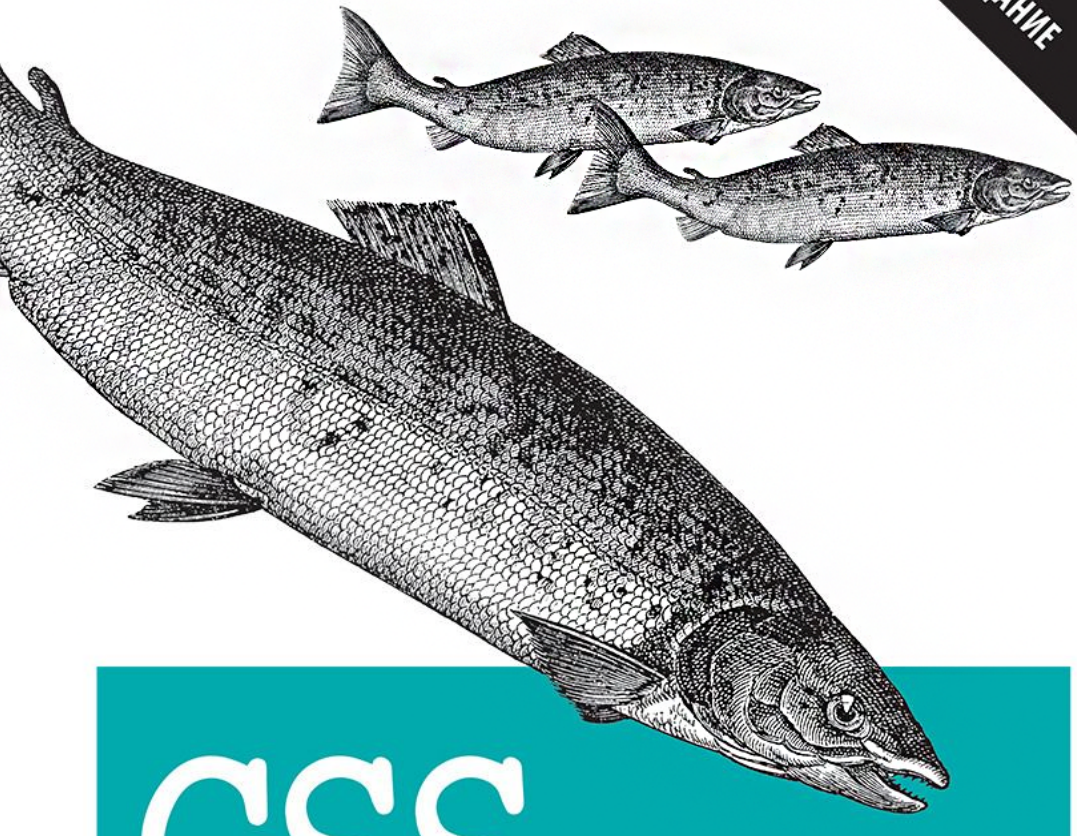
O'REILLY®

4-Е ИЗДАНИЕ



CSS

ПОЛНЫЙ СПРАВОЧНИК

ВИЗУАЛЬНОЕ ФОРМАТИРОВАНИЕ ВЕБ-СТРАНИЦ

Эрик Мейер, Эстелл Уэйл

4-е издание

CSS
Полный справочник

Визуальное форматирование веб-страниц

Fourth Edition

CSS: The Definitive Guide

Visual Presentation for the Web

Eric A. Meyer and Estelle Weyl

Beijing • Boston • Farnham • Sebastopol • Tokyo O'REILLY®

4-е издание

CSS
Полный справочник

Визуальное форматирование веб-страниц

Эрик Мейер, Эстелл Уэйл

Москва · Санкт-Петербург
2019

ББК 32.973.26-018.2.75
М45
УДК 681.3.07

Компьютерное издательство "Диалектика"
Перевод с английского *И.В. Василенко*
Под редакцией *В.Р. Гинзбурга*

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

Мейер, Эрик, **Уэйл**, Эстелл.
М45 CSS: полный справочник, 4-е изд. : Пер. с англ. — СПб. : ООО "Диалектика", 2019. — 1088 с. : ил. — Парал. тит. англ.

ISBN 978-5-907114-56-2 (рус.)

ББК 32.973.26-018.2.75

Все названия программных продуктов являются зарегистрированными торговыми марками соответствующих фирм.

Никакая часть настоящего издания ни в каких целях не может быть воспроизведена в какой бы то ни было форме и какими бы то ни было средствами, будь то электронные или механические, включая фотокопирование и запись на магнитный носитель, если на это нет письменного разрешения издательства O'Reilly Media, Inc.

Authorized Russian translation of the English edition of *CSS: The Definitive Guide, 4th Edition* (ISBN 978-1-449-39319-9) © 2018 Eric Meyer, Estelle Weyl.

This translation is published and sold by permission of O'Reilly Media, Inc., which owns or controls all rights to sell the same.

All rights reserved. No part of this book may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording, or by any information storage or retrieval system, without the prior written permission of the copyright owner and the Publisher.

Научно-популярное издание

Эрик Мейер, Эстелл Уэйл

CSS: полный справочник, 4-е издание

Подписано в печать 11.02.2019.
Формат 70x100/16. Гарнитура Times.
Усл. печ. л. 87,7. Уч.-изд. л. 74,05.
Тираж 400 экз. Заказ № 1675.

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

ООО "Диалектика", 195027, Санкт-Петербург, Магнитогорская ул., д. 30, лит. А, пом. 848

ISBN 978-5-907114-56-2 (рус.)
ISBN 978-1-449-39319-9 (англ.)

© 2019 ООО "Диалектика"
© 2018 Eric Meyer, Estelle Weyl

Оглавление

Введение 13

Глава 1. CSS и HTML-документы 21

Глава 2. Селекторы 51

Глава 3. Приоритетность и каскадирование стилей 125

Глава 4. Значения и единицы измерения 141

Глава 5. Шрифты 177

Глава 6. Текстовые свойства 235

Глава 7. Основы визуального оформления документа 285

Глава 8. Поля, отступы, границы и рамки 343

Глава 9. Цвета, фон и градиенты 411

Глава 10. Обтекание и форма элемента 513

Глава 11. Позиционирование 551

Глава 12. Гибкая верстка 591

Глава 13. Верстка по сетке 687

Глава 14. Верстка таблиц 765

Глава 15. Списки и генерируемое содержимое 801

Глава 16. Трансформации 853

Глава 17. Переходы 891

Глава 18. Анимация 927

Глава 19. Фильтры, смешивание цветов, обтравочные контуры и маски 981

Глава 20. Форматирование, зависящее от носителя 1027

Приложение А. Анимируемые свойства 1055

Приложение Б. Базовые свойства 1061

Приложение В. Таблица соответствия цветов 1071

Предметный указатель 1075

Содержание

Об авторах 11
Об изображении на обложке 12

Введение 13

Принятые соглашения 13
Синтаксис описания свойств CSS 14
Файлы примеров и цветные иллюстрации 17
Ждем ваших отзывов! 18
Благодарности 19

Глава 1. CSS и HTML-документы 21

История CSS 21
Элементы 23
Взаимодействие CSS и HTML 28
Таблицы стилей 37
Медиа-запросы 41
Запросы поддержки 47
Резюме 50

Глава 2. Селекторы 51

Базовые правила 51
Группирование 56
Селекторы идентификаторов и классов 61
Селекторы атрибутов 68
Структура документа 78
Селекторы псевдоклассов 87
Псевдоэлементы 119
Резюме 122

Глава 3. Приоритетность и каскадирование стилей 125

Приоритетность 125
Наследование 131
Каскадирование 134
Резюме 140

Глава 4. Значения и единицы измерения 141

Ключевые слова, строки и другие текстовые значения 141
Числовые и процентные значения 148
Длина и расстояние 149
Вычисляемые значения 159
Значения атрибутов 161

Цвета 161
Углы 171
Время и частота 172
Положение 173
Пользовательские переменные 173

Глава 5. Шрифты 177

Семейства шрифтов 177
Правило `@font-face` 182
Насыщенность шрифта 195
Размер шрифта 203
Начертание шрифта 214
Уплотнение и расширение текста 218
Кернинг шрифта 220
Варианты строчных букв 221
Особенности шрифта 224
Генерирование начертаний 226
Свойство `font` 228
Замена шрифтов 232
Резюме 234

Глава 6. Текстовые свойства 235

Отступы и выравнивание 235
Выравнивание по высоте 244
Интервалы между словами и символами 254
Регистр символов 258
Оформление текста 260
Оптимизация отображения текста 264
Текст, отбрасывающий тень 266
Обработка пробелов 267
Разрывы и переносы текстовых строк 271
Направление письма 277
Резюме 284

Глава 7. Основы визуального оформления документа 285

Контейнеры 285
Представление элемента 288
Строчные элементы 312
Резюме 342

Глава 8. Поля, отступы, границы и рамки 343

Контейнеры элемента 343
Поля 346
Границы 356
Внешний контур 395

Отступы 401
Резюме 410

Глава 9. Цвета, фон и градиенты 411

Цвета 411
Фон элемента 416
Градиенты 474
Тень элемента 509
Резюме 512

Глава 10. Обтекание и форма элемента 513

Обтекание 513
Отмена обтекания 530
Обтекание по форме элемента 534
Резюме 549

Глава 11. Позиционирование 551

Общие положения 551
Свойства смещения 553
Высота и ширина 556
Переполнение и обрезка содержимого 560
Сокрытие элемента 562
Абсолютное позиционирование 563
Фиксированное положение 581
Относительное позиционирование 583
Липкое позиционирование 585
Резюме 589

Глава 12 Гибкая верстка 591

Основы верстки flex-контейнеров 591
Flex-контейнеры 598
Упорядочение flex-элементов 616
Flex-контейнер 617
Выравнивание flex-элементов 618
Выравнивание flex-элементов вдоль поперечной оси 627
Свойство `align-self` 634
Выравнивание содержимого 635
Flex-элементы 641
Свойства форматирования отдельных flex-элементов 646
Свойство `flex` 646
Свойство `flex-grow` 648
Свойство `flex-shrink` 655
Свойство `flex-basis` 666
Свойство общего назначения `flex` 676
Порядок следования элементов 682

Глава 13. Верстка по сетке 687
Создание grid-контейнера 687
Основные определения 690
Размещение линий сетки 692
Привязка элементов к сетке 720
Поток grid-элементов 736
Автоматическое размещение линий сетки 741
Свойство настройки сетки общего назначения 744
Интервалы 748
Выравнивание grid-элементов 755
Размещение на слоях и порядок наложения 761
Резюме 763

Глава 14. Верстка таблиц 765
Форматирование таблиц 765
Границы ячеек таблицы 779
Размер таблицы 788
Резюме 799

Глава 15. Списки и генерируемое содержимое 801
Списки 801
Генерируемое содержимое 813
Шаблон счетчика 829
Резюме 851

Глава 16. Трансформации 853
Система координат 853
Трансформация элементов 857
Другие свойства трансформации 877
Резюме 889

Глава 17. Переходы 891
Переходы в CSS 891
Свойства настройки переходов 893
Независимый обратный переход 916
Анимируемые свойства и значения 920
Переход как эффект оформления 925
Печать переходов 925

Глава 18. Анимация 927
Определение ключевых кадров 928
Настройка анимации 929
Селекторы ключевых кадров 930
Анимация элементов 936
Анимация общего назначения 971

Приоритетность и порядок выполнения анимации 975
Эпилептические приступы и потеря ориентации при просмотре анимации 977
События анимации и вендорные префиксы 978
Печать анимации 980

Глава 19. Фильтры, смешивание цветов, обтравочные контуры и маски 981

CSS-фильтры 981
Наложение элементов и смешивание цветов 988
Наложение фоновых слоев 996
Обтравочные контуры и маскирование 1001
Маски 1008
Подгонка объектов 1023

Глава 20. Форматирование, зависящее от носителя 1027

Стилевое форматирование, зависящее от устройства 1027
Бумажные носители 1039
Резюме 1054

Приложение А. Анимируемые свойства 1055

Приложение Б. Базовые свойства 1061

Приложение В. Таблица соответствия цветов 1071

Предметный указатель 1075

Об авторах

Эрик Мейер — всемирно известный эксперт по HTML, CSS и веб-стандартам. Основатель компании Complex Spiral Consulting, один из разработчиков формата презентаций S5 и учредитель серии конференций по веб-дизайну An Event Apart. Автор целого ряда книг, посвященных CSS и веб-дизайну. Проживает в Кливленде, штат Огайо. Ведет собственный сайт `meyerweb.com`.

Эстелл Уэйл — профессиональный веб-разработчик, с 1999 года занимается проектированием сайтов с использованием веб-стандартов. Ведет блог на сайте `standardista.com`, посвященный CSS3, HTML5 и JavaScript, а также инструментам мобильного веб-дизайна. Регулярно выступает на конференциях по всему миру.

Об изображении на обложке

Рыбы, изображенные на обложке книги, — это представители семейства лососевых (*salmonidae*), включающего большое количество видов, наиболее распространенные из которых — семга (атлантический лосось), кета и горбуша (розовый лосось).

Тихоокеанские лососи водятся в прибрежных водах Северного Ледовитого океана и северной части Тихого океана. Основные виды: горбуша, кета, кижуч, нерка и чавыча. Средний вес взрослой особи — от 2 до 13 кг, в зависимости от вида. Тихоокеанский лосось нерестится в основном осенью, для чего заходит в реки, откладывая икринки в углубления в грунте. Мальки вылупляются зимой, к весне вырастая до нескольких сантиметров в длину. В реке или озере мальки проводят год-два, уходя затем на несколько лет в море или океан. Впоследствии тихоокеанский лосось возвращается к местам нереста лишь один раз — для откладывания икры и неизбежной гибели.

Атлантический лосось, или семга, водится в северной части Атлантического океана, вдоль побережий Северной Америки и Европы. Это представитель рода настоящих лососей, насчитывающего множество видов, среди которых — мраморная форель и кумжа. Вес взрослой семги составляет в среднем от 4 до 9 кг. Жизненный цикл атлантического лосося такой же, как и у тихоокеанского, — он также начинается в пресной воде и продолжается взрослением в океане. Но, в отличие от тихоокеанских видов, семга и форель не погибают при нересте, а возвращаются в океан, заходя в реки и озера для откладывания икры еще несколько раз.

Лососи отличаются темно-серебристым окрасом, с четко выделяющимися пятнами на спине и плавниках. Основное питание лосося — планктон, личинки насекомых, креветки и мелкая рыбешка. Идя на нерест, лосось преодолевает множество преград, а путь к родной реке находит по запаху. Отдельные виды лосося имеют озерные и ручьевые формы, т.е. проводят в пресной воде всю жизнь.

Лосось является важной составляющей экосистемы: погибшие после нереста рыбы становятся кормом для других животных и прекрасно удобряют почву берегов рек. К сожалению, из года в год количество рыбы, приходящей на нерест, уменьшается, что вызвано интенсивным выловом и вмешательством человека в среду обитания: постройкой дамб, перекрывающих пути миграции лосося, неконтролируемым сбросом отходов в реки, а также кислотными дождями.

Изображение на обложке сделано с гравюры XIX столетия, которая предоставлена в библиотеке Dover Pictorial Archive.

Введение

В первую очередь книга рассчитана на дизайнеров и разработчиков веб-страниц, нуждающихся в эффективных инструментах оформления документов и управления их содержимым. Для полного понимания материала книги достаточно знакомства с языком HTML 4.0, и чем глубже ваши познания HTML, тем проще вам будет изучать CSS. Не отчаивайтесь, если ваши навыки написания веб-документов далеки от совершенства, — книга будет понятной для разработчиков с базовым уровнем знания языка HTML.

Работа над четвертым изданием книги была завершена еще летом 2017 года, и в ней отражен текущий вариант спецификации CSS. В книге описаны все средства CSS, уже получившие распространение в большинстве современных браузеров, а также инструменты, решение о поддержке которых будет принято в ближайшем будущем. Чего вы в ней точно не найдете, так это сведений о технологиях, разработка которых еще не завершена, и устаревших спецификациях, давно утративших актуальность для разработчиков браузеров.

Принятые соглашения

В книге приняты следующие условные обозначения. (Обязательно ознакомьтесь с синтаксисом описания свойств CSS в следующем разделе, чтобы понять, как он согласуется с этими условными обозначениями.)

Курсив
: Служит для выделения новых терминов и ключевых понятий, без знания которых вам не обойтись.

`Моноширинный шрифт`
: Предназначен для выделения элементов программного кода, таких как имена переменных, функций, операторов и других ключевых слов.

`Моноширинный полужирный шрифт`
: Используется для обозначения вводимых пользователем значений, команд, знаков и цифр там, где это уместно.

`Моноширинный курсивный шрифт`
: Применяется для выделения значений заменяемых параметров в синтаксисе команд, выражений, функций и методов.

Этой пиктограммой помечены советы и рекомендации.

Этой пиктограммой помечены примечания.

Этой пиктограммой помечены предупреждения.

Синтаксис описания свойств CSS

По всей книге встречаются врезки, содержащие подробные сведения о рассматриваемых свойствах CSS, включая допустимые значения. И хотя формат таких врезок почти полностью заимствован из спецификации CSS, он все же требует определенных пояснений.

Допустимые значения свойства указываются следующим образом.

Значение: *<family-name>*#

Значение: *<url>* || *<color>*

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

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

Названия, заключенные в угловые скобки и выделенные курсивом, определяют тип значения или ссылаются на значения другого свойства. Например, свойство `font` может иметь значения, которые исходно принадлежат свойству `font-family`, что в его описании отражается записью *<font-family>*. Подобным образом запись *<color>* указывает на возможность использования названий цветов в качестве значения свойства.

Ключевые слова, введенные моноширинным шрифтом, а также символы / (косая черта) и , (запятая) используются в коде разметки и не требуют заключения в скобки.

В определении значения допускается комбинировать сразу несколько ключевых слов.

- Разделение ключевых слов пробелом требует использования их в значении свойства в строго заданном порядке. Например, запись `help me` предполагает использование данных ключевых слов только в указанном виде.
- Вертикальной чертой (`X | Y`) разделяются взаимоисключающие ключевые слова. В частности, запись `[ X | Y | Z ]` указывает на использование в значении свойства только одного из ключевых слов: `X`, `Y` или `Z`.

- Ключевые слова, разделенные двойной вертикальной чертой (`X || Y`), могут использоваться в значении свойства по отдельности или вместе, причем в произвольной последовательности. Это означает, что возможны такие варианты: `X`, `Y`, `X Y` и `Y X`.
- Двойной амперсанд (`X && Y`) требует включения в значение обоих ключевых слов, причем в произвольном порядке. Это означает, что возможны два варианта: `X Y` и `Y X`.
- Квадратные скобки позволяют группировать ключевые слова, включаемые в значение свойства. В частности, запись `[please || help || me] do this` указывает на то, что каждое из ключевых слов `please`, `help` и `me` включается в значение только один раз, хотя и в произвольном порядке, а выражение `do this` добавляется в конец записи с сохранением исходного формата. Это делает возможными такие варианты, как `please help me do this`, `help me please do this` или `me please help do this`.

Отдельно стоящие значения, как и группы в квадратных скобках, могут дополняться специальными модификаторами.

- Символ `*` (звездочка) означает многократное повторение в значении свойства ключевого слова или группы ключевых слов, расположенных перед ним. Например, запись `bucket*` позволяет добавить ключевое слово `bucket` в значение свойства произвольное количество раз (в том числе ни разу). Строгих ограничений на количество повторений не существует.
- Знак `+` (плюс) указывает на необходимость включения расположенного перед ним символа не менее одного раза. Таким образом, запись `mop+` говорит о том, что ключевое слово `mop` добавляется в значение по крайней мере единожды.
- Символ `#` (решетка) означает, что в значение нужно включить несколько копий ключевого слова, расположенного перед ним, при необходимости разделяя их запятыми. В частности, запись `floor#` указывает на добавление в значение свойства группы ключевых слов `floor, floor, floor, floor` и т.д. Чаще всего данный формат записи применяется при включении в значение сгруппированных элементов или при типизации значения.
- Символ `?` (вопросительный знак) указывает на необязательное использование в значении свойства ключевого слова или группы элементов, расположенных перед ним. Например, запись `[pine tree]?` означает возможность включения в значение ключевых слов `pine tree`. (В случае их использования порядок слов должен сохраняться.)
- Символ `!` (восклицательный знак) обязывает включить в значение свойства ключевое слово или группу ключевых слов, указанных перед ним, даже если предыдущий синтаксис этого не предполагает. Например, запись `[ what? is? happening? ]!` означает необходимость включения в значение одного их трех ключевых слов, перечисленных в квадратных скобках, хотя по отдельности они имеют необязательный статус.

- Числа, заключенные в фигурные скобки ({*M*, *N*}), указывают на включение в значение свойства от *M* до *N* экземпляров ключевого слова или группы ключевых слов, расположенных перед ними. Таким образом, запись `ha{1,3}` означает, что в значение свойства необходимо добавить одно, два или три ключевых слова `ha`.

Рассмотрим несколько примеров использования описанных выше операторов и модификаторов для комбинирования ключевых слов в значении свойства.

`give || me || liberty`

> В значение добавляется по меньшей мере одно из трех ключевых слов. При использовании нескольких ключевых слов их можно располагать в произвольном порядке, например так: `give liberty`, `give me`, `liberty me give` или `give me liberty`.

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

> В начало значения добавляется одно из двух первых ключевых слов (`I` или `am`) либо ни одно из них. Вторая половина значения состоит из ключевого слова `the` или `walrus`, либо их обоих, следующих в произвольном порядке. Допустимые варианты: `I the walrus`, `am walrus the`, `am the`, `I walrus`, `walrus the` и др.

`koo+ ka-choo`

> Перед ключевым словом `ka-choo` добавляется один или несколько экземпляров ключевого слова `koo`. Среди допустимых вариантов: `koo koo ka-choo`, `koo koo koo ka-choo` и `koo ka-choo`. По сути, в одном значении можно использовать произвольное количество ключевых слов `koo`, хотя на практике их число будет ограничено в каждой конкретной реализации CSS.

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

> Позволяет строить фразы наподобие `I love Netscape`, `I really love Microsoft` и т.п. В значение разрешается включать до четырех экземпляров ключевого слова `really`, которые не разделяются запятыми.

`It's a [ mad ]# world`

> В значение добавляется произвольное (не менее одного) количество ключевых слов `mad`, разделяемых запятыми. При использовании всего одного слова `mad` запятая после него не ставится. Таким образом, оба варианта — `It's a mad world` и `It's a mad, mad, mad, mad, mad world` — допустимы.

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

Перед фразой `and Delphi` добавляются две или три произвольные комбинации ключевых слов `Alpha`, `Baker` и `Cray`. Один из возможных вариантов: `Cray, Alpha, and Delphi`. Обратите внимание на добавление запятой после каждого из ключевых слов: ее расположение предопределяется позицией снаружи вложенных квадратных скобок. (Такой способ добавления запятых между ключевыми словами характерен для старых версий CSS, в которых отсутствует поддержка модификатора `#`.)

Файлы примеров и цветные иллюстрации

Примеры программных кодов встречаются в книге повсеместно. Все они доступны для загрузки на сайте автора (`https://meyerweb.github.io/csstdg4figs/`) либо на GitHub (`https://github.com/meyerweb/csstdg4figs`).

Используйте оглавление на начальной странице для перехода к необходимой главе и целевому примеру. На сайте содержатся файлы HTML и CSS, а также графические изображения, необходимые для воссоздания иллюстраций книги. Обязательно ознакомьтесь с файлом *README.md* и другими текстовыми файлами, содержащими сведения о репозитории.

Также архив материалов продублирован на сайте издательства "Диалектика":

`http://go.dialektika.com/CSSDG`

По этому же адресу можно скачать файл цветных иллюстраций, содержащий цветные версии ряда иллюстраций, которые приведены в книге в черно-белом виде. Для доступа к файлам перейдите по ссылке Файлы к книге.

