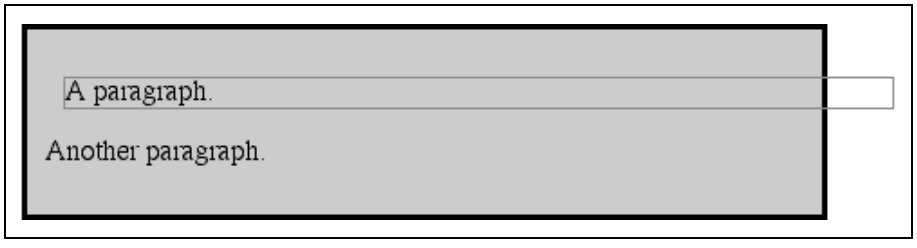


***Рис. 7.10.** Увеличение ширины дочернего элемента за счет отрицательных полей*

```
div {width: 400px; border: 3px solid black;}
p.wide {margin-left: 10px; width: auto; margin-right: -50px; }
```

Совершенно верно, дочерний элемент шире своего родителя! Это математически верно:

$$10px + 0 + 0 + 440px + 0 + 0 - 50px = 400px$$

Величина `440px` – это вычисленное значение выражения `width: auto`, необходимое для сохранения равенства. Несмотря на то что в результате дочерний элемент вышел за рамки своего родителя, спецификация не нарушена, потому что значения семи свойств образуют требуемую сумму. Это семантическая хитрость, но все работает правильно.

Теперь добавим сюда рамку:

```
div {width: 400px; border: 3px solid black;}
p.wide {margin-left: 10px; width: auto; margin-right: -50px;
  border: 3px solid gray;}
```

В результате вычисленное значение свойства `width` уменьшится:

$$10px + 3px + 0 + 434px + 0 + 3px - 50px = 400px$$

Если добавить и отступы, то значение `width` станет еще меньше.

И наоборот, можно сделать так, чтобы вычисляемый при указании ключевого слова `auto` размер правого поля был отрицательным. Это происходит в том случае, если значения других свойств заставляют задавать отрицательное значение правого поля, чтобы элемент не был шире своего блока-контейнера. Рассмотрим следующий пример:

```
div {width: 400px; border: 3px solid black;}
p.wide {margin-left: 10px; width: 500px; margin-right: auto;
  border: 3px solid gray;}
```

Уравнение будет таким:

$$10px + 3px + 0 + 500px + 0 + 3px - 116px = 400px$$

Вычисленное значение ширины правого поля составит `-116px`. Даже если явно задать другое значение, ничего не изменится из-за правила, утверждающего, что если размеры элемента сверхограничены (over-

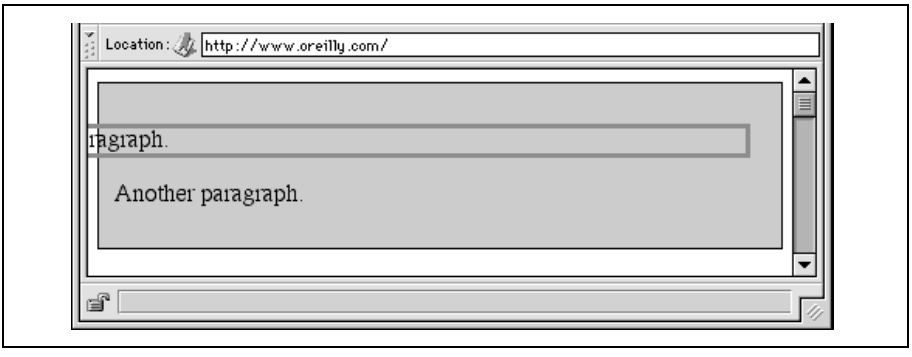


Рис. 7.11. *Задание отрицательного поля слева*

constarined), правому полю присваивается значение, необходимое для обеспечения соответствия (для языков с написанием справа налево это правило применяется к полю слева).

Давайте рассмотрим другой пример, проиллюстрированный на рис. 7.11, в котором отрицательное значение присвоено левому полю:

```
div {width: 400px; border: 3px solid black;}
p.wide {margin-left: -50px; width: auto; margin-right: 10px;
  border: 3px solid gray;}
```

При отрицательном значении поля слева абзац выходит за границы не только элемента `div`, но и за край самого окна броузера!

Помните, что ширина (и высота) отступов, рамок и содержимого никогда не может быть отрицательной. Отрицательные значения допускаются только для полей.

Процентные значения

Для процентных значений ширины, отступов и полей действует то же основное правило. При этом безразлично, как объявлены эти величины: как числовые значения или как процентные.

Процентные значения могут быть очень полезными. Предположим, требуется, чтобы содержимое элемента занимало две трети ширины блока-контейнера, отступы слева и справа составляли по 5% каждый, поле слева – 5%, а поле справа дополняло оставшееся пространство. Это можно было бы написать примерно так:

```
<p style="width: 67%; padding-right: 5%; padding-left: 5%; margin-right: auto;
margin-left: 5%;">экспериментируем с процентами</p>
```

Ширина правого поля составила бы 18% (100% – 67% – 5% – 5% – 5%) ширины блока-контейнера.

Однако сочетать процентные и числовые значения длины может быть достаточно сложно. Рассмотрим следующий пример:

```
<p style="width: 67%; padding-right: 2em; padding-left: 2em; margin-right:
auto;
margin-left: 5em;">смешанные единицы задания длины</p>
```

В этом случае блок элемента может быть определен вот так:

```
5em + 0 + 2em + 67% + 2em + 0 + auto = ширина блока-контейнера
```

Чтобы значение правого поля получилось равным нулю, ширина блока-контейнера должна составлять `27.272727em` (включая область содержимого элемента шириной `18.272727em`). Немного шире – и правое поле будет положительным. Немного уже – и правое поле будет отрицательным.

Все еще больше усложняется, если применять разные единицы измерения длин, как в следующем примере:

```
<p style="width: 67%; padding-right: 15px; padding-left: 10px;
     margin-right: auto;
margin-left: 5em;">еще более запутанное задание длины</p>
```

И чтобы стало понятно, что все еще сложнее: рамки не могут задаваться процентными значениями, а только значениями, выраженными в единицах измерения длины. Итог таков: создать абсолютно гибкую компоновку элемента исключительно с помощью процентных значений нельзя, разве что отказаться от применения рамок.

Замещаемые элементы

До сих пор мы имели дело с горизонтальным форматированием незамещаемых блочных элементов в нормальном потоке текста. Упорядочивать замещаемые блочные элементы немного проще. Сохраняются все правила для незамещаемых блоков, за одним исключением: если значение `width` задано как `auto`, ширина элемента приравнивается к действительной ширине содержимого. Ширина изображения в следующем примере составит 20 пикселов, потому что именно такова ширина изображения, хранящегося в файле:

```
<img src="smile.png" style="display: block; width: auto; margin: 0;">
```

Если бы реальная ширина изображения была равна 100 пикселам, оно и заняло бы 100 пикселов в ширину.

Это правило можно переопределить, указывая конкретное значение свойства `width`. Предположим, предыдущий пример изменили таким образом, чтобы изображение было представлено три раза с разными значениями ширины:

```
<img src="smile.png" style="display: block; width: 25px; margin: 0;">
<img src="smile.png" style="display: block; width: 50px; margin: 0;">
<img src="smile.png" style="display: block; width: 100px; margin: 0;">
```

Это проиллюстрировано на рис. 7.12.

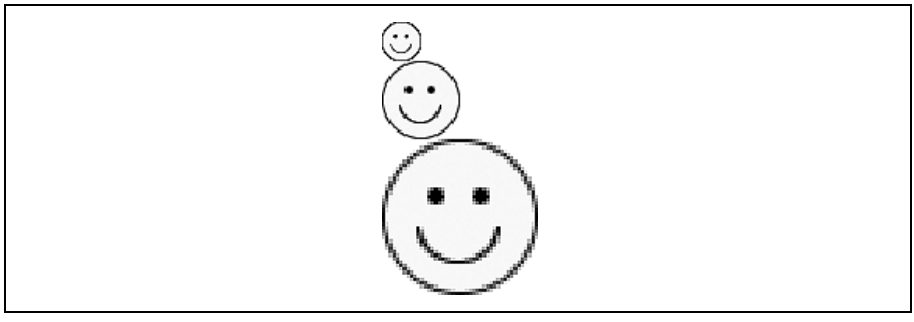


***Рис. 7.12.** Изменение ширины замещаемого элемента*

Обратите внимание, что высота элемента также меняется. Когда исходная ширина замещаемого элемента меняется, значение `height` тоже соответственно масштабируется, если только ей не было явно присвоено собственное значение. Верно и обратное: если задана высота, а свойство `width` оставлено как `auto`, тогда ширина масштабируется пропорционально изменению высоты.

Итак, вы задумались о высоте, и можно переходить к вертикальному форматированию блочных элементов в нормальном потоке.

Вертикальное форматирование

У вертикального форматирования блочных элементов, как и у горизонтального, есть своя специфика. Стандартная высота элемента определяется его содержимым. На высоту также влияет ширина содержимого: чем уже становится абзац, тем выше он должен быть, чтобы содержимое в него поместилось.

В CSS можно явно задать высоту любого блочного элемента. Если это сделать, то результирующее поведение будет зависеть от ряда других факторов. Предположим, что заданная высота больше, чем необходимо для представления содержимого:

```
<p style="height: 10em;">
```

В этом случае излишек высоты выглядит как дополнительный отступ. Но предположим, значение `height` *меньше*, чем необходимо для представления содержимого:

```
<p style="height: 3em;">
```

В таком случае предполагается, что броузер предоставит средства просмотра всего содержимого без увеличения высоты блока элемента. Броузер может добавить в элемент полосу прокрутки, как показано на рис. 7.13.

В случае, когда содержимое элемента выше, чем высота его блока, реальное поведение броузера будет зависеть от значения (и поддержки броузером) свойства `overflow`. Этот сценарий рассматривается в главе 10.

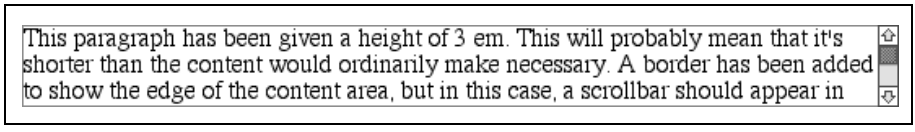


Рис. 7.13. *Высота, не соответствующая высоте содержимого элемента*

Спецификация CSS1 определяла, что если элемент не является замещаемым (таким как изображение), то агенты пользователя могут игнорировать любое значение `height`, отличное от `auto`. В CSS2 и CSS2.1 значение `height` не может быть проигнорировано за исключением одного особого случая, относящегося к процентным значениям. Обсудим это чуть позже.

Аналогично `width` свойство `height` определяет высоту области содержимого, а не высоту видимого блока элемента. Любые отбивки, рамки и поля сверху или снизу блока элемента добавляются к значению `height`.

Свойства вертикального форматирования

С вертикальным форматированием, как и с горизонтальным, связаны семь свойств: `margin-top`, `border-top`, `padding-top`, `height`, `padding-bottom`, `border-bottom` и `margin-bottom`. Эти свойства схематически представлены на рис. 7.14.

Значения этих семи свойств должны равняться высоте блока контейнера элемента. Обычно это значение `height` родителя блочного элемен-

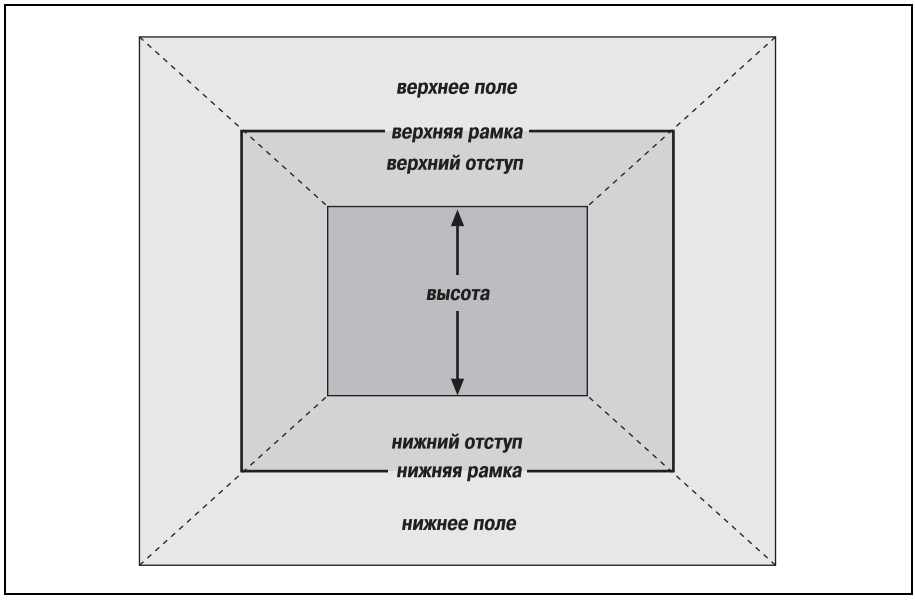


Рис. 7.14. *Семь свойств вертикального форматирования*

та (поскольку родителями блочных элементов практически всегда являются блочные элементы).

Только для трех из этих семи свойств может быть задано значение `auto`: для свойства `height` содержимого элемента и для верхнего и нижнего полей. Верхнему и нижнему отступам и рамкам должны быть присвоены конкретные значения, в противном случае по умолчанию им будут присвоены нулевые значения (если не объявлено свойство `border-style`). Если `border-style` задано, ширина рамки задается как довольно неопределенное значение `medium`. На рис. 7.15 показана схема, помогающая запомнить, какие части блока могут иметь значение `auto`, а какие – нет.

Любопытно, что если для свойств `margin-top` или `margin-bottom` блочного элемента в нормальном потоке задано значение `auto`, то им автоматически присваивается `0`. Нулевое значение, к сожалению, усложняет вертикальное центрирование элементов в их блоках-контейнерах. Это также означает, что если задать для верхнего и нижнего полей значение `auto`, то они переопределяются в `0` и удаляются из блока элемента.

Если положение элементов задается в абсолютных единицах, то верхние и нижние поля, имеющие значение `auto`, обрабатываются по-разному. Более подробно об этом рассказано в главе 8.

Свойству `height` должно быть присвоено значение `auto` или некоторое неотрицательное значение.

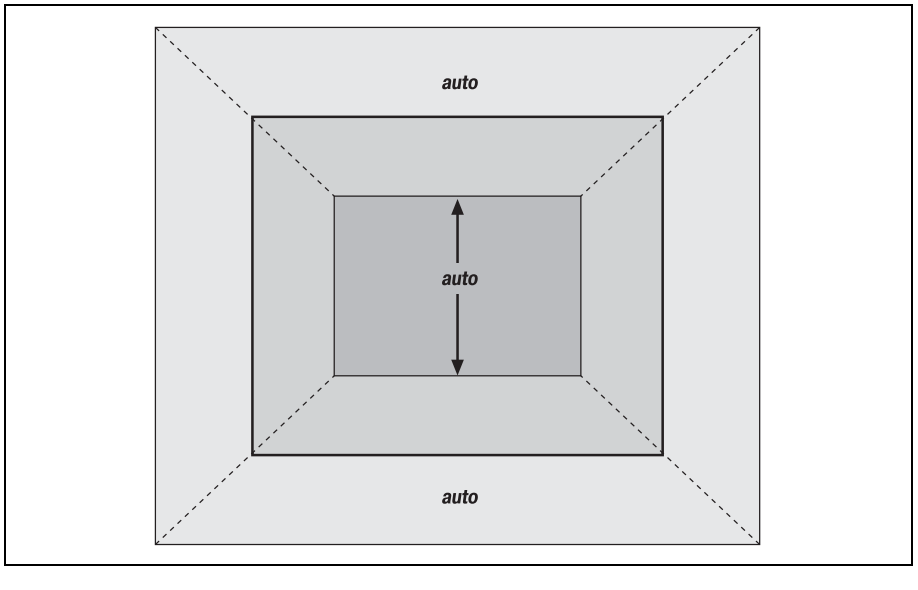


Рис. 7.15. *Свойства вертикального форматирования, которым может быть задано значение auto*

Процентные значения для задания высоты

Вы уже видели, как обрабатываются высоты, заданные в единицах измерения длины, теперь давайте обратимся к процентным значениям. Если значение свойства `height` блочного элемента в нормальном потоке задано в процентах, то его реальное значение вычисляется как часть высоты блока-контейнера. В результате применения следующей разметки высота абзаца будет `3em`:

```
<div style="height: 6em;">
 <p style="height: 50%;">Half as tall</p>
</div>
```

Поскольку присваивание верхнему и нижнему полям значения `auto` обеспечит их нулевую высоту, центрировать элемент по вертикали можно, лишь присвоив им обоим значение `25%`.

Однако в тех случаях, когда значение `height` блока-контейнера не задано явно, процентные значения заменяются на `auto`. Если предыдущий пример изменить так, чтобы свойство `height` элемента `div` имело значение `auto`, то высота параграфа будет точно такой же, как и высота самого `div`:

```
<div style="height: auto;">
 <p style="height: 50%;">NOT half as tall; height reset to auto</p>
</div>
```

Эти две возможности проиллюстрированы на рис. 7.16. (Пространство между рамкой абзаца и рамкой `div` – это верхнее и нижнее поля абзаца.)

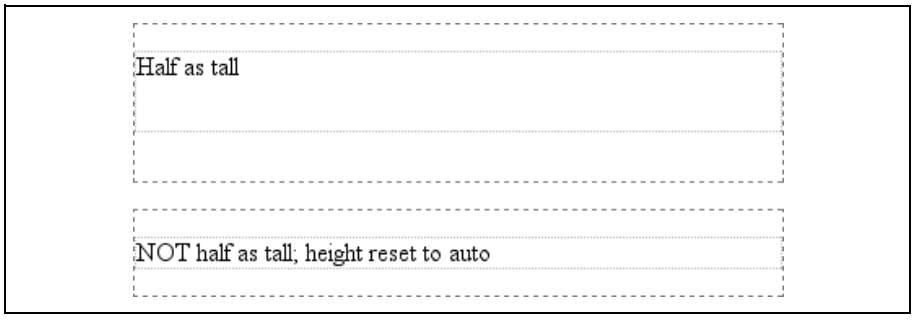


Рис. 7.16. *Процентные значения в разных ситуациях*

Автоматическое определение высоты

В самом простом случае блочный элемент со значением `height: auto` в нормальном потоке генерируется таким образом, чтобы его высоты как раз хватало для размещения строк его содержимого (включая текст). Значение `auto` задает границы абзаца и предполагает отсутствие отступов – ожидается, что нижняя рамка должна находиться сразу под нижней строкой текста, а верхняя рамка – прямо над верхней строкой текста.

Если у блочного элемента с высотой `auto` в нормальном потоке есть только блочные потомки, тогда по умолчанию высота будет расстоянием от верха внешнего края рамки самого верхнего дочернего блочного элемента до низа внешнего края нижней рамки самого нижнего дочернего блочного элемента. Следовательно, поля дочерних элементов будут выходить за границы элемента, в котором они находятся. (Это поведение объясняется в следующем разделе.) Однако если у блочного элемента есть отступ сверху или снизу или верхняя или нижняя рамки, его высотой будет расстояние от верха внешнего края верхнего поля его самого верхнего дочернего элемента до внешнего края нижнего поля самого нижнего дочернего элемента:

```
<div style="height: auto; background: silver;">
<p style="margin-top: 2em; margin-bottom: 2em;">A paragraph!</p>
</div>
<div style="height: auto; border-top: 1px solid; border-bottom: 1px solid;
background: silver;">
<p style="margin-top: 2em; margin-bottom: 2em;">Another paragraph!</p>
</div>
```

Оба этих варианта продемонстрированы на рис. 7.17.

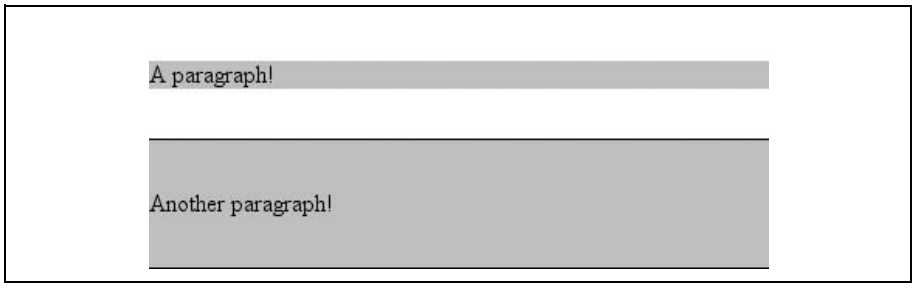


Рис. 7.17. *Результат автоматического определения высоты при наличии дочерних блочных элементов*

Если в предыдущем примере заменить рамки отступами, результат для высоты `div` не изменится: она по-прежнему будет содержать поля абзаца.

Сворачивание вертикальных полей

Еще один важный аспект вертикального форматирования – *сворачивание* (*collapsing*) смежных вертикальных полей. Сворачивание применяется только к полям. Отступы и рамки, если они присутствуют, никогда не сворачиваются.

Ненумерованный список, в котором элементы списка следуют друг за другом, – прекрасный пример сворачивания полей. Предположим, что для списка, содержащего пять элементов, объявлена следующая разметка:

```
li {margin-top: 10px; margin-bottom: 15px;}
```

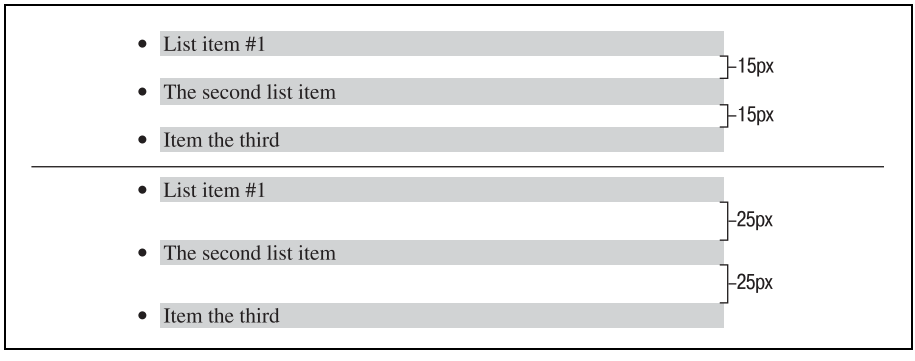


Рис. 7.18. *Свернутые поля в сравнении с несвернутыми*

У каждого элемента списка есть верхнее поле в 10 пикселов и нижнее поле в 15 пикселов. Однако при формировании визуального представления списка расстояние между соседними элементами списка составляет 15 пикселов, а не 25. Это происходит потому, что в вертикальном направлении соседние поля сворачиваются. Иначе говоря, меньшее из двух полей игнорируется в пользу большего. На рис. 7.18 показана разница между свернутыми и несвернутыми полями.

Правильно реализованные агенты пользователя осуществляют сворачивание граничащих в вертикальном направлении полей, как показано для первого списка на рис. 7.18, где расстояние между каждым элементом списка составляет 15 пикселов. Второй список показывает, что произошло бы, если бы агент пользователя не реализовывал сворачивание полей: в результате элементы списка отстояли бы друг от друга на 25 пикселов.

Те, кому не нравится слово «сворачивание», могут употреблять термин «перекрытие». То, что происходит с полями, не совсем перекрытие, но наглядно представить происходящее поможет следующая аналогия. Скажем, каждый элемент (например, абзац) представлен маленьким кусочком бумаги, на котором написано содержимое элемента. Каждый листок бумаги обрамлен прозрачным пластиком, который представляет поля. Первый листок бумаги (скажем, `h1`) выкладывается на холст. Второй листок бумаги (абзац) выкладывается под ним и затем пододвигается до тех пор, пока край обрамления первого листка не коснется края второго листка. Если у первого листка по нижнему краю имеется поле в половину дюйма, а у второго – поле в треть дюйма сверху, то когда они будут совмещены, поле первого листка будет касаться верхнего края второго листка бумаги. Мы разместили эти два листка на холсте, и поля этих листков перекрываются.

Сворачивание также происходит при встрече нескольких полей, например, в конце списка. Дополняя предыдущий пример, применим следующие правила:

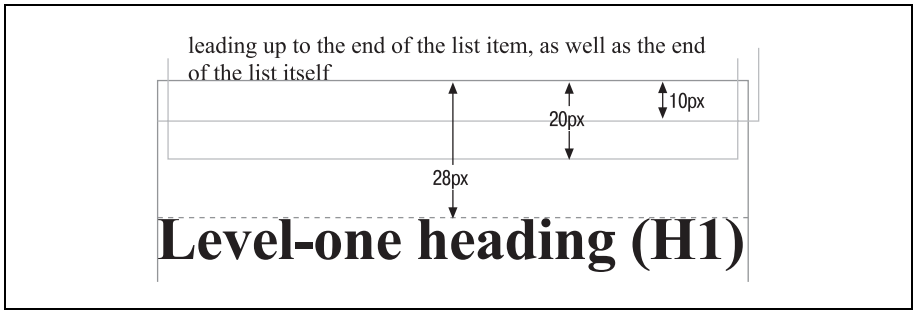


Рис. 7.19. *Сворачивание в деталях*

```
ul {margin-bottom: 15px;}
li {margin-top: 10px; margin-bottom: 20px;}
h1 {margin-top: 28px;}
```

Нижнее поле последнего элемента списка составляет 20 пикселов, нижнее поле `ul` – 15 пикселов, а верхнее поле следующего `h1` – 28 пикселов. Итак, поскольку поля сворачиваются, расстояние между концом `li` и началом `h1` составит 28 пикселов, как показано на рис. 7.19.

Теперь вспомним примеры из предыдущего раздела, где введение рамки или отступов в блоке-контейнере требовало, чтобы поля дочерних элементов размещались внутри заданных границ. Увидеть это поведение можно, добавив рамку в элемент `ul` из предыдущего примера:

```
ul {margin-bottom: 15px; border: 1px solid;}
li {margin-top: 10px; margin-bottom: 20px;}
h1 {margin-top: 28px;}
```

В результате этого изменения нижнее поле элемента `li` окажется внутри его элемента-родителя (`ul`). Поэтому сворачивание полей имеет место только между `ul` и `h1`, как показано на рис. 7.20.

Поля с отрицательными значениями

Отрицательные поля оказывают влияние на вертикальное форматирование и на то, как происходит сворачивание полей. Если среди сворачиваемых полей есть какие-то отрицательные значения, то из них бро-

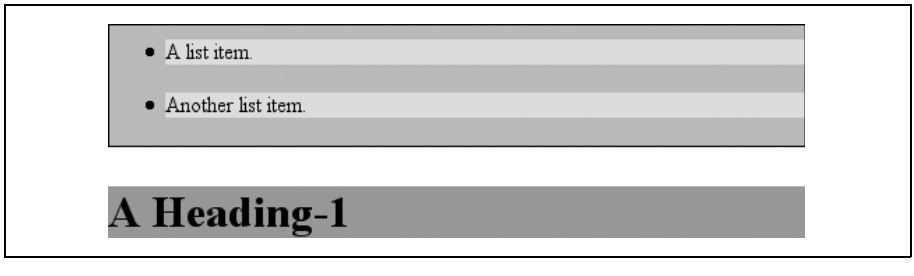


Рис. 7.20. *Сворачивание (и отсутствие сворачивания) при введении рамок*

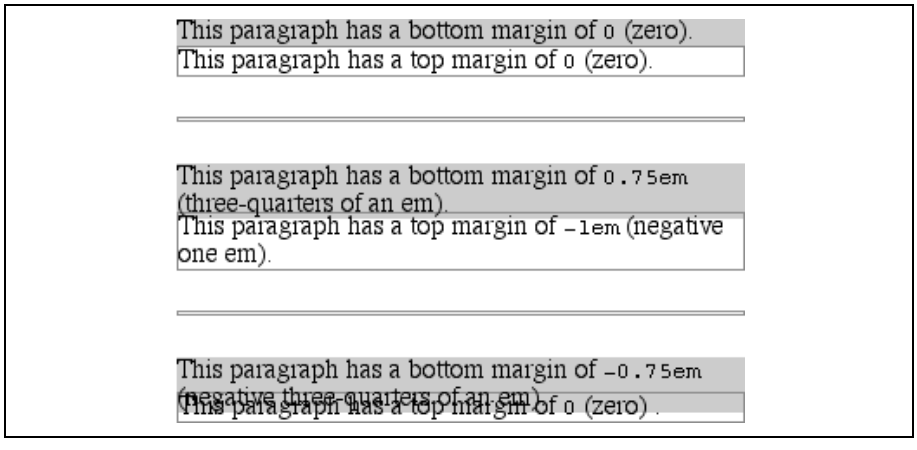


Рис. 7.21. *Примеры отрицательных вертикальных полей*

узер должен выбрать абсолютный максимум (максимум по модулю). Это абсолютное максимальное значение отрицательного поля вычитается из положительного значения поля. Иначе говоря, отрицательное значение добавляется к положительному, и полученное в результате значение и есть расстояние между элементами. На рис. 7.21 представлены два конкретных примера.

Обратите внимание на эффект «выталкивания» отрицательных верхнего и нижнего полей. На самом деле практически так же задание отрицательных значений для горизонтальных полей приводит к «выталкиванию» элемента за границы его родителя:

```
p.neg {margin-top: -50px; margin-right: 10px;
  margin-left: 10px; margin-bottom: 0;
  border: 3px solid gray;}
```

```
<div style="width: 420px; background-color: silver;
   padding: 10px; margin-top: 50px; border: 1px solid;">
<p class="neg">
A paragraph.
</p>
A div.
</div>
```

Как видно на рис. 7.22, отрицательное верхнее поле просто вытолкнуло абзац вверх. Заметьте, что содержимое элемента `div`, который расположен в разметке следом за абзацем, также было поднято вверх на 50 пикселов.

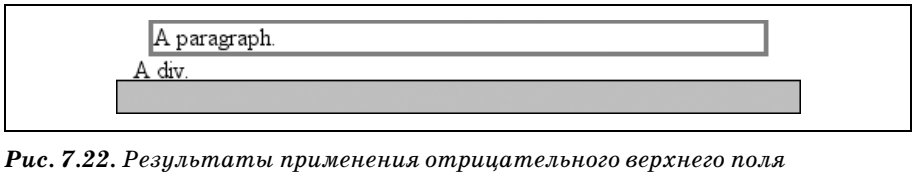


Рис. 7.22. *Результаты применения отрицательного верхнего поля*

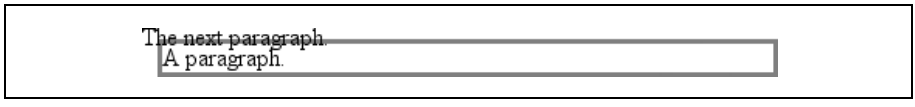


Рис. 7.23. *Результаты применения отрицательного нижнего поля*

При отрицательном нижнем поле абзац выглядит так, как будто он был вытолкнут вниз. Сравните следующую разметку с ситуацией, изображенной на рис. 7.23:

```
p.neg {margin-bottom: -50px; margin-right: 10px;
  margin-left: 10px; margin-top: 0;
  border: 3px solid gray;}

<div style="width: 420px; margin-top: 50px;">
<p class="neg">
A paragraph.
</p>
</div>
<p>
The next paragraph.
</p>
```

На рис. 7.23 происходит вот что: элементы, следующие за `div`, размещаются в зависимости от положения нижней границы элемента `div`. Как видите, элемент `div` оканчивается выше визуального низа его дочернего абзаца. Следующий после `div` элемент находится на заданном расстоянии от низа `div`. Вот чего следует ожидать, применяя эти правила.

Теперь рассмотрим пример, в котором поля последнего элемента, ненумерованного списка и абзаца сворачиваются. В этом случае ненумерованному списку и абзацу назначаются отрицательные поля:

```
li {margin-bottom: 20px;}
ul {margin-bottom: -15px;}
h1 {margin-top: -18px;}
```

Большее из двух отрицательных полей (`-18px`) добавляется к самому большому положительному полю (`20px`), что в результате дает 20px – 18px = 2px. Таким образом, между низом содержимого элемента списка и верхом содержимого `h1` всего два пиксела, как видно на рис. 7.24.

Одна из областей неразрешимого поведения – перекрытие элементов из-за отрицательных полей, в этом случае сложно сказать, какой из

Рис. 7.24. *Сворачивание полей в детальном рассмотрении*

элементов находится сверху. Можно также заметить, что ни в одном из примеров данного раздела не применяются фоновые цвета. А если бы применялись, то содержимое элементов могло бы перекрываться фоновым цветом следующего элемента. Этого следует ожидать, поскольку броузеры обычно генерируют визуальное представление элементов последовательно, руководствуясь порядком их расположения, поэтому элементы, следующие в нормальном потоке документа позже, должны располагаться поверх предыдущих элементов, допуская перекрытие с обоих концов.

Элементы списка

Для элементов списка определено несколько особых правил. Обычно им предшествует маркер, например маленькая точка или число. Этот маркер на самом деле не является частью области содержимого элемента списка, поэтому в эффектах, показанных на рис. 7.25, нет ничего необычного.

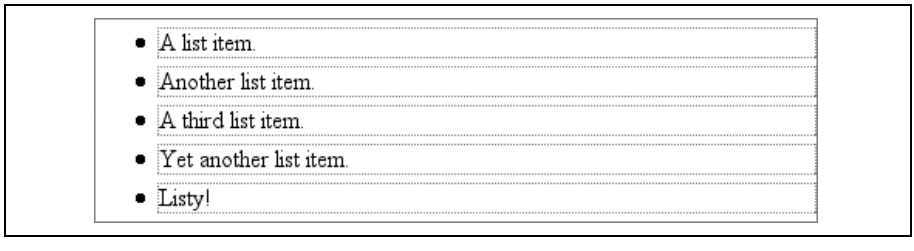
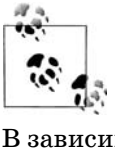


Рис. 7.25. *Содержимое элементов списка*

В CSS1 очень мало сказано о размещении этих маркеров и их влиянии на верстку документа. CSS2 предоставляет свойства, специально созданные для решения этой проблемы, например `marker-offset`. Однако их реализация запаздывала, подходы изменились, и эти свойства были выброшены из CSS2.1; похоже на то, что в будущих версиях CSS будет введен другой способ определения расстояния между содержимым и маркером. Поэтому размещение маркеров в значительной степени находится вне компетенции авторов (по крайней мере, на момент написания этой книги).



Списки и способы их оформления более подробно рассмотрены в главе 12.

В зависимости от значения свойства `list-style-position` маркеры, прикрепленные к элементу списка, могут находиться вне содержимого этого элемента или обрабатываться как строковый маркер, находящийся в начале содержимого. Если маркер вносится внутрь, элемент списка будет взаимодействовать со своими соседями точно так же, как блочный элемент (рис. 7.26).

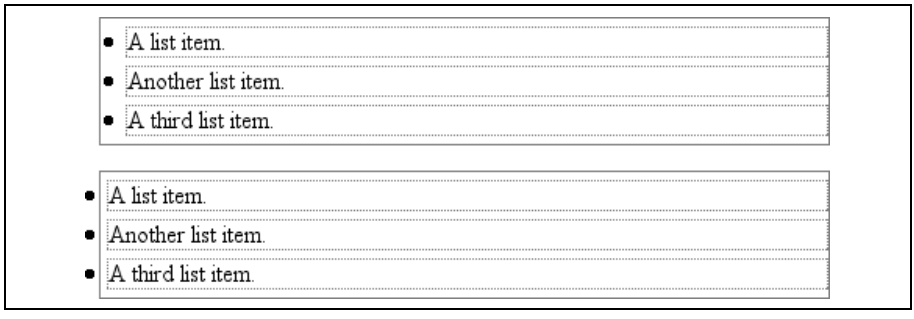


Рис. 7.26. *Маркеры внутри и вне списка*

Если маркер остается вне содержимого, он размещается на некотором расстоянии от левого края содержимого (в языках с написанием слева направо). Независимо от того, как меняются стили списка, маркер остается на фиксированном расстоянии от края содержимого. Иногда маркеры выносятся из самого списка, как видно на рис. 7.26.

Строковые элементы

После блочных элементов самыми распространенными являются строковые. Определение свойств блока для строковых элементов ставит перед нами еще ряд интересных вопросов. В качестве примера строковых элементов приведем теги `em` и `a` (оба – незамещаемые элементы) и замещаемые элементы – изображения.

Ни одна из перечисленных особенностей не относится к элементам таблиц. Для обработки таблиц и их содержимого CSS2 предоставляет новые свойства и характеристики, и поведение этих элементов очень отличается от форматирования блочных или строковых элементов. Стилевое оформление таблиц обсуждается в главе 11.

Обработка незамещаемых и замещаемых элементов в строковом контексте немного отличается, и мы будем рассматривать их по очереди по мере изучения структуры строковых элементов.

Компоновка строки

Во-первых, надо понимать, как компонуется строковое содержимое. Этот процесс не такой простой и ясный, как в блочных элементах, которые просто генерируют контейнеры и обычно не допускают сосуществования чего-то еще. Для сравнения загляните *внутрь* блочного элемента, такого как абзац. Вы вполне можете спросить: «Как все эти строки текста попали туда? Что управляет их размещением? Как я могу влиять на это?»

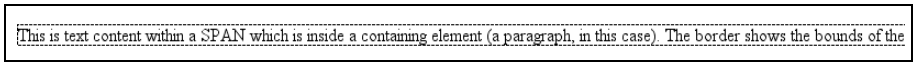


Рис. 7.27. *Однострочный строковый элемент*

Чтобы понять, как происходит генерирование строк, сначала рассмотрим элемент, содержащий одну очень длинную строку текста, как показано на рис. 7.27. Заметьте, что рамка установлена вокруг строки, целиком помещена в элемент `span` и для нее назначен стиль:

```
span {border: 1px dashed black;}
```

На рис. 7.27 представлен самый простой вариант строкового элемента, содержащегося в блочном элементе. В определенном смысле он ничем не отличается от абзаца, содержащего два слова. Единственная разница в том, что на рис. 7.27 несколько дюжин слов, и большинство абзацев не включают явных строковых элементов, таких как `span`.

Чтобы перейти от этого упрощенного варианта к чему-то более привычному, достаточно определить, насколько широким должен быть элемент, и затем разбить строку таким образом, чтобы получившиеся в результате части помещались в элемент по ширине. Результат показан на рис. 7.28.

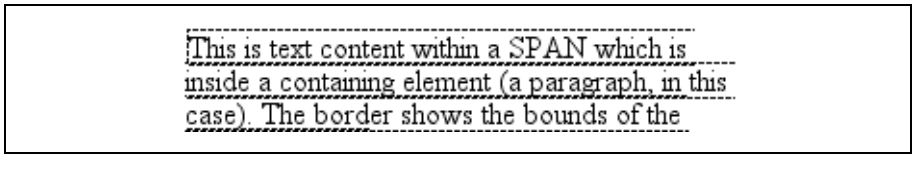


Рис. 7.28. *Многострочный строковый элемент*

Практически ничего не изменилось. Мы лишь взяли одну строку и разбили ее на части, а затем поставили эти части друг над другом.

На рис. 7.28 верхние и нижние рамки каждой строки совмещаются. Это происходит лишь потому, что для текста строкового элемента не были заданы ни отступы, ни поля. Обратите внимание, что рамки слегка перекрывают друг друга; например, нижняя рамка первой строки находится немного ниже верхней рамки второй строки. Дело в том, что на самом деле рамка отрисовывается на следующем пикселе (на мониторе) с *внешней стороны* каждой строки. Поскольку строки соприкасаются, их рамки будут перекрываться, как показано на рис. 7.28.

Если изменить стили элемента `span`, чтобы задать цвет фона, реальное размещение строк становится ясно видимым. Рассмотрим рис. 7.29, на котором представлены четыре абзаца, свойство `text-align` каждого из которых имеет разные значения, и фон строк закрашен.

Как видите, не каждая строка доходит до края области содержимого своего абзаца-родителя, которая обозначена серой пунктирной рамкой. Для абзаца, выровненного по левой границе, все строки выстроены вплотную к левому краю области содержимого абзаца и каждая

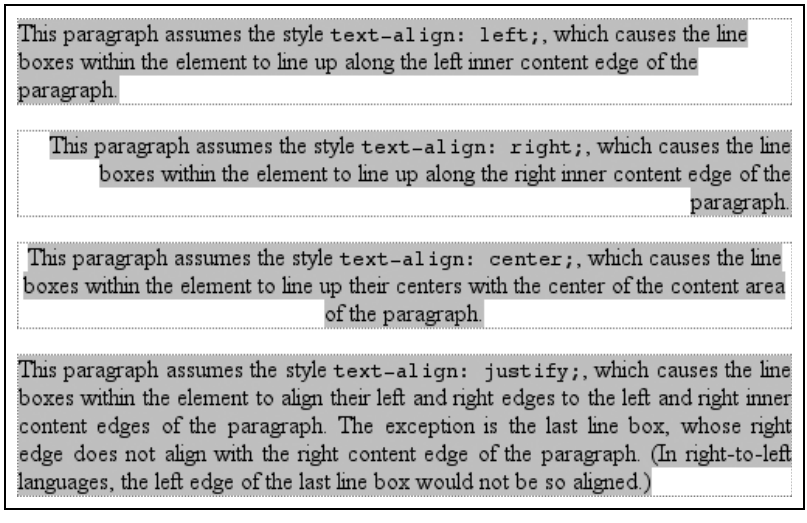


Рис. 7.29. *Различное выравнивание строк*

строка оканчивается там, где находится ее разрыв. Для абзаца, выровненного по правому краю, имеет место обратное. Для абзаца, выровненного по центру, центры строк выравниваются относительно центра абзаца. В последнем случае, когда значением `text-align` является `justify`, ширина каждой строки выравнивается соответственно ширине области содержимого абзаца, в результате края строки касаются обоих краев области содержимого абзаца. Разница между исходной длиной строки и шириной абзаца компенсируется за счет изменения расстояний между буквами и словами каждой строки. Следовательно, при выравнивании по ширине значение свойства `word-spacing` может переопределяться. (Если значение свойства `letter-spacing` выражено в единицах длины, оно не может быть переопределено.)

Это довольно точное описание процесса создания строк в самых простых случаях. Однако, как вы вскоре увидите, строковая модель форматирования далеко не проста.

Основные принципы и понятия

Перед тем как пойти дальше, давайте кратко просмотрим основные термины компоновки строки, которые будут важны для понимания следующих разделов:

Анонимный текст (anonymous text)

Это любая строка символов, не входящая в состав срочного элемента. Таким образом, в разметке `<p> Я <em>так</em> рад! </p>` последовательности «Я» и «рад!» – анонимный текст. Заметьте, что пробелы являются частью текста, поскольку пробел – такой же символ, как и остальные.

Кегельный квадрат (em box)

Определяется заданным шрифтом, еще известен как квадрат символа. Реальные глифы могут быть выше или ниже, чем их кегельные квадраты, что обсуждалось в главе 5. В CSS высоту кегельного квадрата определяет значение свойства `font-size`.

Область содержимого (content area)

Для незамещаемых элементов есть два варианта определения области содержимого, и спецификация CSS2.1 разрешает агентам пользователя выбирать один из них. Областью содержимого может быть блок, описанный объединенными вместе кегельными квадратами каждого из символов элемента, или ею может быть блок, описанный глифами символов элемента. В этой книге для простоты рассматривается определение кегельными квадратами. В замещаемых элементах область содержимого – это собственная высота элемента плюс все поля, рамки или отступы.

Межстрочный интервал (leading)

Межстрочный интервал – это разница между значениями `font-size` и `line-height`. Эта разница на самом деле делится на два и добавляется к области содержимого сверху и снизу. Эти дополнения называются, что и не удивительно, половинным межстрочным интервалом. Межстрочные интервалы применяются только к незамещаемым элементам.

Строковый блок (inline box)

Это блок, образуемый путем добавления межстрочного интервала к области содержимого. Для незамещаемых элементов высота строкового блока элемента равна значению `line-height`. Для замещаемых элементов высота строкового блока элемента эквивалентна области содержимого, поскольку к замещаемым элементам межстрочные интервалы не применяются.

Контейнер строки (line box)

Это наименьший блок, охватывающий самую верхнюю и самую нижнюю точки строковых блоков, находящихся в строке. Иначе говоря, верхний край контейнера строки размещается вдоль вершины самого высокого строкового блока, и низ совпадает с низом строкового блока, расположенного ниже всех.

CSS также включает ряд характеристик и полезных концепций, которые следуют из приведенного выше списка терминов и определений:

- Область содержимого – это аналог блока содержимого элемента уровня блока.
- Фон строкового элемента применяется к области содержимого и рамкам.
- Любая рамка строкового элемента окружает область содержимого плюс все отступы и рамки.

- Отступы, рамки и поля незамещаемых элементов не оказывают влияния на вертикальное форматирование строковых элементов или генерируемых ими блоков; т. е. они *не* влияют на высоту строкового блока элемента (и соответственно строкового блока, в котором расположен этот элемент).
- Поля и рамки замещаемых элементов влияют на высоту строкового блока этого элемента и косвенно на высоту строкового блока строки, в которой находится этот элемент.

Необходимо отметить еще вот что: в строке происходит вертикальное выравнивание строковых блоков в соответствии со значениями их свойства `vertical-align`. Я касался этого вопроса в главе 6, а здесь мы рассмотрим его более подробно.

Перед тем как двигаться дальше, шаг за шагом рассмотрим процесс построения контейнера строки, который поможет вам понять, как происходит совмещение различных частей строки при определении ее высоты:

1. Определяем высоту строкового блока каждого элемента строки в такой последовательности:
   a. Находим значения свойств `font-size` и `line-height` каждого строкового незамещаемого элемента и любого текста, который не является частью дочернего строкового элемента. Затем вычитаем значение `font-size` из `line-height` и получаем межстрочный интервал блока. Межстрочный интервал делится на два и добавляется сверху и снизу кегельных квадратов.
   b. Находим значения свойств `height`, `margin-top`, `margin-bottom`, `padding-top`, `padding-bottom`, `border-top-width` и `border-bottom-width` для каждого замещаемого элемента и суммируем их.
2. Для каждой области содержимого определяем, какая ее часть находится выше базовой линии всей строки и какая ее часть находится под базовой линией. Это непростая задача: должно быть известно положение базовой линии каждого элемента и каждого куска анонимного текста и базовой линии самой строки, затем они все должны быть выстроены в ряд. Кроме того, нижний край замещаемого элемента располагается на базовой линии строки.
3. Определяем вертикальное смещение всех элементов, для которых задано `vertical-align`. Это скажет вам, на сколько вверх или вниз будет смещен этот строковый блок элемента, и изменит местоположение элемента относительно базовой линии.
4. Теперь, когда вы знаете местоположение всех строковых блоков, окончательно вычисляем высоту контейнера строки. Для этого просто добавьте расстояние между базовой линией и вершиной самого высоко расположенного строкового блока к расстоянию между базовой линией и низом расположенного ниже всех строкового блока.

Рассмотрим подробно весь процесс, дающий ключ к грамотному стилевому оформлению строкового содержимого.

Строковое форматирование

Как было сказано в главе 6, у всех элементов есть свойство `line-height`. Его значение оказывает огромное влияние на отображение строковых элементов, так что давайте уделим этому вопросу должное внимание.

Во-первых, установим, как определяется высота строки. Высота строки (или высота контейнера строки) определяется высотой составляющих его элементов и другого содержимого, например текста. Важно понимать, что `line-height` влияет на строковые элементы и другое строковое содержимое, но *не* на блочные элементы, по крайней мере не прямо. Для блочного элемента можно задать значение `line-height`, но оно окажет заметное действие, только будучи примененным к строковому содержимому блочного элемента. Рассмотрим следующий пустой абзац:

```
<p style="line-height: 0.25em;"></p>
```

Без содержимого абзацу будет нечего отображать – мы ничего не увидим. Каким было значение `line-height` этого абзаца – `0.25em` или `25in` – в отсутствие содержимого, необходимого для создания контейнера строки, не имеет никакой разницы.

Конечно, можно задать значение `line-height` для блочного элемента и применить его ко всему содержимому блока независимо от того, есть или нет содержимое в строковых элементах. В определенном смысле тогда каждая строка текста, содержащаяся в блочном элементе, является его собственным строковым элементом независимо от того, окружена она тегами или нет. Если хотите, изобразим фиктивную последовательность тегов вот так:

```
<p>
<line>Это абзац с рядом</line>
<line>строк текста, которые образуют</line>
<line>содержимое.</line>
</p>
```

Даже несмотря на то, что на самом деле тегов `line` нет, абзац ведет себя так, как будто они есть: каждая строка текста наследует стили абзаца. Следовательно, достаточно побеспокоиться о создании правила `line-height` для блочного элемента, чтобы не потребовалось в лоб объявлять `line-height` для всех строковых элементов, фиктивных или нет.

Фиктивный элемент `line` на самом деле проясняет поведение, проистекающее из задания `line-height` для блочного элемента. Согласно спецификации CSS при объявлении `line-height` для блочного элемента задается *минимальная* высота контейнера строки для содержимого этого блочного элемента. Таким образом, объявление `p.spacious {line-height: 24pt;}` означает, что минимальная высота каждого контейнера строки составляет 24 пункта. Формально содержимое может наследовать эту высоту строки, только если это делает строковый элемент.

Большая часть текста не входит в состав строкового элемента. Поэтому, если представить, что каждая строка содержится в вымышленном элементе `line`, модель работает довольно хорошо.

Строковые незамещаемые элементы

Опираясь на уже полученную информацию о форматировании, перейдем к построению строк, содержащих только незамещаемые элементы (или анонимный текст). Это поможет вам понять разницу между незамещаемыми и замещаемыми элементами в компоновке строки.

Построение блоков

Во-первых, для строковых незамещаемых элементов или фрагмента анонимного текста значение свойства `font-size` определяет высоту области содержимого. Если `font-size` строкового элемента – `15px`, то высота области содержимого составляет 15 пикселов, потому что все кегельные квадраты элемента имеют высоту 15 пикселов, как показано на рис. 7.30.

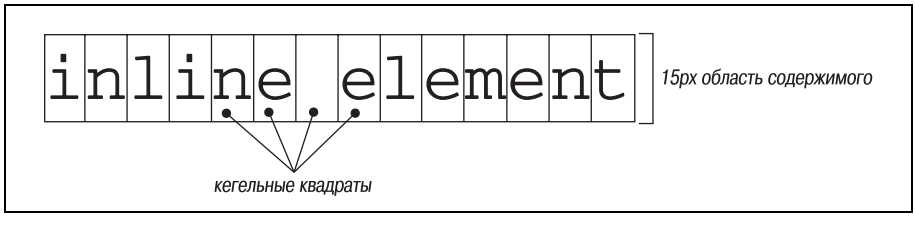


Рис. 7.30. *Кегельные квадраты определяют высоту области содержимого*

Затем следует рассмотреть значение свойства `line-height` элемента и разницу между ним и значением `font-size`. Если `font-size` строкового незамещаемого элемента составляет `15px`, а его `line-height` – `21px`, разница – 6 пикселов. Агент пользователя делит шесть пикселов пополам и половину добавляет сверху, а половину – снизу области содержимого; так формируется строковый блок. Этот процесс проиллюстрирован на рис. 7.31.

Предположим, имеется следующая разметка:

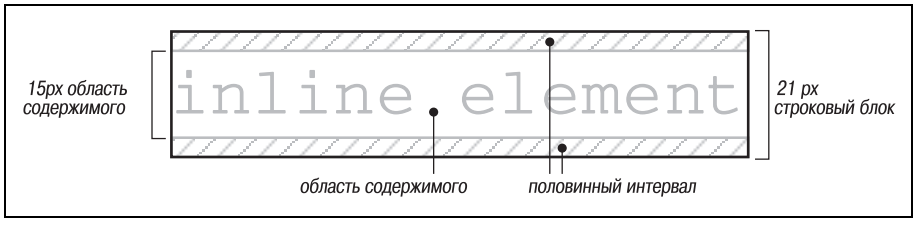


Рис. 7.31. *Область содержимого плюс межстрочный интервал равняется строковому блоку*

```
<p style="font-size: 12px; line-height: 12px;">
This is text, <em>some of which is emphasized</em>, plus other text<br>
which is <strong style="font-size: 24px;">strongly emphasized</strong>
and which is<br>
larger than the surrounding text.
</p>
```

В этом примере `font-size` большей части текста составляет `12px`, тогда как текст одного из строковых незамещаемых элементов имеет размер `24px`. Однако значение `line-height` *всего* текста – `12px`, поскольку `line-height` – это наследуемое свойство. Таким образом, `line-height` элемента `strong` – также `12px`.

Получается, что для каждой части текста, для которой и `font-size`, и `line-height` имеют значение `12px`, высота содержимого не меняется (поскольку разница между `12px` и `12px` равна нулю), т. е. высота строкового блока – 12 пикселов. Однако для текста элемента `strong` разница между значениями `line-height` и `font-size` составляет – `12px`. Эта величина делится пополам для определения половинного межстрочного интервала (`-6px`), и половинный интервал добавляется сверху и снизу к области содержимого для получения строкового блока. Поскольку в обоих случаях мы прибавляем отрицательное значение, в результате получается строковый блок высотой 12 пикселов. Этот строковый блок центрируется по вертикали в границах области содержимого элемента высотой 24 пиксела, поэтому фактически строковый блок меньше, чем область содержимого.

Пока все выглядит так, будто для каждого фрагмента текста выполняется одна и та же операция, и все строковые блоки имеют один размер, но на этом совпадение заканчивается. Строковые блоки второй строки, хотя и одного и того же размера, но не выстроены в одну линию, потому что весь текст выровнен по базовой линии (рис. 7.32).

Поскольку строковые блоки определяют высоту всего контейнера строки, важно их размещение относительно друг друга. Контейнер строки определяется как расстояние от верха самого высокого строкового блока строки до низа расположенного ниже всех строкового блока, и верх каждого контейнера строки состыковывается с низом контейнера предыдущей строки. Результат, показанный на рис. 7.32, дает нам абзац, приведенный на рис. 7.33.

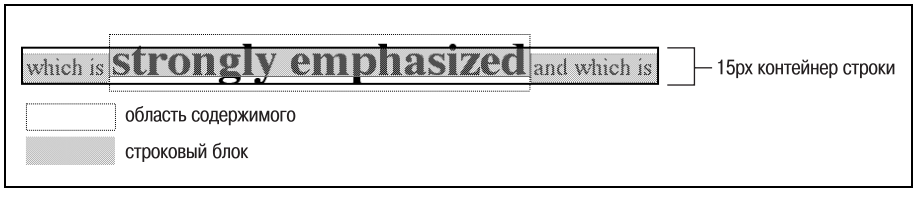


***Рис. 7.32.** Строковые блоки в строке*

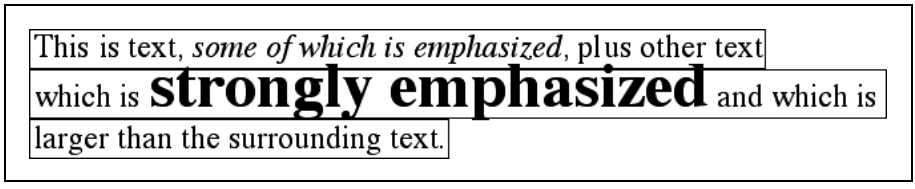


Рис. 7.33. *Контейнеры строк в рамках параграфа*

На рис. 7.33 видно, что средняя строка выше, чем остальные две, но она все-таки недостаточно велика, чтобы в ней поместился весь находящийся в ней текст. Строковый блок анонимного текста определяет низ контейнера строки, тогда как верх строкового блока элемента `strong` задает верхний уровень контейнера строки. Поскольку верх этого строкового блока находится внутри области содержимого элемента, содержимое элемента выходит за границы контейнера строки и фактически перекрывает другие контейнеры строк. В результате строки текста выглядят неровными. Позже в этой главе мы изучим, как можно справиться с этим поведением, а также методы получения постоянного интервала между базовыми линиями.

Вертикальное выравнивание

При изменении вертикального выравнивания строковых блоков применяются такие же принципы определения высоты. Предположим, для элемента `strong` задано вертикальное выравнивание в `4px`:

```
<p style="font-size: 12px; line-height: 12px;">
This is text, <em>some of which is emphasized</em>, plus other text<br>
that is <strong style="font-size: 24px; vertical-align: 4px;">strongly
emphasized</strong> and that is<br>
larger than the surrounding text.
</p>
```

Это маленькое изменение поднимает элемент на четыре пиксела, что повышает и его область содержимого, и его строковый блок. Поскольку строковый блок элемента `strong` уже был самым высоким в строке, это изменение вертикального выравнивания переносит верх контейнера строки еще на четыре пиксела вверх, как показано на рис. 7.34.

Рассмотрим другую ситуацию. Здесь в одной строке с текстом `strong` мы имеем другой строковый элемент, и он выровнен не по базовой линии:

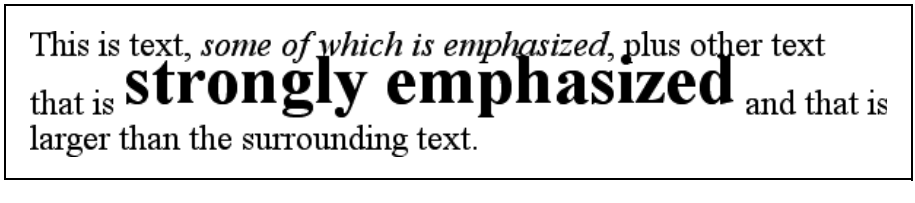


Рис. 7.34. *Вертикальное выравнивание влияет на высоту контейнера строки*

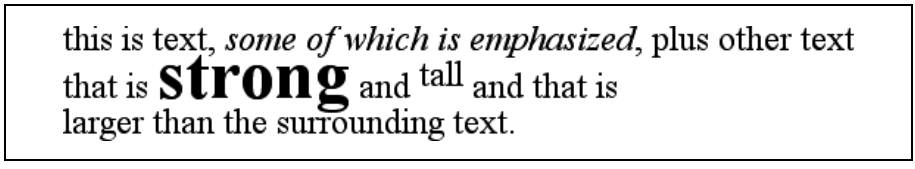


***Рис. 7.35.** Выравнивание строкового элемента по контейнеру строки*

```
<p style="font-size: 12px; line-height: 12px;">
this is text, <em>some of which is emphasized</em>, plus other text<br>
that is <strong style="font-size: 24px;">strong</strong>
and <span style="vertical-align: top;">tall</span> and that is<br>
larger than the surrounding text.
</p>
```

Получаем такой же результат, как и в предыдущем примере, в котором средний контейнер строки выше, чем остальные. Однако обратите внимание на то, как выравнивается фрагмент текста с увеличенным кеглем на рис. 7.35.

В этом случае верх строкового блока выровнен по верхней границе контейнера строки. Поскольку значения `font-size` и `line-height` этого текста равны, высота содержимого и строкового блока одинакова. Однако взгляните сюда:

```
<p style="font-size: 12px; line-height: 12px;">
This is text, <em>some of which is emphasized</em>, plus other text<br>
that is <strong style="font-size: 24px;">strong</strong>
and <span style="vertical-align: top; line-height: 4px;">tall</span>
and that is<br>
larger than the surrounding text.
</p>
```

Поскольку значение `line-height` «высокого» текста меньше, чем его значение `font-size`, строковый блок для этого элемента меньше, чем его область содержимого. Этот факт меняет размещение самого текста, т. к. верх его строкового блока должен быть выровнен по верхней границе контейнера строки. Таким образом, получается результат, показанный на рис. 7.36.

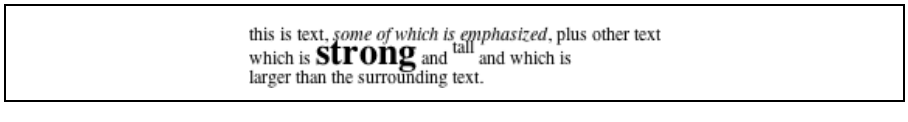


***Рис. 7.36.** Текст, выступающий из контейнера строки (вновь)*

С другой стороны, можно задать для текста значение `line-height`, превышающее значение `font-size`. Например:

```
<p style="font-size: 12px; line-height: 12px;">
This is text, <em>some of which is emphasized</em>, plus other text<br>
that is <strong style="font-size: 24px;">strong</strong>
```

```
and <span style="vertical-align: top; line-height: 18px;">tall</span>
and that is<br>
larger than the surrounding text.
</p>
```

Поскольку задано значение `line-height` в `18px`, разница между `line-height` и `font-size` составляет 6 пикселов. Половинный межстрочный интервал в 3 пиксела добавляется к области содержимого, и в результате получается строковый блок, высота которого равна 18 пикселам. Верх этого строкового блока выравнивается по верхней границе контейнера строки. Аналогичным образом значение свойства `bottom` выровняет низ строкового блока строкового элемента по нижней границе контейнера строки.

В терминах данной главы результаты применения соответствующих ключевых слов задания значения свойства `vertical-align` такие:

`top`

Выравнивает верх строкового блока элемента по верхней границе содержащего его контейнера строки.

`bottom`

Выравнивает низ строкового блока элемента по нижней границе содержащего его контейнера строки.

`text-top`

Выравнивает верх строкового блока элемента по верхней границе области содержимого родителя.

`text-bottom`

Выравнивает низ строкового блока элемента по нижней границе области содержимого родителя.

`middle`

Выравнивает среднюю точку по вертикали строкового блока элемента с точкой, расположенной на половину *ex* выше базовой линии родителя.

`super`

Перемещает область содержимого и строковый блок элемента вверх. Расстояние не определено и может меняться агентом пользователя.

`sub`

Аналогично `super`, за исключением того, что элемент перемещается вниз, а не вверх.

`<процентное_значение>`

Сдвигает элемент вверх или вниз на расстояние, определяемое как объявленная в процентах часть значения свойства `line-height` элемента.

Эти значения более подробно рассмотрены в главе 6.

Управление свойством line-height

Из предыдущих разделов мы знаем, что изменение line-height строкового элемента может привести к перекрытию одной строки текстом другой. Хотя в каждом случае изменения осуществлялись в отдельных элементах. Как же оказать на line-height элементов более глобальное влияние, чтобы уберечь содержимое от наложения?

Один из способов – применить в элементе, значение font-size которого изменилось, для задания line-height единицы измерения em. Например:

```
p {font-size: 14px; line-height: 1em;}
big {font-size: 250%; line-height: 1em;}

<p>
Not only does this paragraph have "normal" text, but it also<br>
contains a line in which <big>some big text </big>is found.<br>
This large text helps illustrate our point.
</p>
```

Задавая line-height для элемента big, мы увеличиваем общую высоту строкового блока, обеспечивая достаточное пространство для отображения элемента big без перекрытия какого-либо другого текста и без изменения line-height всех строк параграфа. Задано значение 1em, поэтому свойству line-height элемента big будет присвоено такое же значение, как и у свойства font-size элемента big. Помните, line-height задается относительно font-size самого элемента, а не его родителя. Результаты показаны на рис. 7.37.

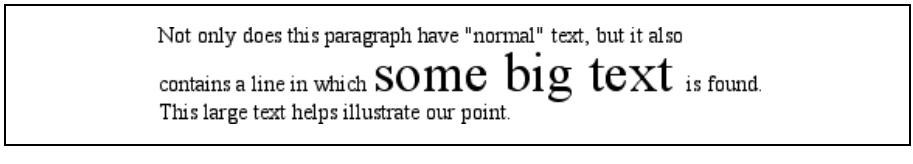


Рис. 7.37. *Задание свойства line-height строковых элементов*

Обратите внимание, что следующие стили могут обеспечить результаты, аналогичные представленным на рис. 7.37:

```
p {font-size: 14px; line-height: 1;}
big {font-size: 250%;}
```

Значения line-height не наследуются как масштабные коэффициенты, поэтому и для элемента p, и для элемента big значение line-height равно 1. Высота заданного таким образом строкового блока совпадала бы с высотой области содержимого точно так, как на рис. 7.37.

Убедитесь, что вы действительно поняли материал предыдущих разделов, потому что дальше, когда вы попытаетесь добавить рамки, все будет еще сложнее. Скажем, требуется установить вокруг всех гиперссылок рамки шириной 5 пикселов:

```
a:link {border: 5px solid blue;}
```

Если не задать свойству line-height значение, достаточно большое для вмещения рамки, возникнет опасность наложения строк. Конечно, можно увеличить размер строкового блока для непосещенных ссылок, задав line-height, как вы делали это для элемента big в предыдущем примере; в этом случае достаточно задать значение line-height, на 10 пикселов превышающее значение font-size для этих ссылок. Однако это сделать трудно, если неизвестен размер шрифта в пикселах.

Другое решение – увеличить line-height абзаца. Тогда изменятся все строки всего элемента, а не только строка, в которой находится гиперссылка, помещаемая в рамку:

```
p {font-size: 14px; line-height: 24px;}
a:link {border: 5px solid blue;}
```

Поскольку над и под каждой строкой есть дополнительные промежутки, рамка вокруг гиперссылки не вторгается ни в одну другую строку, как видно на рис. 7.38.

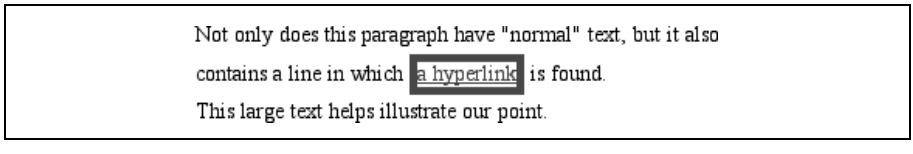


***Рис. 7.38.** Увеличиваем значение свойства line-height, чтобы обеспечить пространство для строковых рамок*

Здесь этот подход эффективен, конечно, потому, что весь текст имеет один размер. Если бы в строке присутствовали другие элементы, влияющие на высоту контейнера строки, ситуация с рамками тоже могла бы измениться:

```
p {font-size: 14px; line-height: 24px;}
a:link {border: 5px solid blue;}
big {font-size: 150%; line-height: 1.5em;}
```

Согласно этим правилам высота строкового блока элемента big составит 31,5 пиксела (14×1.5×1.5), и такой же будет высота контейнера строки. Чтобы сохранить интервалы между базовыми линиями одинаковыми, необходимо сделать line-height элемента p равным или большим, чем 32px.

Масштабирование высоты строки

Как выясняется, самый лучший способ задания line-height состоит в том, чтобы задавать в качестве значений числа без указания единиц измерения. Этот метод лучший потому, что число становится масштабным коэффициентом, и этот коэффициент – наследуемое, а не вычисляемое значение. Скажем, требуется, чтобы значение line-height всех элементов документа в полтора раза превышало их font-size:

```
body {line-height: 1.5;}
```

Базовые линии и высота строк

Фактическая высота каждого контейнера строки зависит от того, как выровнены элементы его компонентов относительно друг друга. Это выравнивание очень сильно зависит от того, где проходит базовая линия каждого элемента (или части анонимного текста), потому что ее местоположение определяет, как организованы их строковые блоки. Размещение базовой линии в рамках каждого кегельного квадрата для каждого шрифта различно. Эта информация встроена в файлы шрифтов и может быть изменена только путем редактирования этих файлов.

Таким образом, получение равноудаленных базовых линий – скорее искусство, чем наука. Если задать размеры всех шрифтов и высоты строк, используя одну единицу измерения, например `em`, тогда шансы добиться равноудаленности базовых линий будут довольно велики. Однако если единицы измерения разные, все сильно усложняется, чтобы не сказать «становится невозможным». На момент написания данной книги планируется создание свойств, которые могли бы позволить авторам обеспечивать равноудаленность базовых линий независимо от строкового содержимого, что сильно упростило бы определенные аспекты оформления текстов. Однако ни одно из этих свойств до сих пор не реализовано, что делает их введение в лучшем случае отдаленной перспективой.

Этот масштабный коэффициент `1.5` передается вниз по иерархии от элемента к элементу, и на каждом уровне коэффициент используется как множитель значения `font-size` каждого элемента. Таким образом, следующая разметка отображалась бы так, как показано на рис. 7.39:

```
p {font-size: 15px; line-height: 1.5;}
small {font-size: 66%;}
big {font-size: 200%;}
```

```
<p>This paragraph has a line-height of 1.5 times its font-size. In addition,
any elements within it <small>such as this small element</small> also have
line-heights 1.5 times their font-size...and that includes <big>this big
```

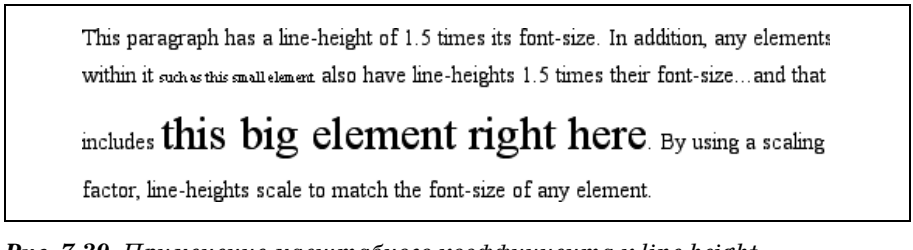


Рис. 7.39. *Применение масштабного коэффициента к line-height*

```
element right here</big>. By using a scaling factor, line-heights scale
to match the font-size of any element.</p>
```

В этом примере высота строки элемента `small` составляет `15px`, а для элемента `big` – `45px`. (Эти числа могут показаться слишком большими, но они согласуются с дизайном всей страницы.) Конечно, если вы не хотите, чтобы текст элемента `big` генерировал слишком большой межстрочный интервал, можете задать значение `line-height`, переопределяющее унаследованный масштабный коэффициент:

```
p {font-size: 15px; line-height: 1.5;}
small {font-size: 66%;}
big {font-size: 200%; line-height: 1em;}
```

Другое решение, возможно, самое простое из всех, – задать такие стили, чтобы строки были не выше, чем необходимо для вмещения их содержимого. Вот где можно задать для `line-height` значение `1.0`. Оно будет умножаться на величину `font-size` для получения `font-size` каждого элемента. Таким образом, строковый блок всех элементов будет таким же, как и область содержимого, что означает абсолютный минимум, необходимый для вмещения области содержимого каждого элемента.

Большинство шрифтов до сих пор отображают небольшой пробел между строками глифов символов, потому что обычно символы меньше, чем их кегельные квадраты. Исключением являются рукописные шрифты («курсив»), в которых глифы символов обычно *больше*, чем кегельные квадраты.

Добавление свойств блока

Как вы уже знаете, к строковым незамещаемым элементам могут применяться и отступы, и поля, и рамки. Эти аспекты строкового элемента не оказывают никакого влияния на высоту контейнера строки. Если применить к элементу `span` рамки без всяких полей или отступов, получится результат, показанный на рис. 7.40.

Край рамки строкового элемента контролируется свойством `font-size`, а не `line-height`. Иначе говоря, если `font-size` элемента `span` равно `12px`, а `line-height` – `36px`, то высота его области содержимого составляет `12px`, и рамка будет окружать область содержимого.

В качестве альтернативы можно назначить в строковом элементе отступ, который отодвинет рамку от текста:

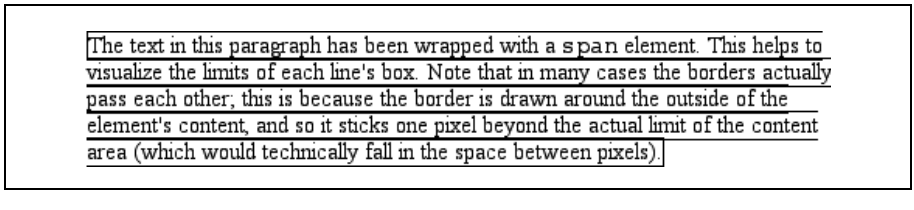


Рис. 7.40. *Строковые рамки и компоновка контейнера строки*

```
span {border: 1px solid black; padding: 4px;}
```

Заметьте, что этот отступ не меняет фактическую высоту содержимого, таким образом он не влияет на высоту строкового блока этого элемента. Аналогично добавление рамок в строковый элемент не повлияет на генерирование и компоновку контейнеров строк, что показано на рис. 7.41.

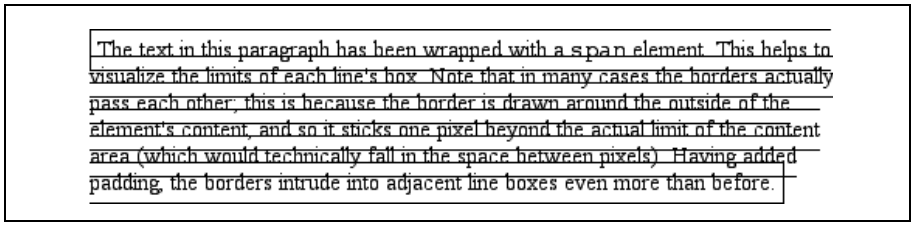


Рис. 7.41. *Отступы и рамки не меняют высоту строки*

Что касается полей, то они практически не применяются сверху и снизу строковых незамещаемых элементов, поскольку не оказывают влияния на высоту контейнера строки. Края элемента – другое дело.

CSS2.1 фактически делает размещение полей явным: он определяет свойства `margin-top` и `margin-bottom`, применяемые ко всем элементам за исключением строковых незамещаемых, вместо того, чтобы просто объявить, что агенты пользователя должны игнорировать поля сверху и снизу.

Вспомните, что строковый элемент по существу компонуется как единая строка, а затем разбивается на части. Итак, если применять поля к строковому элементу, эти поля появятся в его начале и в конце: это левое и правое поля соответственно. Отступы также появляются по краям. Таким образом, хотя отступы и поля (и рамки) не влияют на высоту строки, они все-таки могут изменять компоновку содержимого элемента, отодвигая текст от его концов. Кстати, отрицательные левое и правое поля могут придвинуть текст к строковому элементу или даже стать причиной перекрытия, как показано на рис. 7.42.

Представьте строковый элемент как полоску бумаги, вставленную в рамку. Отображение строкового элемента в виде нескольких строк аналогично разрезанию полоски бумаги. Однако к фрагментам полоски не добавляется часть рамки. Единственная часть рамки – та, которая окружала исходную полоску, поэтому рамка присутствует только в начале и в конце исходных концов полоски бумаги (строкового элемента).

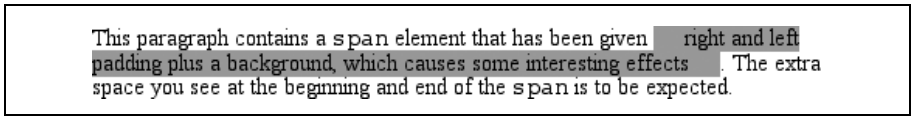


Рис. 7.42. *Отступы и поля на концах строкового элемента*

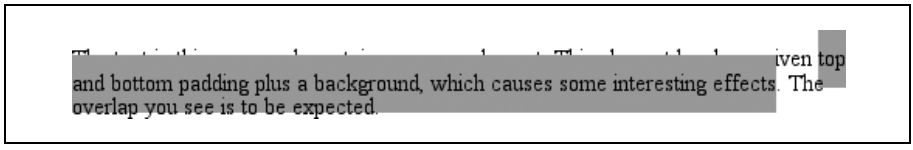


Рис. 7.43. *Перекрытие строк фоном*

Теперь посмотрим, что происходит, когда у строкового элемента есть фон и отступы, значения которых достаточно велики, чтобы обеспечить перекрытие фона строк. В качестве примера рассмотрим следующую ситуацию:

```
p {font-size: 15px; line-height: 1em;}
p span {background: #999; padding-top: 10px; padding-bottom: 10px;}
```

Высота области содержимого всего элемента `span` равна 15 пикселам, и вы добавили каждой области содержимого 10-пиксельный отступ сверху и снизу. Дополнительные пикселы не увеличат высоту контейнера строки, и все было бы замечательно, если бы не фоновый цвет. Результат представлен на рис. 7.43.

CSS2.1 явно определяет, что контейнеры строк отрисовываются в порядке представления документа: «В результате рамки последующих строк будут отрисовываться поверх рамок и текста предыдущих». Такой же принцип применяется и к фону, что и демонстрирует рис. 7.43. В то же время CSS2 позволял агентам пользователя «„срезать“ рамки и области отступов (т. е. не генерировать их визуальное представление)». Поэтому результаты сильно зависят от того, какой спецификации следует тот или иной агент пользователя.

Строковые замещаемые элементы

Предполагается, что строковые замещаемые элементы, такие как изображения, имеют собственную высоту и ширину; например, высота и ширина изображения равны определенному количеству пикселов. Следовательно, замещаемые элементы с собственной высотой могут привести к увеличению высоты контейнера строки по сравнению с обычной. Это *не меняет* значения свойства `line-height` любого элемента строки, *включая сам замещаемый элемент*. Вместо этого контейнер строки просто становится достаточно высоким, чтобы вместить замещаемый элемент, и к нему добавляются любые свойства блока. Иначе говоря, для определения строкового блока элемента используется весь замещаемый элемент – его содержимое, поля, рамки и отступы. Применение следующих стилей приводит к результату, показанному на рис. 7.44:

```
p {font-size: 15px; line-height: 18px;}
img {height: 30px; margin: 0; padding: 0; border: none;}
```

Несмотря на все это пустое пространство, эффективное значение `line-height` не изменилось ни для абзаца, ни для самого изображения. Про-

сто line-height не влияет на строковый блок изображения. Поскольку изображение на рис. 7.44 не имеет отступов, полей или рамок, его строковый блок эквивалентен его области содержимого, высота которой в данном случае составляет 30 пикселов.

Тем не менее свойству line-height замещаемого строкового элемента все-таки присваивается значение. Зачем? Обычно значение необходимо, чтобы правильно позиционировать элемент в случае вертикального выравнивания. Вспомним, например, что процентные значения свой-

Глифы и область содержимого

Даже в тех случаях, когда вы пытаетесь не допустить перекрытия фонов строковых незамещаемых элементов, все равно это может произойти. Это зависит от того, какой шрифт используется. Проблема заключается в разнице между кегельным квадратом шрифта и глифами его символов. Как выясняется, у большинства шрифтов высоты кегельных квадратов и глифов символов не совпадают.

Это, возможно, звучит весьма абстрактно, но имеет практические последствия. В CSS2.1 мы находим следующее: «Высота области содержимого должна основываться на высоте шрифта, но эта спецификация не определяет, как именно это делается. Агент пользователя может... применять кегельный квадрат или максимальный вынос верхнего и нижнего выносных элементов шрифта. (Последнее гарантировало бы, что глифы, части которых выступают за границы кегельного квадрата, все еще умещались бы в область содержимого, но в реальности приводит к тому, что блоки для разных шрифтов меняют размеры по-разному.)»

Иначе говоря, «область отрисовки» строкового незамещаемого элемента остается за агентом пользователя. Если агент пользователя за высоту области содержимого принимает кегельный квадрат, тогда фон строкового незамещаемого элемента будет равен высоте кегельного квадрата (что является значением font-size). Если агент пользователя использует максимальный вынос верхнего и нижнего выносного элементов шрифта, тогда фон может быть выше или ниже кегельного квадрата. Следовательно, вы могли бы задать строковому незамещаемому элементу line-height в 1em, а фон по-прежнему перекрывал бы содержимое других строк.

В CSS2 или CSS2.1 нет возможности предотвратить это перекрытие, но для CSS3 предлагаются свойства, которые могли бы предоставить автору возможность управлять поведением агента пользователя. До тех пор, пока эти свойства не будут реализованы широко, по-настоящему точная разметка текстов с использованием CSS невозможна.

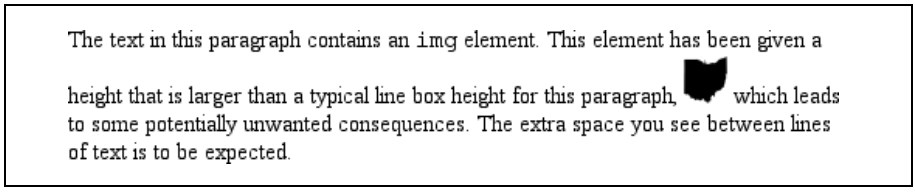


Рис. 7.44. *Замещаемые элементы могут увеличить высоту контейнера строки, но не значение свойства line-height*

ства `vertical-align` вычисляются относительно значения `line-height` элемента. Таким образом:

```
p {font-size: 15px; line-height: 18px;}
img {vertical-align: 50%;}

<p>Изображение в этом предложении <imgsrc="test.gif" alt="test image"> будет
поднято на 9 пикселов.</p>
```

Унаследованное значение `line-height` обусловливает поднятие изображения на девять пикселов, а не на какую-либо другую величину. Если бы значение `line-height` отсутствовало, было бы невозможно осуществить вертикальное выравнивание с применением процентных значений. Когда речь идет о вертикальном выравнивании, высота самого изображения значения не имеет, важно только значение `line-height`.

Однако для других замещаемых элементов может быть важным передавать значение `line-height` элементам-потомкам данного замещаемого элемента. К примеру, SVG-изображение, которое использует CSS для стилевого оформления любого текста, находящегося в изображении.

Дополнительные свойства блока

После всего этого применение полей, рамок и отступов к замещаемым строковым элементам кажется почти простым.

Отступы и рамки применяются к замещаемым элементам как обычно; отступ добавляет пространство вокруг фактического содержимого, и рамка окружает отступ. Необычно в данном случае то, что эти два фактора в действительности влияют на высоту контейнера строки, потому что они – часть строкового блока замещаемого строкового элемента (в отличие от строковых незамещаемых элементов). Рассмотрим рис. 7.45, показывающий результат применения следующих стилей:

```
img {height: 20px; width: 20px;}
img.one {margin: 0; padding: 0; border: 1px dotted;}
img.two {margin: 5px; padding: 3px; border: 1px solid;}
```

Обратите внимание, что высота первого контейнера строки достаточна для включения изображения, тогда как высоты второй строки хватает, чтобы вместить изображение, его отступы и рамки.

Поля также входят в контейнер строки, но с ними связаны определенные сложности. Задание положительного поля не приводит к загадоч-

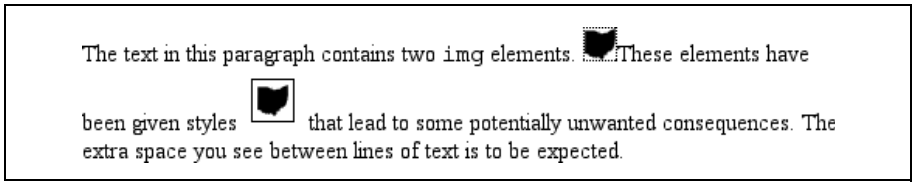


Рис. 7.45. *Добавление отступов, рамок и полей в строковый замещаемый элемент увеличивает его строковый блок*

ным явлениям; оно просто поднимет строковый блок замещаемого элемента. Точно так же задание отрицательных полей имеет подобный эффект: уменьшает размер строкового блока замещаемого элемента. Это проиллюстрировано на рис. 7.46, на котором видно, что отрицательное верхнее поле опускает строку над изображением:

```
img.two {margin-top: -10px;}
```

Отрицательные поля в блочных элементах работают так, как описано выше. Они уменьшают строковый блок замещаемого элемента по сравнению с обычным размером. Отрицательные поля – единственный способ заставить замещаемые строковые элементы перекрывать другие строки.

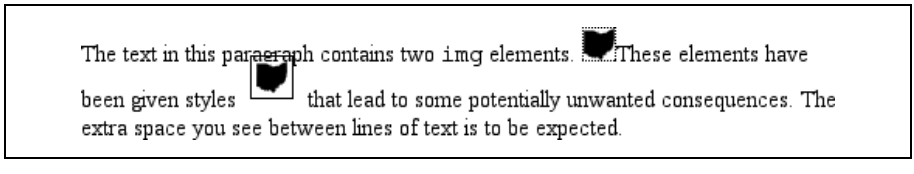


Рис. 7.46. *Эффект применения отрицательных полей к строковым замещаемым элементам*

Замещаемые элементы и базовая линия

К этому времени вы, возможно, заметили, что по умолчанию замещаемые строковые элементы располагаются на базовой линии. Если добавить в замещаемый элемент отступ снизу, поле или рамку, область содержимого переместится вверх. На самом деле у замещаемых элементов нет собственных базовых линий, так что лучше всего выровнять низ их строковых блоков по базовой линии. Таким образом, по базовой линии выравнивается нижний внешний край поля, как показано на рис. 7.47.

Это выравнивание по базовой линии приводит к неожиданным (и нежелательным) последствиям: изображение, помещенное в ячейку таблицы, само по себе должно сделать ячейку достаточно высокой, чтобы вместить содержащий изображение контейнер строки. Размер изменяется, даже если в ячейке с изображением нет ни текста, ни пробела. Поэтому обычные в прошлом варианты верстки с сегментированными изображениями и GIF-разделителями в современных броузерах могут развалиться. Рассмотрим самый простой случай:

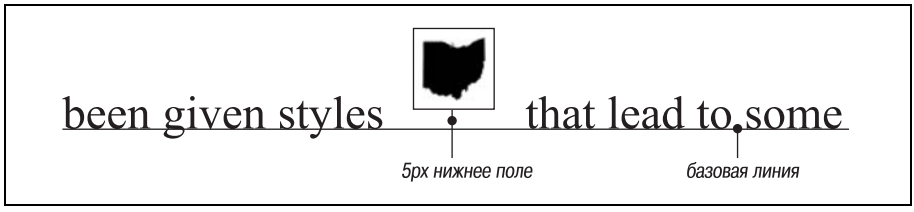


Рис. 7.47. *Строковые замещаемые элементы располагаются на базовой линии*

```
td {font-size: 12px;}

<td><img src="spacer.gif" height="1" width="10"></td>
```

При использовании строковой модели форматирования CSS высота ячейки таблицы будет 12 пикселов и изображение будет располагаться на базовой линии ячейки. Так что под изображением может оставаться просвет в 3 пиксела и в 8 над ним, хотя точные расстояния зависели бы от выбранного семейства шрифтов и размещения его базовой линии. Такое поведение не ограничивается изображениями, находящимися в ячейках таблицы; это произойдет в любом случае, если замещаемый строковый элемент – единственный потомок блочного элемента или ячейки таблицы. Например, изображение внутри `div` также будет располагаться на базовой линии.

На момент написания данной книги многие броузеры фактически игнорируют эту строковую модель форматирования CSS, но броузеры компании Netscape Communications при формировании визуального представления документов XHTML и строгого HTML действуют, как описывает текст. Для получения более подробной информации см. статью «Images, Tables, and Mysterious Gaps» по адресу *http://developer.mozilla.org/en/docs/ Images,_Tables,_and_Mysterious_Gaps*.

Проще всего в таких обстоятельствах сделать изображения-разделители блочными элементами, чтобы они не генерировали контейнер строки. Например:

```
td {font-size: 12px;}
img.block {display: block;}

<td><img src="spacer.gif" height="1" width="10" class="block"></td>
```

Еще можно задать свойствам `font-size` и `line-height` ячейки таблицы значения `1px`, при этом высота контейнера строки будет соответствовать изображению в один пиксел, находящемуся в ней.

Вот еще один интересный эффект от применения замещаемых строковых элементов, располагающихся на базовой линии: если задать отрицательное нижнее поле, элемент на самом деле будет сдвинут вниз, потому что низ его строкового блока будет располагаться выше, чем низ

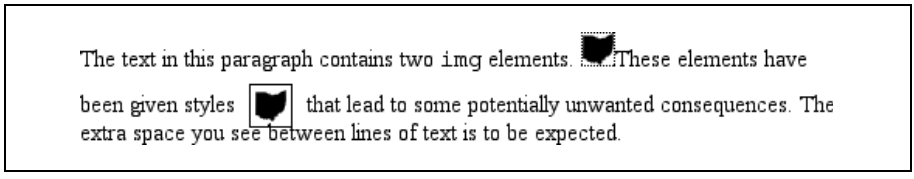


***Рис. 7.48.** Смещение строковых замещаемых элементов вниз в результате применения отрицательного нижнего поля*

его области содержимого. Таким образом, применение следующего правила дало бы результат, представленный на рис. 7.48:

```
p img {margin-bottom: -10px;}
```

Это может запросто привести к перетеканию замещаемых элементов на следующие строки текста, как показано на рис. 7.48.

Некоторые броузеры просто размещают область содержимого на базовой линии и игнорируют отрицательные значения нижнего поля.

Изменение представления элемента

Как я кратко упоминал в главе 1, автор может влиять на способ представления элемента агентом пользователя, задавая значение свойства `display`. Теперь, когда мы поближе познакомились с визуальным форматированием, давайте вернемся к свойству `display` и обсудим еще два его значения, основываясь на понятиях, рассмотренных в данной главе.

display

Значения:	`none | inline | block | inline-block | list-item | run-in | table | inline-table | table-row-group | table-header-group | table-footer-group | table-row | table-column-group | table-column | table-cell | table-caption | inherit`
Начальное значение:	`inline`
Область применения:	все элементы
Наследование:	нет
Вычисляемое значение:	меняется для перемещаемых, абсолютно позиционированных и корневых элементов (см. CSS2.1, раздел 9.7); в противном случае – как задано
Примечание:	значения `compact` и `marker` появились в CSS2, но были изъяты из CSS2.1 из-за отсутствия широкой поддержки

Мы не будем рассматривать касающиеся таблиц значения, поскольку они обсуждаются в главе 11, а также проигнорируем значение `list-item` – подробнее списки будут рассмотрены в главе 12. Мы довольно много времени посвятили обсуждению элементов `block` и `inline`, но перед тем как обратиться к `inline-block` и `run-in`, давайте немного поговорим о том, как изменение роли элемента в формировании представления может изменить верстку.

История строковой модели

Строковая модель форматирования CSS может показаться чрезмерно сложной и в некоторой степени даже несоответствующей ожиданиям автора. К сожалению, сложность есть результат создания языка стилей, который является обратно-совместимым с веб-броузерами эпохи до CSS и одновременно оставляет возможность для будущего распространения на более сложные задачи – неудобная смесь прошлого и будущего. Это также результат принятия некоторых решений, основанных на здравом смысле, которые, отменяя один нежелательный эффект, становятся причиной возникновения другого.

Например, «разведение» строк текста изображением и вертикально выровненным текстом берет начало еще в поведении Mosaic 1.0. В этом броузере любое изображение в абзаце просто раздвигало строки, чтобы высвободить достаточное количество места для своего размещения. И это правильно, поскольку предотвращает перекрытие изображением текста других строк. Вот почему, когда CSS вводил способы стилевого оформления текста и строковых элементов, его авторы пытались создать модель, которая обеспечивала (по умолчанию) отсутствие перекрытия других строк текста изображениями. Однако эта же модель предполагает, что, например, элемент верхнего надстрочного индекса (`sup`) скорее всего также разведет строки текста.

Подобные эффекты раздражают некоторых авторов, которые хотят, чтобы базовые линии в их документах находились на строго определенном расстоянии. Рассмотрим альтернативный подход. Если в соответствии со значением `line-height` базовая линия располагается на заданном расстоянии, можно запросто распрощаться с замещаемыми строковыми и сдвигаемыми по вертикали элементами, которые перекрывают другие строки текста, что также расстроило бы авторов. К счастью, CSS предлагает достаточные средства для создания желаемых эффектов тем или иным способом, и в CSS, таким образом, закладывается даже еще больший потенциал.

Изменение ролей

Когда дело доходит до стилевого оформления документа, очевидно, удобно иметь возможность менять роли элемента в представлении документа. Пусть имеется ряд ссылок в элементе `div`, которые требуется сверстать в виде вертикальной врезки:

```
<div id="navigation">
<a href="index.html">WidgetCo Home</a><a href="products.html">Products</a>
<a href="services.html">Services</a><a href="fun.html">Widgety Fun!</a>
<a href="support.html">Support</a><a href="about.html" id="current">About
Us</a>
<a href="contact.html">Contact</a>
</div>
```

Можно поместить все ссылки в таблицу, или заключить каждую в элемент `div`, или просто сделать их все блочными элементами, вот так:

```
div#navigation a {display: block;}
```

В результате каждый элемент `a`, находящийся внутри `div` с идентификатором `navigation`, станет блочным элементом. Если добавить еще несколько стилей, можно получить результат, представленный на рис. 7.49.

***Рис. 7.49**. Изменение роли элемента в формировании представления (со строкового элемента на блочный)*

Изменение ролей в формировании представления может быть полезным в тех случаях, когда требуется, чтобы броузеры, не поддерживающие CSS, получали навигационные ссылки как строковые элементы, но компоновали эти ссылки как блочные элементы. Если ссылки представлены как блоки, можно оформлять их так, как оформлялись бы элементы `div`, благодаря чему весь блок элемента становится частью ссылки. Таким образом, возможность щелкнуть по ссылке появляется, когда указатель мыши проходит над любым местом блока элемента.

Можно также сделать элементы строковыми. Предположим, есть ненумерованный список имен:

```
<ul id="rollcall">
<li>Bob C.</li>
<li>Marcio G.</li>
<li>Eric M.</li>
<li>Kat M.</li>
```

```
<li>Tristan N.</li>
<li>Arun R.</li>
<li>Doron R.</li>
<li>Susie W.</li>
</ul>
```

Предположим, требуется представить имена рядом строковых элементов, между которыми (и по краям списка) находятся вертикальные черточки. Единственный способ сделать это – изменить их роли в формировании представления. Применение следующих правил даст результат, показанный на рис. 7.50:

```
#rollcall li {display: inline; border-right: 1px solid; padding: 0 0.33em;}
#rollcall li:first-child {border-left: 1px solid;}
```

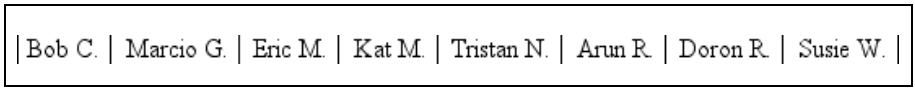


Рис. 7.50. *Изменение роли элемента в формировании представления (с элемента списка на строковый элемент)*

Существует огромное количество других полезных вариантов применения свойства `display`. Творите и увидите, что можно изобрести!

Однако будьте осторожны и помните, что вы меняете роли элементов в формировании представления, а не их суть. Иначе говоря, заставляя абзац генерировать строковые блоки, вы *не* превращаете этот абзац в строковый элемент. В XHTML, например, некоторые элементы являются блочными, тогда как другие – строковыми. (А другие представляют собой «обтекаемые» элементы, но пока забудем о них.) Строковые элементы могут быть потомками блочных, но не наоборот. Таким образом, ссылку можно поместить в абзац, но абзац не может находиться в ссылке. Это справедливо независимо от того, как оформлены рассматриваемые элементы. Рассмотрим следующую разметку:

```
<a href="http://www.example.net" style="display: block;">
<p style="display: inline;">это неверно!</p>
</a>
```

Эта разметка не будет действительной, потому что блочный элемент (`p`) вложен в строковый (`a`). Изменение ролей в формировании представления ничего не меняет. Свойство `display` так названо, потому что оказывает влияние на представление документа, а не на тип элемента.

Строчно-блочные элементы

В соответствии с гибридным внешним видом имени значения `inline-block` строчно-блочные элементы и в самом деле представляют собой гибрид блочных и строковых элементов. Это значение свойства `display` введено в CSS2.1.

Строчно-блочный элемент ведет себя по отношению к другим элементам и содержимому как строковый блок. Иначе говоря, он компонуется в строку текста, как это происходило бы с изображением, и, по сути, строчно-блочные элементы форматируются в строке как замещаемый элемент. Это означает, что низ такого элемента по умолчанию располагается на базовой линии строки текста и не разрывает строку.

Содержимое строчно-блочного элемента форматируется так же, как для блочных элементов. К нему применяются свойства `width` и `height`, как это происходит для любого блочного или замещаемого строкового элемента, и эти свойства увеличивают высоту строки, если их значение превышает эти величины для остального содержимого.

Рассмотрим пример разметки, который поможет прояснить ситуацию:

```
<div id="one">
This text is the content of a block-level level element.  Within this block-
level element is another block-level element.  <p>Look, it's a block-level
paragraph.</p>  Here's the rest of the DIV, which is still block-level.
</div>
<div id="two">
This text is the content of a block-level level element.  Within this block-
level element is an inline element.  <p>Look, it's an inline paragraph.</p>
Here's the rest of the DIV, which is still block-level.
</div>
<div id="three">
This text is the content of a block-level level element.  Within this block-
level element is an inline-block element.  <p>Look, it's an inline-block
paragraph.</p>  Here's the rest of the DIV, which is still block-level.
</div>
```

К этой разметке применяем следующие правила:

```
div {margin: 1em 0; border: 1px solid;}
p {border: 1px dotted;}
div#one p {display: block; width: 6em; text-align: center;}
div#two p {display: inline; width: 6em; text-align: center;}
div#three p {display: inline-block; width: 6em; text-align: center;}
```

Результат применения данной таблицы стилей отражен на рис. 7.51.

Обратите внимание, что во втором `div` абзац форматируется как обычное строковое содержимое, т. е. свойства `width` и `text-align` игнорируются (поскольку они не применяются к строковым элементам). Однако для третьего `div` строчно-блочный абзац обработал оба свойства, поскольку он форматируется как блочный элемент. Кроме того, строки текста этого абзаца более высокие, поскольку он оказывает на высоту строки такой эффект, какой оказывал бы замещаемый элемент.

Если свойство `width` строчно-блочного элемента не определено или прямо объявлено как `auto`, блок элемента сожмется до размера содержимого. То есть ширина блока элемента будет именно такой, какая необходима для вмещения содержимого, но не шире. Строковые блоки ведут себя аналогично, хотя они могут добавлять разрывы в строки тек-

ста, чего строчно-блочные элементы не могут. Таким образом, если применить к предыдущему примеру разметки следующее правило:

```
div#three p {display: inline-block; height: 2em;}
```

будет создан высокий блок, ширина которого будет как раз достаточной для того, чтобы вместить содержимое, как показано на рис. 7.52.