Все доступные для загрузки файлы призваны помочь вам лучше изучить материал книги, поэтому не бойтесь использовать их в собственных целях. Любые файлы, которые вы найдете по указанным адресам, можно смело применять в других проектах или ссылаться на них в учебных пособиях. При этом совсем не обязательно ставить в известность авторов, за исключением случаев коммерческого использования больших фрагментов кода. В частности, использование отдельных фрагментов кода для создания персонального проекта вполне допустимо, а вот для продажи или распространения файлов примеров на любом из носителей необходимо получать разрешение. Аналогичным образом разрешается свободно цитировать фрагменты кода из книги на сайтах, однако для включения больших фрагментов кода в документацию к собственному продукту необходимо запрашивать разрешение.

По возможности ссылайтесь на материалы, взятые из этой книги, если используете их в своих проектах, хотя это не обязательное требование. Если вы не знаете, нужно ли получать специальное разрешение на использование программных кодов, приведенных в книге, в каждом конкретном случае, не поленитесь связаться с издательством по электронной почте (`permissions@oreilly.com`).

Ждем ваших отзывов!

Вы, читатель этой книги, и есть главный ее критик. Мы ценим ваше мнение и хотим знать, что было сделано нами правильно, что можно было сделать лучше и что еще вы хотели бы увидеть изданным нами. Нам интересны любые ваши замечания в наш адрес.

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

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

Наши электронные адреса:

E-mail: `info@williamspublishing.com`

WWW: `http://www.williamspublishing.com`

Благодарности

Книга стала настоящим вызовом для меня как с профессиональной точки зрения, так и по личным причинам. Неоценимую помощь в ее написании мне оказали очень многие люди. Хочется выразить им всем искреннюю благодарность за проделанную работу.

Огромное спасибо создателям CSS: Хокону Виуму Ли и Берту Босу. Благодаря вашему детищу Интернет стал таким, каким он нам всем нравится: неповторимым и самобытным.

Отдельной благодарности заслуживают разработчики, отвечающие за обеспечение современных браузеров средствами поддержки всех описанных в книге технологий.

Хотелось бы поблагодарить также разработчиков Git, программного обеспечения, ответственного за сохранение резервных копий файлов. Его использование позволило восстановить целую главу книги (15, "Списки и генерируемое содержимое"), по неосторожности утерянную на последних этапах подготовки.

Спасибо Саймону Сен-Лорану, поверившему в то, что столь масштабную книгу можно написать в несколько этапов с большими перерывами. Он проявил недюжинное терпение, дожидаясь моего возращения к написанию каждой следующей главы, с завидным упорством поддерживая любые мои предложения и самые смелые идеи.

Огромное спасибо Эстелл Уэйл, моему соавтору, за своевременную помощь в самых сложных ситуациях, профессиональное рецензирование и неоценимый вклад в написание нового издания.

Разобраться во всех тонкостях CSS мне помогало очень много людей. Спасибо всем, кто не пожалел своего времени на объяснение одних и тех же вещей по нескольку раз. С некоторыми из них я знаком по работе над предыдущими изданиями книги, хотя чаще всего помощь требовалась мне при описании новых возможностей CSS. Назову несколько имен: Рэйчел Эндрю, Россен Атаноссов, Тэб Эткинс, Амелия Беллами-Ройдс, Дейв Креймер, Элика Этемад, Джен Симмонс, Сара Суидан, Мел Самнер и Грег Уитворт. Заранее прошу прощения у всех, кому довелось со мной поработать и кого я забыл включить в приведенный выше список.

Хочется выразить благодарность всему сообществу веб-дизайнеров и разработчиков, которые помогали разобраться в самых сложных примерах, а также друзьям, коллегам по работе и случайным знакомым — я перед вами в неоплачиваемом долгу. Спасибо за поддержку и своевременную помощь!

Громадное спасибо моей семье — жене Кэт и детям — Кэролин, Ребекке и Джошуа. Вы оберегаете меня, согреваете своим теплом и вдохновляете на новые свершения! Все, чего мне удалось достичь, — это только благодаря вам.

Эрик Мейер,
Кливленд-Хайтс, штат Огайо,
19 июля 2017 г.

Хочется выразить признательность всем, кто внес свою лепту в совершенствование CSS, а также способствует развитию и популяризации IT-технологий.

Стоит отметить всех тех, кто проделал огромную работу по согласованию стандартов CSS с производителями браузеров и независимыми разработчиками. Без слаженных действий рабочей группы — в прошлом, настоящем и будущем — CSS не получила бы четких спецификаций, технических стандартов и всесторонней поддержки браузерами. Меня бесконечно восхищает процесс принятия решений по каждому свойству, которое включается или исключается из спецификации CSS. Нужно отдать должное Тэбу Эткинсу, Элике Этемад, Дейву Барону, Леони Уотсон и Грегу Уитворту за упорство в разработке спецификации CSS и настойчивость, проявленную при ее согласовании с широким кругом разработчиков, вовлеченных в индустрию (в том числе и со мной).

Отдельных благодарностей удостоиваются все те, кто, независимо от степени занятости в рабочей группе CSS, помогал разобраться в тонкостях спецификации менее осведомленным коллегам. Среди них Сара Драснер, Вал Хэд, Сара Суидан, Крис Койер, Джен Симмонс и Рэйчел Эндрю. Кроме того, хочется поблагодарить Алексиса Деверию за создание прекрасного справочного ресурса `https://caniuse.com/`, призванного помочь разработчикам и веб-дизайнерам, которые используют CSS при оформлении сайтов.

Самых высоких похвал также заслуживают все, кто, несмотря на постоянную занятость, принимал активное участие в развитии сообщества разработчиков, пополняя его новыми людьми из самых разных социальных групп. Технология CSS всего лишь открывает новые возможности — конечный результат всегда зависит от опытности и слаженности работы всей команды разработчиков.

Впервые посетив IT-конференцию в далеком 2007 году, я была удивлена тем, что докладчиками в основном оказались мужчины (93%), причем исключительно белые. Картина в зале была почти такой же: среди посетителей почти не наблюдалось гендерного и этнического разнообразия. По правде говоря, причиной, побудившей меня участвовать в конференции, как раз и было присутствие женщины в списке докладчиков. Я поняла, что ситуацию нужно менять самым коренным образом. Но тогда я даже представить не могла, насколько много поистине талантливых людей мне удастся привлечь в индустрию в течение последующих десяти лет.

Мне хотелось бы поблагодарить людей, каждодневный усердный труд которых обычно остается незамеченным. К сожалению, я смогу отметить далеко не всех, а только тех, с кем приходилось непосредственно сотрудничать. Самые приятные впечатления остались от работы с Эрикой Стэнли из сообщества Women Who Code Atlanta, Кариной Зоной из Callback Women и Мей Ву из Oakland Maker Space. Чтобы поддержать в развитии такие сообщества, как The Last Mile, Black Girls Code, Girls Incorporated, Sisters Code и многие другие, я решила опубликовать список `http://www.standardista.com/feeding-the-diversity-pipeline/`, взглянув на который легко удостовериться, что карьера веб-разработчика по силам каждому из нас.

Большое спасибо всем вам! Спасибо за вашу помощь, без нее у меня не получилось бы сделать и малой части из задуманного десять лет назад.

Эстелл Уэйл,
Пало-Альто, Калифорния,
19 июля 2017 г.

ГЛАВА 1

CSS и HTML-документы

Каскадные таблицы стилей (Cascading Style Sheets — CSS) относятся к основополагающим инструментам разработки, применяемым для оформления HTML-документов. На сегодняшний день CSS используется для определения внешнего вида не только веб-страниц, но и любого другого контента, зачастую без видимых на то причин. Например, технология CSS задействуется при отображении графического интерфейса браузеров с движком Gecko, форматировании статей и лент новостей в RSS-клиентах, а также оформлении окон всевозможных мессенджеров (служб обмена мгновенными сообщениями). О важности CSS в веб-разработке свидетельствует включение средств ее поддержки во многие фреймворки JavaScript и базовые синтаксические конструкции языка.

История CSS

Впервые CSS была представлена широкой публике в 1994 году, на заре популярности Интернета. В те далекие времена стилизация документов, отображаемых в окне браузера Mosaic, выполнялась конечными пользователями, определяющими такие немногочисленные параметры форматирования элементов страницы, как шрифт, его размер, цвет и т.п. Разработчикам HTML-документов подобные средства оформления были недоступны. Авторы документа всего лишь размечали в тексте заголовки, абзацы и другие элементы стандартных типов, а их конечный вид всецело определялся настройками браузера. В частности, если пользователю хотелось отображать заголовки первого уровня мелким шрифтом розового цвета, а заголовки шестого уровня — крупным шрифтом красного цвета, то изменить это решение разработчику HTML-документа было не под силу.

Подобное положение вещей многих не устраивало и стало причиной возникновения технологии CSS. Перед ее разработчиками ставилась задача создать простой язык описания внешнего вида документа, доступный для изучения авторами HTML-документов и, что особенно важно, снабжающий их и конечных пользователей эффективными средствами стилизации любых аспектов представления веб-страниц. "Каскадность" предполагает очередность применения разных стилей к одним и тем же элементам, назначаемых согласно их приоритетности. Изменять внешний вид документа

с помощью CSS получили возможность в первую очередь создатели документов, и только затем — конечные пользователи.

Работа над первой редакцией языка — CSS1 — была закончена к концу 1996 года. Несмотря на образование отдельной рабочей группы (CSS Working Group), сфокусированной на скорейшей разработке спецификации CSS2, большинство браузеров того времени в той или иной степени уже поддерживало стандарты CSS1. Недочеты технологии стали проявляться по мере насыщения веб-страниц множественными CSS-стилями, использование каждого из которых по отдельности не вызывало затруднений. В частности, в ранних спецификациях CSS наблюдались заметные сложности с блочной верстской документа, что не позволяло разработчикам документов в полной мере использовать весь инструментарий CSS. В результате активное использование спецификации началось только после внесения в нее вполне обоснованных изменений. Потребовалось несколько лет, чтобы CSS начали применять в крупных проектах, таких как сайт журнала *Wired* и CSS Zen Garden. Начиная с этого момента востребованность технологии не поддавалась сомнению.

Стоит отметить, что работа над CSS2 была завершена в 1998 году — еще до всеобщего внедрения поддержки технологии каскадных таблиц стилей в большинстве браузеров. Рабочая группа сразу же переключилась на разработку следующей редакции языка (CSS3), параллельно исправляя недостатки текущей спецификации, — так у CSS2 появилась первая ревизия (CSS2.1). Следуя всеобщей тенденции, CSS3 была представлена не единой спецификацией, а набором (гипотетически) независимых программных модулей. Переход к модульной структуре, заимствованной у набирающей популярность технологии XHTML, был обусловлен вполне очевидными причинами, рассмотренными ниже.

Представив CSS3 набором программных модулей, можно добиться поэтапного внедрения поддержки отдельных спецификаций — по мере необходимости, роста популярности или появления рекомендаций консорциума W3C. Это очень удобно, поскольку позволяет использовать в браузерах только актуальные спецификации и отстрочить внедрение стандартов, не востребованных на данный момент. Например, в начале 2012 года статус обязательных (Recommendation) для внедрения в браузеры получили всего три модуля CSS3 (а также CSS1 и CSS2.1): Color Level 3, Namespaces и Selectors Level 3. В то же время статус кандидатов на внедрение (Candidate Recommendation) имели целых семь модулей. При этом на стадии разработки (Working Draft) с различной степенью готовности находилось сразу несколько десятков модулей. Если отказаться от модульного подхода, то цветами, селекторами и пространствами имен можно было бы управлять только после завершения разработки всех остальных спецификаций, включаемых в CSS. Благодаря модульной структуре они доступны в браузерах уже сейчас.

С другой стороны, при таком подходе к разработке программных решений сложно определить версию CSS, поддерживаемую браузером. Судите сами: если предположить, что на конец 2018 года большинство внедряемых модулей будет иметь версию 3 (что далеко не так) и один только модуль CSS Selectors будет включать средства четвертого поколения, то можно ли утверждать, что браузер поддерживает CSS4? Можно ли считать переход к CSS4 завершенным, если в ее составе все еще присутствуют

модули третьего поколения? И как быть с модулем CSS Grid Layout, представленным единственным (первым) поколением?

Таким образом, можно с уверенностью говорить только о текущей версии каждого отдельного модуля, но не о технологии CSS в целом. Как бы там ни было, гибкость, обеспечиваемая модульной структурой технологии, важнее семантически выверенного названия и однозначно стоит неразберихи с определением номера конечной версии CSS. (Рабочая группа CSS ежегодно выпускает официальные пресс-релизы, в которых информирует о текущих спецификациях CSS.)

Завершив краткий исторический экскурс, перейдем к изучению самой технологии CSS. Но начнем мы издалека — с разметки HTML-документа.

Элементы

Элементы — это базовые структурные компоненты документа. В HTML все основные элементы, такие как `p`, `table`, `span`, `a` и `div`, легко сопоставляются с содержимым документа. Каждый из элементов представляет отдельную часть документа.

Замещаемые и незамещаемые элементы

Правила CSS применяются к элементам документа, и далеко не все элементы одинаковые. В частности, абзац с текстом и графическое изображение относятся к элементам разных типов. Подобным образом отличаются элементы `div` и `span`. С точки зрения CSS все элементы документа делятся на два основных типа: замещаемые и незамещаемые.

Замещаемые элементы

Замещаемыми считаются элементы, содержимое которых может представляться данными, отсутствующими в самом документе. Самый очевидный замещаемый элемент в HTML — это `img`, с помощью которого на странице отображается графический файл, хранящийся отдельно от самого документа. Если быть предельно точным, то содержимое у элемента `img` вообще отсутствует, что видно из следующего примера:

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

Приведенный выше код разметки содержит только название элемента и значение одного из его атрибутов. Таким образом, элемент не хранит вообще ничего до тех пор, пока в него не будет добавлена ссылка на внешний файл (указывается в атрибуте `src`). Изображение появляется в документе только при указании действительного графического файла по существующему расположению. Если ссылка указывает на несуществующее расположение, то вместо изображения в документ добавляется заполнитель с извещением об ошибке загрузки файла или вообще ничего не отображается.

К замещаемым также относится элемент `input`, представляющий в документе переключатель, флажок или текстовое поле. Добавляемый в документ элемент управления зависит от типа элемента `input`.

Незамещаемые элементы

Документы HTML состоят преимущественно из незамещаемых элементов. Содержимое таких элементов определяется исключительно пользовательским агентом (обычно браузером) и выводится в блоке, образованном самим элементом. Например, к незамещаемому типу относится элемент `<span>Привет!</span>`, поскольку при выполнении этого кода в браузере в документ добавляется текст `Привет!`. Незамещаемыми элементами представлены абзацы с текстом, заголовки, ячейки таблицы, списки и многие другие компоненты документа.

Строчные и блочные элементы

Наряду с приведенной выше классификацией большинство элементов документа относится к одному из двух типов: *блочные* (block-level) и *строчные* (inline-level). Существуют и другие, более сложные типы элементов, но все они сводятся к одному из указанных базовых типов. Строчные и блочные элементы хорошо знакомы каждому, кто пытался создавать документы, форматирование которых задается одними только средствами языка HTML. Примеры элементов обоих типов приведены на рис. 1.1.

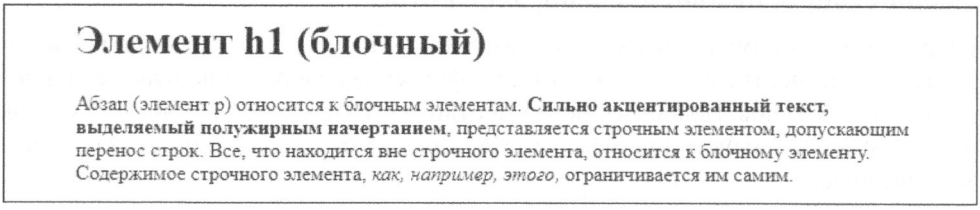


Рис. 1.1. Строчные и блочные элементы HTML-документа

Блочные элементы

Блочным называется элемент, который представляется на странице прямоугольной областью, занимающей всю доступную ширину родительского элемента. Другими словами, в родительском элементе у блочного элемента не бывает соседних боковых элементов, а от содержимого сверху и снизу он отделяется специальными отступами и полями. К блочным относятся такие распространенные HTML-элементы, как `p` и `div`. Замещаемые элементы также могут быть блочными, хотя на практике это встречается крайне редко.

Элементы списка представляются блочными элементами специального типа. Наряду с общими характеристиками, свойственными всем блочным элементам, они снабжаются специальными маркерами — символами (неупорядоченный список) или цифрами (упорядоченный список), которые привязываются к прямоугольнику элемента. За исключением маркеров элементы списка ничем не отличаются от блочных элементов регулярного типа.

Строчные элементы

Строчные элементы включаются в текстовую строку, не прерывая ее течение. Самый распространенный строчный элемент в HTML — это, конечно же, `a`.

К строчным также относятся элементы акцентированного текста `strong` и `em`. Перед строчными элементами и после них разрывы не образуются, поскольку они являются непосредственной частью другого (родительского) элемента и не изменяют его структуру.

Несмотря на подобные характеристики, строчные и блочные элементы в HTML и CSS имеют несколько разные свойства. В частности, в HTML блочные элементы запрещается вкладывать в строчные. В CSS подобного ограничения нет, и порядок вложения элементов может быть самым произвольным, независимо от их типа.

Чтобы понять, как такое возможно, рассмотрим свойство `display`.

display

Значение	[<*display-outside*> || <*display-inside*>] | <*display-listitem*> | <*display-internal*> | <*display-box*> | <*display-legacy*>
Определение	См. ниже
Начальное значение	`inline`
Применяется	Все элементы
Вычисляется	Согласно определению
Наследуется	Нет
Анимируется	Нет

<*display-outside*>
`block | inline | run-in`

<*display-inside*>
`flow | flow-root | table | flex | grid | ruby`

<*display-listitem*>
`list-item &&` <*display-outside*>`? && [ flow | flow-root ]?`

<*display-internal*>
`table-row-group | table-header-group | table-footer-group | table-row | table-cell | table-column-group | table-column | table-caption | ruby-base | ruby-text | ruby-base-container | ruby-text-container`

<*display-box*>
`contents | none`

<*display-legacy*>
`inline-block | inline-list-item | inline-table | inline-flex | inline-grid`

У этого свойства огромное количество возможных значений, но на данный момент нас интересуют только три из них: `block`, `inline` и `list-item`. Остальные значения будут детально описаны в последующих главах. Например, значения `grid` и `inline-grid` рассмотрены в главе, посвященной верстке документа по сетке, а все значения, в названиях которых встречается слово "table", — в главе, где описываются средства CSS, отвечающие за оформление таблиц.

Изучим поведение элементов, свойство `display` которых представлено значениями `block` и `inline`. Рассмотрим следующий код.

```
<body>
<p>В абзац включен <em>строчный элемент</em>.</p>
</body>
```

В этом HTML-коде имеются один строчный (`em`) и два блочных (`body` и `p`) элемента. Согласно спецификации HTML элемент `em` разрешается вкладывать в элемент `p`, но не наоборот. Иерархическая структура элементов в HTML не предполагает вложение блочных элементов в строчные.

В CSS ограничения на вложение элементов, характерных для HTML, не предусмотрены, что позволяет произвольным образом изменять тип каждого из них. Для изменения типа двух из трех элементов (при неизменной разметке документа) применяется следующий код CSS.

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

Выполнение этого кода приводит к вложению блочного элемента в строчный, что никак не противоречит правилам CSS. Если же попытаться изменить способ вложения элементов с помощью HTML, то можно получить ошибку выполнения кода.

```
<em><p>Абзац некорректно вложен в строчный элемент.</p></em>
```

Независимо от изменений, вносимых в элементы с помощью CSS, в HTML данный способ вложения элементов недопустим.

В то время как для улучшения представления HTML-документа способ вложения элементов изменяется достаточно часто, в документах XML подобные изменения нужно вносить крайне осторожно. Зачастую XML-документы вообще не содержат сведений о типах включенных в них элементов, поэтому их нужно задавать отдельно. В качестве примера рассмотрим следующий фрагмент XML-кода.

```
<book>
   <maintitle>Cascading Style Sheets: The Definitive Guide</maintitle>
   <subtitle>Third Edition</subtitle>
   <author>Eric A. Meyer</author>
   <publisher>O'Reilly and Associates</publisher>
   <pubdate>November 2006</pubdate>
   <isbn type="print">978-0-596-52733-4</isbn>
</book>
<book>
   <maintitle>CSS Pocket Reference</maintitle>
   <subtitle>Third Edition</subtitle>
   <author>Eric A. Meyer</author>
   <publisher>O'Reilly and Associates</publisher>
   <pubdate>October 2007</pubdate>
   <isbn type="print">978-0-596-51505-8</isbn>
</book>
```

Поскольку по умолчанию для свойства `display` устанавливается значение `inline`, при выполнении этого XML-кода в браузере все его элементы выводятся сплошной текстовой строкой, как показано на рис. 1.2.

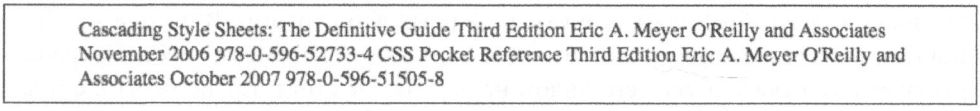


Рис. 1.2. XML-документ в представлении по умолчанию

Для улучшения внешнего вида документа нужно задать значения свойства `display` отдельно для каждого из его элементов.

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

В первой строке приведенного выше кода перечислены XML-элементы блочного типа, а во второй — XML-элементы, сохранившие строчный тип. В данном случае блочные XML-элементы сопоставимы с элементами `div` в HTML, а строчные XML-элементы — с элементами `span`.

Основополагающая возможность изменения типа элемента в CSS применяется для решения самых разных задач. В частности, дополнив приведенный выше код CSS несколькими простыми стилями, можно существенно повысить уровень визуального восприятия документа (рис. 1.3).

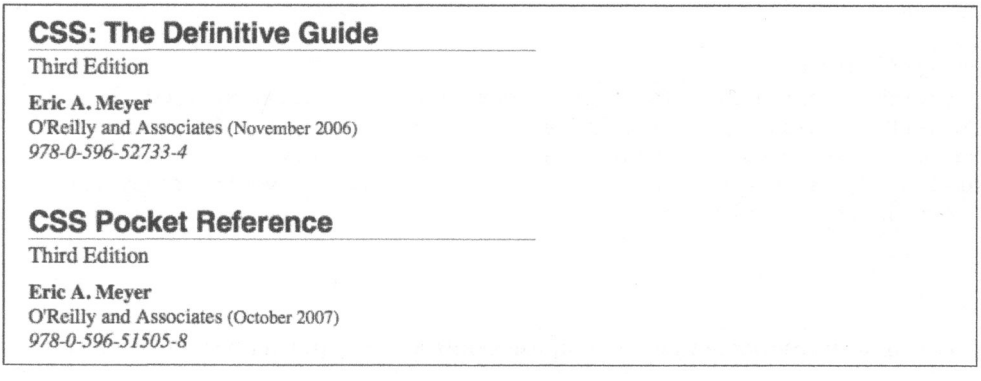


Рис. 1.3. Применение стилей к XML-документу

Прежде чем приступить к изучению методик написания кода CSS, необходимо понять, каким образом он подключается к HTML-документу. В конце концов, для изменения внешнего вида HTML-документа с помощью CSS необходимо добиться однозначного взаимодействия инструментов обеих технологий. Далее вы узнаете о том, как обеспечить выполнение кода CSS в HTML-документах.

Взаимодействие CSS и HTML

Как известно, HTML-документы имеют строгую структуру, которую необходимо учитывать при их визуальном оформлении. Данная проблема не нова — она была знакома разработчикам еще самых первых веб-страниц. В погоне за эффектным внешним видом мы порой забываем, что каждый документ имеет внутреннюю структуру, мало связанную с его видом на экране. Не стоит забывать, что, стараясь получить самую привлекательную веб-страницу в мире, легко нарушить строго выверенный порядок представления информации в HTML-документах.

Внутренняя структура документа — это то, что в наибольшей степени определяет способ взаимодействия разметок CSS и HTML. Документы, лишенные ее, не подлежат изменению представления с помощью инструментария CSS. Чтобы лучше понять, о чем идет речь, рассмотрим следующий пример и попробуем разбить приведенный ниже HTML-документ на основные компоненты.

```
<html>
<head>
<title>Eric's World of Waffles</title>
<meta http-equiv="content-type" content="text/html; charset=utf-8">
<link rel="stylesheet" type="text/css" href="sheet1.css" media="all">
<style type="text/css">
/* Мои собственные стили! */
@import url(sheet2.css);
</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>
```

Результат разметки документа и применения к нему стилей представлен на рис. 1.4.

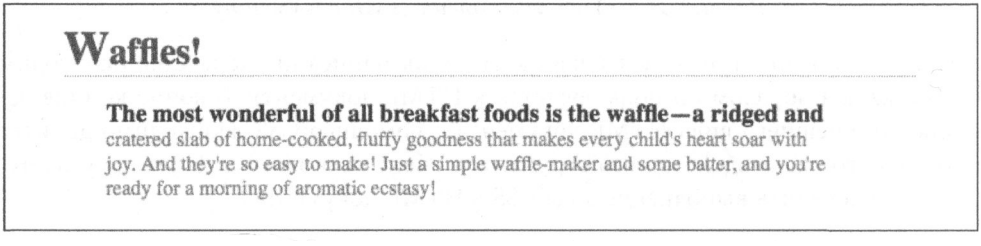


Рис. 1.4. Простой документ

Детально все возможные способы подключения стилей CSS к HTML-документу описаны в следующих разделах.

Тег `link`

Первый способ указания кода CSS, выполняемого в документе, заключается в добавлении в HTML-документ тега `link`.

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

Тег `link` был включен в самые ранние спецификации HTML и используется для подключения стилей CSS к документу незаслужено редко. В общем случае он позволяет связывать с документом, его содержащим, любые внешние файлы, но нас будет интересовать только возможность его использования для подключения CSS-стилей. На рис. 1.5 показано, каким образом к документу можно подключить файл каскадной таблицы стилей `sheet1.css`.

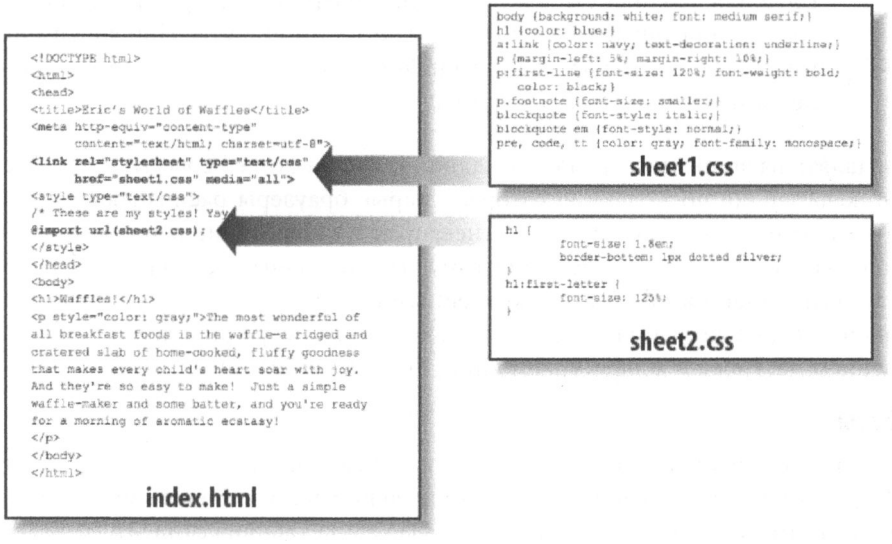


Рис. 1.5. Схема подключения внешних стилей CSS к HTML-документу

Стили CSS, используемые в HTML-документе, но не включенные в него, называются внешними. А файлы, их содержащие, известны как файлы внешних каскадных таблиц стилей. Таким образом, термин "внешний" указывает на сохранение стилей вне самого документа.

Чтобы обеспечить загрузку внешних каскадных таблиц стилей в документ, тег `link` нужно помещать в элемент `head` документа. Только при соблюдении этого правила браузер загрузит файл со стилями и будет применять их для оформления HTML-документа так, как показано на рис. 1.5. Еще один вариант подключения внешней таблицы стилей (файл `sheet2.css` на рис. 1.5) к документу заключается в использовании команды `@import`. Правило `@import` нужно всегда помещать в самом верху блока стилей, в противном случае таблица стилей не импортируется.

Что собой представляет внешняя каскадная таблица стилей? Она содержит набор CSS-правил, подобных рассмотренным в начале главы и приведенных в примере HTML-документа, которые сохранены в виде отдельного файла. Этот файл содержит один только код CSS, в него нельзя добавлять код HTML или любой другой код разметки документа — только правила, содержащие объявления стилей. Например, внешняя таблица стилей может быть представлена следующим CSS-кодом.

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

Как видите, код CSS лишен тегов HTML и комментариев — он включает одни только объявления стилей. Файл каскадной таблицы стилей сохраняется в формате простого текста и снабжается расширением `.css`, например `sheet1.css`.

Внешняя таблица стилей не должна содержать код разметки документа, только описанные далее правила и комментарии CSS. При добавлении тегов HTML в файл каскадной таблицы стилей она, скорее всего, перестанет импортироваться в документ.

Добавлять расширение к файлу каскадной таблицы стилей совсем не обязательно, хотя и желательно, поскольку некоторые старые браузеры распознают файлы CSS исключительно по расширению `.css`, невзирая на включение в тег `link` атрибута `type` со значением `text/css`. А все потому, что отдельные веб-серверы предоставляют доступ к внешним файлам, сохраненным в формате `text/css`, только в случае присвоения им расширения `.css`. К счастью, данная неурядица легко исправляется изменением настроек в конфигурационных файлах сервера.

Атрибуты

Тег `link` включает несколько важных атрибутов, принимающих простые значения. Первый из них — атрибут `rel`, указывающий на отношения между документом и внешним файлом, в данном случае `stylesheet` (таблица стилей). Атрибут `type` всегда устанавливается в значение `text/css` и определяет тип данных, загружаемых из внешнего файла. Таким образом, распознав первые два атрибута, браузеру становится известно о том, что во внешнем файле хранится таблица стилей CSS. Эти сведения позволяют ему правильно обработать импортируемые данные. Вполне вероятно, что вскоре таблицы стилей будут создаваться с помощью языков, отличных от CSS, и браузеру нужно предоставить возможность распознавать их все.

Следующий атрибут — `href` — определяет адрес URL, по которому располагается таблица стилей. В качестве его значения указывается полный или относительный путь к файлу. В нашем примере задан относительный путь расположения файла. В общем случае он может представляться в виде `http://meyerweb.com/sheet1.css`.

Наконец, атрибут `media` задает устройство, для которого следует применять стилевое оформление. Атрибут принимает значения дескрипторов технических

характеристик, которые определяют правила представления документа для каждого из устройств вывода, имеющих разные технические возможности. Допускается указывать сразу несколько дескрипторов технических характеристик, разделяя их запятыми. Например, следующий простой код указывает применять внешнюю таблицу стилей к документу, выводимому и на экран, и на проектор.

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

Медиа-запросы сложны для понимания, поэтому в рассматриваемом примере мы ограничимся только самым простым из них. Подробные сведения о медиа-запросах будут приведены далее.

Обратите внимание на то, что к документу можно присоединить сразу несколько таблиц стилей. В подобных случаях начальный вид документа определяют таблицы стилей, подключаемые с помощью тегов `link`, атрибут `rel` которых установлен в значение `stylesheet`. В частности, для использования в документе сразу двух внешних таблиц стилей — `basic.css` и `splash.css` — применяется следующий HTML-код.

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

При выполнении этого кода браузер загрузит оба внешних файла и применит к документу все объявленные в них стили, как показано в следующем примере.

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

<p class="a1">При подключении таблицы стилей 'basic.css' текст абзаца
              окрашивается в серый цвет.</p>

<p class="b1">При подключении таблицы стилей 'splash.css' текст абзаца
              окрашивается в серый цвет.</p>
```

В описанных выше примерах не упоминается еще один важный атрибут тега `link`: `title`. В реальные документы он добавляется не часто, поскольку при неправильном использовании приводит к получению совершенно непредвиденных результатов. О том, почему это происходит, рассказано в следующем разделе.

Альтернативные таблицы стилей

В случае необходимости к документу можно присоединить *альтернативные таблицы стилей*. Для их определения атрибуту `rel` тега `link`, описанному в предыдущем разделе, присваивается значение `alternate stylesheet`. Альтернативные таблицы стилей применяются для изменения визуального представления документа только по указанию пользователя.

Чтобы позволить браузеру использовать альтернативные таблицы стилей, в тег `link` также нужно включить атрибут `title`, отвечающий за предоставление пользователю всех доступных возможностей. Ниже приведен один из вариантов решения этой задачи.

```
<link rel="stylesheet" type="text/css"
      href="sheet1.css" title="Default">
<link rel="alternate stylesheet" type="text/css"
      href="bigtext.css" title="Big Text">
<link rel="alternate stylesheet" type="text/css"
      href="zany.css" title="Crazy colors!">
```

При выборе одной из доступных опций браузер изменяет стилевое оформление документа с варианта по умолчанию (Default) на указанный пользователем. Один из способов выбора таблицы стилей, применяемой к документу, показан на рис. 1.6 (поддерживается браузерами с первых редакций CSS).

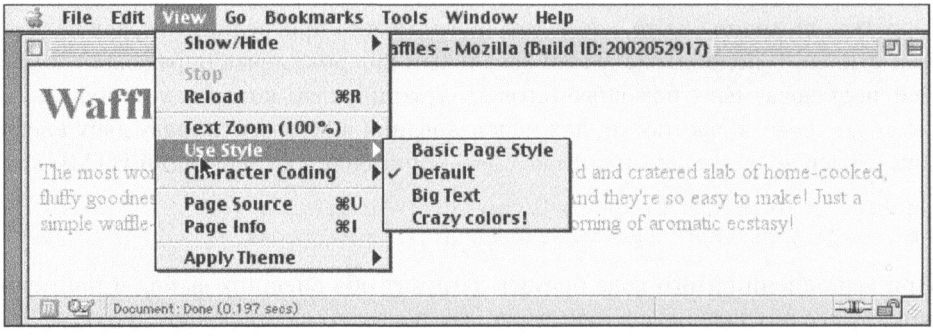


Рис. 1.6. Выбор альтернативной таблицы стилей для документа

К концу 2016 года возможность применения к документу альтернативных таблиц стилей поддерживалась подавляющим большинством браузеров, основанных на движке Gecko (таких, как Firefox и Opera). В Internet Explorer подключение альтернативных таблиц стилей реализуется исключительно средствами JavaScript и не обеспечивается инструментами самого браузера. Браузеры семейства WebKit вообще не поддерживают работу с альтернативными таблицами стилей, что по меньшей мере обескураживает, особенно принимая во внимание возраст браузера, изображенного на рис. 1.6.

Для группировки альтернативных таблиц стилей каждой из них присваиваются одинаковые значения атрибута `title`. В приведенном ниже примере показано, как правильно обеспечить пользователя командами изменения визуального оформления документа при выводе его на экран и отправке на печать.

```
<link rel="stylesheet" type="text/css"
      href="sheet1.css" title="Default" media="screen">
<link rel="stylesheet" type="text/css"
      href="print-sheet1.css" title="Default" media="print">
<link rel="alternate stylesheet" type="text/css"
      href="bigtext.css" title="Big Text" media="screen">
<link rel="alternate stylesheet" type="text/css"
      href="print-bigtext.css" title="Big Text" media="print">
```

Теперь при выборе варианта Big Text браузер перейдет к альтернативному варианту визуального оформления, подключив таблицу стилей `bigtext.css` для отображения документа на экране или таблицу стилей `print-bigtext.css` при отправке его на печать. Обратите внимание на то, что в подобном случае таблицы стилей `sheet1.css` и `print-sheet1.css` (используются по умолчанию) не принимают участие в изменении внешнего вида документа.

Давайте детально разберемся, как работает такой код. Таблица стилей, которая подключается с помощью тега `link`, снабженного атрибутом `rel` со значением `stylesheet` и атрибутом `title`, называется *предпочитаемой*. Она применяется к документу по умолчанию, поскольку браузер предпочитает ее альтернативным таблицам стилей, назначенным в остальных тегах `link`. Тем не менее действие предпочитаемой таблицы стилей *отменяется*, как только пользователь выбирает альтернативную таблицу стилей.

Более того, несмотря на возможность визуального оформления документа с помощью сразу нескольких предпочитаемых таблиц стилей, по умолчанию к нему применяется только одна из них. Рассмотрим следующий пример.

```
<link rel="stylesheet" type="text/css"
      href="sheet1.css" title="Default Layout">
<link rel="stylesheet" type="text/css"
      href="sheet2.css" title="Default Text Sizes">
<link rel="stylesheet" type="text/css"
      href="sheet3.css" title="Default Colors">
```

Благодаря включению атрибута `title` все три тега `link` подключают к документу предпочитаемые таблицы стилей. Если внешний вид документа одномоментно определяет только одна из предпочитаемых таблиц стилей, то как определить, какие из указанных трех файлов не задействуются? Невозможно однозначно ответить на этот вопрос, поскольку в HTML не предусмотрены инструменты точного указания предпочитаемой таблицы стилей для документа.

Если при подключении к документу таблицы стилей опустить аргумент `title`, то она рассматривается браузером как *постоянная*. Такая таблица стилей применяется на постоянной основе, а эффект ее воздействия на документ не может отменяться пользователем.

Элемент `style`

Второй способ связывания таблиц стилей с документом заключается в применении элемента `style`.

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

Элемент `style` непосредственно вставляется в документ и в обязательном порядке, подобно тегу `link`, снабжается атрибутом `type`. При подключении таблицы стилей CSS атрибут `type` принимает значение `"text/css"`.

Как показано выше, элемент `style` начинается открывающим тегом `<style type="text/css">`, после которого указываются применяемые в документе стили, и заканчивается закрывающим тегом `</style>`. Часто элемент `style` снабжается

атрибутом media, назначение которого такое же, как и у равнозначного ему атрибута тега link, описанного в предыдущем разделе.

Стили, объявленные между открывающим и закрывающим тегами элемента style, называются *внедренными* или *глобальными*. (Внедрение подразумевает добавление стиля непосредственно в сам документ.) Кроме стилей, добавляемых непосредственно в документ, элемент style позволяет подключать внешние таблицы стилей, используя команду @import.

Правило @import

Правило @import работает подобно тегу link, но используется не самостоятельно, а включается в код элемента style.

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

Как и тег link, правило @import позволяет загружать стили из внешних источников и применять их для оформления документа. Команды отличаются только синтаксисом и местом расположения в HTML-коде. Правило @import в обязательном порядке заключается в контейнер элемента style, который располагается перед объявлением остальных правил CSS документа. В противном случае загружаемые с его помощью стили применяться в документе не будут. Команду @import также нужно добавлять перед объявлением других стилевых правил, иначе она не будет выполняться.

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

В одной таблице стилей допускается применять сразу несколько команд @import. В отличие от тега link, в правиле @import нельзя указать альтернативные таблицы стилей, поэтому к документу будут подключаться в точности те файлы, что указаны в коде. Ниже приведено несколько примеров корректного задания правила @import.

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

В изменении представления документа принимают участие все стили, которые импортированы из внешних файлов, указанных в приведенных выше правилах @import.

Подобно тегу link, правило @import позволяет устанавливать тип носителя, при выводе на который к документу применяется стилевое форматирование. Тип носителя указывается после адреса URL, по которому располагается внешний файл таблицы стилей.

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

Медиа-запросы сложны для понимания и детально рассматриваются в главе 20.

Чаще всего правило `@import` задействуется в случаях, когда во внешней таблице стилей необходимо использовать стили, объявленные в другом внешнем файле. Для решения этой задачи тег `link` не подходит, поскольку добавлять HTML-код во внешний CSS-файл не разрешается. В противоположность ему правило `@import` относится к программным конструкциям CSS, и подобное ограничение на него не распространяется. Таким образом, во внешний файл таблицы стилей допускается включать команды, подобные следующим.

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

В реальных документах подключаются иные внешние файлы, но используются точно такие же синтаксические конструкции. Обратите внимание на возможность использования в качестве URL как относительных, так и абсолютных путей расположения файлов, содержащих таблицы стилей. Правила здесь такие же, как и при импорте внешних файлов с помощью тега `link`.

Важно помнить, что команды `@import` добавляются в начало CSS-файла, перед объявлением регулярных правил, что наглядно проиллюстрировано выше. Любые синтаксические конструкции, включающие ключевое слово `@import` и добавленные в середину или конец CSS-файла (например, после строки `body{color: red;}`), попросту игнорируются пользовательскими агентами.

Правила `@import`, добавленные в CSS-файл после объявления регулярных стилей, не игнорируются только старыми версиями браузера Internet Explorer для Windows. Таким образом, подключая внешние таблицы стилей, исходно разработанные для Internet Explorer, к другим браузерам, можно столкнуться с неправильным выполнением команд `@import`.

Ссылки на таблицы стилей в HTTP-заголовках

Существует еще один, необычный способ подключения CSS-файлов к HTML-документам. Он заключается в добавлении ссылки на файл в заголовок HTTP.

В Apache ссылка на CSS-файл включается в HTTP-заголовок `Link` файла `.htaccess`, как показано ниже.

```
Header add Link "</ui/testing.css>;rel=stylesheet;type=text/css"
```

Эта команда предписывает всем поддерживаемым сервером браузерам применять указанную внешнюю таблицу стилей ко всем документам, указанным в файле `.htaccess`. Она полностью равнозначна команде подключения CSS-файла с помощью тега `link`. Альтернативное и в чем-то даже более эффективное решение заключается в добавлении равнозначного правила в конфигурационный файл `httpd.conf` сервера.

```
<Directory /path/to/ /public/html/directory>
Header add Link "</ui/testing.css>;rel=stylesheet;type=text/css"
</Directory>
```

Как и в предыдущем случае, эффект воздействия таблицы стилей на документы распространяется только на браузеры, поддерживающие такой способ подключения таблиц стилей к документу. Отличие проявляется лишь в расположении ссылки на внешний CSS-файл.

Что касается поддержки, то к концу 2017 года ссылки на внешние таблицы стилей в HTTP-заголовках обрабатывали пользовательские агенты семейств Firefox и Opera. Это ограничивает применение описанной выше методики. Добавив ссылку на внешний CSS-файл в HTTP-заголовок тестового сервера, можно обособить общедоступные сайты от сайтов, находящихся в разработке. Описанная методика также позволяет скрывать отдельные таблицы стилей от браузеров, основанных на движке WebKit, и браузеров семейства Internet Explorer, если, конечно, на то есть веские причины.

Представленную выше задачу можно решить с помощью таких языков написания сценариев, как PHP и IIS, позволяющих задавать стилевое оформление документов без обращения к HTTP-заголовкам. Кроме того, подобные решения часто применяются для добавления элемента `link` в документы только по требованию сервера. Последний вариант считается наиболее оптимальным, поскольку элемент `link` поддерживается всеми без исключения браузерами.

Встроенные стили

В случаях, когда стили требуется применить только к отдельному элементу документа, можно обойтись без внешних файлов CSS. Чтобы напрямую указать стиль для HTML-элемента, в него добавляется атрибут `style`.

```
<p style="color: gray;">The most wonderful of all breakfast foods is
the waffle—a ridged and cratered slab of home-cooked, fluffy goodness...
</p>
```

Атрибут `style` можно добавлять к любым тегам HTML в элементе `body` и нельзя включать только в теги элементов `head` и `title`.

Синтаксис атрибута `style` очень простой и сопоставим с синтаксисом элемента `style`, за исключением фигурных скобок, которые у атрибута заменены двойными кавычками. В частности, для представления текста *единственного* абзаца темно-бордовым цветом (`maroon`), а его фона — желтым цветом (`yellow`) используется тег `<p style="color: maroon; background: yellow;">`. Никакая другая часть документа под воздействие указанного стиля не подпадает.

В атрибуте `style` указывается только блок объявлений — вводить в качестве его значения всю таблицу стилей недопустимо. Кроме того, при встраивании стиля в HTML-элемент не разрешается использовать команду `@import` и указывать правила целиком. Атрибут `style` должен представлять только ту часть правила, которая заключается в фигурные кавычки.

В общем случае старайтесь избегать использования встроенных стилей в документах. В действительности из всех языков разметки документов встраивание стилей в элементы свойственно только HTML. Указание стилей в HTML-элементах с помощью атрибута `style` нарушает основную концепцию CSS — способность централизованного управления стилями, применяемыми для визуального оформления всего документа или всех документов, размещаемых на веб-сервере. Даже несмотря на несравнимо большую гибкость, проявляемую при форматировании текста, в большинстве случаев встроенные стили успешно заменяются элементами `font`.

Таблицы стилей

Определившись со способами подключения таблиц стилей к HTML-документам, настало время выяснить, что же они представляют собой. Вам уже наверняка знаком приведенный ниже код.

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

Из подобных правил состоят все без исключения стили, применяемые к документу, — простые и сложные, короткие и длинные. Мало какой документ обходится без элемента `style` — даже если в нем отсутствуют стилевые правила, то наверняка содержатся несколько команд `@import`, применяемых для импорта стилей из внешних источников (см. предыдущий раздел).

Перед тем как продолжить дальнейшее обучение, необходимо выяснить, какие данные нужно, а какие нельзя включать в код таблицы стилей.

Код разметки

Код разметки документа недопустимо добавлять в таблицу стилей! Это правило кажется очевидным, но оно нарушается чрезвычайно часто. Исключение сделано только для тегов комментариев, вводимых в элемент `style`.