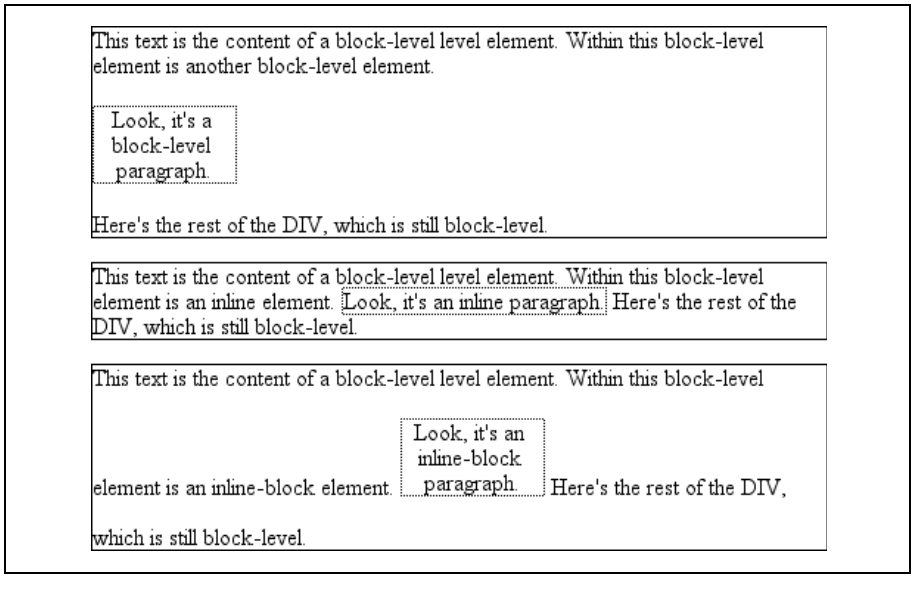


Рис. 7.51. *Поведение строчно-блочного элемента*

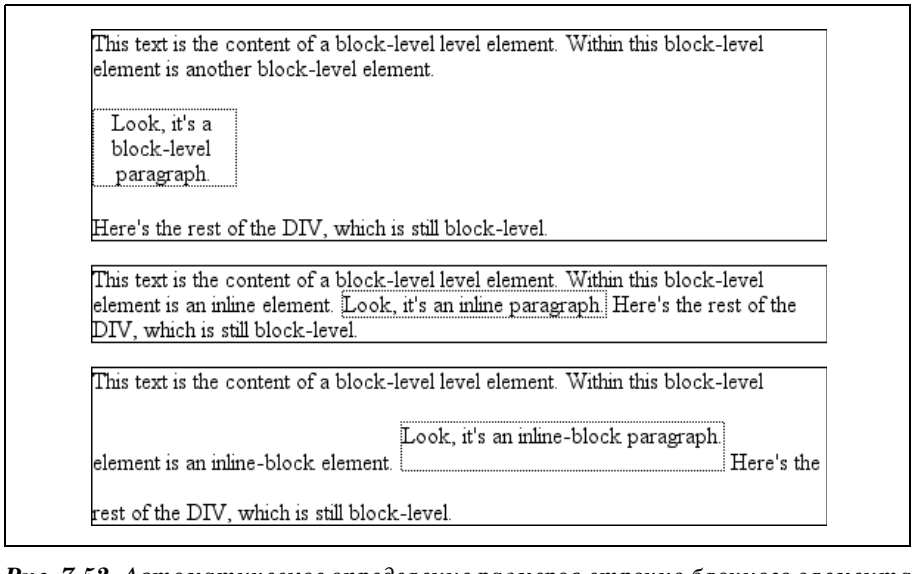


Рис. 7.52. *Автоматическое определение размеров строчно-блочного элемента*

Строчно-блочные элементы могут быть полезны, если, например, у вас есть набор из пяти гиперссылок, которые требуется разместить на панели инструментов на равных расстояниях. Чтобы сделать их ширину равной 20% ширины их родительского элемента, но все-таки оставить их строковыми, объявите:

```
#navbar a {display: inline-block; width: 20%;}
```

Инициальные элементы

CSS2 вводит значение `run-in`, еще один интересный гибрид блока/строки, который может сделать некоторые блочные элементы строковой частью следующего элемента. Эта возможность полезна для реализации некоторых эффектов, привычных в оформлении печатных текстов, когда заголовок выглядит как часть абзаца.

В CSS можно сделать элемент инициальным, лишь изменив значение его свойства `display` *и* объявив следующий блок элемента структурным[1] (block-level box). Заметьте, что я здесь веду речь о *блоках*, а не о самих элементах. Иначе говоря, не имеет значения, строковый это элемент или блочный. Важен только вид блока, генерируемого элементом. Элемент `strong`, для которого задано свойство `display: block`, генерирует структурный блок (block-level box); абзац, для которого задано `display: inline`, генерирует строковый блок (inline box).

Итак, перефразируем: если элемент генерирует инициальный блок и за ним следует структурный блок, то инициальный элемент будет строковым блоком в начале структурного блока. Например:

```
<h3 style="display: run-in; border: 1px dotted; font-size: 125%;
  font-weight: bold;">Run-in Elements</h3>
<p style="border-top: 1px solid black; padding-top: 0.5em;">
Another interesting block/inline hybrid is the value <code>run-in</code>,
introduced in CSS2, which has the ability to take block-level elements and
make them an inline part of a following element. This is useful for certain
heading effects that are quite common in print typography, where a heading
will appear as part of a paragraph of text.
</p>
```

Поскольку элемент, следующий за `h3`, генерирует структурный блок, элемент `h3` будет превращен в строковый элемент, располагающийся в начале содержимого элемента `p`, как показано на рис. 7.53.

Обратите внимание, как размещаются рамки этих двух элементов. Эффект от применения `run-in` в данной ситуации совершенно такой же, как если бы вместо него применили следующую разметку:

---

[1] Поскольку автор подчеркивает, что в этом разделе речь идет не о самих элементах, а об их интерпретации, то во избежание путаницы *block-level box* называется структурным блоком, а не блочным элементом. – *Примеч. ред.*

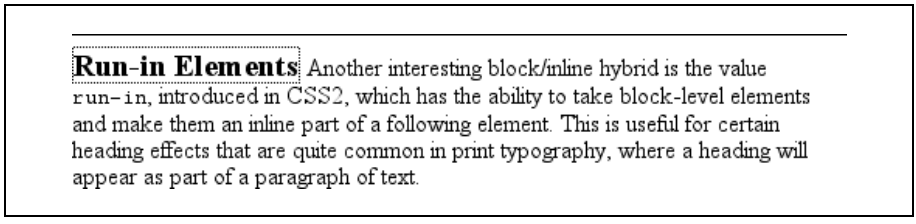


Рис. 7.53. *Делаем заголовок инициальным элементом*

```
<p style="border-top: 1px solid black; padding-top: 0.5em;">
<span style="border: 1px dotted; font-size: 125%; font-weight: bold;">Run-in
Elements</span> Another interesting block/inline hybrid is the value
<code>run-in</code>, introduced in CSS2, which has the ability to take block-
level elements and make them an inline part of a following element. This is
useful for certain heading effects that are quite common in print typography,
where a heading will appear as part of a paragraph of text.
</p>
```

Однако есть небольшое различие между инициальными блоками и этим примером разметки. Даже несмотря на то, что инициальные блоки форматируются как находящиеся в другом элементе строковые блоки, они все-таки наследуют свойства своего родительского элемента в документе, а не элемента, в который они помещены. Дополним наш пример и введем контейнер `div` и какой-нибудь цвет:

```
<div style="color: silver;">
<h3 style="display: run-in; border: 1px dotted; font-size: 125%;
  font-weight: bold;">Run-in Elements</h3>
<p style="border-top: 1px solid black; padding-top: 0.5em; color: black;">
Another interesting block/inline hybrid is the value <code>run-in</code>,
introduced in CSS2, which has the ability to take block-level elements and
make them an inline part of a following element.
</p>
</div>
```

В этой ситуации `h3` будет серебряным, а не черным, как показано на рис. 7.54. Дело в том, что он наследует значение цвета своего родительского элемента до того, как вставляется в абзац.

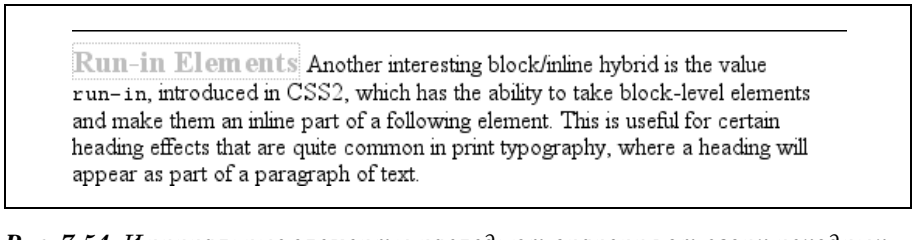


Рис. 7.54. *Инициальные элементы наследуют значения от своих исходных родителей*

Важно запомнить, что run-in будет работать, только если блок, следующий за инициальным блоком, является структурным блоком. Если нет, инициальный блок сам станет структурным. Таким образом, в результате применения следующей разметки элемент h3 будет оставаться или даже станет структурным блоком, поскольку значение свойства display для элемента table (как это ни странно) – таблица:

```
<h3 style="display: run-in;">Цены</h3>
<table>
<tr><th>Яблоки</th><td>$0.59</td></tr>
<tr><th>Персики</th><td>$0.79</td></tr>
<tr><th>Тыква</th><td>$1.29</td></tr>
<tr><th>Пирог</th><td>$6.99</td></tr>
</table>
```

Вряд ли автор когда-нибудь применит значение run-in к строковому элементу, но если это случится, элемент, скорее всего, сгенерирует структурный блок. Например, элемент em в следующей разметке стал бы структурным, поскольку за ним не следует структурный блок:

```
<p>
Это <em>на самом деле</em> лишнее, <strong>но</strong> вы могли бы сделать
это, если бы захотели.
</p>
```

На момент написания данной книги поддержку run-in предлагают лишь несколько броузеров.

Вычисляемые значения

Вычисляемое значение свойства display может меняться, если элемент свободно перемещаемый или абсолютно позиционированный. Оно также может меняться, если объявлено для корневого элемента. Кстати, значения display, position и float весьма интересно взаимодействуют.

Если элемент абсолютно позиционированный, float присваивается значение none. Для перемещаемых или абсолютно позиционированных элементов вычисляемое значение определяется объявленным значением, как показано в табл. 7.1.

Таблица 7.1. Вычисляемые значения свойства display

Объявляемое значение	Вычисляемое значение
inline-table	table
inline, run-in, table-row-group, table-column, table-column-group, table-header-group, table-footer-group, table-row, table-cell, table-caption, inline-block	block
Все остальные	Как задано

Для корневого элемента объявление значений `inline-table` или `table` в результате дает вычисляемое значение `table`, тогда как объявление `none` приводит к такому же вычисляемому значению. Все остальные значения свойства `display` приводятся к значению `block`.

Заключение

Некоторые аспекты модели форматирования CSS поначалу могут показаться сложными для понимания, но по мере работы с ними обретают более ясный смысл. Во многих случаях правила, на первый взгляд бессмысленные или даже абсурдные, оказывается, предупреждают ненормальное или нежелательное представление документа. Блочные элементы во многом проще для понимания, и их компоновка обычно не связана со сложностями. Строковые элементы, с другой стороны, может быть, сложно организовывать, поскольку в игру вступает ряд факторов, не последний из которых – тип элемента (замещаемый или незамещаемый). Итак, фундамент верстки документа установлен, и пора обратить внимание на способы применения различных свойств макета. Это займет несколько глав, и начнем мы с самых распространенных свойств блока: отступов, рамок и полей.

8

Отступы, рамки и поля

Если вы похожи на большинство веб-разработчиков конца 1990-х, то при верстке всех своих страниц применяете таблицы. И делаете это, конечно, потому, что таблицы позволяют создавать врезки и настраивать сложную структуру всего представления страницы. Не исключено применение таблиц и в более простых случаях, таких как размещение текста в цветном блоке с рамкой. Хотя, если подумать, такую простую задачу можно решить и без помощи таблиц. Если вам нужен абзац с красной рамкой и желтым фоном, не проще ли его просто создать, а не устраивать канитель с помещением его в таблицу, состоящую из единственной ячейки?

Создатели CSS чувствовали, что это и на самом деле должно быть проще, поэтому уделили много внимания тому, чтобы обеспечить вам возможность определять рамки для абзацев, заголовков, элементов `div`, ссылок, изображений – практически для всего, что может содержать веб-страница. Эти рамки могут отделять элемент от остальных, выделять его, отмечать определенный тип данных при их изменении и многое другое.

CSS также позволяет определять области вокруг элемента, которые управляют размещением рамки относительно содержимого и тем, насколько близко к ней могут находиться другие элементы. Между содержимым элемента и его рамкой мы находим *отступы* (*padding*) элемента, а за рамкой – *поля* (*margins*). Эти свойства, конечно, оказывают влияние на компоновку всего документа, но что еще важнее – они очень сильно влияют на представление конкретного элемента.

Основные блоки элементов

В главе 7 говорилось, что все элементы документа генерируют прямоугольный блок, называемый *блоком элемента* (*element box*), который

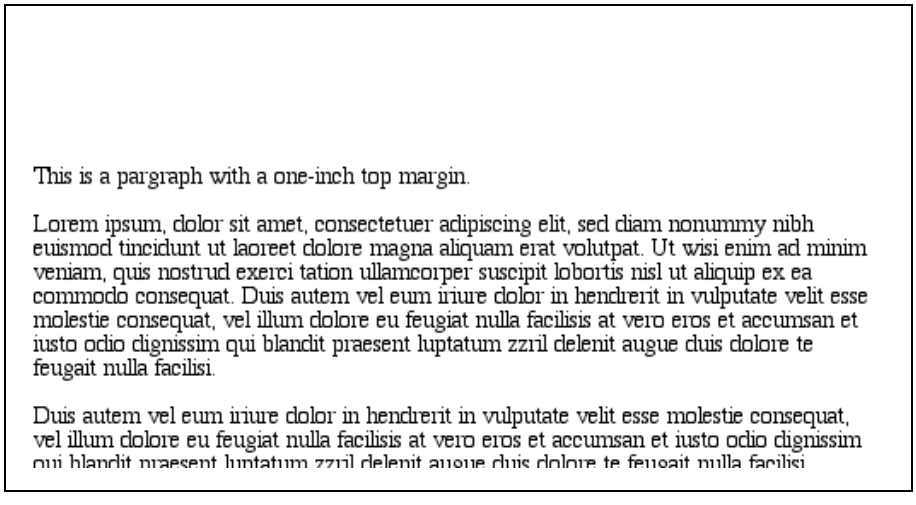


Рис. 8.1. *Как один элемент влияет на все остальные*

описывает пространство, занимаемое элементом в компоновке документа. Следовательно, каждый блок влияет на расположение и размер блоков других элементов. Например, если высота блока первого элемента документа – один дюйм, следующий блок будет начинаться как минимум на дюйм ниже верха документа. Если высота блока первого элемента увеличивается до двух дюймов, то каждый последующий блок элемента будет сдвинут на дюйм вниз, а блок второго элемента будет начинаться как минимум на два дюйма ниже верха документа, как показано на рис. 8.1.

По умолчанию визуальное представление документа составлено из ряда прямоугольных блоков, не перекрывающих друг друга. Кроме того, при определенных ограничениях эти блоки занимают минимально возможное пространство, обеспечивая при этом достаточное разделение, чтобы было понятно, какое содержимое к какому элементу относится.

Блоки могут перекрываться, если они были позиционированы вручную; также визуальное наложение может иметь место, если в элементах, размещенных в нормальном потоке, использованы отрицательные поля.

Чтобы полностью разобраться с полями, отступами и рамками, необходимо четко понимать модель блоков (также рассматриваемую в главе 7). Для справки я включу схему модели блоков из главы 7 (рис. 8.2).

Ширина и высота

Как показано на рис. 8.2, ширина элемента – это расстояние от его левого внутреннего края до правого внутреннего края, а высота – расстояние от внутреннего верхнего края до внутреннего нижнего края.

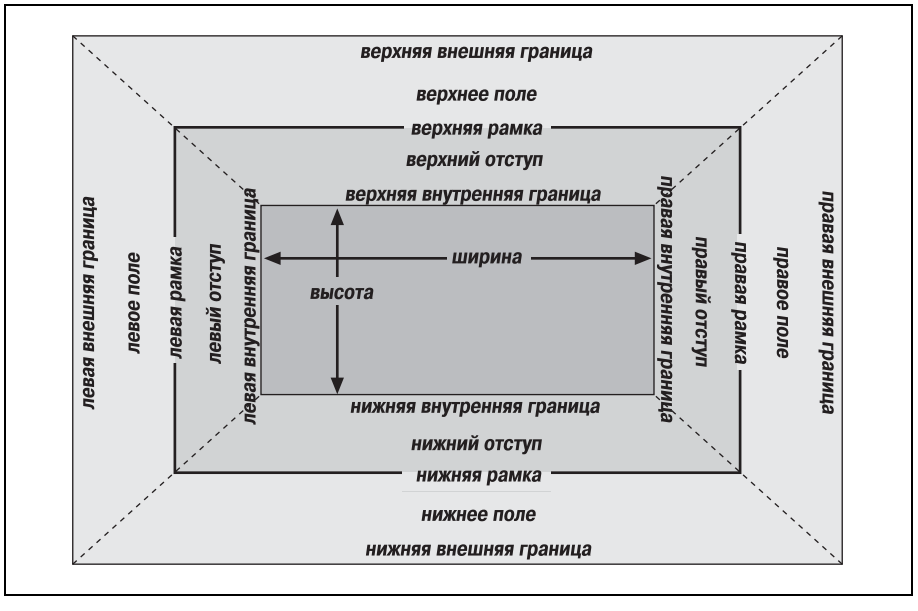


Рис. 8.2. *Модель блоков CSS*

Относительно этих двух свойств важно помнить, что они не применяются к строковым незамещаемым элементам. Например, если вы пытаетесь объявить свойства `height` и `width` для ссылки, совместимые с CSS броузеры должны будут проигнорировать эти объявления. Предположим, что применяются следующие правила:

```
a:link {color: red; background: silver; height: 15px; width: 60px;}
```

В результате получатся красные ссылки на серебряном фоне, высота и ширина которых определяется их содержимым. Их высота и ширина *не будут* равны 15 и 60 пикселам соответственно.

width

Значения:	<длина> | <процентное значение> | `auto` | `inherit`
Начальное значение:	`auto`
Область применения:	блочные и замещаемые элементы
Наследование:	нет
Процентное соотношение:	относительно ширины блока-контейнера
Вычисляемое значение:	для значений `auto` и процентных значений – как задано; в противном случае – абсолютная длина, если только свойство не применяется к элементу (тогда `auto`)

height

Значения:	<длина> | auto | inherit
Начальное значение:	auto
Область применения:	блочные и замещаемые элементы
Наследование:	нет
Процентное соотношение:	вычисляется относительно высоты блока-контейнера
Вычисляемое значение:	для значений auto и процентных значений – как задано; в противном случае – абсолютная длина, если только свойство не применяется к элементу (тогда auto)

Далее в этой главе предполагается, что высота элемента всегда вычисляется автоматически. Если элемент состоит из восьми строк и высота каждой строки составляет восьмую часть дюйма, то высота всего элемента будет равна одному дюйму. Если он включает 10 строк, его высота – 1,25 дюйма. В любом случае высота определяется содержимым элемента, а не автором. В нормальном потоке высота элемента задается крайне редко.

Исторически сложившиеся проблемы

До версии 6 обработка свойств width и height в Internet Explorer для Windows не соответствовала требованиям CSS. Два основных отличия:

- IE/Win определял размеры видимого блока элементов по значениям свойств width и height, а не по его содержимому. Если бы для свойства width элемента было задано значение 400px, IE/Win применил бы эту величину для определения расстояния от левого внешнего края рамки до правого внешнего края рамки. Иначе говоря, IE/Win использовал бы width для описания всей области содержимого элемента, левого и правого отступов и левой и правой рамки. CSS3 включает предложения о том, чтобы предоставить автору возможность выбирать, что именно означают width и height.
- IE/Win применял свойства width и height к строковым незамещаемым элементам. Например, если бы вы применили width и height к гиперссылке, она была бы отрисована согласно предоставленным значениям.

Все это было исправлено в IE6, но только для режима совместимости со стандартом. Если IE6 генерирует визуальное представление документа в режиме обратной совместимости, он ведет себя так, как описано выше.

Сравнение полей и отступов

Блоки оставляют между элементами лишь небольшие промежутки. Существует три способа сгенерировать вокруг элементов дополнительное пространство: добавить отступы, поля или комбинацию отступов и полей. Что будет выбрано в конкретных обстоятельствах, фактически не имеет значения. Однако если у элемента есть фон, этот выбор уже сделан за вас, потому что фон будет распространяться на отступы, но не на поля.

Таким образом, размер отступов и полей, назначаемый данному элементу, будет определять, где заканчивается его фон. Если задать фоновые цвета для элементов, как показано на рис. 8.3, разница станет очевидной. В элементах с отступами фон занимает большую площадь, чем там, где заданы поля.

И последнее: выбор способа задания полей и отступов остается за автором, который должен соразмерить различные возможности с намеченными эффектами и выбрать лучшую альтернативу. Чтобы суметь сделать этот выбор, конечно, не помешает знать, какие свойства можно для этого использовать.

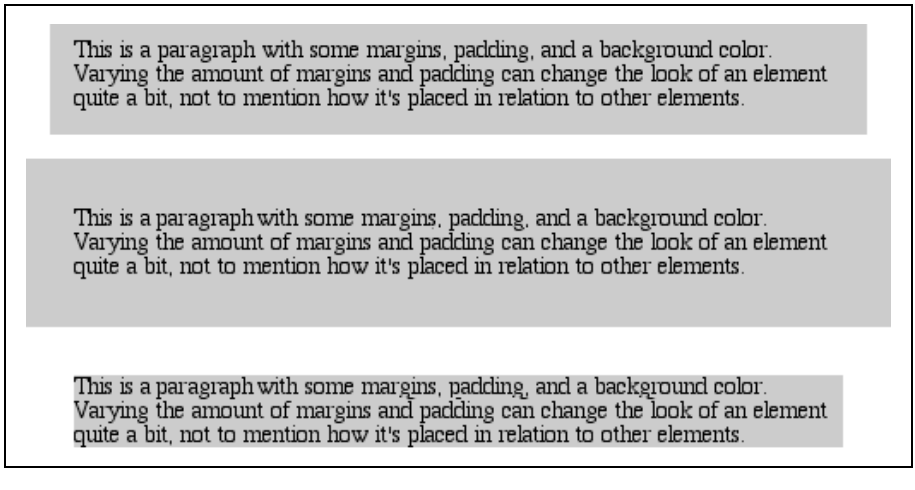


Рис. 8.3. *Абзацы с разными отступами и полями (фон добавлен для иллюстрации разницы)*

Поля

Разделение большинства элементов в нормальном потоке происходит за счет полей, призванных создавать вокруг элемента дополнительное пустое пространство. Под «пустым пространством» обычно подразумевается область, в которой не может быть других элементов и просматривается фон родительского элемента. Так, на рис. 8.4 показана раз-

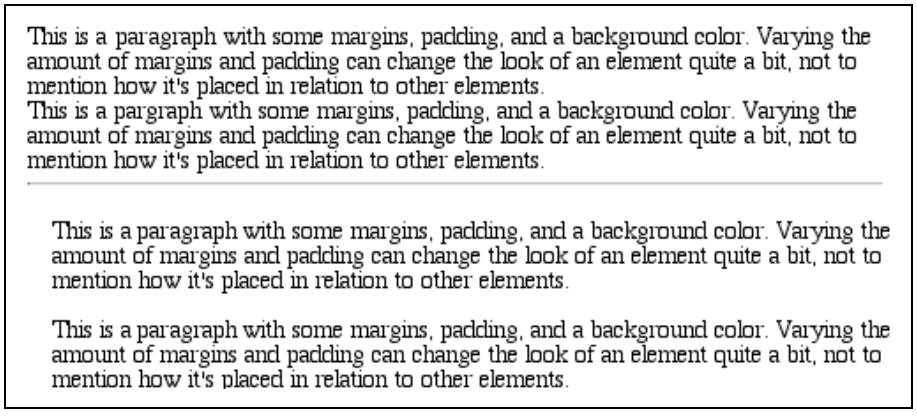


Рис. 8.4. *Абзацы с полями и без них*

ница между двумя абзацами без каких-либо полей и теми же абзацами с заданными полями.

Самый простой способ задать поле – обратиться к свойству `margin`.

Значение `auto` подробно обсуждалось в главе 7, поэтому не будем повторяться. Кроме того, обычно поля задаются в единицах длины. Предположим, для элементов `h1` требуется задать поле в четверть дюйма, как показано на рис. 8.5. (Фоновый цвет добавлен, чтобы края области содержимого были хорошо видны.)

```
h1 {margin: 0.25in; background-color: silver;}
```

В результате с каждой стороны элемента `h1` создается пустое пространство в четверть дюйма. На рис. 8.5 пустое пространство обозначено пунктирными линиями, но они добавлены в иллюстративных целях, а в веб-броузере их не будет.

Свойство `margin` принимает любые единицы измерения длины: пикселы, дюймы, миллиметры или «эмы». Однако стандартным значением `margin` является `0` (нуль), так что если значение не задано, поля не будет.

margin

Значения:	[<длина> | <процентное значение> | `auto`] {1,4} | `inherit`
Начальное значение:	не определено
Область применения:	все элементы
Наследование:	нет
Процентное соотношение:	относительно ширины блока-контейнера
Вычисляемое значение:	см. отдельные свойства

***Рис. 8.5.** Задание поля для элементов h1*

Однако на практике для многих элементов в броузере есть встроенные стили, применяемые по умолчанию, и в этих стилях могут быть заданы поля. Например, в совместимых с CSS броузерах с помощью полей создается «пустая строка» над и под каждым абзацем. Поэтому, если поля для элемента `p` не объявлены, броузер может применить собственные значения. Но, конечно, любые задаваемые вами значения будут переопределять применяемые по умолчанию стили.

И наконец, для задания свойства `margin` можно применять процентные значения. Связанные с ними особенности будут обсуждаться позже в разделе «Процентные значения и поля».

Абсолютные значения и поля

Как утверждалось ранее, при задании полей элемента могут использоваться значения, выраженные в любых единицах измерения длины. Это достаточно просто: например, создадим вокруг абзацев свободное место в 10 пикселов. Следующее правило обеспечивает абзацам серебряный фон и поле в 10 пикселов, как показано на рис. 8.6:

```
p {background-color: silver; margin: 10px;}
```

(Как и раньше, фоновый цвет помогает показать область содержимого, а пунктирные линии приведены только для иллюстративных целей.) Как показывает рис. 8.6, с каждой стороны области содержимого было добавлено пространство в 10 пикселов. Результат отчасти похож на применение атрибутов `hspace` и `vspace` в HTML. Кстати, `margin` позволяет за-

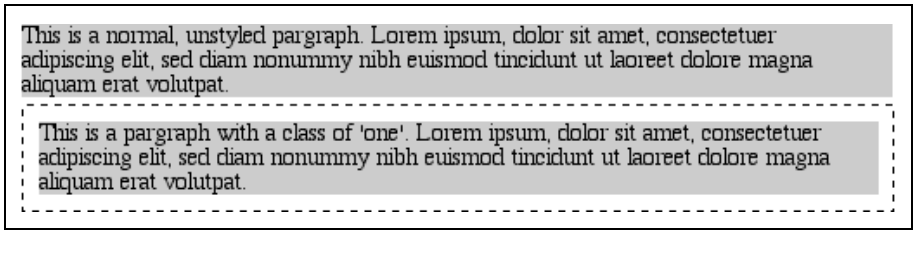


***Рис. 8.6.** Абзацы для сравнения*

давать дополнительные поля вокруг изображения. Скажем, требуется, чтобы все изображения были окружены областью шириной один em:

```
img {margin: 1em;}
```

Вот и все.

Иногда, впрочем, может потребоваться, чтобы поля с каждой стороны элемента были разными. Это так же просто. В следующем примере верхнее поле всех элементов h1 равно 10 пикселов, правое поле – 20 пикселов, поле снизу – 15 пикселов и поле слева – 5 пикселов, вот так:

```
h1 {margin: 10px 20px 15px 5px;}
```

Порядок значений важен и соответствует следующему шаблону:

```
margin: top right bottom left
```

Чтобы было проще запомнить шаблон, обратите внимание, что эти четыре значения перечисляются по ходу часовой стрелки, начиная сверху. Значения *всегда* применяются в этом порядке.

Еще один способ запоминания порядка объявления сторон состоит в том, чтобы помнить, что расстановка сторон в правильном порядке помогает избежать неприятностей (в английском написании – «TRouBLe», т. е. TRBL: Top (верхнее) Right (правое) Bottom (нижнее) Left (левое)).

Допускается также применение разных единиц измерения длины:

```
h2 {margin: 14px 5em 0.1in 3ex;}  /* разнообразие единиц измерения! */
```

На рис. 8.7 показаны результаты применения этого объявления с небольшими дополнительными пояснениями.

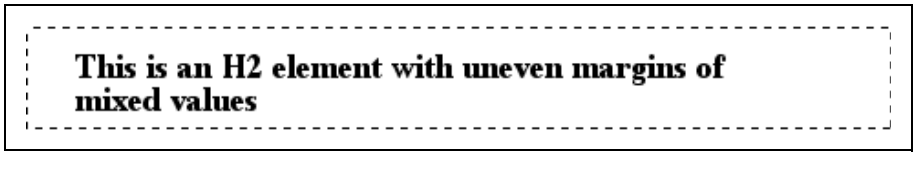


Рис. 8.7. *Поля, заданные с применением различных единиц измерения длины*

Процентные значения и поля

Как уже говорилось, значения полей элемента можно задавать в процентах. Процентные значения вычисляются относительно ширины родительского элемента, следовательно, они меняются, если каким-либо образом изменяется ширина родительского элемента. Рассмотрим следующую разметку, действие которой проиллюстрировано на рис. 8.8:

```
p {margin: 10%;}

<div style="width: 200px; border: 1px dotted;">
<p>This paragraph is contained within a DIV that has a width of 200 pixels,
```

```
so its margin will be 10% of the width of the paragraph's parent (the DIV).
Given the declared width of 200 pixels, the margin will be 20 pixels on all
sides.</p>
</div>
<div style="width: 100px;  border: 1px dotted;">
<p>This paragraph is contained within a DIV with a width of 100 pixels, so
its margin will still be 10% of the width of the paragraph's parent. There
will, therefore, be half as much margin on this paragraph as that on the
first paragraph.</p>
</div>
```

Для сравнения рассмотрим случай, когда значение `width` элементов не объявлено. В таких ситуациях общая ширина блока элемента (включая поля) зависит от значения свойства `width` родительского элемента. Это может приводить к появлению «нестабильных» страниц, в которых поля элементов подгоняются под фактический размер элемента-родителя (или области отображения). В документе, оформленном таким образом, элементы используют поля, заданные процентными зна-

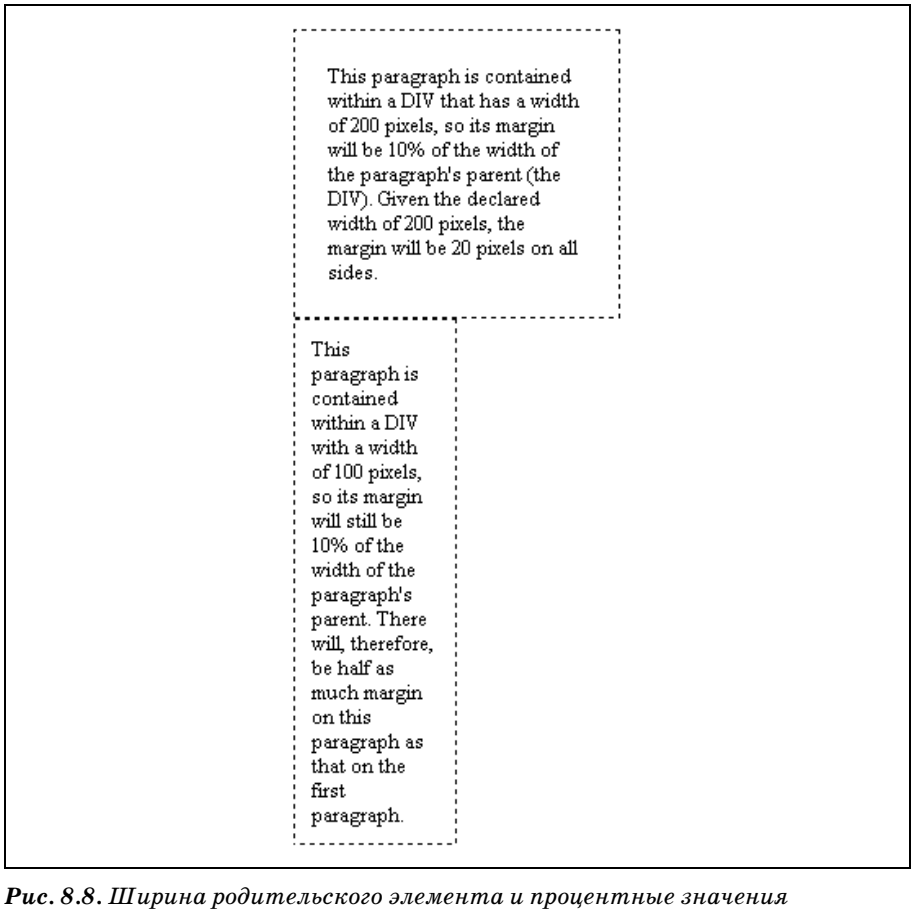


Рис. 8.8. *Ширина родительского элемента и процентные значения*

чениями, и при изменении ширины окна броузера пользователем поля будут расширяться или сужаться. Конструктивное решение этой проблемы зависит от вас.

Вы могли обратить внимание на некоторую странность в поведении абзацев, представленных на рис. 8.8. В соответствии с изменениями ширины родительских элементов меняются не только боковые поля, но и поля сверху и снизу. Такого поведения требует CSS. Вернемся к объявлению свойства, и вы увидите, что процентные значения определены относительно *ширины* родительского элемента. Это также относится и к верхнему и нижнему полям. Таким образом, исходя из следующих стилей и разметки высота верхнего поля абзаца составит `50px`:

```
div p {margin-top: 10%;}

<div style="width: 500px;">
<p>Это параграф, и его верхнее поле составляет 10% ширины
его элемента-родителя.</p>
</div>
```

Если значение `width` элемента `div` изменится, изменится также и верхнее поле. Кажется странным? Считаем, что высота большинства элементов в нормальном потоке (как мы предполагаем) такова, чтобы вместить их элементов-потомков, включая поля. Если верхнее и нижнее поля элемента задать как процент от высоты родителя, это может привести к бесконечному циклу: высота родителя увеличивалась бы, чтобы вместить верхнее и нижнее поля, которые затем должны были бы увеличиться соответственно новой высоте и т. д. Вместо того чтобы просто игнорировать процентные значения для верхнего и нижнего полей, авторы спецификации решили соотнести их с шириной родительского элемента, которая не меняется в зависимости от ширины его потомков.

Обработка процентных значений для верхнего и нижнего полей абсолютно позиционированных элементов отличается и более подробно рассмотрена в главе 10.

Процентные значения можно сочетать с единицами длины. Таким образом, чтобы задать для элементов `h1` верхнее и нижнее поля высотой 0,5 em и боковые поля, составляющие 10% ширины окна броузера, можно объявить следующее (рис. 8.9):

```
h1 {margin: 0.5em 10% 0.5em 10%;}
```

Рис. 8.9. *Поля, заданные с применением различных единиц измерений*

Здесь верхнее и нижнее поля будут оставаться постоянными всегда, а боковые поля будут меняться в зависимости от ширины окна броузера. Конечно, предполагается, что `h1` являются дочерними элементами `body` и что ширина `body` равна ширине окна броузера. Как указано, боковые поля элементов `h1` будут составлять 10% ширины родителя элемента `h1`.

Давайте на мгновение вернемся к этому правилу:

```
h1 {margin: 0.5em 10% 0.5em 10%;}
```

Выглядит несколько избыточным, не так ли? Ведь приходится набирать одни и те же пары значений дважды. К счастью, CSS предлагает простой способ избежать этого.

Тиражирование значений

Иногда значения, вводимые для свойства `margin`, повторяются:

```
p {margin: 0.25em 1em 0.25em 1em;}
```

Хотя не обязательно вот так набирать пары чисел. Вот возможная замена предыдущему правилу:

```
p {margin: 0.25em 1em;}
```

Этих двух значений достаточно, чтобы заменить четыре. Но как? CSS определяет несколько правил в случае присваивания свойству `margin` менее четырех значений. Вот они:

- Если значение поля `left` пропущено, взять значение, заданное для `right`.
- Если значение поля `bottom` пропущено, взять значение, заданное для `top`.
- Если значение поля `right` пропущено, взять значение, заданное для `top`.

Если вы предпочитаете более наглядный подход, взгляните на схему, приведенную на рис. 8.10.

Иначе говоря, если для `margin` заданы три значения, то четвертое (`left`) копирует второе (`right`). Если заданы два значения, то четвертое копирует второе, а третье (`bottom`) копирует первое (`top`). И наконец, если задано только одно значение, остальные копируют его.

Этот простой механизм позволяет авторам предоставлять лишь необходимое количество значений, как показано здесь:

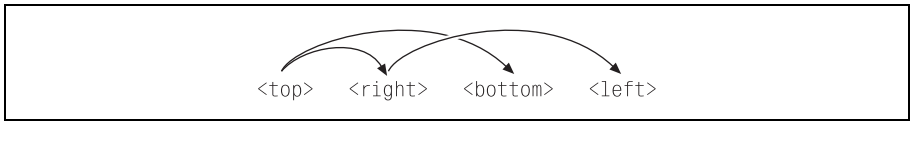


Рис. 8.10. *Шаблон тиражирования значений*

```
h1 {margin: 0.25em 0 0.5em;} /* аналогично '0.25em 0 0.5em 0' */
h2 {margin: 0.15em 0.2em;}  /* аналогично '0.15em 0.2em 0.15em 0.2em' */
p {margin: 0.5em 10px;}     /* аналогично '0.5em 10px 0.5em 10px' */
p.close {margin: 0.1em;}    /* аналогично '0.1em 0.1em 0.1em 0.1em' */
```

У этого метода есть небольшой недостаток, с которым вы когда-нибудь столкнетесь. Предположим, требуется задать для элементов `h1` верхнее и левое поля в 10 пикселов и нижнее и правое поля в 20 пикселов. В этом случае надо записать:

```
h1 {margin: 10px 20px 20px 10px;} /* никак нельзя сократить */
```

Результат достигается, но это требует некоторого времени. К сожалению, количество значений, необходимых в таких обстоятельствах, никак нельзя уменьшить. Возьмем другой пример: пусть требуется, чтобы размер всех полей вычислялся автоматически, за исключением левого поля, которое должно составлять `3em`:

```
h2 {margin: auto auto auto 3em;}
```

Опять же вы добились желаемого эффекта. Проблема в том, что набирать слово `auto` несколько утомительно. Хотелось бы задать поле лишь с одной стороны, что подводит нас к следующей теме.

Свойства для задания поля с одной стороны

К счастью, существует способ задать поле только с одной стороны элемента. Скажем, надо задать элементам `h2` поле в `3em` только слева. Вместо свойства `margin`, требующего набора большого количества символов, можно выбрать такой подход:

```
h2 {margin-left: 3em;}
```

Свойство `margin-left` – одно из четырех свойств, предназначенных для задания полей с одной из четырех сторон блока элемента. Их имена могут стать небольшим сюрпризом.

Применение любого из этих свойств позволяет задавать поле только с одной стороны, совершенно не влияя на все остальные поля.

margin-top, margin-right, margin-bottom, margin-left

Значения:	<длина> | <процентное значение> | `auto` | `inherit`
Начальное значение:	`0`
Область применения:	все элементы
Наследование:	нет
Процентное соотношение:	относительно ширины блока-контейнера
Вычисляемое значение:	для процентных значений – как задано; в противном случае – абсолютная длина

***Рис. 8.11.** Применение различных односторонних полей*

В одном правиле можно задать несколько этих свойств, например:

```
h2 {margin-left: 3em; margin-bottom: 2em;
  margin-right: 0; margin-top: 0;
  background: silver;}
```

Как видно из рис. 8.11, поля заданы так, как вы хотели. Конечно, в этом случае, наверное, было бы проще обратиться к свойству `margin`:

```
h2 {margin: 0 0 2em 3em;}
```

Независимо от того, что вы предпочтете – односторонние свойства или сокращенную форму записи, – результат будет одним и тем же. В общем, если надо задать поля с нескольких сторон, лучше применить свойство `margin`. Однако с точки зрения представления документа на самом деле не важно, какой выбран подход, так что остановитесь на том, что кажется вам проще.

Отрицательные и свернутые поля

Как подробно обсуждалось в главе 7, существует возможность задавать элементу отрицательные поля. Это может привести к тому, что блок элемента будет выходить за рамки его родителя или перекрывать другие элементы без нарушения модели блоков. Рассмотрим следующие правила, которые проиллюстрированы на рис. 8.12:

```
div {border: 1px dotted gray; margin: 1em;}
p {margin: 1em; border: 1px dashed silver;}
p.one {margin:  0 -1em;}
p.two {margin: -1em 0;}
```

В первом случае вычисляемое значение свойства `width` абзаца плюс его правое и левое поля точно равны ширине элемента-родителя `div`. Итак, абзац получается на два эма шире, чем родительский элемент, хотя в действительности (с математической точки зрения) он не шире. Во втором случае отрицательные верхнее и нижнее поля уменьшают вычисляемую высоту элемента и сдвигают его внешние верхний и нижний края внутрь, вот почему в результате этот абзац перекрывает близлежащие.

Комбинирование отрицательных и положительных полей на самом деле очень полезно. Например, творчески подойдя к применению положительных и отрицательных полей, можно сдвинуть абзац относи-

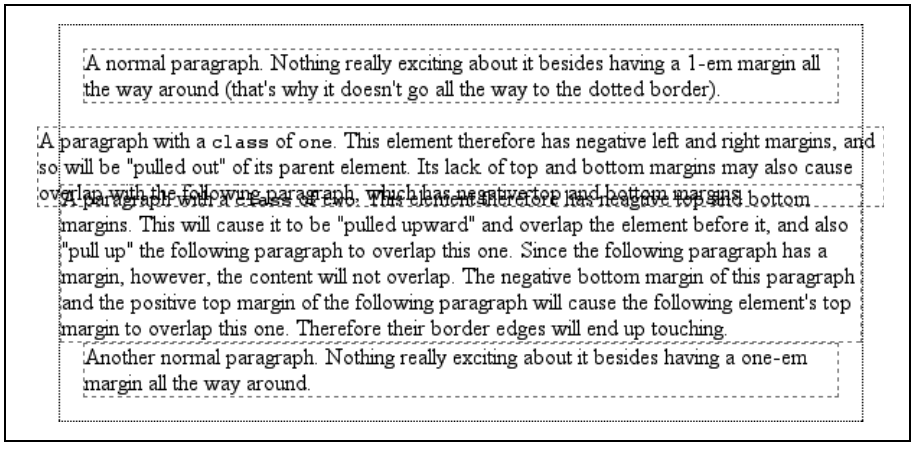


Рис. 8.12. *Отрицательные поля в действии*

тельно родительского элемента или создать нечто похожее на картины Мондриана, с несколькими наложениями и случайно расположенными блоками, как показано на рис. 8.13:

```
div {background: silver; border: 1px solid;}
p {margin: 1em;}
p.punch {background: white; margin: 1em -1px 1em 25%;
  border: 1px solid; border-right: none; text-align: center;}
p.mond {background: #333; color: white; margin: 1em 3em -3em -3em;}
```

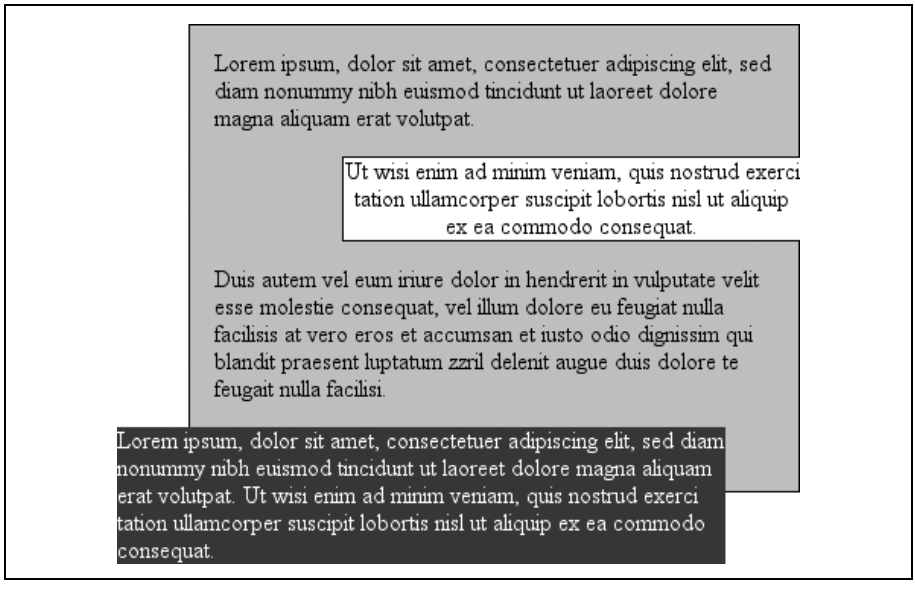


Рис. 8.13. *Выход за рамки родительского элемента*

Благодаря отрицательному нижнему полю сдвинутого абзаца, низ его родительского элемента поднят вверх, а параграф выступает за нижний край своего родителя.

Говоря о верхних и нижних полях, важно также помнить, что смежные вертикальные поля в нормальном потоке будут сворачиваться; это мы рассматривали в предыдущей главе. Сворачивание полей встречается практически в каждом документе. Например, вот простое правило:

```
p {margin: 15px 0;}
```

В результате абзацы будут разделены полем в 15 пикселов. Если бы поля не сворачивались, промежуток между двумя смежными абзацами составлял бы 30 пикселов, но такое поведение не соответствовало бы ожиданиям авторов.

Однако это означает, что при стилевом оформлении полей необходимо соблюдать аккуратность. Скорее всего, потребуется убрать промежуток между заголовком и следующим абзацем. Поскольку абзацы в HTML-документах имеют верхнее поле, недостаточно просто задать нулевое нижнее поле для заголовка; надо убрать верхнее поле абзаца. С помощью сестринского селектора CSS2 сделать это очень просто:

```
h2 {margin-bottom: 0;}
h2 + p {margin-top: 0;}
```

К несчастью, поддержка сестринских селекторов броузерами довольно ограниченна (на момент написания данной книги), и большинство пользователей увидят промежуток в `1em` между заголовком и следующим абзацем. Желаемого эффекта можно добиться и без применения селекторов CSS2, но придется проявить смекалку:

```
h2 {margin-bottom: 0;}
p {margin: 0 0 1em;}
```

Верхнее поле над всеми абзацами действительно исчезнет, но, поскольку имеются также и нижние поля размером `1em`, желаемое разделение абзацев сохранится, как показано на рис. 8.14.

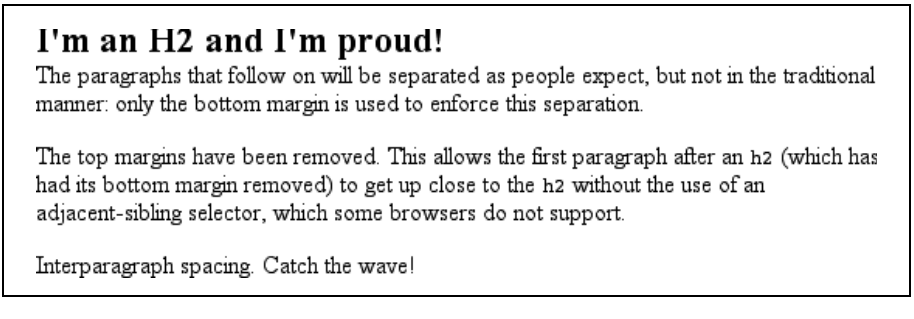


Рис. 8.14. *Разумное задание полей*

Так получается потому, что обычный промежуток между абзацами в 1em представляет собой результат сворачивания полей. Таким образом, если убрать одно из этих полей – в данном случае верхнее – визуальный результат будет таким же, как если бы вы оставили поле на месте.

Поля и строковые элементы

Поля могут применяться и в строковых элементах, хотя последствия будут несколько иными. Скажем, требуется задать верхнее и нижнее поля для особо выделяемого текста:

```
strong {margin-top: 25px; margin-bottom: 50px;}
```

Это разрешено в спецификации, но поскольку поля применяются к строковому незамещаемому элементу, объявление абсолютно не повлияет на высоту строки. Поскольку поля действительно прозрачные, оно вообще не будет иметь никакого визуального эффекта. Дело в том, что поля строковых незамещаемых элементов не меняют высоты строки элемента.

Единственными свойствами, которые могут изменить расстояние между содержащими только текст строками, являются line-height, font-size и vertical-align, как описывалось в главе 7.

Вышесказанное относится только к верхней и нижней сторонам строковых незамещаемых элементов; левая и правая стороны – совсем другое дело. Для начала рассмотрим простой пример небольшого строкового незамещаемого элемента в рамках единственной строки. Если задать значения левого и правого полей, они будут видимыми, что очевидно на рис. 8.15:

```
strong {margin-left: 25px; background: silver;}
```

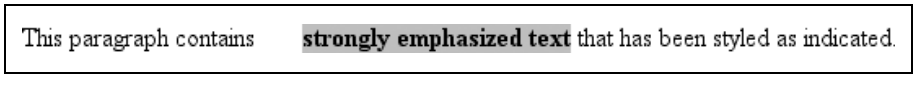


***Рис. 8.15.** Строковый незамещаемый элемент с полем слева*

Обратите внимание на дополнительное пространство между концом слова прямо перед строковым незамещаемым элементом и краем фона строкового элемента. Можно добавить дополнительный промежуток с обоих концов строкового элемента:

```
strong {margin: 25px; background: silver;}
```

Как и ожидалось, на рис. 8.16 показан небольшой дополнительный промежуток справа и слева от строкового элемента, а сверху и снизу никаких полей нет.

Когда строковый незамещаемый элемент простирается на несколько строк, ситуация немного меняется. На рис. 8.17 видно, что происхо-

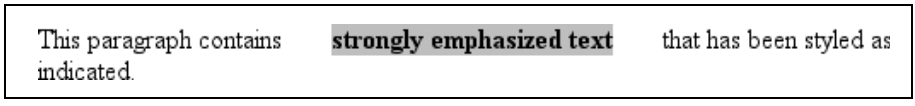


Рис. 8.16. *Строковый незамещаемый элемент с полем в 25 пикселов*

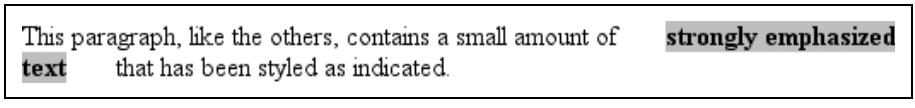


Рис. 8.17. *Строковый незамещаемый элемент с полем в 25 пикселов, отображаемый в нескольких строках текста*

дит, когда строковый незамещаемый элемент, имеющий поле, отображается в нескольких строках:

```
strong {margin: 25px; background: silver;}
```

Левое поле применяется к началу элемента, а правое поле – к его концу. Поля *не применяются* к левой и правой границам каждой строки. Кроме того, видно, что если бы не поля, разрыв строки был бы размещен после слова «text», а не после «strongly emphasized». Поля влияют на место разрыва строки только тем, что меняют точку начала содержимого элемента в строке.

Все становится еще интереснее, если задать для строковых незамещаемых элементов отрицательные поля. Верх и низ элемента не изменяются, это относится и к высоте строк, но левый и правый концы элемента могут перекрывать остальное содержимое, как показано на рис. 8.18:

```
strong {margin: -25px; background: silver;}
```

Совсем другая история с замещаемыми элементами: заданные для них поля *влияют* на высоту строки, увеличивая или уменьшая ее в зависимости от величины верхнего и нижнего полей. Левое и правое поля

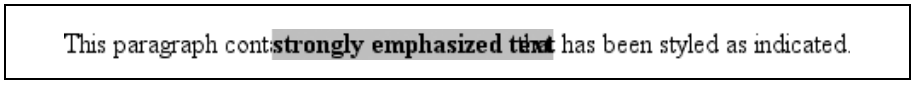


Рис. 8.18. *Строковый незамещаемый элемент с отрицательным полем*

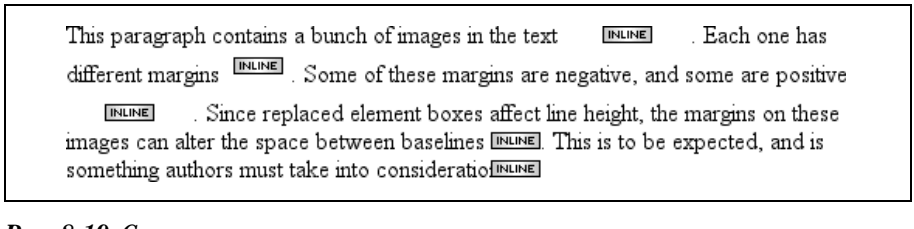


Рис. 8.19. *Строковые замещаемые элементы с разными полями*

Исторически сложившиеся проблемы, связанные с использованием полей

Как бы ни были полезны поля, с их применением связан ряд проблем, причем все они концентрируются вокруг Netscape Navigator 4.x (NN4.x), что и не удивительно.

Первая заминка: в Navigator 4.x заданные правилами значения полей добавляются к применяемым по умолчанию полям, а не заменяют их стандартные значения. Например:

```
h1 {margin-bottom: 0;}
p {margin-top: 0;}
```

NN4.x отображает элементы с обычными промежутками между ними, потому что нулевые значения добавляются к его собственным применяемым по умолчанию полям. От этого пробела всегда можно избавиться, задав отрицательные поля, например верхнее поле `-1em` для абзаца. Проблема в том, что это решение не универсально. В броузерах, совместимых с CSS, будет происходить наложение текста, поскольку верхнее поле абзаца удалено.

К сожалению, дело принимает еще худший оборот. Если поля применяются к строковым элементам, верстка в большей или меньшей степени будет нарушена. NN4.x предполагает, что поле любого элемента, строкового или нет, относится к левому краю окна броузера. Это совершенно неправильно. Если сайт будет посещать большое количество пользователей NN4.x, применение полей в строковых элементах окажется очень рискованным, и это не тот случай, когда можно быть беспечным. Но поскольку не составит труда спрятать CSS от NN4.x, то можно не беспокоиться, что NN4.x все испортит, особенно если помимо этого обеспечить представление этих страниц и для NN4.x.

строкового замещаемого элемента ведут себя так же, как и в незамещаемом элементе. На рис. 8.19 показан ряд различных эффектов от применения полей к строковым замещаемым элементам.

Рамки

Внутри полей элемента находятся его *рамки* (*borders*). Рамка элемента – это одна или несколько линий, окружающих содержимое и отступы элемента. Таким образом, фон элемента будет ограничен внешним краем рамки, поскольку фон не распространяется на поля, а рамки как раз граничат с внутренней стороны полей.

Каждая рамка характеризуется тремя параметрами: ее шириной (толщиной), стилем (представлением) и цветом. Ширина рамки по умолча-

нию имеет значение medium, которое не определено явно, но обычно составляет два пиксела. Несмотря на это, причина, по которой рамки обычно не видны, в том, что применяемый по умолчанию стиль – none, что обусловливает их отсутствие. Если стиль рамки не задан, нет смысла в ее существовании. (При этом также может быть переопределено значение ширины, но к этому мы еще вернемся чуть позже.)

И наконец, применяемый по умолчанию цвет рамки – это основной цвет самого элемента. Если для рамки цвет не задан, он будет таким же, как и цвет текста элемента. Если, с другой стороны, в элементе нет текста, скажем он содержит только таблицу с изображениями, цвет рамки для этой таблицы будет таким же, как цвет текста родительского элемента (благодаря тому факту, что цвет наследуется). Этим элементом может быть body, div или другой table. Таким образом, если у элемента table есть рамка и его родителем является body, то исходя из правила:

```
body {color: purple;}
```

по умолчанию рамка вокруг table будет фиолетовой (если агент пользователя не задает цвета таблиц). Конечно, чтобы эта рамка появилась, придется сначала немного потрудиться.

Рамки и фон

В спецификации CSS совершенно определенно подразумевается, что фон элемента распространяется до внешнего края рамки, поскольку упоминается, что рамки отрисовываются «поверх фона элемента». Это важно, потому что некоторые рамки «прерывистые», например точечные или пунктирные, и фон элемента должен просматриваться между видимыми частями рамки.

Когда была опубликована спецификация CSS2, в ней говорилось, что фон распространяется только на отступы, но не на рамки. Позже это положение исправили, и в CSS2.1 прямо сказано, что фон элемента – это фон областей содержимого, отступов и рамок. Большинство броузеров согласуются с определением CSS2.1, хотя некоторые более старые, возможно, ведут себя по-другому. Вопросы, связанные с применением фоновых цветов, более подробно обсуждаются в главе 9.

Рамки, имеющие стиль

Начнем со стилей рамок – самого важного их параметра – не потому, что они управляют представлением рамки (хотя, конечно, они это делают), но потому, что без стиля рамки вообще не будут отображены.

CSS определяет 10 отдельных отличных от inherit стилей для свойства border-style, включая применяемое по умолчанию значение none. Эти стили представлены на рис. 8.20.

Значение hidden эквивалентно значению none, за исключением случая применения к таблицам, когда оно призвано разрешить конфликт рамок. (Более подробно об этом говорится в главе 11.)

border-style

Значения:	`[ none | hidden | dotted | dashed | solid | double | groove | ridge | inset | outset ]{1,4} | inherit`
Начальное значение:	для сокращенной формы записи свойств не определено
Область применения:	все элементы
Наследование:	нет
Вычисляемое значение:	смотрите отдельные свойства (`border-top-style` и др.)
Примечание:	Согласно CSS1 и CSS2 от агентов пользователя HTML требуется поддерживать только значения `solid` и `none`; остальные значения (кроме `hidden`) могут интерпретироваться как `solid`; это ограничение было опущено в более поздних вариантах CSS2.1

Самый непредсказуемый стиль рамки – `double` (двойной). Он определяется как ширина двух линий плюс пробел между ними и эквивалентен значению `border-width` (обсуждаемому в следующем разделе). Однако спецификация CSS не говорит о том, должна ли одна линия быть толще другой, или они должны быть одинаковыми, или пробел между ними должен быть уже или шире линий. Решение всех этих вопросов остается за агентом пользователя, и автор не может влиять на этот процесс.

Значение свойства `color` всех рамок, показанных на рис. 8.20, – `gray`, благодаря чему проще увидеть все визуальные эффекты. Вид рамки все-

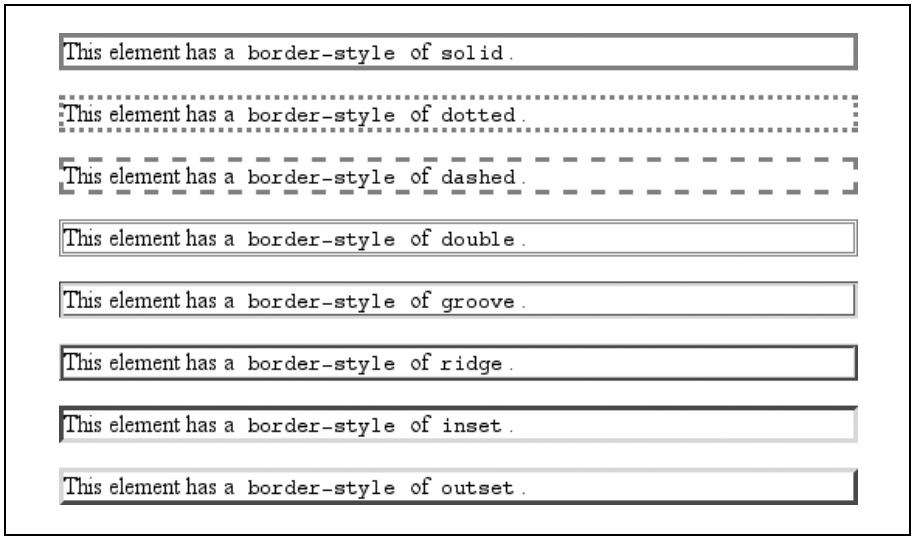


Рис. 8.20. *Стили рамок*

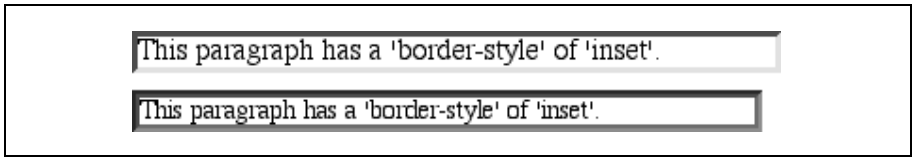


Рис. 8.21. *Два действительных способа визуального представления рамки со стилем inset*

гда зависит в некоторой мере от ее цвета, хотя представление может меняться в зависимости от агента пользователя. На рис. 8.21 показаны различные варианты визуального представления рамки со стилем `inset`.

Итак, предположим, требуется задать стиль рамки для изображений, находящихся внутри непосещенной гиперссылки. Им можно было бы присвоить значение `outset`, тогда бы они выглядели как ненажатая кнопка (рис. 8.22):

```
a:link img {border-style: outset;}
```

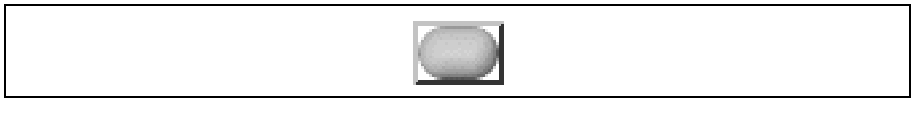


Рис. 8.22. *Применение выпуклой рамки к изображению в гиперссылке*

Цвет рамки основывается на значении свойства `color` элемента. В данных обстоятельствах, скорее всего, это будет синий (хотя этого не видно на печати), потому что изображение содержится в гиперссылке, а основной цвет гиперссылок обычно синий. При желании можно изменить этот цвет на `silver` вот так:

```
a:link img {border-style: outset; color: silver;}
```

Рамка будет светло-серой, потому что теперь это основной цвет изображения – даже несмотря на то, что в изображении он фактически отсутствует, он все равно передается рамке. Позже в этой главе мы рассмотрим другой способ изменения цвета рамки.

Несколько стилей

Для одной рамки можно задавать несколько стилей. Например:

```
p.aside {border-style: solid dashed dotted solid;}
```

В результате получаем абзац со сплошной верхней рамкой, пунктирной рамкой справа, точечной рамкой снизу и сплошной левой рамкой.

Опять мы видим порядок задания значений TRBL, так же как при задании разных значений для полей. Все те же правила, как и в случае с полями и отступами, применяются и к тиражированию значений стилей рамок. Таким образом, следующие два выражения должны иметь один и тот же эффект, что показано на рис. 8.23:

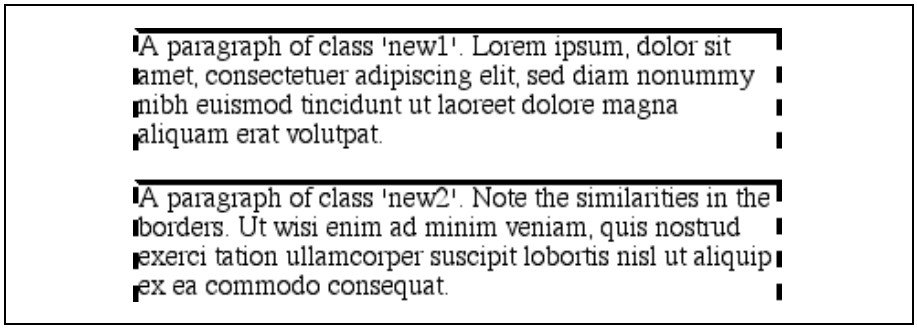


***Рис. 8.23.** Эквивалентные правила стилей*

```
p.new1 {border-style: solid dashed none;}
p.new2 {border-style: solid dashed none dashed;}
```

Свойства задания стиля для одной стороны

Иногда требуется задать стили рамок только для одной стороны блока элемента, а не для всех четырех. Вот где понадобятся свойства задания стиля рамки для одной стороны.

Свойства задания стиля рамки для одной стороны довольно наглядны. Например, для того чтобы изменить стиль нижней рамки, можно обратиться к свойству `border-bottom-style`.

Нет ничего необычного в том, чтобы применить `border` в сочетании со свойством задания стиля с одной стороны. Предположим, надо задать сплошную рамку с трех сторон заголовка, а слева рамка должна отсутствовать, как показано на рис. 8.24.

Это можно сделать двумя совершенно эквивалентными способами:

```
h1 {border-style: solid solid solid none;}
  /* приведенное выше аналогично нижнему */
h1 {border-style: solid; border-left-style: none;}
```

Важно помнить – второй подход требует поместить свойство задания стиля с одной стороны после сокращенной формы записи свойства,

border-top-style, border-right-style, border-bottom-style, border-left-style

Значения:	`none | hidden | dotted | dashed | solid | double | groove | ridge | inset | outset | inherit`
Начальное значение:	`none`
Область применения:	все элементы
Наследование:	нет
Вычисляемое значение:	как задано

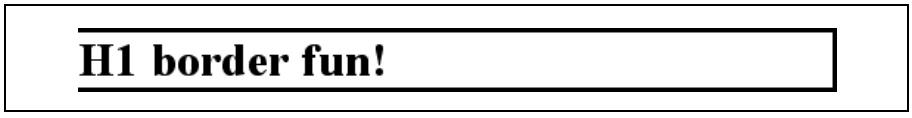


***Рис. 8.24.** Удаление рамки слева*

как это обычно делается в сокращенной форме записи свойств. Дело в том, что объявление `border-style: solid` фактически представляет собой объявление `border-style: solid solid solid solid`. Если поместить `border-style-left: none` перед объявлением `border-style`, то его значение переопределит значение `none` свойства задания стиля с одной стороны.

Вы могли также обратить внимание на то, что до сих пор рамки во всех примерах имели одну и ту же ширину. Это происходило потому, что их ширина не задавалась, поэтому подставлялось значение по умолчанию. Сейчас мы узнаем, что это за значение, и еще многое другое.

Ширина рамки

Следующий шаг после назначения стиля рамки – задание некоторой ширины с помощью свойства `border-width`.

border-width

Значения:	`[ thin | medium | thick | <длина> ]{1,4} | inherit`
Начальное значение:	не определено для сокращенной формы записи свойств
Область применения:	все элементы
Наследование:	нет
Вычисляемое значение:	смотрите отдельные свойства (`border-top-style` и др.)

Также можно обратиться к одному из родственных свойств.

border-top-width, border-right-width, border-bottom-width, border-left-width

Значения:	`thin | medium | thick | <длина> | inherit`
Начальное значение:	`medium`
Область применения:	все элементы
Наследование:	нет
Вычисляемое значение:	абсолютная длина; `0`, если стиль рамки `none` или `hidden`

Каждое из этих свойств служит для задания ширины рамки с определенной стороны, конечно, аналогично свойствам для определения полей.

В CSS2.1 ширина рамки по-прежнему не может быть задана процентным значением, что крайне досадно.

Задать ширину рамки можно четырьмя способами: присвоить ей значение в единицах измерения длины, например `4px` или `0.1em`, или посредством одного из трех ключевых слов. Эти ключевые слова: `thin`, `medium` (применяемое по умолчанию значение) и `thick`. Не обязательно, чтобы эти ключевые слова точно соответствовали какой-либо конкретной ширине, они просто определены относительно друг друга. Согласно спецификации рамка `thick` всегда шире, чем `medium`, которая, в свою очередь, всегда шире `thin`.

Тем не менее точные значения ширины не определены, поэтому один агент пользователя мог бы приравнять их значениям `5px`, `3px` и `2px`, тогда как в другом это были бы `3px`, `2px` и `1px`. Неважно, какую ширину агент пользователя ставит в соответствие каждому ключевому слову, она будет одинакова по всему документу независимо от того, где встречается рамка. Итак, если значение `medium` аналогично `2px`, ширина рамки среднего размера всегда будет равна 2 пикселам независимо от того, какой элемент она окружает – `h1` или `p`. Рисунок 8.25 иллюстрирует один из способов обработки этих трех ключевых слов, а также то, как они соотносятся друг с другом и с окружаемым ими содержимым.

Предположим, для абзаца заданы поля, цвет фона и стиль рамки:

```
p {margin: 5px; background-color: silver;
   border-style: solid;}
```

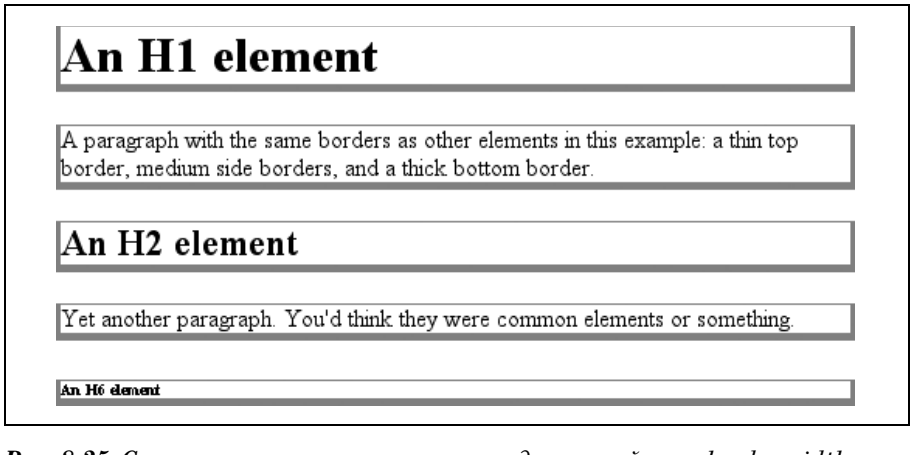


Рис. 8.25. *Сравнение трех ключевых слов задания свойства border-width*

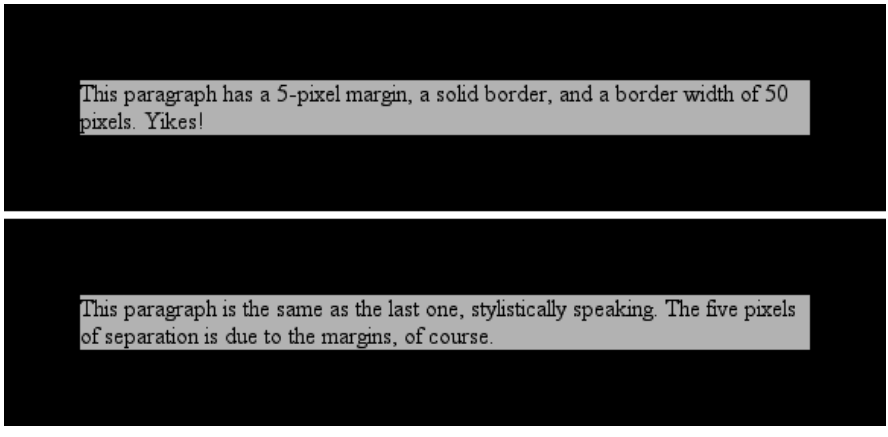


Рис. 8.26. *Действительно широкая рамка*

Ширина рамки по умолчанию равна `medium`. Изменить ее довольно просто:

```
p {margin: 5px; background-color: silver;
   border-style: solid; border-width: thick;}
```

Конечно, для ширины рамки можно задавать довольно нелепые предельные значения, как, например, на рис. 8.26, где мы видим 50-пиксельную рамку:

```
p {margin: 5px; background-color: silver;
   border-style: solid; border-width: 50px;}
```

Кроме того, посредством двух хорошо знакомых нам методов можно задавать ширину рамки для каждой стороны отдельно. Первый метод – использовать специальные свойства, о которых упоминалось в начале этого раздела, такие как `border-bottom-width`. Другой способ – обратиться к тиражированию значения в свойстве `border-width`, что показано на рис. 8.27:

```
h1 {border-style: dotted; border-width: thin 0;}
p {border-style: solid; border-width: 15px 2px 7px 4px;}
```

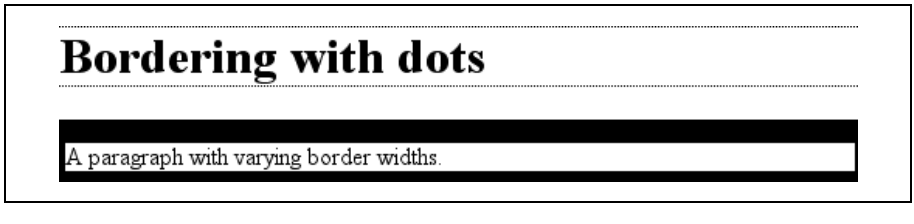


Рис. 8.27. *Тиражирование значения и рамка с непостоянной шириной*

Полное отсутствие рамки

До сих пор мы говорили только о применении видимых стилей рамок, таких как `solid` или `outset`. Теперь посмотрим, что происходит, когда свойству `border-style` присваивается значение `none`:

```
p {margin: 5px; border-style: none; border-width: 20px;}
```

Ширина рамки равна `20px`, однако для стиля задано значение `none`. В этом случае аннулируется не только стиль рамки, но и ее ширина. Рамка просто прекращает существование. Почему?

Если помните, мы уже говорили, что рамка, имеющая стиль `none`, не существует. Эти слова были выбраны с большой тщательностью, потому что они помогают объяснить происходящее. Поскольку рамки не существует, она не может иметь ширины, так что ширине автоматически присваивается значение `0` (нуль) независимо от того, что вы пытаетесь задать. В конце концов, если стакан пуст, его нельзя описывать как наполовину ничем не заполненный. Можно обсуждать объем содержимого стакана, только если оно действительно в нем есть. Точно так же разговор о ширине рамки имеет смысл только в контексте существующей рамки.

Об этом важно помнить, потому что отсутствие объявления стиля рамки – очень распространенная ошибка. Это ведет к недовольству авторов, потому что на первый взгляд стили выглядят заданными правильно. Исходя из следующего правила ни один из элементов `h1` вообще не будет иметь никакой рамки, не говоря уже о рамке шириной 20 пикселов:

```
h1 {border-width: 20px;}
```

По умолчанию свойство `border-style` имеет значение `none`, и отсутствие объявления стиля абсолютно аналогично объявлению `border-style: none`. Поэтому, если требуется, чтобы рамка появилась, надо объявить ее стиль.

Цвета рамок

По сравнению с другими параметрами рамок, задавать цвета довольно просто. CSS использует единственное свойство `border-color`, которое может принимать до четырех цветов одновременно.

border-color

Значения:	`[ <цвет> | transparent ]{1,4} | inherit`
Начальное значение:	не определено для сокращенной формы записи свойств
Область применения:	все элементы
Наследование:	нет
Вычисляемое значение:	см. отдельные свойства (`border-top-color` и др.)

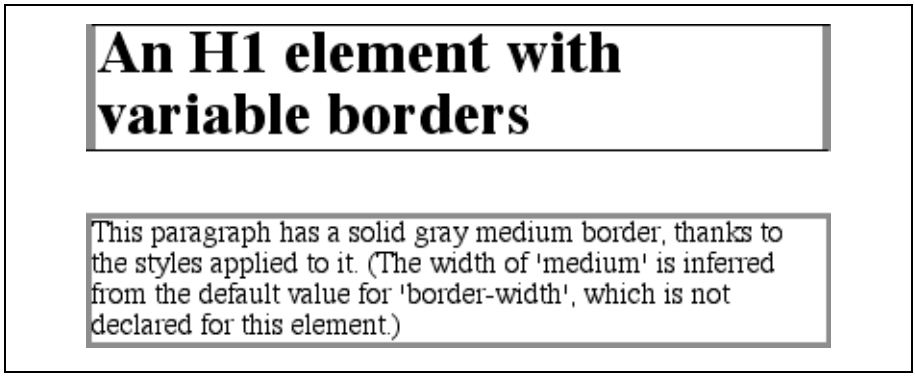


Рис. 8.28. *Рамки имеют много параметров*

Если задано меньше четырех значений, в силу вступает тиражирование. Итак, если требуется, чтобы элементы `h1` имели тонкую черную рамку сверху и снизу и толстую серую рамку по бокам, а также серые рамки средней ширины вокруг элементов `p`, подойдет такая разметка. Результат показан на рис. 8.28:

```
h1 {border-style: solid; border-width: thin thick; border-color: black gray;}
p {border-style: solid; border-color: gray;}
```

Конечно, для всех четырех сторон будет применено одно значение цвета, как это было с абзацем в предыдущем примере. С другой стороны, если указать четыре цвета, то получится рамка, все стороны которой будут окрашены в разные цвета. Могут применяться любые типы значений цветов, от именованных до шестнадцатеричных и RGB-значений:

```
p {border-style: solid; border-width: thick;
  border-color: black rgb(25%,25%,25%) #808080 silver;}
```

Как я упоминал ранее в этой главе, если цвет не объявлен, то по умолчанию применяется основной цвет элемента. Таким образом, следующее объявление будет представлено, как показано на рис. 8.29:

```
p.shade1 {border-style: solid; border-width: thick; color: gray;}
p.shade2 {border-style: solid; border-width: thick; color: gray;
   border-color: black;}
```

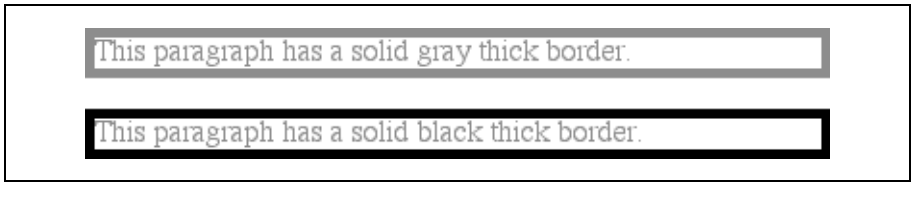


Рис. 8.29. *Цвета рамок по умолчанию и на основании значения свойства border-color*

border-top-color, border-right-color, border-bottom-color, border-left-color

Значения:	<цвет> | transparent | inherit
Начальное значение:	значение свойства color элемента
Область применения:	все элементы
Наследование:	нет
Вычисляемое значение:	если значение не задано, используется вычисляемое значение свойства color того же элемента; в противном случае – как задано

В результате у первого абзаца будет серая рамка, получившая значение gray от основного цвета абзаца. Однако рамка второго абзаца черная, потому что этот цвет был явно задан свойством border-color.

Существуют также и свойства для задания цвета рамки с одной стороны. Они во многом похожи на аналогичные свойства для стиля и ширины. Задать заголовкам сплошную черную рамку, серую справа, можно следующим образом:

```
h1 {border-style: solid; border-color: black; border-right-color: gray;}
```

Прозрачные рамки

Как вы помните, если у рамки нет стиля, то она не имеет и ширины. Однако возможны ситуации, когда надо создать невидимую рамку. Тут-то и появляется значение цвета рамки transparent (введено в CSS2). Это значение применяется для создания невидимой рамки, имеющей ширину.

Скажем, вы хотите задать трем ссылкам рамки, которые по умолчанию будут невидимыми, но при проведении указателем по ссылке получат значение inset. Этого можно достичь, делая рамки прозрачными, если указатель не находится над ссылкой:

```
a:link, a:visited {border-style: solid; border-width: 5px;
  border-color:  transparent;}
a:hover {border-color: gray;}
```

Полученный эффект показан на рис. 8.30.

В некотором смысле значение transparent дает нам возможность использовать рамки как дополнительные отступы, которые можно делать

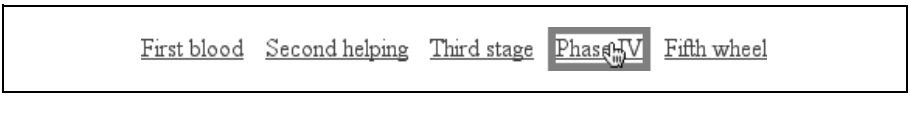
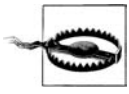


Рис. 8.30. *Применение прозрачных рамок*

видимыми по своему желанию. Они действуют как отступы, потому что фон элемента, если он видимый, распространяется на область рамки.



Поддержки значения `transparent` в IE/Win до версии IE7 нет. В этих версиях для определения цвета рамки применяется значение свойства `color` элемента.

Сокращенная форма определения рамок

К сожалению, сокращенные варианты задания свойств, такие как `border-color` и `border-style`, не всегда так уж полезны, как может показаться. Предположим, вам захотелось применить толстую серую сплошную рамку ко всем элементам `h1`, но только снизу. Ограничившись уже рассмотренными свойствами, для задания такой рамки придется потратить немало времени. Вот два примера:

```
h1 {border-bottom-width: thick;   /* вариант #1 */
  border-bottom-style: solid;
  border-bottom-color: gray;}

h1 {border-width: 0 0 thick;      /* вариант #2 */
  border-style: none none solid;
  border-color: gray;}
```

Ни один из них не является на самом деле удобным, поскольку приходится набирать значительный объем текста. К счастью, есть лучшее решение:

```
h1 {border-bottom: thick solid gray;}
```

В результате значения применяются только к нижней рамке, как показано на рис. 8.31, а все остальные значения будут определены по умолчанию. Поскольку по умолчанию применяется стиль рамки `none`, с остальных трех сторон элемента рамок не будет.

Как уже можно догадаться, таких сокращенных форм записи свойств всего четыре.

Эти свойства можно применять для создания сложных рамок; пример см. на рис. 8.32.

border-top, border-right, border-bottom, border-left

Значения:	`[ <border-width> || <border-style> || <border-color> ] | inherit`
Начальное значение:	не определено для сокращенной формы записи свойств
Область применения:	все элементы
Наследование:	нет
Вычисляемое значение:	см. отдельные свойства (`border-width` и др.)

Рис. 8.31. *Задание нижней рамки с помощью сокращенной формы записи*

Рис. 8.32. *Очень сложные рамки*

```
h1 {border-left: 3px solid gray;
  border-right: black 0.25em dotted;
  border-top: thick silver inset;
  border-bottom: double rgb(33%,33%,33%) 10px;}
```

Как видите, порядок значений не важен. Следующие три правила обеспечат совершенно аналогичную рамку:

```
h1 {border-bottom: 3px solid gray;}
h2 {border-bottom: solid gray 3px;}
h3 {border-bottom: 3px gray solid;}
```

Некоторые значения можно опустить и позволить вступить в силу значениям по умолчанию, как здесь:

```
h3 {color: gray; border-bottom: 3px solid;}
```

Поскольку цвет рамки не объявлен, вместо него подставляется значение по умолчанию (основной цвет элемента). Только помните, что если стиль рамки пропустить, то его стандартное значение `none` обеспечит ее отсутствие.

Напротив, если задать только стиль, рамка будет. Скажем, надо лишь, чтобы стиль верхней рамки был `dashed`, и требуется оставить для ее ширины значение по умолчанию `medium`, а цвет должен быть таким же, как цвет текста самого элемента. В подобном случае подойдет следующая разметка (рис. 8.33):

```
p.roof {border-top: dashed;}
```

Также надо отметить, что, поскольку каждое из этих свойств применяется только к определенной стороне рамки, не существует никакой возможности тиражирования значений – это не имело бы никакого смысла. Здесь может быть только по одному значению каждого типа, т. е. только одно значение ширины, только одно значение цвета, толь-

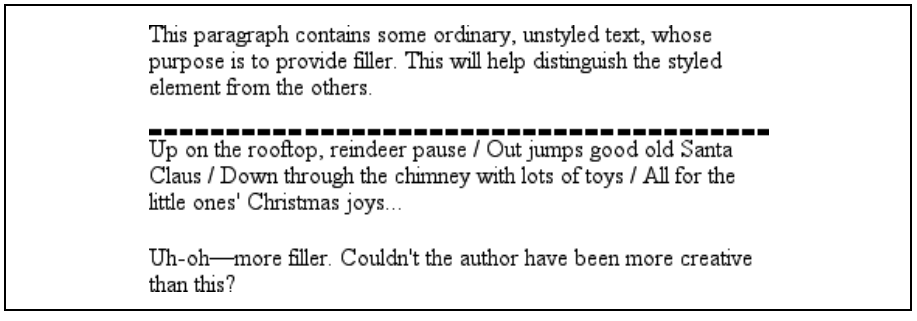


Рис. 8.33. *Проведение пунктирной линии над элементом*

ко один стиль рамки. Так что не пытайтесь объявить более одного типа значения:

```
h3 {border-top: thin thick solid purple;}  /* два значения ширины--НЕВЕРНО */
```

В подобном случае все выражение будет недействительным, и агент пользователя полностью проигнорирует его.

Задание рамки одним свойством

Теперь обратимся к самому краткому формату задания свойств рамок: `border`.

Преимущество данного свойства состоит в его исключительной компактности, хотя она и вводит некоторые ограничения. Прежде чем заняться этим, давайте посмотрим, как работает свойство `border`. Если вы хотите, чтобы все элементы `h1` имели толстую серебряную рамку, все очень просто. Визуальное представление этого объявления показано на рис. 8.34:

```
h1 {border: thick silver solid;}
```

Значения применяются ко всем четырем сторонам рамки. Эту альтернативу, конечно, надо предпочесть следующему варианту:

```
h1 {border-top: thick silver solid;
  border-bottom: thick silver solid;
```

border

Значения:	`[ <border-width> || <border-style> || <border-color> ] | inherit`
Начальное значение:	см. отдельные свойства
Область применения:	все элементы
Наследование:	нет
Вычисляемое значение:	как задано

Рис. 8.34. *Действительно короткое объявление рамки*

```
    border-right: thick silver solid;
    border-left: thick silver solid;}  /* аналогично предыдущему примеру */
```

Недостаток свойства `border` – стиль, ширина и цвет могут быть заданы только глобально. Иначе говоря, значения `border` будут применяться ко всем четырем сторонам рамки. Для того чтобы рамка элемента была разной с каждой стороны, задавая ее, придется применять другие свойства. Конечно, можно использовать каскад:

```
H1 {border: thick silver solid;
  border-left-width: 20px;}
```

Второе правило переопределяет значение ширины для рамки слева, назначенное первым правилом, заменяя таким образом `thick` на значение `20px`, как видно на рис. 8.35.

Рис. 8.35. *Применение каскада*

Но все равно применение сокращенной формы задания свойств требует аккуратности: если значение не указано, его место автоматически займет значение по умолчанию, что может породить непредусмотренные эффекты. Посмотрим:

```
h4 {border-style: dashed solid double;}
h4 {border: medium green;}
```

Здесь во втором правиле не задан стиль рамки, т. е. будет подставлено значение по умолчанию `none`, а у элементов `h4` вообще не будет рамки.

Рамки и строковые элементы

Работа с рамками в строковых элементах не должна казаться вам чем-то незнакомым, поскольку правила преимущественно аналогичны тем, которые касались применения полей в строковых элементах, что обсуждалось в главе 7. Тем не менее я кратко коснусь этой темы вновь.

Во-первых, независимо от того, насколько широкими вы делаете рамки строковых элементов, высота строки элемента не изменится. Зададим верхнюю и нижнюю рамки для текста, выделенного полужирным шрифтом:

```
strong {border-top: 10px solid gray; border-bottom: 5px solid silver;}
```

Повторяю, этот синтаксис допускается спецификацией, но абсолютно никак не повлияет на высоту строки. Но поскольку рамки видимые, они будут отрисовываться, что ясно видно на рис. 8.36.

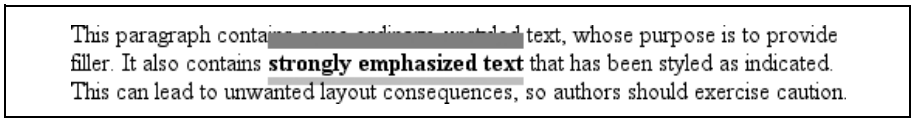


Рис. 8.36. *Рамки в строковых незамещаемых элементах*

Рамки должны были куда-то деться. Вот где они оказались.

Опять же все это выполняется только для верха и низа строковых элементов; левая и правая стороны – совсем другое дело. Если задаются рамки слева или справа, они не только будут видимыми, но также раздвинут текст вокруг, как можно увидеть на рис. 8.37:

```
strong {border-left: 25px double gray; background: silver;}
```

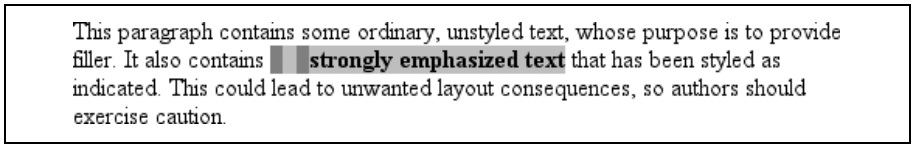


Рис. 8.37. *Строковый незамещаемый элемент, имеющий рамку слева*

При задании рамок, так же как и полей, любые свойства блоков, указанные для строковых элементов, не влияют напрямую на вычисления броузеров при определении местонахождения разрывов строки. Единственный эффект – пространство, занимаемое рамками, может немного подвинуть части строки, что, в свою очередь, может изменить местоположение разрыва строки.

С рамками CSS связано несколько проблем совместимости. Самое неприятное заключается в том, что Navigator 4.x не отрисовывает рамку непосредственно вокруг области отступа блочного элемента, а оставляет между отступом и рамкой еще некоторый промежуток. Что касается Navigator 4.x, здесь вообще крайне опасно задавать рамки или любые другие свойства блоков для строковых элементов. Это в той же степени касается и полей практически по тем же причинам (которые обсуждались в этой главе ранее).

Отступы

Между рамками и областью содержимого мы обнаруживаем *отступы* (*padding*) блока элемента. Неудивительно, что самое простое свойство, применяемое для работы с этой областью, называется `padding`.

padding	
Значения:	[<длина> | <процентное значение>]{1,4} | `inherit`
Начальное значение:	не определено для сокращенной формы записи
Область применения:	все элементы
Наследование:	нет
Процентное соотношение:	относительно ширины блока-контейнера
Вычисляемое значение:	см. отдельные свойства (`padding-top` и др.)
Примечание:	отступ никогда не может быть отрицательным

Как видите, это свойство принимает значения, выраженные в любых единицах измерения длины, или процентные значения. Итак, если требуется, чтобы все элементы `h1` имели отступ по 10 пикселов со всех сторон, это делается очень просто:

```
h1 {padding: 10px; background-color: silver;}
```

Но можно сделать и так, чтобы отступ элементов `h1` был непостоянным, а элементов `h2` – постоянным.

```
h1 {padding: 10px 0.25em 3ex 3cm;} /* непостоянный отступ */
h2 {padding: 0.5em 2em;} /* значения тиражируются для низа и левой стороны */
```

Однако если добавить только отступ, увидеть его будет достаточно сложно, поэтому давайте включим еще и цвет фона, как показано на рис. 8.38:

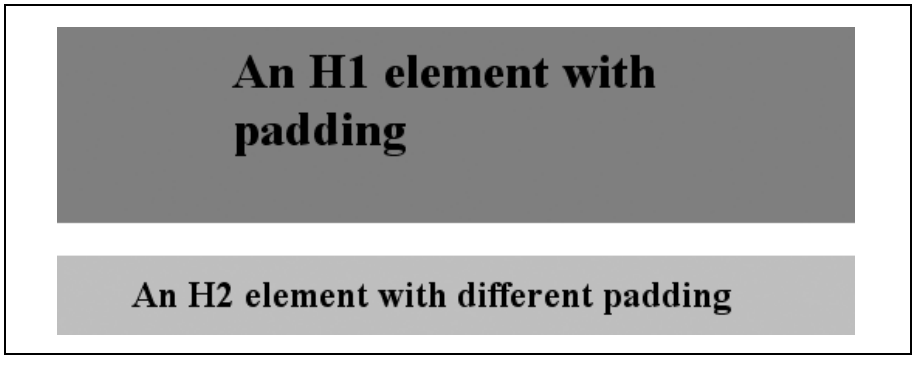


Рис. 8.38. *Непостоянный отступ, видимый с помощью фоновых цветов*

```
h1 {padding: 10px 0.25em 3ex 3cm; background: gray;}
h2 {padding: 0.5em 2em; background: silver;}
```

Как показывает рис. 8.38, фон элемента распространяется на отступы. Мы уже говорили, что он также распространяется до внешнего края рамки, но фон должен пересечь отступ, перед тем как достичь рамки.

По умолчанию у элементов нет отступов. Разделение между абзацами, например, традиционно обеспечивается только полями. Справедливо и то, что без отступов рамка элемента будет располагаться очень близко к содержимому самого элемента. Таким образом, при задании рамок элемента обычно хорошо добавить и некоторый отступ, как показано на рис. 8.39.

Даже если рамки не задаются, отступы могут вести себя по-особенному. Рассмотрим следующие правила:

```
p {margin: 1em 0; padding: 1em 0;}
p.one, p.three {background: gray;}
p.two, p.four {background: silver;}
p.three, p.four {margin: 0;}
```

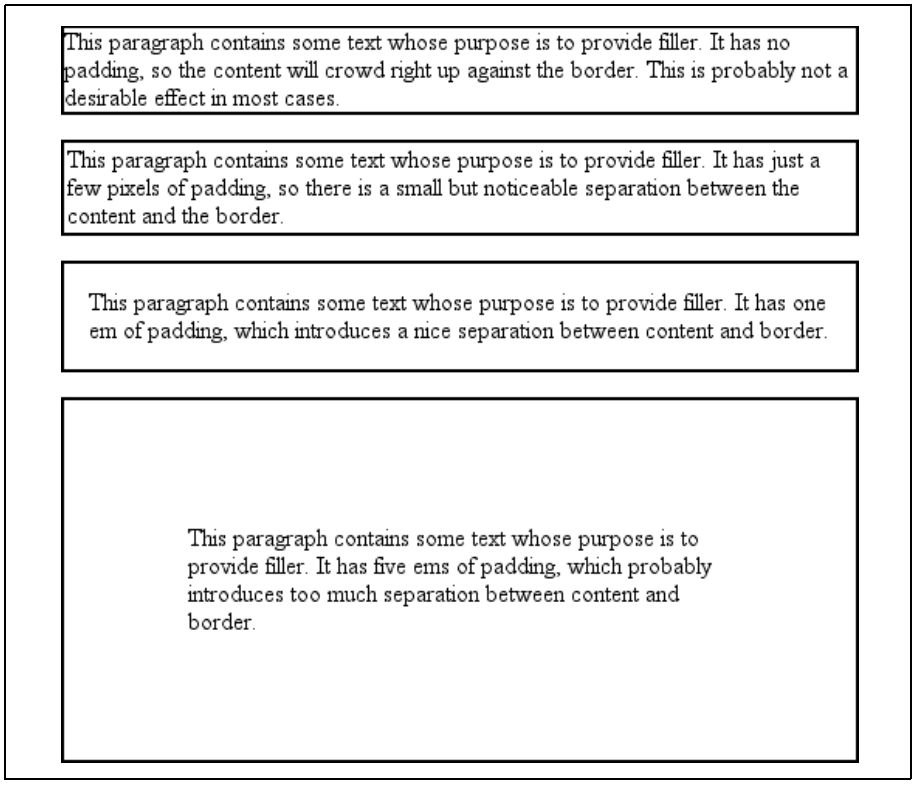


Рис. 8.39. *Результат применения отступов в ограниченных рамками блочных элементах*

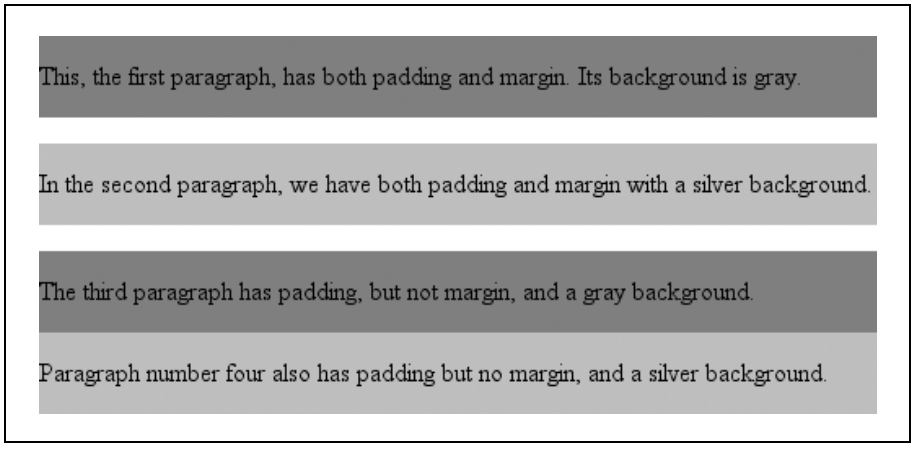


Рис. 8.40. *Разница между отступами и полями*

Здесь складывается ситуация, когда все четыре абзаца имеют отступ в `1em` сверху и снизу и два из четырех имеют верхнее и нижнее поля размером `1em`. Результаты применения этой таблицы стилей приведены на рис. 8.40.

Первые два абзаца имеют отступы и отделены друг от друга пустым пространством в `1em`, поскольку их поля свернулись. Второй и третий абзацы также отделены промежутком в `1em` из-за нижнего поля второго абзаца. Третий и четвертый абзацы не разделены, потому что у них нет полей. Однако обратите внимание на расстояние между областью содержимого двух последних абзацев: оно составляет `2em`, потому что отступы не сворачиваются. Разные цвета фона показывают, где заканчивается один абзац и начинается другой.

Следовательно, применение отступов для разделения областей содержимого элементов может быть более сложной задачей, чем применение полей, хотя здесь есть и свои преимущества. Например, чтобы с помощью отступов обеспечить разделение абзацев традиционной пустой строкой, пришлось бы написать:

```
p {margin: 0; padding: 0.5em 0;}
```

Верхний и нижний отступы каждого абзаца размером в `0,5em` объединятся и образуют общий промежуток в `1em`. Казалось бы, к чему такие хлопоты? К тому, что тогда при желании можно было бы вставить между абзацами разделительные рамки, и боковые рамки создали бы иллюзию сплошной линии. Оба этих эффекта показаны на рис. 8.41:

```
p {margin: 0; padding: 0.5em 0; border-bottom: 1px solid gray;
  border-left: 3px double black;}
```

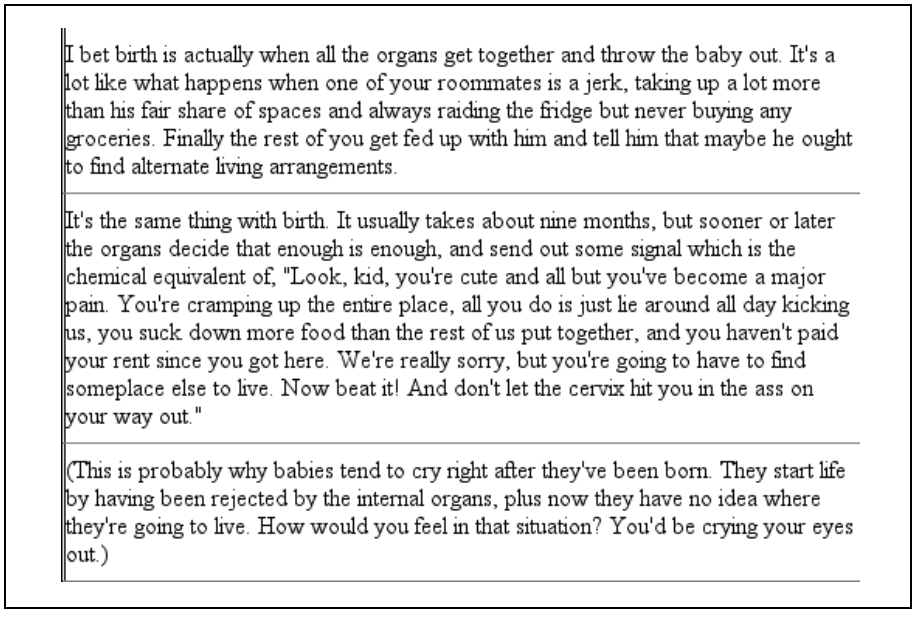


Рис. 8.41. *Применение отступов вместо полей*

Процентные значения и отступы

Мы уже говорили, что отступы элементов можно задавать процентными значениями. Как и в случае с полями, процентные значения для отступов вычисляются относительно ширины родительского элемента, таким образом они могут меняться при изменении ширины элемента-родителя. Например (рис. 8.42):

```
p {padding: 10%; background-color: silver;}

<div style="width: 200px;">
<p>This paragraph is contained within a DIV that has a width of 200 pixels, so its padding will be 10% of the width of the paragraph's parent element. Given the declared width of 200 pixels, the padding will be 20 pixels on all sides.</p>
</div>
<div style="width: 100px;">
<p>This paragraph is contained within a DIV with a width of 100 pixels, so its padding will still be 10% of the width of the paragraph's parent. There will, therefore, be half as much padding on this paragraph as that on the first paragraph.</p>
</div>
```

Заметьте, что отступы сверху и снизу равны отступам слева и справа; иначе говоря, процентное значение для верхнего и нижнего отступов вычисляется относительно ширины элемента, а не его высоты. Конеч-

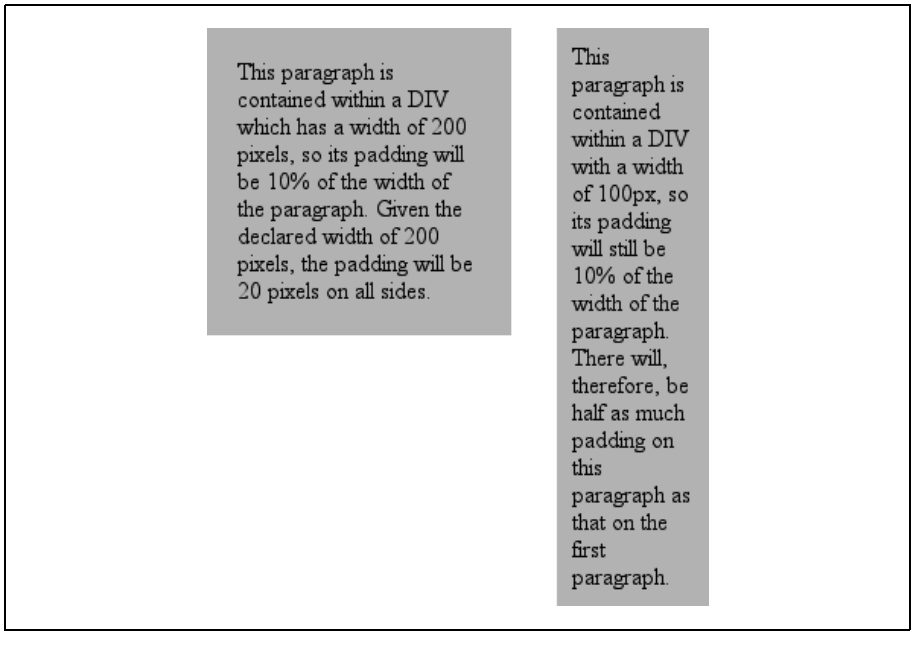


Рис. 8.42. *Отступы, процентные значения и ширина родительского элемента*

но, вы встречались с этим и раньше – в разделе «Поля», но на это следует обратить внимание еще раз, просто чтобы понять принцип работы.