```
<style type="text/css"><!--
h1 {color: maroon;}
body {background: yellow;}
--></style>
```

Поддержка данного исключения в последних спецификациях CSS неподвластна логическому объяснению и имеет исключительно исторические корни.

Стилевые правила

Чтобы понять, как работают правила, определяющие стилевое оформление документа, нужно изучить их структуру.

Каждое правило состоит из двух основных частей: *селектора* и *блока объявлений*. Блок объявлений содержит одно или несколько объявлений, каждое из которых представлено парой *свойство: значение*. Таблица стилей — это не что иное, как набор стилевых правил, собранных в общем файле. Компоненты правила показаны на рис. 1.7.

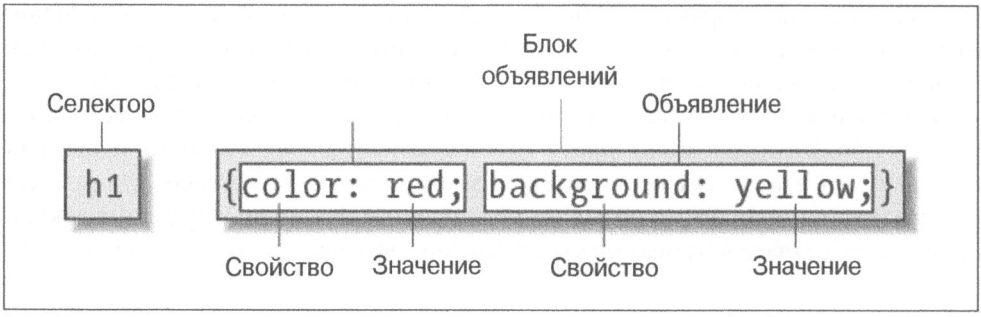


Рис. 1.7. Структура правила в CSS

Селектор, вводимый в начале записи (слева), определяет часть документа, к которой применяется форматирование, устанавливаемое правилом. Например, правило, показанное на рис. 1.7, применяется к элементу `h1`. Если селектор представлен ключевым словом `p`, то правило будет применяться сразу ко всем абзацам (элементам `p`) документа.

В правой части правила находится блок объявлений, включающий не менее одного компонента. Каждое объявление представляет собой комбинацию свойства CSS и задаваемого ему значения. В примере, приведенном на рис. 1.7, правило состоит всего из двух объявлений: первое устанавливает красный цвет для элементов, определяемых селектором, а второе размещает их на желтом фоне. В результате все заголовки первого уровня (элементы `h1`) документа получают желтый фон и красный цвет шрифта.

Вендорные свойства

Названия некоторых свойств CSS начинаются с дефиса и снабжаются специальными префиксами, например `-o-border-image`. Такие префиксы называются вендорными и обозначают экспериментальные или запатентованные отдельными производителями программного обеспечения (браузеров) средства CSS — обычно свойства и их значения. К концу 2016 года общепринятыми считались всего несколько вендорных префиксов (табл. 1.1).

Таблица 1.1. Общепринятые вендорные префиксы CSS

Префикс	Программный продукт
`-epub-`	Проекты в формате ePub
`-moz-`	Браузеры семейства Mozilla (Firefox)
`-ms-`	Internet Explorer
`-o-`	Браузеры семейства Opera
`-webkit-`	Браузеры на движке WebKit (Safari и Chrome)

В табл. 1.1 показан правильный формат записи префиксов, хотя в отдельных случаях первые дефисы ошибочно опускаются.

Назначение и область применения вендорных префиксов являются извечным камнем преткновения, вызывая многочисленные споры и разногласия среди разработчиков. Они не относятся к основополагающим инструментам CCS, а потому не рассматриваются в данной книге. Достаточно знать, что чаще всего они применяются производителями программного обеспечения для тестирования новых функциональных возможностей собственных браузеров, исключая влияние на сторонние браузеры. На ранних этапах развития технологии CSS такой подход был вполне оправдан и позволил избежать целого ряда проблем с совместимостью. Но, к сожалению, общедоступность и бесконтрольность использования вендорных свойств рядовыми разработчиками породила целый класс новых проблем.

В результате к концу 2016 года производители браузеров начали постепенно отказываться от поддержки устаревших вендорных свойств и значений. При написании кода CSS, вне всяких сомнений, можно обойтись без использования вендорных свойств, но с ними приходится считаться при редактировании или верстке старых документов.

Обработка отступов

Код CSS нечувствителен к отступам между правилами и совершенно нечувствителен к отступам, расставляемым внутри правил. Но у каждого правила есть свои исключения.

В общем случае отступы используются в CSS так же, как и в HTML, — как бы много их ни располагалось между ключевыми словами, синтаксический анализатор воспринимает их как один пробел. Это позволяет упорядочивать код CSS так, чтобы по нему можно было судить о структурной организации стилей, как показано в следующем примере гипотетического правила `rainbow`.

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

Это только один из возможных вариантов — расставляя отступы по-иному, этот же код можно представить в совершенно другом виде. Для упорядочивания кода допускается применять отступы, представленные символами пробела, табуляции и разрыва строки, комбинируя их самым произвольным образом.

Подобным образом отступы применяются для объявления правил. Ниже приведен пример пяти способов (из бесконечного количества вариантов) представления правил в CSS.

```
html{color: black;}
body {background: white;}
p {
   color: gray;}
h2 {
      color : silver ;
   }
ol
   {
      color
         :
      silver
         ;
   }
```

Первое правило показывает, что в коде CSS вообще можно отказаться от использования пробелов — такой способ представления кода CSS называется *минималистическим*. Начиная с третьего правила количество отступов в коде становится все больше и больше, увеличиваясь до запредельного для понимания уровня в последнем правиле (каждое ключевое слово располагается в отдельной строке).

Каждый из представленных выше вариантов форматирования кода CSS имеет право на существование. Остановитесь на том из них, который проще для восприятия и представляет информацию в наиболее наглядном виде.

В отдельных случаях без пробелов обойтись просто невозможно, например при перечислении ключевых слов в значении свойства, что наглядно проиллюстрировано в примере гипотетического правила `rainbow`, описанного выше.

Комментарии

В код CSS можно добавлять комментарии. Синтаксис комментариев CSS заимствован из C/C++ и имеет следующий вид.

```
/* Комментарий CSS1 */
```

В CSS, как и в C++, комментарий может занимать несколько строк.

```
/* Комментарий CSS1 может
занимать несколько строк */
```

Комментарии CSS нельзя вкладывать друг в друга. Ниже приведен пример вложенного комментария, который распознается синтаксическим анализатором как ошибка.

```
/* Вложение комментариев в CSS запрещено
   /* Вложенный комментарий*/
Распознается только первая часть комментария */
```

Добиться вложения комментариев в CSS очень просто: достаточно временно закомментировать большой блок кода, уже содержащий комментарии. Поскольку в CSS вложение комментариев запрещено, то обработчиком будет распознана только та часть комментариев, которая заканчивается первым найденным символом `*/` (в конце вложенного комментария).

К сожалению, при внедрении комментариев в CSS-коде нельзя использовать символы `//` и `#` (последний зарезервирован для обозначения селекторов идентификаторов). Единственно допустимые для выделения комментариев символы — это `/* */`. Добавляя комментарии в общую с кодом CSS строку, необходимо быть особенно внимательным. Ниже приведен пример правильного размещения комментариев в коде правил.

```
h1 {color: gray;}   /* Комментарий CSS, не помещающийся */
h2 {color: silver;} /* в одну строку, разделяется на несколько */
p {color: white;}   /* комментариев, каждый из которых */
pre {color: gray;}  /* занимает отдельную строку */
```

Если комментарии, помещенные в каждую строку, не выделять символами `/* */`, то закомментированной окажется большая часть стилей.

```
h1 {color: gray;}   /* Комментарий CSS, не помещающийся
h2 {color: silver;}    в одну строку, разделяется на несколько
p {color: white;}      комментариев, каждый из которых
pre {color: gray;}     занимает отдельную строку */
```

В последнем примере справедливым считается только первое правило — `h1 {color: gray;}`. Остальные правила включаются в комментарий и игнорируются браузером.

Синтаксический анализатор CSS упускает комментарии из вида, как если бы их вообще не существовало. Это означает, что комментарии можно помещать внутрь правил (даже в код объявления свойств), не беспокоясь о том, что они нарушат стилевое оформление документа.

Медиа-запросы

Медиа-запросы определяют техническую среду, для которой в браузер загружаются те или иные таблицы стилей. Решение этой задачи возлагается на целый спектр средств CSS — атрибут `media` элементов `link` и `style`, а также команды `@import` и `@media`. Медиа-запросы появились как универсальное продолжение технологии, позволяя определять стилевое форматирование не только для разных типов носителей, но и для технических параметров устройств. Технические условия, определяющие применимость стилей в документе, задаются в медиа-запросах с помощью *дескрипторов технических характеристик*.

Область применения медиа-запросов

Медиа-запросы можно добавлять в следующие места документа:

- в атрибут `media` элемента `link`;
- в атрибут `media` элемента `style`;
- вместо типа носителя в правиле `@import`;
- вместо типа носителя в правиле `@media`.

В запросах допускается указывать не только тип носителя, но и комбинацию типа носителя и технических характеристик или условий.

Простые медиа-запросы

Прежде чем перейти к детальному рассмотрению всех возможностей медиа-запросов, ознакомимся с их базовой структурой. Рассмотрим ситуацию, когда при показе презентации на проекторе в ней необходимо использовать стили, отличные от применяемых при ее отображении на экране. Эта задача решается с помощью всего двух строк кода, приведенного ниже.

```
h1 {color: maroon;}
@media projection {
   body {background: yellow;}
}
```

Его выполнение в браузере приводит к окрашиванию элемента `h1` в желтый цвет при выводе документа через проектор и в темно-бордовый цвет — при отображении его на любых других носителях.

В таблицу стилей разрешается добавлять произвольное количество команд `@media`, в каждой из которых определяются собственные дескрипторы технических характеристик. При необходимости в команду `@media` можно включать целые блоки объявлений правил.

```
@media all {
   h1 {color: maroon;}
   body {background: yellow;}
}
```

В действительности, если в приведенном выше коде удалить первую и последнюю строки, то ровным счетом ничего не изменится. Таким образом, в представленном виде блок объявлений правил только усложняет код CSS.

Для повышения удобочитаемости кода в приведенных выше примерах использованы отступы. Добавлять отступы перед правилами, включаемыми в блок `@media`, не обязательно, хотя и не запрещается. Их применение позволяет визуально разграничить структурные элементы кода CSS.

В приведенных выше примерах объявление медиа-запросов начинается с ключевых слов `projection` и `all`. Не забывайте, что в медиа-запросах можно указываться

не только тип носителя, но и технические характеристики устройств (например, разрешение или ширину экрана), для которых задается форматирование документа (согласно объявленным ниже правилам).

Типы носителей

В самом простом случае медиа-запрос содержит сведения только о *типе носителя*. Возможность указания типа носителя впервые была представлена в CSS2. Ниже перечислены все поддерживаемые спецификацией CSS дескрипторы носителей.

`all`

Все устройства.

`print`

Печатающее устройство типа принтера. Позволяет установить стили для вывода документа на бумагу.

`screen`

Экран монитора. Все без исключения браузеры относятся к пользовательским агентам, выводящим информацию на экран монитора.

На момент написания книги многие браузеры поддерживали еще один тип носителей — `projection`, используемый при написании CSS-кода стилизации презентаций. Теоретически браузеры, запускаемые на смартфонах, должны поддерживать тип `handheld`, но такая поддержка неполная и реализована далеко не во всех устройствах.

Для указания в медиа-запросе сразу нескольких типов носителей введите их названия через запятую. В следующем примере продемонстрированы четыре способа применения стилевого форматирования к документу, который выводится на экран или распечатывается на принтере.

```
<link type="text/css" href="frobozz.css" media="screen, print">
<style type="text/css" media="screen, print">...</style>

@import url(frobozz.css) screen, print;
@media screen, print {...}
```

Ситуация усложняется, когда необходимо указать не только носитель, на который выводится документ, но и технические условия, при которых эта операция осуществляется. Чаще всего критериями служат разрешающая способность или глубина цвета выбранного носителя.

Дескрипторы носителей

Использование медиа-запросов не станет в новинку для тех разработчиков, кому доводилось указывать тип носителя в элементе `link` или команде `@import`. Ниже

показаны два равнозначных способа применения внешней таблицы стилей к документу, выводимому на цветной принтер.

```
<link href="print-color.css" type="text/css"
      media="print and (color)" rel="stylesheet">

@import url(print-color.css) print and (color);
```

Медиа-запросы можно использовать в любом месте кода, в котором разрешено указывать тип носителя, используемого для вывода документа. Следуя этому принципу, пример из предыдущего раздела можно переписать, включив в него несколько запросов, разделенных запятыми.

```
<link href="print-color.css" type="text/css"
      media="print and (color), screen and (color-depth: 8)"
      rel="stylesheet">

@import url(print-color.css) print and (color), screen and
   (color-depth: 8);
```

Данный код указывает применять таблицу стилей к документу при выполнении условий хотя бы одного из запросов. В частности, команда `@import` требует применять внешнюю таблицу стилей `print-color.css` при цветной печати документа *или* отображении его на полноцветном экране. Если документ отправляется на черно-белый принтер, то не выполняются условия обоих запросов, и таблица стилей `print-color.css` к документу не применяется. Такой же результат будет получен при выводе документа, например, на монохромном мониторе.

В каждом дескрипторе технических характеристик указывается тип носителя, а также одна или несколько технических характеристик, заключенных в скобки. Если тип носителя не указан, то подразумевается значение `all` (любые носители). Следующие два примера абсолютно равнозначны.

```
@media all and (min-resolution: 96dpi) {...}

@media (min-resolution: 96dpi) {...}
```

В общем случае технические характеристики и их значения указываются в медиа-запросе в таком же формате, как и в объявлениях *свойство–значение*. Исключение составляют только технические характеристики, для которых значение не указывается. Например, любое устройство, обеспечивающее вывод документа в цвете, представляется короткой записью `(color)`, в то время как для указания устройства вывода с глубиной цвета 16 бит необходимо использовать запись `(color: 16)`. Чтобы понять, как рассматривать запись без значения, необходимо узнать, обладает ли носитель указанной технической характеристикой. В случае записи `(color)` необходимо ответить на вопрос: выводится ли документ в цвете?

В объявлениях медиа-запросов, включающих несколько технических критериев, применяются два логических оператора: `and` и `not`.

`and`

Объединяет два или более условия, которые должны быть истинными для выполнения запроса. Например, запись `(color) and (orientation: land scape) and (min-device-width: 800px)` выдвигает сразу три технических требования: документ должен выводиться на цветной носитель, поддерживающий показ изображения в альбомной ориентации и имеющий ширину экрана не менее 800 пикселей.

`not`

Отрицает условие, перед которым стоит, не позволяя выполняться запросу, если все его условия истинны. Например, при истинности всех условий записи `not (color) and (orientation: landscape) and (min-device-width: 800px)` запрос выполняться не будет. Буквально это означает, что документ *не будет* выводиться на цветном носителе, поддерживающем показ изображения в альбомной ориентации и имеющем ширину экрана более 800 пикселей. Для вывода документа подходят только носители, характеристики которых отличаются от указанных.

Обратите внимание на то, что логический оператор `not` добавляется только в начало дескриптора технических характеристик. Не допускается использовать его в середине записи, например так: `(color) and not (min-device-width: 800px)`. Подобные запросы браузерами попросту игнорируются. Более того, старые браузеры, не поддерживающие работу с медиа-запросами, никогда не применяют таблицы стилей, дескрипторы технических характеристик которых начинаются с ключевого слова `not`.

Логический оператор `or` (или) в медиа-запросах не применяется. Вместо него список условий разделяется запятыми. Например, для применения таблицы стилей в случае вывода документа на экране монитора или при печати с помощью принтера используется запись `screen, print`. Кроме того, неправильным считается условие `screen and (max-color: 2) or (monochrome)` — вместо него нужно использовать запись `screen and (max-color: 2), screen and (monochrome)`.

Нами упущено еще одно ключевое слово, играющее важную роль в медиа-запросах. Речь идет об операторе `only`, обеспечивающем обратную совместимость медиа-запросов со старыми браузерами.

`only`

Скрывает таблицу стилей от браузеров, не поддерживающих работу с медиа-запросами. Например, команда `@import url(new.css) only all` требует применять таблицу стилей только к документам, отображаемым в браузерах, которые распознают медиа-запросы. Современные браузеры, снабженные инструментами обработки медиа-запросов, игнорируют ключевое слово `only` и применяют указанную таблицу стилей ко всем (`all`) носителям. Старые браузеры воспринимают запись как неправильную, поскольку ключевому слову `only all` не сопоставляется ни один из известных типов носителей. В результате таблица стилей не подключается к документам, открытым в таких браузерах. Заметьте, что ключевое слово `only` вводится исключительно в начале медиа-запроса.

Дескрипторы технических характеристик и их значения

В предыдущем разделе было представлено только несколько примеров использования дескрипторов технических характеристик для стилевого форматирования документов. Ниже приведен полный список ключевых слов, используемых в медиазапросах при составлении критериев применимости таблиц стилей к документам, выводимым на носителях с требуемыми рабочими параметрами. Данные справедливы на конец 2017 года.

- `width`
- `min-width`
- `max-width`
- `device-width`
- `min-device-width`
- `max-device-width`
- `height`
- `min-height`
- `max-height`
- `device-height`
- `min-device-height`
- `max-device-height`
- `aspect-ratio`
- `min-aspect-ratio`
- `max-aspect-ratio`
- `device-aspectratio`
- `min-device-aspectratio`
- `max-device-aspectratio`
- `color`
- `min-color`
- `max-color`
- `color-index`
- `min-color-index`
- `max-color-index`
- `monochrome`
- `min-monochrome`
- `max-monochrome`
- `resolution`
- `min-resolution`
- `max-resolution`
- `orientation`
- `scan`
- `grid`

В спецификацию также добавлены два новых типа значений:

- *<ratio>*
- *<resolution>*

Детально назначение большинства дескрипторов технических характеристик описано в главе 20.

Запросы поддержки

На стыке 2015 и 2016 гг. в CSS была добавлена возможность применения таблиц стилей к документу, который отображается в пользовательском агенте, распознающем определенные свойства. Критерии, проверяющие применимость стилевых правил в браузерах, устанавливаются в *запросах поддержки*.

По своей структуре запросы поддержки подобны медиа-запросам. В качестве примера рассмотрим ситуацию, в которой цвет элемента изменяется только при поддержке браузером свойства `color`. (Разумеется, это так!) Код такого запроса выглядит следующим образом.

```
@supports (color: black) {
   body {color: black;}
   h1 {color: purple;}
   h2 {color: navy;}
}
```

В переводе на русский язык он означает: "Применить все приведенные ниже правила к документу только при распознавании объявления `color: black`. В противном случае отказаться от использования стилей". При выполнении кода в пользовательском агенте, не поддерживающем команду `@supports`, запрос полностью игнорируется.

Запросы поддержки существенно расширяют возможности браузеров по стилевому оформлению документа. Предположим, что в адаптивный макет документа, созданный традиционным способом, необходимо добавить элементы, позиционируемые по сетке. Для решения задачи включите следующий запрос в конец уже имеющейся таблицы стилей.

```
@supports (display: grid ) {
   section#main {display: grid;}
   /* код для отключения позиционирования элементов старым способом */
   /* стили позиционирования элементов по сетке */
}
```

Заданные в запросе поддержки стили будут применяться только в браузерах, распознающих инструменты верстки документов по сетке. В результате старые стили отменяются, и элементы документа позиционируются по сетке. В действительности старые браузеры, не поддерживающие работу с сетками документа, в большинстве своем не распознают команду `@supports` и будут полностью игнорировать новый код.

Запросы поддержки допускается вкладывать друг в друга, а также в медиа-запросы, и наоборот. Это позволяет, например, разграничить экранные стили и стили для

печати в адаптивном макете, для чего медиа-запросы каждого из блоков стилей заключаются в общую команду `@supports (display: flex)`.

```
@supports (display: flex) {
   @media screen {
      /* экранные стили адаптивного макета */
   }
   @media print {
      /* стили для печати адаптивного макета */
   }
}
```

Наряду с этим команду `@supports` допускается вкладывать в блок кода медиа-запроса, устанавливающего стили форматирования документа для различных носителей.

```
@media screen and (max-width: 30em){
   @supports (display: flex) {
      /* стили адаптивного макета для устройств с небольшим экраном */
   }
}
@media screen and (min-width: 30em) {
   @supports (display: flex) {
      /* стили адаптивного макета для устройств с большим экраном */
   }
}
```

Способ вложения и организации кода в запросах зависит исключительно от предпочтений разработчика.

В критериях запросов поддержки, как и в условиях медиа-запросов, разрешается использовать логические операторы. Рассмотрим пример, в котором стилевое оформление применяется только при поддержке пользовательским агентом модулей CSS Grid Layout и CSS Shapes.

```
@supports (display: grid) and (shape-outside: circle()) {
   /* стили верстки по сетке и управления фигурами CSS */
}
```

Без использования логического оператора приведенный выше код выглядел бы следующим образом.

```
@supports (display: grid) {
   @supports (shape-outside: circle()) {
      /* стили верстки по сетке и управления фигурами CSS */
   }
}
```

В запросах поддержки сложно обойтись одним только логическим оператором `and` (И). Важность оператора `or` (ИЛИ) при работе с фигурами CSS в первую очередь вызвана необходимостью их поддержки WebKit-браузерами, которая долгое время осуществлялась с помощью одних только вендорных свойств (подробно об этом — в главе 10). Следовательно, чтобы воспользоваться фигурами CSS в макете документа, в код необходимо добавить следующую программную структуру.

```
@supports (shape-outside: circle()) or
          (-webkit-shape-outside: circle()) {
   /* стили управления фигурами CSS */
}
```

Теперь при верстке документа можно использовать оба типа свойств: стандартные и вендорные. Приведенный выше код обеспечивает переход к управлению фигурами CSS с помощью вендорных свойств только в WebKit-браузерах, обращаясь к стандартным свойствам (разумеется, при должной поддержке пользовательским агентом) во всех остальных случаях.

Подобный способ применения свойств востребован во многих ситуациях, поскольку позволяет задавать разные способы форматирования в каждом из пользовательских окружений. Вернувшись к рассмотрению примера верстки по сетке, можно добиться установки разных полей и отступов для элементов, отображаемых в старых и новых браузерах. Ниже приведен упрощенный код, обеспечивающий решение этой задачи.

```
div#main {overflow: hidden;}
div.column {float: left; margin-right: 1em;}
div.column:last-child {margin-right: 0;}

@supports (display: grid) {
   div#main {display: grid; grid-gap: 1em 0;
      overflow: visible;}
   div#main div.column{margin: 0;}
}
```

Оператор логического отрицания также применим в командах `@supports`. Следующая инструкция определяет стили, применяемые к документу при отображении в браузере, который *не* поддерживает верстку элементов по сетке.

```
@supports not (display: grid) {
   /* стили оформления документа в отсутствие поддержки сетки */
}
```

Заключая условия в скобки, можно определить точный порядок применения логических операторов в запросе. Предположим, необходимо добиться стилевого оформления документа только в случае поддержки браузером свойства `color` и средств верстки по сетке или адаптивной блочной верстки. Эта задача решается с помощью такого кода.

```
@supports (color: black) and ((display: flex) or (display: grid)) {
   /* стили */
}
```

Обратите внимание на использование внешней пары скобок для объединения той части условия, которая разграничивается оператором "или". Если от них отказаться, то логическое условие выполняться не будет, и стили, определенные в блоке команды `@supports`, останутся невостребованными. Иными словами, следующая инструкция браузером не выполняется, а потому абсолютно бесполезна:

```
@supports (color: black) and (display: flex) or (display: grid) {
```

Возникает вопрос: зачем в логическом условии указывать не только названия свойств, но и их значения? Казалось бы, для проверки того, поддерживает ли браузер работу с фигурами CSS, достаточно удостовериться в распознавании им одного из свойств модуля CSS Shapes, например `shape-outside`. В действительности такая проверка будет неполной, поскольку браузер может поддерживать далеко не все значения свойства. Прекрасный пример тому — свойства позиционирования элементов по сетке. В частности, ошибочной будет следующая проверка поддержки браузером средств верстки по сетке.

```
@supports (display) {
   /* стили верстки по сетке */
}
```

Разумеется, даже Internet Explorer 4 распознавал свойство `display` и многие его значения. То же самое можно сказать о любом другом браузере, поддерживающем команду `@supports`. Но далеко не все они распознают значение `grid` — именно поэтому в условиях запроса поддержки нужно указывать конечное свойство и его значение.

С помощью команды `@supports` создаются запросы поддержки, а не запросы корректности стилевого форматирования. Браузер может поддерживать работу со свойствами, указанными в запросе, но применять их к документу неправильно. Нельзя с уверенностью утверждать, что браузер применяет стили к документу так, как предполагает спецификация. Прохождение проверки на поддержку свойства, указанного в запросе, означает только то, что браузер распознает запрос и выполняет в ответ на него некие действия.

Резюме

Технология CSS позволяет полностью изменить визуальное оформление документа, отображаемого пользовательским агентом. На базовом уровне внешний вид элементов устанавливается свойством `display`, а дополнительные возможности по форматированию открываются при использовании таблиц стилей CSS. С точки зрения пользователя не существует разницы между внешними, внедренными и встроенными таблицами стилей. Преимущество внешних таблиц стилей — в разделении содержимого документа и данных о его форматировании, что существенно упрощает редактирование стилей и внесение изменений в макет документа. Кроме того, такой подход позволяет сократить объем загружаемых данных и ускорить отображение документа в браузере. Повысить эффективность и расширить область применения стилей можно с помощью запросов поддержки (команда `@supports`).

Для правильного использования стилей CSS необходимо научиться привязывать стили к элементам документа. В следующей главе рассказано о том, как CSS определяет элементы, к которым применяются стили.

ГЛАВА 2

Селекторы

Главное преимущество CSS заключается в простоте назначения стилевого форматирования однотипным элементам. Вскоре вы убедитесь, что для замены цвета всех заголовков документа достаточно внести изменения всего в одну строку кода CSS. Не нравится синий цвет? Замените его красным, желтым, бордовым или любым другим, изменив название цвета всего в одной строке кода. Такой подход позволяет дизайнерам сконцентрироваться исключительно на визуальном оформлении документов, избавляя от необходимости многократного повторения одних и тех же действий. Зная, как стили применяются в документе, вы легко удивите коллег или друзей, изменив всего несколько строк кода и перезагрузив страницу. *Вуаля!* Всего несколько щелчков мышью — и цвет заголовков изменился во всем документе.

Не стоит думать, будто CSS избавит вас от любых неприятностей. Для цветовой коррекции PNG-изображений вам по-прежнему придется использовать графические редакторы, хотя для глобального форматирования документа CSS подходит как нельзя лучше. Начать знакомство с возможностями CSS лучше всего с изучения селекторов и структуры документа.

Базовые правила

Вы уже знаете, что основное преимущество CSS состоит в применении отдельных правил стилевого форматирования сразу ко всем элементам одного типа. Предположим, во всем документе необходимо установить серый цвет текста для заголовков второго уровня. Если решать эту задачу с помощью одних только инструментов HTML, то каждый из элементов `h2` необходимо снабдить следующим кодом:

```
<font color="gray">...</font>
```

Такого же результата можно добиться, включив атрибут `style="color: gray"` в каждый элемент `h2`. Приведенные ниже строки кода HTML абсолютно равнозначны.

```
<h2><font color="gray">Текст элемента h2</font></h2>
<h2 style="color: gray;">Текст элемента h2</h2>
```

Вам придется изрядно постараться, чтобы должным образом отформатировать документ, содержащий большое количество заголовков второго уровня. Хуже того,

для дальнейшего их изменения нужно будет отредактировать теги всех без исключения элементов h2. (Да, именно так это делается в "чистом" HTML!)

В отличие от HTML, технология CSS позволяет создать общее правило форматирования текста, которое применяется сразу ко всем указанным в нем элементам. (О том, что такое правило, рассказано в следующем разделе.) В частности, для представления текста всех заголовков второго уровня в сером цвете применяется такой код:

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

Чтобы изменить расцветку элементов h2 во всем документе, достаточно указать имя другого цвета:

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

Селекторы

Селектор элемента чаще всего, но далеко не всегда, указывает на HTML-элемент. Например, в случае использования CSS для форматирования XML-документа селекторы элементов выглядят следующим образом.

```
quote {color: gray;}
bib {color: red;}
booktitle {color: purple;}
myElement {color: red;}
```

Иными словами, селекторы указывают на элементы, выступающие в качестве базовых структурных компонентов документа. В технологии XML, позволяющей создавать собственные расширения языка разметки документов, в качестве селекторов могут использоваться произвольные имена. Наряду с этим в документах HTML селекторы представлены исключительно названиями элементов, оговоренных в спецификации языка (в том числе и html).

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

Результат применения форматирования, устанавливаемого приведенным выше кодом, показан на рис. 2.1.

После назначения всех необходимых стилей в документе применение уже имеющегося форматирования к другому элементу сводится к замене селектора в объявлении стиля. Например, для окрашивания текста абзаца в серый цвет, ранее назначенный заголовкам первого уровня (элемент h1), в приведенном выше коде селектор h1 нужно заменить селектором p.

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

Результат применения такого стилевого форматирования показан на рис. 2.2.

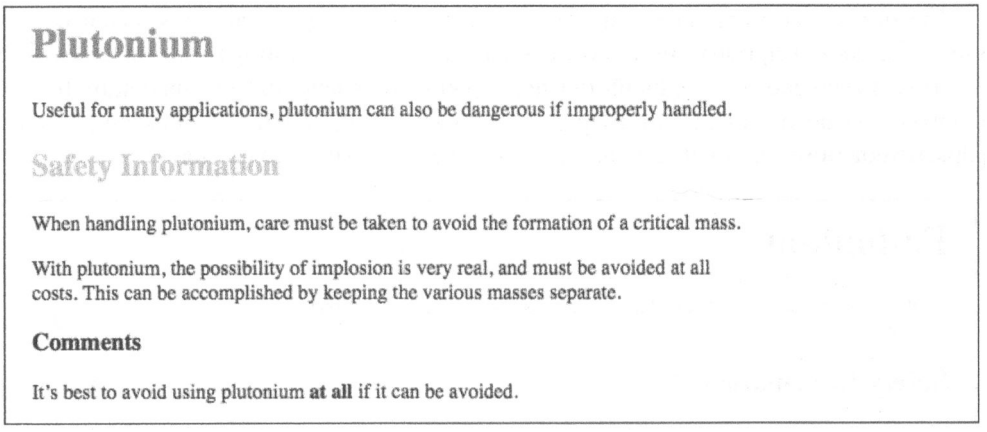


Рис. 2.1. Форматирование простого документа

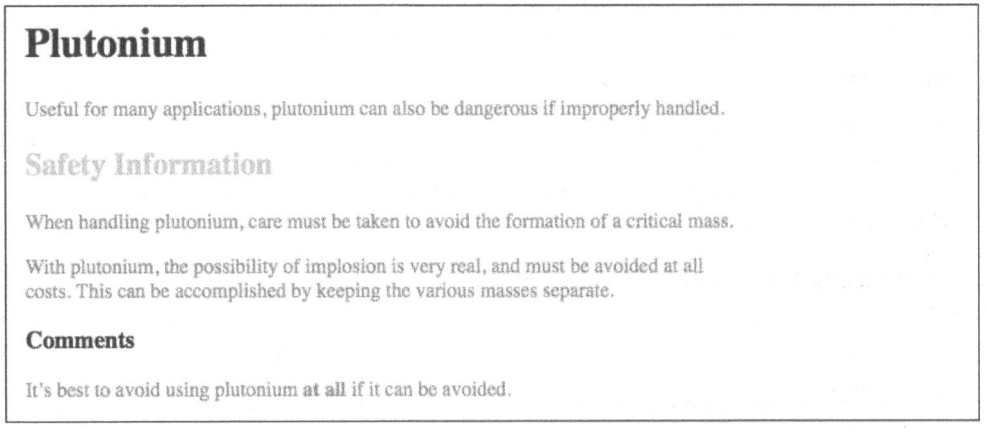


Рис. 2.2. Передача стиля от одного элемента к другому

Объявление стилей и ключевые слова

Блок объявлений включает одно или несколько объявлений правил. Каждое объявление состоит из названия свойства, двоеточия и значения. В конец объявления в обязательном порядке добавляется точка с запятой. Двоеточие и точка с запятой отделяются от остальных компонентов правила произвольным количеством пробелов. В подавляющем большинстве случаев *значение* представляется ключевым словом или списком разделенных пробелами ключевых слов, допустимых для назначения свойству. Если в качестве значения свойства указать недопустимое ключевое слово, то объявление будет проигнорировано. В частности, следующие два объявления недействительны.

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

При определении свойству значения, состоящего из нескольких ключевых слов, они разделяются пробелами, а в отдельных случаях — косой чертой или запятыми. Многие, но далеко не все, свойства поддерживают значения, включающие больше одного ключевого слова (например, `font`). В частности, на рис. 2.3 показан результат форматирования текста абзаца шрифтом Helvetica среднего размера.

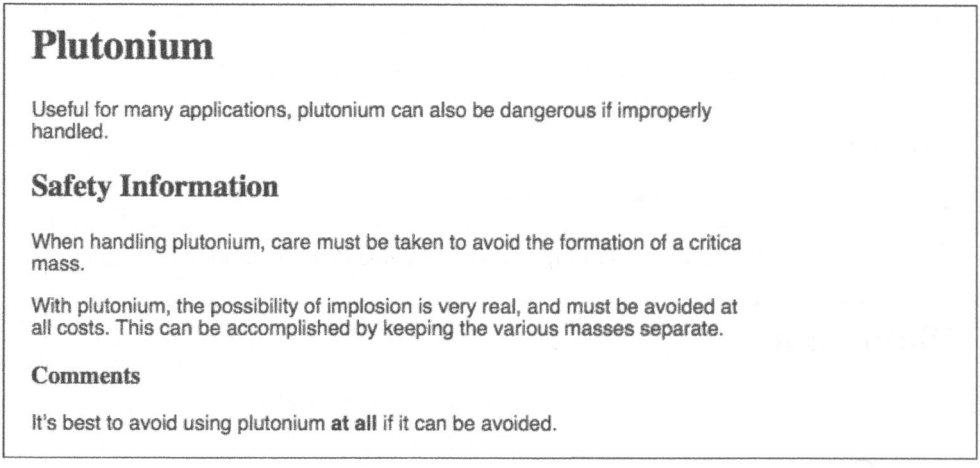


Рис. 2.3. Результат представления значения свойства `font` сразу несколькими ключевыми словами

Правило, применяемое для форматирования документа на рис. 2.3, имеет такой вид:

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

В данном случае ключевые слова `medium` (устанавливает размер шрифта) и `Helvetica` (представляет имя семейства шрифта) разделены пробелом. Пробел позволяет пользовательскому агенту корректно распознать составляющие компоненты значения и применить их к документу. На окончание объявления указывает точка с запятой.

Слова, разделяемые пробелами в объявлении правила, называются ключевыми, поскольку указывают на ключевые особенности свойства. Рассмотрим синтаксис вымышленного свойства `rainbow`:

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

В CSS нет свойства с названием `rainbow`; здесь оно приводится только для демонстрации синтаксических возможностей языка. Из примера видно, что значение свойства `rainbow` состоит из семи ключевых слов: `red orange yellow green blue indigo violet`. Это значение уникально и не может выражаться никаким другим образом. Изменим значение свойства `rainbow` так, как показано ниже.

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

Новое значение включает девять ключевых слов, в противоположность старому значению, в котором их всего семь. Легко заметить, что значения подобны, хотя и отличаются первым и последним ключевыми словами. В тоже время они считаются уникальными, и на этом построена концепция каскадного применения стилей в документе, описанная в следующих главах.

Как уже упоминалось, за редким исключением ключевые слова в синтаксических конструкциях CSS разделяются пробелами. Одно из таких исключений — значение свойства `font`, допускающее разделение определенных ключевых слов символом косой черты (`/`) в одной из позиций. Ниже приведен пример объявления свойства `font`.

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

В данном случае косая черта разделяет ключевые слова, указывающие размер шрифта и высоту строки. Разделение ключевых слов косой чертой допустимо только в одной позиции правила, объявляющего свойство `font`. Все остальные ключевые слова разделяются пробелами.

Косая черта встречается в значениях некоторых других свойств CSS. Ниже приведен неполный список таких свойств, который со временем будет расширяться.

- `background`
- `border-image`
- `border-radius`
- `grid`
- `grid-area`
- `grid-column`
- `grid-row`
- `grid-template`
- `mask-border`

В отдельных случаях ключевые слова могут разделяться запятыми. Такие ситуации возникают при назначении свойству сразу нескольких значений, например фоновых изображений, переходов или теней. Кроме того, запятыми в CSS разделяются параметры, передаваемые функциям, например настройки линейных градиентов и трансформации, что продемонстрировано в следующем примере.

```
.box {box-shadow: inset -1px -1px white, 3px 3px 3px rgba(0,0,0,0.2);
     background-image: url(myimage.png),
        linear-gradient(180deg, #FFF 0%, #000 100%);
     transform: translate(100px, 200px);
}
a:hover {transition: color, background-color 200ms ease-in 50ms;}
```

Выше описан синтаксис только простых объявлений стилевых правил. В реальных документах он намного сложнее. Начиная со следующего раздела мы будет рассматривать сложные концепции CSS.

Группирование

До этого момента описывались простые приемы, заключающиеся в одномоментном объявлении стилей только для одного элемента. Как же быть, если стили нужно применить сразу ко многим элементам? В подобных случаях в правиле нужно указывать несколько селекторов или последовательно применять каждый из стилей к группе элементов.

Группирование селекторов

Рассмотрим ситуацию, когда серым цветом нужно окрасить не только заголовки второго уровня, но и текст абзаца. Ниже показан самый простой способ решения этой задачи.

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

Для одновременного применения стиля к элементам `h2` и `p` их селекторы добавляются в левую часть правила и разделяются запятыми. Правая часть правила содержит стандартное объявление, в котором указывается способ форматирования указанных элементов: `(color:gray)`. Запятая позволяет браузеру корректно распознать все элементы и применить правило к ним обоим. Если опустить запятую, то правило приобретает совершенно иное значение, о чем будет рассказано в разделе "Контекстные селекторы".

Не существует ограничений на количество селекторов, указываемых в правиле. Например, если шрифт серого цвета необходимо назначить сразу многим элементам, то применяется следующая длинная конструкция.

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

Группирование селекторов позволяет сократить количество объявлений свойств в коде CSS и сделать таблицу стилей компактнее. Два следующих варианта кода равнозначны, но первый выглядит более громоздко и требует намного больше времени для ввода.

```
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;}
```

Группирование селекторов раскрывает дополнительные возможности. В частности, все три приведенных ниже блока кода, демонстрирующие разные методы группирования селекторов в стилевых правилах, выполняют одни и те же действия.

```
/* 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;}
```

Результат выполнения каждого из фрагментов показан на рис. 2.4. (Все представленные выше стили описаны в следующем разделе.)

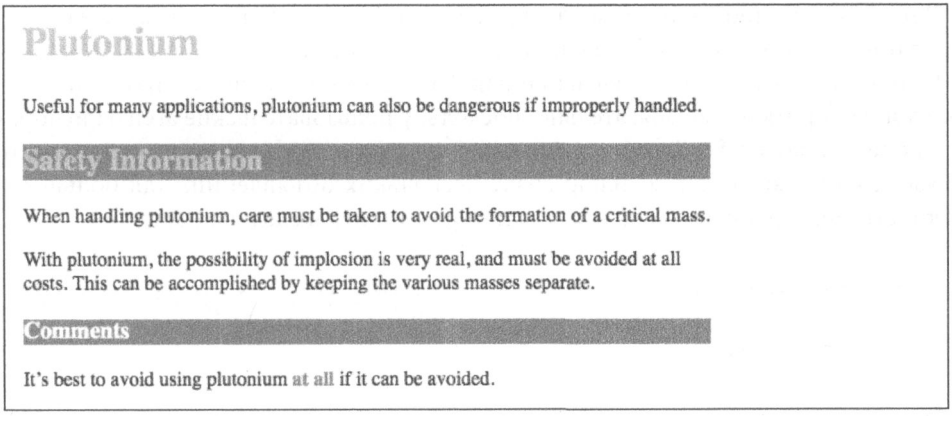


Рис. 2.4. Результат применения равнозначных таблиц стилей

Универсальный селектор

В CSS2 впервые появился универсальный селектор, представленный символом "звездочка" (*). Он указывает сразу на все элементы документа. Например, следующий код применяется, чтобы назначить красный цвет каждому единичному элементу документа:

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

Универсальное объявление подобно объявлению правила со сгруппированными селекторами, указывающего на все элементы, к которым применяется стилевое форматирование. Таким образом, всего одна строка кода позволяет назначить красный цвет каждому элементу документа. Будьте внимательны при использовании универсального селектора: несмотря на явное удобство использования, одновременное назначение общего форматирования всем элементам документа чревато непредвиденными последствиями, о чем рассказывается в последующих главах.

Группирование объявлений

Зная о группировании селекторов, логично предположить, что подобная возможность также свойственна и объявлениям. Рассмотрим задачу, в которой заголовки первого уровня (элемент `h1`) необходимо представить шрифтом Helvetica фиолетового цвета размером 18 пикселей, расположив на фоне цвета морской волны (нас не волнует, что о столь диком сочетании цветов подумают пользователи). В простейшем случае указанное форматирование применяется с помощью следующего CSS-кода.

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

Легко заметить, что такой подход неэффективен — представьте аналогичный код, устанавливающий форматирование с помощью 10 или 15 стилей. Для его упрощения объявления группируются в единую конструкцию.

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

В результате выполнения такого кода к элементу `h1` применяется такое же форматирование, как и в случае объявления трех отдельных правил.

Обратите внимание на разделение пар "свойство–значение" символами точки с запятой. Их применение обязательно, поскольку пользовательские агенты игнорируют пробелы в коде CSS. Точки с запятыми указывают синтаксическому анализатору браузера на включение в правило сразу нескольких объявлений. Для большей наглядности приведенный выше код можно представить в следующем виде.

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

Для минимизации кода из него можно вообще удалить не учитываемые пробелы.

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

Синтаксически все три способа форматирования кода равнозначны, но второй вариант самый удобочитаемый, поэтому рекомендуется использовать именно его. (Не стоит минимизировать код таблицы стилей только для того, чтобы ускорить передачу ее файла через сетевое соединение, поскольку намного эффективнее эта задача решается с помощью специального программного обеспечения, запускаемого на стороне сервера.)

Если не ввести точку с запятой в конце второй строки, то код будет распознаваться пользовательским агентом так, как показано ниже.

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

Данное объявление не будет обрабатываться браузером, поскольку значение свойства `color` не может состоять из нескольких ключевых слов, к тому же в его спецификации отсутствует ключевое слово `background:` (в том числе как часть значения `background: aqua`). Несмотря на правильное указание первой части выражения — `color: purple`, — браузер проигнорирует всю строку и не применит к элементу `h1` вообще никакого цветового форматирования. В результате текст заголовков первого уровня будет отображаться цветом по умолчанию (обычно черным) на прозрачном фоне (также устанавливается по умолчанию). Символом точки с запятой оканчивается только первая строка правила (`font: 18px Helvetica`), поэтому действительной остается только она.

В CSS точку с запятой допускается упускать только в конце последнего объявления, но это не совсем хорошая идея. Во-первых, добавляя точку с запятой в конец каждого объявления, можно выработать правильный стиль кодирования таблиц стилей CSS, ведь их отсутствие является едва ли не самой распространенной синтаксической ошибкой. Во-вторых, в случае добавления в правило новых объявлений не нужно беспокоиться о том, что новый код внесет неразбериху в уже существующий. И в-третьих, использование препроцессора Sass требует добавления точек с запятыми в конец всех без исключения объявлений. Таким образом, выработав привычку вставлять точку с запятой в конец каждого объявления, вы избежите большого количества проблем при использовании и видоизменении правил стилевого форматирования.

Группирование объявлений, как и селекторов, позволяет существенно сократить объем кода в стилевых правилах, упростить их последующее редактирование и повысить удобочитаемость.

Совместное группирование

Вы уже ознакомились со способами группирования селекторов и объявлений. Используя их в реальных документах, можно добиться заметного изменения внешнего вида документа с помощью всего нескольких инструкций CSS. Предположим, что ко всем заголовкам документа нужно добавить общее форматирование, затрагивающее большое количество параметров. Соответствующий код CSS выглядит следующим образом.

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

Группирование селекторов позволяет включить в правило имена заголовков всех доступных в документе уровней. Объявления, сгруппированные в фигурных скобках, предписывают применять устанавливаемое ими форматирование ко всем заголовкам, указанным в левой части инструкции. Результат применения правила к документу показан на рис. 2.5.

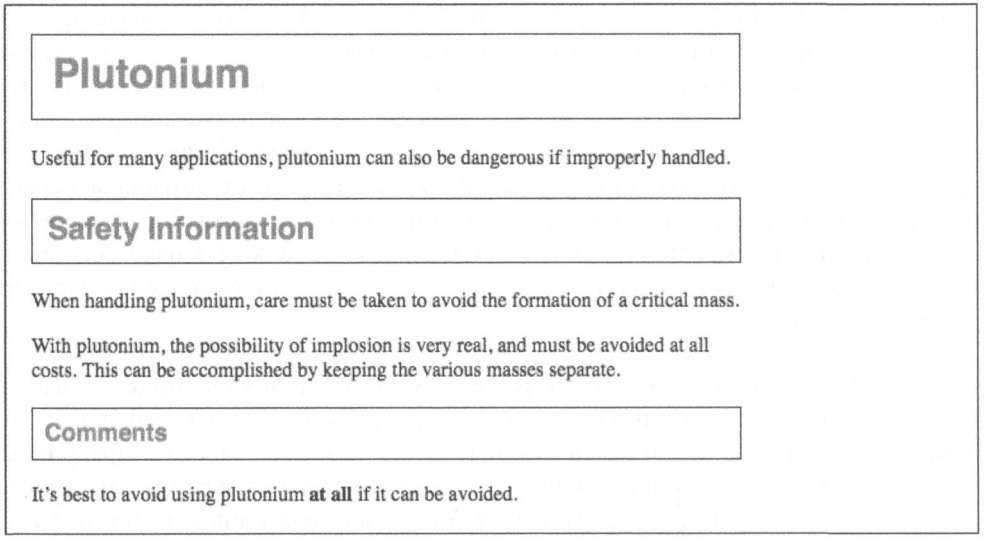


Рис. 2.5. Применение к документу единственного правила, содержащего сгруппированные селекторы и объявления

Такой способ представления правил в коде CSS предпочтительнее стандартного, в котором форматирование каждого из элементов устанавливается по отдельности. Ниже показана только небольшая часть кода, отвечающего за аналогическое стилевое оформление, выполняемое стандартными методами объявления свойств.

```
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;}
```

Полный код равнозначной таблицы стилей занимает намного больше строк — такой способ кодирования крайне нежелателен, поскольку его ввод и редактирование, подобно непосредственному указанию стилей в атрибуте `style` каждого из целевых элементов, длится целую вечность.

Поддержка новых элементов старыми браузерами

Обновление HTML до версии HTML5 знаменуется включением в спецификацию новых элементов. Отдельные браузеры не распознают такие элементы и попросту не отображают их в документе. В частности, Internet Explorer до версии 9 не позволял выделять не распознаваемые им элементы. Чтобы воспользоваться новым элементом в старых браузерах, его необходимо создать, включив в объектную модель документа.

Например, Internet Explorer 8 не распознает элемент `main`. Для его включения в документ с информированием об этом браузера применяется такой код JavaScript:

```
document.createElement('main');
```

Только после выполнения этого кода новый элемент становится доступным для управления и стилевого оформления старыми браузерами.

Селекторы идентификаторов и классов

В процессе описания способов группировки идентификаторов и объявлений нами рассматривались только такие примеры, в которых используются простые селекторы, указывающие на отдельные элементы документа. Селекторы элементов в коде CSS применяются очень часто, но обойтись только ими одними при стилевом форматировании документов не представляется возможным.

В дополнение к селекторам элементов в CSS допускается использовать *селекторы классов* и *селекторы идентификаторов* (ID). Последние позволяют изменять оформление документа без привязки к названиям элементов. Селекторы классов и идентификаторов применяются как отдельно, так и в сочетании с селекторами элементов. Тем не менее эффективность их использования напрямую зависит от правильности разметки документа, а потому требует предварительного планирования и точного расчета.

Представим, что перед нами стоит задача задать внешний вид документа, описывающего методики обеспечения безопасности при работе с плутонием. Такой документ наверняка будет включать большое количество предупреждений и правил, от выполнения которых зависит здоровье рабочего персонала. Каждое из предупреждений нужно акцентировать в тексте документа полужирным начертанием. Ситуация осложняется тем, что предупреждения появляются в произвольных местах документа, а потому представляются самыми разными элементами: целыми абзацами, отдельными элементами списка и текстовыми фрагментами. Таким образом, определять стилевое форматирование предупреждений с помощью селекторов элементов не представляется возможным. Чтобы понять почему, рассмотрим следующий пример.

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

В результате выполнения кода полужирное начертание и красный цвет шрифта получат все без исключения абзацы документа. Нам же необходимо выделить только те из них, которые содержат предупреждения. Наряду с этим нужно выделить предупреждения, представленные не только абзацами, но и любыми другими элементами. Чтобы применить одинаковое форматирование к разнотипным элементам документа, их необходимо пометить специальным образом и представить селекторами класса в стилевых правилах.

Селекторы классов

Самый простой способ задания форматирования для отдельных частей документа без привязки к типизируемым стандартным образом элементам заключается в использовании *селекторов классов*. Но, перед тем как включать их в стилевые правила, необходимо изменить разметку документа, снабдив целевые элементы атрибутом `class`.