Отступ с одной стороны

Наверняка вы догадались: существуют свойства, позволяющие задавать отступы с одной стороны блока, не затрагивая другие.

Эти свойства действуют так, как и предполагается. Например, следующие два правила обеспечат одинаковые отступы:

padding-top, padding-right, padding-bottom, padding-left

Значения:	<длина> | <процентное значение> | inherit
Начальное значение:	0
Область применения:	все элементы
Наследование:	нет
Процентное соотношение:	относительно ширины блока-контейнера
Вычисляемое значение:	для процентных значений – как задано; для значений длины – абсолютная длина
Примечание:	отступы никогда не могут быть отрицательными

```
h1 {padding: 0 0 0 0.25in;}
h2 {padding-left: 0.25in;}
```

Отступы и строковые элементы

Когда дело касается строковых элементов, между полями и отступами существует одно основное отличие. Для иллюстрации давайте начнем с отступов слева и справа. Здесь, если задать значения для отступа справа или слева, они будут видимыми, как показывает рис. 8.43:

```
strong {padding-left: 10px; padding-right: 10px; background: silver;}
```

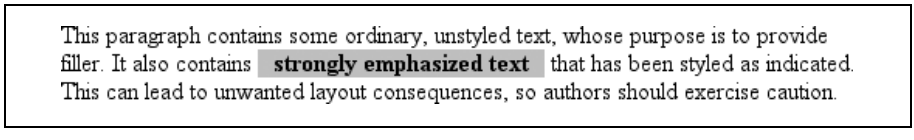


Рис. 8.43. *Отступы в строковом незамещаемом элементе*

Обратите внимание на дополнительное пространство, появляющееся на одном из концов строкового незамещаемого элемента. Это и есть ваш отступ. Как и в случае с полями, отступ слева применяется к началу элемента, а отступ справа – к концу: однако отступ *не* применяется с правой или с левой стороны в каждой строке. Это же справедливо и для замещаемых элементов, хотя, конечно, такие элементы не переносятся со строки на строку.

Теоретически строковый незамещаемый элемент с заданным цветом фона и отступом мог бы иметь фон, распространяющийся выше и ниже элемента:

```
strong {padding-top: 0.5em; background-color: silver;}
```

На рис. 8.44 показано, как это выглядит.

Конечно, высота строки не меняется, но поскольку отступы растягивают фон, он должен быть видимым, правильно? Правильно. Он видимый и перекрывает предыдущие строки – таков ожидаемый результат.

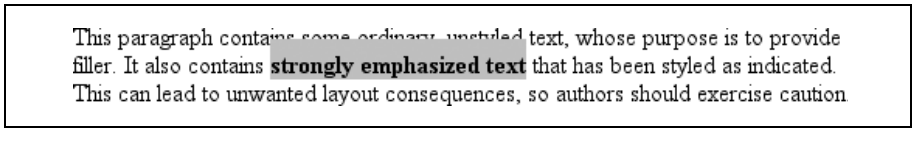


Рис. 8.44. *Большие отступы строкового незамещаемого элемента*

Отступы и замещаемые элементы

Возможно, это покажется удивительным, но можно применять отступы к замещаемым элементам, хотя на момент написания данной книги все еще существуют некоторые ограничения.

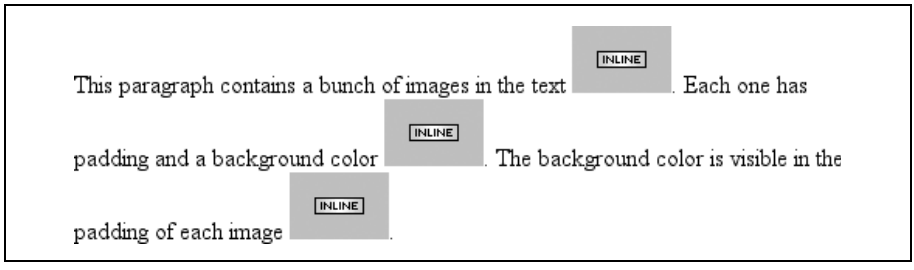


***Рис. 8.45.** Отступ в замещаемых элементах*

Самое удивительное, что отступы можно применять к изображениям, как здесь:

```
img {background: silver; padding: 1em;}
```

Независимо от того, является ли замещаемый элемент строковым или блочным, отступ будет окружать его содержимое, и цвет фона будет заполнять этот отступ, как показано на рис. 8.45. На нем же показано, что отступ отодвигает рамку элемента от его содержимого.

Однако в CSS2.1 присутствовала некоторая путаница с тем, что делать со стилевым оформлением элементов формы, таких как `input`. Не вполне понятно, например, где находится отступ у флажков. Поэтому на момент написания этой книги некоторые броузеры, такие как Mozilla, игнорируют отступы (и другие формы стилевого оформления) для элементов формы. Остается надеяться, что в будущем появится спецификация CSS, которая опишет оформление элементов формы.

Другое возможное ограничение – многие более старые броузеры, включая IE5 для Windows, не применяли отступы к изображениям.

Заключение

Возможность применять поля, рамки и отступы к любому элементу – одна из тех вещей, которая возносит CSS намного выше традиционной веб-разметки. В прошлом заключение заголовка в цветной ограниченный рамкой блок означало помещение его в таблицу, что является ужасно громоздким способом создания такого простого эффекта. Мощь CSS в подобных вопросах и делает их столь неотразимыми.

9

Цвета и фон

Помните, как вы впервые изменили цвета веб-страницы? Вместо привычного черного текста на сером фоне с синими ссылками вы вдруг смогли задавать любые сочетания цветов, например светло-синий текст на черном фоне с желто-зелеными ссылками. Отсюда оставался всего один шаг до цветного текста и в итоге даже до многоцветного текста на странице. И все благодаря тегу `<FONT COLOR="...">`. А когда вы смогли добавлять еще и фоновые изображения, возможным стало практически все, по крайней мере, так казалось. CSS пошел в вопросе цветов и фонов еще дальше, разрешая применять разные цвета и фоны на одной странице, и все без единого тега `FONT` или `TABLE`.

Цвета

Приступая к разработке страницы, прежде всего надо составить ее план. Это правило рекомендуется соблюдать в большинстве случаев, но когда речь идет о цветах, оно необходимо вдвойне. Если сделать все гиперссылки желтыми, будут ли они гармонировать с цветом фона всех частей документа? Если применить слишком много цветов, не будет ли пользователь подавлен (подсказка: да)? Если изменить цвета гиперссылок по умолчанию, найдет ли их пользователь? (Если назначить обычному тексту и тексту гиперссылок один цвет, заметить ссылки намного труднее – просто невозможно, если ссылки не подчеркнуты.)

Необходимость дополнительного планирования наверняка не отвратит почти никого из авторов от возможности изменять цвета, причем делать это они будут, вероятно, довольно часто. Адекватное применение цвета может действительно улучшить представление документа. Допустим, есть конструкция, в которой все элементы `h1` должны быть зелеными, большинство элементов `h2` – синими, и все гиперссылки должны быть темно-красными. Требуется, однако, чтобы элементы `h2`

в некоторых случаях становились темно-синими, потому что они ассоциированы с разными типами информации. Для этого проще всего каждому h2, который должен быть темно-синим, назначить класс и объявить следующее:

```
h1 {color: green;}
h2 {color: blue;}
h2.dkblue {color: navy;}
a {color: maroon;}   /* замечательный темно-красный цвет */
```

Лучше выбирать имена классов, описывающие тип содержащейся в них информации, а не визуальный эффект, который вы пытаетесь получить. Предположим, требуется сделать темно-синими все элементы h2, являющиеся заголовками подразделов. Лучше назвать класс subsec или даже sub-section. Преимущество такого имени не только в том, что оно действительно что-то означает, но, что более важно, оно не зависит от какой-либо концепции представления. Кроме того, впоследствии может потребоваться сделать все заголовки подразделов темно-красными, и тогда правило h2.dkblue {color: maroon;} будет выглядеть довольно глупо.

Из этого простого примера можно увидеть, что применение стилей лучше предварить планированием, это откроет вам доступ ко всем инструментальным средствам, имеющимся в вашем распоряжении. Предположим, надо добавить навигационную панель на страницу из предыдущего примера. Гиперссылки этой панели должны быть желтыми, а не темно-красными. Если идентификатор панели – navbar, достаточно добавить такое правило:

```
#navbar a {color: yellow;}
```

Изменится цвет гиперссылок навигационной панели, но это не повлияет на другие гиперссылки документа.

В CSS в действительности имеется только один тип окраски – плоская сплошная заливка. Если присвоить свойству color документа значение red, весь текст будет одного и того же оттенка красного. Конечно, HTML работает так же. Еще во времена HTML 3.2, объявив <BODY LINK="blue" VLINK="blue">, вы предполагали, что все гиперссылки приобретут один и тот же оттенок синего независимо от того, в каком месте документа они находятся.

Этот подход не должен измениться и в случае применения CSS. Если вы обращаетесь к CSS, чтобы присвоить свойству color всех гиперссылок (как посещенных, так и непосещенных) значение blue, значит, именно такими они и будут. Аналогично, если зеленый фон для элемента body задается при помощи стилей, то весь фон body будет одного оттенка зеленого.

В CSS можно задавать и основной цвет (цвет переднего плана), и цвет фона любого элемента, от body до выделенных элементов и гиперссы-

лок, а также элементов списка, всего списка, заголовков, ячеек таблиц и даже (в некоторой степени) изображений. Однако чтобы понять, как это делается, важно понимать, что относится к переднему плану элемента, а что нет.

Начнем с самого переднего плана; вообще говоря, это текст элемента, хотя передний план также включает и рамки вокруг элемента. Таким образом, существует два способа прямого воздействия на основной цвет элемента: применение свойства `color` и задание цвета рамок с помощью одного из целого ряда свойств задания параметров рамок, которые обсуждались в предыдущей главе.

Основные цвета

Основной цвет элемента проще всего задать при помощи свойства `color`.

color

Значения:	<цвет> | inherit
Начальное значение:	в зависимости от агента пользователя
Область применения:	все элементы
Наследование:	да
Вычисляемое значение:	как задано

Это свойство в качестве значения принимает любой действительный тип цвета, которые описаны в главе 4, например `#FFCC00` или `rgb(100%, 80%, 0%)`, а также ключевые слова системных цветов, описанные в главе 13.

Для незамещаемых элементов `color` задает цвет текста элемента, как показано на рис. 9.1:

```
<p style="color: gray;">This paragraph has a gray foreground.</p>
<p>This paragraph has the default foreground.</p>
```

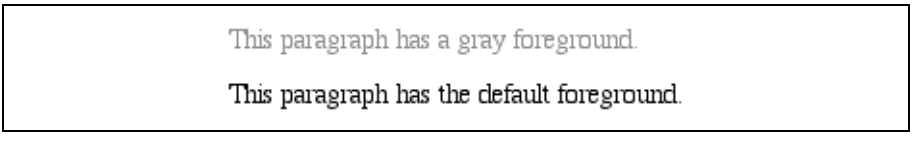


***Рис. 9.1.** Объявленный цвет в противоположность применяемому по умолчанию*

На рис. 9.1 основной цвет по умолчанию – черный. Это вовсе не обязательно, т. к. пользователь может потребовать, чтобы в его броузере (или другом агенте пользователя) цвет переднего плана (текста) был другим. Если по умолчанию текст зеленый, то второй абзац предыдущего примера был бы зеленым, а не черным, но первый абзац остался бы серым.

Конечно, вы не ограничены такими простыми операциями. Существует масса вариантов применения цветов. Допустим, в документе есть абзацы, содержащие текст, предупреждающий пользователя о потенциальной проблеме. Чтобы выделить этот текст, можно окрасить его красным. Достаточно применить к каждому абзацу предупреждения, содержащие текст, класс `warn` (`<p class="warn">`) и следующее правило:

```
p.warn {color: red;}
```

А любые непосещенные ссылки в абзацах предупреждения можно сделать зелеными:

```
p.warn {color: red;}
p.warn a:link {color: green;}
```

Если вы передумаете и решите, что текст должен быть темно-серым, а ссылки в таком тексте – светло-серыми, то внесите в предыдущие правила изменения, отражающие новые значения, как показано на рис. 9.2:

```
p.warn {color: #666;}
p.warn a:link {color: #AAA;}
```

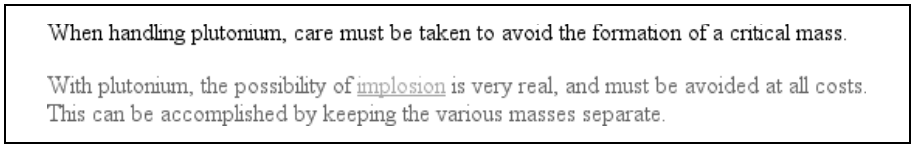


***Рис. 9.2.** Изменяя цвета*

Изменение цвета позволяет привлечь внимание к определенным типам текста. Например, текст, набранный полужирным шрифтом, уже достаточно нагляден, но можно выделить его еще сильнее, задав для него и другой цвет, скажем, `maroon`:

```
b, strong {color: maroon;}
```

И наконец, текст всех ячеек таблиц класса `highlight` можно сделать светло-желтым:

```
td.highlight {color: #FF9;}
```

Конечно, если цвет фона для текста не задан, настройки пользователя могут не очень хорошо сочетаться с вашими. Так, если пользователь задаст цвет фона своего броузера бледно-желтым, `#FFC`, то предыдущее правило сгенерирует светло-желтый текст на бледно-желтом фоне. Поэтому лучше задавать сразу и основной цвет, и цвет фона. (Мы поговорим о фоновых цветах в этой главе чуть позже.)

Будьте внимательны к применению цветов в Navigator 4, который заменяет значения свойства `color`, которые не может распознать. Эти замены не совсем случайные, но, определенно, не-

приятные. Например, invalidValue (неверное значение) получается темно-синим, а inherit (наследовать), действительное значение CSS2, превращается в совершенно отвратительный оттенок желто-зеленого. В других обстоятельствах фон со значением transparent становится черным.

Замещение атрибутов

Существует множество вариантов применения свойства color, из которых самый распространенный – замена атрибутов TEXT, LINK, ALINK и VLINK элемента BODY HTML 3.2. С помощью псевдоклассов ссылок свойство color может полностью заменить атрибуты BODY. Первую строку следующего примера можно заменить приведенным ниже CSS, и будет получен результат, показанный на рис. 9.3:

```
<body text="black" link="#808080" alink="silver" vlink="#333333">

body {color: black;}     /* замена css */
a:link {color: #808080;}
a:active {color: silver;}
a:visited {color: #333333;}
```

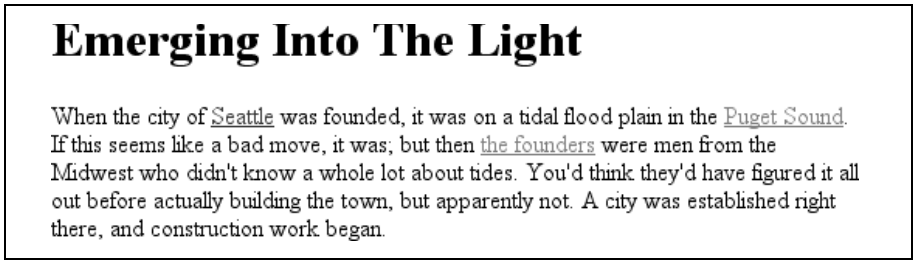


***Рис. 9.3.** Замена атрибутов BODY на CSS*

Может показаться, что при этом приходится набирать слишком много символов, но надо учесть два момента. Во-первых, это – важнейшее усовершенствование по сравнению со старым методом применения атрибутов BODY, в котором можно было вносить изменения только на уровне документа. Во-вторых, если бы потребовалось сделать некоторые ссылки светло-серыми, а другие – темно-серыми, то атрибуты BODY были бы бесполезны. Пришлось бы применять <FONT COLOR="#666666"> для каждой отдельной ссылки, которая должна быть темной. То ли дело CSS: достаточно добавить класс ко всем ссылкам, для которых требуется изменить оттенок серого, и соответствующим образом исправить стили:

```
body {color: black;}
a:link {color: #808080;}      /* светло-серый */
a.external:link {color: silver;}
a:active {color: silver;}
a:visited {color: #333;}      /* очень темный серый */
```

Все ссылки класса external станут серебряными, а не серыми. По-прежнему после посещения они будут становиться темно-серыми, если, конечно, не добавить специальное правило и для этого случая:

```
body {color: black;}
a:link {color: #808080;}      /* умеренно серый */
a.external:link  {color: #666;}
a:active {color: silver;}
a:visited {color: #333;}      /* очень темно-серый */
a.external:visited {color: black;}
```

Теперь все внешние ссылки будут серыми до посещения и черными после посещения, тогда как остальные ссылки будут темно-серыми после посещения и светло-серыми до него.

Воздействие на рамки

Значение цвета также может влиять на рамки, окружающие элемент. Предположим, вы объявили приведенные ниже стили, результат применения которых представлен на рис. 9.4:

```
p.aside {color: gray; border-style: solid;}
```

У элемента <p class="aside"> серый текст и серая сплошная рамка средней ширины. Дело в том, что по умолчанию к рамкам применяется основной цвет элемента. Переопределить это поведение позволяет свойство border-color:

```
p.aside {color: gray; border-style: solid; border-color: black;}
```

Текст станет серым (grey), а цвет рамок – черным (black). Любое значение, заданное для border-color, всегда будет переопределять значение свойства color.

Между прочим, рамки позволяют менять основной цвет изображений. Поскольку изображения уже составлены из цветов, нельзя влиять на них с помощью свойства color, но можно менять цвет любой рамки,

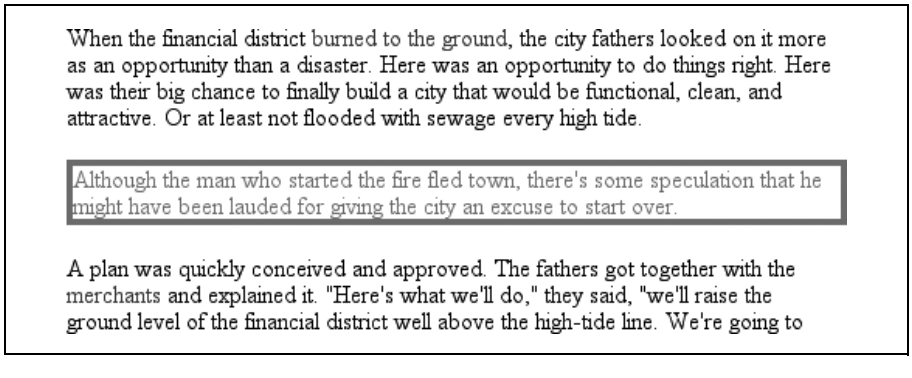


Рис. 9.4. *Цвета рамок, позаимствованные у цвета содержимого*

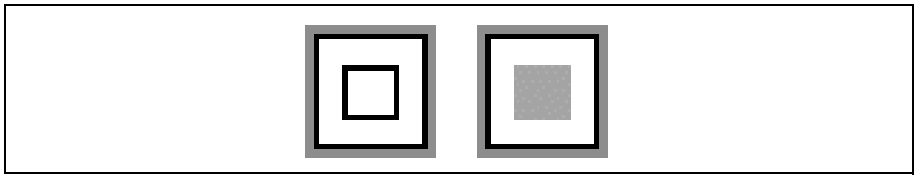


***Рис. 9.5.** Задание цвета рамки для изображений*

окружающей изображение. Это достигается посредством задания свойства `color` или `border-color`. Таким образом, следующие правила будут иметь одинаковый визуальный эффект на изображения класса `type1` и `type2`, как показано на рис. 9.5:

```
img.type1 {color: gray; border-style: solid;}
img.type2 {border-color: gray; border-style: solid;}
```

Воздействие на элементы формы

Заданное значение свойства `color` можно (теоретически всегда) применить к элементам формы. Объявление для элементов `select` темно-серого текста должно быть простым:

```
select {color: rgb(33%,33%,33%);}
```

При этом, возможно, будет задан (а возможно, и нет) цвет рамок, окружающих края элемента `select`. Все зависит от агента пользователя и применяемых по умолчанию стилей.

Также можно было бы задать основной цвет элементов `input`, хотя, как видно на рис. 9.6, в результате данный цвет будет применен ко всем элементам ввода, от текста до переключателей и флажков.

```
select {color: rgb(33%,33%,33%);}
input {color: gray;}
```

Обратите внимание, что на рис. 9.6 текст рядом с флажками остается черным. Дело в том, что стили назначены только таким элементам, как `input` и `select`, а не тексту обычного (или какого-либо иного) абзаца.

В CSS1 не было возможности различать типы элементов `input`. Таким образом, для того чтобы задать для флажков один цвет, а для переключателей другой, пришлось бы назначать им классы:

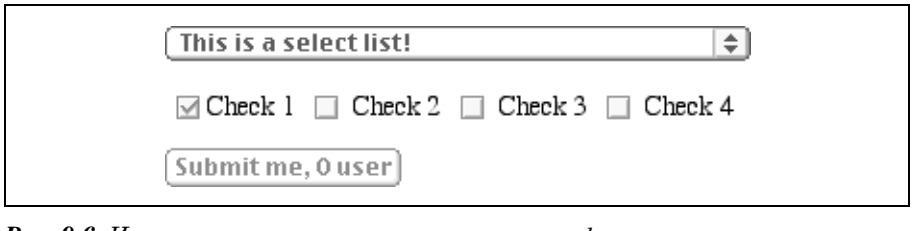


***Рис. 9.6.** Изменение основных цветов элементов формы*

```
input.radio {color: #666;}
input.check {color: #CCC;}

<input type="radio" name="r2" value="a" class="radio" />
<input type="checkbox" name="c3" value="one" class="check" />
```

В CSS2 и более поздних версиях различать элементы на основании их атрибутов несколько проще благодаря селекторам атрибутов:

```
input[type="radio"] {color: #333;}
input[type="checkbox"] {color: #666;}

<input type="radio" name="r2" value="a " />
<input type="checkbox" name="c3" value="one "/>
```

Селекторы атрибутов позволяют обходиться вообще без классов, по крайней мере в этом случае. К сожалению, многие агенты пользователя не поддерживают селекторы атрибутов, поэтому, вероятно, еще какое-то время классы будут необходимы.

Navigator 4 не применяет цвета к элементам формы, но определение цветов для элементов формы работает в Internet Explorer 4 и последующих версиях и Opera 3.5+. Однако многие версии остальных броузеров вообще не обеспечивают стилевого оформления элементов формы из-за неопределенности в том, как они должны быть оформлены.

Наследование цвета

Как показывает объявление `color`, данное свойство относится к наследуемым. Это логично, поскольку если вы задаете `p {color: gray;}`, то, вероятно, ожидаете, что любой текст в этом абзаце тоже будет серым, даже если он выделен, набран жирным шрифтом и т. д. Конечно, если вы *хотите*, чтобы данные элементы имели другой цвет, сделать это довольно просто, как показано на рис. 9.7:

```
em {color: gray;}
p {color: black;}
```

Цвет наследуется, поэтому теоретически возможно задать для всего обычного текста документа какой-нибудь цвет, например красный, объявив `body {color: red;}`. Весь текст, не оформленный каким-либо иным способом (например, ссылки, которые имеют собственные цветовые стили), должен стать красным. Однако до сих пор встречаются броузеры, в которых цвета для таких элементов, как таблицы, предо-

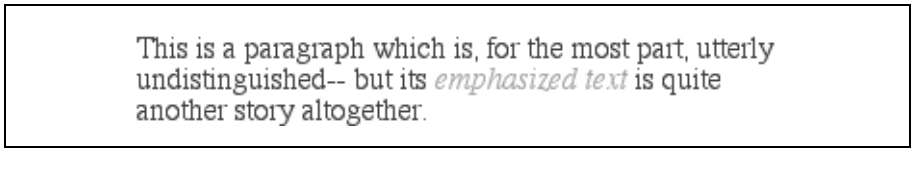


Рис. 9.7. *Разные цвета для разных элементов*

пределены, что не допускает наследование цветов body ячейками таблицы. В подобных броузерах, поскольку значение color для элементов table определяется броузером, значение броузера приоритетней наследуемого значения. Это бессмысленно и вызывает досаду, но, к счастью, с этим нетрудно справиться (как правило) с помощью селекторов, в которых перечислены различные элементы таблицы. Например, так можно сделать все содержимое таблиц красным по всему телу документа:

```
body, table, td, th {color: red;}
```

Это, как правило, решает проблему. Заметьте, что во многих современных броузерах, которые давно установили и исправили ошибки наследования, приносившие неприятности в более ранних версиях, нет необходимости применять подобные селекторы.

Фон

Область фона элемента включает все пространство за передним планом вплоть до внешнего края рамок; таким образом, блок содержимого и отступы – все это части фона элемента, и рамки отрисовываются поверх фона.

CSS позволяет применять чистый цвет или создавать умеренно сложные эффекты с помощью фоновых изображений; его возможности в этой области намного превосходят HTML.

Фоновый цвет

Цвет фона элемента можно объявлять почти так же, как основной цвет. Для этого применяется свойство background-color, принимающее (что и не удивительно) любой действительный цвет, или ключевое слово, делающее фон прозрачным.

Для того чтобы цвет немного выступал за края текста элемента, просто добавьте отступы, как показано на рис. 9.8:

```
p {background-color: gray; padding: 10px;}
```

Фоновый цвет можно задать практически для любого элемента, от body до строковых элементов, таких как em и a. Свойство background-color

background-color

Значения:	<цвет> | transparent | inherit
Начальное значение:	transparent
Область применения:	все элементы
Наследование:	нет
Вычисляемое значение:	как задано

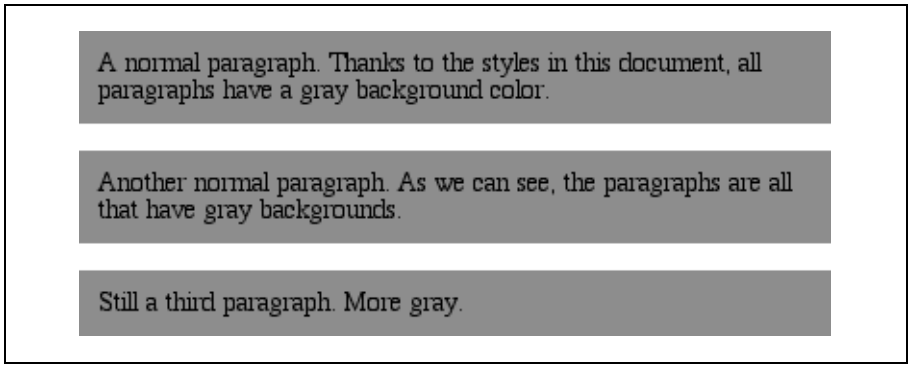


Рис. 9.8. *Фон и отступы*

не наследуется. Для него по умолчанию устанавливается значение `transparent`: если цвет элемента не определен, его фон должен быть прозрачным, чтобы был виден фон его элемента-предка.

Ситуацию с наследованием можно представить в виде прозрачной пластиковой вывески на окрашенной стене. Стена видна сквозь вывеску, но это не фон вывески, а фон стены (во всяком случае с точки зрения CSS). Аналогично, если задается фон для холста, он будет виден сквозь все элементы документа, не имеющие собственного фона. Они не наследуют фон; он просматривается *сквозь* них. Это может показаться несущественным различием, но, как вы увидите в разделе, посвященном фоновым изображениям, в действительности это важно.

Чаще всего в применении ключевого слова `transparent` нет необходимости, т. к. это значение по умолчанию. Однако иногда оно может быть полезным. Представьте, что броузер пользователя делает фон всех ссылок белым. Создавая свою страницу, вы делаете белым основной цвет ссылок и не хотите, чтобы они имели фон. Чтобы обеспечить своему выбору преимущество, объявите:

```
a {color: white; background-color: transparent;}
```

Если бы объявление фонового цвета было опущено, ваш основной белый цвет в сочетании с белым фоном пользователя дал бы совершенно нечитаемые ссылки. Такая ситуация маловероятна, но возможна.

Из-за потенциальной возможности совпадения стилей автора и читателя утилита проверки достоверности CSS будет генерировать предупреждения, такие как «You have no background-color with your color» (Вы не задали фоновый цвет). Она пытается напомнить вам, что может произойти взаимодействие автор-пользователь, а правило не учитывает эту возможность. Предупреждения не означают, что стили заданы неверно, только ошибки мешают подтверждению достоверности.

Проблемы, сложившиеся исторически

Итак, задание фонового цвета – довольно простая вещь, за исключением одного небольшого предостережения. Navigator 4 совершенно неправильно размещает фоновые цвета. Вместо того чтобы применять фоновый цвет ко всему блоку содержимого и отступам, он закрашивает только область непосредственно под текстом, как показано на рис. 9.9.

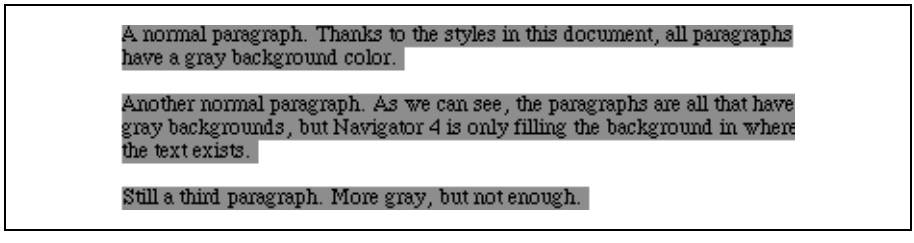


Рис. 9.9. *Неверное поведение Navigator 4.x*

Я повторяю: *это совершенно неправильное поведение*. Чтобы противодействовать ему, необходимо задать рамку элемента, которой потом можно присвоить цвет, совпадающий с цветом фона документа:

```
body {background: silver;}
p {background-color: gray; padding: 0.1px; border: 1px solid silver;}
```

Обратите внимание: чтобы этот метод работал, надо задать свойство `border-style`. Неважно, задаете ли вы это специальное свойство или просто значение свойства `border`.

Конечно, делая это, вы задаете рамку элемента, и эта рамка будет обнаруживаться также и в других агентах пользователя. Кроме того, Navigator не очень хорошо обрабатывает отступы, поэтому в предыдущем примере между блоком содержимого и рамками появляется небольшое пустое пространство. К счастью, более новые броузеры не порождают таких проблем.

Специальные эффекты

Комбинируя свойства `color` и `background-color`, можно создавать некоторые полезные эффекты:

```
h1 {color: white; background-color: rgb(20%,20%,20%);
  font-family: Arial, sans-serif;}
```

Этот пример показан на рис. 9.10.

Конечно, количество сочетаний цветов велико, но, будучи ограниченным черно-белой палитрой, я не могу продемонстрировать здесь их все. Все-таки я попытаюсь высказать некоторые соображения по поводу того, что вы можете сделать.

Эта таблица стилей немного сложнее, как показано на рис. 9.11:

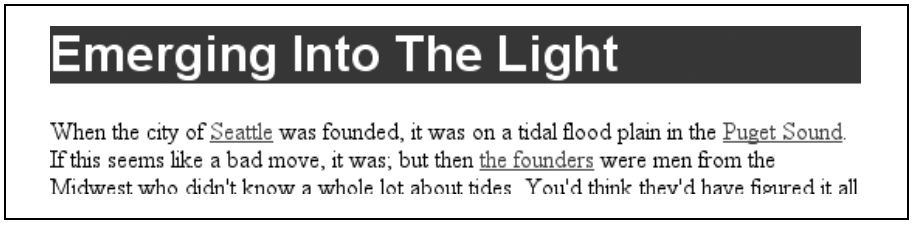


Рис. 9.10. *Стильная раскраска элементов H1*

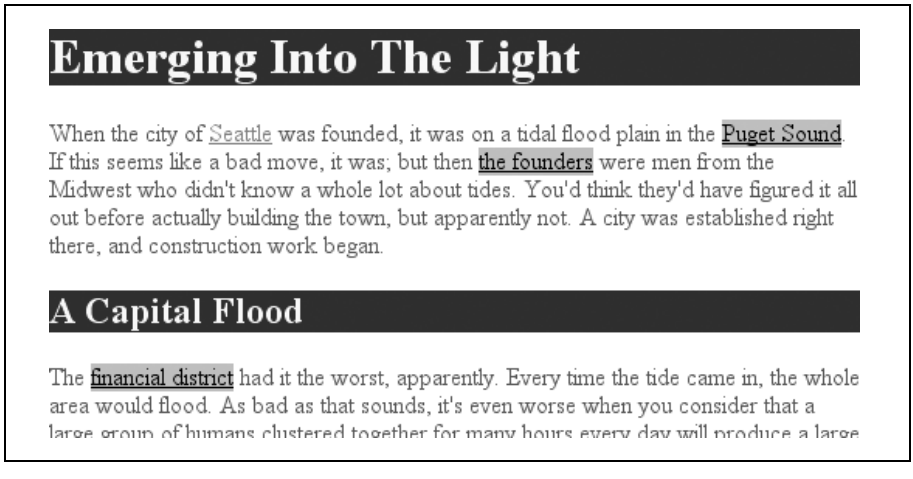


Рис. 9.11. *Результаты применения более сложной таблицы стилей*

```
body {color: black; background-color: white;}
h1, h2 {color: yellow; background-color: rgb(0,51,0);}
p {color: #555;}
a:link {color: black; background-color: silver;}
a:visited {color: gray; background-color: white;}
```

Это, конечно, лишь малая толика возможного. Пожалуйста, обязательно попытайтесь сделать несколько примеров самостоятельно!

Фоновые изображения

Рассмотрев основы задания цветов фона и переднего плана, обратимся теперь к вопросу о фоновых изображениях. В HTML 3.2 с помощью атрибута `BACKGROUND` элемента `BODY` можно делать изображение фоном документа:

```
<BODY BACKGROUND="bg23.gif">
```

При этом агент пользователя загружает файл *bg23.gif* и затем размещает его на заднем плане документа, повторяя его как в горизонтальном, так и в вертикальном направлениях, чтобы оно заполнило весь фон документа. Этот эффект может быть продублирован в CSS, но CSS

позволяет сделать гораздо больше простого выкладывания фона мозаикой из заданных изображений. Начнем с азов, а дальше перейдем к более сложным вещам.

Использование изображений

Прежде всего, чтобы сделать изображение фоном, используем свойство `background-image`.

background-image

Значения:	<uri> | none | inherit
Начальное значение:	none
Область применения:	все элементы
Наследование:	нет
Вычисляемое значение:	абсолютный URI

Применяемое по умолчанию значение `none` приводит именно к тому, чего и можно ожидать: на заднем плане нет никакого изображения. Для того чтобы фоновое изображение было, необходимо присвоить этому свойству значение URL:

```
body {background-image: url(bg23.gif);}
```

Благодаря применяемым по умолчанию значениям других свойств фона, это приведет к тому, что изображение *bg23.gif* будет размещено на заднем плане документа, как показано на рис. 9.12. Однако, как вскоре выяснится, мозаичное расположение не является единственно возможным вариантом.

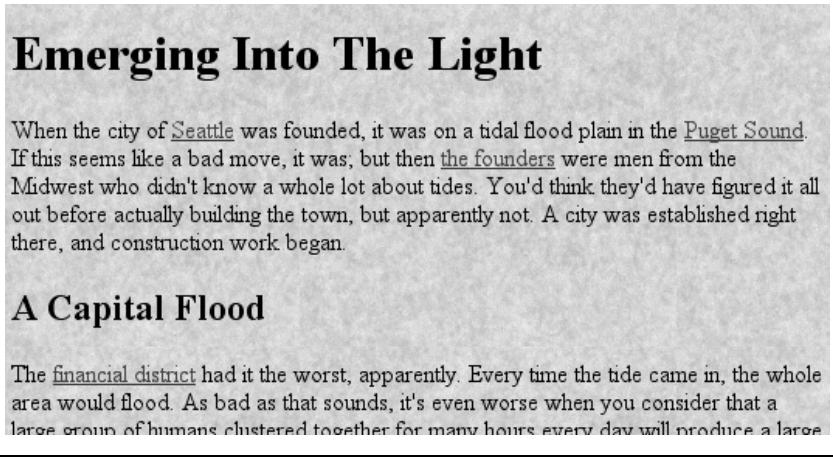


Рис. 9.12. *Применение фонового цвета в CSS*

Как правило, имеет смысл задать вместе с фоновым изображением и цвет фона; мы вернемся к этому немного позже в данной главе.

Фоновые изображения можно применять к любым блочным или строковым элементам, и в основном, конечно, фон применяется к элементам `body`, но останавливаться на этом не обязательно:

```
p.starry {background-image: url(http://www.site.web/pix/stars.gif);
   color: white;}
a.grid {background-image: url(smallgrid.gif);}
```

```
<p class="starry">It's the end of autumn, which means the stars will be
brighter than ever!  <a href="join.html" class="grid">Join us</a> for a
fabulous evening of planets, stars, nebulae, and more...
```

Как видно на рис. 9.13, фон применяется лишь к одному абзацу и ни к какой другой части документа. Можно пойти еще дальше, например разместить фоновые изображения в строковых элементах, таких как гиперссылки, что также отражено на рис. 9.13. Конечно, если вы хотите увидеть мозаичный шаблон, изображение должно быть небольшим. В конце концов, отдельные буквы тоже не так уж велики!

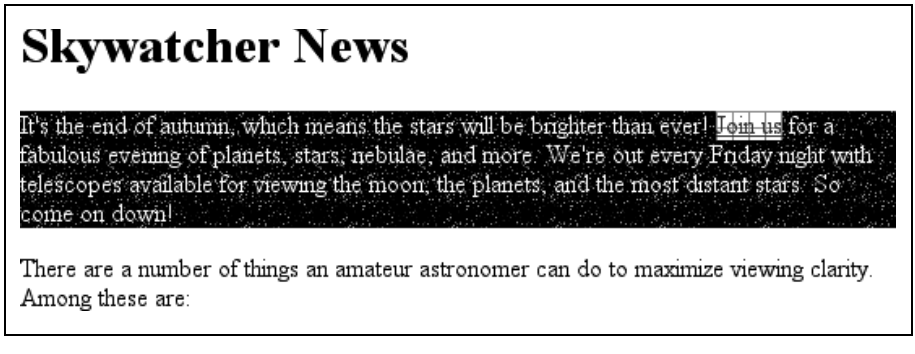


Рис. 9.13. *Применение фоновых изображений к блочным и строковым элементам*

Существует ряд способов применения специальных фоновых изображений. Можно разместить изображение на заднем плане элементов `strong`, чтобы еще больше выделить их. Или заполнить фон заголовков волнистыми разводами или маленькими точками. Можно даже заполнить ячейки таблиц шаблонами, чтобы они выделялись на странице, как показано на рис. 9.14.

```
td.nav {background-image: url(darkgrid.gif);}
```

Есть даже теоретическая возможность применить изображения к фону замещаемых элементов, таких как `textarea` (область текста) и списки `select`, хотя не каждый агент пользователя способен обработать такие стили.

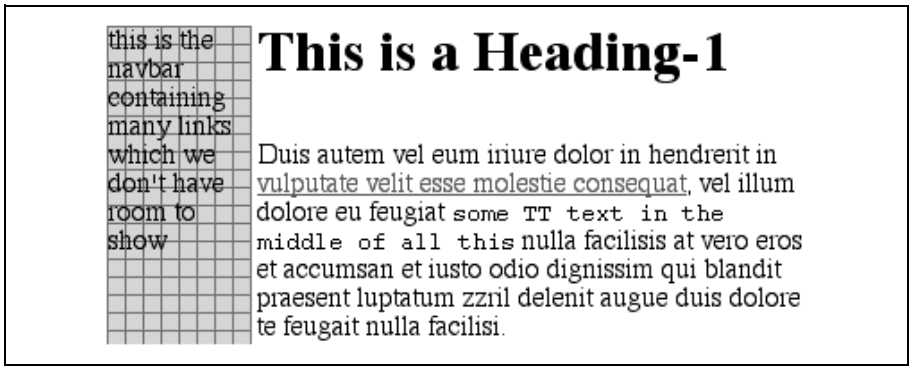


Рис. 9.14. *Задание фонового изображения для ячейки таблицы*

Так же как и `background-color`, свойство `background-image` не наследуется – кстати, не наследуется ни одно из свойств фона. Помните также, что при задании URL фонового изображения он попадает под общие ограничения и предостережения использования значений `url`: относительный URL интерпретируется относительно таблицы стилей, но Navigator 4.x делает это неправильно, поэтому легче задавать абсолютные URL.

Почему фон не наследуется

Ранее я особенно отмечал, что фон не наследуется. Фоновые изображения показывают, почему наследование в данном случае имело бы негативный эффект. Представьте ситуацию: фон наследуется, и вы применили фоновое изображение к элементу `body`. Это изображение выступало бы в качестве фона для каждого элемента документа, и у каждого элемента был бы свой мозаичный шаблон расположения изображения, как показано на рис. 9.15.

Обратите внимание на возобновление шаблона в верхнем левом углу каждого элемента, включая ссылки. Это не то, чего хотелось бы добиться многим авторам, и именно поэтому свойства фона не наследуются. Если же по какой-то причине надо добиться именно такого эффекта, то его можно реализовать посредством следующего правила:

```
* {background-image: url(yinyang.gif);}
```

Как вариант можно использовать значение `inherit` вот так:

```
body {background-image: url(yinyang.gif);}
* {background-image: inherit;}
```

Хорошая практика работы с фоном

Изображения накладываются поверх любого определяемого фонового цвета. Если заполнить изображениями GIF, JPEG или другими типами непрозрачных изображений всю область представления, этот факт не имеет особого значения, поскольку мозаика изображений заполнит

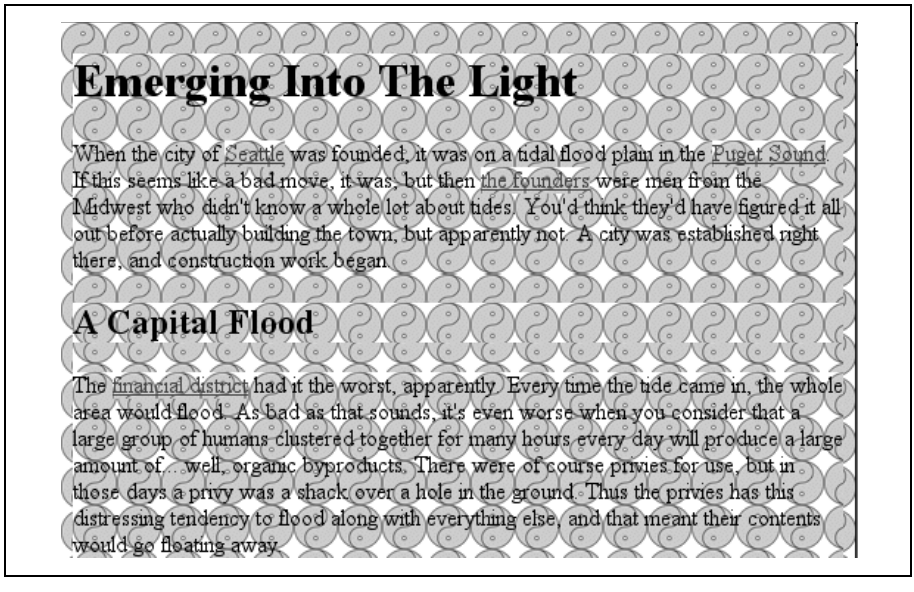


Рис. 9.15. *Что было бы с компоновкой, если бы фон наследовался*

весь фон документа, и цвет фона нигде не будет виден сквозь изображение. Однако форматы изображений с альфа-каналом, такие как PNG, могут быть частично или полностью прозрачными, что будет приводить к комбинированию изображения с цветом фона. Кроме того, если по какой-то причине не удается загрузить изображение, вместо него агент пользователя подставит заданный фоновый цвет. Посмотрим, как выглядел бы пример, показанный на рис. 9.13, если бы фоновое изображение не загрузилось (рис. 9.16).

Рисунок 9.16 демонстрирует, почему при работе с фоновыми изображениями рекомендуется всегда задавать и фоновый цвет: тогда вы, по крайней мере, получаете читаемый результат:

```
p.starry {background-image: url(http://www.site.web/pix/stars.gif);
   background-color: black; color: white;}
```

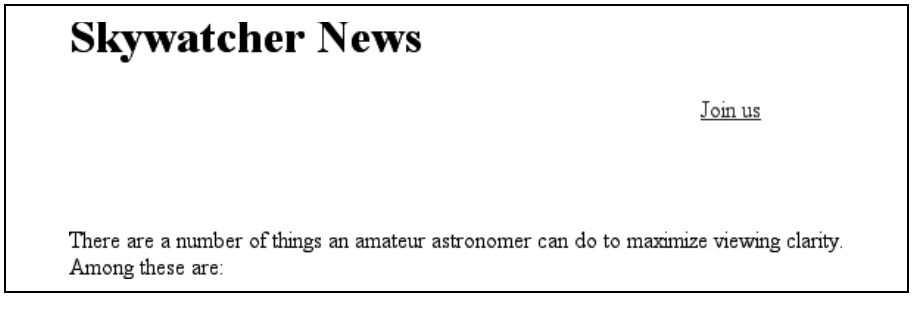


Рис. 9.16. *Последствия отсутствия фонового изображения*

```
a.grid {background-image: url(smallgrid.gif);}

<p class="starry">Конец осени, это значит, что звезды будут ярче, чем когда-
либо!  <a href="join.html" class="grid">Проведем вместе</a> этот
фантастический вечер планет, звезд, галактик...
```

Этот вариант гарантирует сплошной черный фон в случае, если «звездное» изображение по какой-то причине не может быть отображено. Кроме того, если не предполагается выкладывать изображение мозаикой по всему фону документа, понадобится цвет для заполнения свободного пространства.

Повторения в определенном направлении

Раньше, чтобы обеспечить фон для врезки, приходилось создавать очень узкое, но чрезвычайно длинное изображение. Одно время такие изображения чаще всего имели 10 пикселов в высоту и 1500 пикселов в ширину. Большую часть такого изображения, конечно, занимало пустое пространство, и лишь примерно 100 пикселов слева содержали изображение-«врезку». Остальная часть изображения зря занимала место.

Может быть, намного эффективнее создавать для врезки изображение высотой 10 и шириной 100 пикселов, не занимая лишнего места, и потом повторять его только в вертикальном направлении? Это, определенно, облегчило бы вашу работу и намного уменьшило бы время загрузки страницы. Рассмотрим свойство `background-repeat`.

Как вы, возможно, догадались, если задано значение `repeat`, изображение выкладывается мозаикой в вертикальном и в горизонтальном направлениях, как это всегда происходило с фоновыми изображениями раньше. Если задать значения `repeat-x` и `repeat-y`, то изображение повторяется в горизонтальном или вертикальном направлениях соответственно, а значение `no-repeat` вообще предотвращает мозаичное выкладывание изображения в любом направлении.

По умолчанию фоновое изображение размещается, начиная с верхнего левого угла элемента. (Позже в данной главе мы увидим, как изменить это поведение.) Поэтому применение следующих правил даст результат, представленный на рис. 9.17:

```
body {background-image: url(yinyang.gif);
   background-repeat: repeat-y;}
```

background-repeat

Значения:	`repeat | repeat-x | repeat-y | no-repeat | inherit`
Начальное значение:	`no-repeat`
Область применения:	все элементы
Наследование:	нет
Вычисляемое значение:	как задано

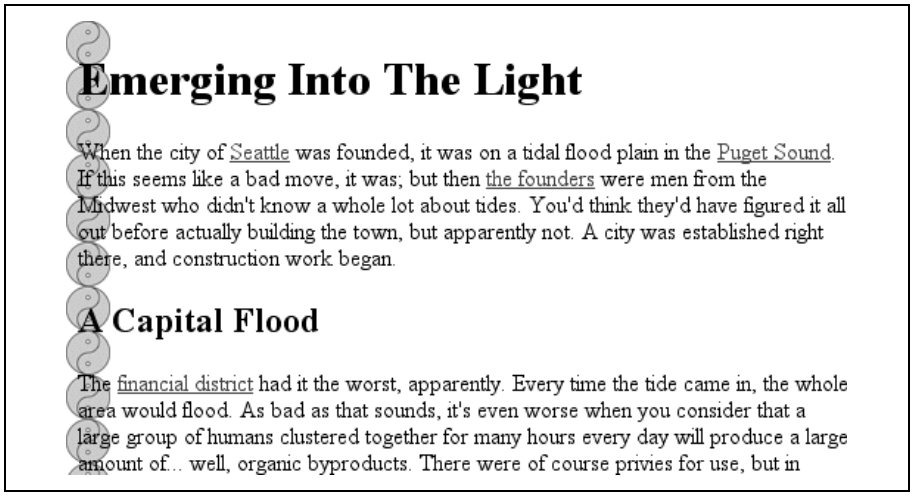


Рис. 9.17. *Мозаичное расположение фонового изображения в вертикальном направлении*

(Я опустил фоновый цвет, чтобы сократить правило, но не забывайте всегда включать его, если задаете фоновое изображение.)

Однако предположим, что требуется повторять изображение вдоль верха документа. Вместо того чтобы создавать специальное изображение с огромным пустым пространством снизу, достаточно немного изменить правило:

```
body {background-image: url(yinyang.gif);
   background-repeat: repeat-x;}
```

Как показано на рис. 9.18, изображение повторяется вдоль оси *x* (т. е. горизонтально) от своего начального положения, в данном случае от верхнего левого угла окна броузера.

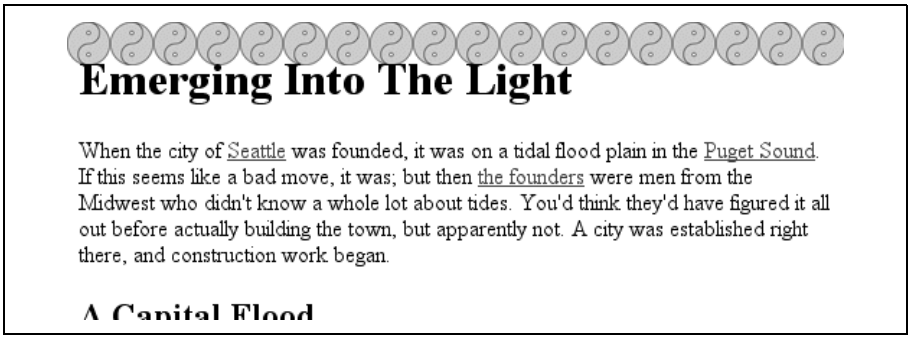


Рис. 9.18. *Мозаичное расположение по горизонтали*

Наконец, можно вообще не повторять фоновое изображение. В этом случае указывается значение `no-repeat`:

```
body {background-image: url(yinyang.gif);
   background-repeat: no-repeat;}
```

Это значение может показаться не особо полезным исходя из того, что данное объявление просто разместило бы изображение в верхнем левом углу документа, но давайте применим его к изображению большего размера, как показано на рис. 9.19:

```
body {background-image: url(bigyinyang.gif);
   background-repeat: no-repeat;}
```

Возможность задавать вектор повторения существенно расширяет диапазон возможных эффектов. Предположим, требуется разместить тройную рамку с левой стороны каждого элемента `h1` документа. Эту идею можно развить и добавить волнистую рамку над каждым элементом `h2`. Из-за окраски изображение сливается с фоном, создавая иллюзию волны, что показано на рис. 9.20:

```
h1 {background-image: url(triplebor.gif); background-repeat: repeat-y;}
h2 {background-image: url(wavybord.gif); background-repeat: repeat-x;
    background-color: #CCC;}
```

Выбирая соответствующее изображение и творчески подходя к работе с ним, можно создавать очень интересные эффекты. И это не предел.

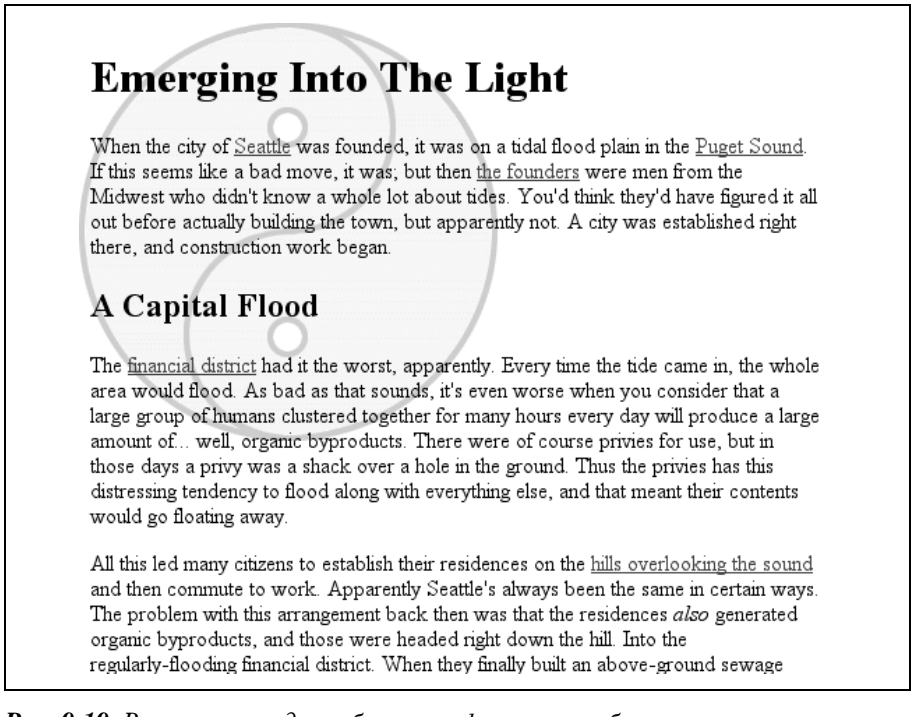


Рис. 9.19. *Размещение одного большого фонового изображения*

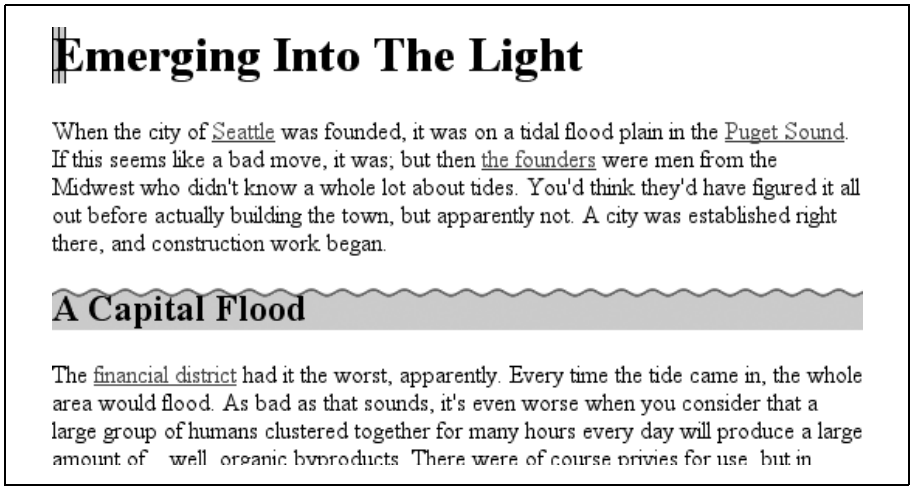


Рис. 9.20. *Обрамление элементов с помощью фоновых изображений*

Теперь вы знаете, как ограничить повторение фонового изображения, и не пора ли обсудить способы его перемещения по области фона?

Позиционирование фона

Благодаря свойству `background-repeat` существует возможность размещать большое изображение на заднем плане документа и не допустить его повторения. Давайте расширим эту возможность и изменим местоположение изображения.

background-position

Значения:	`[[`<процентное значение> `|` <длина> `| left	center	right] [`<процентное значение>`] |` <длина> `| top	center	bottom]?] || [[left	center	right] || [top	center	bottom]]	inherit`
Начальное значение:	`0% 0%`
Область применения:	блочные и замещаемые элементы
Наследование:	нет
Процентные соотношения:	относительно соответствующей точки элемента и исходного изображения (смотрите пояснения в разделе «Процентные значения» далее в этой главе)
Вычисляемое значение:	смещения абсолютной длины, если задана <длина>; в противном случае – процентное значение

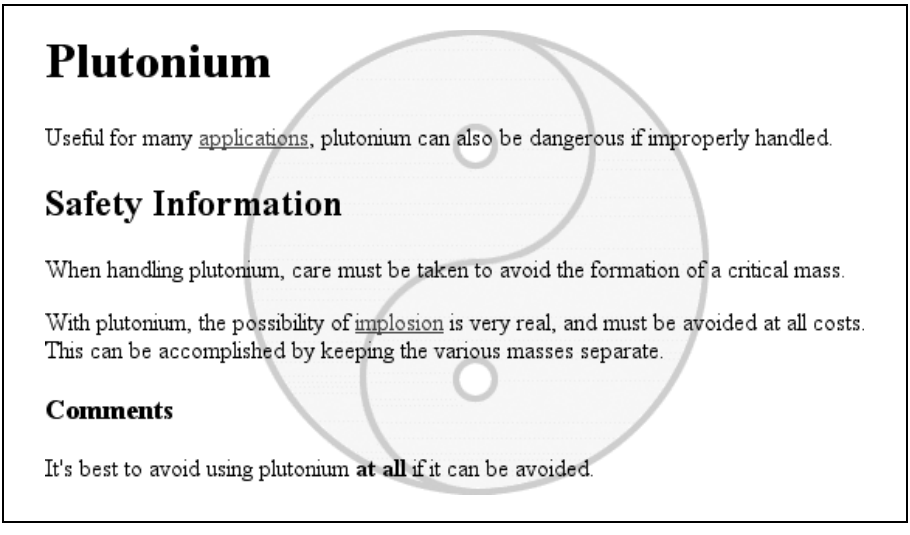


Рис. 9.21. *Центрирование единственного фонового изображения*

Например, можно центрировать фоновое изображение в элементе `body` (рис. 9.21):

```
body {background-image: url(bigyinyang.gif);
   background-repeat: no-repeat;
   background-position: center;}
```

Фактически вы поместили единственное изображение на задний план, а затем предотвратили его повторение с помощью значения `no-repeat`. Любой фон, включающий изображение, начинается с одного изображения, которое затем повторяется (или нет) в соответствии со значением свойства `background-repeat`. Эта начальная точка называется *базовой точкой изображения* (*origin image*).

Размещение базовой точки изображения выполняется с помощью свойства `background-position`, и есть несколько способов задания значений этого свойства. Во-первых, ключевые слова `top`, `bottom`, `left`, `right` и `center`. Как правило, они появляются попарно, но (как показывает предыдущий пример) не всегда. Во-вторых, значения, заданные в единицах измерения длины, такие как `50px` или `2cm`, и процентные значения. Каждый тип значения по-своему размещает фоновое изображение.

Я должен упомянуть еще о контексте, в котором размещаются фоновые изображения. CSS2 и CSS2.1 утверждают, что свойство `background-position` применяется для размещения исходного изображения относительно края отступов элемента. Иначе говоря, контекст размещения изображения – внутренний край рамки, даже несмотря на то, что область фона распространяется до внешнего края рамки. Не все броузеры размещают изображения правильно: некоторые помещают базо-

вую точку изображения относительно внешнего, а не внутреннего края рамки. В любом случае, когда рамок нет, эффект одинаков.

Для тех, кому интересно, как изменился CSS: CSS1 определял местоположение относительно области содержимого.

Независимо от контекста размещения изображения мозаично выложенное фоновое изображение заполняет весь фон области рамки, потому что мозаичное изображение распространяется во всех четырех направлениях. Позже мы поговорим об этом более подробно. Сначала надо выяснить, как исходное изображение может быть размещено в элементе.

Ключевые слова

Ключевые слова очень просты для понимания. Оказываемый ими эффект соответствует их именам; например, согласно значению `top right` исходное изображение будет находиться в верхнем правом углу области отступа элемента. Вернемся к маленькому символу инь-ян:

```
p {background-image: url(yinyang.gif);
   background-repeat: no-repeat;
   background-position: top right;}
```

В результате исходное изображение будет помещаться в верхний правый угол отступа каждого абзаца и не будет повторяться. Между прочим, результат, приведенный на рис. 9.22, был бы совершенно аналогичным, если бы местоположение было объявлено как `right top`. Ключевые слова (согласно спецификации) могут располагаться в любом

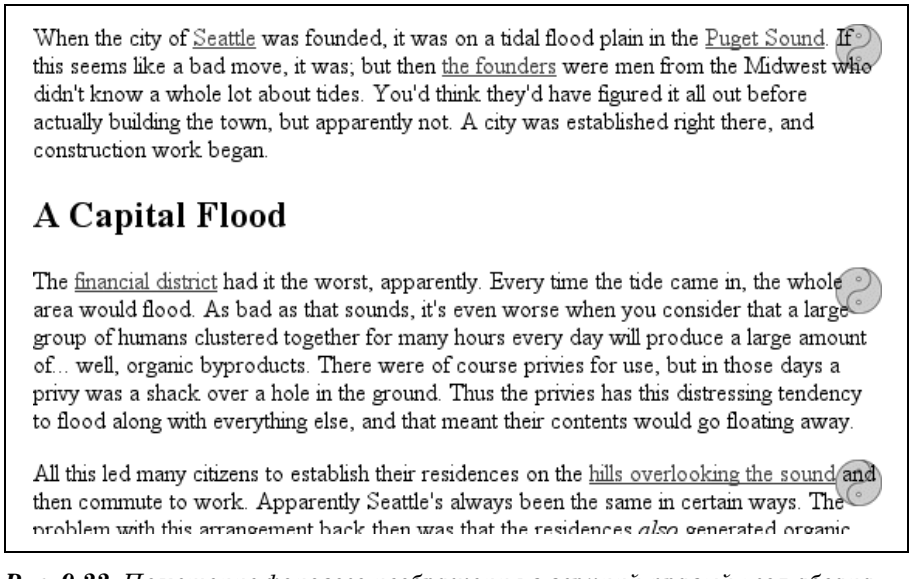


Рис. 9.22. *Помещение фонового изображения в верхний правый угол абзаца*

порядке, но их должно быть не более двух: одно – горизонтальная координата, другое – вертикальная.

В броузерах Netscape 6.x есть ошибка, из-за которой правило игнорируется, если ключевые слова background-position не расположены в определенном порядке. Убедитесь, что ключевые слова задают сначала положение по горизонтали, а затем по вертикали. Другими словами, необходимо писать left center, а не center left.

Если задано только одно ключевое слово, то предполагается, что другое слово – center. В табл. 9.1 показаны эквивалентные выражения.

Таблица 9.1. Эквиваленты ключевых слов, задающих размещение

Одно ключевое слово	Эквивалентные ключевые слова
center	center center
top	top center center top
bottom	bottom center center bottom
right	center right right center
left	center left left center

Итак, если требуется, чтобы изображение располагалось над каждым абзацем посередине, достаточно объявить:

```
p {background-image: url(yinyang.gif);
   background-repeat: no-repeat;
   background-position: top;}
```

Процентные значения

Процентные значения очень похожи на ключевые слова, хотя ведут себя немного сложнее. Пусть требуется центрировать базовую точку изображения в элементе, задавая процентные значения. Это довольно просто:

```
p {background-image: url(bigyinyang.gif);
   background-repeat: no-repeat;
   background-position: 50% 50%;}
```

При этом базовая точка изображения будет размещена так, что ее центр будет выровнен с центром ее элемента. Иначе говоря, процентные значения применяются и к элементу, и к исходному изображению.

Рассмотрим процесс более подробно. Когда исходное изображение центрируется в элементе, точка изображения, которая может быть описана как 50% 50% (центр), выравнивается с точкой элемента, которая мо-

жет быть описана аналогичным образом. Если изображение размещается в точке 0% 0%, то его верхний левый угол находится в верхнем левом углу области отступа элемента. Значение 100% 100% соответствует положению нижнего правого угла исходного изображения в нижнем правом углу области отступа:

```
p {background-image: url(oransqr.gif);
   background-repeat: no-repeat;
   padding: 5px;  border: 1px dotted gray;}
p.c1 {background-position: 0% 0%;}
p.c2 {background-position: 50% 50%;}
p.c3 {background-position: 100% 100%;}
p.c4 {background-position: 0% 100%;}
p.c5 {background-position: 100% 0%;}
```

Эти правила иллюстрирует рис. 9.23.

Таким образом, если вы хотите поместить одно исходное изображение на расстоянии трети ширины элемента и двух третей его высоты, ваше объявление может быть таким:

```
p {background-image: url(bigyinyang.gif);
   background-repeat: no-repeat;
   background-position: 33% 66%;}
```

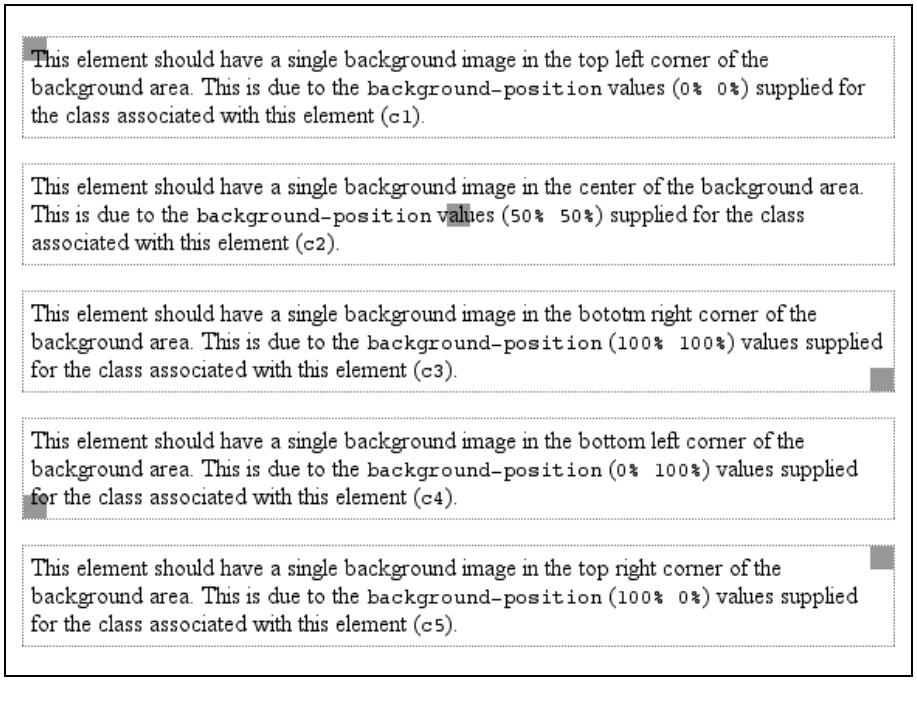


Рис. 9.23. *Различные местоположения, заданные с помощью процентных соотношений*

В соответствии с этими правилами точка исходного изображения, отстоящая на треть ширины и две трети высоты от верхнего левого угла изображения, будет выровнена с наиболее удаленной от верхнего левого угла элемента-контейнера точкой. Заметьте, что в процентных значениях горизонтальная координата *всегда* первая. Если в предыдущем примере поменять местами процентные значения, изображение окажется в точке, отстоящей на две трети ширины элемента и треть высоты.

Если задано одно процентное значение, то оно используется как значение по горизонтали, а значение по вертикали принимается равным `50%`. С ключевыми словами ситуация аналогичная: если задано только одно слово, то предполагается, что другое — `center`. Например:

```
p {background-image: url(yinyang.gif);
   background-repeat: no-repeat;
   background-position: 25%;}
```

Исходное изображение размещается на расстоянии четверти ширины области содержимого элемента и области отступа и половины их высоты, как показано на рис. 9.24.

В табл. 9.2 приведена схема соответствия ключевых слов и процентных значений.

Таблица 9.2. Позиционные эквиваленты

Одно ключевое слово	Эквивалентные ключевые слова	Эквивалентные процентные значения
`center`	`center center`	`50% 50%` `50%`
`top`	`top center` `center top`	`50% 0%`
`bottom`	`bottom center` `center bottom`	`50% 100%`
`right`	`center right` `right center`	`100% 50%` `100%`
`left`	`center left` `left center`	`0% 50%` `0%`
	`top left` `left top`	`0% 0%`
	`top right` `right top`	`100% 0%`
	`bottom right` `right bottom`	`100% 100%`
	`bottom left` `left bottom`	`0% 100%`

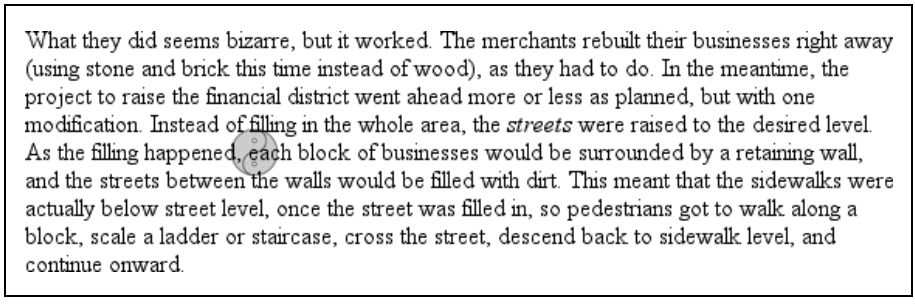


Рис. 9.24. *Объявление только одного значения процентного соотношения означает, что положение по вертикали соответствует 50%*

По умолчанию для свойства `background-position` устанавливается значение `0% 0%`, что в функциональном плане аналогично `top left`. Вот почему, если не задать разные значения для определения местоположения, мозаичное размещение фоновых изображений всегда начинается с верхнего левого угла области отступа элемента.

Значения, заданные в единицах измерения длины

И наконец, обратимся к позиционированию с помощью значений, заданных в единицах измерения длины. Если местоположение исходного изображения задано в единицах длины, то они интерпретируются как смещения от верхнего левого угла области отступа элемента. Точка, относительно которой отсчитываются смещения, – верхний левый угол исходного изображения. Таким образом, если заданы значения `20px 30px`, верхний левый угол исходного изображения будет сдвинут на 20 пикселов вправо и 30 пикселов вниз относительно верхнего левого угла области отступа элемента, как показано на рис. 9.25:

```
p {background-image: url(yinyang.gif);
   background-repeat: no-repeat;
   background-position: 20px 30px;
   border: 1px dotted gray;}
```

В случае применения процентных значений все не так, поскольку происходит смещение одного верхнего левого угла относительно другого.

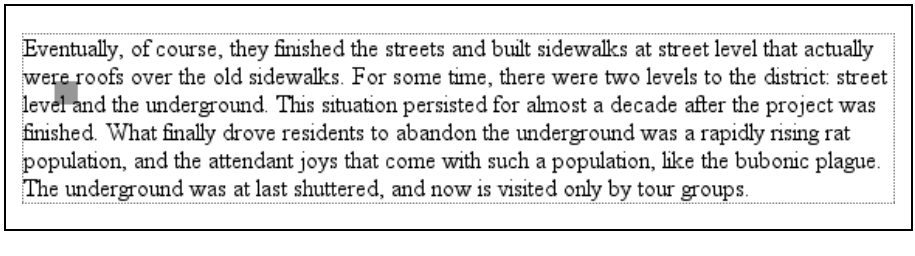


Рис. 9.25. *Смещение фонового изображения на определенное расстояние*

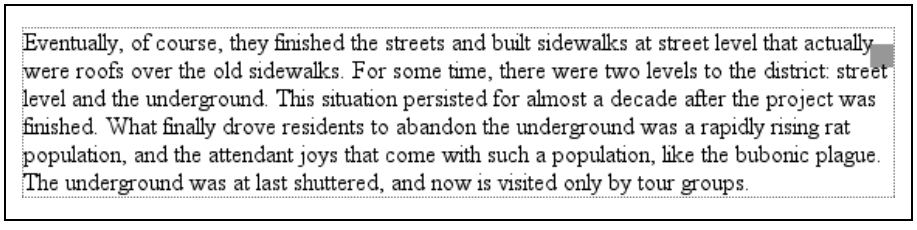


Рис. 9.26. *Смешение процентных соотношений и значений, заданных в единицах длины*

Иначе говоря, верхний левый угол исходного изображения совмещается с точкой, заданной свойством `background-position`. Однако можно комбинировать значения, заданные в единицах длины, с процентными значениями, чтобы воспользоваться преимуществами обоих подходов. Скажем, требуется, чтобы фоновое изображение всегда находилось в правой части элемента на 10 пикселов ниже верхней границы, как показано на рис. 9.26. Как обычно, значение по горизонтали расположено первым:

```
p {background-image: url(bg23.gif);
   background-repeat: no-repeat;
   background-position: 100% 10px;
   border: 1px dotted gray;}
```

В версиях CSS до 2.1 *нельзя* было сочетать ключевые слова с другими значениями. Таким образом, значение `top 75%` не было допустимым, и если указывалось ключевое слово, то приходилось везде придерживаться только ключевых слов. CSS2.1 изменил это, чтобы упростить создание таблиц стилей и чтобы не отставать от других броузеров, в которых это уже разрешено.

Значения длины или процентные значения могут быть отрицательными, благодаря чему можно поместить исходное изображение вне области фона элемента. Рассмотрим пример с применением в качестве фона очень большого символа инь-ян. Мы уже центрировали его, но теперь нам надо, чтобы в верхнем левом углу области отступа элемента была видна только его часть. Нет проблем, по крайней мере, теоретически.

Во-первых, предположим, что исходное изображение имеет 300 пикселов в высоту и 300 пикселов в ширину. Затем примем, что видимой должна быть лишь его одна треть снизу справа. Желаемого эффекта (показанного на рис. 9.27) можно достичь вот так:

```
p {background-image: url(bigyinyang.gif);
   background-repeat: no-repeat;
   background-position: -200px -200px;
   border: 1px dotted gray;}
```

Или, скажем, требуется, чтобы видимой и центрированной в элементе была лишь его правая половина:

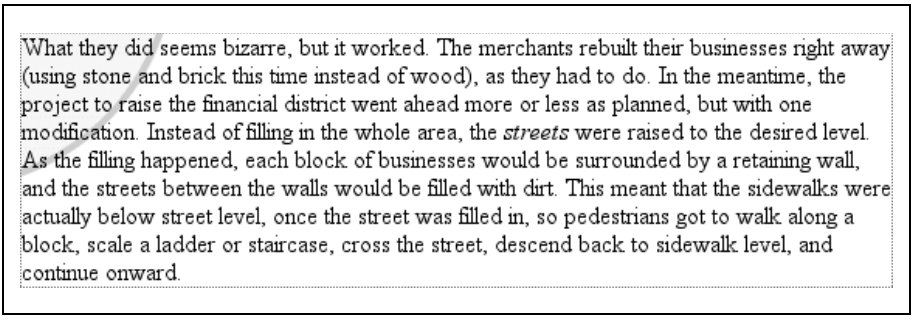


Рис. 9.27. *Использование отрицательных значений длины при размещении исходного изображения*

```
p {background-image: url(bigyinyang.gif);
   background-repeat: no-repeat;
   background-position: -150px 50%;
   border: 1px dotted gray;}
```

Теоретически возможны также и отрицательные значения процентных значений, хотя тут есть две трудности. Первая – ограничения агентов пользователя, которые могут не распознавать отрицательные значения `background-position`. Вторая – отрицательные процентные значения вычисляются несколько странно. Начнем с того, что размеры исходного изображения и элемента, скорее всего, сильно отличаются, и это может привести к неожиданным результатам. Рассмотрим, например, ситуацию, созданную следующим правилом и проиллюстрированную на рис. 9.28:

```
p {background-image: url(pix/bigyinyang.gif);
   background-repeat: no-repeat;
   background-position: -10% -10%;
   border: 1px dotted gray;
   width: 500px;}
```

Правило предусматривает, что точка, находящаяся вне границ изображения, определяемая значением `-10% -10%`, должна быть выровнена с аналогичной точкой абзаца. Размер изображения составляет 300× 300 пикселов, и мы знаем, что точка, по которой будет выравниваться изображение, расположена на 30 пикселов выше верха изображения и на 30 пикселов влево от его левого края (фактически `-30px` и `-30px`). Все абзацы имеют одинаковую ширину (`500px`), поэтому точка горизонтального выравнивания отстоит на 50 пикселов от левого края их областей отступа. Это значит, что левый край каждого изображения будет смещен на 20 пикселов влево от края левого отступа абзацев. Дело в том, что точка выравнивания изображений (`-30px`) выравнивается с точкой с координатой `-50px` абзаца. Разница между этими двумя точками составляет 20 пикселов.

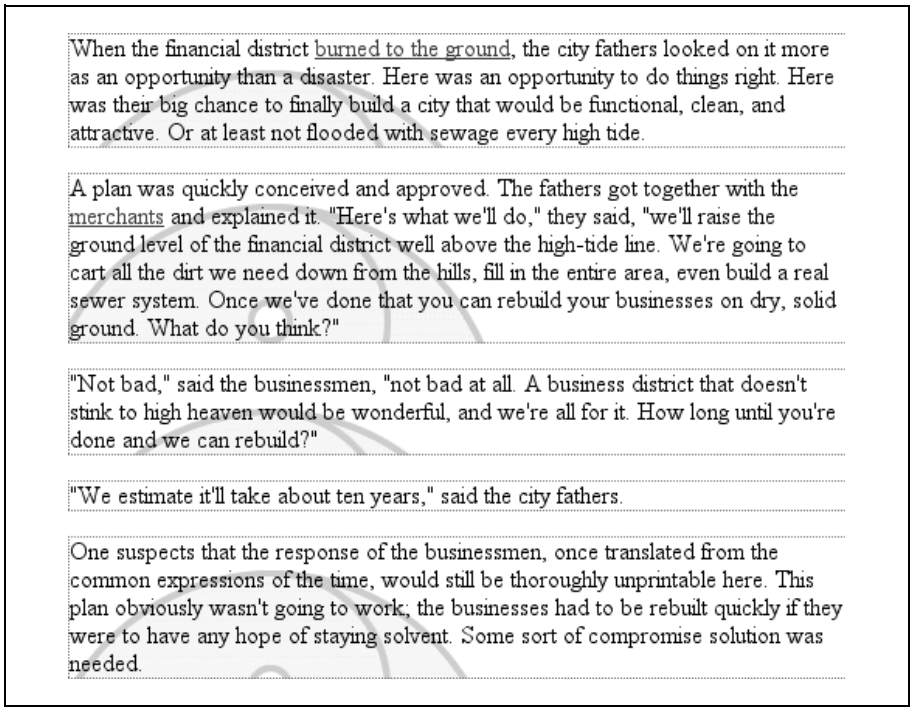


Рис. 9.28. *Различные эффекты применения отрицательных процентных значений*

Однако высота абзацев разная, поэтому точка вертикального выравнивания меняется для каждого абзаца. Если высота абзаца равна 300 пикселам, то верх исходного изображения будет выровнен точно по верху области отступа элемента, потому что оба будут иметь точки вертикального выравнивания с координатой `-30px`. Если высота абзаца равна 50 пикселам, то координата его точки выравнивания была бы `-5px`, и верх исходного изображения был бы на самом деле на 25 пикселов *ниже* верха области отступа.

Такие же проблемы могут возникнуть при задании положительных процентных значений. Представьте, что произошло бы, если бы вы выровняли исходное изображение по низу элемента, ширина которого меньше ширины изображения, хотя я не говорю, что вы не должны использовать отрицательные значения. Это просто напоминание, что, как всегда, есть вопросы, на которые следует обратить внимание.

Во всех примерах этого раздела значением свойства `background-repeat` было `no-repeat`. Причина проста: при наличии лишь одного фонового изображения намного проще увидеть, как позиционирование влияет на размещение первого фонового изображения. Однако не надо препятствовать повторениям фонового изображения:

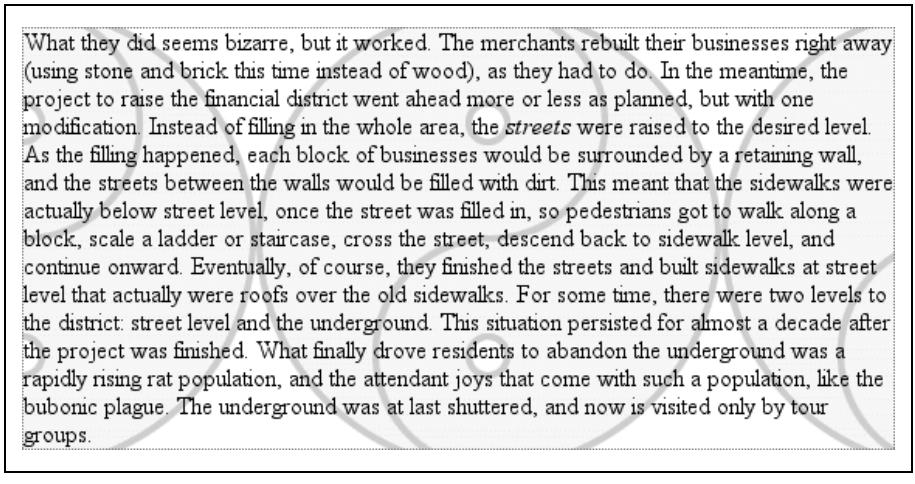


Рис. 9.29. *Применение свойства background-position задает начало мозаичного шаблона*

```
p {background-image: url(bigyinyang.gif);
   background-position: -150px 50%;
   border: 1px dotted gray;}
```

Итак, когда фон повторяется, что можно видеть на рис. 9.29, начало мозаичного шаблона задается свойством `background-position`.

Это еще раз иллюстрирует понятие исходного изображения, очень важное для усвоения следующего раздела.

Еще раз о повторении в определенном направлении

В прошлый раз мы рассматривали значения `repeat-x`, `repeat-y` и `repeat` и влияние этих значений на мозаичное размещение фоновых изображений. Однако во всех этих случаях мозаичный шаблон начинался в верхнем левом углу элемента-контейнера (например, `p`). Конечно, это не обязательное требование; как мы уже видели, по умолчанию свойству `background-position` присваиваются значения `0% 0%`. Поэтому, если не менять местоположение исходного изображения, мозаичный шаблон начнется именно там, где вы указали. Итак, вы знаете, как изменить положение исходного изображения, и пора выяснить, как в этой ситуации будут себя вести агенты пользователя.

Проще всего привести пример и затем объяснить его. Рассмотрим следующую разметку, которая проиллюстрирована на рис. 9.30:

```
p {background-image: url(yinyang.gif);
   background-position: center;
   border: 1px dotted gray;}
p.c1 {background-repeat: repeat-y;}
p.c2 {background-repeat: repeat-x;}
```

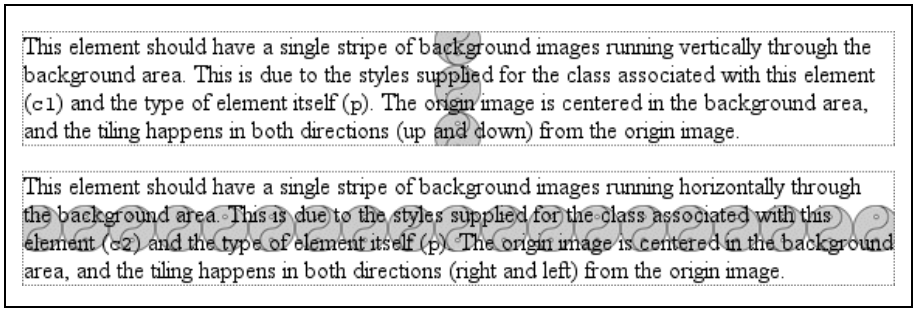


Рис. 9.30. *Центрирование исходного изображения и его повторение*

Вот мы и получили изображения, проходящие полосами через центры элементов. Может показаться, что это неправильно, но это не так.

Примеры, показанные на рис. 9.30, верны, потому что исходное изображение было помещено в центр первого элемента `p` и затем размножено вдоль оси *y* в *обоих направлениях*, т. е. *и вверх*, *и вниз*. Во втором абзаце изображения повторяются влево и вправо.

Следовательно, если поместить большое изображение в центр `p`, а затем позволить ему повторяться до заполнения всего элемента-контейнера, то оно будет продублировано во всех *четырех* направлениях: вверх, вниз, влево и вправо. Свойство `background-position` указывает только место, в котором начинается шаблон мозаики. На рис. 9.31 показана разница между мозаичным шаблоном, начинающимся из центра элемента и из его верхнего левого угла.

Обратите внимание на различия вдоль краев элементов. Когда фон повторяется из центра, как в первом абзаце, сетка символов инь-ян центрирована, в результате чего символы по обоим краям элемента отсекаются симметрично. Во втором абзаце мозаичный шаблон начинается в верхнем левом углу области отступа, поэтому изображения обрезаются только снизу элемента. Разница может показаться небольшой, но полезными могут оказаться оба подхода.

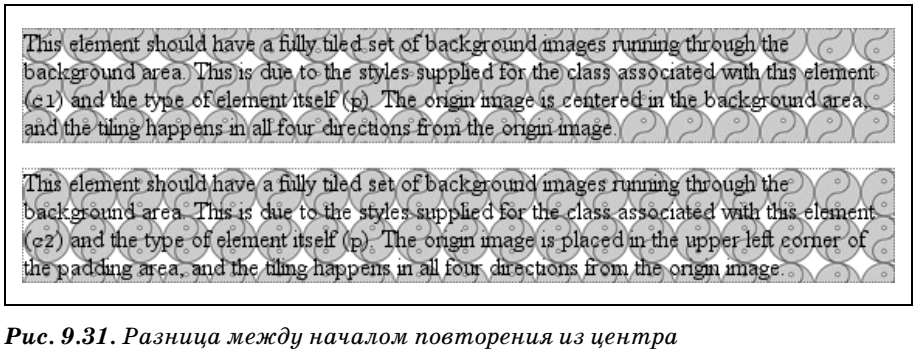


Рис. 9.31. *Разница между началом повторения из центра и из верхнего левого угла*

Предупреждая возможные вопросы, скажу, что нет никаких иных способов, позволяющих управлять повторением изображений, кроме тех, что мы уже обсудили. Например, нет значения `repeat-left`, хотя подобные значения могут быть добавлены в какой-нибудь будущей версии CSS. На данный момент реализовано заполнение шаблоном, мозаичное размещение в горизонтальном и вертикальном направлениях или полное отсутствие мозаичного размещения.

Позиционирование относительно окна

Итак, вы умеете размещать исходное изображение в любом месте фона элемента и можете управлять (в некоторой степени) его мозаичным размещением. Однако, как вы уже, вероятно, поняли, размещение изображения в центре элемента `body`, если документ достаточно велик, может означать, что фоновое изображение не будет видно читателю сразу. Ведь броузер предоставляет для документа только окно. Если документ слишком длинный и не помещается в окне полностью, пользователь может прокручивать его вперед и назад. Центр может располагаться на два или три «окна» ниже начала документа или просто достаточно далеко, тогда большая часть исходного изображения будет находиться за пределами окна броузера.

Более того, даже если исходное изображение изначально видно, оно всегда прокручивается вместе с документом и исчезнет, когда пользователь прокрутит документ дальше. Но и это препятствие можно обойти.

Свойство `background-attachment` позволяет зафиксировать исходное изображение относительно области просмотра и, следовательно, защититься от эффекта прокрутки:

```
body {background-image: url(bigyinyang.gif);
   background-repeat: no-repeat;
   background-position: center;
   background-attachment: fixed;}
```

Достигнутые эффекты показаны на рис. 9.32. Первый – исходное изображение не прокручивается вместе с документом. Второй – местоположение исходного изображения определяется размером области просмотра, а не размером (или размещением в области просмотра) элемента, в котором оно находится.

background-attachment

Значения:	`scroll | fixed | inherit`
Начальное значение:	`scroll`
Область применения:	все элементы
Наследование:	нет
Вычисляемое значение:	как задано

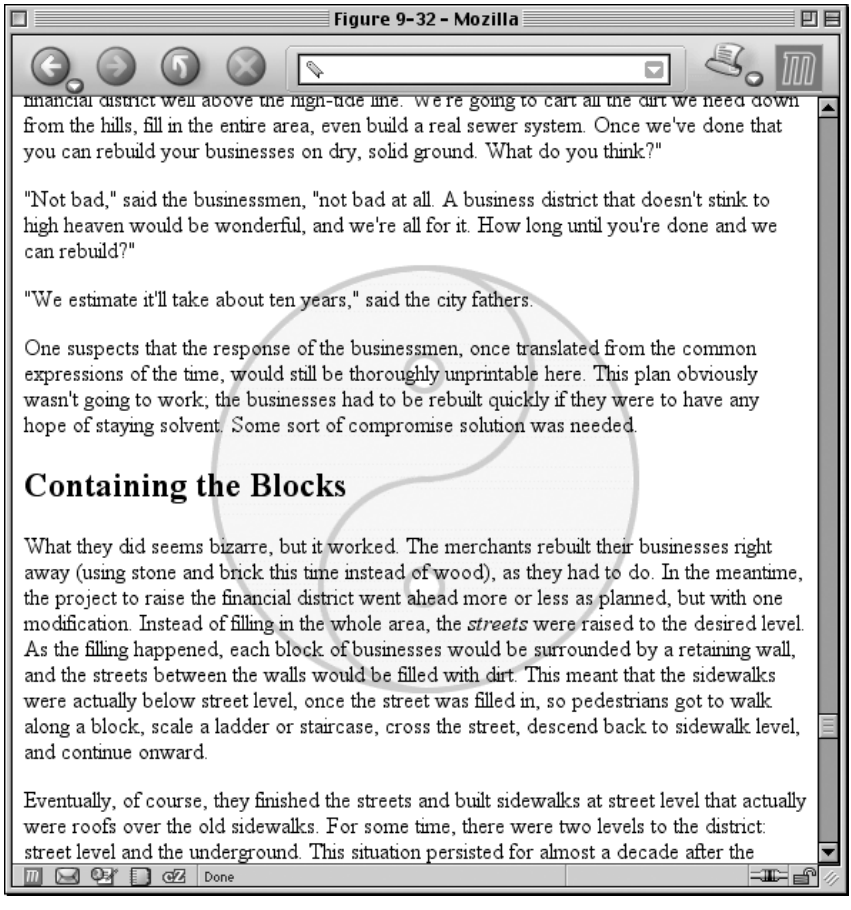


Рис. 9.32. *Закрепление фона*

В веб-броузере пользователь может задавать область просмотра, изменяя размеры окна броузера. При изменении размера окна исходная позиция фонового изображения меняется. На рис. 9.33 показано несколько представлений одного и того же документа. Итак, в некотором смысле местоположение изображения не зафиксировано, но оно остается фиксированным, пока область просмотра не меняет своих размеров.

Осталось еще одно значение свойства `background-attachment` – применяемое по умолчанию значение `scroll`. Как и ожидается, оно заставляет фон прокручиваться вместе с остальным документом при просмотре в веб-броузере и не обязательно меняет положение исходного изображения при изменении размеров окна. Если ширина документа фиксирована (возможно, явно задана ширина элемента `body`), то изменение области просмотра вообще не повлияет на размещение прокручиваемого исходного изображения.

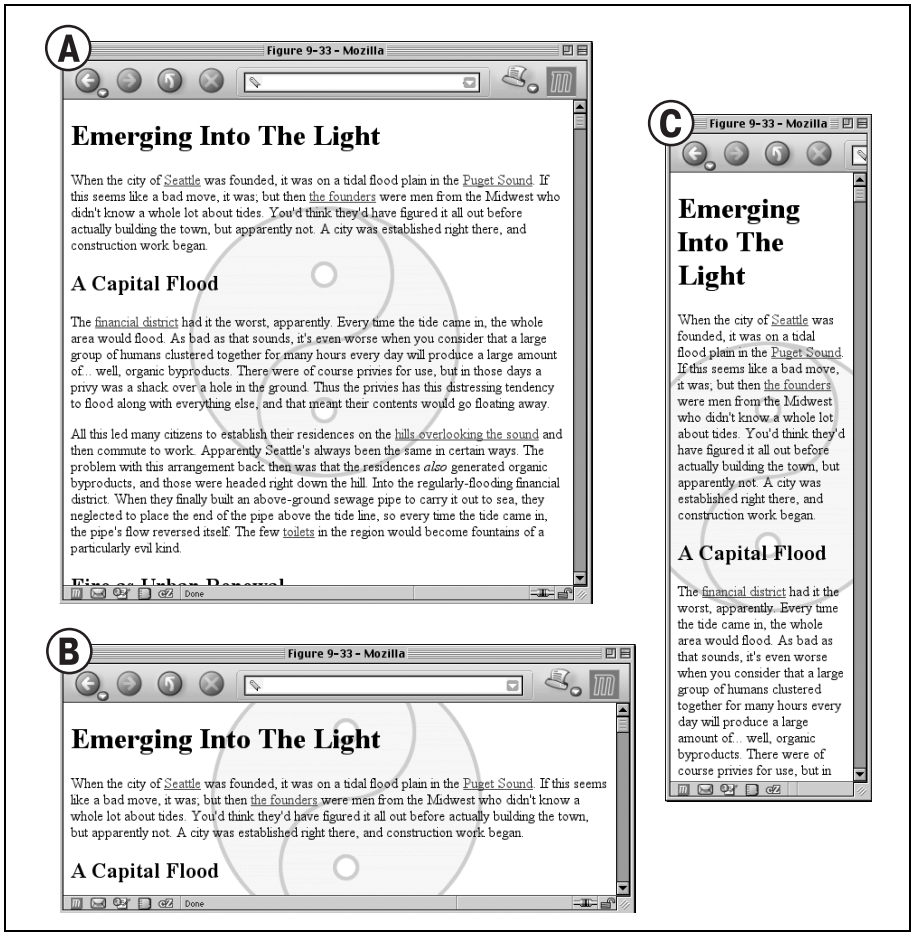


Рис. 9.33. *Сохранение центрирования*

Интересные эффекты

С технической точки зрения, когда фоновое изображение зафиксировано (`fixed`), оно размещается с учетом области просмотра, а не элемента, содержащего его. Однако фон будет видимым только в рамках его элемента-контейнера. Это ведет к весьма интересным последствиям.

Скажем, имеется документ с мозаичным фоном, который действительно выглядит как мозаика, и элемент `h1` с таким же шаблоном, но другого цвета. И у элемента `body`, и у элемента `h1` фоновое изображение зафиксировано (`fixed`), что в результате обеспечивает примерно следующее (рис. 9.34):

```
body {background-image: url(grid1.gif); background-repeat: repeat;
```

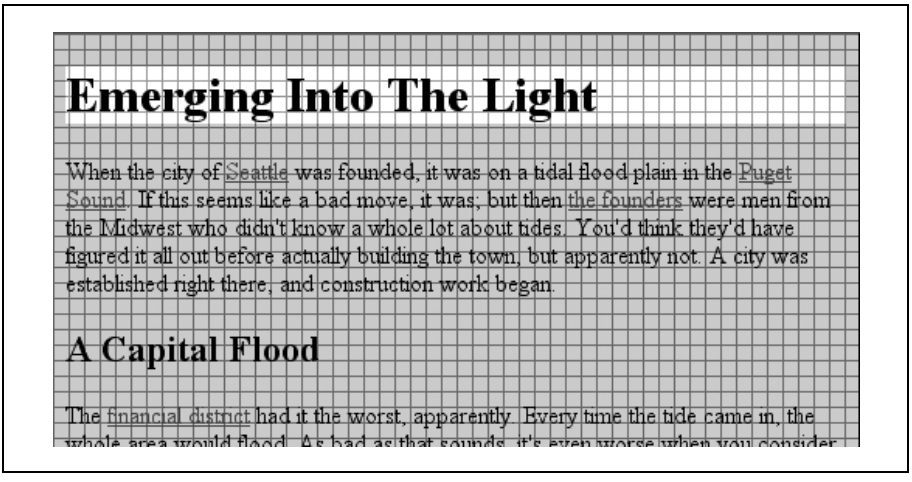


Рис. 9.34. *Идеальное выравнивание фона*

```
    background-attachment: fixed;}
h1 {background-image: url(grid2.gif); background-repeat: repeat;
    background-attachment: fixed;}
```

Как достигается такое идеальное выравнивание? Помните: если фон зафиксирован (`fixed`), исходный элемент размещается относительно окна просмотра. Таким образом, оба шаблона фона начинаются от верхнего левого угла окна просмотра, а не относительно отдельных элементов. Для `body` виден повторяющийся шаблон. Фон для элемента `h1`, однако, виден только в области отступов и содержимого самого `h1`. Поскольку оба фоновых изображения имеют один размер и начинаются в одной точке, получается что они «выровнялись», как показано на рис. 9.34.

Эта возможность помогает создавать ряд очень сложных эффектов. Один из самых известных примеров – демонстрация «искажения сложной спирали» (*http://www.meyerweb.com/eric/css/edge/complexspiral/glassy.html*), которая проиллюстрирована на рис. 9.35.

Визуальные эффекты обусловлены присваиванием различных фиксированных фоновых изображений элементам, не являющимся элементами `body`. Вся демонстрационная версия состоит из одного HTML-документа, четырех JPEG-изображений и таблицы стилей. Поскольку все четыре изображения размещены в верхнем левом углу окна броузера, но видимы только там, где пересекаются со своими элементами, изображения эффектно чередуются, создавая иллюзию полупрозрачного волнистого стекла.

Internet Explorer для Windows вплоть до IE6 неправильно обрабатывает фиксированные фоны в отличных от `body` элементах. Иначе говоря, ожидаемый эффект достигается только для фиксированного фона элемента `body`, но не для других элементов.

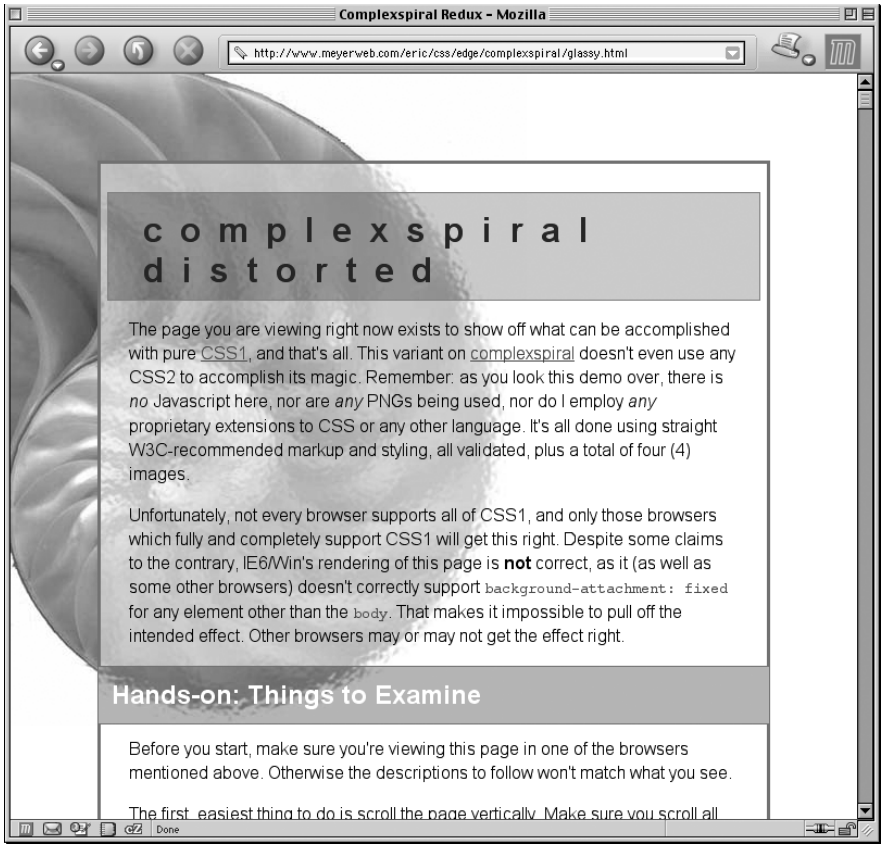


Рис. 9.35. *Искажение сложной спирали*

Это разрушает эффекты, показанные на рис. 9.34 и 9.35. Internet Explorer 7 поддерживает фиксированные фоны для всех элементов.

Кроме того, в устройствах со страничной организацией, таких как печатающие устройства, каждая страница генерирует собственное окно просмотра. Следовательно, фиксированный фон должен появляться на каждой странице печатающего устройства. Это можно использовать для создания таких эффектов, как нанесение водяных знаков на все страницы документа. Проблемы удваиваются: с помощью CSS невозможно организовать распечатку фоновых изображений, и не все броузеры правильно обрабатывают печать фиксированных фонов.

Сведение воедино

Как и свойства шрифтов, все свойства фона можно свести в одну сокращенную форму записи: `background`. Это свойство может принимать по

background	
Значения:	[<background-color> || <background-image> || <background-repeat> || <background-attachment> || <background-position>] | inherit
Начальное значение:	см. отдельные свойства
Область применения:	все элементы
Наследование:	нет
Процентные соотношения:	значения разрешены для свойства <background-position>
Вычисляемое значение:	см. отдельные свойства

одному значению от каждого из свойств фона практически в любом порядке.

Таким образом, все следующие выражения эквивалентны, и их применение будет иметь результат, показанный на рис. 9.36:

```
body {background-color: white; background-image: url(yinyang.gif);
   background-position: top left; background-repeat: repeat-y;
   background-attachment: fixed;}
body {background: white url(yinyang.gif) top left repeat-y fixed;}
body {background: fixed url(yinyang.gif) white top left repeat-y;}
body {background: url(yinyang.gif) white repeat-y fixed top left;}
```

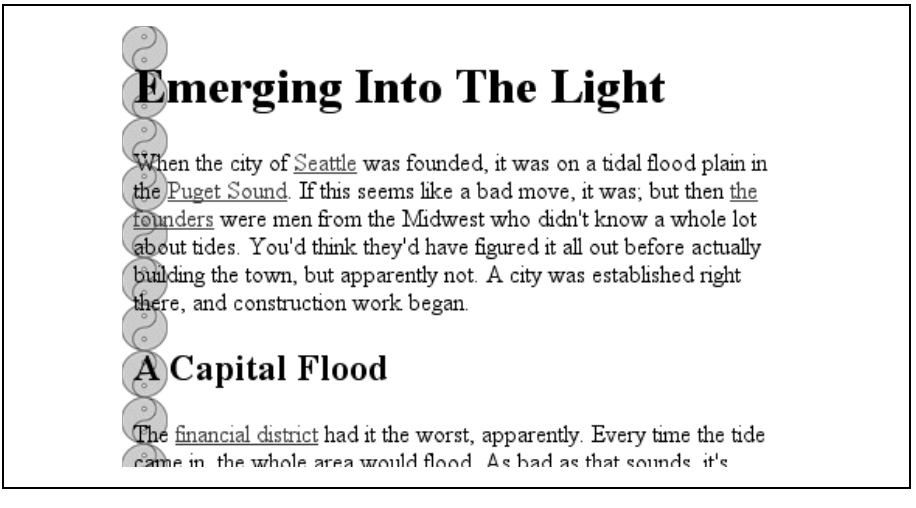


Рис. 9.36. *Применение сокращенной формы записи свойства*

Существует тем не менее одно небольшое ограничение на порядок расположения значений в свойстве `background`: если для `background-position` задаются два значения, они должны стоять рядом, и если приме-

няются значения длины или процентные значения, сначала должно идти значение по горизонтали, а затем по вертикали. Наверное, в этом нет ничего неожиданного, но об этом важно помнить.

Как и во всех сокращенных формах записи свойств, если опустить одно из значений, вместо него автоматически подставляется значение соответствующего свойства по умолчанию. Таким образом, следующие два правила эквивалентны:

```
body {background: white url(yinyang.gif;}
body {background: white url(yinyang.gif) top left repeat scroll;}
```

Что замечательно, свойство background не имеет обязательных значений: если присутствует по крайней мере одно значение, можно опустить все остальные. Следовательно, с помощью сокращенной формы записи можно задавать только фоновый цвет, и так делают очень часто:

```
body {background: white;}
```

Это совершенно законно, и в некоторых случаях такой вариант следует предпочесть как позволяющий уменьшить количество нажатий на клавиши. Кроме того, в результате все остальные свойства фона задаются по умолчанию, следовательно, для background-image будет задано значение none. Это помогает обеспечить удобочитаемость, поскольку предотвращает задание фонового изображения другими правилами (например, в таблице стилей читателя).

Любое из следующих правил также вполне допустимо, что показано на рис. 9.37:

```
body {background: url(yinyang.gif) bottom left repeat-y;}
h1 {background: silver;}
h2 {background: url(h2bg.gif) center repeat-x;}
p {background: url(parabg.gif);}
```

Еще одно, последнее напоминание: background – сокращенная форма записи, и значения по умолчанию могут заменять значения, ранее заданные для данного элемента. Например:

```
h1, h2 {background: gray url(thetrees.jpg) center repeat-x;}
h2 {background: silver;}
```

В результате применения этих правил элементы h1 будут оформлены в соответствии с первым из них, а элементы h2 – в соответствии со вторым. Другими словами, у них будет просто сплошной серебряный фон. На заднем плане h2 вообще не будет никакого изображения, не говоря уже о центрировании и повторении в горизонтальном направлении. Вероятнее всего, автор намеревался сделать следующее:

```
h1, h2 {background: gray url(trees.jpg) center repeat-x;}
h2 {background-color: silver;}
```

Это позволяет изменять фоновый цвет, не уничтожая все остальные значения.

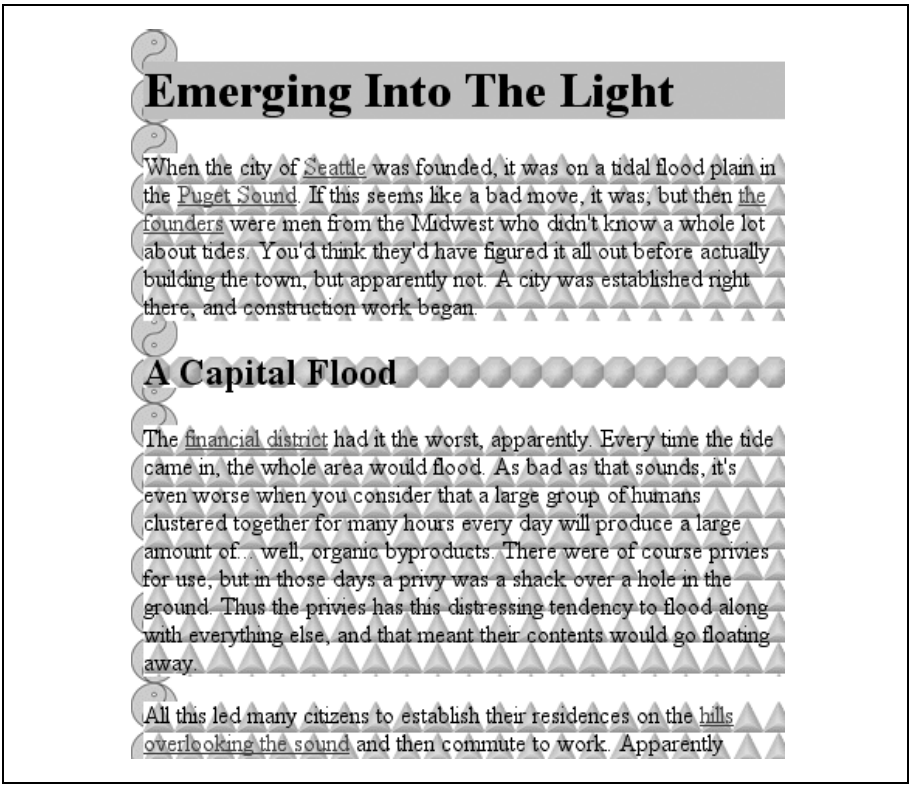


Рис. 9.37. *Применение нескольких фонов в одном документе*

Заключение

Задание цвета и фона для элементов – очень мощный инструмент. Преимущество CSS над традиционными методами в том, что цвет и фон могут применяться к любому элементу документа, а не только к ячейкам таблицы или к тому, что заключено в тег `FONT`. Несмотря на незначительные ошибки в некоторых реализациях, такие как нежелание Navigator 4 применять фон ко всей области содержимого элемента, назначения фона применяются очень широко. Их популярность нетрудно понять, поскольку цвет – простейший способ сделать одну страницу непохожей на другую.

Однако CSS предлагает еще более широкие возможности стилевого оформления элемента: рамки, которые могут быть применены к любому элементу, дополнительные поля и отступы и даже возможность «обтекания» элементов, не являющихся изображениями. Эти понятия рассмотрены в следующей главе.

10

Свободное перемещение и позиционирование

Конечно, CSS преображает содержимое, изменяя шрифты, фон и прочее, но как обстоит дело с выполнением основных задач верстки? Рассмотрим понятия *свободное перемещение* (*floating*) и *позиционирование* (*positioning*). Это инструменты, позволяющие обеспечить размещение содержимого в виде столбцов, наложение одной части макета на другую и вообще реализовать все то, для чего в течение многих лет применялись таблицы.

Идея, лежащая в основе позиционирования, довольно проста. Оно позволяет указать, где появятся блоки элементов относительно того, где они оказались бы в обычном порядке: или относительно родительского элемента, другого элемента, или даже относительно самого окна броузера. Мощь этой возможности одновременно и очевидна, и удивительна. И совершенно естественно, что агенты пользователя поддерживают этот элемент CSS2 лучше, чем многие другие.

Свободное перемещение было впервые предложено в CSS1 и основывается на возможности, введенной Netscape на заре развития Всемирной паутины. Перемещение – это не совсем позиционирование, но и, конечно, не верстка в нормальном потоке. Что конкретно оно означает, мы увидим позже в этой главе.

Свободное перемещение

Вы почти наверняка знакомы с концепцией перемещаемых элементов. Еще со времен Netscape 1 существовала возможность перемещать изображения, объявляя, например, `<img src="b5.gif" align="right">`. При этом изображение будет перемещено вправо, а остальное содержимое (такое как текст) будет «обтекать» его. Термин «обтекание»,

float	
Значения:	`left | right | none | inherit`
Начальное значение:	`none`
Область применения:	все элементы
Наследование:	нет
Вычисляемое значение:	как задано

кстати, пришел из документа «Extensions to HTML 2.0» (Расширения HTML 2.0), который гласит:

> Дополнения к параметру ALIGN нуждаются в подробном объяснении. Во-первых, значения `left` и `right`. Изображения с таким выравниванием – это совершенно новый *обтекаемый* тип изображений.

В прошлом можно было свободно перемещать только изображения, а в некоторых броузерах – и таблицы. CSS же позволяет перемещать любой элемент – от изображений до абзацев и списков. В CSS это поведение реализовано с помощью свойства `float`.

Так, результатом применения следующей разметки будет перемещение изображения влево:

```
<img src="b4.gif" style="float: left;" alt="b4">
```

Как видно на рис. 10.1, изображение перемещается в левую часть окна броузера, и текст обтекает его. Именно этого надо ожидать.

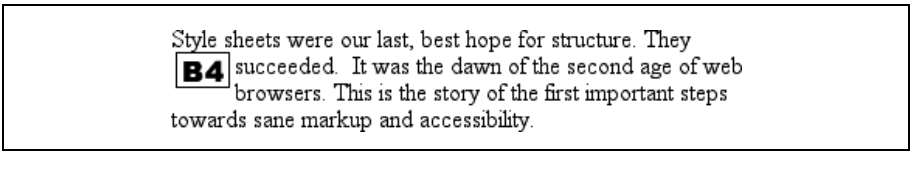


***Рис. 10.1.** Перемещаемое изображение*

Однако при реализации в CSS перемещения элементов возникают некоторые интересные вопросы.

Перемещаемые элементы

О перемещении элементов необходимо помнить несколько моментов. Прежде всего, перемещаемый элемент некоторым образом удален из нормального потока документа, хотя все равно влияет на компоновку. Это совершенно уникально для CSS: перемещаемые изображения существуют практически в своей собственной плоскости, но все-таки влияют на остальной документ.

Это влияние обусловлено тем фактом, что остальное содержимое «обтекает» свободно перемещаемый элемент. Так ведут себя перемещае-

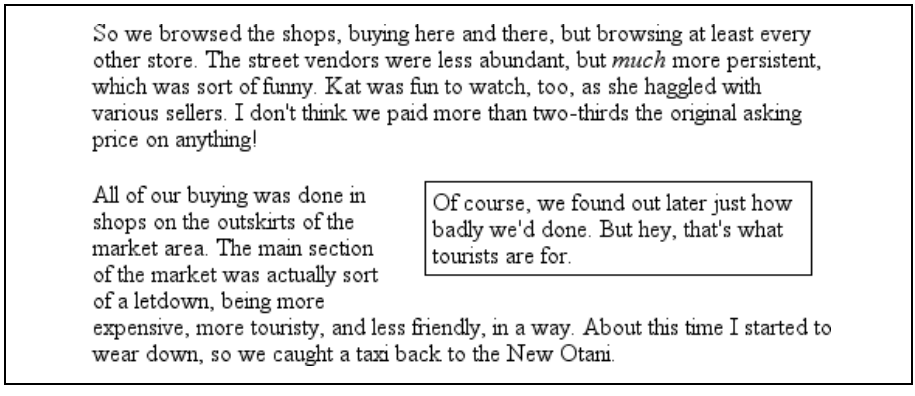


Рис. 10.2. *Перемещаемый параграф*

мые изображения, но то же происходит и при перемещении, скажем, абзаца. На рис. 10.2 этот эффект хорошо виден благодаря полю, добавленному в перемещаемый абзац:

```
p.aside {float: right; width: 15em; margin: 0 1em 1em; padding: 0.25em;
  border: 1px solid;}
```

Прежде всего следует отметить, что поля вокруг перемещаемых элементов не сворачиваются. Если изображение окружить 20-пиксельным полем, то вокруг изображения останется минимум 20 пикселов пустого пространства. Если расположенные рядом с изображением другие элементы – соседи по вертикали *и* горизонтали – также имеют поля, то эти поля не будут поглощаться полями перемещаемого изображения, как видно на рис. 10.3:

```
p img {float: left; margin: 25px;}
```

Вспомним аналогию с листком бумаги, имеющим прозрачные поля, из главы 7: поля вокруг изображения *никогда* не перекрывают поля, окружающие другие перемещаемые элементы.

Перемещая незамещаемый элемент, необходимо объявить его ширину. В противном случае согласно спецификации CSS ширина элемента будет равна нулю. Таким образом, ширина перемещаемого абзаца была бы равной одному символу (если один символ – это минимальное значение, предоставляемое броузером для свойства `width`). Если не объявить значение свойства `width` для перемещаемого элемента, то в результате можно получить нечто похожее на рис. 10.4. (Слава богу, это маловероятно, но тем не менее возможно.)

Запрет перемещения

Кроме значений `left` и `right`, свойство `float` имеет еще одно, `float: none`, которое применяется, чтобы вообще запретить перемещение элемента.

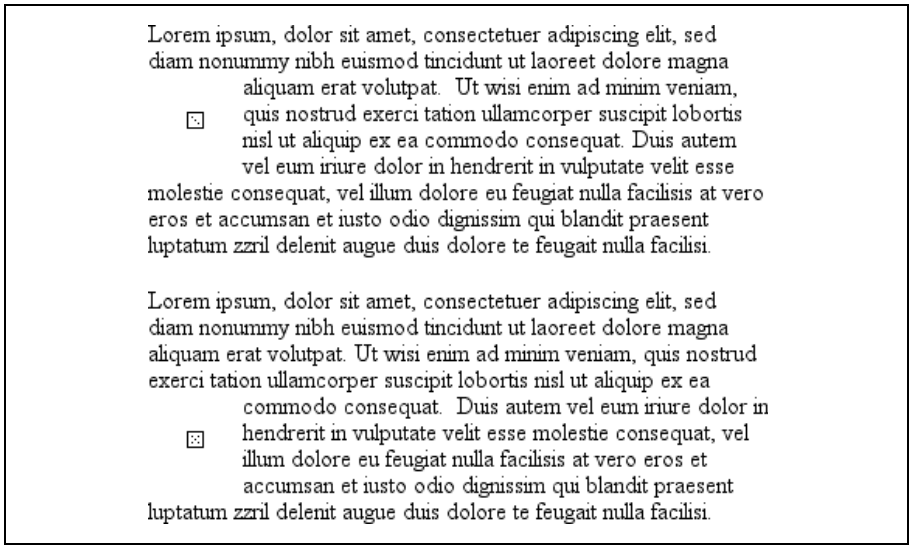


Рис. 10.3. *Перемещаемые изображения, имеющие поля*

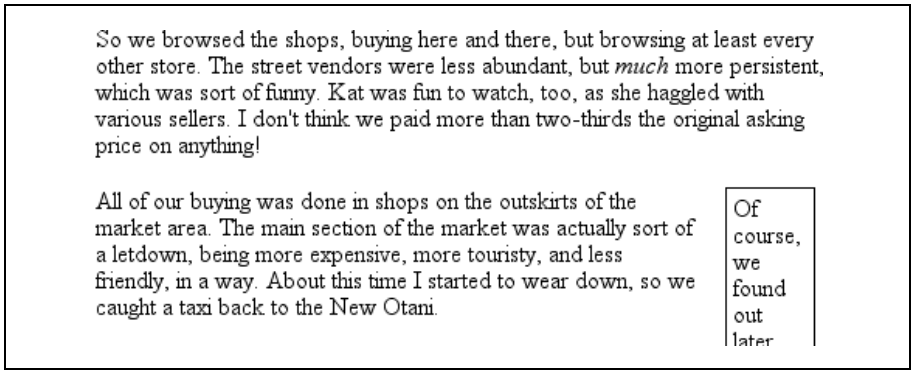


Рис. 10.4. *Перемещаемый текст, не имеющий явно заданной ширины*

Может показаться, что это не слишком умно, поскольку самый естественный путь предотвратить перемещение элемента – просто не объявлять его, так ведь? Однако значение `float` по умолчанию – `none`. Иначе говоря, это значение необходимо для обеспечения обычного неперемещаемого поведения; без него все элементы перемещались бы свободно и царила бы полная анархия.

Во-вторых, может потребоваться переопределить какой-то стиль импортированной таблицы стилей. Представьте, что используется серверная таблица стилей, в которой изображения определены как перемещаемые. Но на одной конкретной странице вы не хотите перемещать изображения. Так вот, чтобы не писать новую таблицу стилей,

достаточно поместить правило `img {float: none;}` во встроенную таблицу стилей вашего документа. Однако кроме этой ситуации применять `float: none` практически негде.

Свободное перемещение: детали

Прежде чем углубиться в детали перемещения, важно определить концепцию *блока-контейнера* (*containing block*). Блок-контейнер перемещаемого элемента – это ближайший блочный элемент-предок. Следовательно, в данной разметке блоком-контейнером перемещаемого элемента является содержащий его абзац:

```
<h1>Тест</h1>
<p>
Это текст абзаца, о чем вы знаете и так.  В состав содержимого данного абзаца
входит изображение, которое было перемещено.
<img src="testy.gif" style="float: right;">
Блок-контейнер для перемещаемого изображения – данный абзац.
</p>
```

К понятию блока-контейнера мы вернемся при обсуждении позиционирования далее в этой главе.

Более того, перемещаемый элемент независимо от своего типа генерирует блочный элемент. Таким образом, если перемещается ссылка, пусть даже элемент является строковым и обычно генерирует строковый блок, при перемещении он генерирует блочный элемент. Он будет компоноваться и действовать так, как если бы был элементом `div`, например. Точно так же действует объявление `display: block` для перемещаемого элемента.

Размещением перемещаемых элементов управляет ряд специальных правил, так что давайте рассмотрим их, перед тем как перейдем к прикладному поведению. Эти правила похожи на те, которые регулируют вычисление полей и ширины, и тоже не лишены здравого смысла. Вот они:

1. Левый (или правый) внешний край перемещаемого элемента не может располагаться левее (или правее) внутреннего края его блока-контейнера.

   Это довольно просто. Внешний левый край перемещаемого влево элемента может располагаться не левее внутреннего левого края блока-контейнера; аналогично перемещаемый вправо элемент не может располагаться правее внутреннего правого края его блока-контейнера, как показано на рис. 10.5. (На этом и последующих рисунках цифры в кружочках показывают месторасположение элемента согласно источнику, а пронумерованные блоки указывают на местоположение и размер перемещаемого видимого элемента.)

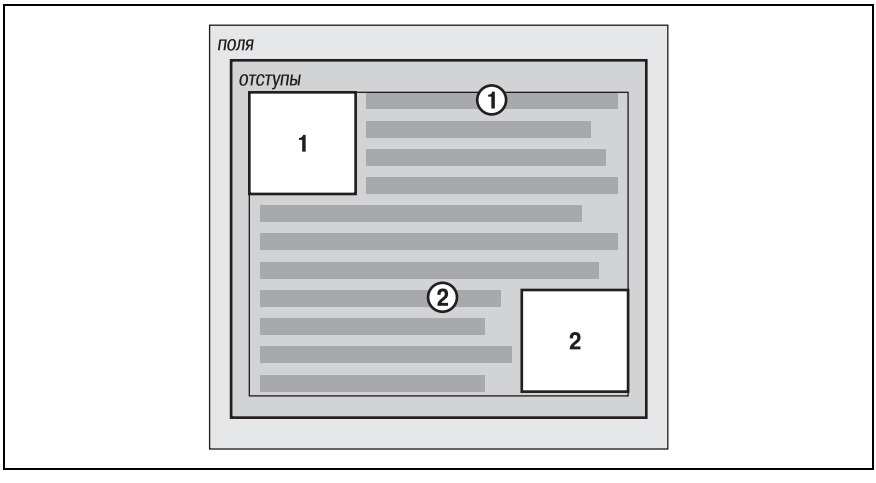


Рис. 10.5. *Перемещение влево (или вправо)*

2. Левый (или правый) внешний край перемещаемого элемента должен быть не правее (не левее) правого (левого) внешнего края перемещаемого влево (или вправо) элемента, встречающегося в исходном файле документа раньше, если только верх второго элемента не находится под низом первого.

   Это правило предупреждает наложение перемещаемых элементов. Если элемент перемещается влево, а там уже находится другой перемещаемый элемент, то второй элемент будет помещен сразу за внешним правым краем ранее перемещенного элемента. Однако если перемещаемый элемент находится ниже всех ранее перемещенных изображений, он может расположиться у внутреннего левого края родителя. Некоторые примеры такого поведения показаны на рис. 10.6.

   Преимущество этого правила в том, что все перемещаемое содержимое будет видимым, поскольку ни один из перемещаемых элементов не скроет другой. Это делает перемещение совершенно безопасным. В случае позиционирования ситуация совершенно другая: можно запросто получить перекрытие элементов.

3. Правый внешний край перемещаемого влево элемента не может располагаться правее левого внешнего края любого перемещаемого вправо элемента. Левый внешний край перемещаемого вправо элемента не может находиться левее правого внешнего края любого перемещаемого влево элемента.

   Это правило предупреждает наложение перемещаемых элементов. Скажем, у вас есть элемент `body` шириной 500 пикселов, и все его содержимое составляют два изображения шириной 300 пикселов. Первое перемещается влево, а второе – вправо. Это правило предупреждает перекрытие первого изображения вторым на 100 пиксе-

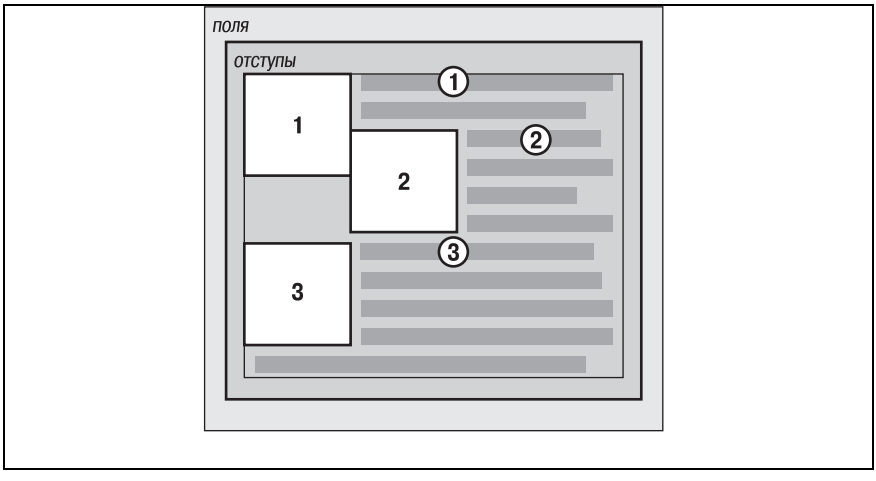


Рис. 10.6. *Предупреждение наложения перемещаемых элементов*

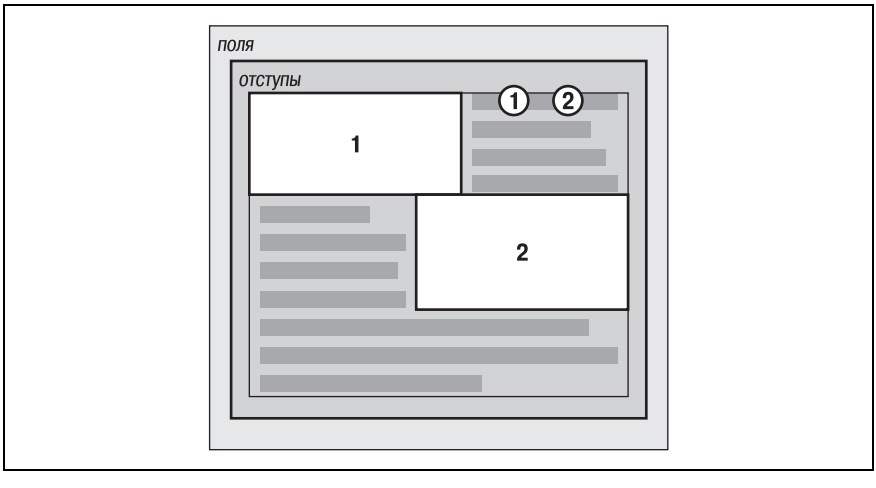


Рис. 10.7. *Дополнительное предупреждение перекрытия*

лов. Вместо этого изображение опускается вниз до тех пор, пока его верх не окажется под низом перемещаемого вправо изображения, как показано на рис. 10.7.

4. Верх перемещаемого элемента не может быть выше внутреннего верхнего края его родителя. Если перемещаемый элемент находится между двумя сворачивающимися полями, то он размещается так, как если бы его родителем был блочный элемент, расположенный между двумя элементами.

   Первая часть этого правила довольно проста и предупреждает расположение перемещаемых элементов в самом верху документа.

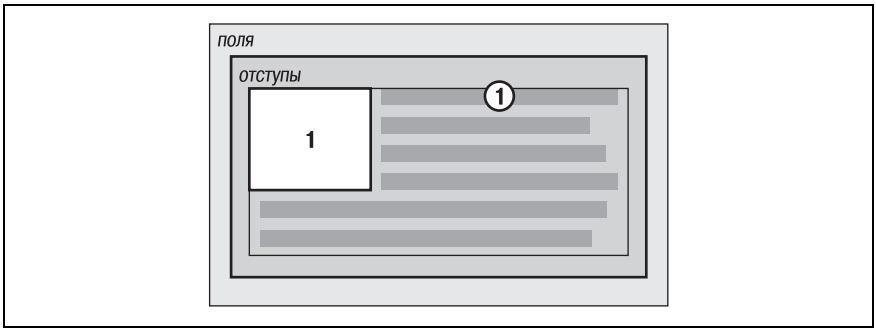


Рис. 10.8. *В отличие от воздушных шариков, перемещаемые элементы не могут «взлетать» вверх*

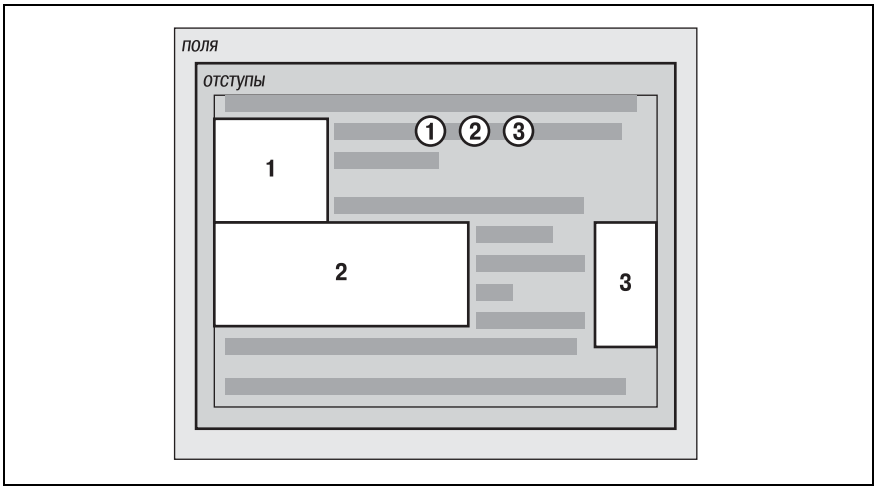


Рис. 10.9. *Размещение перемещаемых элементов ниже их предшественников*

Верное поведение проиллюстрировано на рис. 10.8. Вторая часть этого правила уточняет выравнивание в определенных ситуациях, например, когда перемещается средний из трех абзацев. В этом случае абзац перемещается так, как будто он имеет родительский блочный элемент (скажем, `div`). Это предупреждает перемещение данного абзаца в самый верх любого общего родителя этих трех абзацев.

5. Верх перемещаемого элемента не может быть выше верха любого ранее перемещенного элемента или блочного элемента.

   Аналогично правилу 4 правило 5 предупреждает появление перемещаемых элементов в самом верху их родительских элементов. Также верх перемещаемого элемента не может быть выше верха перемещаемого элемента, появившегося раньше. На рис. 10.9 показан пример – здесь видно, что, поскольку второй перемещаемый эле-

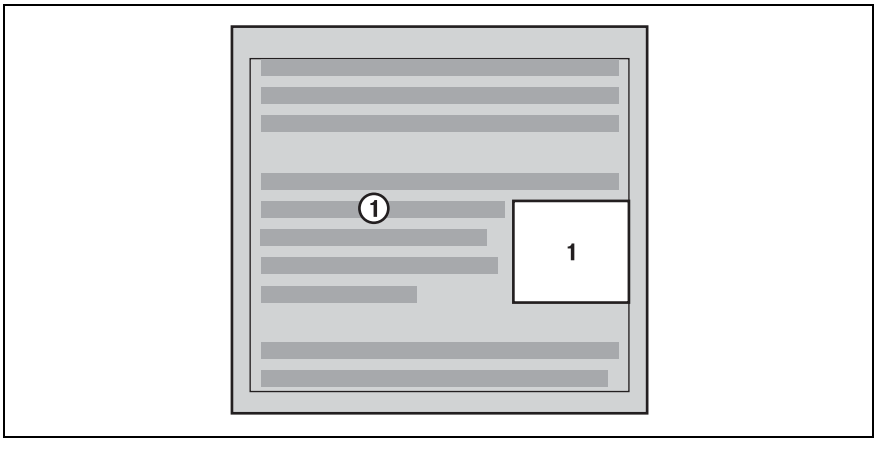


Рис. 10.10. *Обеспечение расположения перемещаемых элементов на одном уровне с их контекстом*

мент помещен под первым, верх третьего выровнен с верхом второго перемещаемого элемента, а не первого.

6. Верх перемещаемого элемента не может располагаться выше верха любого контейнера строки, содержащего блок, который сгенерирован элементом, находящимся в исходном файле документа раньше.

   Аналогично правилам 4 и 5 это правило еще более ограничивает перемещение элемента вверх, предотвращая его размещение выше верха строки, включающей содержимое, предшествующее перемещаемому элементу. Скажем, перемещаемое изображение находится прямо в центре абзаца. Верх этого изображения может находиться не выше верха контейнера строки, из которого происходит это изображение. Как видно на рис. 10.10, это предотвращает перемещение изображений на слишком большую высоту.

7. Правый внешний край перемещаемого влево (или вправо) элемента, слева (справа) от которого находится другой перемещаемый элемент, не может быть правее (левее) правого (левого) края его блока-контейнера.

   Иначе говоря, перемещаемый элемент не может выступать за край своего элемента-контейнера, если только он не занимает всю его ширину. Благодаря этому удается избежать выстраивания перемещаемых элементов в одну линию и предотвратить их выход за края блока-контейнера – перемещаемый элемент размещается ниже предыдущих перемещаемых элементов, как показано на рис. 10.11 (на рисунке перемещаемые элементы располагаются на следующей строке, что яснее иллюстрирует применяемый здесь принцип). Это правило впервые появилось в CSS2.

8. Перемещаемый элемент должен размещаться максимально высоко.

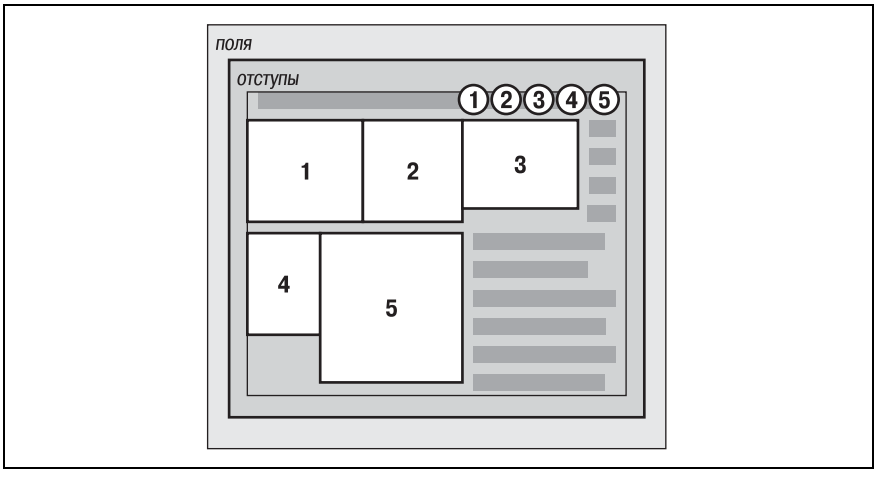


Рис. 10.11. *Если не хватает места, перемещаемые элементы выталкиваются на новую «строку»*

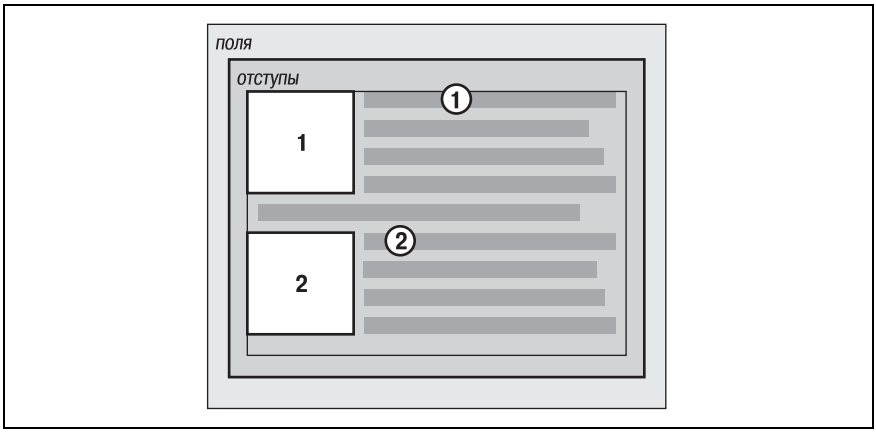
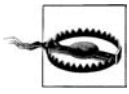


Рис. 10.12. *Учитывая остальные ограничения, располагаем элемент максимально высоко*

К правилу 8, конечно же, применяются ограничения, объявленные предыдущими семью правилами. Исторически броузеры выравнивали верх перемещаемого элемента по верху контейнера строки, следующего за тем, в котором находится тег изображения. Однако правило 8 подразумевает, что его верх должен находиться на одном уровне с верхом того же контейнера строки, в котором находится тег, предполагая наличие достаточного пространства. Теоретически верное поведение представлено на рис. 10.12.



К сожалению, поскольку значение понятия «максимально высоко» не было определено точно (что могло бы означать и, кстати, это уже обсуждалось, «настолько высоко, насколько удобно»), то нельзя надеяться на единообразное поведение даже тех броузеров, которые считаются полностью совместимыми с CSS1. Некоторые броузеры, следуя традициям, переместят изображение вниз на следующую строку, тогда как другие расположат его в текущей строке, если там будет достаточно места.

9. Перемещаемый влево элемент должен смещаться влево на максимально возможное расстояние, а элемент, перемещаемый вправо, – на максимально возможное расстояние вправо. Выше размещается тот элемент, который располагается правее или левее.

   Опять же это правило подчиняется ограничениям предыдущих правил. Пояснения аналогичны представленным в правиле 8, хотя они не настолько туманны. Как видно из рис. 10.13, довольно просто сказать, когда элемент смещается на максимально возможное расстояние влево или вправо.

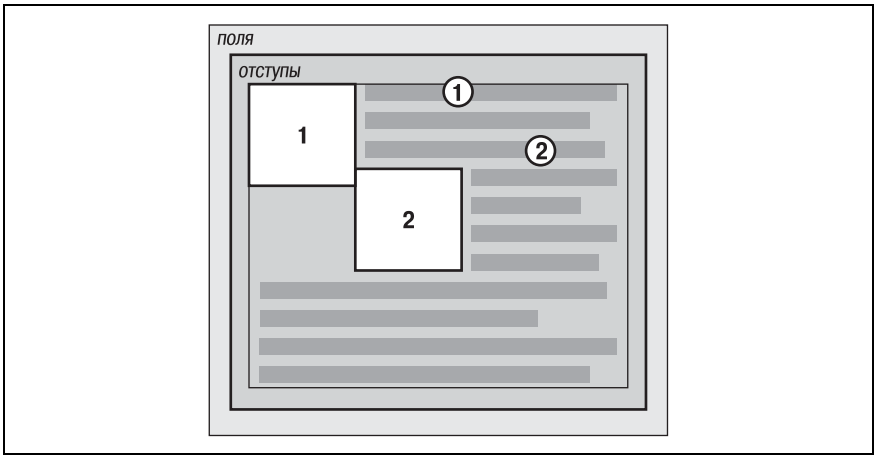


Рис. 10.13. *Смещение на максимально допустимое расстояние влево (или вправо)*

Поведение на практике

Только что рассмотренные нами правила имеют ряд интересных последствий, являющихся как результатом того, о чем в них говорится, так и того, что в них не сказано. В первую очередь обсудим, что происходит, когда перемещаемый элемент располагается выше своего родительского элемента.

Кстати, это встречается довольно часто. Возьмем, к примеру, короткий документ, состоящий из нескольких абзацев и элементов `h3`, в котором первый абзац содержит перемещаемое изображение. Более того,

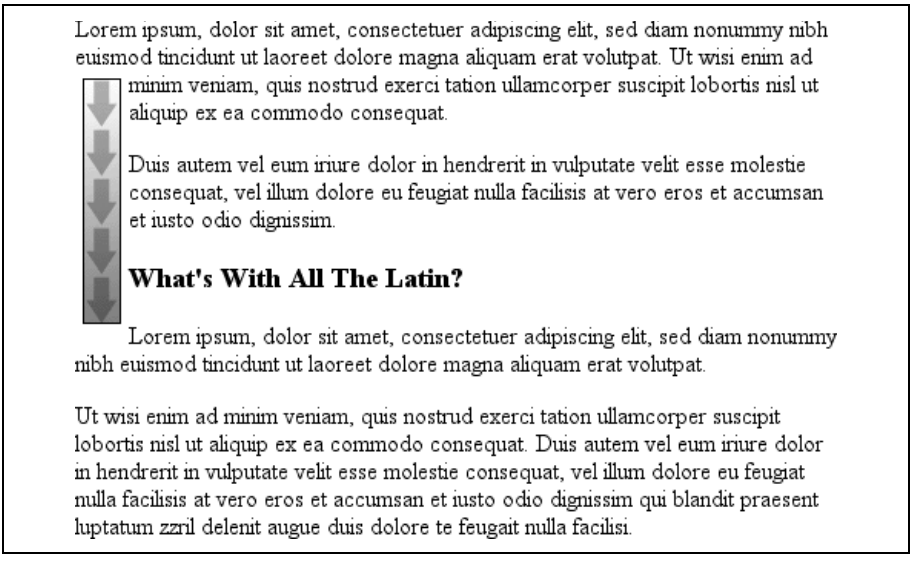


Рис. 10.14. *Ожидаемое поведение перемещения*

поле перемещаемого изображения равно пяти пикселам (`5px`). Можно было бы ожидать, что визуальное представление документа будет таким, как показано на рис. 10.14.

Конечно, в этом нет ничего необычного, но рис. 10.15 показывает, что произойдет, если задать фон для первого абзаца.

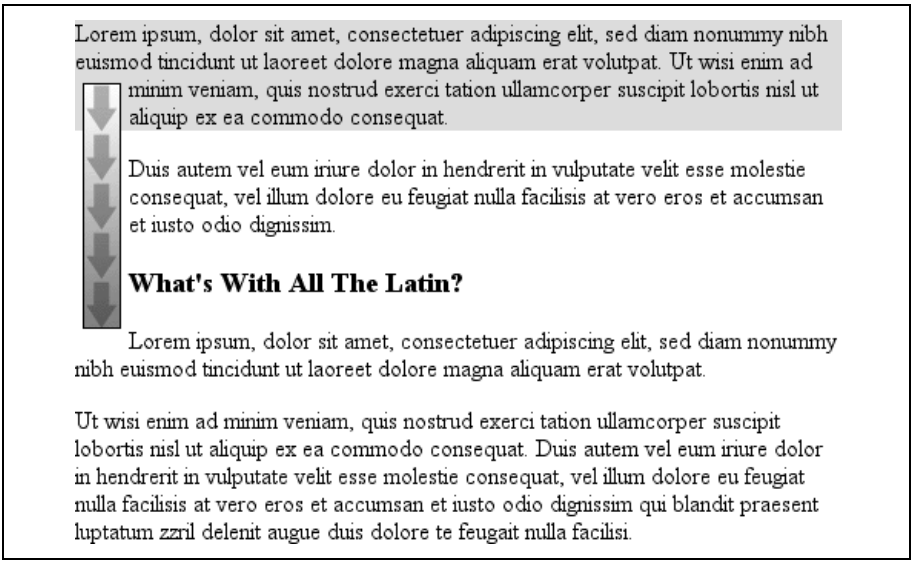


Рис. 10.15. *Фоны и перемещаемые элементы*

Второй пример отличается только наличием видимого фона. Как видите, перемещаемое изображение снизу выходит за границу родительского элемента. Конечно, это же имело место и в первом примере, но не было таким очевидным, потому что вы не видели фон. Правила перемещения, обсуждаемые нами ранее, касались только левого, правого и верхнего краев перемещаемых элементов и их родителей. Намеренное игнорирование нижнего края вызывает поведение, продемонстрированное на рис. 10.15.

На практике некоторые броузеры делают это некорректно. Они увеличивают высоту родительского элемента до тех пор, пока перемещаемый элемент не поместится в него полностью, даже несмотря на то, что это приводит к возникновению значительного пустого пространства в родительском элементе.

CSS2.1 проясняет один из аспектов поведения перемещаемых элементов: перемещаемый элемент увеличивается, чтобы вместить всех перемещаемых потомков. (Предыдущие версии CSS не давали четкого определения того, что должно происходить.) Таким образом, можно вместить перемещаемый элемент в его родительский элемент, делая его перемещаемым, как в этом примере:

```
<div style="float: left; width: 100%;">
  <img src="hay.gif" style="float: left;">
  Этот 'div' распространится вокруг перемещаемого изображения, потому что
'div' тоже стал перемещаемым.
</div>
```

Для сравнения рассмотрим фон и его взаимоотношения с перемещаемыми элементами, которые встречаются в документе раньше, что проиллюстрировано на рис. 10.16.

Перемещаемые элементы находятся как внутри, так и вне потока; такие вещи будут случаться. Что происходит? Содержимое заголовка «вытесняется» перемещаемым элементом. Но ширина элемента заголовка остается равной ширине его родительского элемента. Следовательно, его область содержимого распространяется на ширину родителя, то же самое делает и фон. В действительности содержимое не занимает всей своей области содержимого и не перекрывается перемещаемым элементом.

Отрицательные поля

Любопытно, что если задать отрицательные значения для полей, то перемещаемые элементы могут выйти за границы их родительских элементов. Кажется, что это полностью противоречит описанным выше правилам, но это не так. Точно так же, как элементы могут казаться шире своих родителей из-за отрицательных полей, перемещаемые элементы могут оказываться вне своих родителей.

Рассмотрим перемещаемое изображение, которое смещается влево и имеет поля слева и сверху по `-15px`. Это изображение размещается

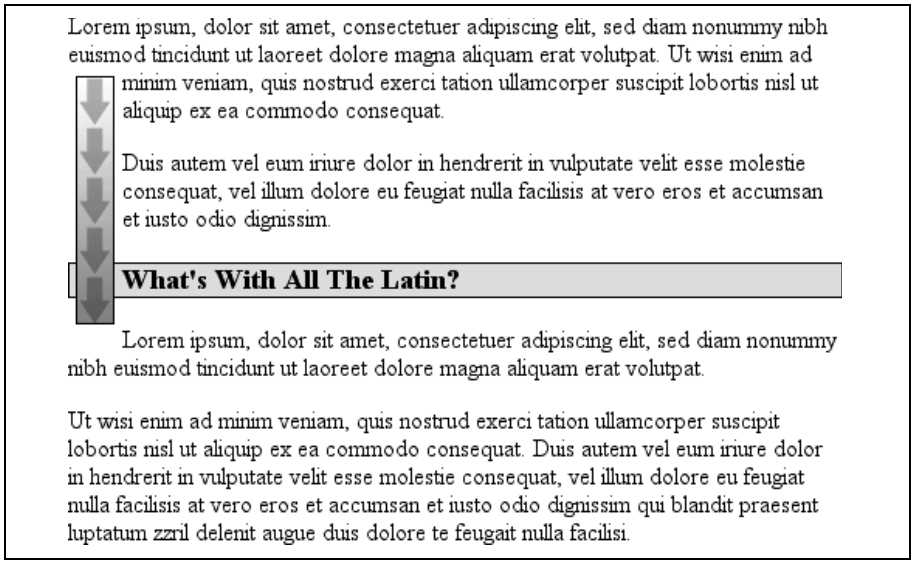


Рис. 10.16. *Фоны элементов подкладываются под перемещаемые элементы*

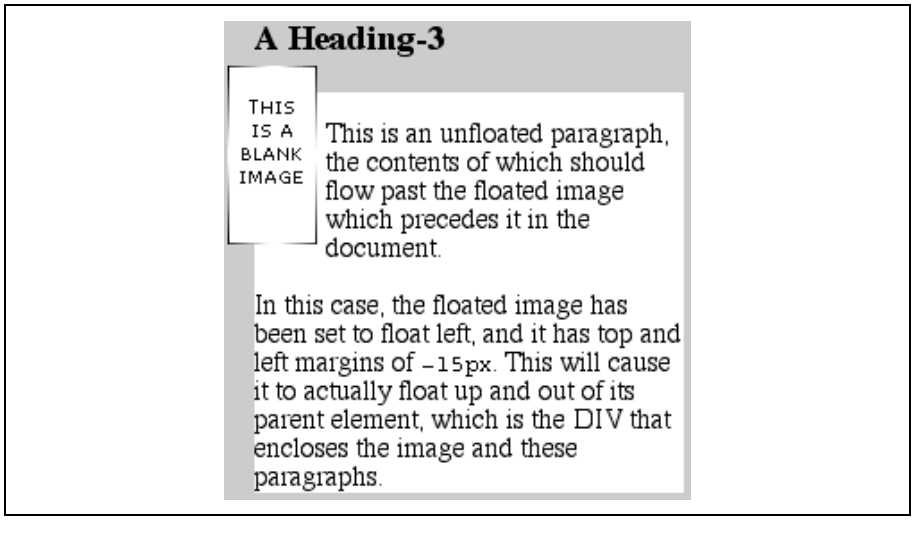


Рис. 10.17. *Перемещение при наличии отрицательных полей*

внутри элемента `div`, у которого нет отступов, рамок или полей. Результат показан на рис. 10.17.

Вопреки очевидности это не нарушает ограничений на перемещаемые элементы, размещаемые вне их родительских элементов.

Внимательно прочитав правила предыдущего раздела, можно понять техническую сторону вопроса, которая делает возможным такое пове-

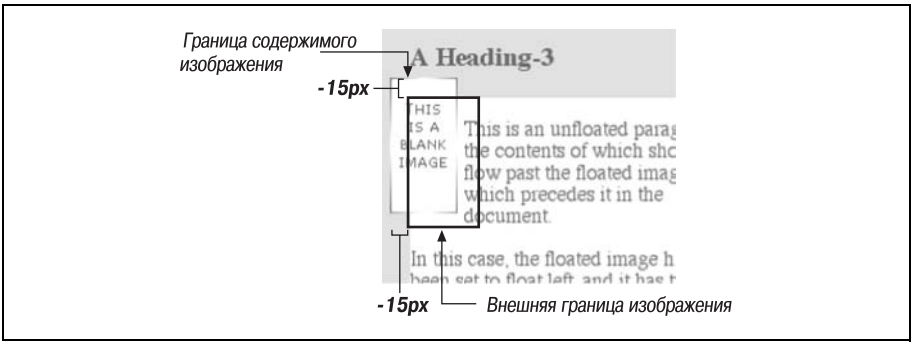


Рис. 10.18. *Подробности перемещения вверх и вниз при наличии отрицательных полей*

дение: внешние края перемещаемого элемента должны находиться в родительском элементе. Однако отрицательные поля могут разместить содержимое перемещаемого элемента так, что оно будет перекрывать собственный внешний край, как показано на рис. 10.18.

С математической точки зрения ситуация выглядит примерно так: примем, что верхний внутренний край элемента `div` соответствует координате 100 пикселов. Чтобы определить, где должен находиться верхний внутренний край перемещаемого элемента, броузер сделает следующее: `100px + (-15px) margin + 0 padding = 85px`. Таким образом, верхний внутренний край перемещаемого элемента должен соответствовать координате 85 пикселов; вычисления не противоречат спецификации, даже несмотря на то, что это выше, чем верхний внутренний край родителя перемещаемого элемента. Аналогичная цепочка доказательств объясняет, как левый внутренний край перемещаемого элемента может оказаться левее левого внутреннего края его родителя.

У многих из вас, возможно, прямо сейчас появилось непреодолимое желание крикнуть: «Это не по правилам!». Лично я вас не осуждаю. Кажется совершенно неправильным, например, позволить верхнему внутреннему краю располагаться выше верхнего внешнего края, но при наличии отрицательного верхнего поля именно так и получается; точно так же, как и применение отрицательных полей к обычным неперемещаемым элементам может сделать их визуально шире родителей. То же самое происходит и со всеми четырьмя сторонами блока перемещаемого элемента: если задано отрицательное поле, содержимое выйдет за внешний край, не нарушая при этом спецификации.

Здесь возникает один важный вопрос: что происходит с представлением документа, когда элемент перемещается за границы родительского элемента при задании отрицательных полей? Например, сильно выступающее вверх изображение могло бы задеть абзац, уже отображенный агентом пользователя. В подобных случаях агент пользователя решает, должен ли быть перестроен поток документа. Спецификации

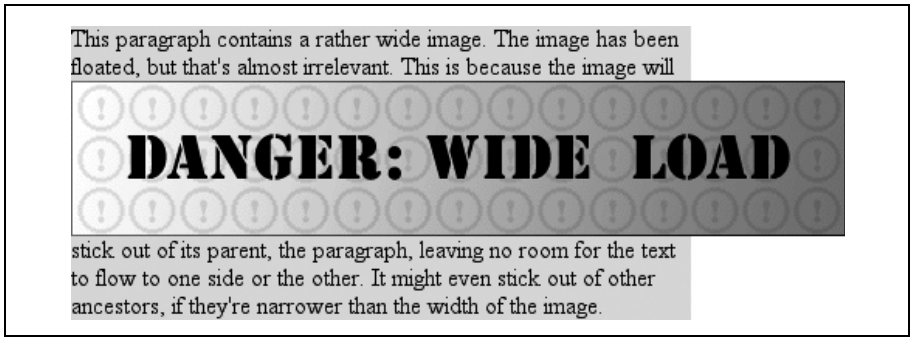


Рис. 10.19. *Перемещение элемента, который шире своего родителя*

CSS1 и CSS2 явно говорят, что от агентов пользователя не требуется перестраивать предыдущее содержимое с целью согласования с тем, что имеет место в документе позже. Иначе говоря, если изображение перемещается в предыдущий абзац, оно может просто перекрыть то, что там находится. С другой стороны, агент пользователя может обрабатывать ситуацию, задавая обтекание перемещаемого изображения содержимым. В любом случае не следует рассчитывать на определенное поведение, что ограничивает применение отрицательных полей в перемещаемых элементах. Перемещаемые элементы довольно безопасны, но пытаться вытолкнуть их вверх не стоит.

У перемещаемого элемента есть еще одна возможность выйти за внутренний левый и правый края его родителя: элемент должен быть шире своего родителя. В этом случае в благородном порыве отобразить себя правильно перемещаемый элемент просто выйдет за правый или левый внутренний край в зависимости от того, как он перемещается. Это приведет к результату, показанному на рис. 10.19.

Перемещаемые элементы, содержимое и перекрытие

Вот еще более интересный вопрос: что происходит, когда перемещаемый элемент перекрывает содержимое в нормальном потоке? Это может произойти, если, например, у перемещаемого элемента отрицательное поле с той стороны, где находится предшествующее содержимое (например, отрицательное поле слева в перемещаемом вправо элементе). Вы уже видели, что происходит с рамками и фоном блочных элементов. А что в строковых элементах?

CSS1 и CSS2 не давали точного определения по поводу ожидаемого поведения в подобных случаях. CSS2.1 проясняет этот вопрос, формулируя следующие точные правила:

- Рамки, фон и содержимое строкового блока, пересекающегося с перемещаемым элементом, генерируются поверх перемещаемого элемента.

- Рамки и фон блочного элемента, которые пересекаются с перемещаемым элементом, генерируются под перемещаемым элементом, тогда как его содержимое генерируется поверх перемещаемого элемента.

Чтобы проиллюстрировать это правило, рассмотрим следующую ситуацию:

```
<img src="testy.gif" class="sideline">
<p class="box">
This paragraph, unremarkable in most ways, does contain an inline element.
This inline contains some <strong>strongly emphasized text, which is
so marked to make an important point</strong>.  The rest of the element's
content is normal anonymous inline content.
</p>
<p>
This is a second paragraph.  There's nothing remarkable about it, really.
Please move along.
</p>
<h2 id="jump-up">A Heading!</h2>
```

К этой разметке применяем следующие стили, результат показан на рис. 10.20:

```
img.sideline {float: left; margin: 10px -15px 10px 10px;}
p.box {border: 1px solid gray; padding: 0.5em;}
p.box strong {border: 3px double black; background: silver; padding: 2px;}
h2#jump-up {margin-top: -15px; background: silver;}
```

Строковый элемент (`strong`) полностью перекрывает перемещаемое изображение: фон, рамку, содержимое и все остальное. У блочных элементов поверх перемещаемого элемента оказывается только содержимое. Их фон и рамки остаются за перемещаемым элементом.

Описанное поведение перекрытия не зависит от исходного порядка документа. Неважно, где находится элемент – перед или после перемещаемого элемента, – все равно поведение остается неизменным.

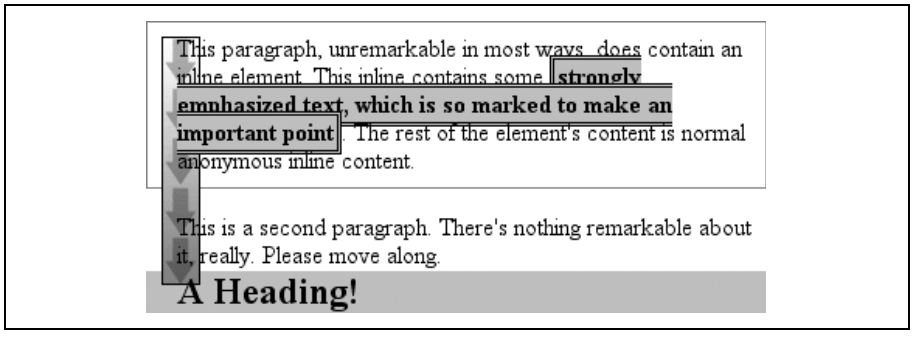


Рис. 10.20. *Пересечение с перемещаемыми элементами*

Запрет на обтекание

Мы достаточно поговорили о поведении при перемещении. Прежде чем перейти к позиционированию, обсудим еще один вопрос. Не всегда хочется, чтобы содержимое обтекало перемещаемый элемент, в некоторых случаях этого надо избежать. Если документ разбит на разделы, то перемещаемые элементы одного раздела не должны будут «свисать» в следующий. В этом случае имеет смысл запретить размещение первого элемента каждой секции следом за перемещаемым элементом. Если бы первый элемент все равно оказался рядом с перемещаемым элементом, он был бы смещен вниз, под перемещаемый элемент, и все последующее содержимое последовало бы за ним, как показано на рис. 10.21.

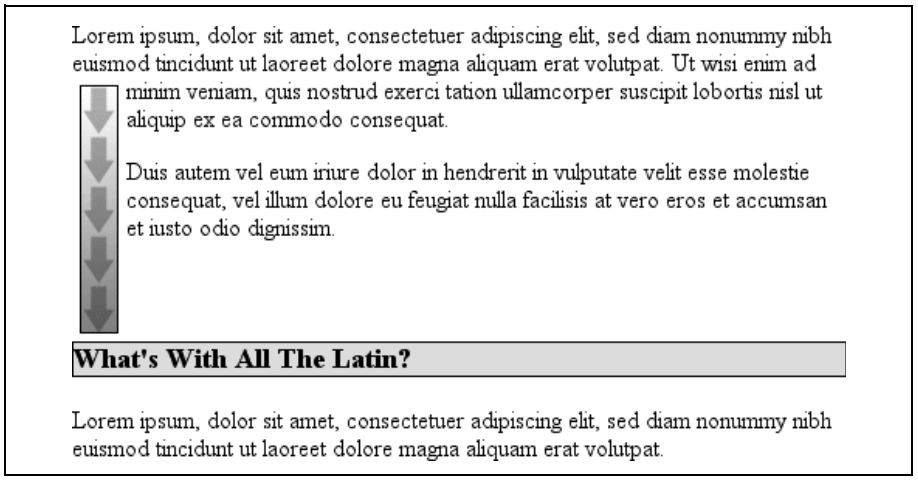


***Рис. 10.21.** Отображение элемента без обтекания*

Это делается с помощью свойства `clear`.

Например, объявив `h3 {clear: left;}`, можно гарантировать, что ни один элемент `h3` не будет размещен справа от перемещаемого влево элемента. Объявление можно перевести так: «убедитесь, что левая сторона `h3` свободна от перемещаемых изображений», и результат его применения очень похож на эффект от HTML-конструкции `<br clear="left">`.

clear

Значения:	`left | right | both | none | inherit`
Начальное значение:	`none`
Область применения:	блочные элементы
Наследование:	нет
Вычисляемое значение:	как задано

(По иронии судьбы в большинстве броузеров элементы `br` по умолчанию формируют строковые блоки, поэтому свойство `clear` может быть применено к ним только после изменения `display`!) Следующее правило применяет свойство `clear`, чтобы предотвратить обтекание перемещаемых влево элементов элементами `h3`:

```
h3 {clear: left;}
```

Это правило гарантирует, что все элементы `h3` будут смещены ниже всех перемещаемых влево элементов, но появление перемещаемых элементов справа от элементов `h3` разрешено, как показано на рис. 10.22.

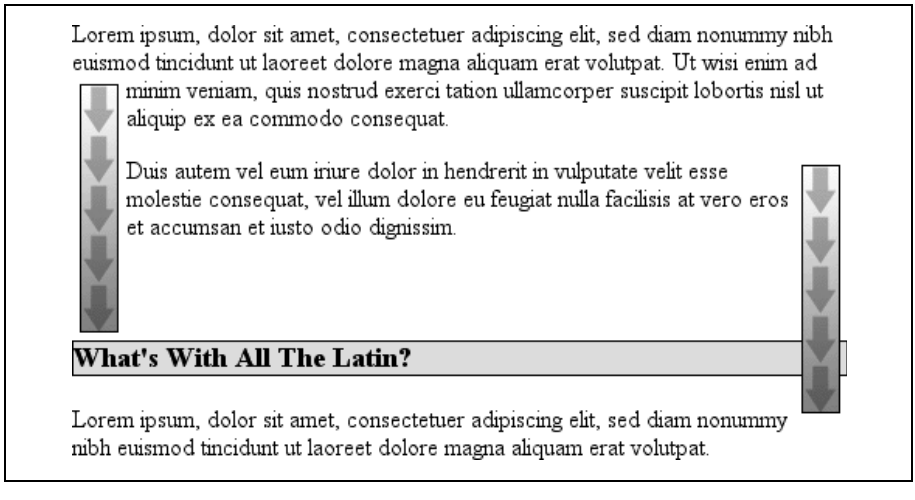


Рис. 10.22. *Запрет обтекания слева, но не справа*

Чтобы избежать подобных явлений и гарантировать отсутствие любых перемещаемых элементов в одной строке с элементами `h3`, задайте значение `both`:

```
h3 {clear: both;}
```

Все понятно: это значение предотвращает существование перемещаемых элементов с обеих сторон необтекаемого элемента, как продемонстрировано на рис. 10.23.

Если же надо было бы беспокоиться только о расположении элементов `h3` относительно перемещаемых вправо элементов, то объявление было бы таким: `h3 {clear: right;}`.

Наконец, существует значение `clear: none`, которое допускает обтекание элементов с любой стороны. Как и `float: none`, это значение главным образом существует для того, чтобы обеспечить обычное поведение документа, в котором перемещаемые элементы могут располагаться с обеих сторон элементов. Конечно, значение `none` может применяться для переопределения других стилей, как показано на рис. 10.24.

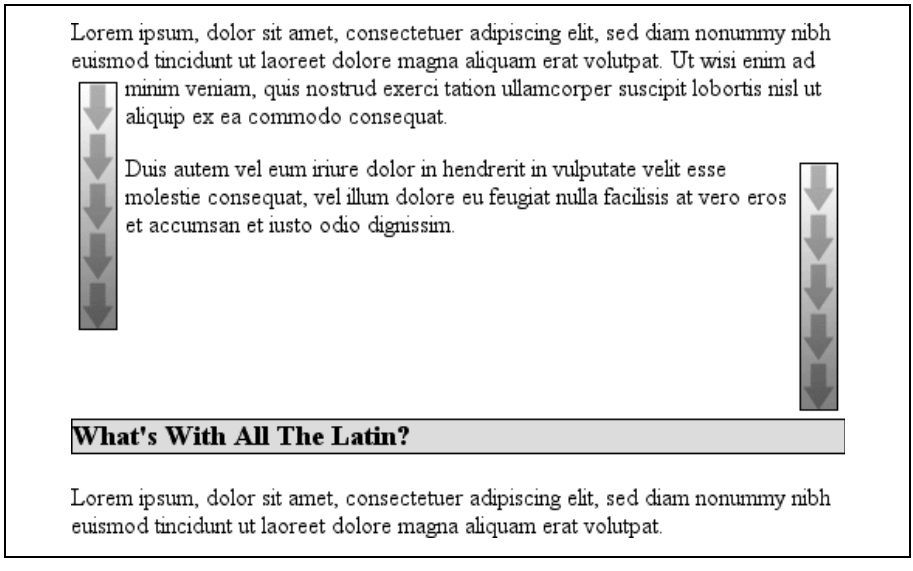


Рис. 10.23. *Запрет обтекания с обеих сторон*

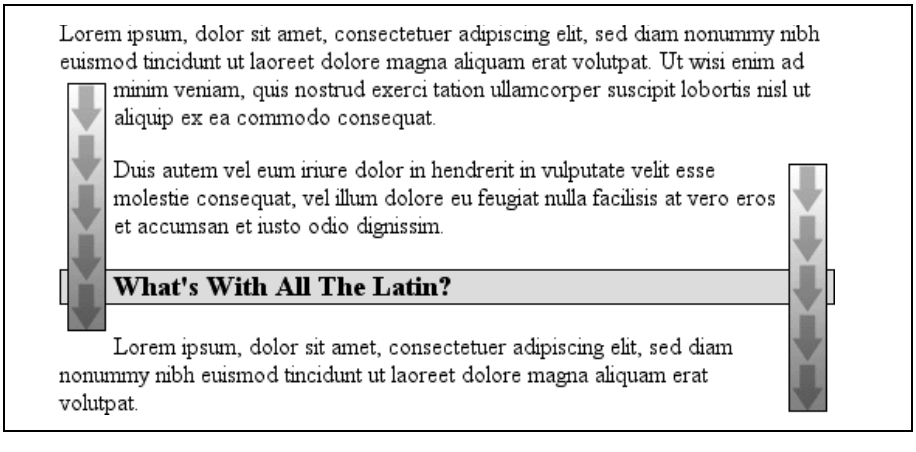


Рис. 10.24. *Полностью обтекаемый*

Нес мотря на общее для всего документа правило, по которому элементы `h3` не допускают размещение перемещаемых элементов с любой из сторон, для одного из `h3` было разрешено размещение перемещаемых элементов со всехсторон:

```
h3 {clear: both;}

<h3 style="clear: none;">What's With All The Latin?</h3>
```

В CSS1 и CSS2 работа свойства `clear` обеспечивалась увеличением верхнего поля элемента таким образом, что оно заканчивается ниже пере-

мещаемого элемента. В итоге любое поле, заданное для верха необтекаемого элемента, игнорировалось. То есть вместо ширины в 1.5em, например, поле увеличилось бы до 10em, или 25px, или 7.133in, или любого другого значения, достаточного для перемещения элемента вниз на такое расстояние, чтобы область содержимого располагалась ниже нижнего края перемещаемого элемента.

В CSS2.1 было введено понятие *превышения* (*clearance*). Превышение – это дополнительное пространство, добавляемое сверху к верхнему полю элемента, чтобы сместить его ниже всех перемещаемых элементов. Это значит, что верхнее поле элемента не меняется, когда элемент объявляется необтекаемым. Его перемещение вниз обусловлено превышением. Внимательно рассмотрите размещение рамки заголовка на рис. 10.25 – результат следующей разметки:

```
img.sider {float: left; margin: 0;}
h3 {border: 1px solid gray; clear: left; margin-top: 15px;}

<img src="boxer.gif" class="sider" height="50" width="50">
<img src="stripe.gif" height="10" width="100">
<h3>Why Doubt Salmon?</h3>
```

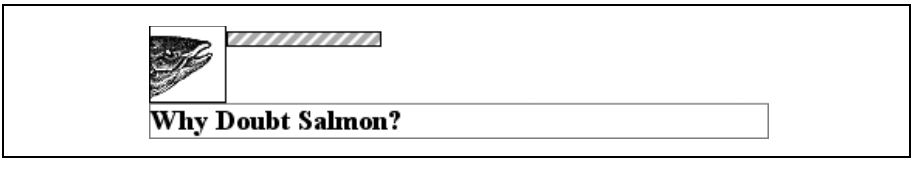


Рис. 10.25. *Превышение и его воздействие на поля*

Между верхней рамкой h3 и нижней рамкой перемещаемого изображения нет пробела, потому что для смещения верхнего края рамки h3 под нижний край перемещаемого элемента над 15-пиксельным верхним полем было добавлено 25 пикселов превышения. Это произойдет, если только верхнее поле h3 будет менее 40 пикселов, в противном случае h3 естественным образом будет размещаться под перемещаемым объектом, и значение clear будет лишним.

В большинстве случаев, конечно, неизвестно, насколько далеко надо переместить элемент. Чтобы гарантировать, что между верхом необтекаемого элемента и низом перемещаемого элемента будет иметься просвет, необходимо задать нижнее поле самому перемещаемому элементу. Поэтому, если бы потребовалось, чтобы в предыдущем примере под перемещаемым элементом был хотя бы 15-пиксельный просвет, надо было бы изменить CSS вот так:

```
img.sider {float: left; margin: 0 0 15px;}
h3 {border: 1px solid gray; clear: left;}
```

Нижнее поле перемещаемого элемента увеличивает размер его блока и таким образом изменяет местоположение точки, ниже которой долж-

ны быть смещены необтекаемые элементы. Дело в том, что, как вы видели ранее, края блока перемещаемого элемента определяют края его полей.

Позиционирование

Идея, положенная в основу позиционирования, довольно проста. Оно позволяет точно определять, где появятся блоки элементов относительно того, где они оказались бы в обычном порядке: относительно родительского элемента, другого элемента или даже относительно окна броузера.

Основные понятия

Прежде чем углубиться в различные виды позиционирования, нелишним будет рассмотреть, какие есть типы позиционирования и чем они отличаются. Нам также понадобится обозначить некоторые основные идеи, имеющие исключительное значение для понимания принципа работы позиционирования.

Типы позиционирования

Свойство `position` позволяет выбрать один из четырех типов позиционирования, которые оказывают влияние на то, как генерируется блок элемента.

Значения свойства `position` имеют следующий смысл:

`static`

Блок элемента генерируется соответственно нормальному потоку. Блочные элементы генерируют прямоугольный блок, представляющий собою часть потока документа, строковые блоки обусловливают создание одного или нескольких контейнеров строк, которые располагаются в своем родительском элементе.

`relative`

Блок элемента смещается на некоторое расстояние. Элемент сохраняет форму, которую имел бы, если бы не был позиционирован, и размер, который бы занимал при обычных условиях.

position

Значения:	`static | relative | absolute | fixed | inherit`
Начальное значение:	`static`
Область применения:	все элементы
Наследование:	нет
Вычисляемое значение:	как задано

`absolute`

Блок элемента полностью удаляется из потока документа и позиционируется относительно его блока-контейнера, которым может быть другой элемент документа или начальный блок-контейнер (описываемый в следующем разделе). Любое пространство, которое элемент, возможно, занимал в нормальном потоке документа, закрывается, как если бы элемент не существовал. Абсолютно позиционированный элемент генерирует структурный блок (block-level box) независимо от типа блока, который он генерировал бы, если бы находился в нормальном потоке.

`fixed`

Блок элемента ведет себя так, как если бы он был абсолютно позиционированным, но его блоком-контейнером являлось бы само окно просмотра.

Не будем сейчас сильно углубляться в детали, поскольку мы рассмотрим каждый из этих типов позиционирования позже в этой главе. Обсудим сначала блоки-контейнеры.

Блок-контейнер

Ранее в этой главе мы обсуждали блоки-контейнеры в связи с перемещаемыми элементами. В том случае блок-контейнер перемещаемого элемента был определен как ближайший блочный элемент-предок. Что касается позиционирования, то все не так просто. CSS2.1 определяет следующие линии поведения:

- Блок-контейнер «корневого элемента» (также называемый *начальным блоком-контейнером*) определяется агентом пользователя. В HTML корневым элементом является элемент `html`, хотя некоторые броузеры используют элемент `body`. В большинстве броузеров начальный блок-контейнер – это прямоугольник, размер которого соответствует размеру окна просмотра.
- Для некорневого элемента, значение свойства `position` которого – `relative` или `static`, блок-контейнер формируется краем содержимого ближайшего блочного элемента-предка, ячейки таблицы или строкового блока.
- Для некорневых элементов, значение свойства `position` которых – `absolute`, блоком-контейнером считается ближайший предок (любой), значение `position` которого отлично от `static`. Это происходит следующим образом:
  - Если предок является блочным элементом, блок-контейнер формируется его внешней границей отступа; иначе говоря, областью, которая была бы ограничена рамкой.
  - Если предок – строковый элемент, блок-контейнер формируется внешней границей содержимого предка. В языках с написанием слева направо верхняя и левая стороны блока-контейнера – это

верхний и левый края содержимого первого блока, порожденного предком, а нижний и правый края – это нижний и правый края содержимого последнего блока. В языках с написанием справа налево правый край блока-контейнера соответствует правому краю содержимого первого блока, а левый край соответствует левому краю последнего блока. Верх и низ аналогично.

- Если предков нет, блок-контейнер элемента является начальным блоком-контейнером.

Важно отметить: элементы могут позиционироваться вне их блока-контейнера. Очень похоже перемещаемые элементы могут применять отрицательные поля, чтобы выйти за пределы области содержимого их родителя. Также предлагается заменить термин «блок-контейнер» на «контекст позиционирования», но поскольку в спецификации говорится «блок-контейнер», так буду поступать и я. (Я действительно пытаюсь свести путаницу к минимуму. Честное слово!)

Свойства смещения

Три схемы позиционирования, описанные в предыдущем разделе, – `relative`, `absolute` и `fixed` – используют четыре разных свойства для описания смещения сторон позиционируемого элемента относительно его блока-контейнера. Эти четыре свойства, которые мы будем называть *свойствами смещения* (*offset properties*), делают большую часть работы по позиционированию.

top, right, bottom, left

Значения:	<длина> | <процентное значение> | `auto` | `inherit`
Начальное значение:	`auto`
Область применения:	позиционированные элементы (т. е. элементы, для которых значение свойства `position` не равно `static`)
Наследование:	нет
Процентные соотношения:	относительно высоты блока-контейнера для `top` и `bottom` и ширины блока-контейнера для `right` и `left`
Вычисляемое значение:	для относительно позиционированных элементов см. примечание ниже; для элементов `static` – `auto`; для значений в единицах длины – соответствующая абсолютная длина; для процентных значений – заданное значение; в противном случае – `auto`
Примечание:	вычисляемые значения зависят от ряда факторов; примеры для отдельных свойств см. в «Приложении А»

Эти свойства описывают смещение относительно ближайшей стороны блока-контейнера (в соответствии со словом *смещение* (*offset*)). Например, свойство `top` описывает, на сколько должен отстоять верхний край поля позиционируемого элемента от верха его блока-контейнера. Для свойства `top` положительные значения перемещают верхний край поля позиционируемого элемента *ниже*, а отрицательные значения перемещают его *выше* верха его блока-контейнера. Аналогично свойство `left` описывает, насколько далеко вправо (для положительных значений) или влево отстоит левый край поля позиционируемого элемента относительно левого края блока-контейнера. Положительные значения будут сдвигать край поля позиционируемого элемента вправо, а отрицательные значения – влево.

Можно взглянуть на это иначе: положительные значения обусловливают смещения внутрь, передвигая края по направлению к центру блока-контейнера, а отрицательные – смещения наружу.

В исходной спецификации CSS2 сказано, что смещаются края содержимого, а не края поля, но это противоречило другим частям CSS2. Ошибка была исправлена в последующем списке опечаток и в CSS2.1. Вычисление смещений выполняется по краям полей во всех активно разрабатываемых (на момент написания данной книги) броузерах.

Как результат смещения краев полей позиционированного элемента в процессе его позиционирования перемещается все: и поля, и рамки, и отступы, и содержимое. Таким образом, для позиционированных элементов можно задавать поля, рамки и отступы; все это будет сохранено и останется в позиционированном элементе и будет находиться в области, определенной свойствами смещения.

Важно помнить, что свойства смещения определяют смещение от аналогичной стороны (например, `left` определяет смещение от левой стороны) блока-контейнера, а не от его верхнего левого угла. Вот почему можно, например, заполнить нижний правый угол блока-контейнера, задав такие значения:

```
top: 50%; bottom: 0; left: 50%; right: 0;
```

В этом примере внешний левый край позиционированного элемента размещается посередине блока-контейнера. Это его смещение от левого края блока-контейнера. Внешний правый край позиционированного элемента, однако, не сдвинут относительно правого края блока-контейнера, поэтому они совпадают. Аналогично для верха и низа позиционированного элемента: внешний верхний край находится посередине блока-контейнера, но внешний нижний край совпадает с нижним краем блока-контейнера. Это приводит к результату, показанному на рис. 10.26.

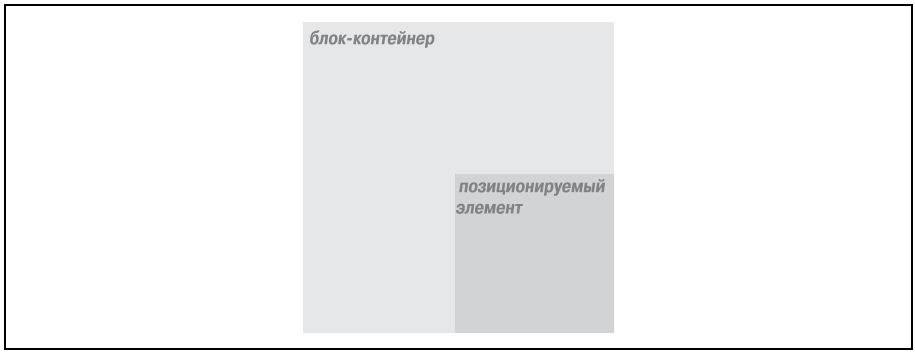


Рис. 10.26. *Заполнение нижней правой четверти блока-контейнера*

Пример, представленный на рис. 10.26, и большинство примеров данной главы касаются абсолютного позиционирования. Поскольку абсолютное позиционирование является самой простой схемой для демонстрации принципа работы свойств `top`, `right`, `bottom`, мы будем придерживаться его.

Обратите внимание, что позиционированный элемент имеет немного другой цвет фона. На рис. 10.26 у него нет полей, но если бы они были, то создавали бы пустое пространство между рамками и смещаемыми краями. При этом казалось бы, что позиционированный элемент не заполняет всю нижнюю правую четверть блока-контейнера. По правде говоря, он бы заполнял область, но этого не было бы видно. Таким образом, следующие два набора стилей имели бы примерно одинаковое представление, если предположить, что размер блока-контейнера равен `100em`×`100em`.

```
top: 50%; bottom: 0; left: 50%; right: 0; margin: 10em;
top: 60%; bottom: 10%; left: 60%; right: 10%; margin: 0;
```

Опять же подобие было бы только кажущимся.

Задавая отрицательные значения, можно расположить элемент за пределами его блока-контейнера. Так, применение следующих значений приведет к результату, показанному на рис. 10.27:

```
top: -5em; bottom: 50%; left: 75%; right: -3em;
```

Кроме значений длины и процентных значений, для свойств смещения также может быть задано значение `auto`, устанавливаемое по умолчанию. Для `auto` не определена какая-то конкретная схема поведения; оно меняется в зависимости от выбранного типа позиционирования. Принцип работы `auto` мы изучим в этой главе несколько позже при рассмотрении каждого из типов позиционирования.

Рис. 10.27. *Позиционирование элемента вне блока-контейнера*

Ширина и высота

Нередко при задании месторасположения элемента требуется объявить, насколько высоким и широким он должен быть. Кроме того, в определенных ситуациях нужно задать ограничения на высоту и ширину элемента; при этом стоит помнить, что когда вы предоставляете все решать броузеру, то он автоматически вычисляет ширину, высоту или оба этих параметра.

Задание ширины и высоты

Если необходимо задать ширину позиционированного элемента, очевидно, надо использовать свойство `width`. Аналогично свойство `height` позволяет объявить высоту позиционированного элемента.

Однако задавать ширину и высоту элемента при позиционировании элементов не всегда необходимо. Так, если размещение четырех сторон элемента описывается свойствами `top`, `right`, `bottom` и `left`, то высота и ширина элемента неявно определяются смещениями. Предположим, абсолютно позиционированный элемент должен заполнить левую половину своего блока-контейнера сверху донизу. Ниже приводятся возможные значения, результат представлен на рис. 10.28:

```
top: 0; bottom: 0; left: 0; right: 50%;
```

По умолчанию для свойств `width` и `height` устанавливается значение `auto`, поэтому результат, показанный на рис. 10.28, будет точно таким же, как и при задании следующих значений:

```
top: 0; bottom: 0; left: 0; right: 50%; width: 50%; height: 100%;
```

Присутствие `width` и `height` в этом примере ничего не добавляет к компоновке этого элемента.

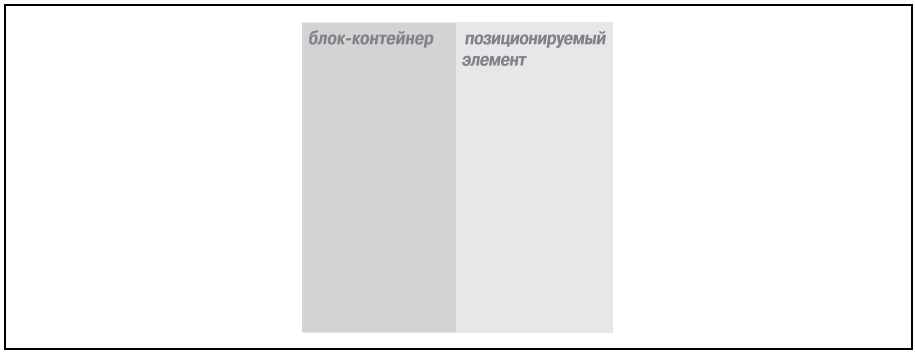


Рис. 10.28. *Позиционирование и изменение размеров элемента только с помощью свойств смещения*

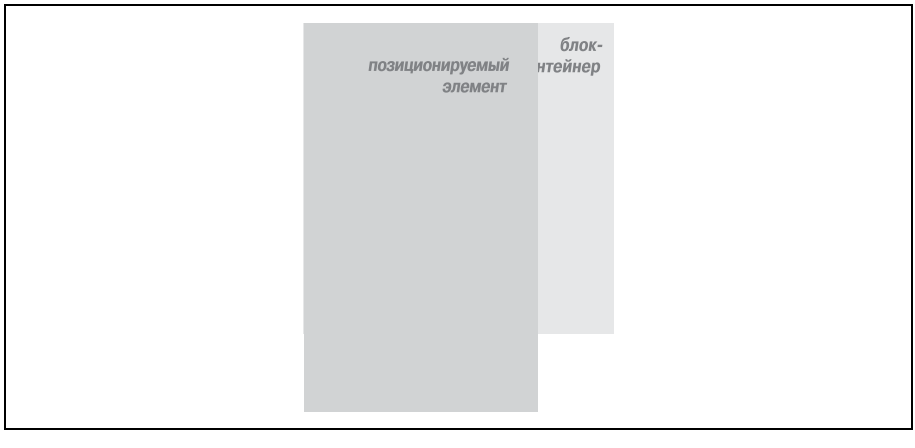


Рис. 10.29. *Позиционирование элемента частично вне его блока-контейнера*

Конечно, если добавить в элемент отступы, рамку или поля, то прямое задание `height` и `width` имело бы смысл:

```
top: 0; bottom: 0; left: 0; right: 50%; width: 50%; height: 100%;
  padding: 2em;
```

В результате получается позиционированный элемент, выходящий за рамки своего блока-контейнера, как показано на рис. 10.29.

Причина в том, что, как мы видели в предыдущих главах, отступы добавляются к области содержимого, а ее размер определяется значениями `height` и `width`. Чтобы получить требуемые отступы и не допустить выхода элемента за рамки его блока-контейнера, надо либо удалить объявления `height` и `width`, либо явно присвоить им обоим значение `auto`.

Ограничение ширины и высоты

Ширина элемента ограничивается при помощи свойств CSS2, которые я называю *свойствами минимума-максимума* (*min-max properties*). Для области содержимого можно задать минимальные размеры с помощью свойств `min-width` и `min-height`.

min-width, min-height

Значения:	<длина> | <процентное значение> | `inherit`
Начальное значение:	`0`
Область применения:	все элементы, кроме незамещаемых строковых элементов и элементов таблиц
Наследование:	нет
Процентные значения:	относительно ширины/высоты блока-контейнера
Вычисляемое значение:	для процентных значений – как задано; для значений, заданных в единицах измерения длины, – абсолютная длина; в противном случае – `none`

Аналогичным образом размеры элемента могут быть ограничены свойствами `max-width` и `max-height`.

max-width, max-height

Значения:	<длина> | <процентное значение> | `none` | `inherit`
Начальное значение:	`none`
Область применения:	все элементы, кроме незамещаемых строковых элементов и элементов таблиц
Наследование:	нет
Процентные значения:	относительно ширины/высоты блока-контейнера
Вычисляемое значение:	для процентных значений – как задано; для значений, заданных в единицах измерения длины, – абсолютная длина; в противном случае – `none`

Прозрачность имен этих свойств практически устраняет необходимость пояснений. С первого взгляда неочевидно, но если подумать, вполне логично, что значения всех этих свойств не могут быть отрицательными.

Internet Explorer для Windows вплоть до версии IE7 не поддерживал свойства `min-height`, `min-width`, `max-height` и `max-width`.

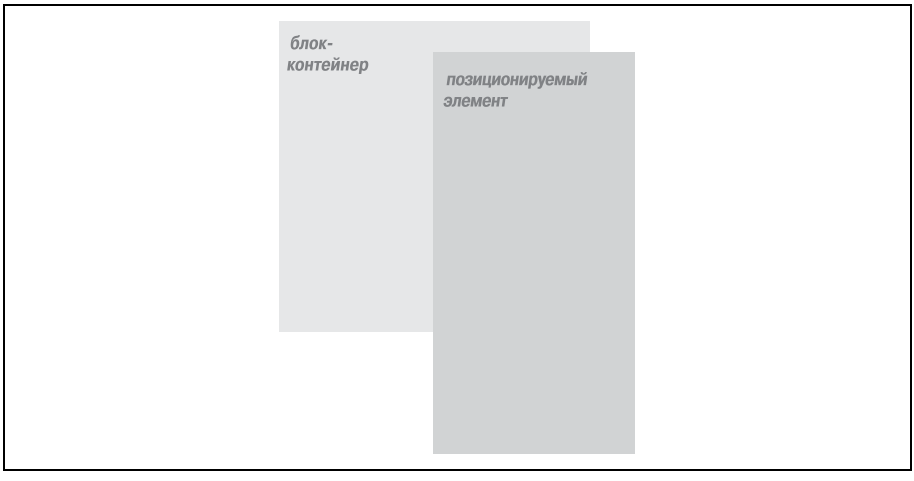


Рис. 10.30. *Задание минимальной ширины и минимальной высоты позиционированного элемента*

Следующие стили зададут для элементов ширину и высоту минимум `10em` и `20em` соответственно, как показано на рис. 10.30:

```
top: 10%; bottom: 20%; left: 50%; right: 10%;
  min-width: 10em; min-height: 20em;
```

Это не очень надежное решение, поскольку размеры элемента оговариваются без учета размера блока-контейнера. Вот вариант получше:

```
top: 10%; bottom: auto; left: 50%; right: 10%; height: auto;
  min-width: 15em;
```

Здесь ширина элемента должна составлять не более 40% ширины блока-контейнера, но не менее `15em`. Размеры `bottom` и `height` также изменены и определяются автоматически. В результате высота элемента будет такой, какая необходима для отображения содержимого независимо от того, насколько узким он становится (но, конечно же, не уже `15em`!).

Роль, которую играет значение `auto` в свойствах `height` и `width` позиционированных элементов, рассматривается в следующем разделе.

Ситуацию можно изменить до обратной и ограничить размеры элемента сверху, задав свойства `max-width` и `max-height`. Рассмотрим вариант, когда по какой-то причине требуется, чтобы элемент составлял три четверти ширины его блока-контейнера, но был не шире 400 пикселов. Для этого подойдут такие стили:

```
left: 0%; right: auto; width: 75%; max-width: 400px;
```

Огромное преимущество свойств минимума-максимума состоит в том, что они позволяют относительно безопасно сочетать единицы измерения. Можно задавать размеры с помощью процентных значений, определяя ограничения при этом с помощью значений, выраженных в единицах измерения длины, или наоборот.

Следует упомянуть, что эти свойства минимума-максимума могут быть очень полезны и для перемещаемых элементов. Так, ширину перемещаемого элемента можно определить относительно ширины его родительского элемента (который является его блоком-контейнером), при этом также гарантируя, что ширина перемещаемого элемента никогда не будет меньше `10em`. Обратный подход также возможен:

```
p.aside {float: left; width: 40em; max-width: 40%;}
```

Здесь для перемещаемого элемента задается ширина `40em`, если только это значение не будет превышать 40% ширины блока-контейнера, а в этом случае перемещаемый элемент будет сужен.

Мы вернемся к вопросу об изменении размеров элемента при обсуждении каждого из типов позиционирования.

Переполнение и отсечение содержимого

Если размер элемента недостаточен для вмещения его содержимого, возникает опасность переполнения самого элемента. В подобных ситуациях CSS2 предлагает на выбор несколько альтернативных решений. Также можно задавать область отсечения, определяя при этом пространство элемента, за пределами которого подобные ситуации порождают трудности.

Переполнение

Итак, предположим, что для некоторого элемента задан определенный размер и содержимое в него не помещается. Ситуацию можно взять под контроль посредством свойства `overflow`.

Применяемое по умолчанию значение `visible` означает, что содержимое элемента будет видимым вне блока элемента. Обычно содержимое

overflow

Значения:	`visible | hidden | scroll | auto | inherit`
Начальное значение:	`visible`
Область применения:	блочные замещаемые элементы
Наследование:	нет
Вычисляемое значение:	как задано

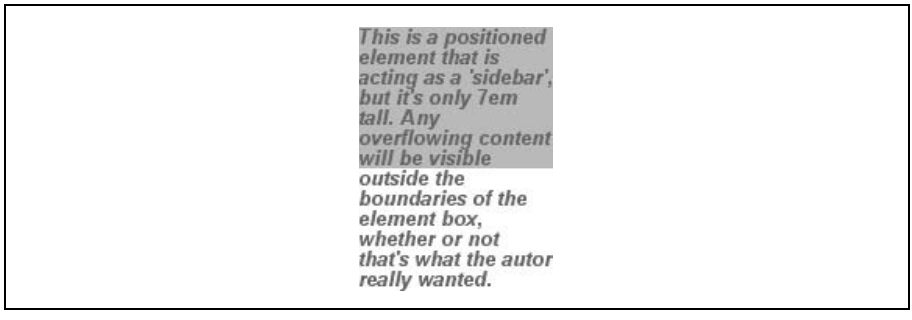


Рис. 10.31. *Видно, как содержимое переполняет блок элемента*

при этом просто выходит за границы собственного блока элемента, но не меняет форму этого блока. Результат применения следующей разметки приведен на рис. 10.31:

```
div#sidebar {position: absolute; top: 0; left: 0; width: 25%; height: 7em;
  background: #BBB; overflow: visible;}
```

Если для свойства `overflow` задано значение `scroll`, содержимое отсекается – т. е. его нельзя увидеть – краями блока элемента, но есть возможность сделать непоместившееся содержимое доступным пользователю. В веб-броузере это могло бы означать появление полосы прокрутки (или набора полос прокрутки) или другого средства доступа к содержимому без изменения формы самого элемента. Один из возможных результатов применения следующей разметки показан на рис. 10.32:

```
div#sidebar {position: absolute; top: 0; left: 0; width: 15%; height: 7em;
  overflow: scroll;}
```

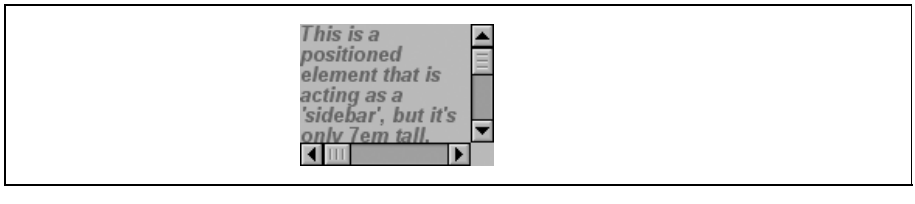


Рис. 10.32. *Непоместившееся содержимое доступно посредством механизма прокрутки*

Если задано значение `scroll`, то механизмы панорамирования (например, полосы прокрутки) должны генерироваться всегда. Цитируем спецификацию: «этим устраняются любые проблемы с появлением или исчезновением полос прокрутки в динамической среде». Таким образом, даже если места для отображения всего содержимого элемента достаточно, полосы прокрутки все равно должны появляться. Кроме того, при распечатке страницы или любом другом представлении документа в печатающем устройстве содержимое может отображаться так, как если бы для свойства `overflow` было задано значение `visible`.

Если для свойства `overflow` задано значение `hidden`, то содержимое элемента отсекается краями блока элемента и интерфейс прокрутки не предоставляется (чтобы содержимое, оказавшееся вне области отсечения, было доступно пользователю). Рассмотрим следующую разметку:

```
div#sidebar {position: absolute; top: 0; left: 0; width: 15%; height: 7em;
  overflow: hidden;}
```

В подобном случае отсеченное содержимое не будет доступно пользователю. Это приведет к ситуации, показанной на рис. 10.33.

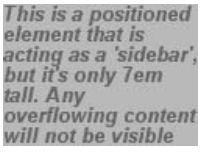


Рис. 10.33. *Отсечение содержимого по краям области содержимого*

И наконец, есть вариант значения `overflow: auto`. Если оно задано, то агенты пользователя могут сами определять, какое поведение выбрать, хотя им предлагается в случае необходимости предоставлять механизм прокрутки. Это перспективный способ устранения трудностей, связанных с переполнением, поскольку агенты пользователя могли бы интерпретировать это, как «предоставлять полосы прокрутки только в случае необходимости» (Не исключено, что они этого не делают, но, определенно, могли бы и, вероятно, должны.)

Отсечение содержимого

Если содержимое абсолютно позиционированного элемента переполняет его блок элемента и свойство `overflow` определено так, что содер-

clip

Значения:	`rect(`*top*`, `*right*`, `*bottom*`, `*left*`) | auto | inherit`
Начальное значение:	`auto`
Область применения:	абсолютно позиционированные элементы (в CSS2 `clip` применяется к блочным и замещаемым элементам)
Наследование:	нет
Вычисляемое значение:	для прямоугольника – набор из четырех вычисляемых длин, представляющих края прямоугольника отсечения; в противном случае – как задано

жимое должно отсекаться, то, задав свойство clip, можно изменить форму области отсечения.

По умолчанию устанавливается значение auto, и в этом случае содержимое элемента не должно отсекаться. Другая возможность состоит в том, чтобы определить форму области отсечения относительно области содержимого элемента. Это не меняет форму области содержимого, но изменяет область, в которой может быть сгенерировано визуальное представление содержимого.

Хотя в CSS2 область отсечения может быть только прямоугольной формы, спецификация предлагает возможность включения в будущие спецификации и других форм.

Это осуществляется с помощью значения формы rect(top, right, bottom, left). Можно задать обычную область отсечения:

```
clip: rect(0, auto, auto, 0);
```

Любопытен синтаксис rect. Формально он может быть таким: rect(*top, right, bottom, left*) – обратите внимание на запятые, но спецификация CSS2 содержит примеры и с запятыми, и без них и определяет обе версии как допустимые. В этой книге я буду придерживаться варианта с запятыми главным образом потому, что он более удобен для чтения, а также потому, что ему отдается предпочтение в CSS2.1.

Исключительно важно отметить, что значения rect(...) – это *не* смещения сторон. Это расстояния от верхнего левого угла элемента (или верхнего правого для языков с написанием справа налево). Таким образом, прямоугольник отсечения, образованный квадратом 20×20 пикселов в верхнем левом углу элемента, определялся бы так:

```
rect(0, 20px, 20px, 0)
```

Для rect(...) допускаются только значения, заданные в единицах измерения длины, и auto, что аналогично совмещению края отсечения с краем содержимого. Таким образом, следующие два выражения означают одно и то же:

```
clip: rect(auto, auto, 10px, 1em);
clip: rect(0, 0, 10px, 1em);
```

Поскольку все смещения в clip отсчитываются от верхнего левого угла и процентные значения не допускаются, создать «центрированную» область отсечения практически невозможно, если неизвестны размеры самого элемента. Рассмотрим:

```
div#sidebar {position: absolute; top: 0; bottom: 50%; right: 50%; left: 0;
  clip: rect(1em,4em,6em,1em);}
```

Поскольку нельзя заранее узнать высоту и ширину элемента, нельзя и определить прямоугольник отсечения, который заканчивается на один em правее или ниже области содержимого элемента. Это можно узнать, только задав высоту и ширину самого элемента:

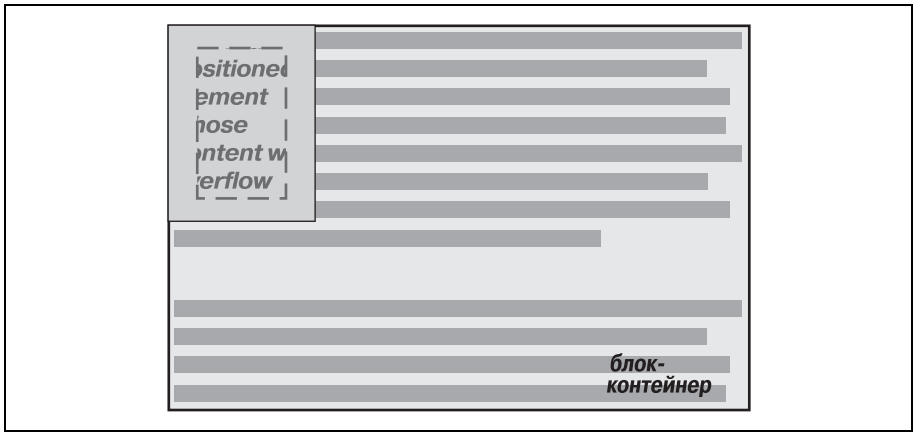


Рис. 10.34. *Задание области отсечения избыточного содержимого*

```
div#sidebar {position: absolute; top: 0; left: 0; width: 5em; height: 7em;
  clip: rect(1em,4em,6em,1em);}
```

Результат подобен приведенному на рис. 10.34. Пунктирная линия здесь добавлена для обозначения краев области отсечения. На самом деле в агенте пользователя, генерирующем визуальное представление документа, этой линии не будет.