```
<p class="warning">When handling plutonium, care must be taken to
avoid the formation of a critical mass.</p>
<p>With plutonium, <span class="warning">the possibility of implosion
is very real, and must be avoided at all costs</span>.
This can be accomplished by keeping the various masses separate.</p>
```

Атрибут `class` позволяет связать стиль, в котором определен селектор класса, с содержащим его элементом. В предыдущем фрагменте кода атрибут `class` со значением `warning` добавлен сразу к двум элементам: первому абзацу и элементу `span` второго абзаца.

Рассмотрим, каким способом устанавливаются стилевые правила, которые применяются к элементам, относящимся к специально оговоренным в документе классам. В правилах CSS допускается использовать короткую форму записи селектора класса, предварив имя класса точкой, как показано ниже.

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

В сочетании с приведенным выше HTML-кодом текущее правило видоизменяет документ так, как показано на рис. 2.6. В частности, объявление `font-weight: bold` назначается каждому элементу, относящемуся к классу `warning`, как предполагает неявный универсальный селектор.

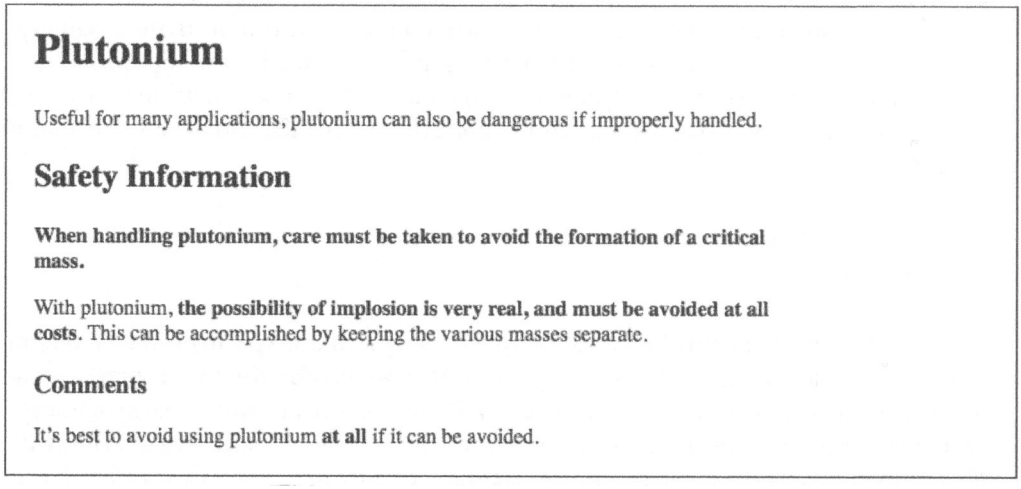


Рис. 2.6. Форматирование, установленное с помощью селектора класса

Универсальный селектор, представленный символом "звездочка" (*), используется в селекторах идентификаторов, классов, атрибутов, псевдоклассов и псевдоэлементов, задаваемых без привязки к названиям элементов документа.

Селектор класса представляет элементы, атрибут `class` которых установлен в значение, отраженное в его названии. Имя селектора класса *всегда* предваряется символом точки (.), который отличает его от селекторов остальных типов и отделяет имя класса от элементов класса, к которым применяется стилевое форматирование. В следующем примере полужирное начертание применяется не ко всем членам класса, а только к абзацам. Таким образом, в тексте документа выделяются не все предупреждения, а только те, что представлены целыми абзацами.

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

Такой селектор указывает только на элементы `p`, снабженные атрибутом `class` со значением `warning`. Никакие другие элементы с классом `warning` с его помощью не выбираются. Так как элемент `span` не является абзацем, полужирное начертание для него не устанавливается.

Чтобы задать элементам `span`, относящимся к классу `warning`, иное форматирование, следует применить селектор `span.warning`.

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

Таким образом, предупреждение, выводимое в абзаце, отображается полужирным начертанием, а предупреждение, встроенное в строку, — курсивом. Поскольку каждое правило в точности определяет целевой элемент/класс, оно не будет воздействовать на любые другие элементы документа.

Широкий спектр возможностей открывается при комбинировании общего селектора класса с селекторами классов отдельных элементов. Такой подход расширяет возможности форматирования документа стилевыми правилами, что демонстрирует следующий код.

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

Результат применения данного кода к описанному выше документу HTML приведен на рис. 2.7.

Как видите, теперь курсивом выделяются все предупреждения, и только те из них, что представлены элементами `span`, относящимися к классу `warning`, дополнительно выделены полужирным начертанием.

Обратите внимание на формат представления имени селектора класса в предыдущем примере. Оно состоит из названия класса, предваряемого точкой, перед которой не указывается ни имя элемента, ни универсальный селектор. Если стилевое правило применяется ко всем элементам класса, то добавлять символ * в начало селектора класса нет никакой необходимости.

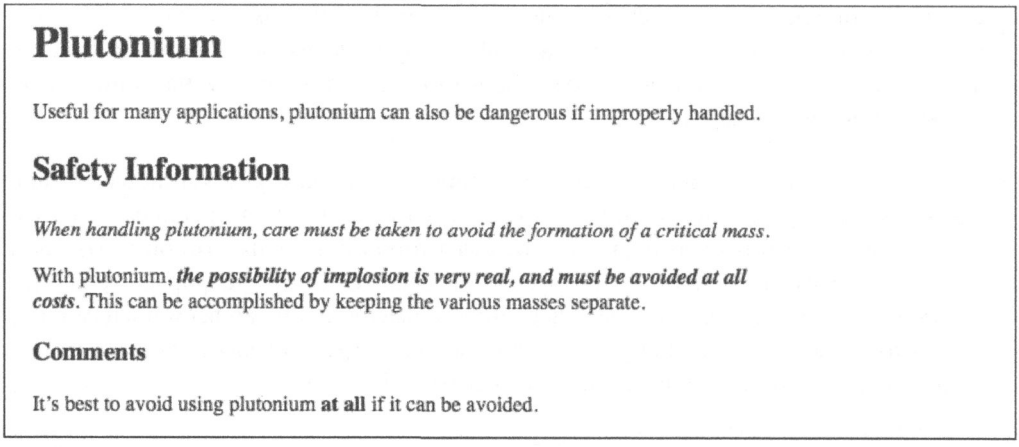


Рис. 2.7. Форматирование, установленное правилами, которые включают общий селектор класса и селектор класса для отдельных элементов

Комбинированные классы

В предыдущем разделе рассматривались селекторы классов, названия которых состоят только из одного ключевого слова. Тем не менее в HTML для атрибута `class` допустимы значения, состоящие из нескольких ключевых слов, разделенных пробелами. Например, следующий код разметки предполагает отнесение отдельных элементов документа сразу к двум классам: предупреждения (`warning`) и настоятельного предупреждения (`urgent`).

```
<p class="urgent warning">When handling plutonium, care must be taken
to avoid the formation of a critical mass.</p>
<p>With plutonium, <span class="warning">the possibility of implosion
is very real, and must be avoided at all costs</span>. This can be
accomplished by keeping the various masses separate.</p>
```

Порядок следования ключевых слов в имени комбинированного класса не играет особой роли. Точно такой же результат будет получен при установке атрибуту `class` значения `warning urgent`.

В приведенном ниже коде CSS предполагается, что элементы класса `warning` будут выделяться полужирным начертанием, элементы класса `urgent` — курсивом, а элементы, принадлежащие к обоим классам, — серебристым фоном.

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

Как и в атрибуте HTML, порядок следования названий классов в селекторе правила CSS не играет особой роли. Но в отличие от HTML, в котором ключевые слова в значении атрибута разделяются пробелами, в селекторе CSS в качестве разделителя значения применяется точка. При подключении приведенного выше кода CSS к документу HTML серебристым фоном выделяется только абзац "When handling plutonium..."

(рис. 2.8). Как видите, порядок указания названий классов в значении атрибута и селекторе действительно не столь важен. (При этом порядок отнесения элементов к классам порой имеет особую важность, о чем рассказано в следующих главах.)

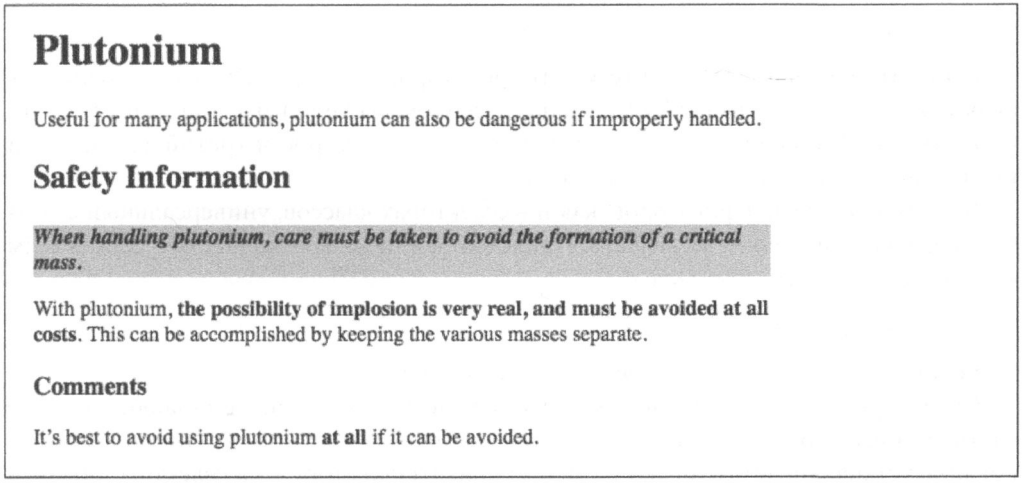


Рис. 2.8. Выделение элемента, принадлежащего сразу к нескольким классам

Если селектору, включающему название нескольких классов, не сопоставляется атрибут HTML с такими же именами классов в значении, разделенными пробелами, то стилевое правило в документе не применятся. Рассмотрим следующий пример:

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

Можно ожидать, что селектор данного правила указывает на все абзацы, относящиеся к классам `warning` и `help` (определяется атрибутом `class` элемента `p`). Наряду с этим, если в HTML-документе в элементах `p` объявлены только классы `warning` и `urgent`, то приведенное выше стилевое правило к ним применяться не будет. Однако с его помощью на странице будут выделяться следующие элементы `p`:

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

Селекторы идентификаторов

Несмотря на схожесть, селекторы классов и идентификаторов имеют несколько важных отличий. Во-первых, селекторы идентификаторов начинаются с символа "решетки" (`#`). Ниже показано, как выглядит правило CSS, содержащее селектор идентификатора.

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

Данное правило определяет полужирное начертание всем элементам, атрибут `id` которых установлен в значение `first-para`.

Из этого следует второе отличие: селектор идентификатора в правиле CSS соответствует значению атрибута `id` элемента HTML (но никак не атрибута `class`). Ниже

приведен фрагмент кода, в котором селектор идентификатора применяется для представления элемента, к которому применяется стилевое форматирование.

```
*#lead-para {font-weight: bold;}
<p id="lead-para">This paragraph will be boldfaced.</p>
<p>This paragraph will NOT be bold.</p>
```

Обратите внимание на то, что идентификатором `lead-para` можно обозначить любой элемент документа HTML — в приведенном выше примере он назначен первому абзацу. При необходимости им можно снабдить второй и третий абзацы. Или неупорядоченный список — да что угодно!

В селекторах идентификаторов, как и в селекторах классов, универсальный селектор (символ `*`) можно не указывать. Приведенный ниже код полностью сопоставим с кодом предыдущего примера.

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

Результат в обоих случаях будет одним и тем же.

Селекторы идентификаторов, как и селекторы классов, представляют целевую часть документа без привязки к названиям элементов. Во многих случаях форматирование устанавливается только по одному идентификатору — в селекторе правила целевой элемент не упоминается (как и в случае стилевого оформления предупреждений, описанного в предыдущем разделе). Например, при назначении стилей элементам с идентификатором `mostImpotant` знать его тип (абзац, текстовый фрагмент, элемент списка или заголовок) совсем не обязательно. Достаточно удостовериться, что такой элемент присутствует в документе, распознается синтаксическим анализатором и встречается единожды. Для форматирования такого элемента применим следующий код CSS:

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

Приведенное выше правило подходит для изменения внешнего вида следующих элементов (в отдельный документ можно включать только один из них, поскольку у них общий идентификатор).

```
<h1 id="mostImportant">This is important!</h1>
<em id="mostImportant">This is important!</em>
<ul id="mostImportant">This is important!</ul>
```

Выбор типа селектора

Как показано выше, классы разрешается назначать большому количеству элементов HTML. В рассмотренном примере класс `warning` присваивался только двум элементам: `p` и `span`. В общем случае атрибутом `class` можно снабдить огромное количество самых разных элементов документа. В противоположность классу один и тот же идентификатор назначается в документе только одному элементу. Таким образом, если в документе уже имеется элемент с идентификатором `lead-para`, то назначить этот же идентификатор другому элементу документа невозможно.

В действительности браузеры не проверяют загружаемые документы на уникальность имеющихся в них идентификаторов. В частности, это означает, что при назначении одного и того же идентификатора сразу нескольким элементам документа к ним всем будет применено форматирование, которое определяется стилевым правилом, включающим селектор данного идентификатора. Такое поведение документа считается неправильным, хотя и встречается повсеместно. Добавление в документ сразу нескольких неуникальных идентификаторов чревато множественными ошибками выполнения сценариев JavaScript, поскольку такие функции, как `getElementById()`, не предполагают назначение одного и того же идентификатора множественному числу элементов.

В отличие от селекторов классов, селекторы идентификаторов невозможно комбинировать с другими селекторами, поскольку атрибут `id` не позволяет хранить значения, состоящие из нескольких ключевых слов, разделенных пробелами.

Учтите, что идентификаторы имеют более высокий приоритет при назначении элементу стилевого форматирования, о чем подробно рассказано в следующем разделе.

Кроме того, в зависимости от языка разметки документа имена селекторов классов и идентификаторов могут быть чувствительны к регистру. В HTML при вводе названий классов и идентификаторов регистр символов нужно соблюдать в обязательном порядке. Именно поэтому в коде CSS селекторы классов и идентификаторов должны включать точно такие же значения, как в соответствующих атрибутах элементов документа. Таким образом, в приведенном ниже примере полужирное начертание не применяется к элементу.

```
p.criticalInfo {font-weight: bold;}
<p class="criticalinfo">Не смотреть вниз!</p>
```

Регистры символа *i* в значении атрибута элемента и селекторе правила CSS отличаются, что и является причиной неприменимости правила для форматирования документа HTML.

Строго говоря, спецификация XML не гарантирует распознавание селекторов классов, начинающихся с точки (например, `.warning`). На момент написания книги такой формат записи селекторов поддерживался только языками HTML, SVG, MathML и, возможно, будет разрешен в еще не разработанных языках разметки документов. В общем случае правила именования и идентификации элементов документа определяются спецификацией языка. Наряду с этим селекторы, начинающиеся с символа решетки (например, `#lead`), поддерживаются всеми без исключения языками разметки, в спецификацию которых включен атрибут, обеспечивающий элемент уникальностью в пределах документа. Для получения статуса уникального в элемент не обязательно включать атрибут `id` — достаточно снабдить его любым другим атрибутом, значение которого не повторяется больше нигде в документе.

Селекторы атрибутов

В действительности селекторы классов и идентификаторов представляют элементы, ссылаясь на значения включенных в них атрибутов. Синтаксис селекторов, рассмотренный в двух предыдущих разделах, справедлив для документов, созданных с помощью таких языков, как HTML, XHTML, SVG и MathML. Документы, разметка которых выполнена с помощью других языков, могут не поддерживать использование селекторов классов и идентификаторов (необходимые атрибуты могут попросту отсутствовать в их спецификации). Чтобы расширить возможности по форматированию таких документов, в CSS2 была представлена поддержка *селекторов атрибутов*, позволяющих ссылаться на элементы, атрибутам которых назначены определенные значения. Существуют четыре основных типа селекторов атрибутов: простые, селекторы значений, селекторы частичных значений и селекторы обязательных значений.

Простые селекторы атрибутов

Простые селекторы атрибутов используются для представления в стилевых правилах элементов документа, снабженных определенными атрибутами, независимо от принимаемых ими значений. Например, следующий код позволяет задать стилевое форматирование всем элементам `h1` документа, снабженным атрибутом `class`:

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

Если применить указанное правило к документу, разметка которого задана следующим кодом, то он примет внешний вид, показанный на рис. 2.9.

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

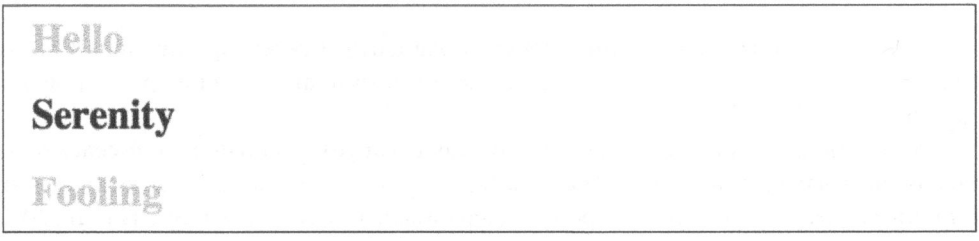


Рис. 2.9. Форматирование документа с помощью стилевого правила, включающего простой селектор атрибута

Простые селекторы атрибутов удобно применять в XML-документах, поскольку названия элементов и атрибутов в них задаются произвольным образом и чаще всего определяются назначением.

Рассмотрим документ XML, содержащий описание планет некой солнечной системы (назовем ее PlanetML). Для выделения полужирным начертанием всех элементов `pml-planet`, снабженных атрибутом `moon`, можно использовать такое правило CSS:

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

При подключении этого правила к следующему фрагменту XML-кода полужирным начертанием выделяются только второй и третий элементы.

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

В документах HTML простые селекторы атрибутов применяются более изобретательно. Например, с их помощью можно установить специальное форматирование для всех элементов, снабженных атрибутом `alt`, что позволяет выделить в документе все корректно подключенные изображения.

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

Данный метод часто применяется при отладке документов и позволяет отслеживать в документе все неправильно размещенные изображения.

Для назначения полужирного начертания всем элементам, снабженным атрибутом `title` (содержит текст, который выводится на экран в виде всплывающей подсказки, отображаемой при наведении указателя мыши на элемент), применяется такой код:

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

Подобным образом в документе можно выделить все гиперссылки (элементы `a`), снабженные атрибутом `href`, что позволяет применить целевое форматирование только к гиперссылкам, но не к анкерам заполнителей. В частности, для выделения полужирным начертанием гиперссылок, представленных атрибутами `href` и `title`, применяется такой код CSS:

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

В результате начертание изменяется у первой гиперссылки, но ни в одной из последующих.

```
<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>
```

Селектор атрибутов с точным значением

Указав точное значение в селекторе атрибутов, можно существенно сузить область применения стилевого форматирования. В частности, воспользовавшись таким подходом, можно легко выделить полужирным начертанием все гиперссылки, которые указывают на документ, хранящийся по строго заданному адресу. Ниже показан код соответствующего правила.

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

В результате в документе выделяются только те гиперссылки, в атрибуте `href` которых указано значение `http://www.css-discuss.org/about.html`. Текст гиперссылки должен в точности повторять указанный в селекторе (не допускается даже опускать префиксы `www.` и `https://`).

В селекторе можно указывать любые комбинации атрибутов и значений, допустимых для использования в элементах документа. Тем не менее, если селектор указывает на атрибут или значение, отсутствующие в документе, то стилевое форматирование в нем применяться не будет. С этой точки зрения документы, размеченные с помощью языка XML, позволяющего создавать произвольные элементы и атрибуты, имеют неоспоримое преимущество. Вернувшись к примеру с документом, описывающим солнечную систему PlanetML, напишем код выделения только тех элементов `planet`, атрибут `moons` которых имеет значение 1.

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

При его применении к следующему фрагменту XML-кода полужирное начертание устанавливается для второго элемента, но не первого и третьего.

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

Как и в более простом случае, в селектор атрибута с точным значением допускается включать сразу несколько условий. Например, для двукратного увеличения размера шрифта гиперссылок, снабженных атрибутом `href` со значением `http://www.w3.org/` и атрибутом `title` со значением `W3C Home`, используется следующий селектор:

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

В результате применения правила к следующему фрагменту HTML-документа шрифтом большего размера представляется первая гиперссылка, но не две последующие.

```
<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 форматирования гиперссылок показан на рис. 2.10.

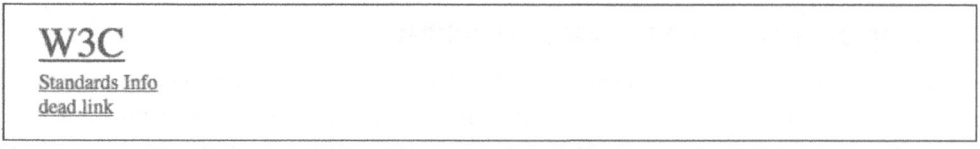


Рис. 2.10. Выделение элементов с помощью селекторов атрибутов и их значений

Снова-таки, в селекторе нужно указывать значения атрибутов, *в точности* повторяющие заданные в элементах документа. Особенно это касается значений, которые состоят из нескольких ключевых слов, разделенных пробелами (в частности, значений атрибута `class` в HTML). Рассмотрим следующий фрагмент кода HTML:

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

Единственно верный способ обращения к такому элементу в селекторе атрибута, снабженного точным значением, заключается в использовании следующего правила:

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

Правило гарантированно не будет выполняться браузером, если в качестве селектора использовать выражение `planet[type="barren"]`. Такое поведение пользовательского агента характерно даже для атрибута `class`. Рассмотрим следующий HTML-код:

```
<p class="urgent warning">When handling plutonium, care must be
taken to avoid the formation of a critical mass.</p>
```

Для стилевого форматирования данного элемента, снабженного атрибутом с точным значением, нужно применить следующее правило:

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

Следует отличать этот тип селекторов от селекторов класса (начинаются с символа точки), которые также ссылаются на значение атрибута, состоящего из нескольких ключевых слов. В данном примере правило применяется к элементу, снабженному атрибутом `class` со значением `"urgent warning"`. Для его выполнения порядок ключевых слов в селекторе должен в точности повторять указанный в атрибуте элемента.

Старайтесь избегать использования в коде CSS селекторов атрибутов, решающих те же задачи, что и селекторы идентификаторов. Иными словами, существует большая разница между селекторами `h1#page-title` и `h1[id="page-title"]`. Детально мы поговорим об этом в конце главы.

Селекторы атрибутов с частичными значениями

Во многих случаях стилевое форматирование элементов выполнять намного проще, указывая в селекторах атрибуты, значения которых только частично совпадают с заданными в документе HTML. Для задания частичного значения атрибута в селекторе правила применяется целый ряд специальных знаков (табл. 2.1).