Можно задавать и отрицательные значения длины, при этом область отсечения выйдет за границы блока элемента. Для того чтобы выдвинуть область отсечения вверх и влево на четверть дюйма, используются следующие стили (рис. 10.35):

```
clip: rect(-0.25in, auto, auto, -0.25in);
```

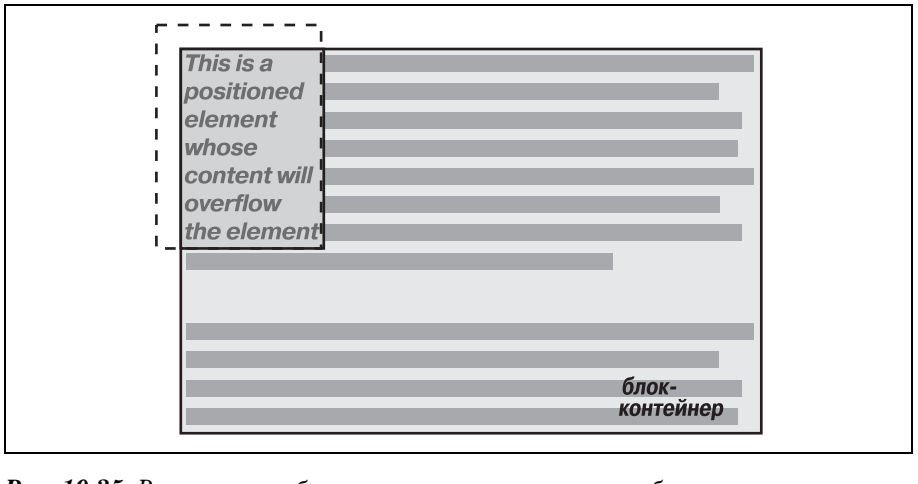


Рис. 10.35. *Расширение области отсечения за границы блока элемента*

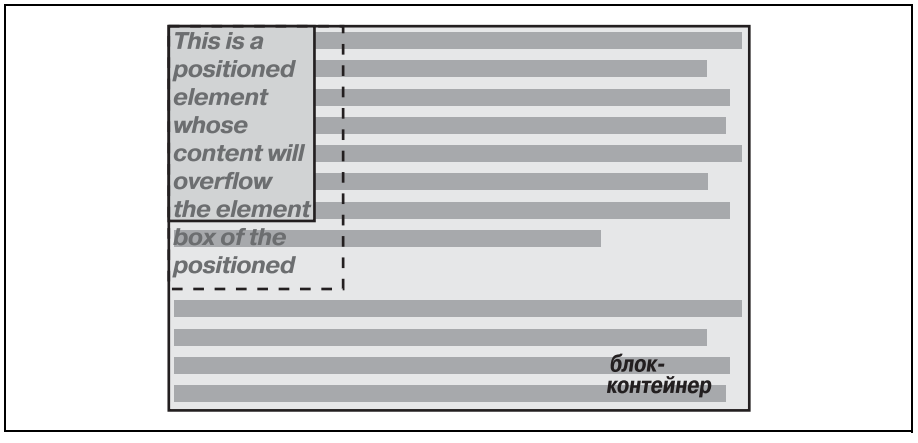


Рис. 10.36. *Расширение области отсечения вниз и вправо от блока элемента*

Как видите, особой пользы от этого нет. Прямоугольник отсечения расширяется вверх и влево, но поскольку там нет содержимого, то разницы почти никакой.

С другой стороны, было бы неплохо расширить прямоугольник отсечения снизу и справа, а не сверху или слева. На рис. 10.36 показаны результаты применения этих стилей (и помните, пунктирные линии проведены только в иллюстративных целях!):

```
div#sidebar {position: absolute; top: 0; left: 0; width: 5em; height: 7em;
  clip: rect(0,6em,9em,0);}
```

Тем самым расширяется область, в которой можно видеть содержимое. Однако это не меняет потока содержимого, так что единственный визуальный эффект состоит в том, что под элементом можно увидеть больше содержимого. Текст не распространяется вправо, потому что ширина его контейнеров строк по-прежнему ограничена шириной позиционированного элемента. Если бы на этом месте находилось изображение более широкое, чем позиционированный элемент, или форматированный текст с длинной строкой, их можно было бы увидеть справа от позиционированного элемента, вплоть до края прямоугольника отсечения.

Синтаксис `rect(...)`, как вы могли заметить, довольно необычен в сравнении с остальным CSS. Он происходит от ранних проектов раздела позиционирования, основывавшихся на схеме смещения от верхнего левого угла. Internet Explorer реализовал ее до того, как CSS2 стал принятой рекомендацией, и таким образом вступил в конфликт с самыми последними изменениями, которые требуют от `rect(...)` использовать смещения сторон, как это происходит во всем остальном CSS2. Это было довольно разумно, т. к. обеспечивало позиционированию непротиворечивость.

К тому времени, однако, было слишком поздно: продукт уже вышел на рынок, и вместо того чтобы заставлять Microsoft менять броузер и таким образом создать угрозу разрушения уже созданных страниц, был изменен стандарт, соответствующий реализации продукта. К несчастью, как мы видели ранее, это значит, что задавать согласованный прямоугольник отсечения в ситуациях, когда высота и ширина не определены строго, невозможно.

Еще больше обостряет проблему то, что `rect(...)` принимает только значения, заданные в единицах измерения длины, и `auto`. Добавление процентных значений как действительных значений `rect(...)` сделало бы огромный вклад в улучшение сложившейся ситуации, и, будем надеяться, следующие версии CSS введут эту возможность.

Длинная и извилистая история `clip` означает, что в современных броузерах его поведение противоречиво, и на него нельзя полагаться в любой среде, предполагаемой для работы в нескольких броузерах.

Видимость элементов

Кроме отсечения и переполнения, можно также управлять видимостью всего элемента.

Тут все довольно просто. Если для элемента задано `visibility: visible`, то он, конечно же, видимый.

Если для элемента задано `visibility: hidden`, то он становится «невидимым» (invisible, формулировка из спецификации). В состоянии невидимости элемент все еще оказывает влияние на компоновку документа так, как если бы был видимым. Иначе говоря, элемент находится на своем месте, но мы его не видим. Заметьте отличие от объявления `display: none`. В последнем случае элемент не отображается и удаляется из документа, поэтому никак не влияет на его компоновку. На рис. 10.37 показан документ, в котором абзац определен как `hidden` на основании следующих стилей и разметки:

```
em.trans {visibility: hidden; border: 3px solid gray; background: silver;
  margin: 2em; padding: 1em;}

<p>
```

visibility

Значения:	`visible | hidden | collapse | inherit`
Начальное значение:	`visible`
Область применения:	все элементы
Наследование:	да
Вычисляемое значение:	как задано

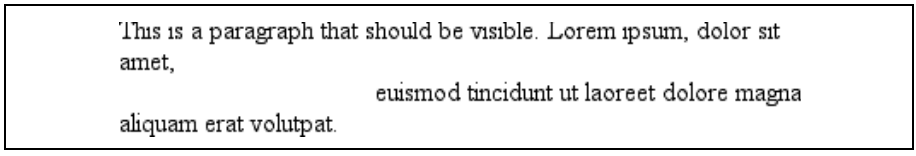


Рис. 10.37. *Делаем элементы невидимыми без удаления их блоков*

```
This is a paragraph that should be visible. Lorem ipsum, dolor sit amet,
<em class="trans">consectetuer adipiscing elit, sed diam nonummy nibh </em>
euismod tincidunt ut laoreet dolore magna aliquam erat volutpat.
</p>
```

Все видимые детали – такие как содержимое, фон и рамки – для скрытого элемента станут невидимыми. Обратите внимание, что остается пустое пространство, потому что элемент все еще является частью компоновки документа. Вы просто не можете видеть его.

Также заметьте, что элементы-потомки невидимого элемента можно делать видимыми. Это обусловливает появление элемента там, где он должен быть, несмотря на тот факт, что его предок (и, возможно, сестринские элементы) невидимы. Для этого надо прямо объявить элемент-потомок как `visible`, поскольку свойство `visibility` – наследуемое:

```
p.clear {visibility: hidden;}
p.clear em {visibility: visible;}
```

Что касается `visbility: collapse`, это значение используется в CSS при генерировании визуального представления таблиц, что рассматривается в следующей главе. Согласно спецификации CSS2, если значение `collapse` применяется для нетабличных элементов, его поведение аналогично `hidden`.

Абсолютное позиционирование

Большинство примеров и рисунков предыдущих разделов иллюстрируют абсолютное позиционирование, поэтому вы уже на полпути к пониманию его принципа. Осталось только рассмотреть детали применения абсолютного позиционирования.

Блоки-контейнеры и абсолютно позиционированные элементы

Абсолютно позиционированный элемент полностью удаляется из потока документа. Он позиционируется относительно его блока-контейнера, и местоположение его краев определяется с помощью свойств смещения (`top`, `left` и др.). Позиционированный элемент не обтекает содержимое других элементов, а их содержимое не обтекает позиционированный элемент. Это означает, что абсолютно позиционированный элемент может перекрывать другие элементы или быть перекрытым

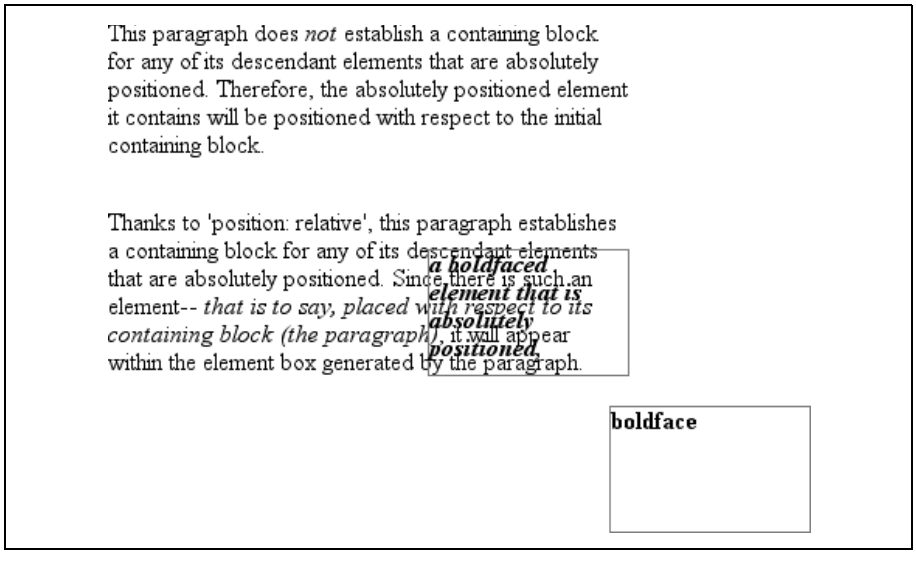


Рис. 10.38. *Использование относительного позиционирования для определения блока-контейнера*

ими. (Способы влияния на порядок наложения рассмотрены в этой главе ниже.)

Блок-контейнер абсолютно позиционированного элемента – это ближайший элемент-предок, значение свойства `position` которого отлично от `static`. Обычно автор выбирает элемент, который будет выступать в роли блока-контейнера абсолютно позиционированного элемента, и задает для его свойства `position` значение `relative` без смещений:

```
p.contain {position: relative;}
```

Рассмотрим пример на рис. 10.38, иллюстрирующий действие следующей разметки:

```
p {margin: 2em;}
p.contain {position: relative;}   /* определяет блок-контейнер */
b {position: absolute; top: auto; right: 0; bottom: 0; left: auto;
   width: 8em; height: 5em; border: 1px solid gray;}
```

```
<body>
<p>
This paragraph does <em>not</em> establish a containing block for any of its
descendant elements that are absolutely positioned.  Therefore, the
absolutely positioned <b>boldface </b> element it contains will be positioned
with respect to the initial containing block.
</p>
<p class="contain">
Thanks to 'position: relative', this paragraph establishes a containing block
for any of its descendant elements that are absolutely positioned.  Since
```

```
there is such an element-- <em>that is to say, <b>a boldfaced element that is
absolutely positioned,</b> placed with respect to its containing block (the
paragraph)</em>, it will appear within the element box generated by the
paragraph.
</p>
</body>
```

Элементы `b` обоих абзацев позиционированы абсолютно. Они отличаются блоком-контейнером, используемым для каждого. Элемент `b` первого абзаца позиционирован относительно начального блока-контейнера, потому что свойство `position` всех его элементов-предков было `static`. Однако для второго абзаца было задано `position: relative`, поэтому он образует блок-контейнер для своих потомков.

Возможно, вы обратили внимание, что во втором абзаце позиционированный элемент перекрывает часть текстового содержимого абзаца. Избежать этого можно, только разместив элемент `b` вне абзаца (с помощью отрицательного значения свойства `right` или какого-то другого из свойств смещения) или определив для абзаца отступ, достаточно широкий, чтобы вместить позиционированный элемент. Кроме того, через прозрачный фон элемента `b` просматривается текст абзаца. Единственная возможность избежать этого – задать фон позиционированного элемента или полностью убрать его из абзаца.

Иногда требуется, чтобы начальный блок-контейнер выбирался не агентом пользователя, – тогда роль блока-контейнера для всех его потомков выполняет элемент `body`. Для этого достаточно объявить:

```
body {position: relative;}
```

В подобный документ можно ввести абсолютно позиционированный абзац, как показано ниже, и получить результаты, приведенные на рис. 10.39:

```
<p style="position: absolute; top: 0; right: 25%; left: 25%; bottom: auto;
 width: 50%; height: auto; background: silver;">...</p>
```

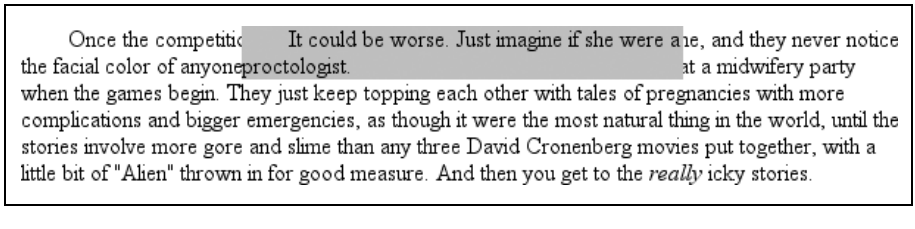


Рис. 10.39. *Позиционирование элемента, блоком-контейнером которого является корневой элемент*

Теперь абзац расположен в самом начале документа, составляет половину ширины документа и перекрывает несколько первых элементов.

Важно отметить, что абсолютно позиционированный элемент также образует блок-контейнер для своих элементов-потомков. Например,

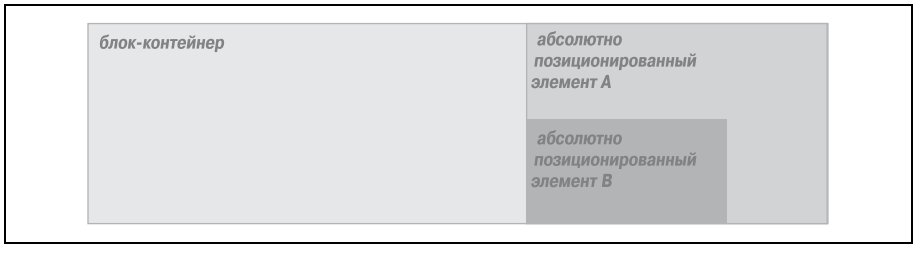


Рис. 10.40. *Абсолютно позиционированные элементы образуют блоки-контейнеры*

можно абсолютно позиционировать элемент и затем абсолютно позиционировать один из его дочерних элементов, как показано на рис. 10.40, который был сгенерирован с применением следующих стилей и базовой разметки:

```
div {position: relative; width: 100%; height: 10em;
   border: 1px solid; background: #EEE;}
div.a {position: absolute; top: 0; right: 0; width: 15em; height: 100%;
   margin-left: auto; background: #CCC;}
div.b {position: absolute; bottom: 0; left: 0; width: 10em; height: 50%;
   margin-top: auto; background: #AAA;}
```

```
<div>
  <div class="a">absolutely positioned element A
    <div class="b">absolutely positioned element B</div>
  </div>
  containing block
</div>
```

Помните, что, если документ прокручивается, позиционированные элементы будут прокручиваться вместе с ним. Это справедливо для всех абсолютно позиционированных элементов, которые не являются потомками элементов, позиционированных фиксированно. Причина в том, что в конечном счете элементы позиционированы относительно чего-то, что является частью нормального потока. Так, если блоком-контейнером абсолютно позиционированной таблицы является начальный блок-контейнер, она будет прокручиваться, потому что начальный блок-контейнер – это часть нормального потока. Аналогично, даже если заданы четыре уровня вложенности абсолютно позиционированных элементов, самый внешний из этих элементов позиционирован относительно начального блока-контейнера. Поэтому он будет прокручиваться вместе с начальным блоком-контейнером, и все его потомки вместе с ним.

Если вы хотите позиционировать элементы так, чтобы они размещались относительно окна просмотра и не прокручивались вместе с остальным документом, читайте дальше. Об этом рассказывается в разделе, посвященном фиксированному позиционированию.

Размещение и изменение размеров абсолютно позиционированных элементов

Сочетание понятий размещения и изменения размеров может показаться странным, но такова необходимость для абсолютно позиционированных элементов, потому что спецификация очень тесно объединила эти два процесса. При более детальном рассмотрении это не такое уж странное сочетание. Вот что происходит, если элемент позиционируется с объявлением всех четырех свойств смещения, вот так:

```
#masthead h1 {position: absolute; top: 1em; left: 1em; right: 25%; bottom: 10px;
  margin: 0; padding: 0; background: silver;}
```

Здесь высота и ширина блока элемента `h1` определяются расположением внешних краев его полей, как показано на рис. 10.41.

***Рис. 10.41.** Определение высоты и ширины элемента на основании свойств смещения*

Если увеличить высоту блока-контейнера, то и `h1` станет выше, а если уменьшить ширину блока-контейнера, то и `h1` станет уже. А если добавить в `h1` поля или отступы, это повлияет на вычисляемую высоту и ширину `h1`.

Но что будет, если кроме всего этого попытаться явно задать высоту и ширину?

```
#masthead h1 {position: absolute; top: 0; left: 1em; right: 10%; bottom: 0;
  margin: 0; padding: 0; height: 1em; width: 50%; background: silver;}
```

Чем-то придется пожертвовать, потому что совершенно невероятно, чтобы все эти значения были правильными. Кстати, чтобы все приведенные значения были верными, блок-контейнер должен быть в два с половиной раза шире, чем вычисляемое значение свойства `font-size` для `h1`. Любая другая ширина означала бы, что по крайней мере одно из значений ошибочно и должно быть проигнорировано. Точное определение этого значения зависит от ряда факторов, а факторы меняются в зависимости от того, о каком элементе идет речь – замещаемом или незамещаемом.

В связи с этим рассмотрим следующее:

```
#masthead h1 {position: absolute; top: auto; left: auto;}
```

Каким должен быть результат? Ответ для этого случая: «значения установлены в нуль» *не будут*. Настоящий ответ мы найдем в следующем разделе.

Края, заданные значением auto

При абсолютном позиционировании элемента существует специальное поведение, которое применяется, когда всем свойствам смещения, кроме `bottom`, присваивается значение `auto`. В качестве примера возьмем свойство `top`:

```
<p>
When we consider the effect of positioning, it quickly becomes clear that
authors can do a great deal of damage to layout, just as they can do very
interesting things.<span style="position: absolute; top: auto; left: 0;"
>[4]</span>
This is usually the case with useful technologies: the sword always has at
least two edges, both of them sharp.
</p>
```

Что должно произойти? Для свойства `left` все просто: левый край элемента должен быть помещен вдоль левого края его блока-контейнера (который можно принять за начальный блок-контейнер). Однако для свойства `top` все намного интереснее. Верх позиционированного элемента должен выравниваться по тому месту, где находился бы его верх, если бы он вообще не был позиционирован. Иначе говоря, представьте, куда был бы помещен `span`, если бы его свойство `position` имело значение `static`; это его *статическое положение* (*static position*) – то место, куда в результате вычислений должен быть помещен его верхний край. Вот что говорится в CSS2.1:

> ...термин «статическое положение» (элемента) относится, в общих чертах, к местоположению, которое занимал бы элемент в нормальном потоке. Более строго, статическое положение для «top» – это расстояние от верхнего края блока-контейнера до края верхнего поля гипотетического блока, который был бы первым блоком элемента, если бы его свойство `position` имело значение `static`. Значение отрицательное, если гипотетический блок находится выше блока-контейнера.

Следовательно, должен получиться результат, как на рис. 10.42.

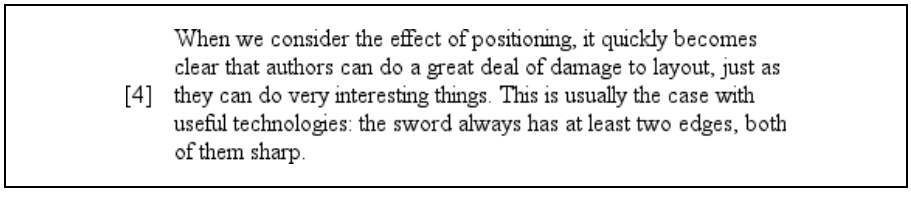


Рис. 10.42. *Абсолютное позиционирование элемента согласно его «статическому» положению*

Элемент «[4]» находится за пределами содержимого абзаца, потому что левый край его начального блока-контейнера расположен левее левого края абзаца.

Аналогичны основные правила и для свойств `left` и `right`, для которых задано значение `auto`. В этих случаях левый (или правый) край позиционированного элемента выравнивается по тому месту, где находился бы край, если бы элемент не был позиционирован. Модифицируем наш предыдущий пример так, чтобы и свойство `top`, и свойство `left` имели значение `auto`:

```
<p>
When we consider the effect of positioning, it quickly becomes clear that
authors can do a great deal of damage to layout, just as they can do very
interesting things.<span style="position: absolute; top: auto; left: auto;"
>[4]</span>
This is usually the case with useful technologies: the sword always has at
least two edges, both of them sharp.
</p>
```

Результат показан на рис. 10.43.

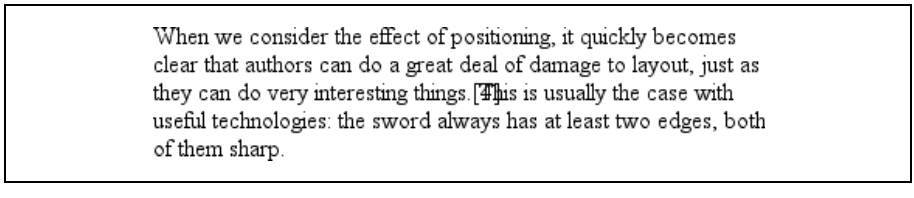


Рис. 10.43. *Абсолютное позиционирование элемента согласно его «статическому» положению*

Теперь текст «[4]» находится прямо там, где был бы, если бы не был позиционирован. Обратите внимание, что поскольку он позиционирован, пространство, занимаемое им в нормальном потоке, ликвидируется. Это и есть причина перекрытия позиционированным элементом содержимого, находящегося в нормальном потоке.

Необходимо отметить, что и CSS2, и CSS2.1 утверждают, что в подобных случаях «...агенты пользователя вольны делать предположения по поводу его возможного [статического] положения». Современные броузеры довольно хорошо обрабатывают значения `auto` свойств `top` и `left` и размещают элемент согласно местоположению, которое он занимал бы в нормальном потоке.

Это автоматическое размещение действует только в определенных ситуациях, обычно там, где наложены некоторые ограничения на остальные размеры позиционированного элемента. Автоматическое размещение стало возможным в нашем предыдущем примере, потому что в нем не было ограничений на высоту или ширину и не ограничива-

лось размещение нижнего и правого краев. Но предположим, что по какой-то причине такие ограничения есть. Рассмотрим:

```
<p>
When we consider the effect of positioning, it quickly becomes clear that
authors can do a great deal of damage to layout, just as they can do very
interesting things. <span style="position: absolute; top: auto; left: auto;
right: 0; bottom: 0; height: 2em; width: 5em;">[4]</span>  This is usually
the case with useful technologies: the sword always has at least two edges,
both of them sharp.
</p>
```

Требования всех этих значений удовлетворить невозможно. Разговор о том, что происходит в этом случае, – тема следующего раздела.

Размещение и изменение размеров незамещаемых элементов

В общем случае размер и размещение элемента зависят от его блока-контейнера. Значения его различных свойств (`width`, `right`, `padding-left` и т. д.) влияют на ситуацию, но главную роль играет блок-контейнер.

Рассмотрим ширину и горизонтальное размещение позиционированного элемента. Оно может быть представлено как уравнение `left` + `margin-left` + `border-left-width` + `padding-left` + `width` + `padding-right` + `border-right-width` + `margin-right` + `right` = ширина блока-контейнера.

Расчет довольно обоснованный. Это, по существу, уравнение, по которому определяются размеры блочных элементов в нормальном потоке, за исключением того, что сюда введены еще `left` и `right`. Как же все это взаимодействует? Необходимо проработать ряд правил.

Во-первых, если для всех свойств `left`, `width` и `right` задать значение `auto`, получится результат, приведенный в предыдущем разделе: левый край помещен в его статическое положение (если это язык с написанием слева направо). В языках с написанием справа налево свое статическое положение занимает правый край. Свойство `width` элемента определено как «обжимающее», т. е. ширина области содержимого элемента будет не больше необходимого для вмещения содержимого. Очень похоже ведут себя ячейки таблицы. Свойство, не определяющее статического положения (`right` в языках с написанием слева направо, `left` в языках с написанием справа налево), задается так, чтобы занималось оставшееся пространство. Например:

```
<div style="position: relative; width: 25em; border: 1px dotted;">
An absolutely positioned element can have its content
<span style="position: absolute; top: 0; left: 0; right: auto; width: auto;
  background: silver;">shrink-wrapped</span>
thanks to the way positioning rules work.
</div>
```

Результат показан на рис. 10.44.

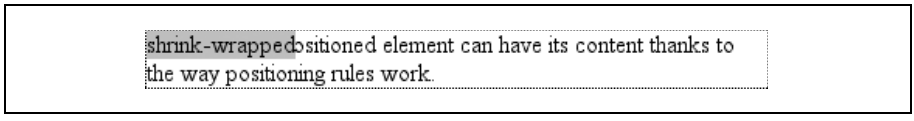


Рис. 10.44. *«Обжимающее» поведение абсолютно позиционированных элементов*

Верх элемента совпадает с верхом его блока-контейнера (в данном случае `div`), и ширина элемента именно такая, какая нужна для вмещения содержимого. Оставшееся расстояние от правого края элемента до правого края блока-контейнера становится вычисляемым значением свойства `right`.

Теперь предположим, что значение `auto` задано только для левого и правого полей, а не для `left`, `width` и `right`, как в данном примере:

```
<div style="position: relative; width: 25em; border: 1px dotted;">
An absolutely positioned element can have its content
<span style="position: absolute; top: 0; left: 1em; right: 1em; width: 10em;
  margin: 0 auto; background: silver;">shrink-wrapped</span>
thanks to the way positioning rules work.
</div>
```

Здесь левое и правое поля, оба имеющие значение `auto`, уравниваются. В результате элемент будет центрирован, как показано на рис. 10.45.

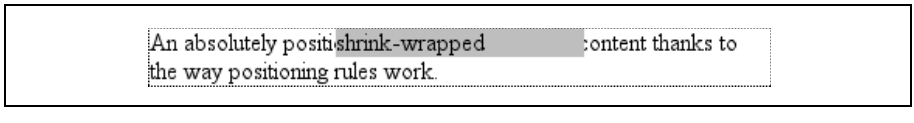


Рис. 10.45. *Горизонтальное центрирование абсолютно позиционированного элемента при значении полей auto*

Это, по сути, аналогично центрированию элементов с автоматически вычисляемыми полями в нормальном потоке. Итак, сделаем поля отличными от `auto`:

```
<div style="position: relative; width: 25em; border: 1px dotted;">
An absolutely positioned element can have its content
<span style="position: absolute; top: 0; left: 1em; right: 1em; width: 10em;
  margin-left: 1em; margin-right: 1em; background: silver;">shrink-wrapped</
span>
thanks to the way positioning rules work.
</div>
```

Теперь возникла проблема. Свойства позиционирования элемента `span` в сумме дают всего лишь `14em`, тогда как ширина блока-контейнера – `25em`. Очевиден дефицит в `11em`, который надо как-то компенсировать.

Правила утверждают, что в этом случае агент пользователя игнорирует значение свойства `right` (в языках с написанием слева направо; в про-

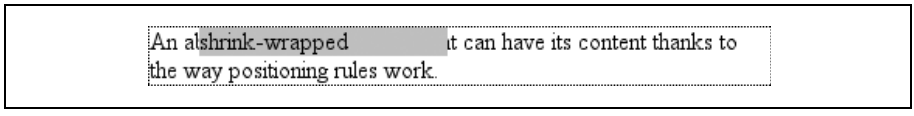


Рис. 10.46. *Значение свойства right в сверхограниченной ситуации игнорируется*

тивном случае он игнорирует `left`) и вычисляет его. Иначе говоря, аналогичный результат (рис. 10.46) мы получили бы, если бы объявили:

```
<span style="position: absolute; top: 0; left: 1em; right: 12em; width: 10em;
margin-left: 1em; margin-right: 1em; background: silver;">shrink-wrapped</ span>
```

Если бы для одного из полей было оставлено значение `auto`, менялось бы оно. Предположим, стили изменены так:

```
<span style="position: absolute; top: 0; left: 1em; right: 1em; width: 10em;
margin-left: 1em; margin-right: auto; background: silver;">shrink-wrapped</
span>
```

Визуально результат будет аналогичен представленному на рис. 10.46, только он будет получен путем вычисления значения правого поля в `12em`, а не переопределения значения свойства `right`. Если же присвоить `auto` левому полю, тогда *оно* будет вычислено, как показано[1] на рис. 10.47:

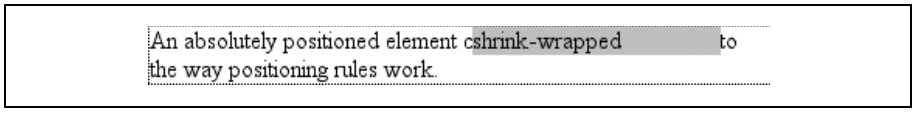


Рис. 10.47. *Значения свойств margin в сверхограниченной ситуации*

```
<span style="position: absolute; top: 0; left: 1em; right: 1em; width: 10em;
  margin-left: auto; margin-right: 1em; background: silver; ">shrink-wrapped</
span>
```

В общем, если значение `auto` задано только для одного свойства, то именно оно будет меняться, чтобы удовлетворить уравнению, приведенному ранее в этом разделе. В соответствии со следующими стилями элемент по ширине развернется до любого нужного размера, а не «обожмет» содержимое:

```
<span style="position: absolute; top: 0; left: 1em; right: 1em; width: auto;
margin-left: 1em; margin-right: 1em; background: silver;">shrink-wrapped</ span>
```

До сих пор нас интересовали только события, происходящие на горизонтальной оси, но очень похожие правила действуют и для верти-

[1] Отображение зависит от агента пользователя. Показано для броузера Firefox 1.5.0.7. Но, например, для IE 6.0 результат соответствует значению `margin-left`: 0. – *Примеч. науч.ред.*

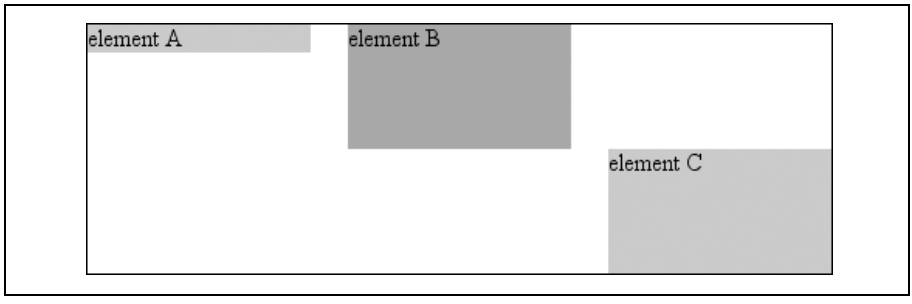


Рис. 10.48. *Поведение при вертикальной компоновке абсолютно позиционированных элементов*

кальной. Если в предыдущем обсуждении повернуть все параметры на 90°, получится практически аналогичное поведение. Например, на рис. 10.48 представлен результат применения следующей разметки:

```
<div style="position: relative; width: 30em; height: 10em;
  border: 1px solid;">
<div style="position: absolute; left: 0; width: 30%; background: #CCC; top: 0;">
element A
</div>
<div style="position: absolute; left: 35%; width: 30%; background: #AAA;
  top: 0; height: 50%;">
element B
</div>
<div style="position: absolute; left: 70%; width: 30%; background: #CCC;
  height: 50%; bottom: 0;">
element C
</div>
</div>
```

В первом случае высота элемента была такой, что он «обжимал» содержимое. Во втором случае не заданное явно свойство (`bottom`) было установлено, чтобы восполнить расстояние между низом позиционированного элемента и низом его блока-контейнера. В третьем случае не было определено свойство `top`, и поэтому разницу компенсировало оно.

В этом случае применение автоматически определяемых полей может привести к вертикальному центрированию. Следующие стили определяют, что абсолютно позиционированный `div` будет вертикально центрирован в рамках его блока-контейнера, как показано на рис. 10.49:

```
<div style="position: relative; width: 10em; height: 10em;
  border: 1px solid;">
<div style="position: absolute; left: 0; width: 100%; background: #CCC;
  top: 0; height: 5em; bottom: 0; margin: auto;">
element D
</div>
</div>
```

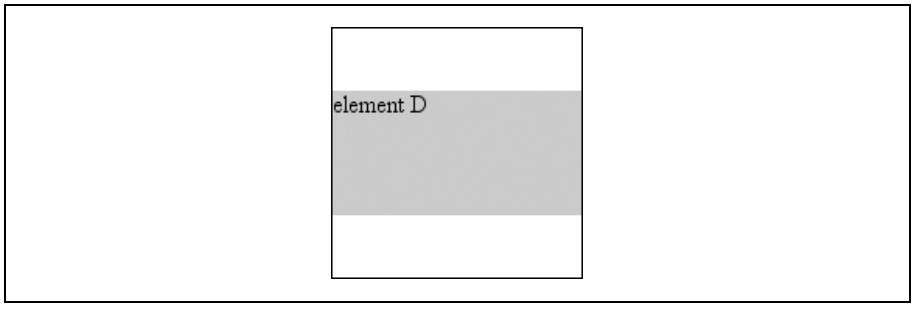


Рис. 10.49. *Вертикальное центрирование абсолютно позиционированного элемента с автоматически определяемыми полями*

Необходимо отметить два небольших отличия. При горизонтальной компоновке или `right`, или `left`, если имеют значение `auto`, могут размещаться соответственно статическому положению. При вертикальном только `top` может занимать статическое положение; `bottom` не может в любом случае.

На момент написания данной книги ни одна из версий Internet Explorer, включая IE7, не поддерживала вертикальное центрирование абсолютно позиционированных элементов при помощи значения `auto` их верхнего и нижнего полей.

Кроме того, если размер абсолютно позиционированного элемента сверхограничен в вертикальном направлении, значение `bottom` игнорируется. Таким образом, в следующей ситуации объявленное значение `bottom` было бы переопределено вычисленным значением `5em`:

```
<div style="position: relative; width: 10em; height: 10em;
  border: 1px solid;">
<div style="position: absolute; left: 0; width: 100%; background: #CCC;
  top: 0; height: 5em; bottom: 0; margin: 0;">
element D
</div>
</div>
```

Средств обеспечения игнорирования значения `top` в случае сверхограниченности свойств нет.

Размещение и изменение размеров замещаемых элементов

Правила позиционирования для незамещаемых элементов отличаются от правил для замещаемых элементов. Причина в том, что замещаемые элементы имеют собственную высоту и ширину и, следовательно, меняют свои размеры, только если прямо определяются автором. Таким образом, в позиционировании замещаемых элементов не действует принцип «обжима».

Схемы поведения, связанные с размещением и изменением размеров замещаемых элементов, проще всего выразить рядом последовательных правил:

1. Если ширине присвоено значение `auto`, то ее реальное значение определяется шириной содержимого элемента. Таким образом, если ширина изображения составляет 50 пикселов, то и реальное значение будет равно `50px`. Если ширина объявлена явно (т. е. что-то вроде `100px` или `50%`), то именно это значение ей и присваивается.
2. В языках с написанием слева направо статическим положением заменяется значение `auto` свойства `left`. В языках с написанием справа налево статическим положением заменяем значение `auto` для свойства `right`.
3. Если или `left`, или `right` все еще имеют значение `auto` (иначе говоря, оно не было заменено на предыдущем шаге), заменяем нулем значение `auto` любого из свойств `margin-left` или `margin-right`.
4. Если на данный момент оба свойства, `margin-left` и `margin-right`, все еще имеют значения `auto`, уравниваем их, таким образом центрируя элемент в его блоке-контейнере.
5. И наконец, если осталось всего одно значение `auto`, меняем его так, чтобы уравнять оставшуюся часть уравнения.

В результате поведение макета в основном будет таким же, как то, которое мы наблюдали для абсолютно позиционированных незамещаемых элементов, когда предполагали, что ширина незамещаемого элемента задана явно. Поэтому следующие два элемента будут иметь одинаковую ширину и размещение (ширина изображения составляет 100 пикселов, см. рис. 10.50):

```
<div style="width: 200px; height: 50px; border: 1px dotted gray;">
 <img src="frown.gif" alt="a frowny face"
  style="position: absolute; top: 0; left: 50px; margin: 0;">
 <div style="position: absolute; top: 0; left: 50px;
   width: 100px; height: 100px; margin: 0;">
   it's a div!
 </div>
</div>
```

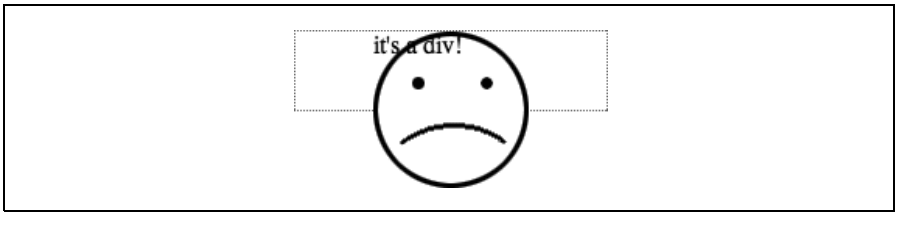


Рис. 10.50. *Абсолютное позиционирование замещаемого элемента*

Как и в случае незамещаемых элементов, если значения сверхограничены, агент пользователя обязан игнорировать значение свойства `right` в языках с написанием слева направо и значение свойства `left` в языках с написанием справа налево. Таким образом, в следующем примере объявленное значение `right` переопределяется вычисляемым значением `50px`:

```
<div style="position: relative; width: 300px;">
 <img src="frown.gif" alt="a frowny face"
  style="position: absolute; top: 0; left: 50px; right: 125px; width: 200px;
    margin: 0;">
</div>
```

Аналогично компоновка вдоль вертикальной оси подчиняется ряду правил, которые утверждают:

1. Если для свойства `height` задано значение `auto`, то вычисляемое значение высоты определяется собственной высотой содержимого элемента. Таким образом, вычисляемое значение `height` для изображения 50 пикселов высотой составляет `50px`. Если высота объявлена явно (т. е. что-то вроде `100px` или `50%`), то свойству `height` присваивается это значение.
2. Если `top` имеет значение `auto`, заменяем его статическим положением замещаемого элемента.
3. Если `bottom` имеет значение `auto`, заменяем любое значение `auto` свойств `margin-top` или `margin-bottom` нулем.
4. Если на данный момент и `margin-top`, и `margin-bottom` все еще имеют значение `auto`, уравниваем их, таким образом центрируя элемент в его блоке-контейнере.
5. И наконец, если осталось всего одно значение `auto`, меняем его так, чтобы уравнять оставшуюся часть уравнения.

Как и в случае незамещаемых элементов, если значения сверхограничены, агент пользователя обязан игнорировать значение свойства `bottom`.

Таким образом, результат применения следующей разметки был бы таким, как показано на рис. 10.51:

```
<div style="position: relative; height: 200px; width: 200px; border: 1px
solid;">
<img src="one.gif" alt="one" width="25" height="25"
  style="position: absolute;
  top: 0; left: 0; margin: 0;">
<img src="two.gif" alt="two" width="25" height="25"
  style="position: absolute;
   top: 0; left: 60px; margin: 10px 0; bottom: 4377px;">
<img src="three.gif" alt=" three" width="25" height="25"
  style="position: absolute;
   left: 0; width: 100px; margin: 10px; bottom: 0;">
<img src="four.gif" alt=" four" width="25" height="25"
  style="position: absolute;
```

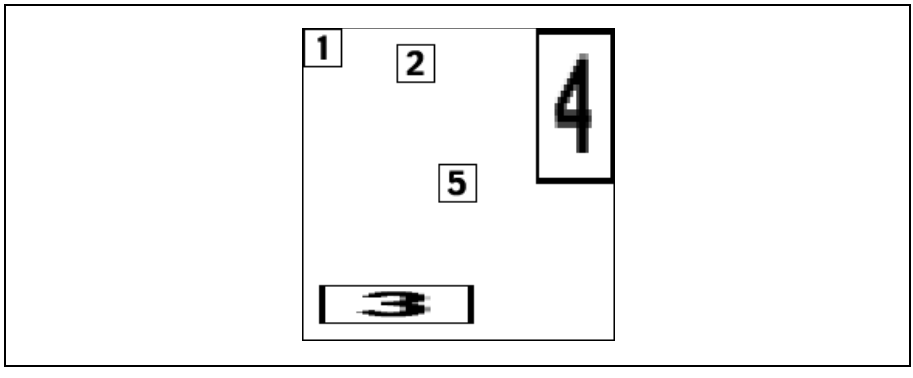


***Рис. 10.51.** Растягивание замещаемых элементов посредством позиционирования*

```
    top: 0; height: 100px; right: 0; width: 50px;">
<img src="five.gif" alt="five" width="25" height="25"
  style="position: absolute;
    top: 0; left: 0; bottom: 0; right: 0; margin: auto;">
</div>
```

Размещение относительно оси z

Продолжая разговор о позиционировании, неизбежно возникнут ситуации, когда два элемента попытаются сосуществовать на одном месте с визуальной точки зрения. Очевидно, одному из них придется перекрывать другой, но как управлять тем, какой элемент окажется сверху? Вот где в игру вступает свойство `z-index`.

Свойство `z-index` позволяет менять способ перекрытия элементов друг другом. Свое имя оно получило от системы координат, в которой горизонталь – это ось x, а вертикаль – ось y. В этом случае третья ось, которая проходит по нормали к поверхности отображения, обозначается как ось z. Таким образом, элементам присваиваются значения координат по этой оси, и представлены эти значения с помощью свойства `z-index`. Рис. 10.52 иллюстрирует эту систему.

В этой системе координат элемент, значение `z-index` которого больше, располагается ближе к читателю, чем те, значение `z-index` которых

z-index

Значения:	<целое> | `auto` | `inherit`
Начальное значение:	`auto`
Область применения:	позиционированные элементы
Наследование:	нет
Вычисляемое значение:	как задано

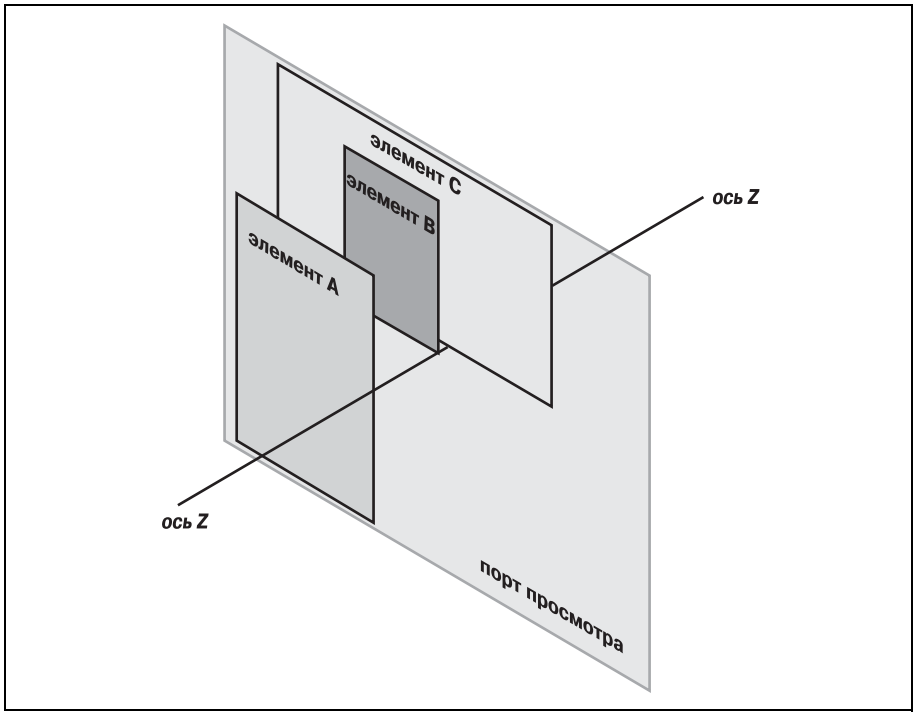


Рис. 10.52. *Концептуальное представление процесса занесения в стек посредством свойства z-index*

меньше. В результате элемент с большим значением перекрывает остальные, как показано на рис. 10.53, являющимся «видом сверху» представления рис. 10.52. Этот процесс называется *занесением в стек* (*stacking*).

В качестве значения `z-index` может выступать любое целое, включая отрицательные числа. Присвоение элементу отрицательного `z-index` отодвинет его от читателя, т. е. он будет помещен в стек глубже. Рассмотрим следующие стили, действие которых проиллюстрировано на рис. 10.54:

```
p#first {position: absolute; top: 0; left: 0;
  width: 20%; height: 10em; z-index: 8;}
p#second {position: absolute; top: 0; left: 10%;
  width: 30%; height: 5em; z-index: 4;}
p#third {position: absolute; top: 15%; left: 5%;
  width: 15%; height: 10em; z-index: 1;}
p#fourth {position: absolute; top: 10%; left: 15%;
  width: 40%; height: 10em; z-index: 0;}
```

Каждый элемент позиционируется в соответствии с его стилями, но значения `z-index` меняют обычный порядок занесения в стек. Если абзацы располагаются в соответствии с их номерами, то логичным был

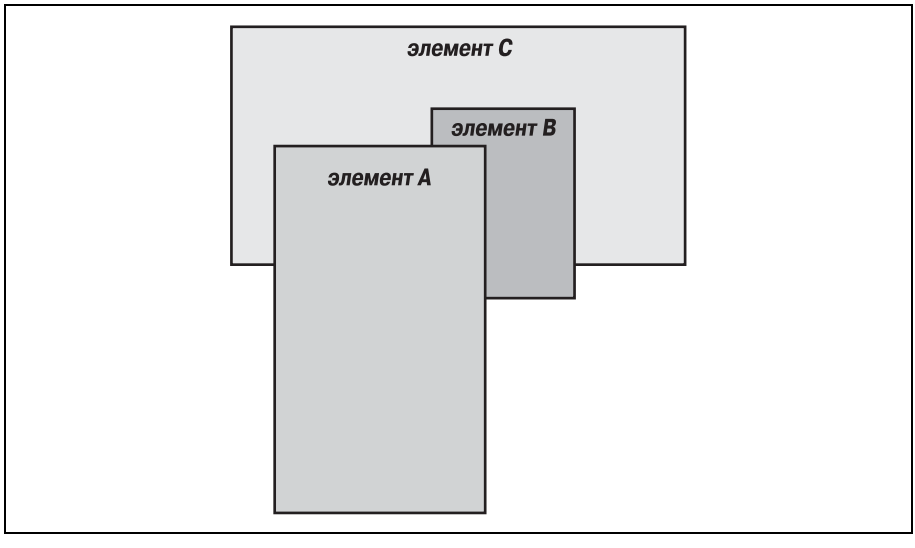


Рис. 10.53. *Как происходит занесение элементов в стек*

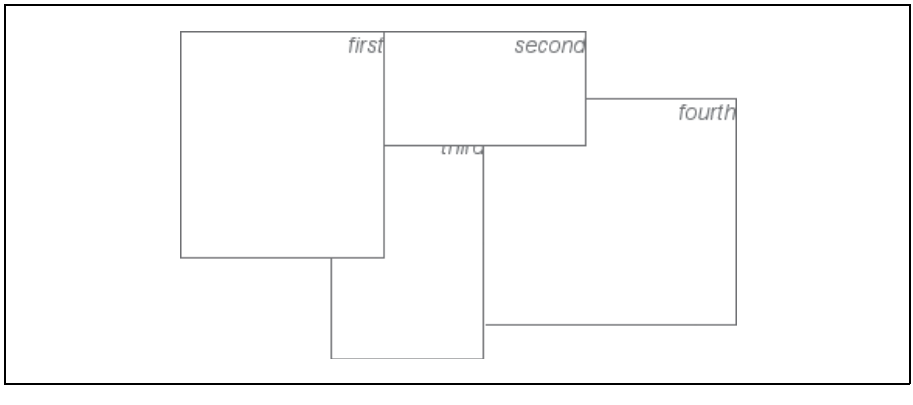


Рис. 10.54. *Помещенные в стек элементы могут перекрывать друг друга*

бы порядок от меньшего к большему: `p#first` (первый), `p#second` (второй), `p#third` (третий), `p#fourth` (четвертый). При этом `p#first` был бы помещен позади остальных трех элементов, а `p#fourth` – впереди.

Как показывает предыдущий пример, вовсе не обязательно, чтобы значения `z-index` были последовательными. Можно задать любое целое любой величины. Чтобы гарантированно оставить элемент поверх всего остального, можно применить, например, такое правило: `z-index: 100000`. В большинстве случаев это обеспечивает ожидаемое поведение, но если объявить `z-index` другого элемента равным `100001` (или выше), то этот другой окажется впереди.

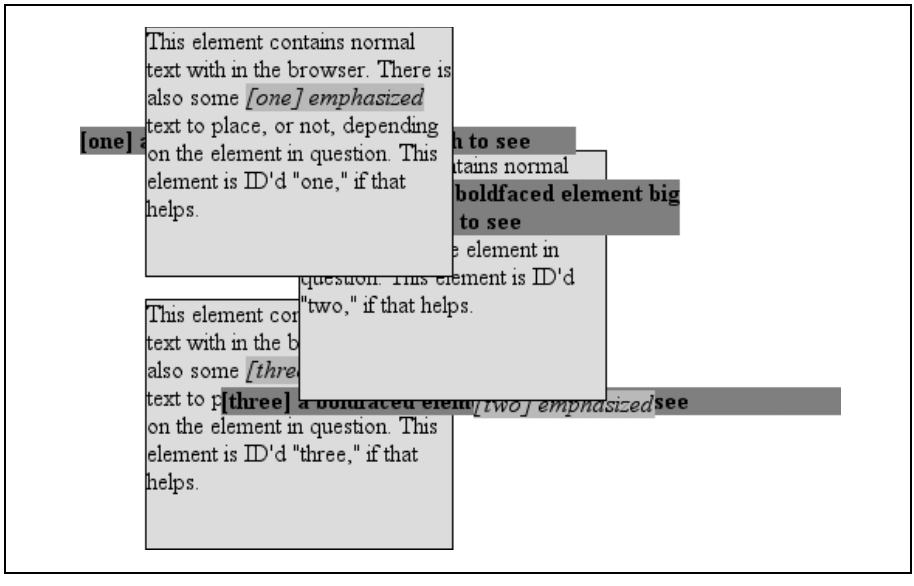


Рис. 10.55. *Позиционированные элементы образуют локальные контексты занесения в стек*

Элемент, свойству `z-index` которого присвоено значение (отличное от `auto`), создает собственный локальный контекст *занесения в стек* (*stacking context*). Следовательно, все потомки элемента имеют собственный порядок занесения в стек относительно элемента-предка. Очень похожим образом элементы формируют новые блоки-контейнеры. Применение следующих стилей даст результат, аналогичный представленному на рис. 10.55:

```
p {border: 1px solid; background: #DDD; margin: 0;}
b {background: #808080;}
em {background: #BBB;}
#one {position: absolute; top: 0; left: 0; width: 50%; height: 10em;
  z-index: 10;}
#two {position: absolute; top: 5em; left: 25%; width: 50%; height: 10em;
  z-index: 7;}
#three {position: absolute; top: 11em; left: 0; width: 50%; height: 10em;
  z-index: 1;}
#one b {position: absolute; right: -5em; top: 4em; width: 20em;
  z-index: -404;}
#two b {position: absolute; right: -3em; top: auto;
  z-index: 36;}
#two em {position: absolute; bottom: -0.75em; left: 7em; right: -2em;
  z-index: -42;}
#three b {position: absolute; left: 3em; top: 3.5em; width: 25em;
  z-index: 23;}
```

Заметьте, куда в порядке занесения в стек попадают элементы `b` и `em`. Каждый из них позиционирован правильно, относительно своего родительского элемента, конечно. Однако внимательно рассмотрите потомков элемента `p#two`. Элемент `b` расположен поверх своего родителя, `em` – сзади, и оба они расположились поверх `p#three`! Причина в том, что значения `36` и `-42` свойства `z-index` рассматриваются относительно `p#two`, а не документа в целом. В некотором смысле `p#two` и все его дочерние элементы имеют одинаковый `z-index`, равный `7`, но при этом у каждого из них есть еще и собственный мини-`z-index` в рамках контекста `p#two`.

Иначе говоря, это как будто `z-index` элемента `b` равен `7,36`, а значение для `em` – `7,-42`. Все это просто предполагаемые абстрактные значения; они никак не связаны со спецификацией. Однако такая система помогает проиллюстрировать, как устанавливается общий порядок занесения в стек. Рассмотрим:

```
p#one          10
p#one b        10,-404
p#two b        7,36
p#two          7
p#two em       7,-42
p#three b      1,23
p#three        1
```

Эта концептуальная структура совершенно точно описывает порядок, в котором эти элементы будут помещены в стек. Потомки элемента могут располагаться в стеке над или под ним, и все они группируются вместе со своим предком.

Кроме того, элемент, образующий контекст занесения в стек для своих потомков, размещается в нулевом положении относительно оси *z* этого контекста. Следовательно, приведенную выше структуру можно расширить так:

```
p#one          10,0
p#one b        10,-404
p#two b        7,36
p#two          7,0
p#two em       7,-42
p#three b      1,23
p#three        1,0
```

Осталось рассмотреть еще одно значение. Вот что говорит спецификация о применяемом по умолчанию значении `auto`:

> Уровень занесения в стек генерируемого блока в текущем контексте занесения в стек аналогичен уровню его родительского блока. Блок не образует нового локального контекста занесения в стек. (CSS2.1: 9.9.1)

Таким образом, любой элемент со значением `z-index: auto` может быть интерпретирован, как если ему задано значение `z-index: 0`. Теперь, однако, вас, наверное, интересует, что происходит с элементами, имеющими

отрицательное значение z-index и являющимися частью контекста занесения в стек начального блока-контейнера. Например, спросите себя, что должно произойти в результате применения следующей разметки:

```
<body>
 <p style="position: absolute; z-index: -1;">Where am I?</p>
</body>
```

Учитывая правила занесения в стек, элемент body должен находиться в том же контексте занесения в стек, что и блок его родителя, поэтому он принимается за 0. Он не образует новый контекст занесения в стек, поэтому абсолютно позиционированный элемент p размещается в том же контексте занесения в стек, что и элемент body (т. е. в начальном блоке-контейнере). Иначе говоря, абзац размещается *позади* элемента body. Если body имеет непрозрачный фон, абзац исчезнет.

Таким, во всяком случае, был возможный результат в CSS2. В CSS2.1 правила занесения в стек изменились, поэтому элемент не может оказаться под фоном его контекста занесения в стек. Другими словами, рассматривается случай, когда элемент body образует блок-контейнер для своих потомков (если бы он был относительно позиционированным, например). Абсолютно позиционированный элемент, происходящий от элемента body, не может быть помещен под фоном body, хотя может располагаться под его содержимым.

На момент написания данной книги Mozilla и родственные броузеры полностью скрывают абзац, даже если и для элемента body, и для элемента html задан прозрачный фон. Это ошибка. Другие агенты пользователя, такие как Internet Explorer, размещают абзац поверх фона body, даже если он у него есть. Согласно CSS2.1 это поведение правильное. Вывод таков, что отрицательные значения z-index могут привести к непредсказуемым результатам, поэтому задавать их надо осторожно.

Фиксированное позиционирование

Как следует из предыдущего раздела, фиксированное позиционирование – это практически то же, что и абсолютное позиционирование, только блоком-контейнером фиксированного элемента является окно просмотра. В этом случае элемент полностью удаляется из потока документа, и его положение никак не связано ни с одной частью документа.

Есть ряд интересных вариантов применения фиксированного позиционирования. Прежде всего, с его помощью можно создать интерфейсы, организованные в виде фреймов. Рассмотрим рис. 10.56, на котором показана довольно распространенная схема компоновки.

Это можно было бы сделать, применив следующие стили:

```
div#header {position: fixed; top: 0; bottom: 80%; left: 20%; right: 0;
  background: gray;}
div#sidebar {position: fixed; top: 0; bottom: 0; left: 0; right: 80%;
  background: silver;}
```

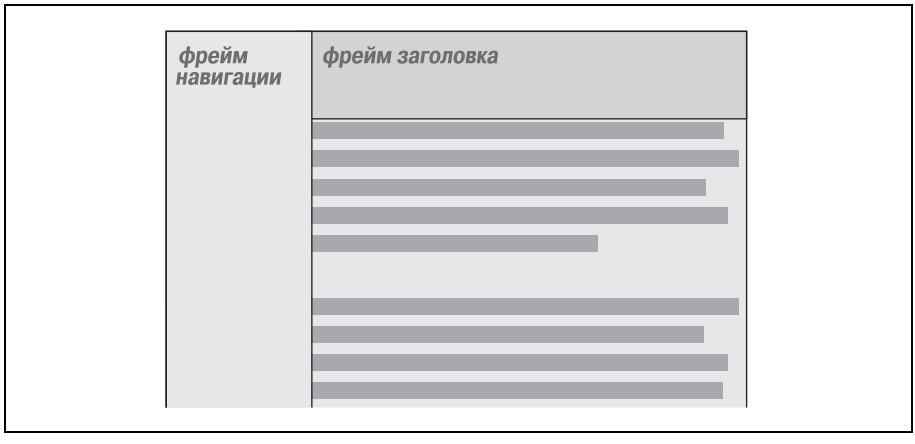


***Рис. 10.56.** Имитация фреймов с помощью фиксированного позиционирования*

Заголовок и врезка будут зафиксированы сверху и сбоку окна просмотра, где они будут оставаться независимо от прокручивания документа. Недостаток здесь в том, что документ будет перекрываться фиксированными элементами. Поэтому остальное содержимое, вероятно, должно находиться в собственном `div` и характеризоваться таким стилем:

```
div#main {position: absolute; top: 20%; bottom: 0; left: 20%; right: 0;
  overflow: scroll; background: white;}
```

Также с помощью добавления соответствующих полей можно было бы создать небольшие зазоры между тремя позиционированными элементами `div`, что продемонстрировано на рис. 10.57:

```
body {background: black; color: silver;}  /* colors for safety's sake */
div#header {position: fixed; top: 0; bottom: 80%; left: 20%; right: 0;
  background: gray; margin-bottom: 2px; color: yellow;}
div#sidebar {position: fixed; top: 0; bottom: 0; left: 0; right: 80%;
  background: silver; margin-right: 2px; color: maroon;}
div#main {position: absolute; top: 20%; bottom: 0; left: 20%; right: 0;
  overflow: auto; background: white; color: black;}
```

Исходя из этого в качестве фона `body` можно было бы применить мозаичное изображение. Это изображение просматривалось бы через зазоры, созданные полями, которые, конечно, можно было бы расширить, если бы автор счел это уместным.

Другое применение фиксированного позиционирования – размещение на экране «постоянных» элементов, таких как краткий перечень ссылок. Постоянный нижний колонтитул с данными об авторском праве и другой информацией можно было бы создать так:

```
div#footer {position: fixed; bottom: 0; width: 100%; height: auto;}
```

***Рис. 10.57**. Разделение «фреймов» с помощью полей*

В результате нижний колонтитул разместится внизу окна просмотра и будет оставаться там, несмотря на прокрутку документа.

Один из недостатков фиксированного позиционирования – оно не поддерживается Internet Explorer для Windows до версии IE7. Существуют обходные приемы, основанные на применении JavaScript, позволяющие в некоторой степени реализовать эту поддержку в более старых версиях IE/Win, но они не всегда приемлемы, потому что в этом случае результат намного менее привлекателен, чем при полной поддержке фиксированного позиционирования. Другая возможность – в IE/Win позиционировать элемент абсолютно и применять фиксированное позиционирование в более совершенных броузерах, хотя такой вариант подходит не для всех макетов.

Об имитации фиксированного позиционирования в более старых версиях IE/Win можно также прочитать на странице *http://css-discuss.incutio.com/?page=EmulatingFixedPositioning*.

Относительное позиционирование

Самая простая для понимания схема позиционирования – относительное позиционирование. В этом случае позиционированный элемент сдвигается за счет применения свойств смещения. Однако это может иметь некоторые интересные последствия.

На первый взгляд все кажется довольно простым. Предположим, вы хотите сдвинуть изображение вверх и влево. На рис. 10.58 показан результат применения следующих стилей:

```
img {position: relative; top: -20px; left: -20px;}
```

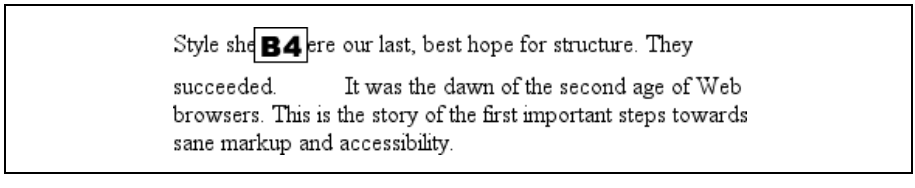


***Рис. 10.58.** Относительно позиционированный элемент*

Здесь всего лишь смещен верхний край изображения на 20 пикселов вверх и левый край на 20 пикселов влево. Однако обратите внимание на пустое место там, где должно было бы находиться изображение, если бы не было позиционировано. Дело в том, что когда элемент позиционирован относительно, он сдвигается со своего обычного места, но пространство, которое он должен был бы занимать, не исчезает. Результат применения следующих стилей показан на рис. 10.59:

```
em {position: relative; top: 8em; color: gray;}
```

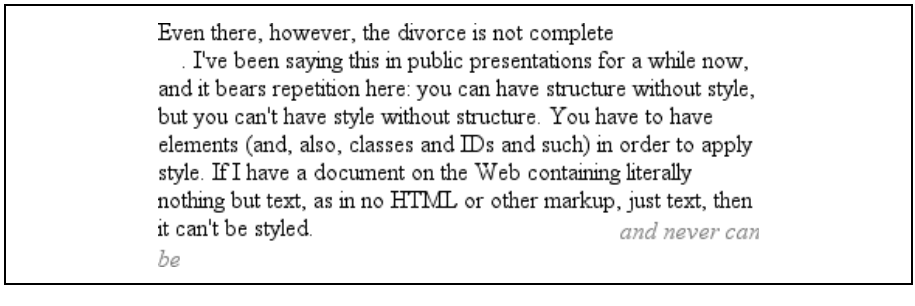


***Рис. 10.59.** Относительно позиционированный элемент*

Как видите, в абзаце есть пропуск. Это то место, где должен был бы находиться элемент `em`, и пространство, занимаемое элементом `em` в его новом местоположении, точно соответствует области, им оставленной.

Конечно, относительно позиционированный элемент можно передвинуть так, что он будет перекрывать другое содержимое. Например, действие следующих стилей и разметки проиллюстрировано на рис. 10.60:

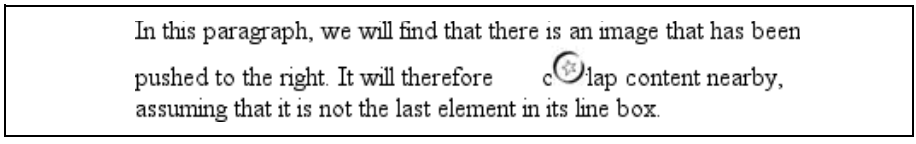


Рис. 10.60. *Элементы, позиционированные относительно, могут перекрывать другое содержимое*

```
img.slide {position: relative; left: 30px;}

<p>
In this paragraph, we will find that there is an image that has been pushed
to the right. It will therefore <img src="star.gif" alt="A star!" class="
slide"> overlap content nearby, assuming that it is not the last element in
its line box.
</p>
```

В предыдущих разделах мы видели, что элемент, позиционируемый относительно, сразу же образует новый блок-контейнер для своих дочерних элементов, соответствующий местоположению этих элементов.

С относительным позиционированием связана одна интересная деталь. Что происходит, когда относительно позиционированный элемент сверхограничен? Например:

```
strong {position: relative; top: 10px; bottom: 20px;}
```

Здесь мы имеем значения, предусматривающие две совершенно разные схемы поведения. Если учитывать только `top: 10px`, то элемент следует сместить вниз на 10 пикселов, но значение `bottom: 20px` четко требует сместить элемент на 20 пикселов вверх.

Исходная спецификация CSS2 не говорит, что должно произойти в этом случае. CSS2.1 утверждает, что когда речь идет о сверхограниченном относительном позиционировании, одному из значений присваивается значение, противоположное другому. Таким образом, `bottom` всегда эквивалентно `-top`. Это означает, что предыдущий пример был бы интерпретирован так, как если бы имел вид:

```
strong {position: relative; top: 10px; bottom: -10px;}
```

Таким образом, элемент `strong` будет смещен вниз на 10 пикселов. Спецификация также учитывает направление написания. При относительном позиционировании `right` всегда эквивалентно `-left` для языков с написанием слева направо, но в языках с написанием справа налево все наоборот: `left` всегда эквивалентно `-right`.

Заключение

Свободное перемещение и позиционирование – очень привлекательные возможности CSS, но при небрежном использовании они могут

стать причиной большого разочарования. Наложение элементов, порядок укладки и размещение – все это должно быть тщательно продумано при позиционировании элементов. Кроме того, необходимо учитывать расположение перемещаемых элементов относительно нормального потока. Таким образом, создание макетов с применением перемещаемых элементов и позиционирования может потребовать некоторой отладки, но получаемые преимущества этого стоят.

Все эти средства и в самом деле позволяют практически избавиться от таблиц в макете, но область их применения во Всемирной паутине не исчезла, например, кроме всего прочего, в таблицах удобно представлять биржевые котировки или результаты спортивных состязаний. В следующей главе мы рассмотрим, как вырос CSS в том, что касается верстки таблиц.

11

Верстка таблиц

Взглянув на название этой главы, вы, наверное, удивились: «Верстка таблиц? Разве не этого мы пытаемся избежать?» Да, пытаемся, но эта глава не о применении таблиц *для* верстки. Она о том, как в CSS планируются сами таблицы, и эта задача намного сложнее, чем может показаться на первый взгляд. Вот почему данному вопросу посвящена целая глава.

Таблицы, по сравнению с остальным макетом документа, уникальны. В CSS2.1 только таблицы обладают способностью увязывать размеры элементов с другими элементами. Например, все ячейки строки имеют одинаковую высоту независимо от того, какое количество содержимого находится в каждой отдельной ячейке. То же справедливо и для ширины ячеек одного столбца. В верстке больше нет других таких примеров, когда элементы различных частей дерева документа оказывают влияние на размеры и компоновку друг друга столь непосредственно.

Как мы увидим, эта уникальность достигнута ценой огромного количества применяемых к таблицам, и только к таблицам, схем поведения и правил. Мы рассмотрим, как происходит визуальная компоновка таблиц, два разных способа отрисовки рамок ячеек и механизмы, управляющие высотой и шириной таблиц и их внутренних элементов.

Форматирование таблиц

Прежде чем озаботиться отрисовкой рамок ячеек и заданием размеров таблиц, надо тщательно изучить фундаментальные методы их верстки и взаимосвязи элементов таблиц друг с другом. Это называется *форматированием таблиц* (*table formatting*) и означает совсем не то же самое, что верстка таблиц: второе возможно только после завершения первого.

Визуальная организация таблицы

Первое, что надо понять, – как CSS определяет организацию таблиц. Хотя эти сведения могут показаться элементарными, они имеют ключевое значение для овладения мастерством стилевого оформления таблиц.

CSS различает элементы таблиц и внутренние элементы таблиц. В CSS внутренние элементы таблиц генерируют прямоугольные блоки, имеющие содержимое, отступы и рамки, но не имеющие полей. Следовательно, *нельзя* определить расстояния между ячейками, задавая поля. Совместимые с CSS броузеры игнорируют любые попытки применения полей к ячейкам, строкам или любому другому элементу таблицы (за исключением заголовков, которые обсуждаются в этой главе позже).

Существует шесть правил организации таблиц. Базовое понятие этих правил – «ячейка сетки», одиночный участок между линиями сетки, на которых отрисовывается таблица. Рассмотрим рис. 11.1, на котором представлены две таблицы с обозначенными ячейками сетки, которые очерчены пунктирными линиями, нанесенными поверх таблиц.

В простой таблице из двух столбцов и двух строк, показанной на рис. 11.1 слева, ячейки сетки соответствуют ячейкам таблицы. В более сложной таблице (рис. 11.1 справа) контуры ячеек сетки соответствуют рамкам ячеек таблицы, но пересекают те ячейки, которые охватывают несколько строк или столбцов.

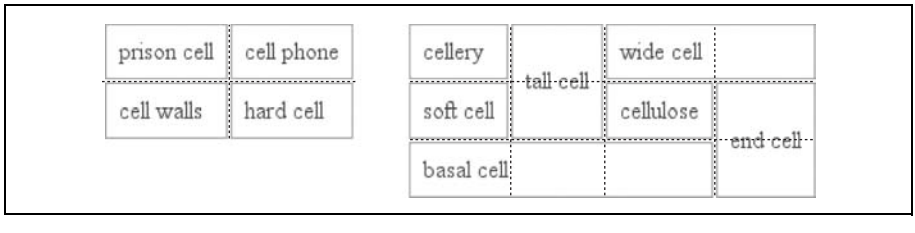


Рис. 11.1. *Ячейки сетки образуют основу компоновки таблицы*

Эти ячейки сетки представляют собой конструкции в значительной мере теоретические, и их нельзя оформить или организовать к ним доступ посредством объектной модели документа. Они служат лишь средством описания способов построения таблицы для организации стилевого оформления.

Правила организации таблиц

- Каждый блок строки включает одну строку ячеек сетки. Все блоки строк таблицы заполняют ее сверху вниз в порядке их размещения в исходном документе (за исключением блоков строк заголовка или нижнего заголовка, которые располагаются в начале и в конце таблицы соответственно). Таким образом, таблица содержит столько строк, сколько в ней есть элементов строк.

- Блок группы строк занимает те же ячейки сетки, что и составляющие его блоки строк.
- Блок столбца включает один или несколько столбцов ячеек сетки. Все блоки столбцов располагаются друг за другом в указанном порядке. Первый блок столбца располагается слева для языков с написанием слева направо и справа для языков с написанием справа налево.
- Блок группы столбцов занимает те же ячейки сетки, что и составляющие его блоки столбцов.
- Ячейки могут объединять несколько строк или столбцов, но в CSS не определено, как это происходит. За этот процесс отвечает язык документа. Каждая объединенная ячейка – это прямоугольный блок шириной и высотой в одну или более ячеек сетки. Верхняя строка этого прямоугольника находится в родительской строке ячейки. Прямоугольник ячейки должен быть расположен как можно левее в языках с написанием слева направо, но не может перекрывать другие блоки ячеек. Также в языках с написанием слева направо он должен располагаться правее всех ячеек, находящихся в той же строке и встречающихся в исходном документе раньше. В языках с написанием справа налево объединенная ячейка должна располагаться как можно правее без перекрытия других ячеек и должна быть левее всех ячеек этой строки, встречающихся в исходном документе до нее.
- Блок ячейки не может выходить за границы блока последней строки таблицы или группы строк. Если структура таблицы обусловливает это, ячейка должна быть сокращена так, чтобы помещаться в таблицу или группу строк, в которую она входит.

Спецификация CSS не одобряет, но и не запрещает позиционирование ячеек таблиц и других внутренних элементов таблиц. Позиционирование строки, содержащей объединенные строки ячейки, например, могло бы сильно изменить макет таблицы за счет полного удаления строк из нее и таким образом удаления объединенных ячеек из макета других строк.

По определению ячейки сетки прямоугольные, но не все они должны иметь один размер. Ширина всех ячеек сетки данного столбца будет одинаковой, и высота всех ячеек сетки строки будет одинаковой, но высота одной строки сетки может отличаться от высоты другой ее строки. Ширина столбцов сетки тоже может быть разной.

После ознакомления со всеми этими основными правилами может возникнуть вопрос: откуда именно известно, какие элементы являются ячейками, а какие – нет? Мы выясним это в следующем разделе.

Значения свойства display для таблиц

В HTML нетрудно понять, какие элементы являются частями таблицы, поскольку в броузеры встроена обработка таких элементов, как `tr` и `td`. В XML, напротив, по сути невозможно знать, какие элементы мо-

display	
Значения:	none | inline | block | inline-block | list-item | run-in | table | inline-table | table-row-group | table-header-group | table-footer-group | table-row | table-column-group | table-column | table-cell | table-caption | inherit
Начальное значение:	inline
Область применения:	все элементы
Наследование:	нет
Вычисляемое значение:	меняется для перемещаемых, позиционированных и корневых элементов (см. CSS2.1, раздел 9.7); в противном случае – как задано
Примечание:	значения compact и marker появились в CSS2, но были изъяты из CSS2.1 из-за недостаточно широкой поддержки

гут быть частью таблицы. Вот где в игру вступает полный набор значений свойства display.

В этой главе мы коснемся связанных с таблицами значений, поскольку остальные (block, inline, inline-block, run-in и list-item) обсуждаются в других главах. Касающиеся таблиц значения можно охарактеризовать следующим образом:

table

Это значение показывает, что элемент определяет таблицу уровня блока. Таким образом, оно определяет прямоугольный блок, который обрабатывается как блочный элемент. Соответствующий HTML-элемент, что не удивительно, – table.

inline-table

Это значение показывает, что элемент определяет таблицу строкового уровня. Следовательно, элемент определяет прямоугольный блок, который генерирует строковый блок. Ближайший аналог – значение inline-block. Ближайший HTML-элемент – table, хотя по умолчанию HTML-таблицы не являются строковыми элементами.

table-row

Это значение определяет, что элемент является строкой ячеек. Соответствующий HTML-элемент – tr.

table-row-group

Это значение определяет, что элемент объединяет одну или несколько строк. Соответствующее HTML-значение – tbody.

table-header-group

Это значение очень похоже на table-row-group, за исключением того, что при визуальном форматировании группа заголовков табли-

цы всегда отображается перед всеми остальными строками и группами строк и после всех основных заголовков. При печати агент пользователя может повторять строки заголовков в начале каждой страницы, если для распечатки таблицы необходимо несколько страниц. Спецификация не определяет, что происходит, если `table-header-group` задано для нескольких элементов. Группа заголовков может включать несколько строк. Эквивалент HTML – `thead`.

`table-footer-group`

Это значение во многом похоже на `table-header-group`, за исключением того, что группа строк нижнего заголовка всегда отображается после всех остальных строк и групп строк и перед любыми нижними основными заголовками. При печати агент пользователя может повторять строки нижних заголовков внизу каждой страницы, если для распечатки таблицы требуется несколько страниц. Спецификация не определяет, что происходит, если задать `table-footer-group` для нескольких элементов. Аналогичен HTML-элементу `tfoot`.

`table-column`

Означает, что элемент описывает столбец ячеек. С точки зрения CSS для элемента с таким значением свойства `display` не генерируется визуальное представление, как будто он имеет значение `none`. Они вводятся в основном во вспомогательных целях, чтобы определить представление ячеек в столбце. Аналог HTML-элемента `col`.

`table-column-group`

Означает, что элемент объединяет один или несколько столбцов. Как и для элементов `table-column`, для элементов `table-column-group` визуальное представление не генерируется, но это значение полезно для определения представления элементов группы столбцов. Аналогичен HTML-элементу `colgroup`.

`table-cell`

Это значение определяет, что элемент представляет отдельную ячейку таблицы. В качестве примеров элементов `table-cell` можно назвать элементы HTML `th` и `td`.

`table-caption`

Это значение определяет основной заголовок таблицы. CSS не оговаривает, что должно произойти, если значение `caption` имеют несколько элементов, но предупреждает: «...авторы не должны помещать более одного элемента со значением `display: caption` в таблицу или элемент таблицы строкового уровня».

Краткий обзор основных эффектов от применения этих значений можно найти в приложении C, рассмотрев приведенный там фрагмент примера таблицы стилей HTML 4.0.

```
table          {display: table;}
tr             {display: table-row;}
```

```
thead           {display: table-header-group;}
tbody           {display: table-row-group;}
tfoot           {display: table-footer-group;}
col             {display: table-column;}
colgroup        {display: table-column-group;}
td, th          {display: table-cell;}
caption         {display: table-caption;}
```

В XML, где элементы не имеют семантики отображения по умолчанию, эти значения становятся крайне полезными. Рассмотрим следующую разметку:

```
<scores>
  <headers>
    <label>Команда</label>
    <label>Счет</label>
  </headers>
  <game sport="MLB" league="NL">
    <team>
      <name>Красные</name>
      <score>8</score>
    </team>
    <team>
      <name>Кабс</name>
      <score>5</score>
    </team>
  </game>
</scores>
```

Эту разметку можно отформатировать в виде таблицы с помощью следующих стилей:

```
scores {display: table;}
headers {display: table-header-group;}
game {display: table-row-group;}
team {display: table-row;}
label, name, score {display: table-cell;}
```

Затем различные ячейки можно было бы отформатировать, например, выделить полужирным шрифтом элементы `label` и выровнять по правому краю `scores`.

Теоретически можно присвоить табличные значения свойства `display` любому HTML-элементу, но Internet Explorer вплоть до версии IE7 не поддерживает эту возможность.

Первичность строк

CSS определяет свою модель таблицы как модель «первичности строк». Иначе говоря, модель предполагает, что авторы будут создавать языки разметки, в которых строки объявляются прямо. В то же

время столбцы представляют собой производные строк ячеек. Таким образом, первый столбец образуется первыми ячейками всех строк, второй столбец – вторыми ячейками и т. д.

Первичность строк не порождает заметных трудностей в HTML, поскольку этот язык разметки также строкоориентированный. В XML же это обстоятельство более неприятно, поскольку оставляет авторам меньше возможностей для определения таблиц. Из-за того, что модель таблицы в CSS строкоориентированна по природе, язык разметки, в котором столбцы составляют основу макета таблицы, невозможен (при условии, что представление такого документа формируется на основе CSS).

Первичность строк модели CSS будет находиться в поле нашего зрения на всем протяжении этой главы, пока мы будем изучать представление таблиц.

Столбцы

Хотя модель таблиц CSS ориентирована на строки, столбцы все-таки играют важную роль в макете. Ячейка может принадлежать обоим контекстам (строки и столбца), даже если в исходном файле документа они происходят от элементов строк. В CSS столбцы и группы столбцов могут принимать только четыре свойства стиля: `border`, `background`, `width` и `visibility`.

Кроме того, каждое из этих четырех свойств имеет специальные правила, применяемые только в контексте столбца:

`border`

Рамки могут быть заданы для столбцов и групп столбцов, только если свойство `border-collapse` имеет значение `collapse`. В этом случае рамки столбцов и групп столбцов участвуют в алгоритме слияния, который определяет стили рамок для всех сторон ячейки (см. раздел «Сливающиеся рамки ячеек» в этой главе).

`background`

Фон столбца или группы столбцов будет видимым только в ячейках, в которых и ячейка, и ее строка имеют прозрачный фон (см. раздел «Слои таблицы» далее в этой главе).

`width`

Свойство `width` определяет минимальную ширину столбца или группы столбцов. Содержимое ячеек столбца (или группы) может привести к увеличению его ширины.

`visibility`

Если свойство `visibility` столбца или группы столбцов имеет значение `collapse`, визуальное представление ни одной из ячеек столбца (или группы) не генерируется. Ячейки, которые распространяются на отсекаемый столбец и другие столбцы, отсекаются, как и ячейки, распространяющиеся на скрытый столбец. Более того, общая

ширина таблицы уменьшается на ширину этого столбца. Объявление любого отличного от `collapse` значения свойства `visibility` для столбца или группы столбцов игнорируется.

Анонимные объекты таблицы

Может случиться, что в языке разметки реализованы не все элементы, которые необходимы для полного соответствия представления таблиц их описанию в CSS, или автор забудет включить все необходимые элементы. Рассмотрим следующий HTML:

```
<table>
  <td>Name:</td>
  <td><input type="text"></td>
</table>
```

Вероятно, взглянув на эту разметку, вы подумали, что она описывает таблицу, состоящую из двух ячеек, размещенных в одной строке, но структурно здесь нет элемента, определяющего строку (потому что `tr` пропущен).

Предвосхищая такую возможность, CSS определяет механизм введения «пропущенных» компонентов таблицы как анонимных объектов. Для иллюстрации вернемся к нашему XHTML с пропущенной строкой. С точки зрения CSS на самом деле здесь между элементом `table` и происходящими от него ячейками таблицы вставляется анонимный объект `table-row`:

```
<table>
 [начинается анонимный объект table-row]
  <td>Name:</td>
  <td><input type="text"></td>
 [заканчивается анонимный объект table-row]
</table>
```

Этот процесс изображен на рис. 11.2.

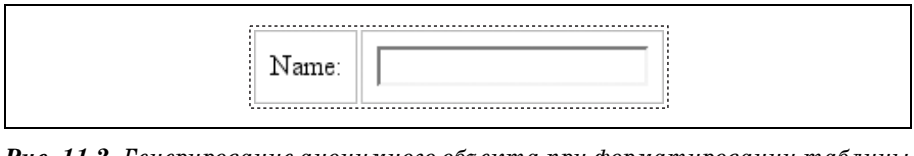


Рис. 11.2. *Генерирование анонимного объекта при форматировании таблицы*

В модели таблиц CSS определяется семь различных случаев введения анонимных объектов. Как наследование и специфичность, они являют собой пример механизма, который пытается сделать схему поведения CSS интуитивно понятной.

Правила введения объектов

1. Если родителем элемента `table-cell` не является элемент `table-row`, то между элементом `table-cell` и его родителем вставляется ано-

нимный объект table-row. Введенный объект будет охватывать все последующие сестринские элементы элемента table-cell. Рассмотрим следующие стили и разметку:

```
system {display: table;}
name, moons {display: table-cell;}
```

```
<system>
  <name>Mercury</name>
  <moons>0</moons>
</system>
```

Анонимный объект table-row вставляется между элементами ячеек и элементом system и охватывает и элемент name, и элемент moons.

То же правило действует, даже если родительским элементом является table-row-group. Чтобы расширить пример, добавим следующее:

```
system {display: table;}
planet {display: table-row-group;}
name, moons {display: table-cell;}
```

```
<system>
 <planet>
  <name>Mercury</name>
  <moons>0</moons>
 </planet>
 <planet>
  <name>Venus</name>
  <moons>0</moons>
 </planet>
</system>
```

В этом примере оба набора ячеек будут заключены в анонимный объект table-row, который вставляется между ними и элементами planet.

2. Если родителем элемента table-row не являются table, inline-table или table-row-group, то между элементом table-row и его родителем вставляется анонимный элемент table. Вставленный объект будет охватывать все последующие сестринские элементы элемента table-row. Рассмотрим следующие стили и разметку:

```
docbody {display: block;}
planet {display: table-row;}
```

```
<docbody>

 <planet>
  <name>Mercury</name>
  <moons>0</moons>
 </planet>
  <planet>
  <name>Venus</name>
  <moons>0</moons>
 </planet>

</docbody>
```

Поскольку значение свойства display родителя элемента planet — block, анонимный объект table вставляется между элементами planet и элементом docbody. Этот объект будет охватывать оба элемента planet, потому что они являются последовательными сущностями.

3. Если родителем элемента table-column не является элемент table, inline-table или table-column-group, то между элементом table-column и его родителем вставляется анонимный элемент table. Это правило в основном аналогично только что рассмотренному с поправкой на его ориентированность на столбец.
4. Если родителем элемента table-row-group, table-header-group, table-footer-group, table-column-group или table-caption не является элемент table, то между элементом и его родителем вставляется анонимный элемент table.
5. Если дочерним элементом элемента table или inline-table не является элемент table-row-group, table-header-group, table-footer-group, table-row или table-caption, то анонимный элемент table-row вставляется между элементом table и его дочерним элементом. Этот анонимный объект охватывает все последующие сестринские элементы дочернего элемента, которые не являются элементами table-row-group, table-header-group, table-footer-group, table-row или table-caption. Рассмотрим следующие стили и разметку:

```
system {display: table;}
planet {display: table-row;}
name, moons {display: table-cell;}
```

```
<system>
 <planet>
  <name>Mercury</name>
  <moons>0</moons>
 </planet>
  <name>Venus</name>
  <moons>0</moons>
</system>
```

Здесь между элементом system и вторым набором элементов name и moons будет вставляться анонимный объект table-row. Элемент planet не охватывается анонимным объектом, потому что значением его свойства display является table-row.

6. Если дочерним элементом элементов table-row-group, table-header-group или table-footer-group не является элемент table-row, то анонимный объект table-row вставляется между элементом и его дочерним элементом. Этот анонимный объект охватывает все последующие сестринские элементы дочернего элемента, которые сами по себе не являются объектами table-row. Рассмотрим следующие стили и разметку:

```
system {display: table;}
planet {display: table-row-group;}
```

```
name, moons {display: table-cell;}
```

```
<system>
 <planet>
  <name>Mercury</name>
  <moons>0</moons>
 </planet>
  <name>Venus</name>
  <moons>0</moons>
</system>
```

В этом случае каждый набор элементов `name` и `moons` будет включен в анонимный элемент `table-row`. Для второго набора анонимный объект вставляется между элементом `planet` и его дочерним элементом, потому что элемент `planet` является элементом `table-row-group`.

7. Если дочерним элементом элемента `table-row` не является элемент `table-cell`, то анонимный объект `table-cell` вставляется между элементом и его потомком. Этот анонимный объект охватывает все последующие сестринские элементы дочернего элемента, которые сами по себе не являются элементами `table-cell`. Рассмотрим следующие разметку и стили:

```
system {display: table;}
planet {display: table-row;}
name, moons {display: table-cell;}
```

```
<system>
 <planet>
  <name>Mercury</name>
  <num>0</num>
 </planet>
</system>
```

Поскольку элемент `num` не имеет табличного значения свойства `display`, анонимный объект `table-cell` вставляется между элементом `planet` и элементом `num`.

Это поведение также распространяется на инкапсуляцию анонимных строковых блоков. Предположим, что теперь не был включен и элемент `num`:

```
<system>
 <planet>
  <name>Mercury</name>
  0
 </planet>
</system>
```

В этом случае в анонимный объект `table-cell` был бы включен `0`. Чтобы продолжить иллюстрацию этого момента, приведу пример, адаптированный из спецификации CSS:

```
example {display: table-cell;}
row {display: table-row;}
```

```
hi {font-weight: 900;}

<example>
  <row>This is the <hi>top</hi> row.</row>
  <row>This is the <hi>bottom</hi> row.</row>
</example>
```

В каждом элементе `row` фрагменты текста и элемент `hi` заключены в анонимный объект `table-cell`.

Слои таблицы

Для сборки представления таблицы CSS определяет шесть отдельных слоев, в которых размещаются различные части таблицы. Эти слои представлены на рис. 11.3.

В сущности, стили каждой части таблицы применяются к отдельным слоям. Таким образом, если элемент `table` имеет зеленый фон и черную рамку шириной один пиксел, эти стили отрисовываются на самом нижнем слое. Любые стили для групп столбцов применяются к следующему слою, сами столбцы располагаются в слое, находящемся над ним, и т. д. Самый верхний слой, соответствующий ячейкам таблицы, отрисовывается последним.

В значительной мере это просто логический процесс; кроме того, если для ячеек таблицы объявляется фоновый цвет, то его, наверное, имеет смысл отрисовывать поверх фона элемента `table`. Самое важное, что следует из рис. 11.3, – стили столбцов располагаются под стилями строк, поэтому фон строки будет перекрывать фон столбца.

Важно помнить, что по умолчанию фон всех элементов прозрачный. Таким образом, в следующей разметке фон элемента `table` будет про-

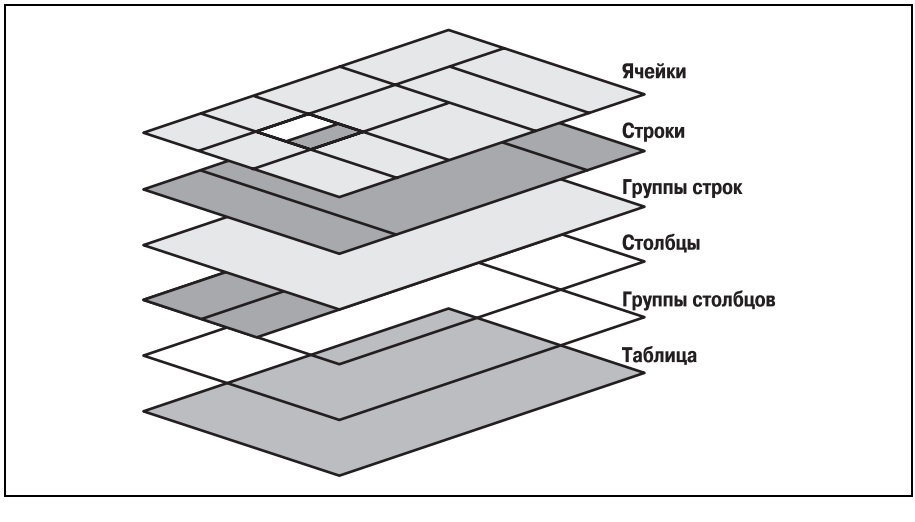


Рис. 11.3. *Слои форматирования, используемые в представлении таблицы*

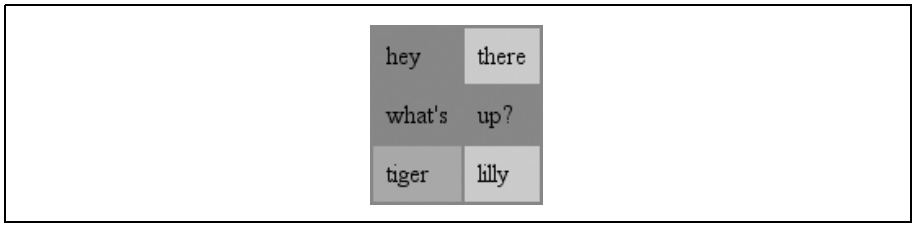


Рис. 11.4. *Фон слоев форматирования таблицы просматривается через другие слои*

сматриваться сквозь ячейки, строки, столбцы и т. д. – все, что не имеет собственного фона, как показано на рис. 11.4:

```
<table style="background: #888;">
 <tr>
  <td>hey</td>
  <td style="background: #CCC;">there</td>
 </tr>
<tr>
  <td>what's</td>
  <td>up?</td>
 </tr>
<tr style="background: #AAA;">
  <td>tiger</td>
  <td style="background: #CCC;">lilly</td>
 </tr>
</table>
```

Основные заголовки

Основной заголовок таблицы представляет собой, как и можно ожидать, короткий фрагмент текста, описывающий суть содержимого таблицы. Следовательно, диаграмма биржевых котировок за четвертый квартал 2003 года могла бы иметь элемент основного заголовка такого содержания «Рост курса акций в 4 квартале 2003 г.». С помощью свойства `caption-side` этот элемент можно поместить или над, или под таблицей независимо от того, где находится заголовок в структуре таблицы. (В HTML элементу `caption` должен предшествовать открывающий элемент `table`, но в других языках могут быть другие правила.)

Основные заголовки немного необычны, по крайней мере с визуальной точки зрения. Спецификация CSS утверждает, что основной заголовок форматируется так, как если бы он являлся блочным элементом, размещенным непосредственно перед (или после) блока таблицы, за некоторыми исключениями. Во-первых, основной заголовок все-таки может наследовать значения свойств таблицы, во-вторых, агент пользователя игнорирует блок основного заголовка, если ему требуется решить, что делать с элементом вставки, предшествующим таблице. Следовательно, элемент вставки, располагающийся перед таблицей,

caption-side

Значения:	top | bottom
Начальное значение:	top
Область применения:	элементы, свойство display которых имеет значение table-caption
Наследование:	нет
Вычисляемое значение:	как задано
Примечание:	значения left и right появились в CSS2, но были изъяты из CSS2.1 из-за недостаточно широкой поддержки

не войдет ни в основной заголовок, ни в таблицу, но будет интерпретироваться так, как если бы его свойство display имело значение block.

Простого примера должно быть достаточно для иллюстрации наиболее важных аспектов представления основного заголовка (рис. 11.5):

```
caption {background: gray; margin: 1em 0;
  caption-side: top;}
table {color: white; background: black; margin: 0.5em 0;}
```

Текст элемента caption наследует значение white свойства color от таблицы, а фон основной заголовок получает собственный. Интервал между внешним краем рамки таблицы и внешним краем поля основного заголовка составляет один em, поскольку верхнее поле таблицы и нижнее поле заголовка свернулись, как описывалось в главе 7. Наконец, ширина заголовка соответствует ширине содержимого элемента table, который считается блоком-контейнером заголовка. То же самое получится, если изменить значение caption-side на bottom, за исключением того, что заголовок расположился бы под блоком таблицы, и сворачивание будет иметь место между верхним полем основного заголовка и нижним полем таблицы.

***Рис. 11.5.** Стилевое оформление основных заголовков и таблиц*

По большей части основные заголовки оформляются точно так же, как и любой блочный элемент; они могут иметь отступы, рамки, фон и т. д. Например, для изменения горизонтального выравнивания текста заголовка применяется свойство `text-align`. Таким образом, чтобы выровнять по правому краю основной заголовок в предыдущем примере, можно написать:

```
caption {background: gray; margin: 1em 0;
  caption-side: top; text-align: right;}
```

И в середине 2006 г. стилевое оформление основных заголовков по-прежнему остается рискованным предприятием. Некоторые броузеры отображают верхнее и нижнее поля заголовков, тогда как некоторые – нет; некоторые броузеры вычисляют ширину основного заголовка относительно ширины таблицы, тогда как остальные используют иной подход. Перечислять здесь все возможные варианты поведения бессмысленно, поскольку эта область стремительно меняется. Данное примечание, главным образом, предупреждает читателей о возможных проблемах.

Рамки ячеек таблицы

На самом деле в CSS определены две совершенно разные модели рамок. Модель отдельных рамок действует, когда ячейки отделены друг от друга с точки зрения компоновки. Другой вариант – модель сливающихся рамок, в которой между ячейками нет визуального разделения и рамки ячеек объединяются, или сливаются, друг с другом. По умолчанию применяется первая модель, хотя в предыдущей версии CSS таковой была вторая модель.

Автор может выбирать модель, задавая свойство `border-collapse`.

Это свойство предоставляет автору возможность определять, какой модели рамки будет следовать агент пользователя. Если применяется значение `collapse`, выбирается модель сливающихся рамок, а если `separate`, то модель отдельных рамок. Сначала обратимся ко второй модели, поскольку описать ее намного проще.

border-collapse

Значения:	`collapse | separate | inherit`
Начальное значение:	`separate`
Область применения:	элементы, свойство `display` которых имеет значения `table` или `inline-table`
Наследование:	да
Вычисляемое значение:	как задано
Примечание:	в CSS2 применяемым по умолчанию было значение `collapse`

Отдельные рамки ячеек

В этой модели каждая ячейка таблицы отстоит от других на некоторое расстояние и рамки ячеек не сливаются друг с другом. Результат применения следующих стилей и разметки показан на рис. 11.6:

```
table {border-collapse: separate;}
td {border: 3px double black; padding: 3px;}
```

```
<table cellspacing="0">
  <tr>
    <td>cell one</td>
    <td>cell two</td>
  </tr>
  <tr>
    <td>cell three</td>
    <td>cell four</td>
  </tr>
</table>
```

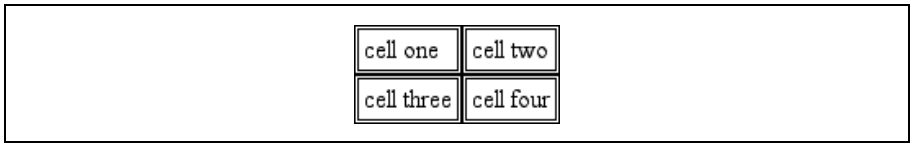


Рис. 11.6. *Разделенные (и таким образом отдельные) рамки ячеек*

Обратите внимание, что рамки ячеек касаются, но не сливаются друг с другом. Три линии между ячейками – это на самом деле две двойные рамки, расположенные впритык.

Атрибут HTML `cellspacing` был включен в приведенный выше пример, чтобы гарантировать отсутствие промежутка между ячейками. Кроме того, если в CSS есть способ разделять эти рамки, то должен быть и способ изменять расстояние между ячейками. К счастью, он есть.

Промежуток между рамками

Иногда требуется, чтобы рамки ячеек таблиц отстояли друг от друга на некоторое расстояние. Это легко осуществляется с помощью свойства `border-spacing`, предоставляющего более мощную замену HTML-атрибута `cellspacing`.

В качестве значения этого свойства может быть задана одна или две длины. Для того чтобы все ячейки были разделены промежутком в один пиксел, достаточно задать `border-spacing: 1px;`. Если же требуется чтобы по горизонтали ячейки отстояли друг от друга на один пиксел, а по вертикали на пять, то надо написать `border-spacing: 1px 5px;`. Если заданы две длины, то первая всегда определяет горизонтальный промежуток, а вторая – вертикальный.

border-spacing	
Значения:	<длина> <длина>? | inherit
Начальное значение:	0
Область применения:	элементы, свойство display которых имеет значение table или inline-table
Наследование:	да
Вычисляемое значение:	две абсолютные длины
Примечание:	свойство игнорируется, если значение свойства border-collapse не separate

Значения промежутков также применяются между рамками ячеек по внешней границе таблицы и отступами самого элемента таблицы. Приведенные ниже стили дают результат, показанный на рис. 11.7:

```
table {border-collapse: separate; border-spacing: 3px 5px;
 padding: 10px; border: 2px solid black;}
td { border: 1px solid gray;}
td#squeeze {border-width: 5px;}
```

На рис. 11.7 расстояние между рамками любых двух соседних по горизонтали ячеек составляет 3 пиксела, а между рамками самой правой и самой левой ячеек и правой и левой рамками элемента table – 13 пикселов. Аналогично по вертикали рамки отстоят друг от друга на 5 пикселов, а рамки ячеек верхней и нижней строк – на 15 пикселов от верхней и нижней рамок таблицы соответственно. Промежуток между рамками ячеек постоянен по всей таблице независимо от ширины рамок самих ячеек.

Также обратите внимание, что border-spacing применяется к таблице в целом, а не к отдельным ячейкам. Если в предыдущем примере border-spacing было задано для элементов td, оно было бы проигнорировано.

В модели отдельных рамок нельзя задать рамки для строк, групп строк, столбцов и групп столбцов. Любые свойства рамок, объявленные для этих элементов, должны игнорироваться совместимыми с CSS агентами пользователя.

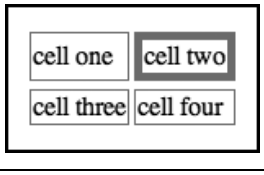


Рис. 11.7. *Разделение рамок происходит между ячейками и таблицей, в которой они находятся*

Обработка пустых ячеек

Поскольку каждая ячейка в визуальном смысле отделена от всех остальных ячеек таблицы, что делать с пустыми ячейками (т. е. не имеющими содержимого)? Тут есть две возможности, отраженные в значениях свойства `empty-cells`.

Если свойству `empty-cells` задано значение `show`, рамки и фон пустой ячейки будут отрисовываться так же, как для ячейки таблицы, имеющей содержимое. Если задано значение `hide`, то ни одна из частей ячейки не отрисовывается – так же, как если бы для ячейки было задано `visibility: hidden`.

Если в ячейке есть какое-либо содержимое, она не может считаться пустой. Понятие «содержимое» в данном случае включает не только текст, изображения, элементы формы и т. д., но и неразрывные пробелы (` `) и любой другой символ-разделитель, *кроме* CR (carriage-return – возврат каретки), LF (linefeed – перевод строки), символов табуляции и пробела. Если все ячейки строки пусты и значение `empty-cells` для всех равно `hide`, то вся строка интерпретируется так, как будто для элемента строки задано `display: none`.

empty-cells

Значения:	`show | hide | inherit`
Начальное значение:	`show`
Область применения:	элементы, свойство `display` которых имеет значение `table-cell`
Наследование:	да
Вычисляемое значение:	как задано
Примечание:	свойство игнорируется, если значение свойства `border-collapse` не `separate`

На момент написания данной книги свойство `empty-cells` не вполне поддерживается в Internet Explorer.

Сливающиеся рамки ячеек

Модель сливающихся ячеек почти полностью соответствует традиционной верстке HTML-таблиц, когда никакого разделения ячеек не было, но она немного сложнее, чем модель отдельных рамок. Здесь также действуют некоторые правила, которые отличают модель сливающихся рамок ячеек от модели отдельных рамок.

- Элементы, свойство `display` которых имеет значения `table` или `inline-table`, не могут иметь отступов, хотя у них могут быть поля. Та-

ким образом, между рамкой вокруг таблицы и ее внешними ячейками никогда не может быть пробела.

- Рамки могут применяться к ячейкам, строкам, группам строк, столбцам и группам столбцов. Сама таблица, как обычно, может иметь рамку.
- Между рамками ячеек не может быть никаких промежутков. Кстати, рамки сливаются, когда примыкают друг к другу, так что в действительности отрисовывается только одна из сливающихся рамок. Это похоже на сворачивание полей, при котором побеждает большее поле. Когда рамки ячейки сливаются, победу одерживает «наиболее интересная» из них.
- При слиянии рамки между ячейками центрируются по гипотетическим линиям сетки.

В следующих двух разделах последние два момента рассмотрены более подробно.

Компоновка со сливающимися рамками

Чтобы лучше понять, как работает модель сливающихся рамок, рассмотрим макет одной строки таблицы, показанной на рис. 11.8.

Для каждой ячейки отступы и содержимое располагаются внутри рамок, как и ожидалось. Что касается рамок между ячейками, то одна

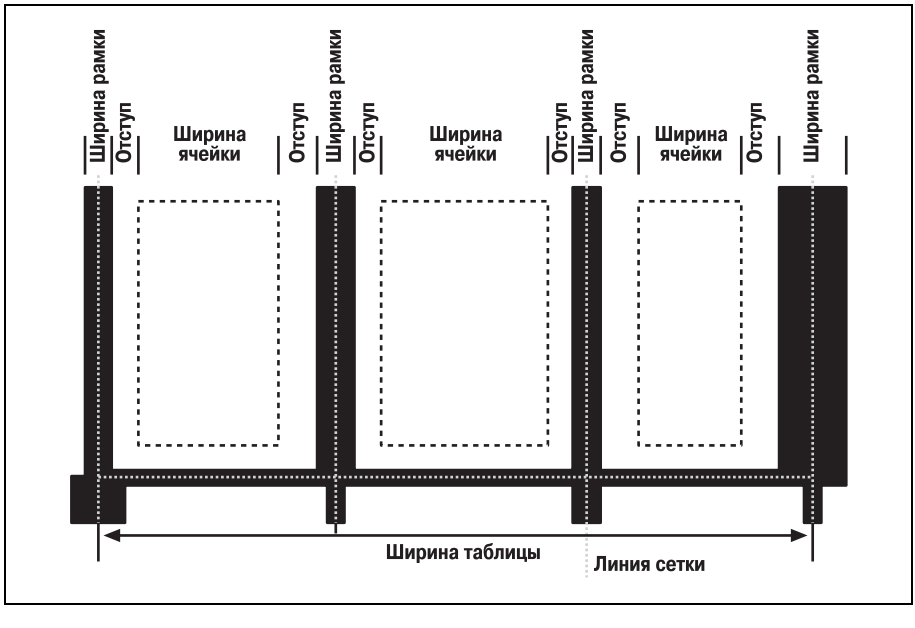


Рис. 11.8. *Макет строки таблицы (модель сливающихся рамок)*

половина рамки располагается с одной стороны линии сетки, разделяющей ячейки, а вторая половина – с другой стороны. В каждом случае вдоль каждого края ячейки отрисовывается только одна рамка. Возможно, вы думаете, что половина рамки каждой ячейки отрисовывается с каждой из сторон линии сетки, но это не так.

Предположим, что сплошные рамки средней ячейки окрашены в зеленый цвет, а сплошные рамки двух внешних ячеек – в красный. Рамки с правой и левой сторон средней ячейки (которые сливаются с соседними рамками внешних ячеек) будут полностью зелеными или полностью красными в зависимости от того, какая из них одерживает верх. Определение приоритетов рамок рассматривается в следующем разделе.

Вы могли заметить, что внешние рамки выступают за границы таблицы. Причина в том, что в этой модели *половина* ширины рамки таблицы включена в ширину таблицы. Другая половина выступает за пределы этого расстояния, занимая поле. Это может показаться немного странным, но именно так определен принцип работы этой модели.

Спецификация включает формулу компоновки, которую я воспроизведу здесь для тех, кому нравятся формулы:

$$\text{ширина строки} = (0.5 * \text{border-width}_0) + \text{padding-left}_1 + \text{width}_1 + \text{padding-right}_1 + \text{border-width}_1 + \text{padding-left}_2 + \ldots + \text{padding-right}_n + (0.5 * \text{border-width}_n)$$

Каждое свойство `border-width` описывает рамку между ячейкой *i* и следующей ячейкой; таким образом, `border-width3` описывает рамку между третьей и четвертой ячейками. Значение `n` определяет общее количество ячеек в строке. Здесь есть небольшое исключение. Начиная компоновку таблицы со сливающимися рамками, агент пользователя вычисляет исходную левую и правую рамки для самой таблицы. Он делает это, проверяя левую рамку первой ячейки первой строки таблицы и принимая половину ширины этой рамки за ширину исходной левой рамки таблицы. Затем агент пользователя проверяет правую рамку последней ячейки первой строки и за ширину исходной правой рамки таблицы принимает половину ее ширины. Для всех последующих строк, если левая или правая их рамка шире исходных рамок, она выступает в область поля таблицы.

В тех случаях, когда рамка содержит в ширину нечетное число элементов отображения (пикселов, точек принтера и т. д.), центрированием рамки по линии сетки управляет агент пользователя. Он может сдвинуть рамку так, что она будет располагаться немного не по центру, округлить ее ширину вверх или вниз до ближайшего четного числа или сделать что-либо еще, что кажется ему подходящим.

Слияние рамок

Когда рядом располагаются две или более рамки, они сливаются. Кстати, этот процесс не происходит случайным образом. Здесь действуют строгие правила, которые определяют, какие рамки одержат верх:

- Если свойство `border-style` одной из рамок имеет значение `hidden`, она получает более высокий приоритет, чем остальные рамки. Все рамки в этом месте будут скрытыми.
- Если свойство `border-style` одной из сливающихся рамок имеет значение `none`, она получает самый низкий приоритет. В этом месте рамки не будут отрисовываться, только если все границы имеют значение `none`. Обратите внимание, что `none` – это применяемое по умолчанию значение свойства `border-style`.
- Если хотя бы одна из сливающихся рамок имеет значение, отличное от `none`, и ни одна из сливающихся рамок не имеет значения `hidden`, более узкие рамки проигрывают более широким. Если несколько сливающихся рамок имеют одинаковую ширину, то стили рамки применяются в порядке от более предпочтительного к менее: `double`, `solid`, `dashed`, `dotted`, `ridge`, `outset`, `groove`, `inset`. Таким образом, если сливаются две рамки одинаковой ширины, из которых одна – `dashed`, а другая – `outset`, то рамка в этом месте будет пунктирной (dashed).
- Если сливающиеся рамки имеют одинаковые стиль и ширину, но отличаются цветом, тогда цвет определяется более приоритетным элементом в соответствии с приведенным в порядке уменьшения приоритета списком: ячейка, строка, группа строк, столбец, группа столбцов, таблица. Таким образом, если сливаются рамки ячейки и столбца (полностью идентичные, за исключением цвета), тогда будет применен цвет рамки ячейки (и стиль, и ширина). Если сливающиеся рамки относятся к одному типу элементов, например две рамки строк имеют одинаковые стиль и ширину, но разные цвета, тогда принимается цвет самой верхней рамки, расположенной левее всех (в языках с написанием слева направо; в противном случае берется самая верхняя рамка, расположенная правее всех).

Следующие стили и разметка, результат действия которых представлен на рис. 11.9, помогают проиллюстрировать каждое из четырех правил:

```
table {border-collapse: collapse;
  border: 3px outset gray;}
td {border: 1px solid gray; padding: 0.5em;}
#r2c1, #r2c2 {border-style: hidden;}
#r1c1, #r1c4 {border-width: 5px;}
#r2c4 {border-style: double; border-width: 3px;}
#r3c4 {border-style: dotted; border-width: 2px;}
#r4c1 {border-bottom-style: hidden;}
#r4c3 {border-top: 13px solid silver;}
```

```
<table>
<tr>
<td id="r1c1">1-1</td><td id="r1c2">1-2</td>
<td id="r1c3">1-3</td><td id="r1c4">1-4</td>
</tr>
<tr>
```

```
<td id="r2c1">2-1</td><td id="r2c2">2-2</td>
<td id="r2c3">2-3</td><td id="r2c4">2-4</td>
</tr>
<tr>
<td id="r3c1">3-1</td><td id="r3c2">3-2</td>
<td id="r3c3">3-3</td><td id="r3c4">3-4</td>
</tr>
<tr>
<td id="r4c1">4-1</td><td id="r4c2">4-2</td>
<td id="r4c3">4-3</td><td id="r4c4">4-4</td>
</tr>
</table>
```

Рассмотрим по очереди, что происходит с каждой из ячеек:

- Для ячеек 1–1 и 1–4 рамки шириной пять пикселов были шире, чем рамки всех их соседей, поэтому они одержали верх не только над рамками смежных ячеек, но и над рамкой самой таблицы. Единственное исключение – нижняя рамка ячейки 1–1, которая была скрыта.
- Нижняя рамка ячейки 1–1 была скрыта, потому что ячейки 2–1 и 2–2 с их явно скрытыми рамками полностью удалили все рамки по краям ячеек. Рамка таблицы (слева от ячейки 2–1) уступила место рамке ячейки. Нижняя рамка ячейки 4–1 также была скрыта, поэтому не допустила появления рамки под ячейкой.
- Двойная рамка ячейки 2–4 шириной три пиксела была перекрыта сплошной рамкой ячейки 1–4 шириной пять пикселов. Рамка ячейки 2–4 в свою очередь закрыла рамку ячейки 2–3, потому что была и шире, и «интересней». Ячейка 2–4 также перекрыла рамку ячейки 3–4, даже несмотря на одинаковую с ней ширину, потому что стиль `double` ячейки 2–4 определен как более приоритетный, чем рамка `dotted` ячейки 3–4.
- Нижняя серебряная рамка в 13 пикселов ячейки 3–3 не только перекрыла верхнюю рамку ячейки 4–3, но также повлияла на компоновку содержимого обеих ячеек *и* строк, в которых эти ячейки находятся.

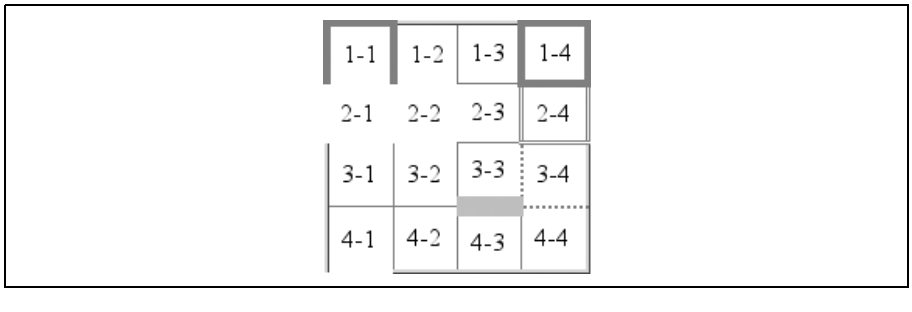


Рис. 11.9. *Изменение ширины, стилей и цветов рамок приводит к некоторым необычным результатам*

- Сплошные рамки шириной один пиксел ячеек, расположенных по внешнему краю таблицы и не оформленных специальными стилями, перекрыты трехпиксельной выпуклой рамкой самого элемента `table`.

Это, кстати, практически так же сложно, как и выглядит, хотя схемы поведения во многом интуитивно понятны и приобретают несколько больший смысл при отработке на практике. Однако следует отметить, что базовое представление HTML-таблицы в Netscape первого поколения может быть получено с помощью довольно простого набора правил, описанных здесь и проиллюстрированных на рис. 11.10:

```
table {border-collapse: collapse; border: 2px outset gray;}
td {border: 1px inset gray;}
```

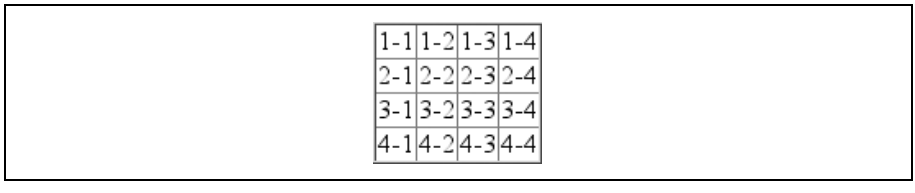


Рис. 11.10. *Представление таблиц в традициях старой школы*

Задание размеров таблиц

Итак, мы весьма подробно рассмотрели подробности форматирования таблиц и представления рамок ячеек, и у нас есть информация, необходимая для того, чтобы понимать, как задаются размеры таблиц и их внутренних элементов. Ширина таблицы может определяться в соответствии с одним из двух подходов: компоновка с фиксированной шириной и компоновка с автоматической шириной. Высота вычисляется автоматически независимо от того, по какому алгоритму вычисляется ширина.

Ширина

Поскольку существует два разных способа определения ширины таблицы, вполне логично, что должна быть возможность объявить, какой из них применяется для данной таблицы. Для выбора одного из двух алгоритмов вычисления ширины таблицы предназначено свойство `table-layout`.

Понятно, что эти две модели позволяют создавать различные макеты конкретной таблицы, но более фундаментальное их отличие кроется в скорости отрисовки. Агент пользователя быстрее рассчитает макет для таблицы с фиксированной шириной, чем для таблицы с автоматически определяемой шириной.

Фиксированная компоновка

Основная причина такой быстроты модели фиксированной компоновки в том, что в этом случае компоновка не зависит от содержимого яче-

table-layout

Значения:	`auto | fixed | inherit`
Начальное значение:	`auto`
Область применения:	элементы, свойство `display` которых имеет значение `table` или `inline-table`
Наследование:	да
Вычисляемое значение:	как задано

ек таблицы. Она определяется значениями свойства `width` таблицы, столбцов и ячеек этой таблицы.

Работу модели фиксированной компоновки можно представить в виде последовательности простых этапов:

1. Любой элемент столбца, свойство `width` которого имеет значение, отличное от `auto`, определяет ширину этого столбца.
2. Если столбец имеет ширину `auto`, но свойство `width` ячейки первой строки таблицы этого столбца имеет значение, отличное от `auto`, эта ячейка определяет ширину столбца. Если ячейка охватывает несколько столбцов, ширина распределяется между столбцами.
3. Ширина всех столбцов, по-прежнему сохраняющих размер `auto`, по возможности максимально уравнивается.

В данном случае ширина таблицы определяется или значением свойства `width` таблицы, или суммой ширин столбцов – тем, что *имеет большее значение*. Если таблица оказывается шире суммарной ширины всех ее столбцов, разница делится на число столбцов и добавляется к каждому из них.

Скорость при применении этого подхода велика, потому что значения ширины всех столбцов определяются первой строкой таблицы. Размеры ячеек любой последующей строки приводятся в соответствие ширинам столбцов, определенным первой строкой. Ячейки в этих строках не меняют ширины столбца, а это означает, что любое значение свойства `width`, заданное для этих ячеек, будет проигнорировано. Если содержимое не помещается, значение свойства `overflow` ячейки определяет, будет ли содержимое отсечено, будет ли оно видимым или будет генерироваться полоса прокрутки.

Рассмотрим следующие стили и разметку, результат применения которых показан на рис. 11.11:

```
table {table-layout: fixed; width: 400px;
  border-collapse: collapse;}
td {border: 1px solid;}
col#c1 {width: 200px;}
#r1c2 {width: 75px;}
#r2c3 {width: 500px;}
```

1-1	1-2	1-3	1-4
2-1	2-2	2-3	2-4
3-1	3-2	3-3	3-4
4-1	4-2	4-3	4-4

Рис. 11.11. *Макет таблицы с фиксированной шириной*

```
<table>
<colgroup>
<col id="c1"><col id="c2"><col id="c3"><col id="c4">
</colgroup>
<tr>
<td id="r1c1">1-1</td><td id="r1c2">1-2</td>
<td id="r1c3">1-3</td><td id="r1c4">1-4</td>
</tr>
<tr>
<td id="r2c1">2-1</td><td id="r2c2">2-2</td>
<td id="r2c3">2-3</td><td id="r2c4">2-4</td>
</tr>
<tr>
<td id="r3c1">3-1</td><td id="r3c2">3-2</td>
<td id="r3c3">3-3</td><td id="r3c4">3-4</td>
</tr>
<tr>
<td id="r4c1">4-1</td><td id="r4c2">4-2</td>
<td id="r4c3">4-3</td><td id="r4c4">4-4</td>
</tr>
</table>
```

Как видно из рис. 11.11, ширина первого столбца равна 200 пикселам, что составляет половину ширины таблицы в 400 пикселов. Второй столбец имеет ширину 75 пикселов, потому что ширина ячейки первой строки этого столбца была задана явно. Третий и четвертый столбцы – каждый по 61 пикселу шириной. Почему? Потому, что сумма ширин первого и второго столбцов (275px) плюс различные рамки между столбцами (3px) равняется 278 пикселов. 400 минус 278 получается 122, и 122 делится на два, что дает 61 пиксел – такова ширина третьего и четвертого столбцов. А что же ширина в 500 пикселов, заданная для `#r2c3`? Она игнорируется, потому что эта ячейка находится не в первой строке таблицы.

Обратите внимание, что для применения модели с фиксированной шириной совершенно не обязательно явно задавать ширину таблицы, хотя это, конечно, помогает. Например, исходя из следующего агент пользователя мог бы подсчитать ширину таблицы, которая получилась бы на 50 пикселов меньше ширины родительского элемента:

```
table {table-layout: fixed; margin: 0 25px;
  width: auto;}
```

Однако это не обязательно. Агенты пользователя могут построить любую таблицу, имеющую значение `auto` для свойства `width`, с помощью модели с автоматической шириной.

Автоматическая компоновка

Модель автоматической компоновки не так быстра, как фиксированная, но, вероятно, намного лучше знакома вам, потому что, по существу, аналогична модели, давно используемой HTML-таблицами. В большинстве современных агентов пользователя применение этой модели будет инициировано таблицей, свойство `width` которой имеет значение `auto`, независимо от значения `table-layout`, хотя это не гарантируется.

Причина, по которой процесс автоматической компоновки медленнее, в том, что таблица не может планироваться до тех пор, пока агент пользователя не просмотрит все ее содержимое. То есть при автоматической компоновке агент пользователя должен при каждом добавлении новой ячейки строить таблицу заново. При этом обычно агент пользователя выполняет некоторые вычисления, а затем вновь тщательно анализирует таблицу, чтобы выполнить вычисления второй очереди. Содержимое всех ячеек должно быть полностью обследовано, потому что, как и с таблицами HTML, компоновка таблицы зависит от него. Если в ячейке последней строки находится изображение шириной 400 пикселов, оно сделает ширину всех ячеек над ним (находящихся в этой же колонке) равной 400 пикселам. Таким образом, ширина каждой ячейки должна быть вычислена, и перед тем, как таблица сможет быть скомпонована, должны быть сделаны некоторые корректировки (возможно, это приведет к запуску второго круга вычислений ширины содержимого).

Работа модели может быть описана следующими этапами:

1. Для каждой ячейки столбца вычисляются минимальное и максимальное значения ширины ячейки.
   a. Определяется минимальная ширина, необходимая для представления содержимого. Помните, что содержимое может занимать различное число строк, но не может выходить за границы блока ячейки. Если значение свойства `width` ячейки больше, чем минимально возможная ширина, то оно принимается за минимальную ширину ячейки. Если ширина ячейки задана как `auto`, то минимальная ширина ячейки приравнивается к минимальной ширине содержимого.
   b. Для максимальной ширины определяется ширина, необходимая для представления содержимого без введения других, кроме заданных явно (например, элементом `<br>`), разрывов строки. Это значение и есть максимальная ширина ячейки.
2. Для каждого столбца вычисляются и минимальная, и максимальная ширина.
   a. Минимальная ширина столбца определяется по самой большой минимальной ширине ячейки этого столбца. Если для столбца

явно задано значение `width`, большее, чем любая из минимальных ширин ячеек этого столбца, то это значение `width` становится минимальной шириной столбца.

b. Для определения максимальной ширины берется максимальная ширина ячейки всех ячеек столбца. Если столбцу явно задано значение `width`, большее, чем любая из максимальных ширин ячеек этого столбца, то это значение `width` становится максимальной шириной столбца. Эти две модели поведения воссоздают традиционное поведение HTML-таблицы, принудительно расширяя любой столбец до размера его самой широкой ячейки.

3. В случаях, когда ячейка охватывает более одного столбца, сумма минимальных ширин столбцов должна быть равна минимальной ширине объединенной ячейки. Аналогично сумма максимальных ширин столбцов должна быть равна максимальной ширине объединенной ячейки. Агенты пользователя при любых изменениях ширин столбцов должны поровну распределять их величины между объединенными столбцами.

Кроме того, агент пользователя должен учитывать, что когда ширина столбца задана процентным значением, оно вычисляется относительно ширины таблицы, даже если он еще не знает, какой она будет! Он должен сохранить процентное значение и использовать его в следующей части алгоритма.

На данный момент агент пользователя уже определит, насколько широким или узким может быть каждый из столбцов. Имея в распоряжении эту информацию, он может найти действительное значение ширины таблицы. Это происходит следующим образом:

1. Если вычисляемая ширина таблицы не равна `auto`, она сравнивается с суммой ширин всех столбцов и рамок и промежутков между ячейками. (Ширина столбцов, заданная процентными значениями, скорее всего, вычисляется в этот момент.) Большее из этих двух значений и является окончательной шириной таблицы. Если вычисленная ширина таблицы больше суммы ширин столбцов, рамок и промежутков между ячейками, тогда ширина всех столбцов увеличивается на равные величины до тех пор, пока они не займут всю таблицу.

2. Если вычисляемая ширина таблицы равна `auto`, окончательная ее ширина определяется путем суммирования ширин столбцов, рамок и промежутков между ячейками. Это означает, что ширина таблицы будет именно такой, какая необходима для отображения ее содержимого, – точно так же, как и в обычных HTML-таблицах. Для всех столбцов, ширина которых задана процентным значением, он выступает в качестве ограничения, но такого, которое агент пользователь не обязан удовлетворять.

И только после завершения последнего этапа агент пользователя действительно может построить таблицу.

Следующие стили и разметка, результат применения которых представлен на рис. 11.12, помогают проиллюстрировать этот процесс:

```
table {table-layout: auto; width: auto;
  border-collapse: collapse;}
td {border: 1px solid;}
col#c3 {width: 25%;}
#r1c2 {width: 40%;}
#r2c2 {width: 50px;}
#r2c3 {width: 35px;}
#r4c1 {width: 100px;}
#r4c4 {width: 1px;}
```

```
<table>
<colgroup>
<col id="c1"><col id="c2"><col id="c3"><col id="c4">
</colgroup>
<tr>
<td id="r1c1">1-1</td><td id="r1c2">1-2</td>
<td id="r1c3">1-3</td><td id="r1c4">1-4</td>
</tr>
<tr>
<td id="r2c1">2-1</td><td id="r2c2">2-2</td>
<td id="r2c3">2-3</td><td id="r2c4">2-4</td>
</tr>
<tr>
<td id="r3c1">3-1</td><td id="r3c2">3-2</td>
<td id="r3c3">3-3</td><td id="r3c4">3-4</td>
</tr>
<tr>
<td id="r4c1">4-1</td><td id="r4c2">4-2</td>
<td id="r4c3">4-3</td><td id="r4c4">4-4</td>
</tr>
</table>
```

Рассмотрим последовательно, что происходит с каждым из столбцов:

- В первом столбце ширина явно задана только для ячейки 4–1 – `width: 100px;`. Поскольку ее содержимое достаточно мало, `100px` становится и минимальной, и максимальной шириной столбца. (Если бы в столбце была ячейка, содержащая несколько предложений текста, максимальная ширина столбца была бы увеличена до любого значения, необходимого для представления всего текста без разрывов строк.)
- Для второго столбца было объявлено две ширины: свойству `width` ячейки 1–2 было присвоено значение `40%`, а для ячейки 2–2 было за-

1-1	1-2	1-3	1-4
2-1	2-2	2-3	2-4
3-1	3-2	3-3	3-4
4-1	4-2	4-3	4-4

***Рис. 11.12.** Автоматическая компоновка таблицы*

дано `width: 50px;`. Минимальная ширина этого столбца равна `50px`, а максимальная ширина – 40% окончательной ширины таблицы.

- В третьем столбце только ячейка 3–3 имеет явно заданную ширину (`35px`), но свойству `width` самого столбца было задано значение `25%`. Следовательно, минимальная ширина столбца составляет `35px`, а максимальная ширина – 25% окончательной ширины таблицы.
- В четвертом столбце только для ячейки 4–4 было явно задано значение свойства `width` (`1px`). Это меньше, чем минимальная ширина содержимого, поэтому и минимальная, и максимальная ширина столбца приравниваются минимальной ширине содержимого ячеек. После вычисления это значение составило 25 пикселов.

Теперь агент пользователя знает, что четыре столбца имеют следующие минимальные и максимальные ширины:

1. min 100px / max 100px
2. min 50px / max 40%
3. min 35px / max 25%
4. min 25px / max 25px

Таким образом, минимальная ширина таблицы – это сумма всех минимумов столбцов плюс рамки, что в сумме дает 215 пикселов. Максимальная ширина таблицы равна 130px + 65 процентов, что составляет 371,42857143 пикселов (исходя из того, что 130px соответствуют 35% общей ширины таблицы). Примем ее после округления дробной части равной 371 пикселу. Именно этим значением ширины будут руководствоваться агенты пользователя. Таким образом, ширина второго столбца будет равна 148 пикселам, а третьего – 93 пикселам. От агентов пользователя не требуется выбирать максимальное значение; они могут предпочесть другой образ действия.

Конечно, это был очень простой и незатейливый пример (хотя, возможно, он таким не показался): все содержимое имело практически одну ширину, и большинство значений ширины были заданы в пикселах. В случаях, когда таблица содержит GIF-разделители, абзацы текста, элементы формы и т. д., процесс определения макета таблицы, скорее всего, будет намного более длительным.

Высота

После всех этих усилий, затраченных на определение ширины таблицы, вы, вероятно, задаетесь вопросом, насколько же более сложным будет вычисление высоты. На самом деле с точки зрения CSS все совершенно просто, хотя разработчики броузеров, наверное, так не думают.

Самая простая для описания ситуация – та, в которой высота задана явно посредством свойства `height`. В подобных случаях высота таблицы определяется значением свойства `height`. Это значит, что таблица может быть выше или ниже суммы высот ее строк, хотя спецификация CSS 2.1

от 11 апреля 2006 года гласит, что height трактуется как минимальная высота блоков таблицы. В таких случаях спецификация CSS2.1 отказывается явно определять, что должно произойти, но отмечает, что эта проблема может быть решена в будущих версиях CSS. Агент пользователя мог бы растягивать или сжимать строки таблицы, чтобы они соответствовали ее высоте, или делать пропуски внутри блока таблицы, или что-то совершенно другое. Выбор остается за агентом пользователя.

Если высота таблицы задана как auto, она вычисляется как сумма высот всех строк таблицы плюс все рамки и промежутки между ячейками. Вычисления, выполняемые агентом пользователя при определении высоты каждой строки, аналогичны тем, которые он выполнял при определении ширины столбцов. Он подсчитывает минимальную и максимальную высоту содержимого каждой ячейки и затем использует эти значения для выведения минимальной и максимальной высоты строки. Проделав эту операцию для всех строк, агент пользователя определяет, какой должна быть высота каждой строки, выстраивает их друг над другом и использует эти вычисления для подсчета высоты таблицы. Это во многом похоже на строковую компоновку, только с меньшей определенностью в том, как все должно делаться.

Кроме неясности по поводу таблиц с явно заданными высотами и тем, как интерпретировать высоты их строк, в список моментов, которые CSS2.1 не оговаривает, можно добавить следующие:

- Результат применения процентных значений для задания высоты ячеек таблицы.
- Результат применения процентных значений для задания высоты строк и групп строк таблицы.
- Как объединяющая строки ячейка влияет на высоту строк, которые объединены, кроме того, что строки должны вмещать объединенную ячейку.

Как видите, вычисление высоты таблиц преимущественно оставлено за агентами пользователя. Исторический опыт подсказывает, что в результате все агенты пользователя будут поступать по-своему, поэтому, вероятно, авторы должны максимально избегать задания высот.

Выравнивание

Совершенно непонятно почему, но выравнивание содержимого в ячейках описано намного лучше, чем высоты ячеек и строк. Это справедливо даже для вертикального выравнивания, которое может влиять на высоту строки.

Самым простым является горизонтальное выравнивание. Для выравнивания содержимого ячейки предназначено свойство text-align. В сущности, ячейка рассматривается как блочный элемент, и все ее содержимое выравнивается в соответствии со значением text-align (text-align описан в главе 6).

Чтобы выровнять содержимое ячейки таблицы по вертикали, применяется свойство `vertical-align`. Многие его значения аналогичны тем, которые управляют вертикальным выравниванием строкового содержимого, но смысл этих значений в случае применения к ячейке таблицы меняется. Приведем три простейших случая:

`top`

Верх содержимого ячейки выравнивается по верху строки; а в случае объединенных ячеек верх содержимого ячейки выравнивается по верху первой из охватываемых ею строк.

`bottom`

Низ содержимого ячейки выравнивается по низу строки; в случае объединенных ячеек низ содержимого ячейки выравнивается по низу последней из охватываемых ею строк.

`middle`

Середина содержимого ячейки выравнивается по середине строки; в случае объединенных ячеек середина содержимого ячейки выравнивается по середине всех охватываемых ею строк.

Эти примеры проиллюстрированы на рис. 11.13, где применяются следующие стили и разметка:

```
table {table-layout: auto; width: 20em;
  border-collapse: separate; border-spacing: 3px;}
td {border: 1px solid; background: silver;
  padding: 0;}
div {border: 1px dashed gray; background: white;}
#r1c1 {vertical-align: top; height: 10em;}
#r1c2 {vertical-align: middle;}
#r1c3 {vertical-align: bottom;}
```

```
<table>
<tr>
<td id="r1c1">
<div>
The contents of this cell are top-aligned.
</div>
</td>
<td id="r1c2">
```

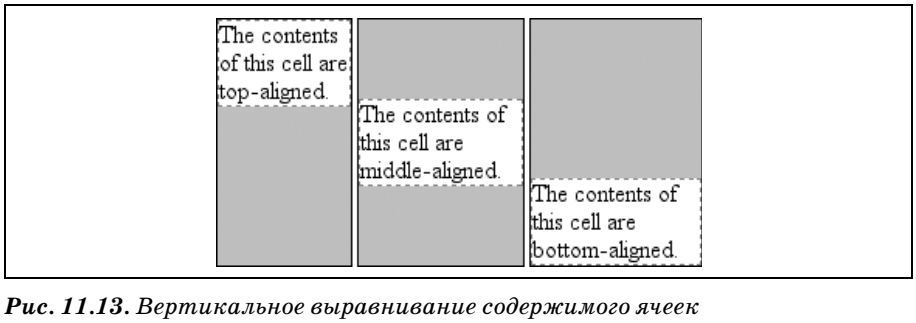


Рис. 11.13. *Вертикальное выравнивание содержимого ячеек*

```
<div>
The contents of this cell are middle-aligned.
</div>
</td>
<td id="r1c3">
<div>
The contents of this cell are bottom-aligned.
</div>
</td>
</tr>
</table>
```

В каждом из случаев выравнивание осуществляется за счет автоматического увеличения отступов самой ячейки для достижения необходимого эффекта. В первой ячейке на рис. 11.13 нижний отступ ячейки был изменен, чтобы уравнять разницу между высотой блока ячейки и высотой ее содержимого. Для второй ячейки отступы сверху и снизу были уравнены, чтобы обеспечить вертикальное центрирование содержимого ячейки. В последней ячейке был изменен верхний отступ.

Четвертое возможное значение выравнивания – `baseline`, и оно немного сложнее первых трех:

`baseline`

> Базовая линия ячейки выравнивается по базовой линии строки; в случае объединенных ячеек базовая линия ячейки выравнивается по базовой линии первой охватываемой ею строки.

Проще представить иллюстрацию (рис. 11.14), а затем обсудить, что происходит.

Базовая линия строки определяется расположенной ниже всех в строке первой базовой линией ячейки (т. е. базовой линией первой строки текста). Таким образом, на рис. 11.14 базовая линия строки была определена третьей ячейкой, у которой первая базовая линия расположена ниже всех. Тогда базовая линия первой строки текста двух первых ячеек будет выровнена по базовой линии строки.

Как и для выравнивания верха, середины и низа, выровненное по базовой линии содержимое ячейки размещается за счет изменения верх-

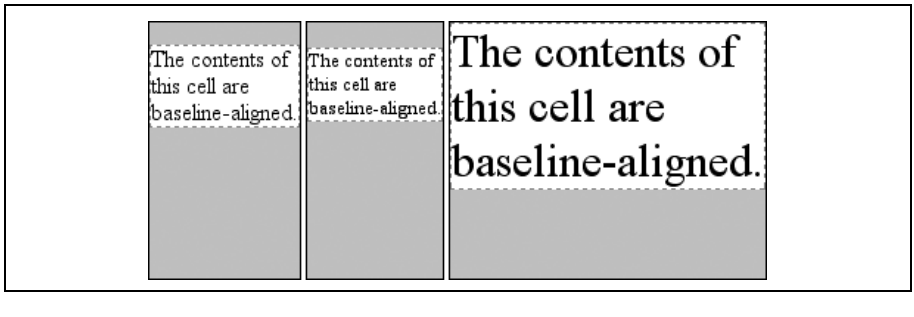


Рис. 11.14. *Выравнивание содержимого ячеек по базовой линии*

него и нижнего отступов ячеек. Если ни одна из ячеек строки не выравнивается по базовой линии, строка даже не имеет базовой линии, она ей просто не нужна.

Детализируем процесс выравнивания содержимого ячеек строки:

1. Если какая-либо ячейка строки выравнивается по базовой линии, сначала определяется базовая линия строки, а затем происходит размещение содержимого выровненных по базовой линии ячеек.
2. Размещается содержимое всех ячеек, выровненных по верху. Теперь строка имеет предварительную высоту, которая определяется низом той ячейки, которая расположена ниже всех ячеек, содержимое которых уже размещено.
3. Если среди оставшихся ячеек есть выровненные по середине или по низу и высота содержимого больше, чем предварительная высота строки, высота строки увеличивается, чтобы вместить самую высокую из этих ячеек.
4. Размещается содержимое во всех оставшихся ячейках. Для любой ячейки, содержимое которой ниже высоты строки, отступы увеличиваются, чтобы привести ее высоту в соответствие высоте строки.

Значения `sub`, `super`, `text-top` и `text-bottom` свойства `vertical-align` в применении к ячейкам таблицы игнорируются. Таким образом, применение следующего правила имело бы эффект, аналогичный представленному на рис. 11.14:

```
th {vertical-align: text-top;}
```

Заключение

Даже если за долгие годы практической работы верстка таблиц стала для вас простым делом, механизмы, управляющие ею, довольно сложны и не все точно описаны. Из-за наследия структуры HTML-таблиц модель таблиц CSS остается строкоориентированной, но она, слава богу, предусматривает столбцы и ограниченное стилевое оформление. Благодаря новым возможностям определения выравнивания ячеек и ширины таблиц, теперь в распоряжении авторов оказалось еще больше инструментальных средств оформления таблиц.

Возможность применять табличные значения свойства `display` к любым элементам открывает путь к созданию подобных таблицам макетов в HTML с помощью таких элементов, как `div`, или в языках XML, где любой элемент может применяться для описания компонентов макета. На момент написания данной книги многие броузеры помимо Internet Explorer поддерживают применение табличных значений `display` к произвольным элементам. Даже в его теперешней форме CSS делает представление более сложным и совершенным, что иллюстрирует генерируемое содержимое – тема, раскрываемая в следующей главе.

12

Списки и генерируемое содержимое

В применении CSS для верстки списки представляют особый интерес. Элементы списка – это простые наборы блочных элементов, имеющие дополнительный блок, «подвешенный» сбоку, который, собственно, не участвует в компоновке документа. В нумерованном списке в этом блоке содержатся возрастающие числа (или буквы), которые вычисляются и форматируются главным образом агентом пользователя, а не автором. Руководствуясь структурой документа, агент пользователя генерирует номера и их основное представление.

Ни одна из этих возможностей генерирования содержимого не могла быть описана в CSS1, и, следовательно, ими нельзя было управлять. Но CSS2 содержит средства, позволяющие описывать нумерацию элементов списка. Теперь CSS предоставляет автору возможность определять собственные шаблоны и форматы нумерации и ассоциировать эти счетчики с *любым* элементом, а не только с элементами упорядоченного списка. Более того, этот механизм позволяет вводить в документ другие виды содержимого, в том числе строки текста, значения атрибутов или даже внешние ресурсы. Таким образом, теперь средствами CSS можно вставлять значки ссылок, редакторских символов и т. д., не создавая для этого дополнительную разметку.

Чтобы увидеть, как совмещаются все эти возможности списков, мы изучим основы оформления списков, а затем перейдем к рассмотрению генерирования содержимого и счетчикам.

Списки

В некотором смысле практически все, что не является повествовательным текстом, может считаться списком. Перепись населения США, Солнечная система, мое генеалогическое дерево, ресторанное меню и даже все друзья, какие у вас когда-либо были, – все может быть пред-

ставлено в виде списка или, возможно, списка списков. Такое разнообразие применений делает списки очень важными, вот почему досадно, что оформление списков в CSS не усовершенствовано еще больше.

Самый простой (и поддерживаемый лучше всех) способ повлиять на стили списка состоит в том, чтобы изменить тип его маркера. *Маркер* (*marker*) элемента списка – это, например, жирная метка, располагающаяся рядом с каждым элементом ненумерованного списка. В нумерованном списке маркером может быть буква, число или символ любой другой системы счета. В качестве маркеров могут выступать даже изображения. Все это осуществляется с помощью различных свойств стилевого оформления списков.

Типы списков

Чтобы изменить тип маркера, обозначающего элементы списка, применяют свойство `list-style-type`.

Да, ключевых слов многовато. Многие из них были введены в CSS2, но затем изъяты в CSS2.1. В табл. 12.1 перечислены ключевые слова, присутствующие в CSS2.1.

Таблица 12.1. Ключевые слова свойства list-style-type в CSS2.1

Ключевое слово	Эффект от применения
`disc`	В качестве маркера элементов списка выступает диск (обычно закрашенный кружок)
`circle`	В качестве маркера выступает кружок (обычно незакрашенный)
`square`	В качестве маркера выступает квадрат (закрашенный или незакрашенный)
`decimal`	1, 2, 3, 4, 5, ...
`decimal-leading-zero`	01, 02, 03, 04, 05, ...
`upper-alpha` `upper-latin`	A, B, C, D, E, ...
`lower-alpha` `lower-latin`	a, b, c, d, e, ...
`upper-roman`	I, II, III, IV, V, ...
`lower-roman`	i, ii, iii, iv, v, ...
`lower-greek`	Классические греческие символы нижнего регистра
`armenian`	Традиционная армянская нумерация
`georgian`	Традиционная грузинская нумерация
`none`	Маркера нет

В табл. 12.2 перечислены ключевые слова, которые были введены в CSS2, но отсутствуют в CSS2.1.

list-style-type

Значения CSS2.1:	`disc | circle | square | decimal | decimal-leading-zero | lower-roman | upper-roman | lower-greek | lower-latin | upper-latin | armenian | georgian | none | inherit`
Значения CSS2:	`disc | circle | square | decimal | decimal-leading-zero | upper-alpha |lower-alpha | upper-roman | lower-roman | lower-greek | hebrew | armenian | georgian | cjk-ideographic | hiragana | katakana | hiragana-iroha | none | inherit`
Начальное значение:	`disc`
Область применения:	элементы, свойство `display` которых имеет значение `list-item`
Наследование:	да
Вычисляемое значение:	как задано

Таблица 12.2. Ключевые слова свойства list-style-type в CSS2

Ключевое слово	Эффект от применения
`hebrew`	Традиционная еврейская нумерация
`cjk-ideographic`	Идеографическая нумерация
`katakana`	Японская нумерация (A, I, U, E, O...)
`katakana-iroha`	Японская нумерация (I, RO, HA, NI, HO...)
`hiragana`	Японская нумерация (a, i, u, e, o...)
`hiragana-iroha`	Японская нумерация (i, ro, ha, ni, ho...)

Агент пользователя должен интерпретировать любое неизвестное ему значение как `decimal`.

Свойство `list-style-type`, как и все остальные свойства списков, может применяться только к элементу, свойство `display` которого имеет значение `list-item`, но CSS не различает нумерованные и ненумерованные элементы списка. Таким образом, на роль маркеров нумерованного списка можно вместо чисел определить диски. Кстати, по умолчанию свойству `list-style-type` присваивается значение `disc`, поэтому теоретически, если не объявлять противоположное явно, маркерами элементов всех списков (нумерованных и ненумерованных) будут диски. Это было бы логично, но, как выясняется, решение принимает агент пользователя. Даже если последний не имеет предопределенного правила, такого как `ol {list-style-type: decimal;}`, он может запретить применение упорядоченных маркеров к ненумерованным спискам, и наоборот. Но уверенности в том, что он это сделает, нет, поэтому будьте аккуратны.

Для значений CSS2, таких как `hebrew` и `georgian`, спецификация CSS2 не определяет точно ни то, как работают эти системы счета, ни то, как агенты пользователя должны поступать с ними. Такая неясность приводит к отсутствию реализации, вот почему значения, приведенные в табл. 12.2, исчезли из CSS2.1.

Для того чтобы запретить отображение маркеров, надо задать значение `none`. В этом случае агент пользователя воздерживается от размещения чего бы то ни было в том месте, где должен находиться маркер, хотя это не прерывает счета в нумерованных списках. Таким образом, применение следующей разметки обеспечило бы результат, приведенный на рис. 12.1:

```
ol li {list-style-type: decimal;}
li.off {list-style-type: none;}
```

```
<ol>
<li>Item the first
<li class="off">Item the second
<li>Item the third
<li class="off">Item the fourth
<li>Item the fifth
</ol>
```

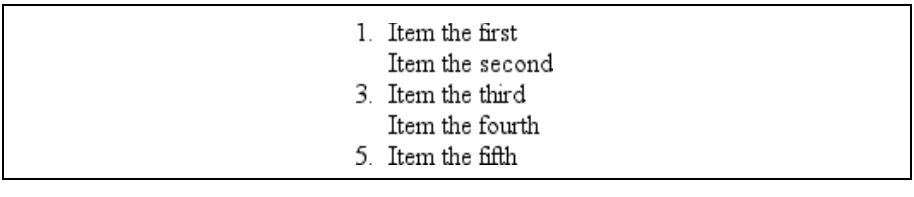


***Рис. 12.1.** Отключение маркеров элементов списка*

Свойство `list-style-type` – наследуемое, поэтому если требуется применить во вложенных списках другой стиль маркеров, их, скорее всего, надо будет определять отдельно. Вероятно, придется также прямо объявлять стили для вложенных списков, потому что они уже могли быть определены таблицей стилей агента пользователя. Предположим, что в агенте пользователя заданы следующие стили:

```
ul {list-style-type: disc;}
ul ul {list-style-type: circle;}
ul ul ul {list-style-type: square;}
```

Если это так (и, скорее всего, так оно и будет), придется объявлять собственные стили, чтобы перекрыть стили агента пользователя. В подобном случае наследования недостаточно.

Изображения для элементов списка

Иногда обычный маркер заменяют изображением, привлекая для этого свойство `list-style-image`.

list-style-image

Значения:	<uri> | none | inherit
Начальное значение:	none
Область применения:	элементы, свойство display которых имеет значение list-item
Наследование:	да
Вычисляемое значение:	для значений <uri> – абсолютный URI; в противном случае – none

Вот так:

```
ul li {list-style-image: url(ohio.gif);}
```

Да, на самом деле все просто. Задайте значение `url()`, и маркеры будут заменены изображениями, как показано на рис. 12.2.

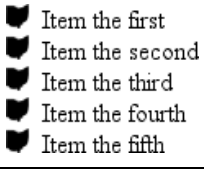


Рис. 12.2. *Применение изображений в качестве маркеров*

Конечно, имея дело с изображениями, необходимо проявлять осторожность, что совершенно очевидно из примера, представленного на рис. 12.3:

```
ul li {list-style-image: url(big-ohio.gif);}
```

Рис. 12.3. *Применение слишком больших изображений в качестве маркеров*

В общем случае целесообразно предусмотреть резервный тип маркеров. Они вступят в дело, если основное изображение не загружается, повреждено или его формат таков, что отображается не всеми агентами пользователя. Это можно сделать, объявив для списка запасной list-style-type:

```
ul li {list-style-image: url(ohio.png); list-style-type: square;}
```

Кроме того, свойству list-style-image можно присвоить применяемое по умолчанию значение none. Идея правильная, потому что свойство list-style-image наследуется, т. е. в любых вложенных списках в качестве маркера будет выступать одно и то же изображение, если только не отменить такое поведение явно:

```
ul {list-style-image: url(ohio.gif); list-style-type: square;}
ul ul {list-style-image: none;}
```

Вложенный список наследует тип элемента square, при этом указано, что в качестве маркеров не могут выступать изображения, поэтому во вложенном списке роль маркеров играют квадраты, как показано на рис. 12.4.

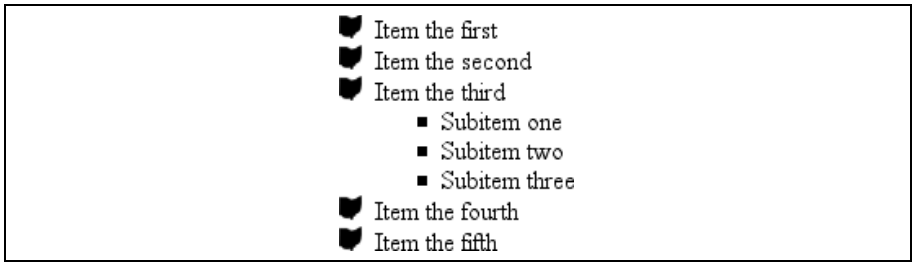


***Рис. 12.4.** Отключение маркеров-изображений во вложенных списках*

Помните, что такой сценарий может не заработать в реальной жизни: если агент пользователя уже определил list-style-type для ul ul, то значение square не будет унаследовано. Ваш браузер может вести себя иначе.

Местоположение маркера списка

В CSS2.1 предусмотрены и другие средства воздействия на представление элементов списка: можно располагать маркер вне или внутри содержимого элемента списка. Это делается при помощи свойства list-style-position.

Если местоположение маркера задано значением outside (применяется по умолчанию), то вид списка будет традиционным для Всемирной паутины. Можно немного видоизменить представление, если ввести маркер в содержимое, задав значение inside. При этом маркер размещается внутри содержимого элемента списка. Как именно это проис-

list-style-position	
Значения:	`inside | outside | inherit`
Начальное значение:	`outside`
Область применения:	элементы, свойство `display` которых имеет значение `list-item`
Наследование:	да
Вычисляемое значение:	как задано

ходит, не определено, но на рис. 12.5 показан один из возможных вариантов:

```
li.first {list-style-position: inside;}
li.second {list-style-position: outside;}
```

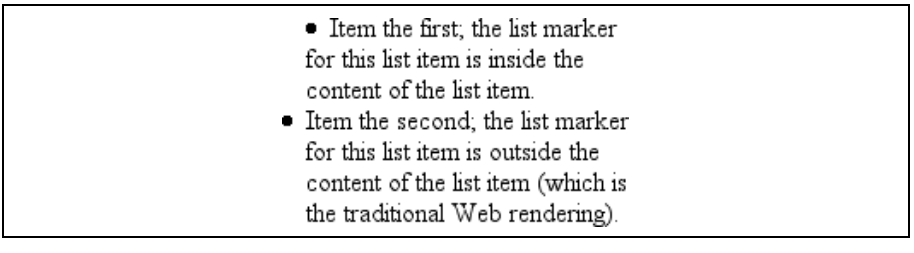


Рис. 12.5. *Размещение маркеров внутри и снаружи элементов списка*

Краткая форма задания стилей списка

Для краткости можно свести три свойства задания стиля списка в одно – `list-style`.

Например:

```
li {list-style: url(ohio.gif) square inside;}
```

Как видно на рис. 12.6, к элементам списка применяются все три значения.

list-style	
Значения:	`[ <list-style-type> || <list-style-image> || <list-style-position> ] | inherit`
Начальное значение:	смотрите отдельные свойства
Область применения:	элементы, свойство `display` которых имеет значение `list-item`
Наследование:	да
Вычисляемое значение:	см. отдельные свойства

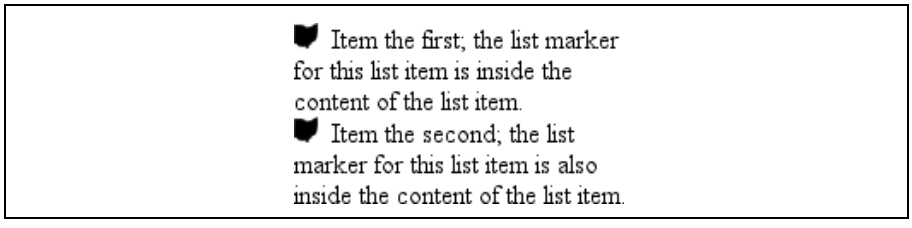


***Рис. 12.6.** Все вместе*

Значения свойства `list-style` могут перечисляться в произвольном порядке, любое из них может быть опущено. Если присутствует только одно значение, вместо остальных будут подставлены значения по умолчанию. Так, следующие два правила будут иметь совершенно аналогичный визуальный эффект:

```
li.norm {list-style: url(img42.gif);}
li.odd {list-style: url(img42.gif) disc outside;} /* то же самое */
```

Они совершенно одинаково переопределят все предыдущие правила. Например:

```
li {list-style-type: square;}
li.norm {list-style: url(img42.gif);}
li.odd {list-style: url(img42.gif) disc outside;} /* то же самое */
```

Результат будет аналогичным представленному на рис. 12.6, потому что применяемое в правиле для `li.norm` значение свойства `list-style-type`, `disc`, переопределит ранее объявленное значение `square`, точно так же явно заданное значение `disc` переопределяет его в правиле для `li.odd`.

Компоновка списка

Итак, мы рассмотрели основы оформления маркеров списка, и пора рассмотреть построение списков в различных броузерах. Начнем с набора из трех элементов списка, не имеющих никаких маркеров и еще не организованных в список, как показано на рис. 12.7.

Рамка вокруг элементов списка показывает, что они по сути представляют собой блочные элементы. В самом деле, значение `list-item` определено как генерирующее блочный элемент. Теперь добавим маркеры, как показано на рис. 12.8.

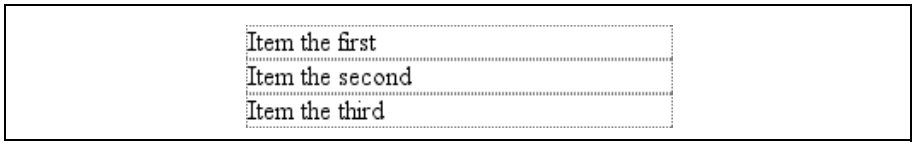


***Рис. 12.7.** Три элемента списка*

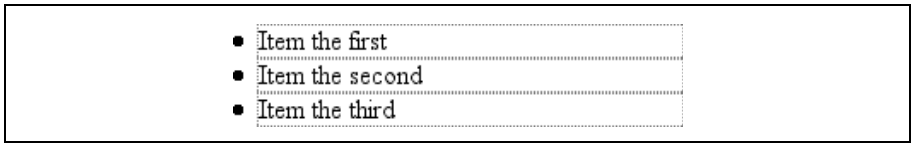


***Рис. 12.8.** Добавляются маркеры*

Расстояние между маркером и содержимым элемента списка в CSS не определено, и CSS2.1 не обеспечивает возможности влиять на это расстояние. Любопытно, что CSS2 делает это, о чем кратко рассказывается далее, во врезке «Позиционирование маркеров списка».

Маркеры, находящиеся вне содержимого элементов списка, не оказывают влияния ни на компоновку других элементов, ни даже на компоновку самих элементов списка. Они просто располагаются на определенном расстоянии от края содержимого, и как бы ни перемещался край содержимого, маркер следует за ним. Маркер ведет себя так, будто он абсолютно позиционирован относительно содержимого элемента списка, примерно следующим образом: `position: absolute; left: -1.5em;`. Когда маркер расположен внутри, он ведет себя как строковый элемент, находящийся в начале содержимого.

Осталось добавить фактический контейнер списка; иначе говоря, на рисунках не представлены ни элемент `ul`, ни элемент `ol`. Его можно добавить, как показано на рис. 12.9 (он обозначен пунктирной рамкой).

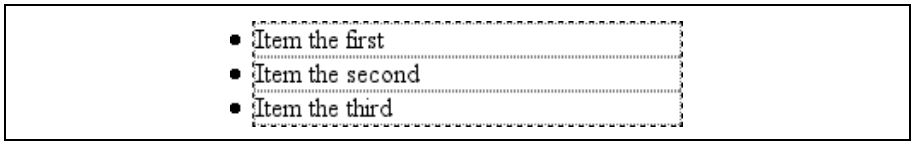


***Рис. 12.9.** Введение элемента «список»*

Как и элементы списка, элемент «список» – это блочный контейнер, который заключает в себе свои элементы-потомки. Однако, как видите, маркеры размещаются не только вне содержимого элементов списка, но также и вне области содержимого элемента «список». Обычное «структурированное расположение», которого можно было бы ожидать от списков, еще не определено.

Большинство броузеров на момент написания данной книги реализуют структурированное расположение элементов списка за счет отступов или полей их списка-контейнера. Например, агент пользователя мог бы применить такое правило:

```
ul, ol {margin-left: 40px;}
```

Это правило реализовано в Internet Explorer и Opera (см. рис. 12.9). А большинство броузеров компании Netscape Communications применяют такое правило:

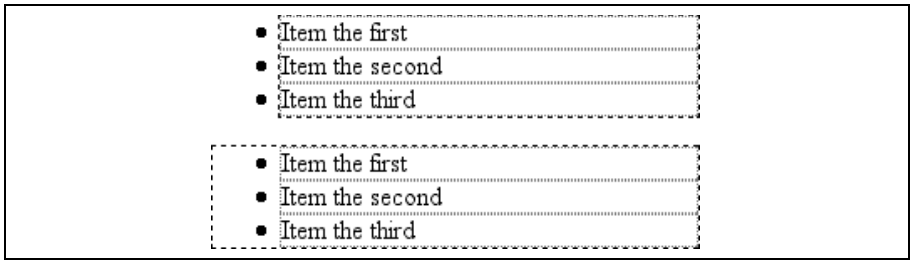


Рис. 12.10. *Поля и отступы в качестве средств структурирования*

```
ul, ol {padding-left: 40px;}
```

Все они правильные, но это различие может породить трудности, если понадобится устранить структурирование элементов списка. Рисунок 12.10 показывает разницу между этими двумя подходами.

Расстояние в `40px` – это наследие ранних веб-броузеров, структурирование списков в которых осуществлялось заданием расстояний в пикселах. Лучше было бы использовать что-то вроде `2.5em`, поскольку при этом происходило бы масштабирование структурирующих отступов при изменении размера текста.

Авторам, желающим изменить величину структурирующих отступов списков, я настоятельно рекомендую задавать и отступы, и поля, чтобы обеспечить совместимость с различными броузерами. Например, для того чтобы структурировать список при помощи отступов, примените такое правило:

```
ul {margin-left: 0; padding-left: 1em;}
```

А если предпочитаете поля, напишите примерно следующее:

```
ul {margin-left: 1em; padding-left: 0;}
```

В любом случае помните, что маркеры позиционируются относительно содержимого элементов списка и поэтому могут оказаться вне основного текста документа или даже за границами окна броузера.

Генерируемое содержимое

CSS2 и CSS2.1 включают новую возможность – *генерируемое содержимое* (*generated content*). Это содержимое, которое создается броузером, но не представлено ни разметкой, ни содержимым.

Например, маркеры списка – это генерируемое содержимое. В разметке элемента списка нет ничего, что прямо представляло бы маркеры, и автору не приходится добавлять маркеры в содержимое документа. Броузер просто генерирует соответствующий маркер автоматически. Для ненумерованных списков маркером является какая-нибудь метка,

Позиционирование маркеров списка

Возможность управлять расстоянием между маркером и содержимым элемента списка – это то, что запрашивают многие авторы. CSS2 определил средства для этого, среди которых можно назвать свойство `marker-offset` (смещение-маркера) и значение `marker` свойства `display`. Но приобретенный с тех пор опыт реализации продемонстрировал, что такой подход неудобен, и эти возможности были изъяты из CSS2.1.

На момент написания данной книги рабочий проект модуля «Списки» CSS3 описывает новый и более компактный способ определения местоположения маркера, а именно – псевдо-элемент `::marker`. Если предположить, что этот модуль не изменится и станет рекомендацией, возможно, когда-нибудь можно будет написать такие правила: `li::marker {margin-right: 0.125em;}`, чтобы расположить маркеры вплотную с содержимым элементов списка, не помещая маркеры внутрь.

кружок, диск или квадрат. В нумерованных списках это счетчик, который увеличивается на единицу для каждого последующего элемента.

Чтобы понять, как можно влиять на маркеры и настраивать нумерацию элементов списков (или что-нибудь другое!), сначала необходимо подробнее остановиться на основах генерируемого содержимого.

На момент написания данной книги ни одна из версий Internet Explorer не поддерживает генерируемое содержимое.

Вставка генерируемого содержимого

Для вставки генерируемого содержимого в документ применяются псевдоэлементы `:before` и `:after`. Они помещают перед содержимым элемента или после него генерируемое содержимое, заданное в виде свойства (как описано в следующем разделе).

Пусть требуется перед каждой гиперссылкой вставить текст «(link)», чтобы обозначить гиперссылки при выводе. Это можно осуществить с помощью следующего правила, эффект применения которого показан на рис. 12.11:

(link)Jeffrey seems to be (link)very happy about (link)something, although I can't quite work out whether his happiness is over (link)OS X, (link)Chimera, the ability to run the Dock and (link)DragThing at the same time, the latter half of my (link)journal entry from yesterday, or (link)something else entirely.

Рис. 12.11. *Генерирование текстового содержимого*

```
a[href]:before {content: "(link)";}
```

Обратите внимание, что между генерируемым содержимым и содержимым элемента нет пробела. Дело в том, что в содержимое предыдущего примера пробел не входит. Объявление можно изменить, чтобы гарантировать, что между генерируемым и имеющимся содержимым будет пробел:

```
a[href]:before {content: "(link) ";}
```

Разница невелика, но очень важна.

Аналогичным образом можно помещать маленький значок в конце ссылок на документы PDF. Правило может выглядеть примерно так:

```
a.pdf-doc:after {content: url(pdf-doc-icon.gif);}
```

Предположим, требуется продолжить оформление этих ссылок и окружить их рамкой. Правило такое:

```
a.pdf-doc {border: 1px solid gray;}
```

Результат применения этих двух правил показан на рис. 12.12.

Обратите внимание, что рамка окружает и генерируемое содержимое, а на рис. 12.11 подчеркивание ссылки продолжается и под текстом «(link)». Причина в том, что генерируемое содержимое размещается внутри блока элемента. В CSS2.1 нет возможности поместить никакое генерируемое содержимое, кроме маркеров списка, вне блока элемента.

Может показаться, что позиционирование позволило бы добиться желаемого результата, но CSS2 и CSS2.1 специально запрещают перемещение или позиционирование содержимого элементов `:before` и `:after`. Свойства стилевого оформления списков, как и свойства таблиц, запрещены. Кроме того, действуют следующие ограничения:

- Если областью действия селекторов `:before` или `:after` является блочный элемент, свойство `display` может принимать только значения `none`, `inline`, `block` и `marker`. Все остальные значения интерпретируются как `block`.
- Если областью действия селекторов `:before` или `:after` является строковый элемент, свойство `display` может принимать только значения `none` и `inline`. Все остальные значения интерпретируются как `inline`.

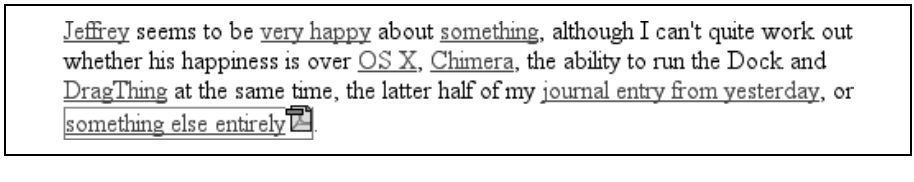


Рис. 12.12. *Генерирование значков*

Рис. 12.13. *Генерирование блочного содержимого*

Например:

```
em:after {content: " (!) "; display: block;}
```

Поскольку `em` – это строковый элемент, генерируемое содержимое не может стать блочным элементом. Следовательно, значение `block` заменяется значением `inline`. В следующем примере генерируемое содержимое становится блочным элементом, потому что целевой элемент также блочный:

```
h1:before {content: "New Section"; display: block; color: gray;}
```

Результат показан на рис. 12.13.

У генерируемого содержимого есть интересная особенность: оно наследует значения от элемента, к которому прикрепляется. Таким образом, в соответствии со следующими правилами, генерируемый текст будет зеленым, как и текст абзаца:

```
p {color: green;}
p:before {content: "::: ";}
```

Для того чтобы сделать генерируемый текст фиолетовым, достаточно простого объявления:

```
p:before {content: "::: "; color: purple;}
```

Подобное наследование значений происходит, конечно, только в наследуемых свойствах. Об этом следует помнить, поскольку такая реализация влияет на использование некоторых эффектов. Рассмотрим:

```
h1 {border-top: 3px solid black; padding-top: 0.25em;}
h1:before {content: "New Section"; display: block; color: gray;
  border-bottom: 1px dotted black; margin-bottom: 0.5em;}
```

Генерируемое содержимое размещается внутри блока элемента `h1`, поэтому оно будет находиться под верхней рамкой элемента. Оно также размещается с учетом отступов, как показано на рис. 12.14.

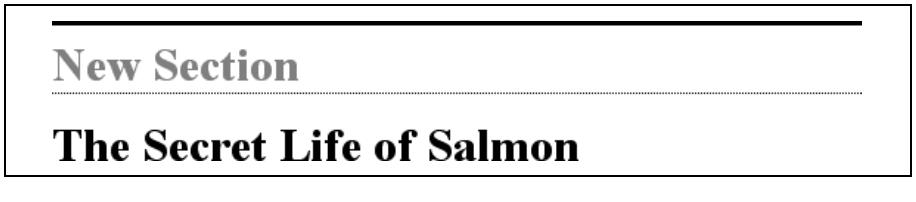


Рис. 12.14. *Учет местоположения*

Рис. 12.15. *Превращение генерируемого содержимого в строковый элемент*

Нижнее поле генерируемого содержимого, преобразованное в блочный элемент, сдвигает фактическое содержимое элемента вниз на 0,5 em. В любом случае эффект от применения генерируемого содержимого в этом примере – разделение элемента `h1` на две части: блок генерируемого содержимого и блок фактического содержимого. Дело в том, что для генерируемого содержимого задано `display: block`. Если изменить это объявление на `display: inline`, получится эффект, показанный на рис. 12.15:

```
h1 {border-top: 3px solid black; padding-top: 0.25em;}
h1:before {content: "New Section"; display: inline; color: gray;
  border-bottom: 1px dotted black; margin-bottom: 0.5em;}
```

Заметьте, как размещаются рамки и как по-прежнему учитывается отступ сверху. То же происходит и с нижним полем генерируемого содержимого, но поскольку теперь генерируемое содержимое является строковым и, следовательно, поля не влияют на высоту строки, применение поля не имеет видимого эффекта.

Определившись с основами генерируемого содержимого, посмотрим, как оно определяется фактически.

Определение содержимого

Для того чтобы генерировать содержимое, необходим способ описания этого содержимого. Как вы уже видели, это делается с помощью свойства `content`, но оно предоставляет гораздо больше возможностей, чем вы видели до сих пор.

content

Значения:	`normal` | [<строка> | <uri> | <счетчик> | `attr(`<идентификатор>`)` | `open-quote` | `close-quote` | `no-open-quote` | `no-close-quote`]+ | `inherit`
Начальное значение:	`normal`
Область применения:	псевдоэлементы `:before` и `:after`
Наследование:	нет
Вычисляемое значение:	для значений <uri> – абсолютный URI; для ссылок на атрибуты – результирующая строка; в противном случае – как задано

¶ **Spawning**

Рис. 12.16. *Строки отображаются дословно*

Вы уже видели в действии значения «строка» и «URI», а счетчики будут рассмотрены чуть позже. Но перед тем как обратиться к `attr()` и значениям, определяющим кавычки, поговорим о строках и URI немного более подробно.

Строковые значения представляются буквально, даже если они содержат текст, который мог бы быть воспринят как разметка. Поэтому текст из следующего правила будет вставлен в документ дословно, как показано на рис. 12.16:

```
h2:before {content: "<em>&para;</em> "; color: gray;}
```

Это значит, что если требуется включить в генерируемое содержимое символ перевода строки, нет смысла использовать `<br>`. Вместо него используется комбинация `\A` – способ представления перевода строки в CSS (на основании шестнадцатеричного числа 0A, представляющего собой код символа перевода строки в Unicode). И наоборот, если имеется длинная строка, разбитая на несколько коротких, то символы перевода строки экранируются при помощи \. Эта возможность продемонстрирована следующим правилом и проиллюстрирована рис. 12.17:

```
h2:before {content: "We insert this text before all H2 elements because \
it is a good idea to show how these things work. It may be a bit long \
but the point should be clearly made.  "; color: gray;}
```

We insert this text before all H2 elements because it is a good idea to show how these things work. It may be a bit long but the point should be clearly made. **Spawning**

Рис. 12.17. *Вставка и подавление символов перевода строки*

Слэш позволяет также обратиться к значениям Unicode, таким как `\00AB`.

На момент написания данной книги поддержка добавления экранируемого содержимого, например `\A` и `\00AB`, распространена не очень широко, даже в тех броузерах, которые поддерживают генерируемое содержимое.

Значения URI просто указывают на внешний ресурс (изображение, фильм, звуковой клип или что-либо еще, поддерживаемое агентом

пользователя), который вставляется в соответствующее место документа. Если агент пользователя по какой-то причине не поддерживает ресурсы того типа, на который вы указываете, – скажем, вы пытаетесь вставить SVG-изображение в броузер, который не понимает SVG, или вставить фильм в документ при его распечатке – тогда от агента пользователя требуется полностью проигнорировать ресурс и ничего не вставлять.

Вставка значений атрибутов

Иногда требуется сделать значение атрибута элемента частью представления документа. Например, можно поместить значение атрибута `href` каждой ссылки сразу после ссылок, как здесь:

```
a[href]:after {content: attr(href);}
```

В результате генерируемое содержимое вклинивается прямо в содержимое документа. Для улучшения внешнего вида документа можно добавить в объявление символьные строки – результат показан на рис.12.18:

```
a[href]:after {content: " [" attr(href) "]";}
```

Jeffrey [http://www.zeldman.com/] seems to be very happy [http://www.zeldman.com/daily/1202b.shtml#joy] about something [http://www.zeldman.com/i/accessories/worthit.jpg], although I can't quite work out whether his happiness is over OS X [http://www.apple.com/macosx/], Chimera [http://chimera.mozdev.org/], the ability to run the Dock and DragThing [http://www.dragthing.com/] at the same time, the latter half of my journal entry from yesterday [http://www.meyerweb.com/eric/thoughts/2002b.html#t20021227], or something else entirely [http://www.roguelibrarian.com/].

***Рис. 12.18.** Добавление URL*

Это полезно, например, для таблиц стилей, используемых при печати. В качестве генерируемого содержимого может быть введено любое значение атрибута: текст `alt`, значения `class` или `id` – все, что угодно. Автор мог бы сделать явной информацию о цитате для блока цитат:

```
blockquote:after {content: "(" attr(cite) ")"; display: block;
  text-align: right; font-style: italic;}
```

В данном случае более сложное правило могло бы показать значения цвета текста и ссылки существующего документа:

```
body:before {content: "Text: " attr(text) " | Link: " attr(link)
  " | Visited: " attr(vlink) " | Active: " attr(alink);
  display: block; padding: 0.33em;
  border: 1px solid black; text-align: center;}
```

Обратите внимание, что если атрибут отсутствует, то он замещается пустой строкой. Именно это происходит на рис. 12.19, где предыдущий пример применяется к документу, у элемента `body` которого нет атрибута `alink`.

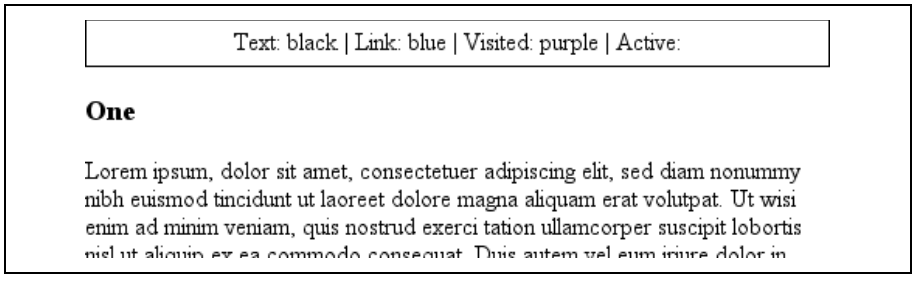


***Рис. 12.19.** Отсутствующие атрибуты пропускаются*

Как видите, в документ вставляется текст «Active: » (включая пробел в конце строки), но после него ничего нет. Это удобно, если требуется вставить значение атрибута, только если он существует.

CSS2.x определяет возвращаемое значение атрибута как не подлежащую синтаксическому разбору строку. Следовательно, если значение атрибута содержит разметку или спецсимволы, они будут отображаться буквально.

Генерируемые кавычки

Особой формой генерируемого содержимого являются кавычки. CSS2.x предоставляет мощную поддержку обоих типов кавычек и их поведения в случае вложенности. Это стало возможным благодаря сочетанию значения `open-quote` свойства `content` и свойства `quotes`.

Изучая синтаксис значений, мы обнаруживаем, что кроме ключевых слов `none` и `inherit` единственным действительным значением является одна или несколько *пар* строковых значений. Первое строковое значение пары определяет символ открывающей кавычки, а второе – символ закрывающей кавычки. Таким образом, из следующих двух объявлений действительно только первое:

```
quotes: ' " '  " ' "; /* действительное */
quotes: ' " '; /* НЕДЕЙСТВИТЕЛЬНОЕ */
```

Первое правило также иллюстрирует один из способов, позволяющих заключить в кавычки сами кавычки. Двойные кавычки заключаются в одинарные, и наоборот.

quotes

Значения:	[<строка> <строка>]+ | none | inherit
Начальное значение:	зависит от агента пользователя
Область применения:	все элементы
Наследование:	да
Вычисляемое значение:	как задано

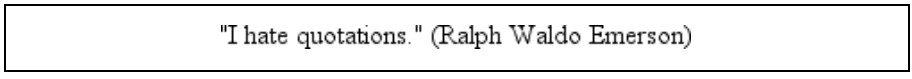


Рис. 12.20. *Вставка кавычек и другого содержимого*

Обратимся к простому примеру. Предположим, вы создаете XML-формат для хранения списка любимых цитат. Вот одна из записей списка:

```
<quotation>
 <quote>I hate quotations.</quote>
 <quotee>Ralph Waldo Emerson</quotee>
</quotation>
```

Чтобы с представленными данными было удобно иметь дело, можно применить следующие правила, результат показан на рис. 12.20:

```
quotation: {display: block;}
quote {quotes: '"' '"';}
quote:before {content: open-quote;}
quote:after {content: close-quote;}
quotee:before {content: " (";}
quotee:after {content: ")";}
```

Значения `open-quote` и `close-quote` применяются для вставки любых подходящих символов кавычек (поскольку в различных языках применяются разные кавычки). Так, эта цитата начинается и заканчивается двойными кавычками.

Если обычные для большинства печатных устройств вертикальные прямые кавычки требуется заменить «скругленными»[1] (curly quotes), правило `quote` примет вид:

```
quote {quotes: '\201C' '\201D';}
```

Здесь заданы шестнадцатеричные коды Unicode символов «скругленных» кавычек, и в результате применения данного правила к предыдущему выражению цитата Эмерсона будет заключена в скругленные, а не в прямые кавычки, как на рис. 12.20.

Свойство `quotes` позволяет определять шаблоны цитат с любым количеством уровней вложенности. В английском языке, например, цитаты обычно начинаются с двойных кавычек, а вложенные цитаты выделяются одинарными кавычками. Следующие правила задают применение «скругленных» кавычек:

```
quotation: display: block;}
quote {quotes: '\201C' '\201D' '\2018' '\2019';}
quote:before, q:before{content: open-quote;}
quote:after, q:after {content: close-quote;}
```

[1] Называемые также английскими двойными. – *Примеч. ред.*

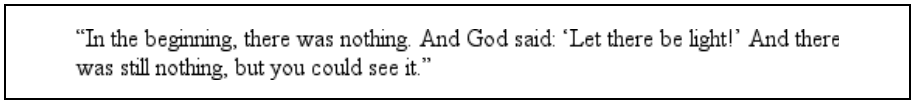


Рис. 12.21. *Вложенные скругленные кавычки*

Если применить эти правила к следующей разметке, получится результат, показанный на рис. 12.21:

```
<quotation>
 <quote> In the beginning, there was nothing. And God said: <q>Let there
  be light!</q> And there was still nothing, but you could see it.</quote>
</quotation>
```

Если уровень вложенности кавычек превышает количество заданных для них пар, то для всех более глубоких уровней применяется последняя пара. Таким образом, если применить к разметке, показанной на рис. 12.21, следующее правило, то внутренняя цитата, как и внешняя, будет заключена в двойные кавычки:

```
quote {quotes: '\201C' '\201D';}
```

Генерируемые кавычки делают возможным еще один обычный типографский прием. Если в кавычки заключается несколько абзацев, то закрывающие кавычки во всех абзацах, кроме последнего, обычно опускаются, и показываются только открывающие кавычки. Этого можно добиться, задав значение `no-close-quote`:

```
blockquote {quotes: '“' '”' “'” “'” '“' '”';}
blockquote p:before   {content: open-quote;}
blockquote p:after  {content: no-close-quote;}
```

При этом каждый абзац будет начинаться двойными открывающими кавычками, но закрывающих кавычек не будет. Это касается и последнего абзаца, поэтому, если потребуется добавить закрывающие кавычки, надо будет классифицировать последний абзац и объявить `close-quote` для его `:after`-содержимого.

Важность этого значения в том, что оно уменьшает уровень вложенности цитаты, не вставляя символ. Вот почему каждый абзац начинается именно с двойных кавычек и двойные и одинарные кавычки не чередуются. Значение `no-close-quote` закрывает вложенный уровень в конце каждого абзаца, и таким образом все абзацы имеют один уровень вложенности.

Это важно потому, что, как отмечает спецификация CSS2.1, «Глубина цитирования не зависит от вложенности исходного документа или форматирующей структуры». Иначе говоря, когда вы начинаете уровень цитирования, он сохраняется для всех элементов до тех пор, пока не встретится закрывающая кавычка, которая понижает уровень вложенности.

Для полноты картины необходимо упомянуть и ключевое слово `no-open-quote`, имеющее противоположный `no-close-quote` эффект. Это ключевое слово увеличивает на единицу уровень вложенности цитаты, но не вставляет никакие символы.

Счетчики

Всем нам хорошо известны счетчики; например, маркеры элементов нумерованного списка – это счетчики. В CSS1 не было способа влиять на них, преимущественно потому, что в этом не было необходимости: HTML определил собственные правила счета для нумерованных списков, которые и использовались. С появлением XML возникла потребность в методе задания счетчиков. Однако CSS2 не удовольствовался простым предоставлением обычной нумерации, введенной в HTML. Два свойства и два значения свойства `content` делают возможным определять практически любой формат нумерации, включая счетчики подразделов с применением различных стилей, например «VII.2.c».

Сброс и приращение

Основа создания счетчиков – возможность задавать для них как точку отсчета, так и приращение на некоторую величину. Первое осуществляется с помощью свойства `counter-reset`.

Идентификатор счетчика – это просто метка, созданная автором. Счетчик подраздела можно было бы назвать `subsection`, `subsec`, `ss` или `bob`. Для создания идентификатора достаточно выполнить его сброс или приращение. В следующем правиле счетчик `chapter` определен через сброс:

```
h1 {counter-reset: chapter;}
```

По умолчанию счетчик обнуляется. Если требуется, чтобы нумерация начиналась с другого числа, его можно указать после идентификатора:

```
h1#ch4 {counter-reset: chapter 4;}
```

Также можно одновременно сбросить несколько идентификаторов, применяя пары идентификатор-целое. Если целое отсутствует, используется значение по умолчанию – нуль:

counter-reset

Значения:	[<идентификатор> <целое>?]+ | none | inherit
Начальное значение:	зависит от агента пользователя
Область применения:	все элементы
Наследование:	нет
Вычисляемое значение:	как задано

```
h1 {counter-reset: chapter 4 section -1 subsec figure 1;}
   /* 'subsec' сбрасывается в 0 */
```

Как следует из предыдущего примера, допускаются и отрицательные значения. Будет совершенно правомерно задать -32768 и начать нумерацию с этого значения.

CSS не определяет, что должны делать агенты пользователя с отрицательными значениями счетчиков в нечисловых системах счета. Например, не оговаривается, что делать, если значение счетчика равно -5, но стиль его представления — upper-alpha.

Для осуществления прямого счета необходимо свойство, показывающее, что элемент увеличивает значение счетчика. В противном случае счетчик сохранял бы значение, заданное ему объявлением counter-reset. Таким свойством является, что и не удивительно, counter-increment.

counter-increment

Значения:	[<идентификатор> <целое>?]+ | none | inherit
Начальное значение:	зависит от агента пользователя
Область применения:	все элементы
Наследование:	нет
Вычисляемое значение:	как задано

Как и counter-reset, counter-increment принимает пары идентификатор-целое, и целое число в этой паре может равняться нулю или быть как отрицательным, так и положительным. Разница в том, что если в свойстве counter-increment целое число опускается, по умолчанию ему присваивается значение 1, а не 0.

Например, вот как агент пользователя мог бы определить счетчики для воссоздания традиционной нумерации списков 1, 2, 3:

```
ol {counter-reset: ordered;}  /* по умолчанию - 0 */
ol li {counter-increment: ordered;}  /* по умолчанию - 1 */
```

Но автор мог бы захотеть считать от нуля в обратном направлении, т. е. нумерация элементов списка производилась бы по возрастающей по модулю в отрицательной числовой области. Для этого потребовалась бы всего лишь небольшая поправка:

```
ol {counter-reset: ordered;}  /* по умолчанию - 0 */
ol li {counter-increment: ordered -1;}
```

Нумерация списков тогда была бы такой: -1, -2, -3 и т. д. Если заменить -1 на -2, списки нумеровались бы так: -2, -4, -6 и т. д.

Применение счетчиков

Однако для действительного отображения счетчиков надо задать свойство `content` в сочетании с одним из определяющих счетчики значением. Чтобы увидеть принцип работы, рассмотрим нумерованный список в стиле XML:

```
<list type="ordered">
 <item>First item</item>
 <item>Item two</item>
 <item>The third item</item>
</list>
```

Применяя приведенные ниже правила к этой разметке XML, вы получили бы результат, показанный на рис. 12.22:

```
list[type="ordered"] {counter-reset: ordered;}  /* по умолчанию - 0 */
list[type="ordered"] item {display: block;}
list[type="ordered"] item:before {counter-increment: ordered;
 content: counter(ordered) ". "; margin: 0.25em 0;}
```

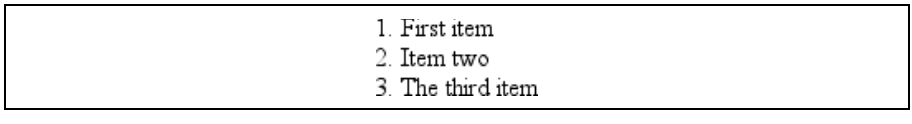


Рис. 12.22. *Нумерация элементов списка*

Обратите внимание, что генерируемое содержимое, как обычно, размещается как строковое содержимое в начале ассоциированного с ним элемента. Таким образом, эффект аналогичен HTML-списку, для которого объявлено `list-style-position: inside;`.

Также заметьте, что элементы `item` – это обычные элементы, генерирующие наборы блочных элементов, т. е. счетчики не ограничены лишь элементами, свойство `display` которых имеет значение `list-item`. Кстати, счетчик может использовать любой элемент. Рассмотрим следующие правила:

```
h1:before {counter-reset: section subsec;
  counter-increment: chapter;
  content: counter(chapter) ". ";}
h2:before {counter-reset: subsec;
  counter-increment: section;
  content: counter(chapter )"." counter(section) ". ";}
h3:before {counter-increment: subsec;
  content: counter(chapter) "." counter(section) "." counter(subsec) ". ";}
```

Результат их применения показан на рис. 12.23.

На рис. 12.23 показаны некоторые важные моменты процессов сброса и приращения счетчика. Обратите внимание, как элемент `h1` применяет счетчик `chapter`, который по умолчанию начинает нумерацию с нуля

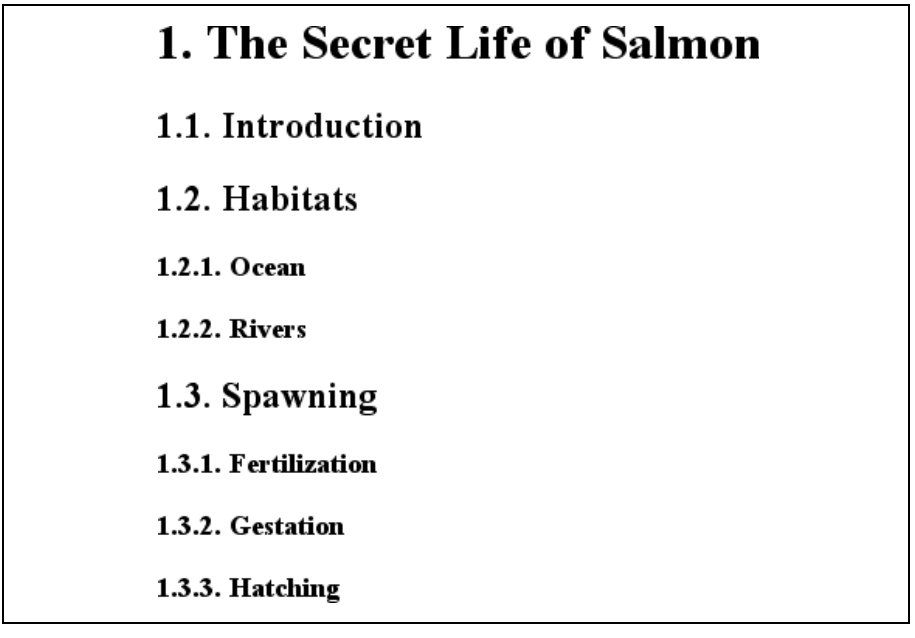


Рис. 12.23. *Введение нумерации заголовков*

и перед текстом элемента вставляет «1.». Если счетчик используется тем же элементом, который обеспечивает и его приращение, приращение происходит *до* выведения значения счетчика на экран. Аналогичным образом, если сброс и отображение счетчика происходят в одном элементе, сброс имеет место перед выведением значения счетчика на экран. Рассмотрим:

```
h1:before, h2:before, h3:before {
  content: counter(chapter) "." counter(section) "." counter(subsec) ". ";}
h1 {counter-reset: section subsec;
  counter-increment: chapter;}
```

Первому элементу `h1` документа предшествовал бы номер «1.0.0. », потому что счетчики `section` и `subsec` были заданы свойством `counter-reset`, а не `counter-increment`. Это значит, что если вы хотите, чтобы первый отображаемый номер счетчика, заданного свойством `counter-increment`, был равен 0, то должны задать его начальное значение равным `-1`, как здесь:

```
body {counter-reset: chapter -1;}
h1:before {counter-increment: chapter; content: counter(chapter) ". ";}
```

Счетчики позволяют реализовать некоторые интересные идеи. Рассмотрим следующий XML:

```
<code type="BASIC">
 <line>PRINT "Здравствуй, Мир!"</line>
```

```
 <line>REM Это то, что детишки называют "комментарием"</line>
 <line>GOTO 10</line>
</code>
```

Приведенные ниже правила позволяют воссоздать традиционный формат листинга программы на BASIC:

```
code[type="BASIC"] {counter-reset: linenum; font-family: monospace;}
code[type="BASIC"] line {display: block;}
code[type="BASIC"] line:before {counter-increment: linenum;
  content: counter(linenum 10) ": ";}
```

Также в формате `counter()` есть возможность определить стиль списка для каждого счетчика. Это можно сделать, добавляя отделенное запятой ключевое слово `list-style-type` после идентификатора счетчика. Результаты следующих изменений примера нумерации заголовков приведены на рис. 12.24:

```
h1:before {counter-reset: section subsec;
  counter-increment: chapter;
  content: counter(chapter,upper-alpha) ". ";}
h2:before {counter-reset: subsec;
  counter-increment: section;
  content: counter(chapter,upper-alpha)"." counter(section) ". ";}
h3:before {counter-increment: subsec;
  content: counter(chapter,upper-alpha) "." counter(section) "."
  counter(subsec,lower-roman) ". ";}
```

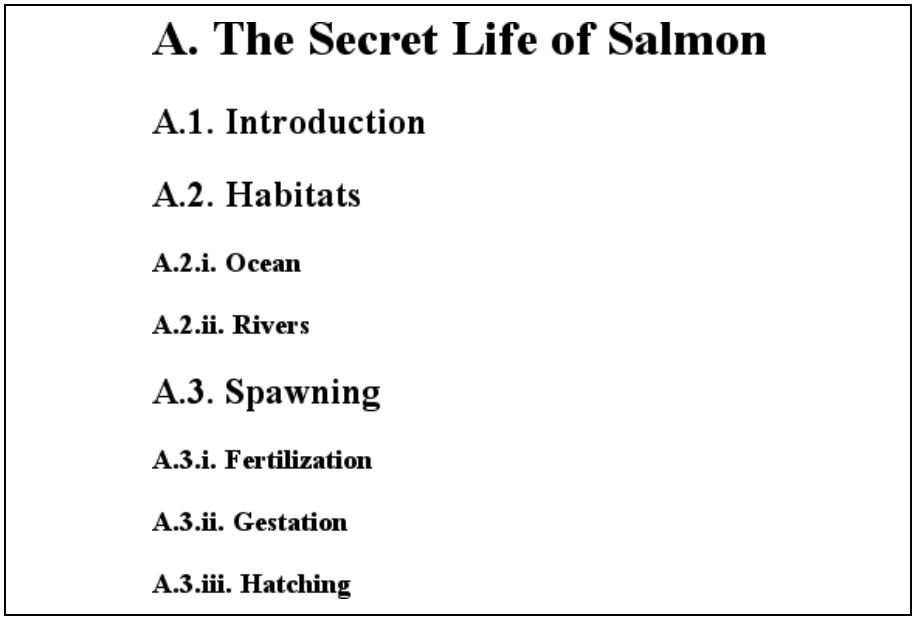


Рис. 12.24. *Изменение стилей счетчика*

Обратите внимание, что для счетчика section не было задано ключевое слово, определяющее стиль, поэтому по умолчанию для него выбрана десятичная нумерация. При желании для счетчиков можно даже задать стили disc, circle, square и none.

Еще интересно отметить, что элементы, свойство display которых имеет значение none, не обеспечивают приращения счетчиков, даже если кажется, что правила оговаривают обратное. В то же время элементы, свойство visibility которых имеет значение hidden, обеспечивают приращение счетчиков:

```
.suppress {counter-increment: cntr; display: none;}
  /* 'cntr' НЕ ПОЛУЧАЕТ приращения */
.invisible {counter-increment: cntr; visibility: hidden;}
  /* 'cntr' ПОЛУЧАЕТ приращение */
```

Счетчики и область видимости

На настоящий момент мы знаем, как связывать несколько счетчиков для создания нумерации разделов и подразделов. Часто именно этого желают авторы и для вложенных нумерованных списков, но попытки создания необходимого количества счетчиков для нумерации глубоко вложенных уровней могут быстро привести к сложностям. Нумерация пятикратно вложенных списков потребовала бы вот такой связки правил:

```
ol ol ol ol ol li:before {counter-increment: ord1 ord2 ord3 ord4 ord5;
  content: counter(ord1) "." counter(ord2) "." counter(ord3) "."
   counter(ord4) "." counter(ord5) ".";}
```

Представьте, сколько пришлось бы написать правил для нумерации вложенных списков с 50 уровнями вложенности! (Я не говорю, что вы должны создавать такие списки. Просто представьте себе это на мгновение.)

К счастью, CSS2.x описал для счетчиков концепцию *области видимости* (*scope*). Она гласит, что каждый уровень вложенности создает новую область видимости для любого заданного счетчика. Область видимости – это то, что обеспечивает возможность следующим правилам осуществлять нумерацию вложенного списка, как это обычно происходило в HTML:

```
ol {counter-reset: ordered;}
ol li:before {counter-increment: ordered;
 content: counter(ordered) ". ";}
```

Все эти правила, даже будучи вложенными в другие, создадут нумерованные списки, начиная счет с 1 и обеспечивая единичное приращение для каждого последующего элемента, – в точности так, как это происходило в HTML с самого начала.

Эта схема работает, потому что на каждом уровне вложенности создается новый экземпляр счетчика ordered. Итак, для первого нумерован-

ного списка создается экземпляр `ordered`. Затем для каждого списка, вложенного в первый, создается новый экземпляр, и нумерация каждого списка начинается сначала.

Но вам надо нумеровать списки так, чтобы каждый уровень вложенности создавал новый номер, добавляющийся к старому: 1, 1.1, 1.2, 1.2.1, 1.2.2, 1.3, 2, 2.1 и т. д. Этого нельзя добиться посредством свойства `counter()` (счетчик), но свойство `counters()` (счетчики) делает это возможным. Вот какое различие обеспечивает буква «s»!

Чтобы создать стиль вложенного счетчика, показанный на рис. 12.25, потребуются такие правила:

```
ol {counter-reset: ordered;}
ol li:before {counter-increment: ordered;
 content: counters(ordered,".") " - ";}
```

По существу, ключевое слово `counters(ordered,".")` отображает счетчик `ordered` каждой области видимости, добавляя в конце точку, и выстраивает подряд все счетчики, в области видимости которых находится данный элемент. Таким образом, элементу списка третьего уровня вложенности предшествуют значения `ordered` для самого внешнего списка, для списка, находящегося между внешним и текущим списками, и данного списка, каждое из которых заканчивается точкой. Затем в соответствии со значением свойства `content` после всех этих чисел добавляются пробел, дефис и еще один пробел.

Как и в случае со свойством `counter()`, можно задавать стили списка для вложенных счетчиков, но один и тот же стиль будет применен ко всем счетчикам. Таким образом, если изменить предыдущую CSS так,

1. - Lists
    1.1. - Types of Lists
    1.2. - List Item Images
    1.3. - List Marker Positions
    1.4. - List Styles in Shorthand
    1.5. - List Layout
2. - Generated Content
    2.1. - Inserting Generated Content
        2.1.1. - Generated Content and Run-In Content
    2.2. - Specifying Content
        2.2.1. - Inserting Attribute Values
        2.2.2. - Generated Quotes
    2.3. - Counters
        2.3.1. - Resetting and Incrementing
        2.3.2. - Using Counters
        2.3.3. - Counters and Scope
3. - Summary

Рис. 12.25. *Вложенные счетчики*

как показано ниже, то все элементы списка с рис. 12.25 использовали бы для нумерации вместо чисел строчные буквы:

```
ol li:before {counter-increment: ordered;
 content: counters(ordered,".",lower-alpha) ": ";}
```

Заключение

Даже несмотря на то, что стилевое оформление списков не настолько совершенно, как нам, возможно, хотелось бы, и поддержка генерируемого содержимого броузерами несколько фрагментарная (на момент написания данной книги, во всяком случае), возможность оформлять списки все-таки весьма полезна. Пример довольно обычного варианта применения – создать список ссылок, удалить из него маркеры и отступы и таким образом создать навигационную врезку. Перед сочетанием простой разметки и гибкой компоновки сложно устоять. Ожидаемые в CSS3 усовершенствования в оформлении списков дают основания надеяться, что популярность списков будет возрастать.

На данный момент в тех ситуациях, когда язык разметки не имеет собственных элементов списка, генерируемое содержимое может оказать неоценимую услугу: скажем, для введения такого содержимого, как значки, для обозначения определенных типов ссылок (PDF-файлов, документов Word или даже просто ссылок на другой веб-сайт). Генерируемое содержимое также устраняет все сложности при выведении на печать URL-ссылок, а возможность вставлять и форматировать кавычки просто вызывает восторг. Без колебаний можно сказать, что область применения генерируемого содержимого ограничена только фантазией автора. Более того, благодаря счетчикам теперь возможна нумерация элементов, не являющихся обычными списками, таких как заголовки или блоки кода. Итак, если вам захочется поддержать такие возможности в конструкции, имитирующей интерфейс пользователя операционной системы, читайте дальше. В следующей главе мы обсудим пути использования системных цветов и шрифтов в CSS-дизайне.

13

Стили пользовательского интерфейса

Основная часть CSS связана со стилевым оформлением документов, но CSS предлагает также массу полезных инструментов оформления интерфейсов. Например, разработчики Mozilla создали его интерфейс (и интерфейс многих его клонов) с помощью *расширенного языка пользовательского интерфейса* (eXtensible User interface Language – XUL). CSS и CSS-подобные определения лежат в основе средств XUL, реализующих представления навигационных кнопок, вкладок, диалоговых окон, окон состояния и других частей интерфейса.

Точно так же заданные пользователем параметры среды могут определять шрифты и цвета документа и даже влиять на выделение фокуса ввода и форму курсора мыши. Возможности интерфейса, созданного на основе CSS2, могут не только сделать работу пользователя приятной, но и сбить его с толку, если не будут тщательно проработаны.

Системные шрифты и цвета

В некоторых случаях вы хотите, чтобы ваш документ максимально точно копировал внешний вид компьютерной среды пользователя. Наглядный пример: вы создаете веб-приложение и хотите, чтобы веб-компонент выглядел как часть операционной системы. Хотя CSS2 и не обеспечивает возможности использовать в документах каждый отдельно взятый вид элементов операционной системы, у вас есть доступ к огромному разнообразию цветов и доступных шрифтов.

Системные шрифты

Скажем, вы создали элемент, функционирующий как кнопка (с помощью JavaScript, например). Делая элемент управления похожим на кнопку, мы реализуем ожидания пользователя о внешнем виде элемента и таким образом делаем его более удобным в использовании.

Чтобы реализовать приведенный пример, достаточно написать такое правило:

```
a.widget {font: caption;}
```

Согласно этому правилу шрифт любого элемента `a` класса `widget` будет принадлежать к тому же семейству шрифтов, иметь те же размер, насыщенность, стиль и вариант, что и текст элементов управления, имеющих надпись, таких как кнопка.

CSS2 определяет шесть системных ключевых слов для задания шрифта. Они перечислены в следующем списке:

`caption`

Стили шрифтов, используемых для элементов, имеющих надпись, таких как кнопки и выпадающие списки.

`icon`

Стили шрифтов, применяемые для подписи значков операционной системы, например, изображающих жесткие диски, папки и файлы.

`menu`

Стили шрифтов, применяемые для представления текста в выпадающих меню и списках меню.

`message-box`

Стили шрифтов, применяемые для представления текста в диалоговых окнах.

`small-caption`

Стили шрифтов, применяемые для подписи небольших элементов управления.

`status-bar`

Стили шрифтов, применяемые для вывода текста в строках состояния окон.

Важно понимать, что эти значения могут задаваться только в свойстве `font` и сами по себе представляют собой форму сокращенной записи. Допустим, что для подписей значков операционной системы пользователя выбран 10-пиксельный шрифт Geneva без применения эффектов полужирного, курсивного начертания или капители. Это значит, что следующие три правила эквивалентны (результат представлен на рис. 13.1):

```
body {font: icon;}
body {font: 10px Geneva;}
body {
  font-weight: normal;
  font-style: normal;
  font-variant: normal;
  font-size: 10px;
```

```
    font-family: Geneva;
}
```

Таким образом, простое значение, например `icon`, на самом деле объединяет ряд других значений. Для CSS это довольно необычно, и поэтому работать с такими значениями несколько сложнее.