Таблица 2.1. Подстановочные знаки, используемые в селекторах

Шаблон	Описание
`[foo|="bar"]`	Представляет элемент с атрибутом `foo`, значение которого начинается с подстроки `bar` с добавленным дефисом (U+002D) или представляется только ею
`[foo~="bar"]`	Представляет элемент с атрибутом `foo`, в значении которого встречается ключевое слово `bar`, отделенное пробелами от остальных ключевых слов
`[foo*="bar"]`	Представляет элемент с атрибутом `foo`, значение которого содержит подстроку `bar`
`[foo^="bar"]`	Представляет элемент с атрибутом `foo`, значение которого начинается с подстроки `bar`
`[foo$="bar"]`	Представляет элемент с атрибутом `foo`, значение которого заканчивается подстрокой `bar`

Правила, основанные на селекторах атрибутов с частичными значениями

Первый способ применения селекторов атрибутов с частичными значениями проще описать на реальном примере, чем попытаться объяснить на словах. Рассмотрим следующее правило:

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

Оно определяет форматирование элемента, значение атрибута `lang` которого представлено ключевым словом `en` или начинается с `en-`. Таким образом, применив его к приведенному ниже фрагменту HTML-кода, можно выделить цветом только первые три элемента — два последних элемента изменению не подлежат.

```
<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 случае селектор типа `[`*атрибут*`|="`*подстрока*`"]` применим к любым атрибутам и значениям. Рассмотрим в качестве примера HTML-документ, включающий несколько изображений с именами файлов `figure-1.gif`, `figure-2.gif`,`figure-3.jpg` и т.д. Следующее правило можно применять для назначения форматирования им всем:

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

При создании фреймворка CSS или библиотеки шаблонов можно отказаться от использования излишних классов, подобных `"btn btn-small btn-arrow btn-active"`, заменив их одним-единственным классом. В подобном случае все необходимое форматирование будет задаваться следующей программной конструкцией.

```
*[class|="btn"] {border-radius: 5px;}
<button class="btn-small-arrow-active">Щелкни тут!</button>
```

Наиболее часто этот подход применяется для форматирования элементов, текст которых введен на разных языках, о чем будет рассказано в разделе "Псевдокласс `:lang`".

Представление элементов ключевым словом, включенным в значение атрибута

Значения атрибутов могут состоять из нескольких ключевых слов, разделенных пробелами. Самый распространенный атрибут этой категории — `class`. В его значении допускается указывать как одно, так и несколько ключевых слов.

```
<p class="urgent warning">When handling plutonium, care must be
taken to avoid the formation of a critical mass.</p>
```

Предположим, форматирование нужно назначить только тем элементам документа, значение атрибута `class` которых включает слово `warning`. Эта задача решается с помощью такого стилевого правила:

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

Обратите внимание на добавление символа тильды (~) в селектор атрибута. Он указывает проводить поиск целевых элементов по ключевому слову, присутствующему в значении атрибута `class`, наряду с другими ключевыми словами, разделенными пробелами. Без него правило будет применяться к элементам, значение атрибута `class` которых в точности соответствует ключевому слову, включенному в селектор (см. предыдущий раздел).

Структурно селектор из предыдущего примера подобен селектору класса, предваряемого точкой (см. раздел "Выбор типа селектора"). В частности, селекторы `p.warning` и `p[class~="warning"]` полностью равнозначны при применении правил, в состав которых они входят, к документу HTML. Рассмотрим HTML-версию документа, содержащего описание солнечной системы PlanetML.

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

Следующее правило можно использовать для выделения курсивом всех элементов документа, в значении атрибута `class` которых содержится слово `barren`:

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

Селектор правила представляет первые два элемента из приведенного фрагмента HTML-кода, поэтому в браузере он будет выглядеть так, как показано на рис. 2.11. Аналогичный результат будет получен при использовании такого правила:

```
span.barren {font-style: italic;}
```

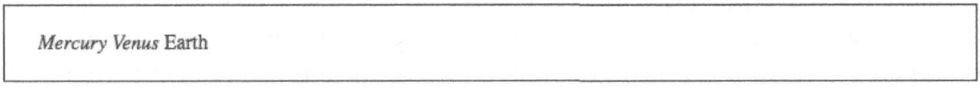


Рис. 2.11. Форматирование элементов, выбираемых по ключевым словам в значениях атрибутов

Возникает вопрос: зачем использовать подстановочный знак (тильда) в селекторе атрибута, если поставленную задачу можно решить с помощью селектора класса? Дело в том, что селектор атрибутов, содержащий подстановочные знаки, применим к любым элементам, а не только к снабженным атрибутом `class`. Предположим, исходный документ включает большое количество графических объектов, и только некоторые из них представляют изображения цифр. Для изменения их внешнего вида применяется следующее правило (его селектор представляет элементы, значение атрибута `title` которых включает строку `Figure`):

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

Таким образом, под форматирование подпадают элементы, атрибут `title` которых содержит текст наподобие "Figure 4. A baldheaded elder statesman". Наличие тильды в селекторе атрибута также позволяет применить стилевое правило к элементам с самыми разными подписями, включающими слово "Figure", например "How to Figure Out Who's in Charge". Правило не применяется к элементам, в значении атрибута `title` которых отсутствует ключевое слово `Figure`, а также к элементам, полностью лишенным атрибута `title`.

Представление элемента по частично совпадающему значению атрибута

В отдельных случаях форматирование нужно применить к элементам, выбираемым согласно атрибутам, значения которых включают определенную подстроку, не отделяемую пробелами от остальной части значения. Для решения такой задачи стилевое правило снабжается селектором вида `[`*`атрибут`*`*="`*`подстрока`*`"]`. Например, приведенный выше код CSS позволяет применить курсивное начертание к любому элементу `span`, в значение атрибута `class` которого включен текст `cloud`. Таким образом, под действие правила подпадают названия планет, относимых к классу `cloud`, что проиллюстрировано на рис. 2.12.

```
span[class*="cloud"] {font-style: italic;}
<span class="barren rocky">Mercury</span>

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

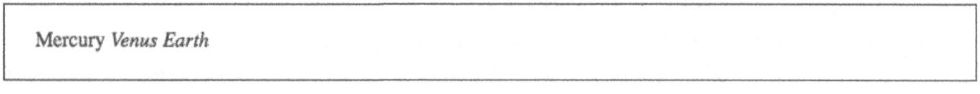


Рис. 2.12. Выделение элементов, значения атрибутов которых содержат требуемую подстроку

Описанная возможность находит широкое применение в CSS. С ее помощью, например, можно определить специальное форматирование для всех гиперссылок, указывающих на определенный сайт или ресурс. Вместо того чтобы выделять гиперссылки (элементы `a`) в отдельный класс и устанавливать форматирование с помощью селектора класса, можно использовать следующее простое правило:

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

Применять форматирование в HTML-документе по совпадающему значению допускается не только для атрибутов `class` и `href`, но и многих других, в том числе `title`, `alt`, `src`, `id` и т.п. Если у атрибута есть значение, то решение о применении к элементу правила зависит от того, содержит ли значение ключевое слово, указанное в селекторе. В частности, следующее правило применяется к каждому изображению, в URL которого содержится слово `"space"`:

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

Подобным образом приведенное ниже правило изменяет форматирование элементов `input`, в заголовке (атрибут `title`) которых указывается формат (`"format"`) ввода данных, запрашиваемых документом.

```
input[title*="format"] {background-color: #dedede;}

<input type="tel"
       title="Telephone number should be formatted as XXX-XXX-XXXX"
       pattern="\d{3}\-\d{3}\-\d{4}">
```

Обычно селекторы данного типа применяются для обозначения элементов, относящихся к классу, название которого представляется заранее оговоренным шаблоном. В продолжение предыдущего примера создадим правило, устанавливающее форматирование для элементов, имя класса которых начинается с ключевого слова `"btn"` и включает слово `"arrow"`, также предваряемое дефисом.

```
*[class|="btn"][class*="-arrow"]:after { content: "▼";}
<button class="btn-small-arrow-active">Click Me</button>
```

Селектор должен в точности совпадать с той частью значения атрибута, на которую он указывает. Таким образом, если селектор наряду с ключевыми словами содержит пробел, то такой же пробел должен стоять в соответствующем месте значения целевого атрибута. При этом чувствительность значения селектора к регистру полностью определяется спецификацией языка разметки документа, к которому применяется стилевое правило. В HTML чувствительны к регистру только имена классов, заголовки, гиперссылки и идентификаторы, но не значения атрибутов, поэтому приведенный ниже код вполне корректен.

```
input[type="CHeckBoX"] {margin-right: 10px;}
<input type="checkbox" name="rightmargin" value="10px">
```

Указание элемента по начальной подстроке значения

Селектор атрибута формата `[`*`атрибут`*`^="`*`подстрока`*`"]` представляет элементы, значения атрибутов которых начинаются с определенной подстроки. Такие селекторы, например, позволяют установить разное форматирование для гиперссылок разных типов (рис. 2.13).

```
a[href^="https:"] {font-weight: bold;}
a[href^="mailto:"] {font-style: italic;}
```

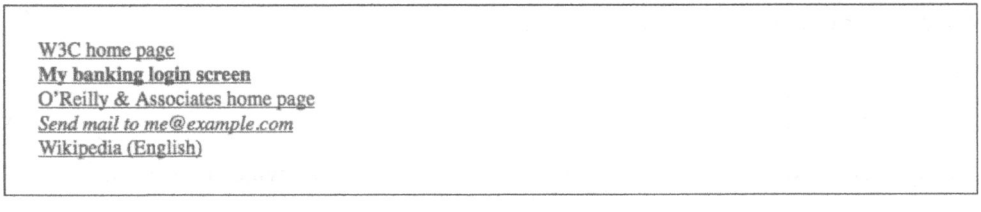


Рис. 2.13. Стилевое форматирование элементов, значение атрибута которых начинается с подстроки, указанной в селекторе правила

В качестве еще одного примера рассмотрим ситуацию назначения форматирования всем графическим объектам, представляющим изображения цифр. Учитывая, что подпись к каждому такому графическому объекту начинается с ключевого слова "Figure" (вполне резонное предположение), соответствующее стилевое правило принимает следующий вид.

```
img[alt^="Figure"] {border: 2px solid gray; display: block;
                    margin: 2em auto;}
```

Будьте готовы к тому, что в результате применения такого форматирования в документе будут выделены все графические объекты, атрибут `atl` которых начинается со значения "Figure", а не только изображения, представляющие фигуры цифр.

В следующей задаче необходимо выделить в календаре все события, происходящие по понедельникам (Monday). При ее решении стоит предположить, что подписи к таким событиям, содержащиеся в атрибуте `alt`, представлены в следующем формате: "Monday, March 5th, 2012". Для указания стилевого правила для таких элементов можно использовать селектор `[title^="Monday"]`.

Представление элемента по завершающей подстроке значения

Логическим продолжением предыдущего способа стилевого форматирования является использование селекторов атрибутов, в которых указывается подстрока, последняя в значении атрибута. Чаще всего стилевые правила, включающие такие селекторы, применяются для назначения разного форматирования гиперссылкам, указывающим на ресурсы разных типов. Например, с их помощью в документе можно специальным образом выделить гиперссылку, указывающую на PDF-файл (рис. 2.14).

```
a[href$=".pdf"] {font-weight: bold;}
```

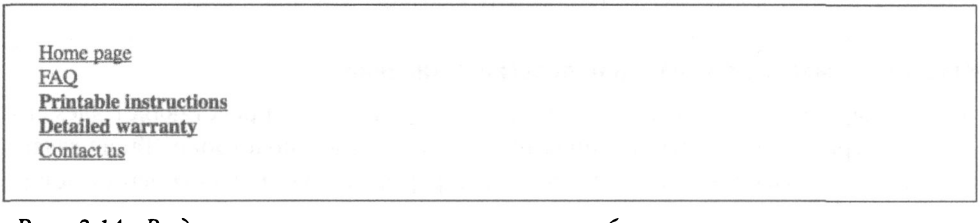


Рис. 2.14. Выделение элементов, значение атрибута которых оканчивается определенной подстрокой

Подобным образом можно установить отдельное форматирование для графических файлов всех представленных в документе типов.

```
img[src$=".gif"] {...}
img[src$=".jpg"] {...}
img[src$=".png"] {...}
```

В примере с календарем, рассмотренном в конце предыдущего раздела, такой поход позволит выделить все события, происходившие за указанный в селекторе год (например, `[title $="2015"]`).

Вы могли заметить, что в тексте главы все подстроки, приводимые в селекторе атрибута, заключены в кавычки. Использование кавычек в селекторе атрибута обязательно, если передаваемое значение содержит специальные символы, начинается с дефиса или цифры и не представлено ключевыми словами, определенными в спецификации. Чтобы обезопасить код от ошибок, старайтесь всегда заключать значения атрибутов в кавычки, кроме случаев передачи им действительных идентификаторов.

Идентификатор нечувствительности к регистру

В модуль CSS Selectors четвертой версии добавлена функция определения чувствительности к регистру значений, указываемых в селекторах атрибутов. Добавив идентификатор `i` перед закрывающей квадратной скобкой, можно сделать значение атрибута нечувствительным к регистру символов, независимо от установок, определяемых спецификацией языка разметки документа.

Например, пусть в документе необходимо применить специальное форматирование к гиперссылкам на PDF-документы, но при этом неизвестно, заканчиваются ли они подстрокой `.pdf`, `.PDF` или `.Pdf`. Идентификатор нечувствительности к регистру позволяет изящно решить эту проблему.

```
a[href$='.PDF' i]
```

Включение символа `i` в селектор указывает ему представлять все элементы `a`, значение атрибута `href` которых заканчивается расширением `.pdf`, независимо от регистра символов, которым оно представлено.

Идентификатор нечувствительности к регистру можно использовать во всех селекторах атрибутов, рассмотренных выше. Обратите внимание на то, что он применяется к *значениям*, а не к названиям атрибутов, упоминаемым в селекторах. Таким образом, стилевое правило `planet[type*="rock" i]` будет успешно применяться к таким элементам документа.

```
<planet type="barren rocky">Mercury</planet>
<planet type="cloudy ROCKY">Venus</planet>
<planet type="life-bearing Rock">Earth</planet>
```

Наряду с этим стилевое форматирование не будет назначено следующему элементу, поскольку название атрибута `TYPE` не подпадает под действие идентификатора `i`.

```
<planet TYPE="dusty rock">Mars</planet>
```

Стоит заметить, что идентификатор `i` востребован только в документах, созданных с помощью языков разметки, чувствительных к регистру (например, XHTML). Во всех остальных спецификациях, в частности HTML5, он не находит применения.

К концу 2017 года идентификатор нечувствительности к регистру не поддерживался Opera Mini, Edge и браузерами, запускаемыми из Android.

Структура документа

Широкие функциональные возможности CSS по стилевому форматированию в первую очередь связаны с высокой степенью его взаимодействия со структурой документа. Именно структура документа предопределяет способы его визуального представления пользовательскими агентами. С ней стоит познакомиться поближе, прежде чем продолжить изучение селекторов, обеспечивающих более совершенные способы представления элементов в стилевых правилах.

Родительские и дочерние элементы

Чтобы понять, как взаимосвязаны элементы документа, необходимо изучить его структуру. Сделаем это на примере следующего простого HTML-документа.

```
<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>
```

Действие наиболее мощных инструментов CSS основано на взаимосвязи родительских и дочерних элементов. В основе всех без исключения документов HTML (как и большинства структурированных документов любого вида) лежит иерархическая

структура элементов, представленная своеобразным деревом (рис. 2.15), в котором указаны все включенные в документ элементы. Каждый элемент документа занимает в структуре строго заданное место и выступает либо *родителем*, либо *потомком*, либо и тем и другим сразу по отношению к другим элементам.

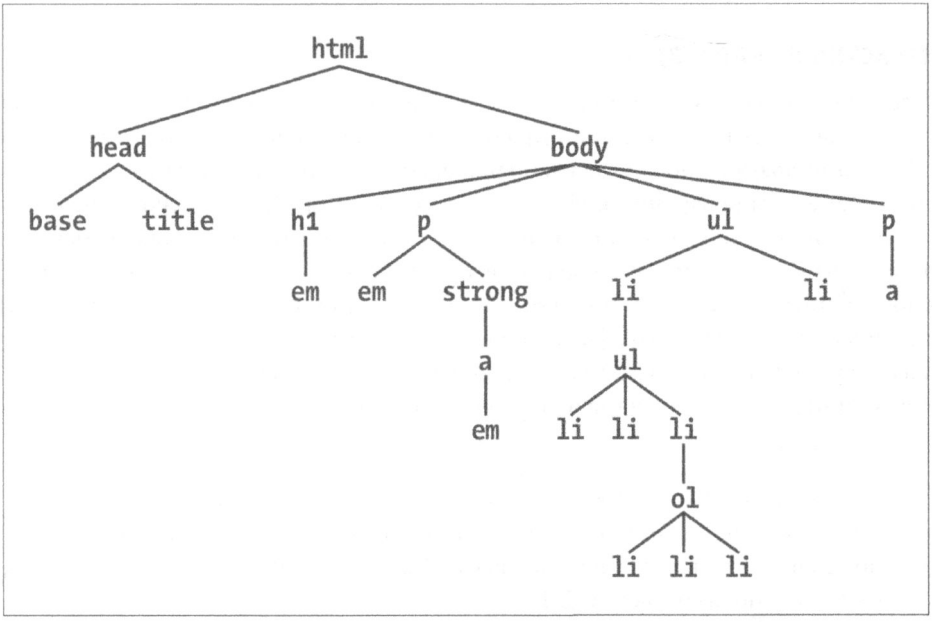


Рис. 2.15. Древовидная структура элементов документа

Элемент является родительским для другого элемента, если в иерархической структуре он находится непосредственно над ним. На рис. 2.15 видно, что первый элемент `p` выступает родителем для элементов `em` и `strong`. При этом элемент `strong` является родителем для элемента ссылки (`a`), а тот, в свою очередь, становится родителем для еще одного элемента — `em`. И наоборот, элемент, расположенный в структуре документа под родительским, считается дочерним по отношению к нему. Таким образом, элемент ссылки `a` является дочерним для элемента `strong`, который сам выступает в качестве дочернего по отношению к элементу `p`, а тот — дочерним по отношению к элементу `body`.

Термины "родительский" и "дочерний" получены как частная форма понятий *предок* и *потомок*. Тем не менее между ними существует определенное различие: о родительско-дочерних взаимоотношениях говорят, когда два элемента находятся на соседних уровнях иерархической структуры документа. Если же элементы отстоят друг относительно друга на большее количество уровней, то их можно называть только предком и потомком (в действительности дочерний элемент — это также потомок, а родительский элемент — предок). Согласно рис. 2.15, первый элемент `ul` является родителем двух элементов `li`, но при этом он также предок для всех элементов, дочерних по отношению к элементам `li`, и всех их потомков — до самого нижнего уровня иерархической структуры.

На рис. 2.15 также видно, что элемент a выступает в качестве дочернего по отношению к элементу strong, но при этом является потомком для элементов p, body и html. Элемент body включает все элементы документа, по умолчанию отображаемые в браузере, а элемент html становится предком для всего документа. Именно поэтому в HTML и XHTML элемент html часто называется корневым.

Контекстные селекторы

Преимущества объектной модели HTML в полной мере проявляются при форматировании документов стилевыми правилами, включающими контекстные селекторы. С их помощью можно создавать правила, которые применяются только к определенным элементам иерархической структуры документа. Предположим, специальное форматирование необходимо применить только к элементам em, являющимся потомками элемента h1. Для решения задачи все элементы em, включенные в элементы h1, можно снабдить общим атрибутом class, но это занимает много времени, а потому контрпродуктивно. Намного эффективнее использовать правила, селекторы которых напрямую указывают на элементы em, являющиеся потомками элемента h1.

Ниже приведен один из вариантов такого правила.

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

Оно назначает серый цвет тексту всех элементов em, вложенных в элементы h1. Остальные элементы em документа (например, включенные в абзацы или текстовые блоки) под действие правила не попадают. Результат применения правила к фрагменту документа показан на рис. 2.16.

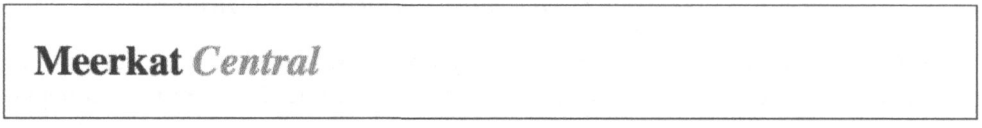


Рис. 2.16. Форматирование элемента в зависимости от контекста его использования

В левой части правила указывается несколько (не менее двух) селекторов, разделенных пробелами. В данном случае пробел используется в качестве *комбинатора*, указывающего на принадлежность одного элемента другому. Наследственные связи между элементами в контекстном селекторе определяются справа налево. Таким образом, селектор h1 em указывает на элементы em, являющиеся потомками элементов h1.

В контекстном селекторе допускается указывать больше двух элементов иерархической структуры документа, как показано в следующем примере:

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

Результат его применения к документу показан на рис. 2.17. Правило назначает серый цвет выделенному курсивом тексту, содержащемуся в элементах неупорядоченного списка, вложенного в упорядоченный список, дочерний по отношению к еще одному неупорядоченному списку (верхнего уровня). Однако нужно сказать, что в реальных документах столь сложные контекстные селекторы применяются очень редко.

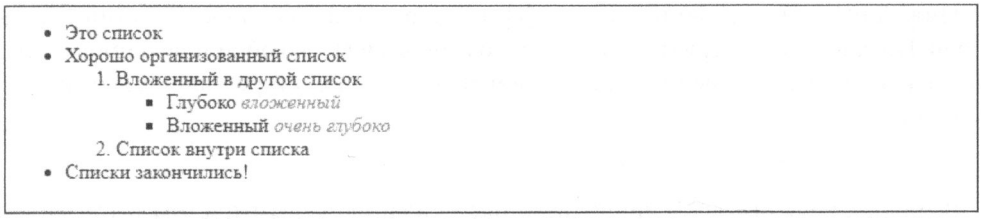


Рис. 2.17. Применение стилевого форматирования с помощью контекстного селектора необычного вида

Контекстные селекторы обладают широкими функциональными возможностями. Они позволяют применять форматирование, недоступное для инструментов HTML, — по крайней мере, без использования большого количества тегов `font`. Для наглядности рассмотрим следующий пример. Пусть требуется создать документ с боковой панелью в левой части. Основная часть документа расположена на белом фоне, а боковая панель — на синем, и обе они содержат гиперссылки. Если тексту гиперссылок назначить стандартный синий цвет, то они будут неразличимы на боковой панели.

Решение состоит в использовании контекстных селекторов. Достаточно назначить элементам, включенным на боковую панель, класс `sidebar`, а основную область заключить в элемент `main`, после чего назначить документу следующее стилевое форматирование.

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

В окне браузера такой документ будет выглядеть так, как показано на рис. 2.18.

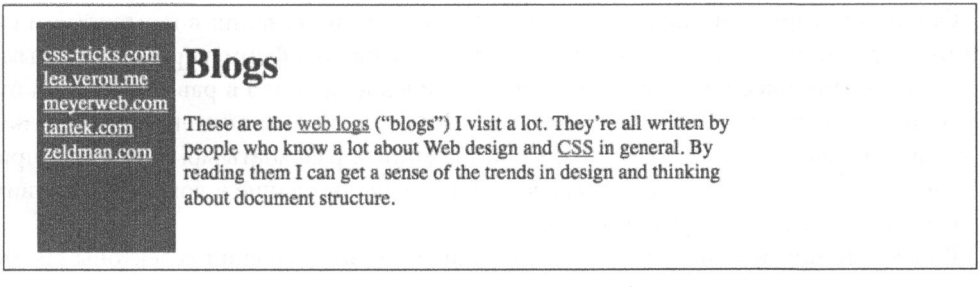


Рис. 2.18. Назначение разных стилей элементам одного типа с помощью контекстных стилей (см. цветные иллюстрации на веб-сайте)

Ключевое слово `:link` определяет непосещенные гиперссылки. Детально его назначение описано в разделе "Псевдоклассы гиперссылок".

Ниже приведен еще один пример эффективного применения контекстных селекторов. Предположим, серым цветом необходимо выделить любой полужирный текст (элементы b), включенный в цитаты (элемент blockquote) и регулярные абзацы документа.

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

Выполнение этого кода приводит к окрашиванию серым цветом всех элементов b — потомков элементов абзацев и цитат.