В качестве примера предположим, что вы хотите задать такой же стиль шрифта, какой применяется в подписях значков, но полужирного начертания, даже если для меток пиктограмм в системе пользователя оно не применяется. Тут понадобилось бы правило со следующим порядком определений:

```
body {font: icon; font-weight: bold;}
```

Если определения записать в таком порядке, то агент пользователя в соответствии с первым из них задаст для элемента `body` шрифт, совпадающий с шрифтом подписей значков, а затем изменит насыщенность этого шрифта в соответствии со вторым определением. Если бы порядок написания был обратным, то значение свойства `font` переопределило бы значение `font-weight` из второго определения и правило о полужирном начертании было бы утеряно. Это соответствует способу, которым обрабатываются сокращенные записи свойств (как для собственно свойства `font`).

Вас может удивить отсутствие базового семейства шрифтов, поскольку обычно авторам рекомендуется писать что-то вроде `Geneva, sans-serif;` (в том случае, если броузер пользователя не поддерживает заданный шрифт). CSS не разрешит добавить базовое семейство шрифтов, но в этом случае в нем нет необходимости. Если агент пользователя может получить семейство шрифтов, используемое для представления какого-либо элемента в интерфейсе операционной системы, будьте уверены, что этот же шрифт доступен и для броузера.

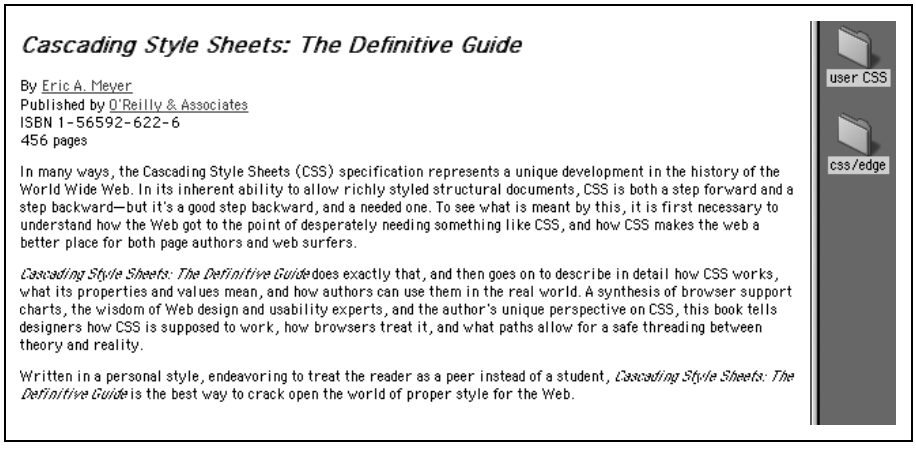


Рис. 13.1. *Текст выглядит как подпись под значком*

Если запрашиваемый системный шрифт недоступен или не может быть обнаружен, агенту пользователя разрешено брать близкие стили шрифтов. Например, приблизительный вариант стиля `small-caption` можно получить, уменьшая размер шрифта `caption`. Если такие приближения невозможны, агент пользователя должен взять применяемый по умолчанию шрифт агента пользователя.

Системные цвета

На момент написания данной книги рабочий проект модуля CSS3 «Цвет» рекомендует вместо ключевых слов системных цветов применять новое свойство `appearance`. Аналогично CSS2.1 не рекомендует применять эти ключевые слова, т. к. ожидаются изменения в CSS3. Авторам настоятельно рекомендуется не использовать системные цвета, поскольку, скорее всего, их не будет в следующих версиях CSS. Эта информация приводится по той причине, что некоторые доступные в настоящий момент броузеры поддерживают системные цвета.

Для тех, кто испытывает необходимость в цветах, определенных в операционной системе пользователя, CSS2 описывает ряд ключевых слов системных цветов. Это значения, разрешенные в любых обстоятельствах, в которых допускается применение значения `<color>`. Например, можно сделать фон элемента аналогичным цвету рабочего стола пользователя, объявив:

```
div#test {background-color: Background;}
```

Или задать для документа применяемый по умолчанию цвет текста и фона окон системы:

```
body {color: WindowText; background: Window;}
```

В результате подобной настройки пользователь скорее всего сможет прочитать документ, поскольку должен был настроить свою операционную систему так, чтобы ею можно было пользоваться. (Если нет, он заслуживает того, что получит!)

Всего существует 28 системных цветов, хотя CSS не определяет их явно. Вместо этого приводится несколько базовых (и очень кратких) описаний значения каждого ключевого слова. В следующем списке перечислены все 28 ключевых слов. В тех случаях, когда существует прямая аналогия с параметрами вкладки «Appearance» (Оформление) утилиты «Display» (Свойства: Экран) панели управления Windows 2000, это отмечается в скобках после описания.

`ActiveBorder`

Цвет внешней рамки активного окна (первый цвет в «Active Windows Border» (Граница активного окна)).

`ActiveCaption`

Фоновый цвет заголовка активного в настоящий момент окна (первый цвет в «Active Title Bar» (Заголовок активного окна)).

`AppWorkspace`

Фоновый цвет, определенный в многодокументном приложении, например цвет фона, находящегося «под» открытыми документами Microsoft Word (первый цвет в «Application Background» (Окно)).

`Background`

Фоновый цвет рабочего стола (первый цвет в «Desktop» (Рабочий стол)).

`ButtonFace`

Цвет поверхности трехмерной кнопки.

`ButtonHighlight`

Цвет подсветки ребер трехмерных элементов изображения, находящихся с противоположной стороны от виртуального источника света. Таким образом, если виртуальный источник света расположен в верхнем левом углу, это цвет подсветки, применяемый к правому и нижнему ребрам элемента изображения.

`ButtonShadow`

Цвет тени трехмерных элементов изображения.

`ButtonText`

Цвет текста кнопок (цвет шрифта в «3D Objects» (Рельефные объекты)).

`CaptionText`

Цвет текста заголовков, элементов управления размерами и элементов полосы прокрутки (цвет шрифта в «Active Title Bar» (Заголовок активного окна)).

`GrayText`

Цвет текста неактивного элемента управления. Это ключевое слово интерпретируется как код `#000`, если текущий драйвер устройства отображения не поддерживает чистый серый цвет.

`Highlight`

Цвет элемента(-ов), выбранного(-ых) в элементе управления (первый цвет в «Selected Items» (Выделенный пункт меню)).

`HighlightText`

Цвет текста элемента(-ов), выбранного(-ых) в элементе управления (цвет шрифта в «Selected Items» (Выделенный пункт меню)).

`InactiveBorder`

Цвет внешней рамки неактивного окна (первый цвет в «Inactive Window Border» (Граница неактивного окна)).

`InactiveCaption`

Фоновый цвет заголовка неактивного окна (первый цвет в «Inactive Title Bar» (Заголовок неактивного окна)).

`InactiveCaptionText`

Цвет текста неактивного заголовка (цвет шрифта в «Inactive Title Bar» (Заголовок неактивного окна)).

`InfoBackground`

Цвет фона всплывающих подсказок (первый цвет в «ToolTip» (Всплывающая подсказка)).

`InfoText`

Цвет текста всплывающих подсказок (цвет шрифта в «ToolTip» (Всплывающая подсказка)).

`Menu`

Цвет фона меню (первый цвет в «Menu» (Строка меню)).

`MenuText`

Цвет текста меню (цвет шрифта в «Menu» (Строка меню)).

`Scrollbar`

Цвет полосы прокрутки.

`ThreeDDarkShadow`

Цвет насыщенной тени трехмерных элементов.

`ThreeDFace`

Цвет «лицевой» стороны трехмерных элементов.

`ThreeDHighlight`

Цвет подсветки трехмерных элементов.

`ThreeDLightShadow`

Светлый цвет трехмерных элементов (ребер, обращенных «лицом» к источнику света).

`ThreeDShadow`

Цвет тени трехмерных элементов.

`Window`

Цвет фона окна (первый цвет в «Window» (Окно)).

`WindowFrame`

Цвет рамки окна.

`WindowText`

Цвет текста окна (цвет шрифта в «Window» (Окно)).

Ключевые слова системных цветов определены в CSS2 как нечувствительные к регистру, но рекомендуется соблюдать принятое в них соче-

тание заглавных и строчных букв, приведенное в предыдущем списке, что делает названия цветов более удобными для чтения. Нетрудно убедиться, что `ThreeDLightShadow` проще понять с первого взгляда, чем `threedlightshadow`.

Очевидный недостаток неопределенной сущности ключевых слов системных цветов в том, что разные агенты пользователя могут интерпретировать их по-разному, даже если они выполняются в одной операционной системе. Поэтому, применяя эти ключевые слова, не рассчитывайте на абсолютное единообразие. Например, не стоит предлагать пользователю текст вроде «Ищите текст, цвет которого совпадает с цветом рабочего стола», поскольку пользователь, возможно, разместил на рабочем столе графическое изображение (обои), которое закрывает стандартный рабочий стол.

Курсоры

Курсор – еще одна важная часть пользовательского интерфейса (в спецификации CSS его называют «указателем»), который управляется такими устройствами, как мышь, координатно-указательное устройство, графический планшет или даже оптические системы считывания. В большинстве броузеров курсор применяется для обеспечения обратной связи; наглядный пример – изменение курсора на изображение руки с вытянутым указательным пальцем при прохождении по гиперссылкам.

Изменение курсора

CSS2 позволяет изменять значок курсора, что намного упрощает создание веб-приложений, функциональность которых аналогична настольным приложениям операционной системы. Например, над ссылкой на файлы справки курсор мог бы превратиться в значок «справка», например в вопросительный знак, как показано на рис. 13.2.

Это делается с помощью свойства `cursor`.

cursor

Значения:	`[[<uri>,]* [auto | default | pointer | crosshair | move | e-resize | ne-resize | nw-resize | n-resize | se-resize | sw-resize | s-resize | w-resize| text | wait | help | progress ]] | inherit`
Начальное значение:	`auto`
Область применения:	все элементы
Наследование:	да
Вычисляемое значение:	для значений `<uri>` – абсолютное значение; в противном случае – как задано

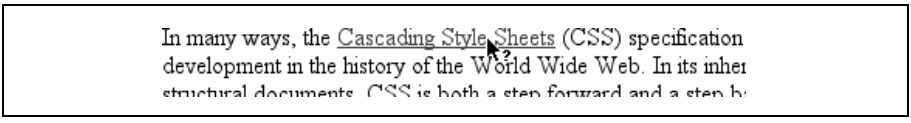


***Рис. 13.2.** Изменение значка курсора*

Если установлено значение по умолчанию, `auto`, то агент пользователя должен сам определить значок курсора, наиболее подходящий для текущего содержимого. Это не то же самое, что значение `default`, выбирающее системный курсор, применяемый по умолчанию. Стандартный курсор – это, как правило, стрелка, но не обязательно, все зависит от текущих настроек рабочей среды.

Указывающие курсоры и курсоры выделения

Значение `pointer` соответствует курсору, который появляется, когда пользователь проводит курсором по гиперссылке. Это поведение можно описать даже для документов HTML:

```
a[href] {cursor: pointer;}
```

Свойство `cursor` позволяет приказать любому элементу изменять значок, как это делают ссылки. Это может смутить пользователя, поэтому я не рекомендую злоупотреблять подобными уловками. С другой стороны, с помощью `cursor` намного проще создавать интерактивные, управляемые сценариями графические элементы экрана из элементов, не являющихся ссылками, и затем изменять значок курсора на соответствующий, как показано на рис. 13.3.

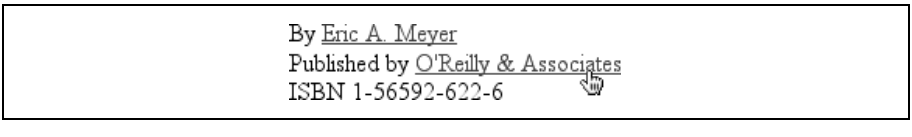


***Рис. 13.3.** Указание на интерактивность элемента*

Internet Explorer для Windows до версии IE6 не распознавал значения `pointer` и вместо него для значка «указывающая рука» использовал значение `hand`. IE6 распознает оба значения. Общая рекомендация – использовать подряд оба значения, как здесь:

```
#example {cursor: pointer; cursor: hand;}
```

Это не пройдет проверку достоверности, но обеспечит идентичный результат в более новых броузерах и старых версиях Explorer. Обратите внимание на порядок: значения нельзя поменять местами и думать, что это будет работать. Более подробно смотрите *http://developer.mozilla.org/en/docs/Giving_'cursor'_a_Hand*.

Другой значок курсора, очень распространенный в веб-броузерах, – значок `text`, который появляется в тех ситуациях, когда пользователь

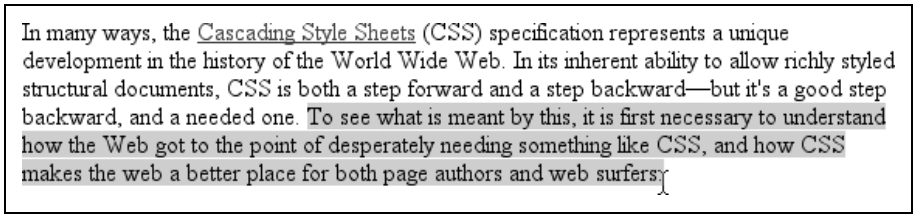


***Рис. 13.4.** Текст, который можно выбрать, и курсор выбора текста*

может выбрать текст. Обычно это значок в виде узкой вертикальной полоски, «I-bar», и служит он в качестве визуального сигнала о том, что пользователь может выделить содержимое, находящееся под курсором. На рис. 13.4 показан значок выбора текста, расположенный в конце некоторого уже выбранного текста.

Другой способ указать на интерактивность – задать значение `crosshair`, изменяющее значок курсора на символ перекрестия. Обычно это пара коротких линий, расположенных под углом 90° друг к другу, одна вертикальная и одна горизонтальная, что очень похоже на знак плюс (+). Однако перекрестие также могло бы быть похожим на знак умножения (или строчную букву «x») или даже на значок, напоминающий прицел винтовки. Перекрестия обычно используются в программах перехвата данных с экрана и могут быть полезны в ситуациях, когда пользователю необходимо знать, по какому именно пикселу происходит щелчок.

Курсоры перемещения

Часто значение `move` дает результат, аналогичный `crosshair`. Значение `move` применяется там, где автору надо показать, что элемент экрана может быть перемещен, и часто его визуальное представление генерируется как жирное перекрестие со стрелками на концах линий. Также оно может представлять собой «хватающую руку», пальцы которой сжимаются, когда пользователь нажимает и удерживает кнопку мыши. Два возможных визуальных представления `move` показаны на рис. 13.5.

Кроме того, существуют различные варианты значений свойства `cursor`, связанные со значением `move`: `e-resize`, `ne-resize` и т. д. Пользователи Windows и большинства графических оболочек Unix знают их как значки, которые появляются, когда курсор мыши располагается над краями или углами окна. Так, при размещении курсора над правым краем окна появится курсор `e-resize`, указывающий на то, что пользо-

***Рис. 13.5.** Разные значки для `move`*

ватель может изменить размер окна, потянув за правый край в ту или иную сторону. При помещении курсора над нижним левым углом появляется значок `sw-resize` курсора. Существует множество различных способов визуального представления этих значков, на рис. 13.6 показаны некоторые возможные варианты.

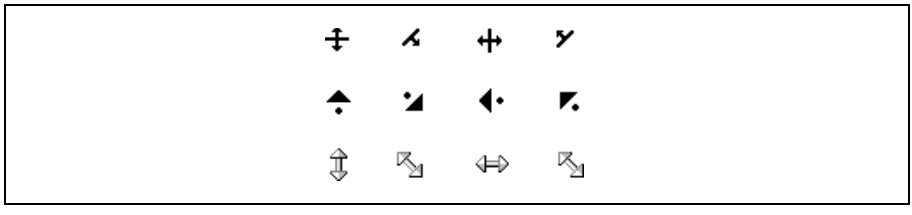


***Рис. 13.6.** Набор курсоров «изменения размеров»*

Ожидание и ход выполнения

И значение `wait`, и значение `progress` показывают, что программа занята выполнением. Однако они не идентичны: `wait` означает, что пользователь должен ждать до тех пор, пока программа освободится, тогда как `progress` показывает, что пользователь может свободно продолжать работу с программой, даже несмотря на то, что она занята. В большинстве операционных систем `wait` обозначается или как часы (возможно, с вращающимися стрелками), или как песочные часы (возможно, переворачивающиеся время от времени). Значение `progress` обычно представляется как вращающийся надувной пляжный мяч или стрелка с небольшими песочными часами справа. На рис. 13.7 показаны некоторые из этих значков.

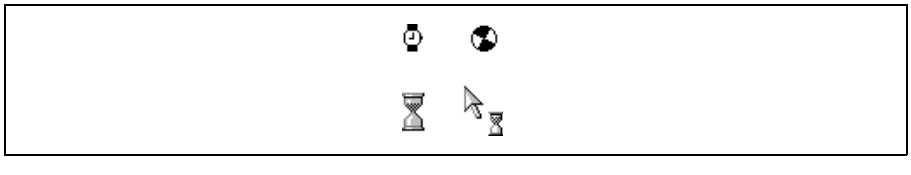


***Рис. 13.7.** Ожидание и ход выполнения*

Значение `progress` было введено в CSS2.1.

Предоставление справки

Когда автор хочет показать, что пользователь может получить некоторую справку, применяется значение `help`. Два самых распространенных представления `help` – знак вопроса и стрелка с маленьким вопро-

сом рядом. Значение help может быть очень полезным, если задан отдельный класс для ссылок, указывающих на более подробную информацию или информацию, которая поможет пользователю лучше понять веб-страницу. Например:

```
a.help {cursor: help;}
```

Иногда help применяется, чтобы показать, что элемент может предоставлять дополнительную информацию, например, как в элементах acronym с атрибутами title. Во многих агентах пользователя размещение курсора над акронимом с заголовком приводит к тому, что агент пользователя покажет содержимое атрибута title во всплывающей подсказке. Однако если курсор не меняется, пользователи, которые быстро перемещают курсор или работают на медленных компьютерах, могут не понять, что в данном месте есть дополнительная информация. Для таких случаев могло бы быть полезным следующее правило, применение которого приведет к результату, показанному на рис. 13.8:

```
acronym[title] {cursor: help; border-bottom: 1px dotted gray;}
```

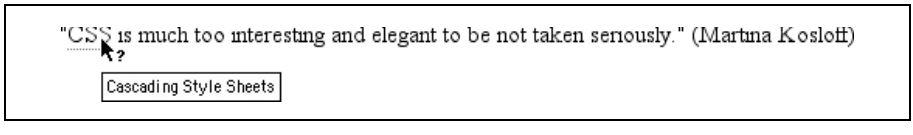


***Рис. 13.8.** Демонстрация доступности справки (в форме дополнительной информации)*

Графические курсоры

И последнее, но наиболее занимательное: возможность вызывать специальные курсоры. Это делается с помощью значения URL:

```
a.external {cursor: url(globe.cur), pointer;}
```

По этому правилу агент пользователя загружает файл *globe.cur* и берет из него значок курсора (рис. 13.9).

Конечно, агент пользователя должен поддерживать формат файла *globe.cur*, хранящего курсор. Если формат не поддерживается, агент пользователя вернется к значению pointer. Обратите внимание, что в описании синтаксиса свойства cursor следом за любым URL должны идти запятая и одно из базовых ключевых слов. В этом и состоит отличие от свойства font-family, позволяющего обращаться к определенному семейству и не предоставлять никаких резервных вариантов. Свой-

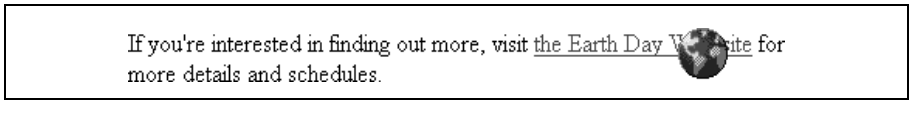


***Рис. 13.9.** Применение специального графического курсора*

ство `cursor` требует запасных вариантов для любых графических курсоров, которые вы могли бы выбрать.

Можно даже указать перед резервным ключевым словом несколько файлов курсоров. Например, можно создать один и тот же базовый курсор в нескольких форматах и все их включить в правило в надежде, что агент пользователя будет поддерживать по крайней мере один из них:

```
a.external {cursor: url(globe.svg#globe), url(globe.cur), url(globe.png),
  url(globe.gif), url(globe.xbm), pointer;}
```

Агент пользователя будет перебирать различные URL до тех пор, пока не найдет файл, из которого сможет взять значок курсора. Если агенту пользователя не удается найти формат, который он поддерживает, он перейдет к заданному ключевому слову.

Фактически можно реализовать и анимированные курсоры, если агент пользователя поддерживает файлы анимированной графики в качестве замены курсора. Например, IE6 поддерживает эту возможность через файлы *.ani*.

Контуры

CSS2 вводит последний основной элемент стилевого оформления пользовательского интерфейса: контуры. Контур немного похож на рамку, но между ними существует два главных отличия. Во-первых, контуры не участвуют в потоке документа, в отличие от рамок, и поэтому при своем появлении или исчезновении не приводят к повторному форматированию. Если задать для элемента 50-пиксельный контур, то он, скорее всего, будет перекрывать другие элементы. Во-вторых, контуры могут быть непрямоугольными, но не спешите радоваться. Это не значит, что вы можете создавать круглые контуры. Напротив, это означает, что контур строкового элемента может вести себя не так, как вела бы рамка этого элемента. Пользователь может «объединять» части контуров, создавая единую непрерывную, но не прямоугольную форму. На рис. 13.10 показан пример.

От агентов пользователя не требуется поддерживать непрямоугольные контуры. Вместо этого они могли бы форматировать контуры строковых незамещаемых элементов так же, как делают это с рамками. Однако поддерживающий данную спецификацию агент пользователя должен гарантировать, что контуры не занимают пространства компоновки.

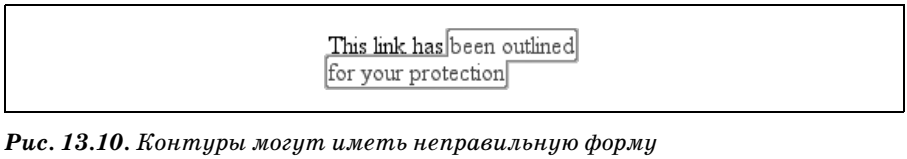


Рис. 13.10. *Контуры могут иметь неправильную форму*

Есть еще одно основное отличие контуров от рамок: это не одно и то же, поэтому они могут сосуществовать в одном элементе. Это может приводить к некоторым интересным эффектам, например к таким, который показан на рис. 13.11.

Спецификация CSS2 утверждает следующее: «Контур может отрисовываться, начиная непосредственно от внешней стороны края рамки». Обратите внимание на слово *может* в этом предложении. Агентам пользователя рекомендуется поступать так, как предлагает спецификация, но это требование может не выполняться. Агент пользователя мог бы принять решение отрисовывать рамки внутри внутреннего края рамки или на некотором небольшом расстоянии от рамки. На момент написания данной книги все броузеры, которые поддерживают контуры, отрисовывают их непосредственно по внешнему краю рамки, так что, слава богу, здесь все согласованно.

Контуры считаются частью стилевого оформления пользовательского интерфейса, потому что они чаще всего применяются для обозначения текущего фокуса ввода. Если пользователь для перехода от ссылки к ссылке применяет клавиатуру, то ссылка, которой в данный момент принадлежит фокус ввода, как правило, очерчена контуром. В Internet Explorer для Windows контур применяется к любой выбранной пользователем ссылке («щелкнутой», если используется мышь) и имеет свойство сохраняться, даже если в этом нет необходимости. Другие броузеры применяют контуры к элементам текстового ввода, которым принадлежит фокус клавиатуры, показывая таким образом, куда будут вставляться набираемые на клавиатуре символы.

Как вы увидите, стилевое оформление контуров во многом похоже на оформление рамок, но кроме только что упомянутых между ними имеются еще некоторые ключевые отличия. Мы лишь вкратце остановимся на подобиях и внимательно рассмотрим различия.

Рис. 13.11. *Сосуществование рамок и контуров*

Задание стиля контура

Основной характеристикой контура, как и рамки, является его стиль, который задается с помощью свойства `outline-style`.

Список ключевых слов задания стиля преимущественно аналогичен ключевым словам стилей рамки, и обеспечиваемые ими визуальные

outline-style	
Значения:	none | dotted | dashed | solid | double | groove | ridge | inset | outset | inherit
Начальное значение:	none
Область применения:	все элементы
Наследование:	нет
Вычисляемое значение:	как задано

эффекты тоже совпадают. Но есть одно исключение: hidden не является действительным стилем контура, и от агентов пользователя требуется интерпретировать его как none. Это на самом деле имеет смысл, – ведь контуры не оказывают влияния на компоновку, даже когда они видны.

Другое отличие между контурами и рамками состоит в том, что для значения outline-style можно задавать только одно ключевое слово (сравните: для рамок может быть задано до четырех ключевых слов). Практический эффект в том, что стиль контуров должен быть одинаковым со всех сторон элемента независимо от того, является контур прямоугольным или нет. Вероятно, это к лучшему, ведь могло оказаться, что довольно сложно определить, как применять различные стили к одному непрямоугольному контуру.

Ширина контура

Создав контур и задав его стиль, неплохо при помощи свойства outline-width определить (вы догадались) ширину контура.

outline-width	
Значения:	thin | medium | thick | <длина> | inherit
Начальное значение:	medium
Область применения:	все элементы
Наследование:	нет
Вычисляемое значение:	абсолютная длина; 0, если стиль рамки – none или hidden

Список ключевых слов должен быть хорошо знаком тем, кто задавал ширину рамок. Единственное настоящее отличие между outline-width и border-width в том, что, как и в случае со стилями, можно объявить только одну ширину для всего контура. Таким образом, в качестве значения допускается применять только одно ключевое слово.

Окрашивание контура

Поскольку можно задавать стиль и ширину, есть смысл в существовании свойства `outline-color`, позволяющего задавать цвет контура.

outline-color

Значения:	`<цвет> | invert | inherit`
Начальное значение:	`invert` (или в зависимости от агента пользователя; смотрите текст)
Область применения:	все элементы
Наследование:	нет
Вычисляемое значение:	как задано

Здесь и проявляется наиболее занятное различие между рамками и контурами: по умолчанию применяется ключевое слово `invert`. Инверсный контур означает, что для пикселов, на которых существует контур, осуществляется инверсия цветов (рис. 13.12).

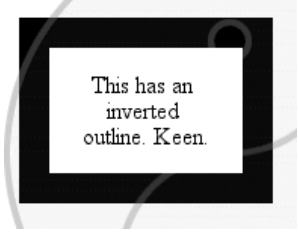


***Рис. 13.12.** Инверсия цветов в контуре*

Инвертирование цвета пикселов, закрываемых контуром, гарантирует, что контур останется видимым независимо от того, что происходит под ним. Если агент пользователя по какой-то причине не поддерживает инверсию цветов, он должен вычислять значение свойства `color` элемента.

Возможность инвертировать цвета пикселов экрана очень интересна, особенно если вспомнить, что теоретически ширина контура не ограничена. Поэтому с помощью контура при желании можно инвертировать значительную часть документа. Это не основное предназначение контуров, но все-таки на рис. 13.13 приведен один подобный пример.

Для того чтобы задать конкретный цвет контура, достаточно указать любое действительное значение цвета. Результаты применения следующих определений должны быть довольно очевидными:

```
outline-color: red;
outline-color: #000;
outline-color: rgb(50%, 50%, 50%);
```

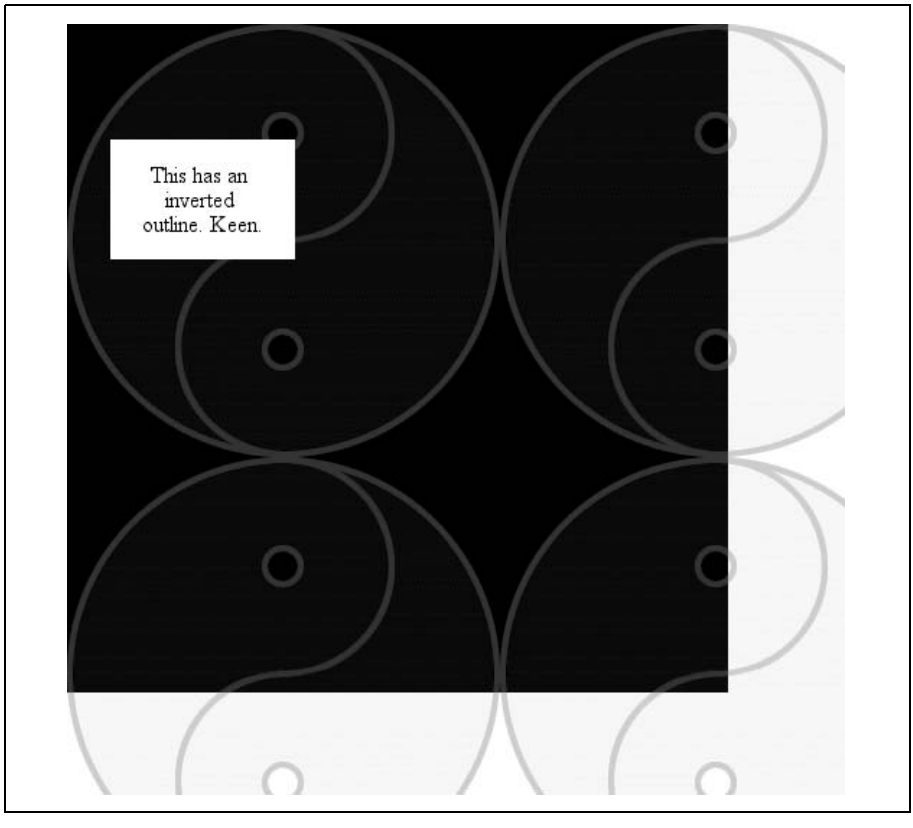


Рис. 13.13. *Обширная инверсия*

Потенциальный недостаток – вероятность того, что цвет контура может совпасть с цветом окружающих его пикселов, в этом случае пользователь не сможет различить его. Вот почему было определено значение `invert`.

Как и для стилей и значений ширины контуров, для всего контура можно определить только один цвет.

Сведение воедино

Свойство `outline`, как и свойство `border` для рамок, есть сокращенная форма для одновременного задания стиля, ширины и цвета контура.

Как и другие варианты сокращенной записи, `outline` сводит воедино несколько свойств. Для него характерна такая же схема поведения, как и для остальных свойств, допускающих сокращенную форму записи и переопределяющих ранее заданные значения. Таким образом, в следующем примере контур будет использовать ключевое слово `invert`, поскольку оно вводится вторым объявлением:

outline	
Значения:	[<outline-color> || <outline-style> || <outline-width>] | inherit
Начальное значение:	для сокращенной формы записи не определено
Область применения:	все элементы
Наследование:	нет
Вычисляемое значение:	см. отдельные свойства (outline-color и т. д.)

```
a:focus {outline-color: red; outline: thick solid;}
```

Поскольку контур должен иметь единый стиль, ширину и цвет, то нет сокращенной формы задания контуров, кроме `outline`. Такие свойства, как `outline-top` или `outline-right`, не существуют.

Для того чтобы имитировать инверсную рамку, можно задать ширину контура в единицах длины и создать поля элемента такой же или большей ширины. Поскольку контур отрисовывается поверх поля, он заполнит часть его пространства, как показано на рис. 13.14:

```
div#callbox {outline: 5px solid invert; margin: 5px;}
input:focus {outline: 1em double gray;}
```

Как я уже говорил, контуры не принимают участия в потоке документа. Это позволяет избежать принудительного переформатирования в таких случаях, как отрисовка контуров у получивших фокус ссылок, когда контур перемещается от ссылки к ссылке по мере перемещения фокуса. Если для обозначения фокуса автор применяет рамки, макет документа может смещаться. Контуры могут обеспечить эффект, аналогичный применению рамок, но без резких скачков.

Контуры справляются с этой задачей, потому что, по определению, отрисовываются поверх всего остального блока элемента. Поскольку в CSS2 контуры не могут перекрывать видимые части блока их элемента, а могут перекрывать только поля (которые являются прозрачными), это небольшая проблема. Если будущая версия CSS разрешит

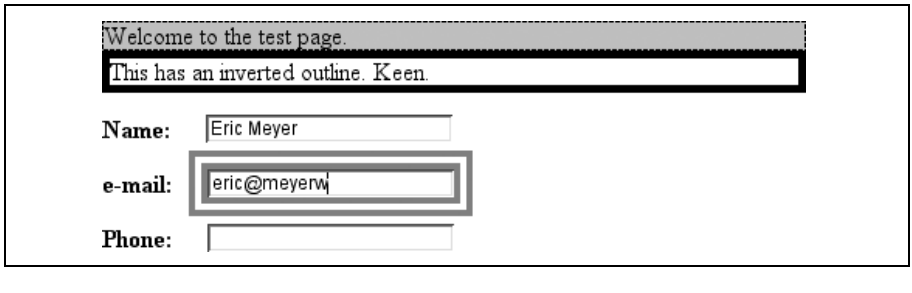


Рис. 13.14. *Разные контуры*

контурам смещаться и перекрывать рамки или другие видимые части блока элемента, вопрос их размещения приобретет большее значение.

Еще одна неясность CSS2 состоит в том, что он явно уклоняется от определения поведения двух контуров, перекрывающих друг друга, и того, что происходит с контурами элементов, частично перекрытых другими элементами. Все это можно свести воедино в следующем примере:

```
div#one {outline: 1em solid invert;}
div#two {outline: 1em solid invert; margin: -2em -2em 0 2em;
  background: white;}
```

Теперь предположим, что `div#two` следует в документе непосредственно за `div#one`. Он перекроет первый `div`, а его фон перекроет части контура первого `div`. Я не привел рисунка к этому примеру, потому что в спецификации CSS2 ничего не говорится о том, что могло бы произойти в этом случае. Должен ли быть видимым контур первого `div`, перекрывающий фон и содержимое второго `div`? Также должны образоваться области, в которых пересекаются два инверсных контура; что должно произойти там? Происходит ли двойная инверсия пикселов и таким образом восстановление исходного их состояния? И должны ли быть пикселы инвертированы один раз? Мы не знаем. Ни одна из иллюстраций не была бы ни верной, ни неверной, это был бы просто возможный результат – и не обязательно, что это был бы именно тот результат, на котором остановятся агенты пользователя или который определит будущая версия CSS.

Заключение

Благодаря стилям пользовательского интерфейса автор получает возможность делать свой документ похожим на интерфейс операционной системы, особенно если творчески подойдет к применению системных цветов и шрифтов. Документ, оформленный в стиле, хорошо знакомом пользователю, с самого начала будет иметь более привычный и удобный для использования вид.

Еще один способ немного облегчить жизнь пользователя – создать таблицы стилей, ориентированные не на мониторы, а на другие устройства. Сюда вошли бы стили, предназначенные специально для печати, звукового (речевого) доступа к веб-странице и даже для проецирования на экран. Все это мы рассмотрим в следующей главе.

14

Неэкранные устройства

Не все обитатели Всемирной паутины могут видеть эффекты, которые мы обсудили в этой книге. В США примерно 1,1 миллиона людей с ослабленным зрением, и очевидно, что они работают с Интернетом совсем не так, как те, кто видит хорошо.

К счастью, CSS не безмолвствуют по вопросу невизуального доступа. В CSS2 добавлена возможность применять стили в неэкранных устройствах. Тогда как Всемирная паутина эволюционировала, преимущественно ориентируясь на мониторы, т. е. на визуальную среду, CSS2 может применяться в невизуальных устройствах, если соответствующую поддержку обеспечивает агент пользователя.

Нельзя отказываться от преимуществ создания документов, с которыми можно работать как визуально, так и невизуально. Вы убережете себя от массы проблем, имея возможность взять один и тот же документ и, применив к нему разные, зависящие от среды отображения таблицы стилей, переформатировать его для воспроизведения на экранных, печатных и звуковых устройствах. Например, не понадобилось бы добавлять на страницу ссылку на ее версию для печати. Вместо создания совершенно различных структур разметки – одной для экрана, другой для печати – можно повысить эффективность сайта, многократно используя один и тот же документ.

Впрочем, можно взять отдельный HTML-документ, содержащий схему слайдовой презентации, и оформить его так, чтобы его было легко читать с экрана, делать красивые и легко читаемые распечатки в формате покадрового вывода и чтобы при этом документ мог воспроизводиться программой генерации речи. В этой главе мы рассмотрим, как реализовать последние три задачи (поскольку вся остальная книга посвящена экранным представлениям).

Разработка зависящих от среды таблиц стилей

Механизмы, определенные в HTML и CSS, позволяют ограничить область действия любой таблицы стилей конкретным устройством. В таблицах стилей, предназначенных для HTML, такие ограничения могут быть наложены посредством атрибута `media`. Его можно применять как для элемента `link`, так и для элемента `style`:

```
<link rel="stylesheet" type="text/css" media="print"
   href="article-print.css">

<style type="text/css" media="projection">
body {font-family: sans-serif;}
</style>
```

Атрибут `media` может принимать одно значение или список значений, разделенных запятыми. Таким образом, чтобы подключить таблицу стилей, которая должна применяться только в экранных и проекционных устройствах, напишите:

```
<link rel="stylesheet" type="text/css" media="screen, projection"
   href="visual.css">
```

В самой таблице стилей можно также наложить ограничения на правила `@import`:

```
@import url(visual.css) screen, projection;
@import url (article-print.css) print;
```

Помните, что если не добавить в таблицу стилей информацию об устройстве, она будет применяться ко *всем* устройствам. Следовательно, если один набор стилей должен применяться только при выводе на экран, а другие – только при печати, то в обе таблицы стилей необходимо добавить информацию об устройстве. Например:

```
<link rel="stylesheet" type="text/css" media="screen"
   href="article-screen.css">
<link rel="stylesheet" type="text/css" media="print"
   href="article-print.css">
```

Если удалить атрибут `media` из первого элемента `link` предыдущего примера, то правила таблицы стилей *article-screen.css* будут применяться во всех устройствах, включая печатные, проекционные, переносные и все остальные.

CSS2 также определяет синтаксис для блоков `@media`, которые позволяют определять стили для нескольких устройств в одной таблице стилей. Рассмотрим следующий стандартный пример:

```
<style type="text/css">
body {background: white; color: black;}
@media screen {
   body {font-family: sans-serif;}
```

```
    h1 {margin-top: 1em;}
}
@media print {
    body {font-family: serif;}
    h1 {margin-top: 2em; border-bottom: 1px solid silver;}
}
</style>
```

Здесь видно, что во всех устройствах у элементов `body` будет белый фон, а их основной цвет будет черным. Затем отдельно представлен блок правил для устройства `screen`, за которым следует другой блок правил, применяемых только в устройстве `print`.

Блоки `@media` могут иметь любой размер и содержать любое количество правил. В тех ситуациях, когда авторы могут управлять только одной таблицей стилей, блоки `@media` дают, возможно, единственный способ задания зависящих от среды таблиц стилей. Это также верно, когда CSS применяется для стилевого оформления документа, основанного на XML, не содержащего атрибут `media` или его эквивалент.

Устройства с постраничной разбивкой

С точки зрения CSS *устройство с постраничной разбивкой* (*paged medium*) – это любое устройство, в котором представление документа обрабатывается как ряд отдельных «страниц». На экране, который является устройством *без разбивки* (*continuous medium*), документы представляются как единая прокручиваемая «страница». Пример устройства без разбивки – свиток папируса. Печатные материалы, такие как книги, журналы и распечатки на лазерном принтере, – все это носители с постраничной разбивкой. К ним относятся и слайдовые презентации, в которых слайды демонстрируются поочередно. С точки зрения CSS каждый слайд – это «страница».

Стили для вывода на печать

Даже в «безбумажном будущем» наиболее распространенным вариантом вывода с постраничной разбивкой будет распечатка некоторого документа – веб-страницы, документа с форматированным текстом, электронной таблицы или чего-то еще, что фиксируется на листках бумаги. У авторов много возможностей сделать распечатки своих документов более привлекательными для пользователя: от впечатляющей разбивки на страницы до создания стилей, специально предназначенных для печати.

Обратите внимание, что стили для вывода на печать также могли бы применяться в случае представления документа в режиме «предварительного просмотра перед печатью». Таким образом, в ряде обстоятельств стили для вывода на печать можно увидеть и на устройстве отображения.

Отличия между экранным и печатным представлениями

Кроме очевидной физической разницы, между экранным и печатным представлениями существует ряд стилистических отличий. Самое основное – выбор шрифтов. Большинство разработчиков скажут, что рубленые шрифты лучше всего подходят для экранного представления, а вот шрифты антиква лучше читаются при печати. Таким образом, можно создать таблицу стилей для печати, выводящую текст документа шрифтом Times вместо Verdana.

Следующее отличие заключается в задании размеров шрифтов. Тот, кто хоть немного занимался веб-разработкой, вероятно, неоднократно слышал, что пункты не подходят для задания размеров шрифтов в Интернете. В основном это верно, особенно если вы хотите, чтобы в различных броузерах и операционных системах размеры текста были одинаковыми. Однако печатное представление – это не веб-представление, а веб-представление отличается от печатного. В печатном представлении применение пунктов или даже сантиметров или пик вполне допустимо, потому что печатные устройства знают физический размер своей области вывода. Принтер знает, что размер зоны печати соответствует размерам листа бумаги (например, 8.5×11 дюймов). Он также знает, сколько точек в дюйме, поскольку осведомлен о том, сколько точек на дюйм (dpi) может сгенерировать. Это значит, что принтер может оперировать такими физическими единицами измерения длины, как пункты.

Поэтому многие таблицы стилей для печати начинаются примерно с такого объявления:

```
body {font: 12pt "Times New Roman", "TimesNR", Times, serif;}
```

Это настолько традиционно, что у графического дизайнера, читающего у вас через плечо, на глаза навернулись бы слезы умиления. Но убедитесь, что он понимает: пункты приемлемы только из-за природы печатного устройства, для веб-представлений они по-прежнему не годятся.

Напротив, отсутствие фона в большинстве распечаток, должно быть, вызывает у этого дизайнера слезы отчаяния. Для сохранения чернил пользователя большинство веб-броузеров заранее сконфигурированы так, чтобы не выводить на печать фоновые цвета и изображения. Если пользователь хочет видеть эту информацию на распечатке, он должен изменить параметры в настройках броузера. CSS никак не может воздействовать на распечатку фона. Однако для вывода на печать можно использовать таблицу стилей и сделать фон ненужным. Например, включить в таблицу стилей для вывода на печать следующее правило:

```
* {color: black !important; background: white !important;}
```

В результате все элементы будут выводиться на печать как черный текст и любой фон, если его включили в универсальную таблицу стилей, будет удален. Поскольку в любом случае большинство принтеров

пользователей будут именно так генерировать визуальное представление страницы, лучше не задавать настройки стилей для вывода на печать в тех же строках. Это также гарантирует, что если в веб-документе желтый текст располагается на темно-сером фоне, пользователь с цветным принтером не получит распечатку, в которой желтый текст будет располагаться на белом листке бумаги.

CSS2.x не содержит механизма выбора таблицы стилей в зависимости от устройства вывода пользователя. Таким образом, все принтеры будут использовать определенные автором таблицы стилей для вывода на печать. Модуль CSS3 «Media Queries» определяет способы выбора разных таблиц стилей для цветных и черно-белых принтеров, но на момент написания данной книги запросы типа устройства не поддерживаются.

Еще одно отличие между устройствами с постраничной разбивкой и без нее – в первых большие трудности связаны с многоколонной верстской. Предположим, имеется статья, в которой текст отформатирован в две колонки. При распечатке первая колонка будет располагаться на каждой странице слева, а вторая колонка – справа. Это вынудит пользователя прочитывать левую часть каждой страницы, а затем возвращаться к началу распечатки и прочитывать правую часть страницы. Это сильно раздражает в экранных представлениях веб-документов, а на бумажном носителе и подавно.

Очевидное решение состоит в том, чтобы использовать CSS для верстки двух колонок (может быть, делая их перемещаемыми) и затем создавать для вывода на печать таблицу стилей, которая помещает содержимое в одну колонку. Таким образом, в таблице стилей для вывода на экран вы могли бы написать нечто подобное:

```
div#leftcol {float: left; width: 45%;}
div#rightcol {float: right; width: 45%;}
```

А это – в таблице стилей для вывода на печать:

```
div#leftcol, div#rightcol {float: none; width: auto;}
```

Если бы в CSS была возможность создавать макеты, состоящие из нескольких колонок, ничего подобного не понадобилось бы. К сожалению, на момент написания данной книги ничего в этом направлении не предложено, хотя попыток за долгие годы было предпринято немало.

Мы могли бы посвятить целую главу подробностям печатного представления, но основная цель этой книги другая. Давайте обратимся к деталям CSS, связанным с устройствами с постраничной разбивкой, и оставим обсуждение дизайна для другой книги.

Определение размера страницы

Практически так же, как он определяет блок элемента, CSS2 определяет и *блок страницы* (*page box*), который описывает компоненты страницы. Блок страницы образован, по существу, двумя областями:

- *Областью страницы* (*page area*), которая является той частью страницы, где располагается содержимое. Это аналог области содержимого обычного блока элемента, если принять, что границы области страницы выступают в роли начального блока-контейнера макета страницы. (Подробно о блоках-контейнерах рассказано в главе 7.)
- *Областью полей* (*margin area*), которая окружает область страницы.

Модель блока страницы проиллюстрирована на рис. 14.1.

В CSS2 можно определять как размер блока страницы, так и размер полей. В CSS2.1 авторы могут задавать только размер области полей. В обоих случаях эти параметры задаются с помощью правила `@page`. Простой пример:

```
@page {size: 7.5in 10in; margin: 0.5in;}
```

Это правило CSS2, поскольку оно использует свойство `size`, которое не было включено в CSS2.1 из-за отсутствия реализации.

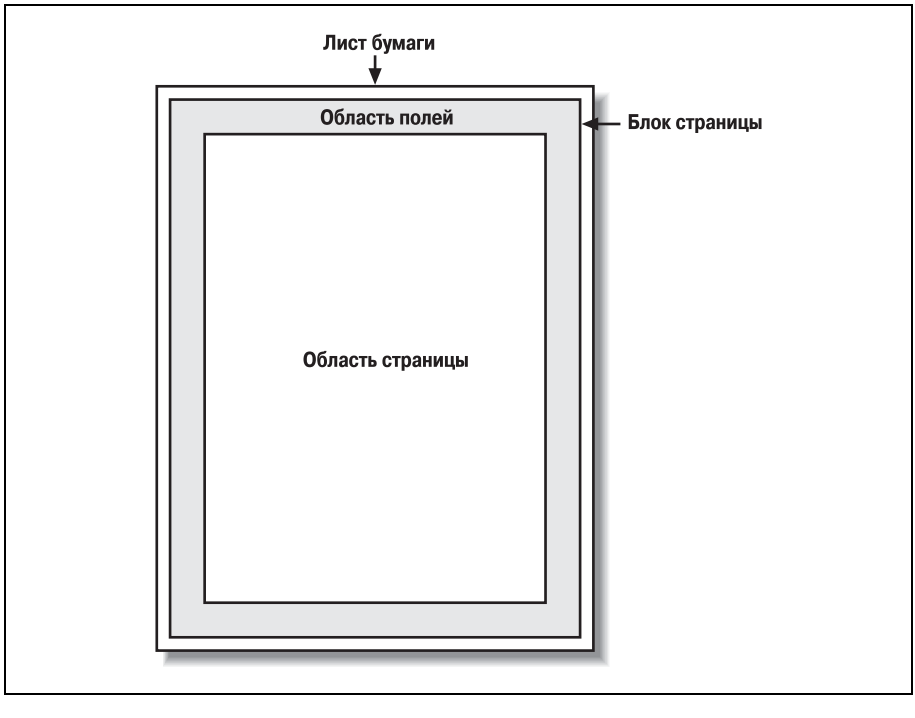


Рис. 14.1. *Блок страницы*

size	
Значения:	<длина>{1,2} | auto | portrait | landscape | inherit
Начальное значение:	auto
Область применения:	область страницы
Наследование:	нет

Это свойство применяется для определения размера области страницы. Значение landscape (альбомная) разворачивает макет на 90°, тогда как portrait (книжная) – это обычная ориентация для распечаток на западных языках. Таким образом, документ при распечатке можно повернуть на 90°, объявив:

```
@page {size: landscape;}
```

Свойство size не является частью CSS2.1. Это означает, что на момент написания книги неизвестно о существовании хотя бы двух реализаций size, способных взаимодействовать. Так что поддержка броузеров, скорее всего, слаба. CSS2.1 включает возможность стилевого оформления области полей блока страницы, что, вероятно, будет работать более надежно. Для того чтобы печать осуществлялась лишь на небольшом участке в центре страницы размером 8.5×11 дюймов, напишите:

```
@page {margin: 3.75in;}
```

В результате область печати будет иметь 1 дюйм в ширину и 3,5 дюйма в высоту.

Самое интересное в блоке страницы то, что поскольку он не имеет никакого отношения к шрифтам, то нельзя описывать ни поля, ни область страницы с помощью единиц длины em или ex. Допускаются только процентные значения и «линейные» единицы длины, такие как дюймы, сантиметры или пункты.

Выбор типов страниц

CSS2 предлагает возможность создания различных типов страниц с помощью именованных правил @page. Скажем, имеется документ по астрономии, состоящий из нескольких страниц и содержащий довольно широкую таблицу со списком физических характеристик всех лун Сатурна. Текст надо распечатать с книжной ориентацией, но таблица должна быть развернута горизонтально. Вот как можно было бы начать:

```
@page normal {size: portrait; margin: 1in;}
@page rotate {size: landscape; margin: 0.5in;}
```

Теперь надо лишь применить эти типы страниц соответствующим образом. Значение id таблицы лун Сатурна – moon-data, т. е. вы пишете следующие правила:

page	
Значения:	<идентификатор> | inherit
Начальное значение:	auto
Область применения:	элементы уровня блока
Наследование:	да

```
body {page: normal;}
table#moon-data {page: rotate;}
```

Согласно этим правилам таблица распечатывается горизонтально, а весь остальной документ – вертикально. Это возможно благодаря свойству page, еще одного изгнанника CSS2.1.

Как видно из описания значений, единственное оправдание существования свойства page в том, что оно обеспечивает авторам возможность создавать именованные типы страниц для различных элементов документа.

Существует ряд базовых типов страниц, к которым можно обращаться посредством специальных псевдоклассов, и, что радует, эта возможность определена и в CSS2, и в CSS2.1. Псевдокласс :first позволяет применять специальные стили к первой странице документа. Пусть надо задать для первой страницы поле сверху большее, чем у других страниц. Вот как это делается:

```
@page {margin: 3cm;}
@page :first {margin-top: 6cm;}
```

В результате на всех страницах, за исключением первой, для которой поле сверху составит шесть сантиметров, будет создано поле высотой три сантиметра (примерно как на рис. 14.2).

Кроме оформления первой страницы, можно также задать размещение страниц слева или справа, имитируя разворот книги. Применяя псевдоклассы :left и :right, можно и их оформить по-разному. Например:

```
@page :left {margin-left: 3cm; margin-right: 5cm;}
@page :right {margin-left: 5cm; margin-right: 3cm;}
```

Эффект от применения этих правил будет таким же, как от задания больших полей «между» содержимым левой и правой страниц – там, где должен был бы располагаться корешок книги. Так всегда поступают, объединяя страницы в книгу. Результат применения предыдущих правил можно увидеть на рис. 14.3.

Разрывы страниц

В устройствах с постраничной разбивкой неплохо было бы уметь влиять на размещение разрывов страниц. Управлять разрывами страниц

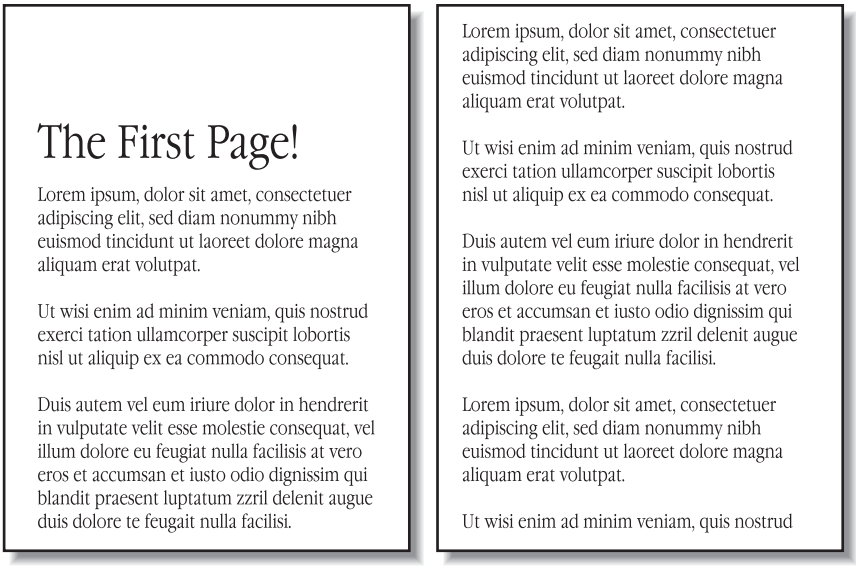


Рис. 14.2. *Специальное оформление первой страницы*

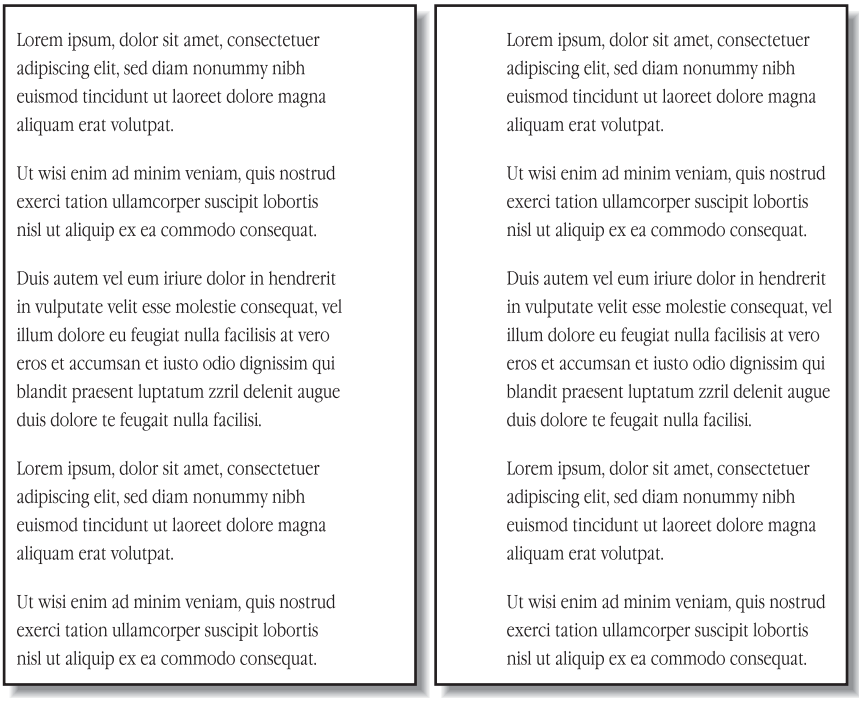


Рис. 14.3. *Различное стилевое оформление левой и правой страниц*

page-break-before, page-break-after	
Значения:	auto | always | avoid | left | right | inherit
Начальное значение:	auto
Область применения:	неперемещаемые блочные элементы, значение свойства position которых – relative или static
Наследование:	нет
Вычисляемое значение:	как задано

можно спомощью свойств page-break-before и page-break-after, которые имеют один и тот же набор значений.

Если задать значение по умолчанию, auto, то перед элементом или после него не будет обязательного разрыва страницы. Совсем как при обычном выводе на печать. Значение always размещает разрыв страницы перед (или за) оформляемым элементом.

Предположим, ситуация такова, что заголовок документа – это элемент h1, а заголовками разделов являются элементы h2. Требуется поместить разрыв страницы прямо перед началом каждого раздела документа и после заголовка документа. Для этого понадобились бы такие правила, результат применения которых проиллюстрирован на рис. 14.4:

```
h1 {page-break-after: always;}
h2 {page-break-before: always;}
```

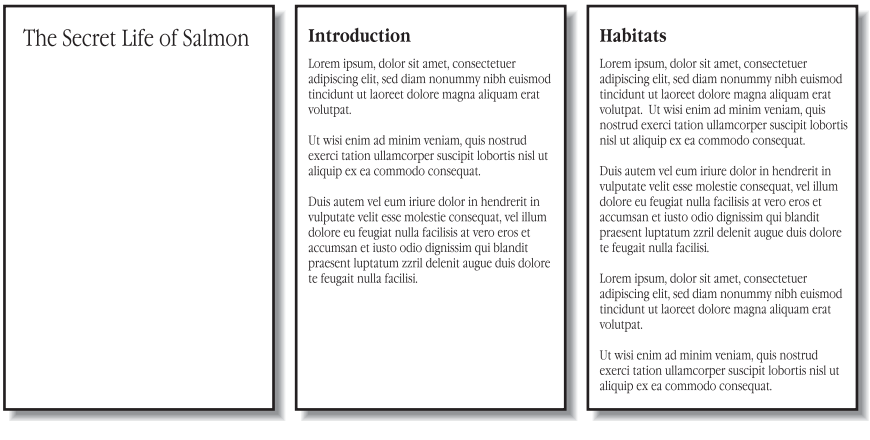


***Рис. 14.4.** Вставка разрывов страниц*

Если заголовок документа должен быть центрирован, то, конечно, надо добавить соответствующие правила. Но их здесь нет, и поэтому получается довольно простое визуальное представление каждой страницы.

Принцип работы значений left и right такой же, как и у always, за исключением того, что они более подробно определяют тип страницы, на которой может быть продолжен вывод на печать:

```
h2 {page-break-before: left;}
```

В соответствии с этим правилом каждому элементу h2 будет предшествовать достаточное количество разрывов страниц, чтобы распечатка h2 начиналась сверху левой страницы, т. е. на странице, которая располагалась бы слева от корешка, если бы вывод был предназначен под переплет. При двусторонней распечатке это означало бы вывод на оборотной стороне листа бумаги.

Итак, предположим, что при выводе на печать элемент, располагающийся прямо перед h2, печатается на правой странице. Предыдущее правило обусловило бы вставку перед h2 одного разрыва страницы и таким образом смещение его на следующую страницу. Однако если следующему h2 предшествует элемент, расположенный на левой странице, перед ним вставляются два разрыва страницы, чтобы он был бы размещен на следующей левой странице. Правая страница, расположенная между этими двумя, была бы оставлена пустой намеренно. Применение значения right имеет такой же эффект, только элемент печатается, начиная с верха правой страницы, и предваряется одним или двумя разрывами страниц.

Значение avoid подобно always, оно приказывает агенту пользователя сделать все возможное, чтобы избежать размещения разрыва страницы как перед, так и после элемента. Чтобы расширить предыдущий пример, предположим, что заголовками некоторых являются элементы h3. Эти заголовки нельзя отрывать от текста, следующего за ними, поэтому необходимо избежать разрыва страницы после h3 везде, где возможно его появление:

```
h3 {page-break-after: avoid;}
```

Однако заметьте, что значение названо avoid, а не never. Абсолютно гарантировать, что разрыв страницы не будет вставлен перед данным элементом или после него, нельзя. Например:

```
img {height: 9.5in; width: 8in; page-break-before: avoid;}
h4 {page-break-after: avoid;}
h4 + img {height: 10.5in;}
```

Далее предположим, что h4 располагается между двумя изображениями и его вычисляемая высота составляет полдюйма. Каждое изображение придется распечатывать на отдельной странице, но h4 может располагаться только в двух местах: внизу страницы с первым элементом или на странице после него. Если он помещается после первого изображения, за ним должен следовать разрыв страницы, поскольку для второго изображения места на этой странице не остается, как показано на рис. 14.5.

С другой стороны, если h4 размещается на новой странице, следующей за первым изображением, здесь не остается места для второго изобра-

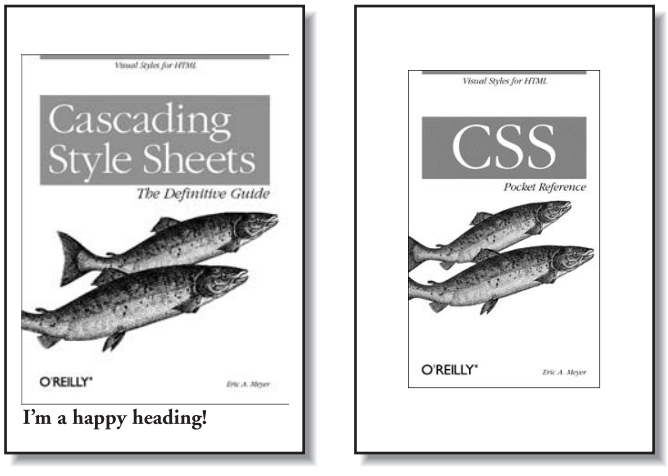


Рис. 14.5. *Неизбежный разрыв страницы*

жения. Итак, после `h4` возникнет разрыв страницы. И в любом случае минимум одному изображению, если не обоим, будет предшествовать разрыв страницы. Это все, что может сделать агент пользователя исходя из сложившей ситуации.

Очевидно, что подобные ситуации редки, но они возможны, например, в том случае, когда документ содержит только таблицы с заголовками. Возможны случаи, когда таблицы распечатываются таким образом, что обусловливают вынужденное размещение разрыва страницы после элемента заголовка, даже несмотря на то, что автор требовал избегать такого размещения разрывов.

Аналогичные проблемы могут возникать в случае применения другого свойства для определения разрыва страниц – `page-break-inside`. Набор его возможных значений более ограничен, чем у родственных ему свойств.

Для свойства `page-break-inside` у вас практически всего один вариант значения, кроме применяемого по умолчанию: можно потребовать, чтобы агент пользователя пытался избегать (`avoid`) размещения разрывов страниц в элементе. Если некую последовательность сносок нельзя разрывать на две страницы, то объявление может быть таким:

page-break-inside

Значения:	`auto | avoid | inherit`
Начальное значение:	`auto`
Область применения:	неперемещаемые блочные элементы, значение свойства `position` которых – `relative` или `static`
Наследование:	да
Вычисляемое значение:	как задано

```
div.aside {page-break-inside: avoid;}
```

И это тоже скорее рекомендация, чем абсолютное правило. Если сноска оказывается длиннее страницы, то очевидно, что агент пользователя не сможет ничем помочь и вставит разрыв страницы в элемент.

Сироты и вдовы

Для того чтобы обеспечить более тонкое влияние на разбиение страниц, CSS2 определяет два свойства, применяемые как в традиционном типографском деле, так и в настольных издательских системах: `widows` и `orphans`.

widows, orphans

Значения:	<целое> | inherit
Начальное значение:	2
Область применения:	элементы уровня блока
Наследование:	да
Вычисляемое значение:	как задано

Цель у этих свойств одна, но подход к ней – с разных точек зрения. Значение `widows` определяет минимальное количество строк элемента, которые могут быть перенесены на следующую страницу без инициирования разрыва страницы перед элементом. Эффект применения `orphans` аналогичен, но с противоположной стороны: он задает минимальное количество строк элемента, которые могут остаться в нижней части страницы, не вызывая разрыва страницы перед элементом.

Возьмем в качестве примера свойство `widows`. Предположим, вы объявляете:

```
p {widows: 4;}
```

Это значит, что вверху страницы может находиться не менее четырех строк любого абзаца. Если на следующей странице оказывается меньшее количество строк, то на новую страницу переносится весь абзац. Рассмотрим ситуацию, представленную на рис. 14.6. Прикройте ладонью верхнюю часть рисунка, чтобы было видно только вторую страницу. Обратите внимание, что на ней находятся две строки, относящиеся к абзацу, начавшемуся на предыдущей странице. Исходя из применяемого по умолчанию значения `widows`, равного `2`, такое визуальное представление допустимо. Однако если бы это значение было равно `3` или более, весь абзац перешел бы на вторую страницу одним блоком. Это привело бы к вставке разрыва страницы перед рассматриваемым абзацем.

Вернемся к рис. 14.6 и на этот раз прикроем рукой вторую страницу. Обратите внимание на четыре строки внизу страницы, относящиеся к последнему абзацу. Все в порядке, пока `orphans` имеет значение `4` или

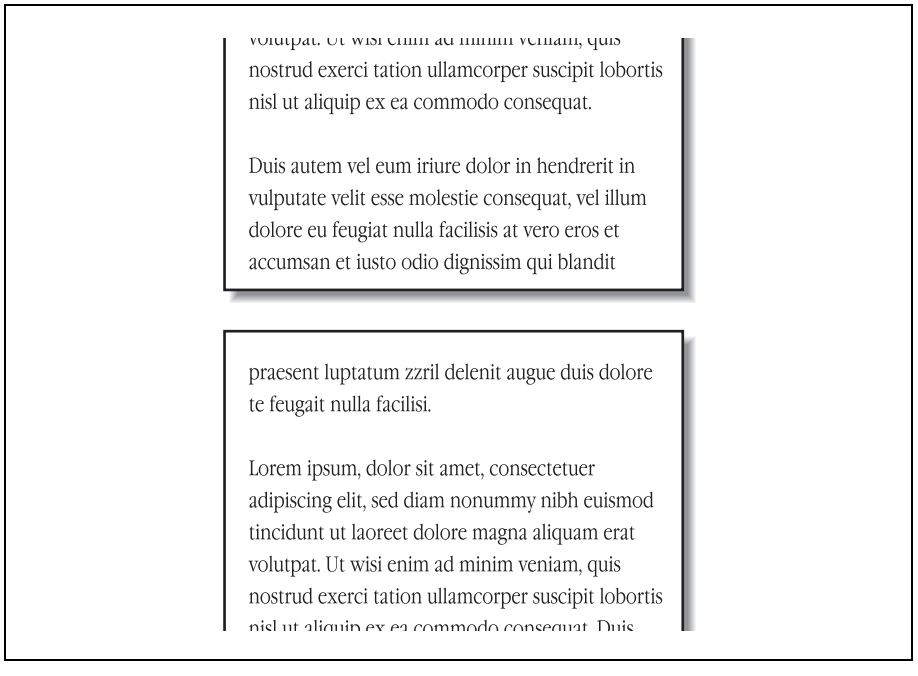


***Рис. 14.6.** Подсчет изолированных строк*

меньше. Если бы значение было равно `5` или более, то перед абзацем опять был бы вставлен разрыв страницы, а сам абзац был бы помещен как единый блок в верхней части второй страницы.

Конечно, при верстке должно учитываться как значение `orphans`, так и значение `widows`. Если бы автор объявил следующее, большинство абзацев не разрывались бы:

```
p {widows: 30; orphans: 30;}
```

В соответствии с этими значениями, абзац должен быть очень длинным, чтобы в него можно было вставить разрыв строки. Конечно, если необходимо избежать разрывов посередине параграфов, это лучше было бы выразить следующим образом:

```
p {page-break-inside: avoid;}
```

Разбивка на страницы

В CSS2 предусмотрены некоторые дополнительные стили разрывов страниц, т. е. он определяет набор схем поведения, допустимых и «наилучших» для организации разрывов страниц. Эти схемы определяют, как агенты пользователя должны обрабатывать разрывы страниц в различных обстоятельствах.

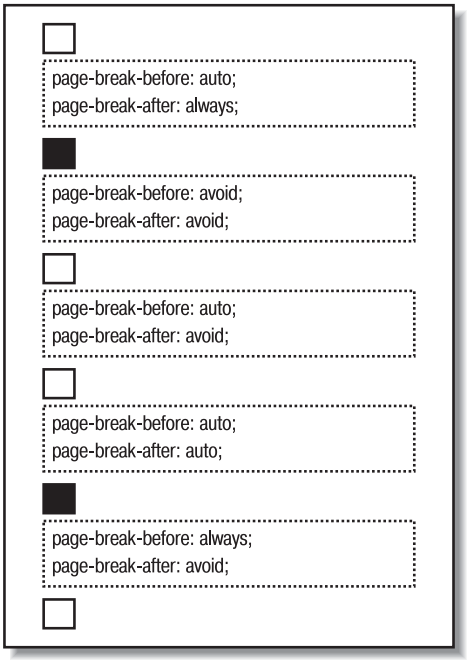


Рис. 14.7. *Размещение возможных разрывов страниц между блочными элементами*

На самом деле разрывы на страницах допускаются в основном в двух местах. Во-первых, между двумя блочными элементами. Если разрыв страницы оказывается между двумя блочными элементами, свойству `margin-bottom` элемента, находящегося перед разрывом страницы, присваивается значение `0`, как и свойству `margin-top` элемента, располагающегося после разрыва. Кроме того, есть два правила, позволяющие разрыву страницы располагаться между двумя блоками элементов:

- Если свойство `page-break-after` первого элемента или свойство `page-break-before` второго элемента имеет значения `always`, `left` или `right`. Это выполняется независимо от значения данного свойства для другого элемента, даже если это `avoid`. (Это *принудительный* (*forced*) разрыв.)
- Если свойство `page-break-after` первого элемента имеет значение `auto`, такое же значение имеет свойство `page-break-before` второго элемента, и у них нет общего элемента-предка, свойство `page-break-inside` которого имеет значение, отличное от `avoid`.

На рис. 14.7 показаны все возможные варианты размещения разрывов страниц между элементами гипотетического документа. Принудительные разрывы страниц представлены заштрихованными квадратами, а возможные (необязательные) разрывы – пустыми квадратами.

Во-вторых, допускаются разрывы страниц между двумя строками, расположенными внутри блочного элемента. Здесь также есть пара правил:

- Разрыв страницы может находиться между двумя строками блока, только если количество строк между началом элемента и строкой, предшествующей разрыву страницы, больше значения свойства `orphans` элемента. Аналогично разрыв страницы может быть помещен только там, где количество строк между строкой, расположенной после разрыва страницы, и концом элемента больше значения свойства `widows`.
- Разрыв страницы может быть помещен между строками, если значение свойства `page-break-inside` элемента отличается от `avoid`.

В обоих случаях второе из двух правил, управляющих размещением разрыва страницы, игнорируется, если нельзя вставить разрыв так, чтобы он удовлетворял всем правилам. Таким образом, если для элемента было задано `page-break-inside: avoid`, но он выходит за пределы страницы, разрыв разрешен внутри элемента, между двумя строками. Иначе говоря, второе правило размещения разрыва страницы между строками игнорируется.

Если даже игнорирование второго правила каждой пары правил не обеспечивает приемлемого размещения разрыва страницы, можно игнорировать и первое правило. В таких случаях агент пользователя, скорее всего, проигнорирует все значения свойств разрыва страниц и обработает их так, как будто все они имеют значение `auto`, хотя этот подход не определен (или не оговорен) спецификацией CSS.

Кроме правил, описанных выше, CSS2 определяет ряд «наилучших» схем поведения при разрыве страниц:

- Вставлять разрывы как можно реже.
- Делать высоту всех страниц, не оканчивающихся принудительным разрывом, примерно одинаковой.
- Избегать разрывов внутри блока, имеющего рамку.
- Избегать разрывов внутри таблицы.
- Избегать разрывов внутри перемещаемого элемента.

Эти рекомендации не подлежат обязательному выполнению агентами пользователя, но они предлагают логичное руководство, призванное обеспечить идеальные схемы поведения при размещении разрывов страниц.

Повторяющиеся элементы

В устройствах с постраничной разбивкой авторы очень часто применяют колонтитул. Это элемент, повторяющийся на каждой странице, например название документа или имя автора. В CSS2 такие элементы оформления можно реализовать при помощи элемента с фиксированным положением. Например:

```
div#runhead {position: fixed; top: 0; right: 0;}
```

В соответствии с этим правилом любой элемент `div`, атрибут `id` которого равен `runhead`, при выводе документа через устройство с постраничной разбивкой будет размещаться в верхнем правом углу каждого блока страницы. Это же правило разместило бы элемент в верхнем правом углу окна просмотра устройства без разбивки, такого как веб-броузер. Любой позиционированный таким образом элемент будет появляться на каждой странице. «Копировать» элемент, чтобы он стал повторяющимся, невозможно. Таким образом, в соответствии со следующим правилом элемент `h1` будет появляться как колонтитул на каждой странице, включая первую:

```
h1 {position: fixed; top: 0; width: 100%; text-align: center;
   font-size: 80%; border-bottom: 1px solid gray;}
```

Недостаток в том, что теперь элемент `h1`, размещенный на первой странице, может быть распечатан только как колонтитул.

Элементы, находящиеся вне страницы

Все эти разговоры о позиционировании элементов в устройствах с постраничной разбивкой приводят к интересному вопросу: что происходит, если элемент позиционирован вне блока страницы? Для создания такой ситуации нет необходимости применять позиционирование. Вспомните элемент `pre`, содержащий строку с 411 символами. Он наверняка будет шире любого стандартного листа бумаги и соответственно шире блока страницы. Что происходит в этом случае?

Оказывается, в CSS2 не оговорено, что именно должны делать агенты пользователя, поэтому поиск выхода из этой ситуации остается за ними. Для каждого широкого элемента `pre` агент пользователя мог бы просто отсечь элемент по границе блока страницы и отбросить остальную часть содержимого. Также можно было бы сгенерировать дополнительные страницы, чтобы представить не поместившуюся часть элемента.

Для обработки содержимого, оказавшегося вне блока страницы, можно привести несколько общих рекомендаций, две из которых действительно важны. Во-первых, допускается небольшой выход содержимого за края блока страницы, т. е. размещение содержимого без полей. Подразумевается, что для содержимого, выходящего за пределы блока страницы, дополнительная страница не генерируется.

Во-вторых, агентам пользователя рекомендуется не генерировать помногу пустых страниц только для того, чтобы выполнить требования позиционирования. Рассмотрим:

```
h1 {position: absolute; top: 1500in;}
```

Пусть высота блоков страниц составляет 10 дюймов, тогда агенту пользователя пришлось бы перед элементом `h1` размещать 150 разрывов страниц (т. е. 150 пустых страниц), только чтобы выполнить это

правило. Вместо этого агент пользователя может проигнорировать пустые страницы и вывести только последнюю, которая содержит элемент h1 фактически.

Есть еще две рекомендации: агенты пользователя не должны позиционировать элементы в необычных местах только для того, чтобы избежать генерирования их визуального представления; содержимое, размещенное вне блока страницы, может быть сгенерировано любым известным способом. (Некоторые элементы CSS полезны и занимательны, тогда как другие вполне тривиальны.)

Стили для проекционных устройств

Кроме печатных страниц, еще одним распространенным устройством с постраничной разбивкой является *проекционное* (*projection*), в котором описывается информация, проецируемая на большой экран, подходящий для просмотра большой группой людей. Microsoft PowerPoint – один из наиболее известных современных редакторов для подготовки проекционных презентаций.

На момент написания данной книги только один агент пользователя поддерживает проекционные устройства CSS: Opera для Windows. Эта возможность называется «OperaShow» и позволяет авторам превращать любой HTML-документ в слайдовую презентацию. Мы рассмотрим основные моменты этой возможности, поскольку в будущем она может появиться в других агентах пользователя и дает представление о том, как CSS могут использоваться с другими устройствами, не только экранными или печатными, что довольно любопытно.

Планирование слайдов

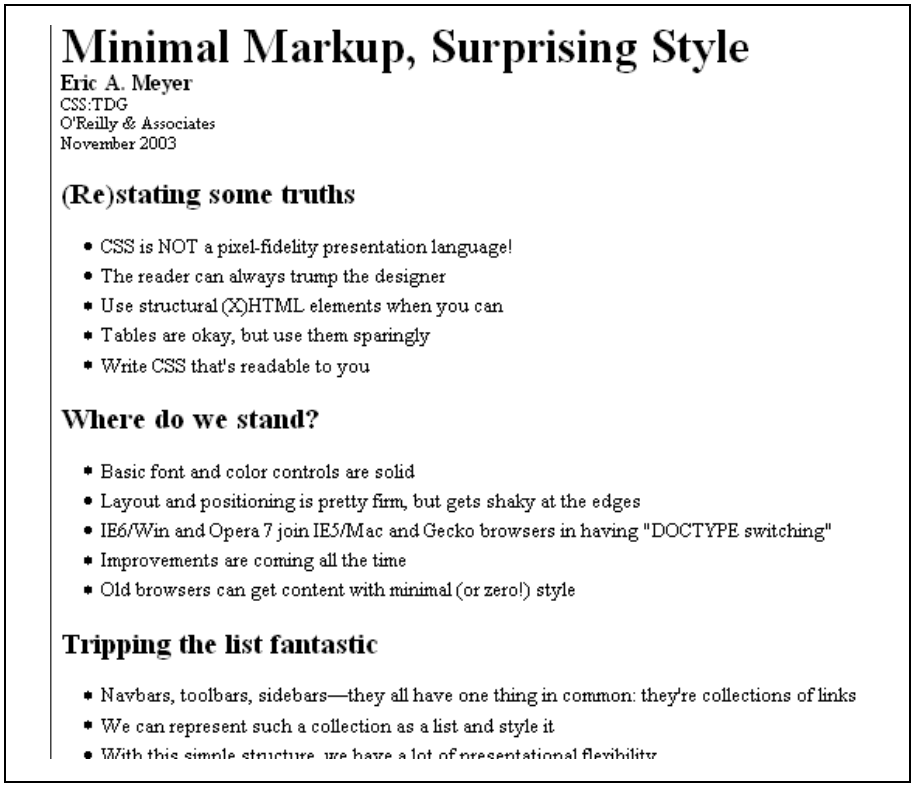
Для того чтобы разбить документ на ряд слайдов, нужен способ определения границ между слайдами. Это делается с помощью свойств разрыва страниц. Какое из них выбрать – page-break-before или page-break-after – преимущественно зависит от того, как организован документ.

В качестве примера рассмотрим HTML-документ, представленный на рис. 14.8. Здесь присутствует ряд элементов h2, каждый из которых сопровождается ненумерованным списком. Они формируют «структуру» слайдовой презентации.

Осталось разбить документ на слайды. Каждый слайд начинается с элемента h2, поэтому достаточно объявить:

```
h2 {page-break-before: always;}
```

Это гарантирует, что каждая страница (т. е. каждый слайд) будет начинаться с элемента h2. Поскольку заголовок каждого слайда представлен элементом h2, все в порядке: в качестве первого элемента каждого слайда будет выступать h2. Визуальное представление слайда можно увидеть на рис. 14.9.



Minimal Markup, Surprising Style

Eric A. Meyer
CSS:TDG
O'Reilly & Associates
November 2003

(Re)stating some truths

- CSS is NOT a pixel-fidelity presentation language!
- The reader can always trump the designer
- Use structural (X)HTML elements when you can
- Tables are okay, but use them sparingly
- Write CSS that's readable to you

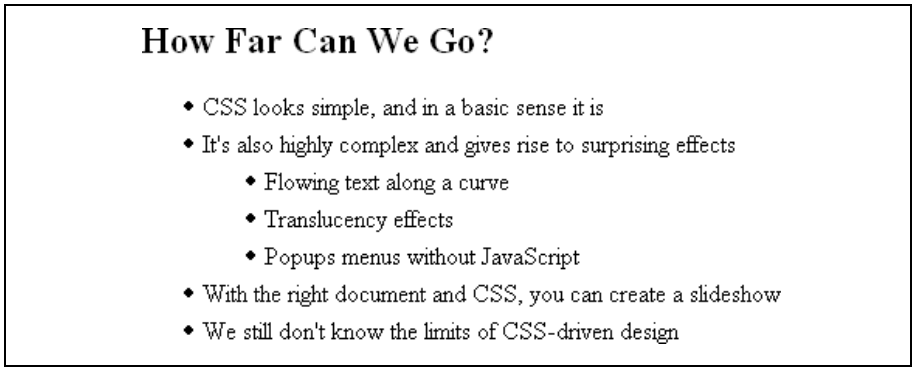
Where do we stand?

- Basic font and color controls are solid
- Layout and positioning is pretty firm, but gets shaky at the edges
- IE6/Win and Opera 7 join IE5/Mac and Gecko browsers in having "DOCTYPE switching"
- Improvements are coming all the time
- Old browsers can get content with minimal (or zero!) style

Tripping the list fantastic

- Navbars, toolbars, sidebars—they all have one thing in common: they're collections of links
- We can represent such a collection as a list and style it
- With this simple structure, we have a lot of presentational flexibility

Рис. 14.8. *План слайдовой презентации в виде простого HTML*



How Far Can We Go?

- CSS looks simple, and in a basic sense it is
- It's also highly complex and gives rise to surprising effects
  - Flowing text along a curve
  - Translucency effects
  - Popups menus without JavaScript
- With the right document and CSS, you can create a slideshow
- We still don't know the limits of CSS-driven design

Рис. 14.9. *Слайд*

Конечно, слайд выглядит довольно незамысловато, потому что ничего не сделано, чтобы приукрасить его; мы просто определили, куда должны быть вставлены разрывы страниц.

В рассматриваемой нами структуре границы слайдов можно было бы определить, вставляя разрывы страниц после списков, а не перед элементами h2:

```
ul {page-break-after: always;}
```

Этот метод будет прекрасно работать, поскольку в документе нет вложенных списков. Если существует вероятность их присутствия, надо или вернуться к размещению разрывов страниц перед элементами h2, или добавить второе правило, чтобы предотвратить разрывы страниц во вложенных списках:

```
ul {page-break-after: always;}
ul ul {page-break-after: auto;}
```

Позиционирование элементов

Начальный блок-контейнер позиционируемых элементов – это блок страницы, в котором они находятся. Таким образом, если требуется, чтобы название каждого слайда располагалось внизу слайда, можно написать приблизительно следующее:

```
h2 {page-break-before: always; position: absolute; bottom: 0; right: 0;}
```

В соответствии с этим правилом элемент h2 размещается в нижнем правом углу его блока страницы (слайда). Конечно, можно позиционировать элементы относительно других элементов, а не блока страницы. Подробности позиционирования рассмотрены в главе 10.

С другой стороны, элемент с фиксированным позиционированием будет появляться в каждом блоке страницы слайдовой презентации, так же как и при печатном представлении. Это значит, что можно взять один элемент, такой как наименование документа, и поместить его на каждый слайд, вот так:

```
h1 {position: fixed; top: 0; right: 0; font-size: 80%;}
```

Эта методика может применяться для создания колонтитулов, графических врезок для каждого слайда и т. д.

Что необходимо учесть при проецировании

Часто говорится, что веб-конструкции должны быть гибкими и способными адаптироваться к любому разрешению, и в большинстве случаев это, безусловно, так. Однако стилевое оформление проекционного представления – это не веб-оформление, поэтому имеет смысл создавать проекционные таблицы стилей, предполагая конкретное разрешение. Например, большинство проекторов (на момент написания данной книги) по умолчанию настроены на разрешение 1024×768. Если известно, что проектор поддерживает данное разрешение, логично настроить CSS именно на него. Размеры шрифтов, размещение элементов и т. д. – все может быть настроено так, чтобы производить наилучшее визуальное впечатление при заданном разрешении.

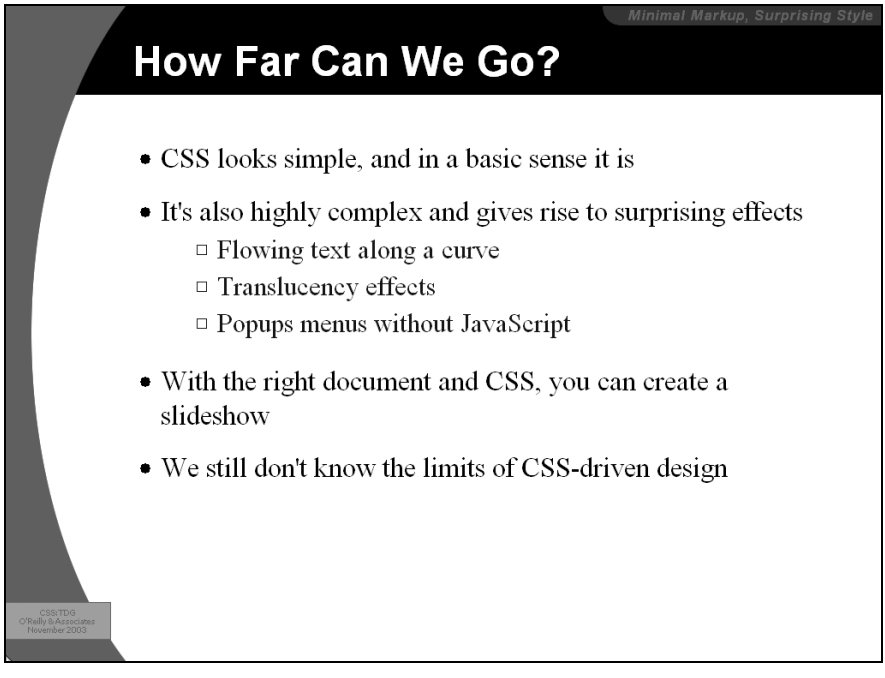


Рис. 14.10. *Полностью оформленный слайд*

Коли на то пошло, можно создавать разные таблицы стилей для разных разрешений: одну для 800×600, другую для 1024×768 и третью для 1280×1024, чтобы охватить наиболее распространенные базовые размеры. На рис. 14.10 показан слайд при разрешении 1024×768.

Еще надо помнить о том, что обычно высокая удобочитаемость проецируемых документов для аудитории обеспечивается контрастными цветами. Это особенно важно, поскольку не все лампы проекторов имеют одинаковую яркость и для более тусклых необходима большая контрастность. В свете (и это не каламбур!) вышеизложенного следует учитывать, что проекционные представления обеспечивают меньшую точность цветопередачи, чем обычные веб-конструкции (это еще мягко сказано).

Стили аудиопредставления

Незрячим пользователям нет никакого дела до визуального стилевого оформления, которому посвящена большая часть CSS. Им важны не отбрасываемые тени или скругленные углы, а реальное текстовое содержимое страницы, которое должно быть представлено внятно и четко. Люди с ослабленным зрением – не единственная категория пользователей, которые оценили бы преимущества звукового представления

веб-содержимого. Встроенный в автомобиль агент пользователя, например, мог бы применять стили аудиопредставления, чтобы разнообразить чтение такого веб-содержимого, как инструкции по вождению или даже электронная почта водителя.

Чтобы отвечать нуждам таких пользователей, в CSS2 введен раздел, описывающий стили аудиопредставления. На момент написания данной книги по крайней мере два агента пользователя частично поддерживают такие стили: Emacspeak и Fonix SpeakThis. Несмотря на это CSS2.1 настоятельно не рекомендует использовать тип устройства `aural` и все связанные с ним свойства. Текущая спецификация содержит примечание о том, что для речевого воспроизведения документов в будущие версии CSS, скорее всего, будет включен тип устройства `speech`, но не приводит никаких подробностей.

Из-за такого странного сочетания выпускаемой реализации и неодобрения, мы лишь вкратце рассмотрим свойства звуковых таблиц стилей.

Воспроизведение речи

Прежде всего необходимо определить, должно ли вообще озвучиваться содержимое данного элемента. В звуковых (`aural`) таблицах стилей это обеспечивается свойством `speak`.

Значение по умолчанию, `normal`, применяется, чтобы показать, что содержимое элемента должно озвучиваться. Если содержимое элемента по какой-то причине не должно озвучиваться, подставляется значение `none`. Значение `none` запрещает формирование аудиопредставления элемента, но его можно переопределить в элементах-потомках, которые, следовательно, будут воспроизводиться. В следующем примере текст «Исходная позиция» озвучивается, а вот текст «Навигация:» – нет:

```
<div style="speak: none;">
Навигация:
<a href="home.html" style="speak: normal;">Исходная позиция</a>
</div>
```

Если *необходимо* запретить формирование аудиопредставления элемента и его потомков, применяется `display: none`. В следующем примере все содержимое элемента `div` не будет воспроизводиться синтезатором речи (а также любым другим устройством):

speak

Значения:	`normal | none | spell-out | inherit`
Начальное значение:	`normal`
Область применения:	все элементы
Наследование:	да
Вычисляемое значение:	как задано

```
<div style="display: none;">
Навигация:
<a href="home.html" style="speak: normal;">Исходная позиция</a>
</div>
```

Третье значение свойства `speak` — `spell-out`, скорее всего, будет применяться к акронимам или другому содержимому, которое должно произноситься по буквам. Например, аудиопредставление следующего фрагмента разметки было бы сгенерировано как T-E-D-S или «ти и ди эс»:

```
<acronym style="speak: spell-out;" title="Technology Evangelism and
  Developer Support">TEDS</acronym>
```

Пунктуация и числа

Есть еще два свойства, оказывающих влияние на то, как генерируется аудиопредставление содержимого элемента. Первое определяет способ воспроизведения пунктуации и называется (что вполне естественно) `speak-punctuation`.

Согласно применяемому по умолчанию значению `none` знаки препинания воспроизводятся как паузы соответствующей длины, хотя CSS не определяет их длительность. Например, пауза, представляющая точку (и таким образом конец предложения), может быть в два раза длиннее паузы, представляющей запятую. Длительность пауз, как можно ожидать, зависит от языка озвучивания.

Согласно значению `code` происходит реальное озвучивание знаков препинания. Таким образом, следующий пример был бы сгенерирован как «стой запятая шалопай восклицательный знак»:

```
<p style="speak-punctuation: code;">Стой, шалопай!</p>
```

Вот другой пример: следующий фрагмент был бы озвучен как «а открывающая скобка href закрывающая скобка открывающая фигурная скобка color двоеточие red точка с запятой закрывающая фигурная скобка»:

```
<code style="speak-punctuation: code;">a[href] {color: red;}</code>
```

Аналогично воспроизведению знаков препинания, свойство `speak-numeral` определяет метод воспроизведения чисел.

speak-punctuation

Значения:	`code | none | inherit`
Начальное значение:	`none`
Область применения:	все элементы
Наследование:	да
Вычисляемое значение:	как задано

speak-numeral

Значения:	digits | continuous | inherit
Начальное значение:	continuous
Область применения:	все элементы
Наследование:	да
Вычисляемое значение:	как задано

Применяемое по умолчанию continuous означает, что числа проговариваются слитно, тогда как digits обусловливает прочтение числа по цифрам. Рассмотрим:

```
<p style="speak-numeral: continuous;">23</p>
<p style="speak-numeral: digits;">23</p>
```

Аудиопредставлением первого абзаца будет «двадцать-три», тогда как второй абзац будет произнесен как «два три». Аудиопредставление чисел, как и знаков препинания, зависит от языка озвучивания, но не определено.

Звуковое воспроизведение заголовков таблиц

При генерировании аудиопредставления таблицы трудно отслеживать, что же на самом деле означают данные ячеек. Допустим, вы находитесь в 9-й строке таблицы, состоящей из 12 строк, и 6-я ячейка этой строки содержит значение «21,77». Каковы шансы, что вы помните, что именно представляет шестой столбец? Вы даже вряд ли вспомните, к чему относятся числа данной строки. Заголовки таблицы предоставляют эту информацию, и с ними легко свериться визуально. Чтобы решить эту проблему в звуковых устройствах, CSS2 вводит свойство speak-header.

По умолчанию агент пользователя генерирует содержимое заголовка таблицы только один раз – при первой встрече с ячейкой. Другой вариант – всегда генерировать информацию заголовка таблицы при генерировании представления каждой ячейки, связанной с этим заголовком.

speak-header

Значения:	once | always | inherit
Начальное значение:	once
Область применения:	элементы, содержащие информацию заголовка таблицы
Наследование:	да
Вычисляемое значение:	как задано

Рассмотрим в качестве примера следующую простую таблицу:

```
<table id="colors">
<caption>Любимый цвет</caption>
<tr id="headers">
<th>Джим</th><th>Джо</th><th>Джейн</th>
</tr>
<tr>
<td>красный</td><td>зеленый</td><td>синий</td>
<td>
</tr>
</table>
```

Без применения каких-либо стилей аудиопредставление этой таблицы было бы таким: «Любимый цвет Джим Джо Джейн красный зеленый синий». Возможно, вы в состоянии определить, что все это означает, но представьте, что таблица содержит любимые цвета 10 или 20 человек. Теперь предположим, что к этой таблице применяются следующие стили:

```
#colors {speak-header: always;}
#headers {speak: none;}
```

Тогда аудиопредставление таблицы должно быть таким: «Любимый цвет Джим красный Джо зеленый Джейн синий». Такое представление понять намного проще независимо от размера таблицы.

Обратите внимание, что метод задания заголовка таблицы определяет язык документа. Языки разметки также могут иметь способы ассоциирования информации заголовка с элементами или группами элементов, например, атрибуты `scope` и `axis` в HTML4.

Скорость речи

Помимо способов влияния на стиль речи CSS предлагает также свойство `speech-rate`, применяемое для задания скорости генерирования аудиопредставления содержимого.

Для него определены следующие значения:

speech-rate

Значения:	<число> | `x-slow` | `slow` | `medium` | `fast` | `x-fast` | `faster` | `slower` | `inherit`
Начальное значение:	`medium`
Область применения:	все элементы
Наследование:	да
Вычисляемое значение:	конкретное число

`<число>`

Определяет скорость произнесения в словах в минуту. Варьируется в зависимости от языка, потому что в некоторых языках скорость речи несколько выше, чем в других.

`x-slow`

Эквивалентно 80 словам в минуту.

`slow`

Эквивалентно 120 словам в минуту.

`medium`

Эквивалентно 180–200 словам в минуту.

`fast`

Эквивалентно 300 словам в минуту.

`x-fast`

Эквивалентно 500 словам в минуту.

`faster`

Увеличивает текущую скорость речи на 40 слов в минуту.

`slower`

Уменьшает текущую скорость речи на 40 слов в минуту.

Вот два примера предельных изменений скорости речи:

```
*.duh {speech-rate: x-slow;}
div#disclaimer {speech-rate: x-fast;}
```

CSS не определяет механизм изменения скорость речи. Агент пользователя может растягивать каждое слово, увеличивать паузы между словами или делать и то, и другое.

Уровень громкости

В звуковых устройствах один из наиболее важных аспектов представления – громкость звука, генерируемого агентом пользователя. Рассмотрим свойство с соответствующим названием – `volume`.

volume

Значения:	<число> | <процентное значение> | `silent` | `x-soft` | `soft` | `medium` | `loud` | `x-loud` | `inherit`
Начальное значение:	`medium`
Область применения:	все элементы
Наследование:	да
Вычисляемое значение:	конкретное число

Для него определены следующие значения:

<число>

Позволяет задать уровень громкости в виде числа. Значение 0 соответствует минимальной слышимости, но это *не* то же самое, что тишина; 100 соответствует максимальному комфортному уровню громкости.

<процентное значение>

Вычисляется как процент от унаследованного уровня громкости.

silent

Звук не воспроизводится, отличается от числового значения 0. Это звуковой эквивалент visibility: hidden.

x-soft

Эквивалентно числовому значению 0.

soft

Эквивалентно числовому значению 25.

medium

Эквивалентно числовому значению 50.

loud

Эквивалентно числовому значению 75.

x-loud

Эквивалентно числовому значению 100.

Важно отметить, что значение свойства volume (произнесите это[1] быстро пять раз!) определяет *средний* уровень громкости, а не точный уровень громкости каждого из произносимых звуков. Таким образом, содержимое элемента, для которого задано volume:50;, может быть отображено звуковой последовательностью с меняющимся относительно заданного уровнем громкости, особенно если голос обладает высокой модуляцией или имеет богатый динамический диапазон.

Числовой диапазон настраивается пользователем, поскольку только конкретный пользователь может определить для себя минимальный уровень громкости (0) и максимальный комфортный уровень громкости (100). Для примера пользователь мог бы решить, что минимально слышимая громкость соответствует силе звука 34 дБ, а максимальная комфортная громкость – 84 дБ. То есть диапазон между значениями 0 и 100 – это 50 дБ, и каждое увеличение значения на единицу будет соответствовать приращению громкости на 0,5 дБ. Иначе говоря, volume: soft; преобразовывалось бы в среднюю громкость 46,5 дБ.

[1] То есть volume value, volume value... – *Примеч. ред.*

Эффект, производимый процентными значениями, аналогичен их применению в свойстве font-size: они увеличивают или уменьшают значение относительно значения родительского элемента. Например:

```
div.marine {volume: 60;}
big {volume: 125%;}

<div class="marine">
Когда я говорю прыгать, я имею в виду <big>кролика</big>, понятно!
</div>
```

Исходя из аудиодиапазона, описанного ранее, содержимое элемента div проговаривалось бы здесь со средней громкостью 64 дБ. Исключение составляет элемент big, громкость которого равна 125% родительского значения 60. Его вычисляемое значение – 75, что эквивалентно 71,5 дБ.

Если в результате вычисления процентного соотношения числовое значение громкости элемента выходит из диапазона от 0 до 100, оно усекается до ближайшего допустимого значения. Предположим, что предыдущие стили изменены так:

```
div.marine {volume: 60;}
big {volume: 200%;}
```

Тогда значение volume для элемента big составит 120; затем это значение будет уменьшено до 100, что в данном случае соответствует средней громкости 84 дБ.

Преимущество такого определения громкости в том, что одна и та же таблица стилей годится для различных сред. Например, установочные параметры, соответствующие 0 и 100, в библиотеке и в машине будут разными, но эти значения будут обеспечивать одинаковые предполагаемые слуховые эффекты при любой настройке.

Озвучивание

До настоящего момента мы говорили о способах воздействия на звуковое представление, но упустили вопрос о том, как выбрать голос для звукового генерирования содержимого. CSS определяет свойство, похожее на font-family, под названием voice-family.

Как и свойство font-family, voice-family позволяет автору предоставить разделяемый запятыми список голосов для формирования содержимого элемента. Если первый голос из списка доступен, то агент пользователя выбирает его. Если нет, агент выбирает следующий по списку голос – и так до тех пор, пока не будет найден один из заданных голосов или пока не закончится список.

Способ объявления синтаксиса значения позволяет указывать последовательность конкретных или базовых семейств в любом порядке. Следовательно, список можно завершить конкретным семейством, а не обязательно базовым. Например:

voice-family	
Значения:	[[<конкретный-голос> | <базовый-голос>],]* [<конкретный-голос> | <базовый-голос>] | inherit
Начальное значение:	зависит от агента пользователя
Область применения:	все элементы
Наследование:	да
Вычисляемое значение:	как задано

```
h1 {voice-family: Mark, male, Joe;}
```

CSS2.x не определяет значения базовых семейств, но упоминает, что значения `male` (мужской), `female` (женский) и `child` (детский) допустимы. Следовательно, элементы документа XML можно было бы оформить так:

```
rosen {voice-family:  Gary, Scott, male;}
guild {voice-family: Tim, Jim, male;}
claud {voice-family: Donald, Ian, male;}
gertr {voice-family: Joanna, Susan, female;}
albert {voice-family: Bobby, Paulie, child;}
```

Фактический голос, выбираемый для генерирования данного элемента, определяет восприятие пользователем этого элемента, поскольку некоторые голоса будут выше или ниже других или будут более или менее монотонными. CSS предоставляет способы воздействия и на эти аспекты голоса.

Изменение голоса

Иногда требуется изменить конкретный голос, которым будет генерироваться аудиопредставление содержимого. Например, голос может звучать нормально, только быть, по-вашему, слишком высоким. Другой голос может быть слишком «динамичным», а во всем остальном отвечать вашим требованиям. CSS определяет свойства, с помощью которых можно менять все параметры воспроизведения голоса.

Изменение высоты

Очевидно, что разные голоса обладают разной высотой. Обычный пример – мужские голоса в среднем имеют частоту 120 Гц, тогда как средняя высота женского голоса приблизительно равна 210 Гц. Таким образом, каждое семейство голосов будет иметь собственную стандартную высоту. CSS позволяет авторам изменять высоту голоса при помощи свойства `pitch`.

Для ключевых слов от `x-low` до `x-high` явного определения нет, так что о них можно сказать только то, что каждое последующее будет озна-

pitch	
Значения:	<частота> | x-low | low | medium | high | x-high | inherit
Начальное значение:	medium
Область применения:	все элементы
Наследование:	да
Вычисляемое значение:	абсолютное значение частоты

чать более высокий тон, чем предыдущее. Можно провести аналогию с ключевыми словами определения размеров шрифта от xx-small до xx-large, которые тоже не заданы точно, но каждое последующее обозначает больший размер, чем предыдущее.

Значения частоты – совсем другое дело. Если частота определяется явно, голос будет изменен так, что его средняя высота будет соответствовать указанному значению. Например:

```
h1 {pitch: 150Hz;}
```

В случае применения непредусмотренного голоса результат может быть непредсказуемым. Рассмотрим пример, в котором для элемента определены два семейства голосов на выбор и частота основного тона:

```
h1 {voice-family: Jethro, Susie; pitch: 100Hz;}
```

Для данного примера предположим, что стандартная частота голоса Jethro составляет 110 Гц, а стандартная частота голоса Susie – 200 Гц. Если выбрать Jethro, то элементы h1 будут прочитываться голосом, несколько более низким, чем обычный. Если голос Jethro недоступен и вместо него выбран Susie, голос будет сильно изменен по сравнению с применяемыми по умолчанию настройками и, скорее всего, будет звучать неестественно.

Независимо от того, какая высота тона устанавливается при воспроизведении элемента, автор может влиять на динамический диапазон частот с помощью свойства pitch-range.

Назначение свойства pitch-range – расширить или сузить модуляцию данного голоса. Чем меньше диапазон высот, тем ближе все высоты бу-

pitch-range	
Значения:	<число> | inherit
Начальное значение:	50
Область применения:	все элементы
Наследование:	да
Вычисляемое значение:	как задано

дут к средней, что в результате обеспечивает монотонный голос. Применяемое по умолчанию значение, 50, обеспечивает «нормальные» модуляции. Чем больше значение, тем выше степень «оживленности» в голосе.

Ударение и насыщенность

Похоже на pitch-range свойство stress, призванное помочь авторам сузить или расширить схемы распределения ударений в языке.

Каждый язык, на котором разговаривают люди, имеет, если можно так выразиться, схемы распределения ударений. В английском, например, предложения состоят из разных частей, которые требуют разных ударений (или акцентирования). Предыдущее предложение[1] могло бы выглядеть примерно так:

```
<sentence>
  <primary>In English,</primary>
  <tertiary>for example,</tertiary>
  <secondary>sentences have different parts that call for
different stress.</secondary>
</sentence>
```

Таблица стилей, определяющая уровни акцентирования каждой части предложения, могла бы быть следующей:

```
primary {stress: 65;}
secondary {stress: 50;}
tertiary {stress: 33;}
```

Это приводит к снижению интонирования менее важных частей предложения и усилению акцентирования частей, которые считаются более важными. Значения свойства stress зависят от языка: одно и то же значение в разных языках может обозначать разные уровни и схемы акцентирования. CSS не определяет такие отличия (что, вероятно, уже вас не удивляет).

Во многом похожим на stress является свойство richness

stress

Значения:	<число> | inherit
Начальное значение:	50
Область применения:	все элементы
Наследование:	да
Вычисляемое значение:	как задано

[1] Оно представлено здесь на языке оригинала, т. к. в русском языке ударения ставятся иначе. – *Примеч. ред.*

richness	
Значения:	<число> | inherit
Начальное значение:	50
Область применения:	все элементы
Наследование:	да
Вычисляемое значение:	как задано

Чем выше значение richness голоса, тем он «ярче» и тем большую аудиторию может охватить. Меньшие значения приведут к мягкому, более «сладкозвучному» («mellifluous») голосу (цитирую спецификацию CSS2). Таким образом, голосу артиста, читающего монолог, можно задать значение richness: 80;, а шепот можно получить, задав richness: 25;.

Паузы и предупредительные сигналы

При визуальной разработке внимание к элементу помогают привлечь дополнительные поля, отделяющие его от всего остального, или рамки. Они как бы призывают обратить взгляд на эти элементы. В аудиопредставлении ближайший эквивалент этим средствам – возможность выделять элемент паузами и звуковыми предупредительными сигналами.

Паузы

Все разговорные языки основаны на различных паузах. Короткие промежутки между словами, фразами и предложениями настолько же важны для понимания значения, как и сами слова. Паузы представляют собой звуковые эквиваленты полей в том смысле, что и те и другие служат для отделения элемента от окружающего его содержимого. В CSS для введения пауз в документ могут применяться три свойства: pause-before, pause-after и pause.

Формат <время> позволяет задать длительность паузы в секундах или миллисекундах. Допустим, требуется вставить полную двухсекундную паузу после элемента h1. Это можно обеспечить любым из двух следующих правил:

```
h1 {pause-after: 2s;}
```

pause-before, pause-after	
Значения:	<время> | <процентное значение> | inherit
Начальное значение:	0
Область применения:	все элементы
Наследование:	нет
Вычисляемое значение:	абсолютное значение времени

```
h1 {pause-after: 2000ms;}  /* продолжительность, аналогичная '2s' */
```

С процентными значениями все немного сложнее, поскольку они вычисляются относительно заданного значения speech-rate. Давайте посмотрим, как это происходит. Во-первых, рассмотрим следующее:

```
h1 {speech-rate: 180;}
```

Это значит, что любой элемент h1 будет воспроизводиться со скоростью примерно три слова в секунду. Теперь рассмотрим:

```
h1 {speech-rate: 180; pause-before: 200%;}
```

Это процентное значение вычисляется на основании средней длины слова. В данном случае произнесение одного слова занимает 333,33 миллисекунды, так что 200% от этого значения составляет 666,66 миллисекунды. Скажем по-другому: каждому h1 будут предшествовать паузы продолжительностью в две трети секунды. Если изменить правило так, что значение speech-rate будет равно 120, пауза будет длиться целую секунду.

Свойство pause – это сокращенная форма записи, которая объединяет pause-before и pause-after.

pause

Значения:	[[<время> | <процентное значение>]{1,2}] | inherit
Начальное значение:	0
Область применения:	все элементы
Наследование:	нет
Вычисляемое значение:	см. отдельные свойства (pause-before и др.)

Если задается только одно значение, оно определяет продолжительность паузы и перед элементом, и после него. Если указывается два значения, тогда первое определяет паузу перед элементом, а второе – паузу после него. Таким образом, следующие правила эквивалентны:

```
pre {pause: 1s;}
pre {pause: 1s 1s;}
pre {pause-before: 1s; pause-after: 1s;}
```

Предупредительные сигналы

Если для привлечения внимания к элементу одних пауз недостаточно, можно добавить перед ним и после него звуковые предупредительные сигналы – звуковой эквивалент рамки. Как и в случае с паузами, для задания звуковых сигналов есть три свойства: cue-before, cue-after и cue.

Если указан URI аудиоресурса, агент пользователя загружает этот ресурс и воспроизводит его перед элементом или после него. Предполо-

cue-before, cue-after

Значения:	<uri> | none | inherit
Начальное значение:	none
Область применения:	все элементы
Наследование:	нет
Вычисляемое значение:	для значений <uri> – абсолютный URI; в противном случае – none

жим, надо предварять каждую непосещенную гиперссылку документа звонком, а каждую посещенную ссылку – гудком. Правила были бы такими:

```
a:link {cue-before: url(chime.mp3);}
a:visited {cue-before: url(beep.wav);}
```

Принцип работы сокращенной формы, cue, вполне ожидаем.

Как и в свойстве pause, если задано одно значение для свойства cue, то это значение используется как перед элементом, так и после него. Если заданы два значения, первое соответствует сигналу перед элементом. Таким образом, следующие правила эквивалентны:

```
a[href] {cue: url(ping.mp3);}
a[href] {cue: url(ping.mp3) url(ping.mp3);}
a[href] {cue-before: url(ping.mp3); cue-after: url(ping.mp3);}
```

cue

Значения:	[<cue-before> || <cue-after>] | inherit
Начальное значение:	none
Область применения:	все элементы
Наследование:	нет
Вычисляемое значение:	см. отдельные свойства (cue-before и др.)

Паузы, предупредительные сигналы и генерируемое содержимое

И паузы, и предупредительные сигналы воспроизводятся «вне» генерируемого содержимого. Рассмотрим следующий пример:

```
h1 {cue: url(trumpet.mp3);}
h1:before {content: "Внемлите! ";}
h1:after {content: ". Воистину!";}

<h1>Начало</h1>
```

Этот элемент был бы воспроизведен примерно так: «(звук трубы) Внемлите! Начало. Воистину! (звук трубы)».

CSS не определяет, идут ли паузы «вне» сигналов или наоборот, так что поведение звуковых агентов пользователя в этом отношении предсказать невозможно.

Звуковое сопровождение

У визуальных элементов может быть фон, и вполне справедливо, что у звуковых элементов тоже есть возможность иметь фон (звуковое сопровождение). В звуковых устройствах роль фона выполняет некий звук, воспроизводимый одновременно с воспроизведением элементом. Для этого предназначено свойство `play-during`.

Самый простой пример – звук воспроизводится в начале звукового элемента:

```
h1 {play-during: url(trumpets.mp3);}
```

В соответствии с этим правилом воспроизведение любого элемента `h1` сопровождалось бы проигрыванием звукового файла *trumpets.mp3*. Звуковой файл проигрывается один раз. Если он короче, чем звуковой эквивалент содержимого элемента, его воспроизведение заканчивается раньше, чем воспроизведение элемента. Если его длительность больше, его воспроизведение завершается одновременно с окончанием воспроизведения содержимого элемента.

Для того чтобы звук повторялся в течение всего воспроизведения элемента, добавьте ключевое слово `repeat`. Это звуковой эквивалент `background-repeat: repeat`:

```
div.ocean {play-during: url(wave.wav) repeat;}
```

Как и фоновые изображения, фоновые звуки по умолчанию не комбинируются. Рассмотрим следующую ситуацию:

```
a:link {play-during: url(chains.mp3) repeat;}
em {play-during: url(bass.mp3) repeat;}

<a href="http://www.example.com/">Это <em>действительно замечательный</em>
сайт!</a>
```

Здесь весь текст ссылки, *кроме* текста элемента, будет воспроизводиться на фоне повторяющегося файла *chains.mp3*. Для элемента `em`

play-during

Значения:	<uri> [mix || repeat]? | auto | none | inherit
Начальное значение:	auto
Область применения:	все элементы
Наследование:	нет
Вычисляемое значение:	для значений <uri> – абсолютный URI; в противном случае – как задано

вместо *chains.mp3* будет воспроизводиться файл *bass.mp3*. Фоновый звук родительского элемента не слышен, так же как его фон не просматривался бы под элементом `em`, если бы фон обоих элементов был видимым.

Для комбинирования фоновых звуков используется ключевое слово `mix`:

```
a:link {play-during: url(chains.mp3) repeat;}
em {play-during: url(bass.mp3) repeat mix;}
```

Теперь *chains.mp3* будет сопровождать чтение текста всей ссылки, включая текст. А во время воспроизведения элемента `em` одновременно будут воспроизводиться и *chains.mp3*, и *bass.mp3*.

Аналогия с визуальными фонами нарушается, если задать значение `none`. Это ключевое слово отменяет все фоновые звуки, включая все те, которые могут принадлежать ссылкам. Таким образом, исходя из следующих правил текст `em` вообще не будет иметь звукового сопровождения – не будет слышно ни файла *bass.mp3*, ни *chains.mp3*:

```
a:link {play-during: url(chains.mp3) repeat;}
em {play-during: none;}

<a href="http://www.example.com/">Это <em>действительно замечательный</em>
сайт!</a>
```

Позиционирование звука

Когда говорит только один человек, звук исходит из одной точки пространства, если только, конечно, человек не перемещается. Когда говорят несколько человек, звук каждого голоса будет исходить из своей точки.

Раз есть высококачественные аудиосистемы и объемный звук, должна быть и возможность создания пространственного звука. CSS2.x предлагает для этого два свойства, одно из которых определяет угол расположения источника звука в горизонтальной плоскости, а второе – угол источника в вертикальной плоскости. За размещение звуков в горизонтальной плоскости отвечает свойство `azimuth`.

azimuth

Значения:	<угол> | [[left-side | far-left | left | center-left | center | center-right | right | far-right | right-side] || behind] | leftwards | rightwards | inherit
Начальное значение:	center
Область применения:	все элементы
Наследование:	да
Вычисляемое значение:	нормированный угол

Значения углов могут выражаться в трех единицах измерения: deg (в градусах), grad (в градах) и rad (в радианах). Допустимые диапазоны этих трех типов единиц измерений: от 0 до 360deg, от 0 до 400grad и от 0 до 6.2831853rad. Допускаются отрицательные значения, но они пересчитываются в положительные. Так, -45deg эквивалентно 315deg (360–45) и -50grad – это то же самое, что и 350grad.

Большинство ключевых слов представляют собой эквиваленты угловых значений. Они приведены в табл. 14.1, где углы заданы в градусах, и проиллюстрированы на рис. 14.11. Последний столбец табл. 14.1 содержит эквиваленты ключевых слов первого столбца, используемых в сочетании со значением behind.

Таблица 14.1. Ключевое слово azimuth и угловые эквиваленты

Ключевое слово	Угол	Behind
center	0	180deg –180deg
center-right	20deg –340deg	160deg –200deg
right	40deg –320deg	140deg –220deg
far-right	60deg –300deg	120deg –240deg
right-side	90deg –270deg	90deg –270deg
center-left	340deg –20deg	200deg –160deg
left	320deg –40deg	220deg –140deg
far-left	300deg –60deg	240deg –120deg
left-side	270deg –90deg	270deg –90deg

Обратите внимание, что ключевое слово behind может сочетаться только с ключевыми словами, перечисленными в табл. 14.1, но не с числовым значением.

Кроме приведенных в табл. 14.1, есть два дополнительных ключевых слова: leftwards и rightwards. В результате применения первого из них из текущего значения угла свойства azimuth вычитается 20deg, а если указать второе, то значение угла увеличивается на 20deg. Например:

```
body {azimuth: right-side;}  /* эквивалентно 90deg */
h1 {azimuth: leftwards;}
```

Вычисляемое значение угла azimuth для элемента h1 составляет 70deg. Теперь рассмотрим следующую ситуацию:

```
body {azimuth: behind;}   /* эквивалентно 180deg */
h1 {azimuth: leftwards;}  /* вычисляемое значение - 160deg */
```

Результатом применения leftwards исходя из данных правил является перемещение звука вправо, а не влево. Это странно, но таков CSS2. Аналогично rightwards в предыдущем примере обусловливает перемещение источника звука элемента h1 на 20 градусов влево.

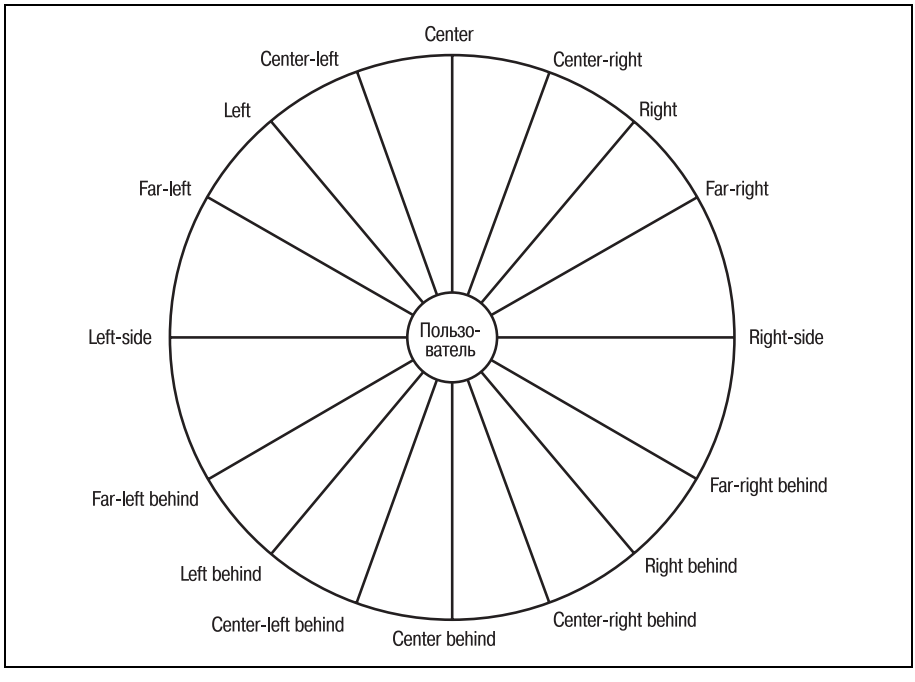


Рис. 14.11. *Горизонтальная плоскость, вид сверху*

Свойство `elevation`, отвечающее за размещение звуков в вертикальной плоскости, во многом аналогично `azimuth`, но несколько проще.

Как и `azimuth`, свойство `elevation` допускает измерение углов в градусах, градах и радианах. Три ключевых слова являются эквивалентами углов: `above` (90 градусов), `level` (0) и `below` (–90 градусов). Их действие проиллюстрировано на рис. 14.12.

Ключевые слова относительного позиционирования, `higher` и `lower`, добавляют по 10 градусов к текущему значению угла возвышения или вычитают их из него. Таким образом, в следующем примере элементы `h1`, являющиеся дочерними элементами `body`, будут подняты на 10 градусов над горизонтальной плоскостью.

elevation

Значения:	<угол> | `below` | `level` | `above` | `higher` | `lower` | `inherit`
Начальное значение:	`level`
Область применения:	все элементы
Наследование:	да
Вычисляемое значение:	нормированный угол

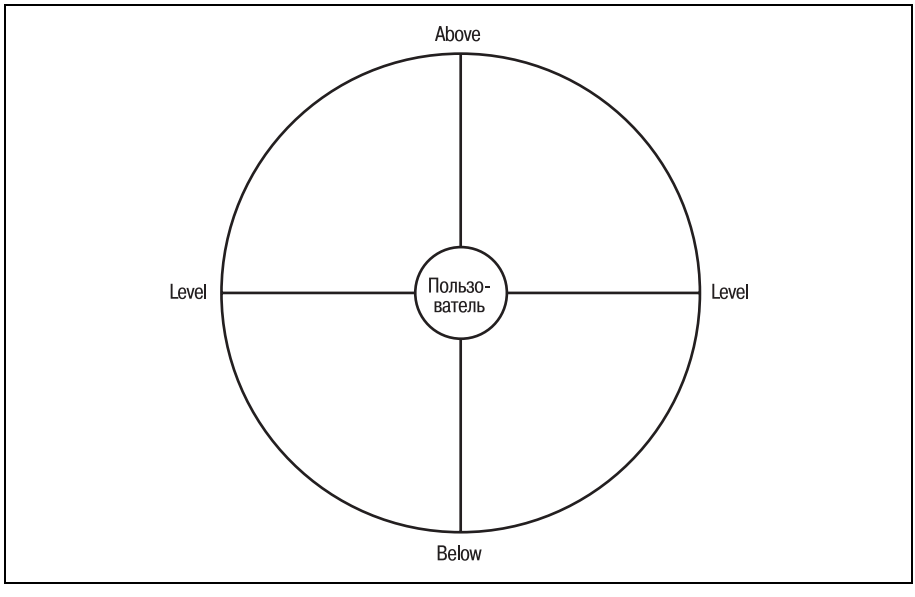


***Рис. 14.12.** Вертикальная плоскость, вид справа*

```
body {elevation: level;}  /* эквивалентно 0 */
body > h1 {elevation: higher;}
```

Сочетание азимута и высоты

Если значения свойств `azimuth` и `elevation` задаются вместе, то определяют точку на воображаемой сфере, в центре которой находится пользователь. На рис. 14.13 показана эта сфера, а также некоторые главные румбы и значения, при которых источники звука были бы помещены в эти положения.

Представьте, что вы сидите в кресле, и некоторая точка находится на полпути между точкой, расположенной прямо перед вами, и точкой, расположенной строго справа, и на полпути между горизонтом и зенитом. Эта точка могла бы быть описана как `azimuth: 45deg; elevation: 45deg;`. Теперь представьте, что источник звука находится на той же высоте, но на полпути между точкой, лежащей строго слева, и точкой, находящейся прямо позади вас. Этот источник мог бы быть описан любым из следующих способов:

```
azimuth: -135deg; elevation: 45deg;
azimuth: 215deg; elevation: 45deg;
azimuth: left behind; elevation: 45deg;
```

Вполне возможно, что пространственно позиционированные звуки будут полезны при разделении сигналов от других звуковых источников или при создании пространственно разделенных специальных материалов:

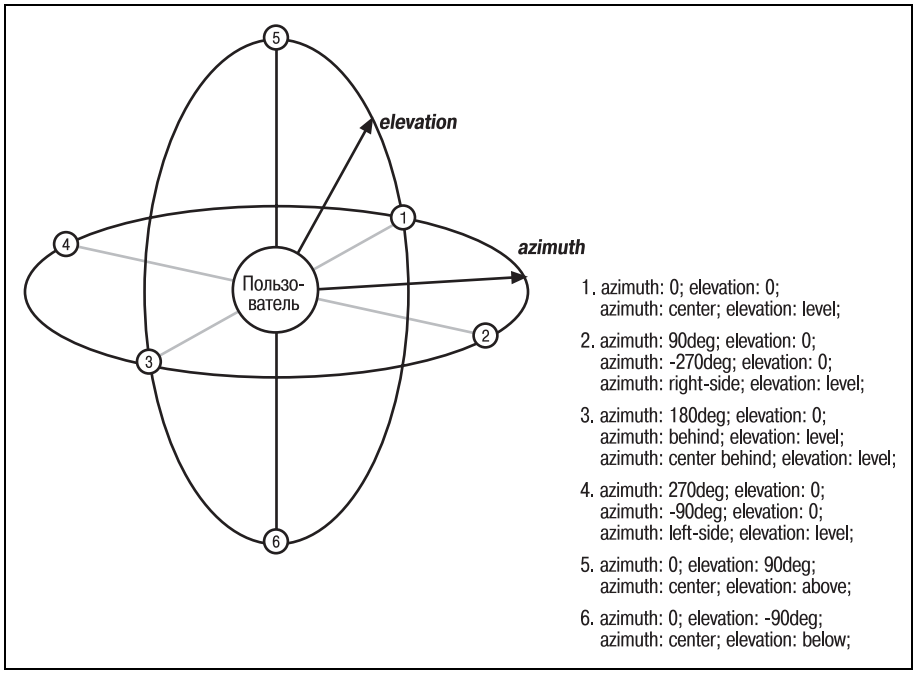


Рис. 14.13. *Трехмерное звуковое пространство*

```
a[href] {cue: url(ping.wav); azimuth: behind; elevation: 30deg;}
voices.onhigh {play-during: url(choir.mp3); elevation: above;}
```

Заключение

Хотя вначале веб-разработка по своей природе была преимущественно визуальной, необходимость представления веб-содержимого в других устройствах привела к появлению в CSS стилевого оформления, ориентированного на определенные устройства. Возможность взять один и тот же документ и настроить его представление так, чтобы он оптимально подходил для различных устройств вывода, – мощный ресурс. Хотя самым распространенным применением описанных возможностей будет создание стилей для печати документов, мы также увидели, как могут применяться стили для проекционных устройств для создания слайдовой презентации в броузере Opera.

Стили аудиопредставления способны существенно облегчить жизнь пользователям с ослабленным зрением, но на момент написания данной книги лишь две программы поддерживают отдельные фрагменты этой части CSS, и тип устройства `aural`, определенный в CSS2.x, не будет перенесен в будущие версии CSS. Тем не менее остается тип устройства `speech`, определенный для звукового представления документов в будущем.

A

Обзор свойств

Визуальные устройства

background

Это сокращенная форма задания всех отдельных свойств фона в одном определении. Применение этого свойства предпочтительнее по сравнению с заданием свойств фона по отдельности, т. к. оно лучше поддерживается старыми броузерами, да и параметры можно ввести быстрее.

Значения: [<background-color> || <background-image> || <background-repeat> || <background-attachment> || <background-position>] | inherit
Начальное значение: см. отдельные свойства
Область применения: все элементы
Наследование: нет
Процентные значения: допустимы для свойства <background-position>
Вычисляемое значение: см. отдельные свойства

background-attachment

Это свойство определяет, будет ли фоновое изображение прокручиваться вместе с элементом при прокрутке документа. Оно может применяться для создания «выровненных» фонов; подробнее см. в главе 9.

Значения: scroll | fixed | inherit
Начальное значение: scroll
Область применения: все элементы
Наследование: нет
Вычисляемое значение: как задано

background-color

Данное свойство задает чистый цвет фона элемента. Этот цвет заполняет области содержимого, отступов и рамок элемента, распространяясь до внешнего края рамки элемента. Для рамок, имеющих прозрачные участки, таких как пунктирные рамки, фоновый цвет будет просматриваться сквозь эти прозрачные участки.

Значения:	<цвет> | transparent | inherit
Начальное значение:	transparent
Область применения:	все элементы
Наследование:	нет
Вычисляемое значение:	как задано

background-image

Размещает изображение на заднем плане элемента. В зависимости от значения свойства background-repeat изображение может выкладываться мозаикой во всех направлениях, вдоль осей или вообще не повторяться. Начальное фоновое изображение (исходное изображение) размещается согласно значению свойства background-position.

Значения:	<uri> | none | inherit
Начальное значение:	none
Область применения:	все элементы
Наследование:	нет
Вычисляемое значение:	абсолютный URI

background-position

Это свойство задает местоположение исходного фонового изображения (заданного свойством background-image); это точка, от которой будет начинаться любое повторение фона.

Значения:	[[<процентное значение> | <длина> | left | center | right] [<процентное значение>] | <длина> | top | center | bottom]?] | [[left | center | right] || [top | center | bottom]] | inherit
Начальное значение:	0% 0%
Область применения:	блочные и замещаемые элементы
Наследование:	нет
Процентные значения:	относительно соответствующей точки элемента и исходного изображения
Вычисляемое значение:	смещения абсолютной длины, если задана <длина>; в противном случае – процентное значение

background-repeat

Определяет шаблон мозаичного размещения фонового изображения. Обратите внимание, что значения, определяющие повторения относительно той или иной оси, фактически обусловливают повторение вдоль соответствующих осей в *обоих* направлениях. Повторение начинается от исходного изображения, которое определяется значением свойства `background-image` и размещается согласно значению свойства `background-position`.

Значения: `repeat | repeat-x | repeat-y | no-repeat | inherit`
Начальное значение: `repeat`
Область применения: все элементы
Наследование: нет
Вычисляемое значение: как задано

border

Это сокращенная форма задания значений свойств, определяющих ширину, цвет и стиль рамки элемента. Обратите внимание, что хотя ни одно из значений не является обязательным, отсутствие значения стиля рамки приведет к ее отсутствию, поскольку по умолчанию стиль рамки – `none`.

Значения: [<border-width> || <border-style> || <border-color>] | `inherit`
Начальное значение: см. отдельные свойства
Область применения: все элементы
Наследование: нет
Вычисляемое значение: как задано

border-bottom

Это свойство определяет ширину, цвет и стиль нижней рамки элемента. Как и в свойстве `border`, отсутствие значения стиля рамки ведет к ее отсутствию.

Значения: [<border-width> || <border-style> || <border-color>] | `inherit`
Начальное значение: для сокращенной формы записи свойств не определено
Область применения: все элементы
Наследование: нет
Вычисляемое значение: см. отдельные свойства (`border-width` и др.)

border-bottom-color

Это свойство задает цвет видимых частей нижней рамки элемента. Может быть задан только чистый цвет; чтобы рамка была видна, ее стиль должен быть отличным от none или hidden.

Значения:	<цвет> | transparent | inherit
Начальное значение:	значение свойства color элемента
Область применения:	все элементы
Наследование:	нет
Вычисляемое значение:	если значение не задано, подставляется вычисляемое значение свойства color этого же элемента; в противном случае – как задано

border-bottom-style

Определяет стиль нижней рамки элемента. Чтобы рамка появилась, значение должно быть отличным от none. В CSS1 HTML-агенты пользователя должны были поддерживать только значения solid и none.

Значения:	none | hidden | dotted | dashed | solid | double | groove | ridge | inset | outset | inherit
Начальное значение:	none
Область применения:	все элементы
Наследование:	нет
Вычисляемое значение:	как задано

border-bottom-width

Задает ширину нижней рамки элемента, которая будет присутствовать, только если стиль рамки отличен от none. Если стиль рамки – none, ширине рамки присваивается значение 0. Отрицательные значения длины не допускаются.

Значения:	thin | medium | thick | <длина> | inherit
Начальное значение:	medium
Область применения:	все элементы
Наследование:	нет
Вычисляемое значение:	абсолютная длина; 0, если стиль рамки – none или hidden

border-color

Это свойство задает цвет видимых частей всей рамки элемента либо задает разные цвета для каждой из четырех сторон. Помните, что для появления видимой рамки ее стиль должен отличаться от none или hidden.

Значения: [<цвет> | transparent]{1,4} | inherit
Начальное значение: для сокращенной формы записи свойств не определено
Область применения: все элементы
Наследование: нет
Вычисляемое значение: см. отдельные свойства (border-top-color и др.)

border-left

Это свойство определяет ширину, цвет и стиль левой рамки элемента. Как и для свойства border, отсутствие значения стиля рамки приводит к ее отсутствию.

Значения: [<border-width> || <border-style> || <border-color>] | inherit
Начальное значение: для сокращенной формы записи свойств не определено
Область применения: все элементы
Наследование: нет
Вычисляемое значение: см. отдельные свойства (border-width и др.)

border-left-color

Это свойство задает цвет видимых частей левой рамки элемента. Задан может быть только чистый цвет; чтобы появилась видимая рамка, ее стиль должен отличаться от none или hidden.

Значения: <цвет> | transparent | inherit
Начальное значение: значение свойства color элемента
Область применения: все элементы
Наследование: нет
Вычисляемое значение: если значение не задано, подставляется вычисляемое значение свойства color этого элемента; в противном случае – как задано

border-left-style

Определяет стиль левой рамки элемента. Чтобы рамка появилась, значение должно отличаться от none. В CSS1 HTML-агенты пользователя должны были поддерживать только значения solid и none.

Значения: none | hidden | dotted | dashed | solid | double | groove | ridge | inset | outset | inherit
Начальное значение: none
Область применения: все элементы
Наследование: нет
Вычисляемое значение: как задано

border-left-width

Задает ширину левой рамки элемента, что будет иметь эффект только в случае, если стиль рамки отличен от none. Если стиль рамки – none, ширине рамки будет присвоено значение 0. Отрицательные значения длины не допускаются.

Значения:	thin | medium | thick | <длина> | inherit
Начальное значение:	medium
Область применения:	все элементы
Наследование:	нет
Вычисляемое значение:	абсолютная длина; 0, если стиль рамки none или hidden

border-right

Это сокращенная форма задания свойств, определяющих ширину, цвет и стиль правой рамки элемента. Как и для значения border, отсутствие значения стиля рамки ведет к ее отсутствию.

Значения:	[<border-width> || <border-style> || <border-color>] | inherit
Начальное значение:	для сокращенной формы записи свойств не определено
Область применения:	все элементы
Наследование:	нет
Вычисляемое значение:	см. отдельные свойства (border-width и др.)

border-right-color

Это свойство задает цвет видимых частей правой рамки элемента. Задан может быть только чистый цвет; чтобы появилась видимая рамка, ее стиль должен отличаться от none или hidden.

Значения:	<цвет> | transparent | inherit
Начальное значение:	значение свойства color элемента
Область применения:	все элементы
Наследование:	нет
Вычисляемое значение:	если значение не задано, подставляется вычисляемое значение свойства color этого элемента; в противном случае – как задано

border-right-style

Определяет стиль правой рамки элемента. Чтобы рамка появилась, значение должно отличаться от none. В CSS1 HTML-агенты пользователя должны были поддерживать только значения solid и none.

Значения: none | hidden | dotted | dashed | solid | double | groove | ridge | inset | outset | inherit
Начальное значение: none
Область применения: все элементы
Наследование: нет
Вычисляемое значение: как задано

border-right-width

Задает ширину правой рамки элемента, что будет иметь эффект, только если стиль рамки отличен от none. Если стиль рамки – none, ширине рамки присваивается значение 0. Отрицательные значения длины не допускаются.

Значения: thin | medium | thick | <длина> | inherit
Начальное значение: medium
Область применения: все элементы
Наследование: нет
Вычисляемое значение: абсолютная длина; 0, если стиль рамки none или hidden

border-style

Эта сокращенная форма задания свойств может применяться для задания стилей всей рамки элемента или каждой из ее сторон в отдельности. Чтобы рамка появилась, ее стиль должен отличаться от none. В CSS1 HTML-агенты пользователя должны были поддерживать только значения solid и none.

Значения: [none | hidden | dotted | dashed | solid | double | groove | ridge | inset | outset]{1,4} | inherit
Начальное значение: для сокращенной формы записи свойств не определено
Область применения: все элементы
Наследование: нет
Вычисляемое значение: см. отдельные свойства (border-top-style и др.)
Примечание: В CSS1 HTML-агенты пользователя должны поддерживать только solid и none; остальные значения (кроме hidden) могут интерпретироваться как solid

border-top

Это сокращенная форма записи свойства, которое определяет ширину, цвет и стиль верхней рамки элемента. Как и для свойства border, отсутствие значения стиля рамки ведет к отсутствию рамки.

Значения: [<border-width> || <border-style> || <border-color>] | inherit

Начальное значение: для сокращенной формы записи свойств не определено
Область применения: все элементы
Наследование: нет
Вычисляемое значение: см. отдельные свойства (`border-width` и др.)

border-top-color

Это свойство задает цвет видимых частей верхней рамки элемента. Может быть задан только чистый цвет; чтобы появилась рамка, ее стиль должен отличаться от `none` или `hidden`.

Значения: <цвет> | `transparent` | `inherit`
Начальное значение: значение свойства `color` элемента
Область применения: все элементы
Наследование: нет
Вычисляемое значение: если значение не задано, подставляется вычисляемое значение свойства `color` этого же элемента; в противном случае – как задано

border-top-style

Определяет стиль верхней рамки элемента. Чтобы рамка появилась, значение должно отличаться от `none`. В CSS1 HTML-агенты пользователя должны были поддерживать только значения `solid` и `none`.

Значения: `none` | `hidden` | `dotted` | `dashed` | `solid` | `double` | `groove` | `ridge` | `inset` | `outset` | `inherit`
Начальное значение: `none`
Область применения: все элементы
Наследование: нет
Вычисляемое значение: как задано

border-top-width

Задает ширину верхней рамки элемента, что будет иметь эффект, только если стиль рамки отличен от `none`. Если стиль – `none`, то ширине присваивается значение `0`. Отрицательные значения длины не допускаются.

Значения: `thin` | `medium` | `thick` | <длина> | `inherit`
Начальное значение: `medium`
Область применения: все элементы
Наследование: нет
Вычисляемое значение: абсолютная длина; `0`, если стиль рамки – `none` или `hidden`

border-width

Это сокращенная форма записи свойств, позволяющая задавать ширину всей рамки элемента или каждой из ее сторон в отдельности. Значение ширины имеет смысл для данной рамки, только если ее стиль отличен от none. Если стиль рамки – none, то ширине присваивается значение 0. Отрицательные значения длины не допускаются.

Значения:	[thin | medium | thick | <длина>]{1,4} | inherit
Начальное значение:	для сокращенной формы записи свойств не определено
Область применения:	все элементы
Наследование:	нет
Вычисляемое значение:	см. отдельные свойства (border-top-style и др.)

bottom

Это свойство определяет смещение между нижним внешним краем поля позиционированного элемента и нижним краем блока-контейнера.

Значения:	<длина> | <процентное значение> | auto | inherit
Начальное значение:	auto
Область применения:	позиционированные элементы (т. е. элементы, для которых значение свойства position отлично от static)
Наследование:	нет
Процентные значения:	относительно высоты блока-контейнера
Вычисляемое значение:	для относительно позиционированных элементов – см. Примечание; для элементов static равно auto; для значений в единицах длины – соответствующая абсолютная длина; для процентных значений – заданное значение; в противном случае – auto
Примечание:	если и bottom, и top позиционированных элементов имеют значения auto, то для обоих вычисляемое значение равно 0; если одно из них имеет значение auto, оно принимает значение, равное по модулю, но противоположное по знаку значению другого; если ни одно из них не auto, bottom принимает значение, равное по модулю, но противоположное по знаку значению top

clear

Определяет, с какой стороны элемента не могут находиться перемещаемые элементы. В CSS1 и CSS2 это достигается путем автоматического увеличения верхнего поля необтекаемого элемента. В CSS2.1 над верхним полем элемента добавляется пространство превышения, но само поле не меняется. В любом случае с объявленной стороны внешний край верхней рамки элемента оказывается сразу под внешним краем нижнего поля перемещаемого элемента.

Значения:	`left | right | both | none | inherit`
Начальное значение:	`none`
Область применения:	блочные элементы
Наследование:	нет
Вычисляемое значение:	как задано

clip

Применяется для определения прямоугольника отсечения, внутри которого содержимое абсолютно позиционированного элемента остается видимым. Содержимое, вышедшее за рамки области отсечения, обрабатывается согласно значению свойства `overflow`. Область отсечения может быть меньше или больше области содержимого элемента.

Значения:	`rect(`*`top`*`, `*`right`*`, `*`bottom`*`, `*`left`*`) | auto | inherit`
Начальное значение:	`auto`
Область применения:	абсолютно позиционированные элементы (в CSS2 `clip` применяется к блочным и замещаемым элементам)
Наследование:	нет
Вычисляемое значение:	для прямоугольника – набор из четырех вычисляемых длин, представляющих границы прямоугольника отсечения; в противном случае – как задано

color

Свойство задает основной цвет элемента, что в генерировании визуального представления HTML означает текст элемента; свойство `color` не оказывает влияния на растровые изображения. Этот цвет также применяется ко всем рамкам элемента, если только для них не задано `border-color` или другое свойство задания цвета рамок (`border-top-color` и др.).

Значения:	<цвет> | `inherit`
Начальное значение:	зависит от агента пользователя
Область применения:	все элементы
Наследование:	да
Вычисляемое значение:	как задано

content

Это свойство используется для определения генерируемого содержимого, размещаемого перед или после элемента. По умолчанию, скорее всего, это будет строковое содержимое, но типом создаваемого содержимым блока можно управлять с помощью свойства `display`.

Значения:	`normal | [` <строка> | <uri> | <счетчик> | `attr(`<идентификатор>`) | open-quote | close-quote | no-open-quote | no-close-quote ]+ | inherit`

Начальное значение: normal
Область применения: псевдоэлементы :before и :after
Наследование: нет
Вычисляемое значение: для <uri> – абсолютный URI; для ссылок на атрибуты – результирующая строка; в противном случае – как задано

counter-increment

Это свойство позволяет увеличить (или уменьшить) счетчики на любую величину, положительную или отрицательную. Если значение <целое> не задано, по умолчанию подставляется 1.

Значения: [<идентификатор> <целое>?]+ | none | inherit
Начальное значение: зависит от агента пользователя
Область применения: все элементы
Наследование: нет
Вычисляемое значение: как задано

counter-reset

Это свойство позволяет задать или сбросить значение счетчика до любого значения, положительного или отрицательного. Если значение <целое> не задано, по умолчанию оно равно 0.

Значения: [<идентификатор> <целое>?]+ | none | inherit
Начальное значение: зависит от агента пользователя
Область применения: все элементы
Наследование: нет
Вычисляемое значение: как задано

cursor

Определяет форму указателя мыши, находящегося в границах элемента (хотя CSS2.1 не определяет, какой край формирует эти границы).

Значения: [[<uri>,]* [auto | default | pointer | crosshair | move | e-resize | ne-resize | nw-resize | n-resize | se-resize | sw-resize | s-resize | w-resize| text | wait | help | progress]] | inherit
Начальное значение: auto
Область применения: все элементы
Наследование: да
Вычисляемое значение: для значения <uri> – абсолютный URI; в противном случае – как задано

direction

Это свойство определяет базовое направление написания блоков и направления встраивания и перекрытия для двунаправленного алгоритма Unicode. Агентам пользователя, не поддерживающим двунаправленное написание текста, разрешается игнорировать это свойство.

Значения: `ltr | rtl | inherit`
Начальное значение: `ltr`
Область применения: все элементы
Наследование: да
Вычисляемое значение: как задано

display

Применяется для определения типа блока представления, генерируемого элементом при компоновке. Неуместное применение свойства с такими типами документов, как HTML, может быть опасным, поскольку оно нарушает иерархию представления, уже имеющуюся в HTML. В случае XML-документов, которые не имеют такой встроенной иерархии, применение `display` обязательно.

Значения: `none | inline | block | inline-block | list-item | run-in | table | inline-table | table-row-group | table-header-group | table-footer-group | table-row | table-column-group | table-column | table-cell | table-caption | inherit`
Начальное значение: `inline`
Область применения: все элементы
Наследование: нет
Вычисляемое значение: меняется для перемещаемых, абсолютно позиционированных и корневых элементов (см. CSS2.1, раздел 9.7); в противном случае – как задано
Примечание: значения `compact` и `marker` появились в CSS2, но были изъяты из CSS2.1 из-за отсутствия широкой поддержки

float

Определяет направление, в котором происходит перемещение элемента. Традиционно применяется к изображениям, чтобы обеспечить возможность обтекания текста вокруг них, но в CSS перемещаемым может быть любой элемент. Перемещаемый элемент будет рассматриваться как блочный независимо от того, каким элементом он является. Для перемещаемых незамещаемых элементов должна быть явно задана ширина, поскольку в противном случае они стремятся стать по возможности максимально узкими.

Значения: `left | right | none | inherit`
Начальное значение: `none`

Область применения: все элементы
Наследование: нет
Вычисляемое значение: как задано

font

Это сокращенная форма записи свойств, которая предназначена для одновременного задания двух или более параметров шрифта элемента. Она также может применяться для приведения шрифта элемента в соответствие с внешним видом вычислительной среды пользователя с помощью ключевых слов, таких как `icon`. Заметьте, что если эти ключевые слова не указаны, то для определения шрифта надо задать как минимум размер шрифта и семейство.

Значения: `[[ <font-style> || <font-variant> || <font-weight> ]? <font-size> [ / <line-height> ]? <font-family>] | caption | icon | menu | message-box | small-caption | status-bar | inherit`
Начальное значение: см. отдельные свойства
Область применения: все элементы
Наследование: да
Процентные значения: для <font-size> вычисляется относительно родительского элемента и относительно <font-size> элемента для <line-height>
Вычисляемое значение: см. отдельные свойства (`font-style` и др.)

font-family

Определяет семейство шрифтов, выбираемое для представления текста элемента. Обратите внимание, что применение определенного семейства шрифтов (например, Geneva) всецело зависит от того, доступно ли это семейство на компьютере пользователя; это свойство не предполагает загрузки шрифтов. Поэтому настоятельно рекомендуется использование имен базовых семейств шрифтов.

Значения: `[[<имя-семейства> | <базовое-семейство>], ]* [<имя-семейства> | <базовое-семейство>] | inherit`
Начальное значение: зависит от агента пользователя
Область применения: все элементы
Наследование: да
Вычисляемое значение: как задано

font-size

Задает размер шрифта элемента. Заметьте, что на самом деле задается высота блоков символов шрифта; фактическая высота глифов символов может быть больше или меньше этих блоков (обычно меньше). Каждое

ключевое слово должно соответствовать размеру, превосходящему тот, который был определен предыдущим ключевым словом, и меньшему, чем определяемый последующим ключевым словом. Отрицательные значения длины и процентных значений не допускаются.

Значения: `xx-small` | `x-small` | `small` | `medium` | `large` | `x-large` | `xx-large` | `smaller` | `larger` | <длина> | <процентное значение> | `inherit`

Начальное значение: `medium`

Область применения: все элементы

Наследование: да

Процентные значения: вычисляются относительно размера шрифта родительского элемента

Вычисляемое значение: абсолютная длина

font-style

Задает гарнитуру шрифта: курсивную, наклонную или обычную. Текст курсивного начертания обычно определяется как отдельная гарнитура в рамках семейства шрифтов. Теоретически агент пользователя может формировать наклонную гарнитуру шрифта из обычной.

Значения: `italic` | `oblique` | `normal` | `inherit`

Начальное значение: `normal`

Область применения: все элементы

Наследование: да

Вычисляемое значение: как задано

font-variant

Это свойство в основном применяется для определения капители. Теоретически агент пользователя может формировать гарнитуру капители из гарнитуры обычного шрифта.

Значения: `small-caps` | `normal` | `inherit`

Начальное значение: `normal`

Область применения: все элементы

Наследование: да

Вычисляемое значение: как задано

font-weight

Это свойство задает насыщенность шрифта, используемую при генерировании визуального представления текста элемента. Числовое значение `400` эквивалентно ключевому слову `normal`, а `700` эквивалентно `bold`. Насыщенность, соответствующая каждому числовому значению, должна быть не меньше определяемой предыдущим числом и не больше определяемой последующим числом.

Значения: normal | bold | bolder | lighter | 100 | 200 | 300 | 400 | 500 | 600 | 700 | 800 | 900 | inherit

Начальное значение: normal

Область применения: все элементы

Наследование: да

Вычисляемое значение: одно из числовых значений (100 и др.) или одно из числовых значений плюс одно из относительных значений (bolder или lighter)

height

Определяет высоту области содержимого элемента, к которой добавляются отступы, рамки и поля. Это свойство игнорируется для строковых незамещаемых элементов. Отрицательные значения длины и процентные значения не допускаются.

Значения: <длина> | <процентное значение> | auto | inherit

Начальное значение: auto

Область применения: блочные и замещаемые элементы

Наследование: нет

Процентные значения: вычисляются относительно высоты блока-контейнера

Вычисляемое значение: для значения auto и процентных значений – как задано; в противном случае – абсолютная длина, если только свойство не применяется к элементу (тогда auto)

left

Это свойство определяет смещение между левым внешним краем поля позиционированного элемента и левым краем его блока-контейнера.

Значения: <длина> | <процентное значение> | auto | inherit

Начальное значение: auto

Область применения: позиционированные элементы (т. е. элементы, для которых значение position отличается от static)

Наследование: нет

Процентные значения: вычисляются относительно ширины блока-контейнера

Вычисляемое значение: для относительно позиционированных элементов см. Примечание; для элементов static равно auto; для значений длины – соответствующая абсолютная длина; для процентных значений – заданное значение; в противном случае – auto

Примечание: для относительно позиционированных элементов вычисляемое значение свойства left всегда эквивалентно значению свойства right

letter-spacing

Определяет количество пробелов, которые должны быть вставлены между блоками символов текста. Поскольку глифы символов обычно уже, чем их блоки, значения длины модифицируют обычное расстояние между буквами. Таким образом, значение `normal` аналогично `0`. Отрицательные значения длины допускаются и обусловливают более плотное размещение букв.

Значения: <длина> | `normal` | `inherit`
Начальное значение: `normal`
Область применения: все элементы
Наследование: да
Вычисляемое значение: для значений длины – абсолютная длина; в противном случае – `normal`

line-height

Это свойство влияет на компоновку контейнеров строк. В применении к блочному элементу определяет минимальное расстояние между базовыми линиями этого элемента, но не максимальное. Разница между вычисляемыми значениями `line-height` и `font-size` (в CSS называемая межстрочным интервалом) делится на два и добавляется сверху и снизу каждого фрагмента содержимого строки текста. Минимальный блок, который может включить все эти фрагменты содержимого, – контейнер строки. Числовое значение без указания единиц измерения устанавливает масштабный коэффициент, который представляет собой наследуемое, а не вычисляемое значение. Отрицательные значения не допускаются.

Значения: <длина> | <процентное значение> | <число> | `normal` | `inherit`
Начальное значение: `normal`
Область применения: все элементы (см. разделы, посвященные замещаемым и блочным элементам)
Наследование: да
Процентные значения: относительно размера шрифта элемента
Вычисляемое значение: для значений длины и процентных значений – абсолютное значение; в противном случае – как задано

list-style

Это сокращенная форма записи свойств, которая собирает воедино все остальные свойства определения стилей списка. Поскольку оно применяется ко всем элементам, свойство `display` которых имеет значение `list-item`, в обычном HTML и XHTML оно будет применяться только к элементам `li`, хотя может применяться к любому элементу и наследоваться элементами `list-item`.

Значения: [<list-style-type> || <list-style-image> || <list-style-position>] | inherit

Начальное значение: см. отдельные свойства

Область применения: элементы, свойство display которых имеет значение list-item

Наследование: да

Вычисляемое значение: см. отдельные свойства

list-style-image

Определяет изображение, выступающее в качестве маркера элемента нумерованного или ненумерованного списка. Размещением изображения относительно содержимого элемента списка можно свободно управлять с помощью свойства list-style-position.

Значения: <uri> | none | inherit

Начальное значение: none

Область применения: элементы, свойство display которых имеет значение list-item

Наследование: да

Вычисляемое значение: для значений <uri> – абсолютный URI; в противном случае – none

list-style-position

Это свойство применяется для определения местоположения маркера списка относительно содержимого элемента списка. Внешние маркеры размещаются на некотором расстоянии от края рамки элемента списка, но это расстояние в CSS не определяется. Внутренние маркеры интерпретируются как строковые элементы, вставленные в начало содержимого элемента списка.

Значения: inside | outside | inherit

Начальное значение: outside

Область применения: элементы, свойство display которых имеет значение list-item

Наследование: да

Вычисляемое значение: как задано

list-style-type

Применяется для объявления типа принятой в представлении списка системы маркировки.

CSS2.1 Значения: disc | circle | square | decimal | decimal-leading-zero | lower-roman | upper-roman | lower-greek | lower-latin | upper-latin | armenian | georgian | none | inherit

CSS2 Значения: disc | circle | square | decimal | decimal-leading-zero | upper-alpha | lower-alpha | upper-roman | lower-roman | lower-greek | hebrew | armenian | georgian | cjk-ideographic | hiragana | katakana | hiragana-iroha | none | inherit

Начальное значение: disc

Область применения: элементы, свойство display которых имеет значение list-item

Наследование: да

Вычисляемое значение: как задано

margin

Это сокращенная форма записи свойств, которая задает ширину всего поля элемента или ширину поля с каждой стороны отдельно. Смежные по вертикали поля блочных элементов сворачиваются, тогда как поля сверху и снизу не оказывают эффекта на строковые элементы. Левое и правое поля строковых элементов не сворачиваются, так же как и поля перемещаемых элементов. Отрицательные значения для задания полей допускаются, но применять их надо с осторожностью.

Значения: [<длина> | <процентное значение> | auto]{1,4} | inherit

Начальное значение: не определено

Область применения: все элементы

Наследование: нет

Процентные значения: относительно ширины блока-контейнера

Вычисляемое значение: см. отдельные свойства

margin-bottom

Задает ширину нижнего поля элемента. Отрицательные значения допускаются, но использовать их надо с осторожностью.

Значения: <длина> | <процентное значение> | auto | inherit

Начальное значение: 0

Область применения: все элементы

Наследование: нет

Процентные значения: относительно ширины блока-контейнера

Вычисляемое значение: для процентных значений – как задано; для значений длины – абсолютная длина

margin-left

Задает ширину левого поля элемента. Отрицательные значения допускаются, но применять их надо с осторожностью.

Значения: <длина> | <процентное значение> | auto | inherit

Начальное значение: 0
Область применения: все элементы
Наследование: нет
Процентные значения: относительно ширины блока-контейнера
Вычисляемое значение: для процентных значений – как задано; для значений длины – абсолютная длина

margin-right

Задат ширину правого поля элемента. Отрицательные значения допускаются, но задавать их надо с осторожностью.

Значения: <длина> | <процентное значение> | auto | inherit
Начальное значение: 0
Область применения: все элементы
Наследование: нет
Процентные значения: относительно ширины блока-контейнера
Вычисляемое значение: для процентных значений – как задано; для значений длины – абсолютная длина

margin-top

Задает ширину верхнего поля элемента. Отрицательные значения допускаются, но задавать их надо с осторожностью.

Значения: <длина> | <процентное значение> | auto | inherit
Начальное значение: 0
Область применения: все элементы
Наследование: нет
Процентные значения: относительно ширины блока-контейнера
Вычисляемое значение: для процентных значений – как задано; для значений длины – абсолютная длина

max-height

Значение этого свойства задает максимально допустимую высоту элемента. Таким образом, элемент может быть ниже заданного значения, но не выше. Отрицательные значения не допускаются.

Значения: <длина> | <процентное значение> | none | inherit
Начальное значение: none
Область применения: все элементы, кроме строковых незамещаемых элементов и элементов таблиц
Наследование: нет
Процентные значения: относительно высоты блока-контейнера
Вычисляемое значение: для процентных значений – как задано; для значений длины – абсолютная длина; в противном случае – none

max-width

Значение этого свойства задает максимально допустимую ширину элемента. Таким образом, элемент может быть уже заданного значения, но не шире. Отрицательные значения не допускаются.

Значения:	<длина> | <процентное значение> | none | inherit
Начальное значение:	none
Область применения:	все элементы, кроме строковых незамещаемых элементов и элементов таблиц
Наследование:	нет
Процентные значения:	относительно ширины блока-контейнера
Вычисляемое значение:	для процентных значений – как задано; для значений длины – абсолютная длина; в противном случае – none

min-height

Значение этого свойства задает минимально допустимую высоту элемента. Таким образом, элемент может быть выше заданного значения, но не ниже. Отрицательные значения не допускаются.

Значения:	<длина> | <процентное значение> | inherit
Начальное значение:	0
Область применения:	все элементы, кроме строковых незамещаемых элементов и элементов таблиц
Наследование:	нет
Процентные значения:	относительно высоты блока-контейнера
Вычисляемое значение:	для процентных значений – как задано; для значений длины – абсолютная длина

min-width

Значение этого свойства задает минимально допустимую ширину элемента. Таким образом, элемент может быть шире заданного значения, но не уже. Отрицательные значения не допускаются.

Значения:	<длина> | <процентное значение> | inherit
Начальное значение:	0
Область применения:	все элементы, кроме строковых незамещаемых элементов и элементов таблиц
Наследование:	нет
Процентные значения:	относительно ширины блока-контейнера
Вычисляемое значение:	для процентных значений – как задано; для значений длины – абсолютная длина; в противном случае – none

outline

Это сокращенная форма записи свойств, которая используется для задания общего контура элемента. Контуры могут быть неправильной формы; они не меняют или, другими словами, не влияют на размещение элементов.

Значения:	[<outline-color> || <outline-style> || <outline-width>] | inherit
Начальное значение:	для сокращенной формы записи свойств не определено
Область применения:	все элементы
Наследование:	нет
Вычисляемое значение:	см. отдельные свойства (outline-color и др.)

outline-color

Это свойство задает цвет видимых частей общего контура элемента. Помните, чтобы появилась рамка, стиль контура должен отличаться от none.

Значения:	<цвет> | invert | inherit
Начальное значение:	invert (или зависит от агента пользователя; см. текст)
Область применения:	все элементы
Наследование:	нет
Вычисляемое значение:	как задано

outline-style

Это свойство предназначено для задания стиля всего контура элемента. Чтобы появился контур, стиль должен отличаться от none.

Значения:	none | dotted | dashed | solid | double | groove | ridge | inset | outset | inherit
Начальное значение:	none
Область применения:	все элементы
Наследование:	нет
Вычисляемое значение:	как задано

outline-width

Это свойство задает ширину общего контура элемента. Значение вступает в силу, только если стиль заданного контура отличен от none. Если стиль — none, ширине присваивается значение 0. Отрицательные значения длины не допускаются.

Значения:	thin | medium | thick | <длина> | inherit
Начальное значение:	medium

Область применения: все элементы
Наследование: нет
Вычисляемое значение: абсолютная длина; равно `0`, если стиль рамки — `none` или `hidden`

overflow

Определяет, что происходит с содержимым, которое переполняет область содержимого элемента. Для значения `scroll` предполагается, что агенты пользователя предоставляют механизм прокрутки независимо от реальной потребности в нем; таким образом, полосы прокрутки появляются, даже если все содержимое помещается в блок элемента.

Значения: `visible | hidden | scroll | auto | inherit`
Начальное значение: `visible`
Область применения: блочные и замещаемые элементы
Наследование: нет
Вычисляемое значение: как задано

padding

Это сокращенная форма записи свойств, которая задает ширину общего отступа элемента или ширину отступов с каждой стороны отдельно. Отступ, заданный для строкового незамещаемого элемента, не оказывает влияния на вычисление высоты строки; следовательно, элемент, имеющий и отступы, и фон, может визуально выходить на другие строки и потенциально перекрывать другое содержимое. Фон элемента будет распространяться на отступы. Отрицательные значения отступов не допускаются.

Значения: [<длина> | <процентное значение>]{1,4} | `inherit`
Начальное значение: не задано для стенографических элементов
Область применения: все элементы
Наследование: нет
Процентные значения: относительно ширины блока-контейнера
Вычисляемое значение: см. отдельные свойства (`padding-top` и др.)
Примечание: отступ не может иметь отрицательного значения

padding-bottom

Это свойство задает ширину нижнего отступа элемента. Отступ снизу, заданный для строковых незамещаемых элементов, не влияет на вычисление высоты строки; поэтому элемент, имеющий и отступы, и фон, может визуально выходить на другие строки и потенциально перекрывать другое содержимое. Отрицательные значения отступов не допускаются.

Значения:	<длина> | <процентное значение> | inherit
Начальное значение:	0
Область применения:	все элементы
Наследование:	нет
Процентные значения:	относительно ширины блока-контейнера
Вычисляемое значение:	для процентных значений – как задано; для значений длины – абсолютная длина
Примечание:	отступ никогда не может иметь отрицательного значения

padding-left

Это свойство задает ширину левого отступа элемента. Отступ слева, заданный для строкового незамещаемого элемента, появится только с левого края первого строкового блока, генерируемого элементом. Отрицательные значения отступов не допускаются.

Значения:	<длина> | <процентное значение> | inherit
Начальное значение:	0
Область применения:	все элементы
Наследование:	нет
Процентные значения:	относительно ширины блока-контейнера
Вычисляемое значение:	для процентных значений – как задано; для значений длины – абсолютная длина
Примечание:	отступ не может иметь отрицательного значения

padding-right

Это свойство задает ширину правого отступа элемента. Отступ справа, заданный для строкового незамещаемого элемента, появится только с правого края последнего строкового блока, генерируемого элементом. Отрицательные значения отступов не допускаются.

Значения:	<длина> | <процентное значение> | inherit
Начальное значение:	0
Область применения:	все элементы
Наследование:	нет
Процентные значения:	относительно ширины блока-контейнера
Вычисляемое значение:	для процентных значений – как задано; для значений длины – абсолютная длина
Примечание:	отступ не может иметь отрицательного значения

padding-top

Это свойство задает ширину верхнего отступа элемента. Отступ сверху, заданный для строковых незамещаемых элементов, не оказывает

влияния на вычисление высоты строки; поэтому элемент, для которого заданы и отступ сверху, и фон, может визуально выходить на другие строки и потенциально перекрывать другое содержимое. Отрицательные значения отступов не допускаются.

Значения:	<длина> | <процентное значение> | inherit
Начальное значение:	0
Область применения:	все элементы
Наследование:	нет
Процентные значения:	относительно ширины блока-контейнера
Вычисляемое значение:	для значений процентных значений – как задано; для значений длины – абсолютная длина
Примечание:	отступ не может иметь отрицательного значения

position

Определяет схему позиционирования, принятую для компоновки элемента. Любой элемент может быть позиционированным, хотя элементы, позиционированные как absolute или fixed, будут рассматриваться как блочные независимо от своего типа. Относительно позиционированный элемент перемещается из его стандартного местоположения в нормальном потоке.

Значения:	static | relative | absolute | fixed | inherit
Начальное значение:	static
Область применения:	все элементы
Наследование:	нет
Вычисляемое значение:	как задано

quotes

Это свойство применяется для определения схемы заключения в кавычки, применяемой в цитатах и во вложенных цитатах. Фактические кавычки вставляются с помощью свойства content.

Значения:	[<строка> <строка>]+ | none | inherit
Начальное значение:	зависит от агента пользователя
Область применения:	все элементы
Наследование:	да
Вычисляемое значение:	как задано

right

Это свойство определяет смещение между внешним краем правого поля позиционированного элемента и правым краем его блока-контейнера.

Значения:	<длина> | <процентное значение> | auto | inherit
Начальное значение:	auto

Область применения: позиционированные элементы (т. е. элементы, для которых значение свойства `position` отлично от `static`)

Наследование: нет

Процентные значения: относительно ширины блока-контейнера

Вычисляемое значение: для относительно позиционированных элементов см. Примечание; для элементов `static` – `auto`; для значений длины – соответствующая абсолютная длина; для процентных значений – заданное значение; в противном случае – `auto`

Примечание: для относительно позиционированных элементов вычисляемое значение `left` всегда эквивалентно значению `right`

text-align

Это свойство задает горизонтальное выравнивание текста блочного элемента, определяя точку, относительно которой выравниваются контейнеры строк. Значение `justify` поддерживается путем предоставления агентам пользователя возможности программно регулировать расстояния между буквами и словами содержимого строки; результаты могут меняться в зависимости от агентов пользователя.

Значения CSS2.1: `left | center | right | justify | inherit`

Значения CSS2: `left | center | right | justify |` <строка> `| inherit`

Начальное значение: зависит от агента пользователя; также может зависеть от направления написания

Область применения: блочные элементы

Наследование: да

Вычисляемое значение: как задано

Примечание: CSS2 включал значение <строка>, которое было изъято из CSS2.1 из-за отсутствия широкой поддержки

text-decoration

Это свойство обеспечивает применение к тексту определенных эффектов, таких как подчеркивание. Эти декоративные элементы будут распространяться на элементы-потомки, не имеющие собственного оформления. От агентов пользователя не требуется поддержки значения `blink`.

Значения: `none | [ underline || overline || line-through || blink ] | inherit`

Начальное значение: `none`

Область применения: все элементы

Наследование: нет

Вычисляемое значение: как задано

text-indent

Применяется для определения абзацного отступа первой строки содержимого блочного элемента. Чаще всего применяется для создания эффекта табуляции. Отрицательные значения допускаются и создают эффекты «выступа».

Значения:	<длина> | <процентное значение> | inherit
Начальное значение:	0
Область применения:	блочные элементы
Наследование:	да
Процентные значения:	относительно ширины блока-контейнера
Вычисляемое значение:	для процентных значений – как задано; для значений длины – абсолютная длина

text-transform

Это свойство меняет регистр букв элемента независимо от регистра текста исходного документа. Не описано точно, регистр каких букв должен быть изменен значением capitalize, поскольку это зависит от агента пользователя, знающего, как распознавать «слово».

Значения:	uppercase | lowercase | capitalize | none | inherit
Начальное значение:	none
Область применения:	все элементы
Наследование:	да
Вычисляемое значение:	как задано

top

Это свойство определяет смещение между внешним краем верхнего поля позиционированного элемента и верхним краем его блока-контейнера.

Значения:	<длина> | <процентное значение> | auto | inherit
Начальное значение:	auto
Область применения:	позиционированные элементы (т. е. элементы, значение свойства position которых отлично от static)
Наследование:	нет
Процентные значения:	относительно высоты блока-контейнера
Вычисляемое значение:	для относительно позиционированных элементов см. Примечание; для элементов static равно auto; для значений длины подставляется соответствующая абсолютная длина; для процентных значений – заданное значение; в противном случае – auto
Примечание:	для относительно позиционированных элементов, если и top, и bottom имеют значение auto, вычисляемое

значение для обоих свойств будет равно 0; если только одно из них имеет значение auto, оно принимает значение, равное по модулю, но противоположное по знаку значению другого; если ни одно из них не имеет значения auto, bottom принимает значение, равное по модулю, но противоположное по знаку значению top

unicode-bidi

Позволяет авторам генерировать уровни встраивания в рамках алгоритма встраивания Unicode. Агентам пользователя, которые не поддерживают двунаправленный текст, разрешается игнорировать это свойство.

Значения: normal | embed | bidi-override | inherit
Начальное значение: normal
Область применения: все элементы
Наследование: нет
Вычисляемое значение: как задано

vertical-align

Определяет вертикальное выравнивание базовой линии строкового элемента относительно базовой линии строки, в которой он находится. Допускаются отрицательные значения длин и процентных значений; в соответствии с ними элемент опускается, а не поднимается. В ячейках таблицы это свойство задает выравнивание содержимого ячейки в блоке ячейки.

Значения: baseline | sub | super | top | text-top | middle | bottom | text-bottom | <процентное значение> | <длина> | inherit
Начальное значение: baseline
Область применения: строковые элементы и ячейки таблиц
Наследование: нет
Процентные значения: относительно значения line-height элемента
Вычисляемое значение: для процентных значений и значений длины – абсолютная длина; в противном случае – как задано
Примечание: в применении к ячейкам таблицы распознаются только значения baseline, top, middle и bottom

visibility

Определяет, будет ли генерироваться визуальное представление блока, генерируемого элементом. Это означает, что элемент может занимать положенное ему место, но при этом быть совершенно невидимым. Значение collapse применяется к таблицам для удаления столбцов или строк из макета таблицы.

Значения: visible | hidden | collapse | inherit

Начальное значение: visible
Область применения: все элементы
Наследование: да
Вычисляемое значение: как задано

white-space

Объявляет, как обрабатываются пробелы элемента во время компоновки. Значения pre-wrap и pre-line были добавлены в CSS2.1.

Значения: normal | nowrap | pre | pre-wrap | pre-line | inherit
Начальное значение: normal
Область применения: все элементы (CSS2.1); элементы уровня блока (CSS1 и CSS2)
Наследование: нет
Вычисляемое значение: как задано

width

Определяет ширину области содержимого элемента, вне которой добавляются отступы, рамки и поля. Это свойство игнорируется для строковых незамещаемых элементов. Отрицательные значения длины и процентных соотношений не допускаются.

Значения: <длина> | <процентное значение> | auto | inherit
Начальное значение: auto
Область применения: блочные и замещаемые элементы
Наследование: нет
Процентные значения: относительно ширины блока-контейнера
Вычисляемое значение: для значения auto и процентных значений – как задано; в противном случае – абсолютная длина, только если свойство не применяется к элементу (тогда auto)

word-spacing

Определяет расстояние, на которое должны отстоять друг от друга слова элемента. Для этого свойства термин «слово» определяется как строка символов, окруженная пробелами. Значения длины модифицируют обычные расстояния между словами; таким образом, значение normal является синонимом 0. Отрицательные значения длины допускаются и задают более плотное размещение слов.

Значения: <длина> | normal | inherit
Начальное значение: normal
Область применения: все элементы
Наследование: да
Вычисляемое значение: для normal – абсолютная длина 0; в противном случае – абсолютная длина

z-index

Это свойство задает размещение позиционированного элемента относительно оси *z*, которая является осью, расположенной перпендикулярно области экрана и направленной в сторону пользователя, т. е. положительные значения приближают элемент, а отрицательные – удаляют его от пользователя.

Значения: <целое> | auto | inherit
Начальное значение: auto
Область применения: позиционированные элементы
Наследование: нет
Вычисляемое значение: как задано

Таблицы

border-collapse

Это свойство определяет модель компоновки, применяемой при планировании рамок таблицы, т. е. рамок ячеек, строк и т. д. Это свойство применяется только к таблицам, но наследуется всеми элементами внутри таблицы.

Значения: collapse | separate | inherit
Начальное значение: separate
Область применения: элементы, свойство display которых имеет значение table или inline-table
Наследование: да
Вычисляемое значение: как задано
Примечание: в CSS2 применяемым по умолчанию значением было collapse

border-spacing

Определяет расстояние между рамками ячеек в модели отдельных рамок. Первое из двух значений длины соответствует промежутку по горизонтали, второе – промежутку по вертикали. Это свойство игнорируется, если только для border-collapse не задано значение separate. Это свойство применяется только к таблицам, но наследуется всеми элементами таблицы.

Значения: <длина> <длина>? | inherit
Начальное значение: 0
Область применения: элементы, свойство display которых имеет значение table или inline-table
Наследование: да
Вычисляемое значение: две абсолютные длины

Примечание: это свойство игнорируется, если значение `border-collapse` не равно `separate`

caption-side

Определяет размещение основного заголовка таблицы относительно блока таблицы. Основной заголовок генерируется так, как будто представляет собою блочный элемент, размещенный сразу перед таблицей (или после нее).

Значения: `top | bottom`
Начальное значение: `top`
Область применения: элементы, свойство `display` которых имеет значение `table-caption`
Наследование: нет
Вычисляемое значение: как задано
Примечание: значения `left` и `right` появились в CSS2, но были изъяты из CSS2.1 из-за недостаточно широкой поддержки

empty-cells

Определяет представление ячеек таблицы, в которых нет содержимого. Если свойство имеет значение `show`, рамки и фон ячеек отрисовываются. Это свойство игнорируется, если значение свойства `border-collapse` не равно `separate`.

Значения: `show | hide | inherit`
Начальное значение: `show`
Область применения: элементы, свойство `display` которых имеет значение `table-cell`
Наследование: да
Вычисляемое значение: как задано
Примечание: это свойство игнорируется, если значение свойства `border-collapse` не равно `separate`

table-layout

Это свойство предназначено для определения алгоритма, применяемого при компоновке таблицы. Алгоритм фиксированной компоновки выполняется быстрее, но обладает меньшей гибкостью, тогда как алгоритм автоматической компоновки медленнее, но результат его работы больше похож на традиционные HTML-таблицы.

Значения: `auto | fixed | inherit`
Начальное значение: `auto`
Область применения: элементы, свойство `display` которых имеет значение `table` или `inline-table`

Наследование: да
Вычисляемое значение: как задано

Устройства с постраничной разбивкой

orphans

Определяет минимальное количество строк текста элемента, которые могут быть оставлены внизу страницы. Может влиять на размещение разрывов страниц в элементе.

Значения: <целое> | inherit
Начальное значение: 2
Область применения: блочные элементы
Наследование: да
Вычисляемое значение: как задано

page-break-after

Объявляет о необходимости размещения разрыва страницы после элемента. Можно ввести принудительные разрывы с помощью значения always, но нельзя гарантировать отсутствие разрывов; максимум, что может сделать автор, – попросить агента пользователя по возможности избегать (avoid) вставки разрыва страницы.

Значения: auto | always | avoid | left | right | inherit
Начальное значение: auto
Область применения: неперемещаемые блочные элементы, свойство position которых имеет значение relative или static
Наследование: нет
Вычисляемое значение: как задано

page-break-before

Объявляет о необходимости размещения разрыва страницы перед элементом. Можно ввести принудительные разрывы с помощью значения always, но нельзя гарантировать отсутствия разрывов; максимум, что может сделать автор, – попросить агента пользователя по возможности избегать (avoid) вставки разрыва страницы.

Значения: auto | always | avoid | left | right | inherit
Начальное значение: auto
Область применения: неперемещаемые блочные элементы, свойство position которых имеет значение relative или static
Наследование: нет
Вычисляемое значение: как задано

page-break-inside

Объявляет о размещении разрывов страниц в элементе. Поскольку элемент может быть выше блока страницы, нельзя гарантировать отсутствие разрывов; максимум, что может сделать автор, – попросить агента пользователя по возможности избегать (`avoid`) вставки разрыва страницы.

Значения: `auto` | `avoid` | `inherit`
Начальное значение: `auto`
Область применения: неперемещаемые блочные элементы, свойство `position` которых имеет значение `relative` или `static`
Наследование: да
Вычисляемое значение: как задано

widows

Определяет минимальное число строк текста элемента, которые могут быть оставлены сверху страницы. Может влиять на размещение разрывов страниц в элементе.

Значения: <целое> | `inherit`
Начальное значение: `2`
Область применения: элементы уровня блока
Наследование: да
Вычисляемое значение: как задано

Изъятые из CSS2.1

Следующие свойства появились в CSS2, но были изъяты из CSS2.1 из-за отсутствия широкой поддержки. Они не содержат информации о вычисляемом значении, поскольку вычисляемые значения впервые были точно определены в CSS2.1.

Стили визуального представления

font-size-adjust

Цель этого свойства – обеспечить авторам возможность организовывать масштабирование шрифтов так, чтобы подставляемые шрифты не слишком отличались от тех, которые задал автор, даже если подставляемый шрифт имеет иную x-высоту. Обратите внимание, что этого свойства нет в CSS2.1.

Значения: <число> | `none` | `inherit`
Начальное значение: `none`
Область применения: все элементы

Наследование: да

font-stretch

С помощью этого свойства глифы символов данного шрифта можно сделать шире или уже; в идеале это осуществляется путем выбора сжатых или расширенных гарнитур из семейства шрифтов. Обратите внимание, что этого свойства нет в CSS2.1.

Значения: `normal | wider | narrower | ultra-condensed | extra-condensed | condensed | semi-condensed | semi-expanded | expanded | extra-expanded | ultra-expanded | inherit`

Начальное значение: `normal`

Область применения: все элементы

Наследование: да

marker-offset

Это свойство определяет расстояние между ближайшим краем рамки блока маркера и блоком ассоциированного с ним элемента.

Значения: <длина> | `auto | inherit`

Начальное значение: `auto`

Область применения: элементы, свойство `display` которых имеет значение `marker`

Наследование: нет

Примечание: в CSS2.1 это свойство вышло из употребления, и, скорее всего, оно не вернется в CSS3. Это же касается и значения `marker` свойства `display`. На момент написания данной книги ситуация складывается так, что для получения этих эффектов будут применяться другие механизмы

text-shadow

Позволяет задавать одну или несколько теней для текста элемента. Первые два значения длины в определении тени задают горизонтальное и вертикальное смещения от текста элемента соответственно. Третья длина определяет радиус размытия. Обратите внимание, что этого свойства нет в CSS2.1.

Значения: `none | [`<цвет> `||` <длина> <длина> `<<`длина`>? , ]* [`<цвет> `||` <длина> <длина> <длина>`?] | inherit`

Начальное значение: `none`

Область применения: все элементы

Наследование: нет

Устройства с постраничной разбивкой

marks

Это свойство определяет, должны ли «метки выравнивания» (также называемые приводочными метками или метками совмещения) размещаться вне области содержимого, но в области печати. Точно размещение и генерирование визуального представления меток не определено. Обратите внимание, что этого свойства нет в CSS2.1.

Значения: [crop || cross] | none | inherit
Начальное значение: none
Область применения: контекст страницы
Наследование: нет данных

page

Это свойство в сочетании со свойством size определяет конкретный тип страницы, который должен задаваться при распечатке документа. Обратите внимание, что этого свойства нет в CSS2.1.

Значения: <идентификатор> | inherit
Начальное значение: auto
Область применения: блочные элементы
Наследование: да

size

Это свойство позволяет авторам объявлять размер и ориентацию блока страницы, используемого при распечатке элемента. Оно может применяться в сочетании со свойством page, хотя это не обязательно. Обратите внимание, что этого свойства нет в CSS2.1.

Значения: <длина>{1,2} | auto | portrait | landscape | inherit
Начальное значение: auto
Область применения: область страницы
Наследование: нет

Стили аудиопредставления

azimuth

Это свойство задает угол в горизонтальной плоскости (также называемой горизонтом), под которым расположен предполагаемый источник звука. В сочетании со свойством elevation применяется для размещения источника звука в гипотетической точке сферы, в центре которой находится пользователь.

Значения:	<угол> | [[left-side | far-left | left | center-left | center | center-right | right | far-right | right-side] || behind] | leftwards | rightwards | inherit
Начальное значение:	center
Область применения:	все элементы
Наследование:	да
Вычисляемое значение:	нормированный угол

cue

Это сокращенная форма записи свойств, позволяющая автору задавать предупредительные сигналы, которые предшествуют и следуют за генерируемым аудиопредставлением содержимого элемента. «Предупредительный сигнал» – это нечто сродни аудиопиктограмме.

Значения:	[<cue-before> || <cue-after>] | inherit
Начальное значение:	none
Область применения:	все элементы
Наследование:	нет
Вычисляемое значение:	см. отдельные свойства (cue-before и др.)

cue-after

Это свойство позволяет задавать предупредительные сигналы, которые следуют за генерируемым аудиопредставлением содержимого элемента.

Значения:	<uri> | none | inherit
Начальное значение:	none
Область применения:	все элементы
Наследование:	нет
Вычисляемое значение:	для значения <uri> – абсолютный URI; в противном случае – none

cue-before

Это свойство позволяет автору задавать предупредительные сигналы, которые предшествуют генерируемому аудиопредставлению содержимого элемента.

Значения:	<uri> | none | inherit
Начальное значение:	none
Область применения:	все элементы
Наследование:	нет
Вычисляемое значение:	для значения <uri> – абсолютный URI; в противном случае – none

elevation

Это свойство задает угол выше или ниже горизонтальной плоскости (также называемой горизонтом), под которым расположен предполагаемый источник звука. В сочетании со свойством azimuth применяется для размещения источника звука в точке на гипотетической сфере, в центре которой находится пользователь.

Значения: <угол> | below | level | above | higher | lower | inherit
Начальное значение: level
Область применения: все элементы
Наследование: да
Вычисляемое значение: нормированный угол

pause

Это сокращенная форма записи свойств, позволяющая автору определять паузы, которые предшествуют и следуют за генерируемым аудиопредставлением содержимого элемента. «Пауза» – это интервал, в котором содержимое не воспроизводится, хотя звуковое сопровождение может звучать.

Значения: [[<время> | <процентное значение>]{1,2}] | inherit
Начальное значение: 0
Область применения: все элементы
Наследование: нет
Вычисляемое значение: см. отдельные свойства (pause-before и др.)

pause-after

Это свойство позволяет определять продолжительность паузы, которая следует за генерируемым аудиопредставлением содержимого элемента.

Значения: <время> | <процентное значение> | inherit
Начальные значения: 0
Область применения: все элементы
Наследование: нет
Вычисляемое значение: абсолютное значение времени

pause-before

Это свойство позволяет автору определять продолжительность паузы, которая предшествует генерируемому аудиопредставлению содержимого элемента.

Значения: <время> | <процентное значение> | inherit
Начальные значения: 0

Область применения: все элементы
Наследование: нет
Вычисляемое значение: абсолютное значение времени

pitch

Определяет среднюю высоту (частоту) голоса, воспроизводящего содержимое элемента. Средняя высота голоса сильно зависит от семейства голосов.

Значения: <частота> | x-low | low | medium | high | x-high | inherit
Начальное значение: medium
Область применения: все элементы
Наследование: да
Вычисляемое значение: абсолютное значение частоты

pitch-range

Это свойство определяет отклонения от средней частоты голоса, воспроизводящего содержимое элемента. Чем больше отклонения, тем более «оживленным» будет казаться голос.

Значения: <число> | inherit
Начальное значение: 50
Область применения: все элементы
Наследование: да
Вычисляемое значение: как задано

play-during

Задает звук, на фоне которого воспроизводится содержимое элемента. Звук может сочетаться с другими фоновыми звуками (задается с помощью свойства play-during элемента-предка) или заменять другие звуки во время воспроизведения элемента.

Значения: <uri> | [mix || repeat]? | auto | none | inherit
Начальное значение: auto
Область применения: все элементы
Наследование: нет
Вычисляемое значение: для значения <uri> – абсолютный URI; в противном случае – как задано

richness

Это свойство задает «яркость» голоса, озвучивающего содержимое элемента. Чем ярче голос, тем большую аудиторию он может охватить.

Значения: <число> | inherit

Начальное значение: 50
Область применения: все элементы
Наследование: да
Вычисляемое значение: как задано

speak

Определяет, как будет происходить генерирование аудиопредставления содержимого элемента, и будет ли оно воспроизводиться вообще. Значение spell-out обычно применяется для акронимов и аббревиатур, например W3C или CSS. Если значение равно none, элемент пропускается (для его воспроизведения время не выделено).

Значения: normal | none | spell-out | inherit
Начальное значение: normal
Область применения: все элементы
Наследование: да
Вычисляемое значение: как задано

speak-header

Определяет, будет ли содержимое заголовков таблицы произносится перед каждой ячейкой, ассоциированной с этим заголовком, или только в случае, если заголовок, ассоциированный с ячейкой, отличается от заголовка, ассоциированного с предыдущей ячейкой, содержимое которой было воспроизведено.

Значения: once | always | inherit
Начальное значение: once
Область применения: элементы, содержащие информацию заголовка таблицы
Наследование: да
Вычисляемое значение: как задано

speak-numeral

Это свойство определяет, как проговариваются числа при генерировании аудиопредставления.

Значения: digits | continuous | inherit
Начальное значение: continuous
Область применения: все элементы
Наследование: да
Вычисляемое значение: как задано

speak-punctuation

Это свойство определяет, как при генерировании аудиопредставления проговариваются знаки препинания. Значение `code` обусловливает произнесение названий знаков препинания.

Значения: `code` | `none` | `inherit`
Начальное значение: `none`
Область применения: все элементы
Наследование: да
Вычисляемое значение: как задано

speech-rate

Задает среднюю скорость, с которой произносятся слова при воспроизведении содержимого элемента.

Значения: <число> | `x-slow` | `slow` | `medium` | `fast` | `x-fast` | `faster` | `slower` | `inherit`
Начальное значение: `medium`
Область применения: все элементы
Наследование: да
Вычисляемое значение: конкретное число

stress

Это свойство задает акценты, расставляемые голосом в воспроизводимой информации. Эти акценты, в свою очередь, определяются знаками ударения языка.

Значения: <число> | `inherit`
Начальное значение: `50`
Область применения: все элементы
Наследование: да
Вычисляемое значение: как задано

voice-family

Это свойство применяется для определения семейства голосов, которые будут использоваться при генерировании аудиопредставления содержимого элемента. Его можно сравнить со свойством `font-family`. Допустимые базовые голоса: `male`, `female` и `child`.

Значения: [[<конкретный-голос> | <базовый-голос>],]* [<конкретный-голос> | <базовый-голос>] | `inherit`
Начальное значение: зависит от агента пользователя
Область применения: все элементы

Наследование: да
Вычисляемое значение: как задано

volume

Задает средний уровень громкости воспроизведения содержимого. Возможны довольно большие отклонения в ту или иную сторону от значения уровня громкости, заданного этим свойством. Обратите внимание, что значение `0` не аналогично `silent`.

Значения: <число> | <процентное значение> | `silent` | `x-soft` | `soft` | `medium` | `loud` | `x-loud` | `inherit`
Начальное значение: `medium`
Область применения: все элементы
Наследование: да
Вычисляемое значение: конкретное число

B

Обзор селекторов, псевдоклассов и псевдоэлементов

Селекторы

Универсальный селектор

Этот селектор сопоставляется с именем любого элемента в языке документа. Если в правиле селектор явно не указан, подразумевается универсальный селектор.

Шаблон: `*`

Примеры:

```
* {color: red;}
div * p {color: blue;}
```

Селектор типа

Этот селектор сопоставляется с одноименным элементом в языке документа. Выбирается каждый экземпляр элемента с заданным именем. (В CSS1 их называют селекторами элементов.)

Шаблон: `элемент1`

Примеры:

```
body {background: #FFF;}
p {font-size: 1em;}
```

Селектор потомков

Позволяет автору выбирать элемент на основании его статуса как потомка другого элемента. Соответствующим элементом может быть дочерний элемент элемента-предка, дочерний элемент второго порядка,

дочерний элемент третьего порядка и т. д. (CSS1 называет их контекстными селекторами.)

Шаблон: элемент1 элемент2

Примеры:

```
body h1 {font-size: 200%;}
table tr td div ul li {color: purple;}
```

Селектор дочерних элементов

Этот тип селекторов применяется для сопоставления элемента на основании его статуса дочернего элемента другого элемента. В данном случае ограничение более жесткое, поскольку выбраны будут только дочерние элементы.

Шаблон: элемент1 > элемент2

Примеры:

```
div > p {color: cyan;}
ul > li {font-weight: bold;}
```

Селектор сестринских элементов

Позволяет автору выбирать элемент, который является сестринским элементом другого элемента. Любой текст между двумя элементами игнорируется; имеют значение только элементы и их расположение в дереве документа.

Шаблон: элемент1 + элемент2

Примеры:

```
table + p {margin-top: 2.5em;}
h1 + * {margin-top: 0;}
```

Селектор классов

В языках, допускающих его, таких как HTML, SVG и MathML, селектор класса, записываемый точечной нотацией, может применяться для выбора элементов, класс которых имеет определенное значение или значения. Имя значения класса должно следовать сразу после точки. Могут быть объединены несколько значений классов. Если перед точкой нет имени элемента, селектор сопоставляется со всеми элементами, содержащими это значение класса.

Шаблоны: элемент1.имя_класса элемент1.имя_класса1.имя_класса2

Примеры:

```
p.urgent {color: red;}
a.external {font-style: italic;}
.example {background: olive;}
```

Селектор идентификатора

В языках, допускающих его, таких как HTML, селектор идентификатора, использующий «запись со знаком фунта», может применяться для выбора элементов, идентификатор которых имеет определенное значение или значения. Имя значения идентификатора должно следовать сразу за символом #. Если перед символом # нет имени элемента, селектор сопоставляется со всеми элементами, содержащими указанное значение идентификатора.

Шаблон: `элемент1#имя_идентификатора`

Примеры:

```
h1#page-title {font-size: 250%;}
body#home {background: silver;}
#example {background: lime;}
```

Простой селектор атрибутов

Позволяет авторам выбирать любой элемент на основании наличия у него указанного атрибута независимо от его значения.

Шаблон: `элемент1[атрибут]`

Примеры:

```
a[rel] {border-bottom: 3px double gray;}
p[class] {border: 1px dotted silver;}
```

Селектор атрибутов с конкретным значением

Позволяет авторам выбирать любой элемент на основании наличия у него указанного атрибута, имеющего заданное значение.

Шаблон: `элемент1[атрибут="значение"]`

Примеры:

```
a[rel="Home"] {font-weight: bold;}
p[class="urgent"] {color: red;;}
```

Селектор частичного значения атрибутов

Позволяет авторам выбирать любой элемент на основании части разделенного пробелами значения атрибута. Заметьте, что `[class~="value"]` эквивалентно `.value` (см. выше).

Шаблон: `элемент1[атрибут~="значение"]`

Примеры:

```
a[rel~="friend"] {text-transform: uppercase;}
p[class~="warning"] {background: yellow;}
```

Селектор начальной подстроки значения атрибута

Позволяет выбирать любой элемент по подстроке, с которой начинается значение атрибута.

Шаблон: `элемент1[attr^="подстрока"]`

Примеры:

```
a[href^="/blog"] {text-transform: uppercase;}
p[class^="test-"] {background: yellow;}
```

Селектор конечной подстроки значения атрибута

Позволяет выбирать любой элемент по подстроке, которой оканчивается значение атрибута.

Шаблон: `элемент1[attr$="подстрока"]`

Пример:

```
a[href$=".pdf"] {font-style: italic;}
```

Селектор произвольной подстроки значения атрибута

Позволяет выбирать любой элемент по подстроке, находящейся в любом месте значения атрибута.

Шаблон: `элемент1[attr*="подстрока"]`

Примеры:

```
a[href*="oreilly.com"] {font-weight: bold;}
div [class*="port"] {border: 1px solid red;}
```

Селектор языковых атрибутов

Позволяет авторам выбирать любой элемент, значение атрибута `lang` которого представлено в виде разделенного дефисами списка значений, начинающегося со значения, приведенного в селекторе.

Шаблон: `элемент1[lang|="код_языка"]`

Примеры:

```
html[lang|="en"] {color: gray;}
```

Псевдоклассы и псевдоэлементы

:active

Применяются к элементам в течение того времени, когда они активны. Самый общий пример – щелчок по гиперссылке HTML-документа: ссылка активна в тот момент времени, когда удерживается кнопка мыши. Существуют и другие способы активировать элементы; теоретически активными могут быть и другие элементы, хотя CSS не определяет этого.

Тип: псевдокласс

Область применения: активный элемент

Примеры:

```
a:active {color: red;}
*:active {background: blue;}
```

:after

Позволяет автору вводить генерируемое содержимое в конце содержимого элемента. По умолчанию псевдоэлемент является строковым элементом, но это можно изменить с помощью свойства `display`.

Тип: псевдоэлемент

Генерирует: псевдоэлемент, содержащий генерируемое содержимое, помещенное после содержимого элемента

Примеры:

```
a.external:after {content: " " url(/icons/globe.gif);}
p:after {content: " | ";}
```

:before

Позволяет автору вводить генерируемое содержимое в начале содержимого элемента. По умолчанию псевдоэлемент является строковым элементом, но это можно изменить с помощью свойства `display`.

Тип: псевдоэлемент

Генерирует: псевдоэлемент, в котором генерируемое содержимое помещено перед содержимым элемента

Примеры:

```
a[href]:before {content: "[LINK] ";}
p:before {content: attr(class);}
```

:first-child

С помощью этого псевдокласса выбирается элемент, являющийся первым дочерним элементом другого элемента. Например, выражение `p:first-child` будет выбирать любой элемент `p`, представляющий собою первый дочерний элемент какого-нибудь другого элемента. По этому выражению *не будет* выбираться, как обычно предполагают, любой элемент, являющийся первым дочерним элементом абзаца; для этого автор написал бы `p > *:first-child`.

Тип: псевдокласс

Область применения: любой элемент, являющийся первым дочерним элементом другого элемента

Примеры:

```
body *:first-child {font-weight: bold;}
p:first-child {font-size: 125%;}
```

:first-letter

Предназначен для оформления первой буквы элемента. Любые предшествующие знаки пунктуации должны быть оформлены вместе с первой буквой. В некоторых языках имеются сочетания букв, которые должны рассматриваться как один символ, и агент пользователя может применять стиль первой буквы к ним обоим. До CSS2.1 `:first-letter` мог использоваться только для блочных элементов. CSS2.1 распространяет его область применения на все элементы. Набор свойств, которые могут применяться к первой букве, ограничен.

Тип: псевдоэлемент

Генерирует: псевдоэлемент, содержащий первую букву элемента

Примеры:

```
h1:first-letter {font-size: 166%;}
a:first-letter {text-decoration: underline;}
```

:first-line

Предназначен для оформления первой строки текста элемента независимо от количества содержащихся в ней слов. Элемент `:first-line` может применяться только к блочным элементам. Набор свойств, которые могут применяться к первой строке, ограничен.

Тип: псевдоэлемент

Генерирует: псевдоэлемент, содержащий отформатированную первую строку элемента

Примеры:

```
p.lead:first-line {font-weight: bold;}
```

:focus

Применяется к элементу в тот период времени, когда ему принадлежит фокус ввода. Один пример из HTML – текстовое поле ввода, в котором находится курсор; т. е. когда пользователь начинает ввод с клавиатуры, текст будет вводиться в это поле. Получать фокус могут и другие элементы, такие как гиперссылки, хотя CSS не определяет, какие элементы имеют фокус.

Тип: псевдокласс

Область применения: элемент, получивший фокус ввода

Примеры:

```
a:focus {outline: 1px dotted red;}
input:focus {background: yellow;}
```

:hover

Применяется к элементу в тот период времени, когда над ним «зависает» указатель мыши. Зависание определяется как указание пользователем на элемент без его активации. Самый распространенный пример – перемещение указателя мыши по гиперссылкам в HTML-документе. Теоретически «зависание» может быть реализовано и для других элементов, хотя CSS не определяет, для каких.

Тип: псевдокласс

Область применения: элемент, над которым «зависает» указатель указывающего устройства

Примеры:

```
a[href]:hover {text-decoration: underline;}
p:hover {background: yellow;}
```

:lang

Выбирает элементы на основании кодировки языка. Подобная информация о языке должна содержаться внутри документа или в противном случае быть каким-либо образом с ним связанной; она не может быть назначена из CSS. Обработка `:lang` аналогична селектору атрибутов `|=`.

Тип: псевдокласс

Область применения: любой элемент, с которым связана информация о его языке

Примеры:

```
html:lang(en) {background: silver;}
*:lang(fr) {quotes: '“ ' ' ”';}
```

:link

Применяется к ссылке на URI, который не был посещен; т. е. URI, на который указывает ссылка, нет в журнале агента пользователя. Это состояние и состояние `:visited` являются взаимоисключающими.

Тип: псевдокласс

Область применения: ссылка на другой ресурс, который не был посещен

Примеры:

```
a:link {color: blue;}
*:link {text-decoration: underline;}
```

:visited

Применяется к ссылкам на URI, который был посещен; т. е. URI, на который указывает ссылка, есть в журнале агента пользователя. Это состояние и состояние `:link` являются взаимоисключающими.

Тип: псевдокласс

Область применения: ссылка на другой ресурс, который уже был посещен

Примеры:

```
a:visited {color: purple;}
*:visited {color: gray;}
```

C

Пример таблицы стилей HTML 4

Приведенная ниже таблица стилей адаптирована из приложения D спецификации CSS2.1. Важно отметить следующее: во-первых, хотя в CSS2.1 говорится, что «разработчикам рекомендуется применять [ее] в качестве таблицы стилей по умолчанию в своих реализациях», это не всегда возможно. Например, в этой таблице стилей имеется правило, утверждающее:

```
ol, ul, dir, menu, dd
                {margin-left: 40px;}
```

Оно описывает традиционное смещение списков на 40 пикселов и реализует его с помощью левого поля. Однако некоторые броузеры используют 40-пиксельный *отступ* слева вместо поля в уверенности, что это лучшее решение. (Подробности можно найти в главе 12.) Таким образом, *нельзя полагаться на данную таблицу как на абсолютно верную стандартную таблицу стилей для любого агента пользователя*. Она предоставляется преимущественно в иллюстративных целях и как средство обучения.

Во-вторых, важно отметить, что в этой таблице стилей не все HTML-элементы описаны полностью, потому что стандарт CSS до сих пор не настолько детализирован. Классический пример – элементы форм, такие как кнопки отправки формы, которые представляют собой замещаемые элементы, и таким образом нижний край их блока должен выравниваться с базовой линией. Однако авторы ожидают, что текст кнопки будет выровнен по базовой линии текста, находящегося в этой же строке. Это вполне обоснованное ожидание, но CSS не может (на момент написания данной книги) описывать такое поведение. Поэтому следующее правило – это все, что сказано о таких элементах:

```
button, textarea, input, object, select, img {
                  display:inline-block;}
```

Все остальные вопросы форматирования подобных элементов оставлены на усмотрение агента пользователя.

Помня обо всех этих предостережениях, взгляните на таблицу стилей (с небольшими изменениями форматирования) из спецификации CSS2. Все изменения, не относящиеся к форматированию, отмечены в комментариях.

```
address, blockquote, body, dd, div, dl, dt, fieldset, form,
frame, frameset, h1, h2, h3, h4, h5, h6, noframes,
ol, p, ul, center, dir, hr, menu, pre {
                 display: block;}
li              {display: list-item;}
head            {display: none;}
table           {display: table;}
tr              {display: table-row;}
thead           {display: table-header-group;}
tbody           {display: table-row-group;}
tfoot           {display: table-footer-group;}
col             {display: table-column;}
colgroup        {display: table-column-group;}
td, th          {display: table-cell;}
caption         {display: table-caption;}
th              {font-weight: bolder; text-align: center;}
caption         {text-align: center;}
body            {padding: 8px; line-height: 1.12em;}
h1              {font-size: 2em; margin: .67em 0;}
h2              {font-size: 1.5em; margin: .75em 0;}
h3              {font-size: 1.17em; margin: .83em 0;}
h4, p, blockquote, ul, fieldset, form, ol, dl, dir, menu {
                 margin: 1.12em 0;}
h5              {font-size: .83em; margin: 1.5em 0;}
h6              {font-size: .75em; margin: 1.67em 0;}
h1, h2, h3, h4, h5, h6, b, strong {
                 font-weight: bolder;}
blockquote      {margin-left: 40px; margin-right: 40px;}
i, cite, em, var, address  {
                 font-style: italic;}
pre, tt, code, kbd, samp {
                 font-family: monospace;}
pre             {white-space: pre;}
button, textarea, input, object, select, img {
                 display:inline-block;}
big             {font-size: 1.17em;}
small, sub, sup {font-size: .83em;}
sub             {vertical-align: sub;}
sup             {vertical-align: super;}
s, strike, del  {text-decoration: line-through;}
hr              {border: 1px inset;}
ol, ul, dir, menu, dd  {
                 margin-left: 40px;}
```

```
ol              {list-style-type: decimal;}
ol ul, ul ol, ul ul, ol ol {
                 margin-top: 0; margin-bottom: 0;}
u, ins          {text-decoration: underline;}
br:before       {content: "\A";}
center          {text-align: center;}
abbr, acronym   {font-variant: small-caps; letter-spacing: 0.1em;}
:link,:visited {text-decoration: underline;}
:focus          {outline: thin dotted invert;}

/* Начинаем настройку двунаправленности (без изменений) */
BDO[DIR="ltr"]  {direction: ltr; unicode-bidi: bidi-override;}
BDO[DIR="rtl"]  {direction: rtl; unicode-bidi: bidi-override;}

*[DIR="ltr"]    {direction: ltr; unicode-bidi: embed;}
*[DIR="rtl"]    {direction: rtl; unicode-bidi: embed;}

@media print {
  h1            {page-break-before: always;}
  h1, h2, h3, h4, h5, h6 {
                 page-break-after: avoid;}
  ul, ol, dl    {page-break-before: avoid;}

@media aural {  /* вместо 'speech', которое не определено в CSS2 */
  h1, h2, h3, h4, h5, h6 {
                 voice-family: paul, male; stress: 20; richness: 90;}
  h1            {pitch: x-low; pitch-range: 90;}
  h2            {pitch: x-low; pitch-range: 80;}
  h3            {pitch: low; pitch-range: 70;}
  h4            {pitch: medium; pitch-range: 60;}
  h5            {pitch: medium; pitch-range: 50;}
  h6            {pitch: medium; pitch-range: 40;}
  li, dt, dd    {pitch: medium; richness: 60;}
  dt            {stress: 80;}
  pre, code, tt {pitch: medium; pitch-range: 0; stress: 0; richness: 80;}
  em            {pitch: medium; pitch-range: 60; stress: 60; richness: 50;}
  strong        {pitch: medium; pitch-range: 60; stress: 90; richness: 90;}
  dfn           {pitch: high; pitch-range: 60; stress: 60;}
  s, strike     {richness: 0;}
  i             {pitch: medium; pitch-range: 60; stress: 60; richness: 50;}
  b             {pitch: medium; pitch-range: 60; stress: 90; richness: 90;}
  u             {richness: 0;}
  a:link        {voice-family: harry, male;}
  a:visited     {voice-family: betty, female;}
  a:active      {voice-family: betty, female; pitch-range: 80; pitch: x-high;}
}
```

Алфавитный указатель

Специальные символы

A

B

C

D

E

F

M

N

O

P

Q

R

S

Б

В

Г

Д

Е

З

И

К

М

Н

О

П

P

C

Т

У

Ф

Ц

Ч

Ш

Э

Я

По договору между издательством «Символ-Плюс» и Интернет-магазином «Books.Ru – Книги России» единственный легальный способ получения данного файла с книгой ISBN 5-93286-107-X, название «CSS – каскадные таблицы стилей. Подробное руководство» – покупка в Интернет-магазине «Books.Ru – Книги России». Если Вы получили данный файл каким-либо другим образом, Вы нарушили международное законодательство и законодательство Российской Федерации об охране авторского права.

Вам необходимо удалить данный файл, а также сообщить издательству «Символ-Плюс» (piracy@symbol.ru), где именно Вы получили данный файл.