Обратите внимание на то, что контекстные селекторы применяются к потомкам независимо от степени их родства с родительским элементом. В частности, селектор ul em указывает на все элементы em, имеющие любую глубину вложения в элементы ul. Таким образом, стилевые правила, включающие селектор ul em, можно применять к следующему HTML-документу.

```
<ul>
   <li>List item 1
      <ol>
         <li>List item 1-1</li>
         <li>List item 1-2</li>
         <li>List item 1-3
            <ol>
               <li>List item 1-3-1</li>
               <li>List item <em>1-3-2</em></li>
               <li>List item 1-3-3</li>
            </ol>
         </li>
         <li>List item 1-4</li>
      </ol>
   </li>
</ul>
```

Обратная сторона медали состоит в невозможности указания в контекстном селекторе уровня вложения элемента, которому назначается форматирование. Иными словами, содержащее контекстный селектор стилевое правило в равной степени будет применяться к вложенному элементу и всем остальным элементам объектной модели, вложенным в него. Об этом важно помнить, разрабатывая стратегию применения правил CSS (см. далее) и предотвращая использование в документе правил, отменяющих действия других правил.

В качестве примера рассмотрим следующий код, включающий селекторы специального типа, описание которых будет приведено позже.

```
div:not(.help) span {color: gray;}
div.help span {color: red;}

<div class="help">
   <div class="aside">
      Текст, содержащий <span>элемент span</span>.
   </div>
</div>
```

Первое правило приведенного выше CSS-кода задает серый цвет для всех элементов span, вложенных в элемент div, отличных от тех, в значении атрибута class которых присутствует слово help. Второе правило устанавливает красный цвет для таких же элементов, но содержащих слово help в значении атрибута class. Если проанализировать разметку документа, то легко понять, что к элементу span применяются сразу *оба* правила.

Поскольку оба правила имеют одинаковый приоритет, но второе правило указывается последним, то элементы span получат красный цвет. Несмотря на то что в объектной модели документа элемент span находится ближе к элементу div class="aside", чем к элементу div class="help", при определении стилевого форматирования это совершенно не учитывается. Как вам уже известно, порядок вложения элементов в документе в контекстном селекторе не играет роли. При выполнении обоих условий к элементу применяется только один из вариантов форматирования — в данном случае красный цвет элемента span определяется скорее общим порядком применения правил CSS (рассматривается далее), чем условиями, устанавливаемыми селекторами.

Селекторы дочерних элементов

В отдельных ситуациях форматирование должно применяться не ко всем потомкам указанного элемента, а только к их первому колену — дочерним элементам. Например, следующее правило изменяет цвет только у элементов strong, являющихся непосредственными потомками (первый уровень вложения) элементов h1. Оно не будет назначаться всем потомкам элемента h1, как в случае контекстных селекторов. Дочерние элементы указываются в селекторе с помощью комбинатора >.

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

При применении правила к следующему фрагменту документа красный цвет назначается первому элементу, но не второму.

```
<h1>This is <strong>very</strong> important.</h1>
<h1>This is <em>really <strong>very</strong></em> important.</h1>
```

Селекторы, включающие комбинатор >, "читаются" справа налево. В данном примере селектор представляет элементы strong, выступающие непосредственными потомками элементов h1. Комбинатор > допускается выделять пробелами любыми известными способами, например h1 > strong, h1> strong или h1>strong. В самом простом случае пробелы вообще не используются.

Для получения представления о родительских элементах, которым назначается форматирование, необходимо изучить иерархическую структуру документа. Взаимоотношения элементов фрагмента документа, код которого представлен выше, можно выразить схемой, показанной на рис. 2.19.

Даже такой небольшой фрагмент схемы позволяет судить о родительско-дочерних отношениях элементов документа. В частности, элемент a выступает родителем для элемента strong, но является дочерним по отношению к элементу p. Таким образом, в селекторах правил, применяемых к указанному фрагменту документа, можно

использовать записи p > a и a > strong, но не запись p > strong, поскольку элемент strong, хотя и является потомком элемента p, не относится к его дочерним элементам.

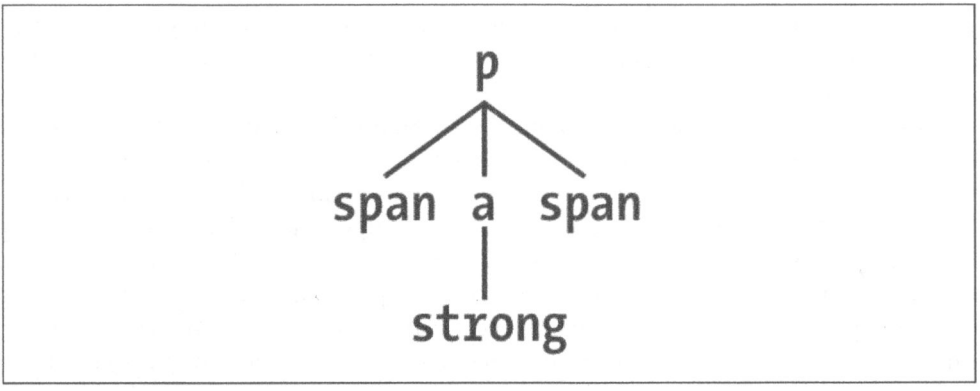


Рис. 2.19. Фрагмент иерархической структуры документа

В правиле допускается совмещать контекстный селектор и комбинатор дочерних элементов. Например, селектор table.summary td > p представляет все элементы p, выступающие потомками элемента td, в свою очередь являющихся потомками элемента table, значение атрибута class которого содержит слово summary.

Селектор соседних элементов

Рассмотрим ситуацию, когда стилевое форматирование применяется к абзацу, расположенному сразу же после заголовка, или списку, следующему после этого абзаца. Чтобы указать в правиле элемент, расположенный сразу после другого элемента, необходимо применить *комбинатор соседних элементов*, представленный символом + ("плюс"). Как и комбинатор дочерних элементов, знак "плюс" разрешается разграничивать с обеих сторон произвольным количеством пробелов.

Например, следующий код позволяет удалить один из отступов у элементов абзаца, расположенных после заголовков первого уровня:

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

Селектор правила указывает на все элементы p, расположенные сразу же после элементов h1 и имеющих с ними *общего родителя*.

Для иллюстрации назначения этого селектора обратимся к фрагменту документа, схема иерархической структуры которого показана на рис. 2.20.

На рис. 2.20 проиллюстрированы взаимоотношения элемента div с потомками — неупорядоченным и упорядоченным списками, каждый из которых состоит из трех элементов. Эти списки являются соседними по отношению друг к другу, чего не скажешь об элементах, принадлежащих к разным спискам, поскольку они относятся к разным родителям (они являются двоюродными родственниками, но в CSS селектор такого типа не предусмотрен).

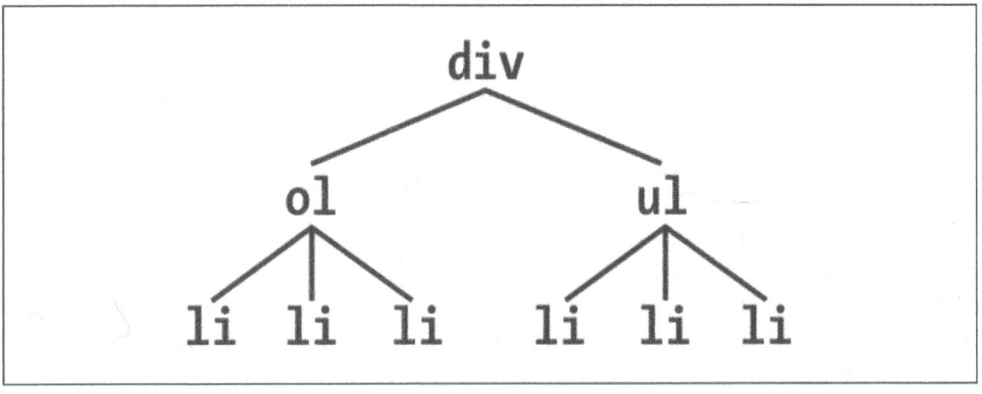


Рис. 2.20. Фрагмент иерархической структуры документа

С помощью селектора соседних элементов можно применять форматирование только ко вторым и последующим элементам обоих списков (но не к первым). В частности, следующее правило назначает полужирное начертание вторым и третьим элементам списков, как показано на рис. 2.21:

```
li + li {font-weight: bold;}
```

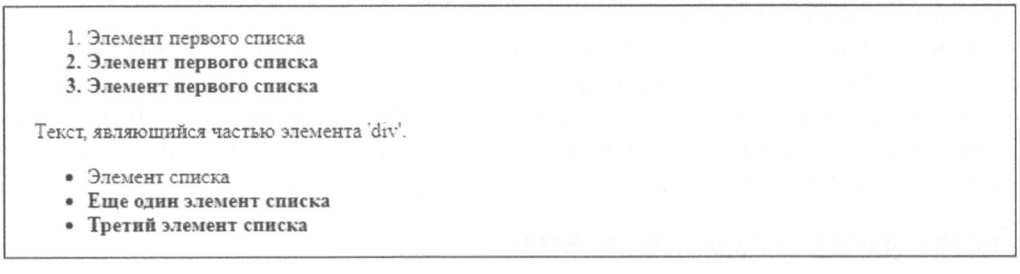


Рис. 2.21. Форматирование соседних элементов

В селекторе соседних элементов элементы указываются исключительно в порядке следования их в иерархической структуре документа. В данном примере элемент `ul` располагается сразу после элемента `ol`, что позволяет указать его с помощью селектора `ol + ul`. К сожалению, подобный синтаксис нельзя использовать для выбора элемента `ol` — селектор `ul + ol` справедлив только для ситуаций, когда упорядоченный список располагается сразу же после неупорядоченного.

Обратите внимание на то, что добавление текста между соседними элементами не влияет на работу селектора. В качестве примера рассмотрим фрагмент HTML-документа, схема иерархической структуры которого изображена на рис. 2.20.

```
<div>
   <ol>
      <li>Элемент первого списка</li>
      <li>Элемент первого списка</li>
      <li>Элемент первого списка</li>
   </ol>
```

```
   Текст, являющийся частью элемента 'div'.
   <ul>
      <li>Элемент списка</li>
      <li>Еще один элемент списка</li>
      <li>Третий элемент списка</li>
   </ul>
</div>
```

Несмотря на наличие текста между упорядоченным и неупорядоченным списками, последний по-прежнему можно представить селектором `ol + ul`. В данном случае текст включается в элемент `div`, а не выступает отдельным элементом одного уровня с элементами `ol` и `ul`. Если заключить его в элемент абзаца, то порядок следования соседних элементов будет нарушен, и селектор `ol + ul` не будет работать должным образом. Вместо него для указания неупорядоченного списка необходимо будет использовать селектор `ol + p + ul`.

В селекторе соседних элементов можно использовать комбинаторы других типов, как показано в следующем примере:

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

Данное правило устанавливает верхнее поле для элементов `ul`, расположенных сразу после элементов `table`, являющихся потомками элемента `body`, родителем которого выступает элемент `html`.

Как и любые другие, комбинатор соседних элементов допускается включать в более сложные конструкции, например `div#content h1 + div ol`. Данный селектор указывает на все элементы `ol` с родительским элементом `div`, соседним по отношению к элементу `h1`, который является дочерним для элементов `div` с атрибутом `id`, установленным в значение `content`.

Селектор всех соседних элементов

В модуль CSS Selectors 3 добавлен *общий комбинатор соседних элементов*, позволяющий применять правило стилевого форматирования сразу ко всем элементам, расположенным после указанного в селекторе и имеющим с ним общего родителя. В селекторах правил он представляется символом "тильда" (~).

В качестве примера создадим правило, назначающее курсивное начертание всем элементам `ol`, расположенным сразу после элемента `h2`, имеющего общего с ними родителя.

```
h2 ~ol {font-style: italic;}
```

Под воздействие правила попадают не только элементы `ol`, соседние с элементами `h2`, но и любые другие элементы общего родителя, находящиеся на одном уровне иерархической структуры. Результат применения правила к следующему фрагменту документа HTML приведен на рис. 2.22.

```
<div>
   <h2>Подзаголовки</h2>
   <p>Далеко не все заголовки верхнего уровня. Некоторые из них
      являются подзаголовками для других заголовков.</p>
```

```
    <ol>
        <li>Наиболее важные заголовки</li>
        <li>Заголовки, второстепенные по отношению к заголовкам
            второго уровня</li>
        <li>Заголовки наименьшей важности среди используемых
            в документе</li>
    </ol>
    <p>Повторение — мать учения!</p>
    <ol>
        <li>Наиболее важные заголовки</li>
        <li>Заголовки, второстепенные по отношению к заголовкам
            верхнего уровня</li>
        <li>Заголовки наименьшей важности среди используемых
            в документе</li>
    </ol>
</div>
```

Легко заметить, что курсивное начертание применяется к обоим упорядоченным спискам, поскольку они следуют за элементом `h2` и имеют общего с ним родителя (элемент `div`).

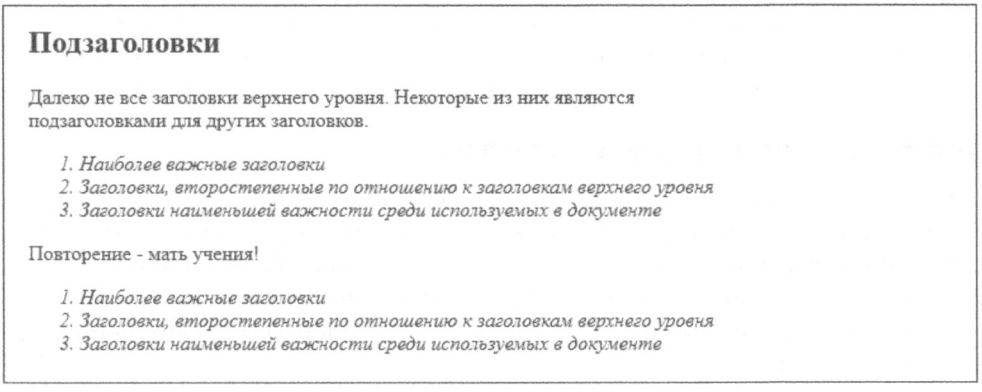


Рис. 2.22. Выделение всех соседних элементов

Селекторы псевдоклассов

Возможности стилевого оформления документов существенно расширились после добавления в спецификацию CSS поддержки *псевдоклассов*. Селекторы псевдоклассов позволяют назначать форматирование элементам, относящимся к фиктивным классам, динамически изменяющим свое состояние или расположение в иерархической структуре документа.

Название "фиктивный класс" не отражает сути псевдокласса. Понять назначение псевдокласса лучше всего на реальном примере. Предположим, в документе необходимо выделить специальным стилем все четные строки таблицы. Чтобы решить эту задачу, можно снабдить элементы таких строк атрибутом `class` со значением `"even"` и назначить им правило, включающее селектор соответствующего класса. Не менее эффективно эту же задачу можно решать, воспользовавшись селектором псевдокласса.

Комбинирование псевдоклассов

Перед тем как начать изучение псевдоклассов, необходимо ознакомиться с понятием комбинирования (составления "цепочки") псевдоклассов в блоке селекторов. В приведенном ниже примере все непосещенные ссылки, на которые наведен указатель мыши, выделяются красным (`red`) цветом, а такие же ссылки после наведения указателя — темно-бордовым (`maroon`) цветом.

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

В данном случае порядок следования названий псевдоклассов в селекторе не играет особой роли. С таким же успехом в правиле можно было использовать селекторы `a:hover:link` и `a:hover:visited`. Псевдоклассы также позволяют назначать отдельные стили гиперссылкам (посещенным и непосещенным), введенным на других языках.

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

При использовании псевдоклассов проявляйте осторожность и избегайте назначения взаимоисключающих селекторов. Например, гиперссылка не может одновременно находиться в посещенном и непосещенном состояниях, поэтому селектор `a:link:visited` недействителен и не указывает ни на один из элементов документа.

Структурная организация псевдоклассов

Большинство псевдоклассов привязано к объектной модели документа, а потому подлежит строгой структурной организации. Многие из них связаны с отдельными элементами иерархической структуры документа, например с каждым третьим абзацем, или же указывают на элементы специального типа. Все без исключения селекторы псевдоклассов начинаются с двоеточия (`:`), и их допускается комбинировать с селекторами других типов.

У псевдоклассов есть особенность, отличающая их от остальных атрибутов: они привязаны к строго заданным элементам и ни к каким другим. Весьма однозначное утверждение, не правда ли? Несмотря на это многим кажется, что некоторые селекторы псевдоклассов указывают не на сами такие элементы, а на их потомков.

Следующая история из жизни служит наглядным тому примером. В 2003 году родилась моя первая дочь, и я (как поступили бы многие из вас) сразу же известил об этом друзей и коллег по работе, сделав объявление по Интернету. Отовсюду посыпались поздравления и, как это часто бывает, профессиональные шутки по сложившемуся поводу. Больше других отличился мой непосредственный начальник, обозначивший мою дочь селектором `#ericmeyer:firstchild`. Несуразность заключалась в том, что этот селектор указывает на меня как на единственного ребенка своих родителей (что, несомненно, так и есть), а не на мою дочь. Селектор, указывающий на дочь, выглядит так: `#ericmeyer > :first-child`.

Приведенный выше случай как нельзя лучше иллюстрирует серьезность проблемы. Чтобы избежать неоднозначности, в дальнейшем материале область применения

псевдоклассов уточняется там, где это необходимо. Для правильного использования селекторов псевдоклассов достаточно помнить, что они представляют элементы "фиктивных классов", к которым привязаны.

Селектор корневого элемента

Псевдокласс `:root` связан с элементом, который включает все остальные элементы документа, — корневым элементом. В HTML в качестве корневого *всегда* выступает элемент `html`. Его преимущества в полной мере раскрываются при стилизации документов, написанных на XML, в которых каждому из расширений сопоставляется свой корневой элемент. Например, в RSS 2.0 корневой элемент называется `rss`. В некоторых языках поддерживается сразу несколько корневых элементов, но в каждом отдельно взятом документе он всегда один.

Ниже приведен пример правила стилевого форматирования корневого элемента документа HTML. Результат его применения показан на рис. 2.23.

```
:root {border: 10px dotted gray;}
body {border: 10px solid black;}
```

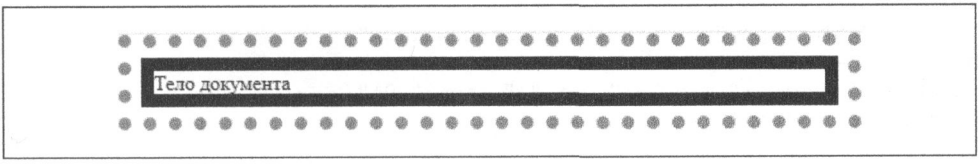


Рис. 2.23. Визуальное оформление корневого элемента

Учитывая, что в стилевых правилах документ HTML может представляться селектором `html`, большой необходимости в псевдоклассе `:root` нет. Тем не менее каждый из упомянутых селекторов имеет свой приоритет применения в документе, о чем рассказано в главе 3.

Селектор пустых элементов

Псевдокласс `:empty` применяется для представления элементов, которые не содержат дочерних элементов, в том числе текста и пробелов. С его помощью можно скрыть в документе все пустые элементы, не имеющие визуального содержимого. В частности, правило `p:empty {display: none;}` позволяет скрыть в документе все пустые абзацы.

Чтобы попасть под воздействие правила, элемент не должен включать совершенно ничего — даже пробелов. Из всех приведенных ниже элементов только первый и последний могут считаться полностью пустыми.

```
<p></p>
<p> </p>
<p>
</p>
<p><!--комментарий--></p>
```

С точки зрения синтаксического анализатора пользовательского агента второй и третий абзацы имеют содержимое — в первом случае пробел, а во втором — символ конца строки. Таким образом, они не относятся к псевдоклассу `:empty`. Последний абзац рассматривается как пустой, поскольку комментарий, в отличие от пробелов и специальных символов, не относится к визуализируемому содержимому документа. Если в начало или в конец строки комментария добавить хотя бы один пробел, то абзац будет исключен из псевдокласса `:empty`.

Самое простое, что приходит в голову, — это скрыть пустые элементы, воспользовавшись следующим общим правилом:

```
*:empty {display: none;}
```

Тем не менее такой подход приводит к заведомо неправильному результату, поскольку в HTML к пустым относятся лишенные содержимого элементы `img` и `input`, а иногда и `textarea` (без текста, заданного по умолчанию). Таким образом, с точки зрения наполненности данными селекторы `img` и `img:empty` представляют идентичные элементы (при этом имеющие разную приоритетность, о чем будет рассказано в следующей главе).

```
<img src="salmon.png" alt="North Pacific Salmon">
<br>
<input type="number" min="-1" max="1" step=".01"/>
<textarea></textarea>
```

Селектор `:empty` уникальный в своем роде — это единственный селектор CSS, представляющий элементы, исходя из наполненности их содержимым. Во всех остальных селекторах, включенных в спецификацию CSS Selectors 3 (например, в селекторе соседних элементов), в расчет принимаются только элементы, но никак не их содержимое.

Селектор единственных дочерних элементов

Псевдокласс `:only-child` поможет решить многие рутинные задачи, подобные выделению в документе всех изображений, заключенных в элементы гиперссылок. В общем случае он представляет единственные дочерние элементы других элементов. Следующий код позволяет добавить границу к изображениям, каждое из которых является единственным у своего родительского элемента:

```
img:only-child {border: 1px solid black;}
```

Форматирование применяется ко всем изображениям, соответствующим указанному критерию. Таким образом, с помощью данного правила изменяется форматирование каждого изображения, включенного в отдельные абзацы, которые не содержат других элементов, независимо от того, наполнены ли они текстом. Для выделения в документе изображений, представляющих гиперссылки, необходимо использовать такое правило (результат его применения к приведенному ниже фрагменту документа HTML показан на рис. 2.24).

```
a[href] img:only-child {border: 2px solid black;}

<a href="http://w3.org/"><img src="w3.png" alt="W3C"></a>
<a href="http://w3.org/"><img src="w3.png" alt=""> The W3C</a>
<a href="http://w3.org/"><img src="w3.png" alt=""> <em>The W3C</em></a>
```

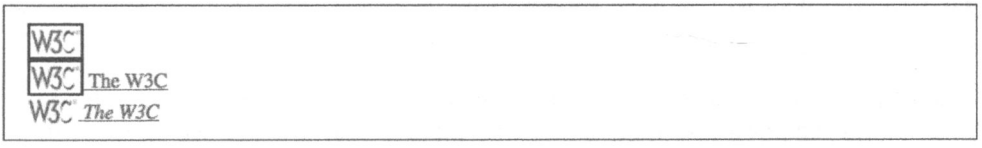


Рис. 2.24. Выделение единственных дочерних изображений гиперссылок

Задавая форматирование с помощью правил, селекторы которых включают псевдокласс `:only-child`, необходимо помнить о том, что он, как уже указывалось, представляет единственные дочерние элементы, а не родительские, содержащие единственные дочерние элементы. Из этого утверждения следует, что при включении псевдокласса `:only-child` в контекстный селектор он будет представлять единственные дочерние элементы своих непосредственных родителей, независимо от степени их родства с указанными элементами.

Вернувшись к предыдущему примеру, можно смело утверждать, что селектор `a[href] img:only-child` представляет все изображения, являющиеся единственными, но не обязательно самыми близкими потомками элемента `a`. Чтобы соответствовать критерию правила, элементы изображения должны быть единственными потомками своих родителей и выступать потомками элементов гиперссылок. При этом родительские элементы таких изображений могут сами включаться в гиперссылки.

Результат применения следующего правила к несколько измененному фрагменту HTML-кода показан на рис. 2.25.

```
a[href] img:only-child {border: 5px solid black;}

<a href="http://w3.org/"><img src="w3.png" alt="W3C"></a>
<a href="http://w3.org/"><span><img src="w3.png" alt="W3C"></span></a>
<a href="http://w3.org/">A link to <span>the <img src="w3.png"
   alt="">web</span> site</a>
```

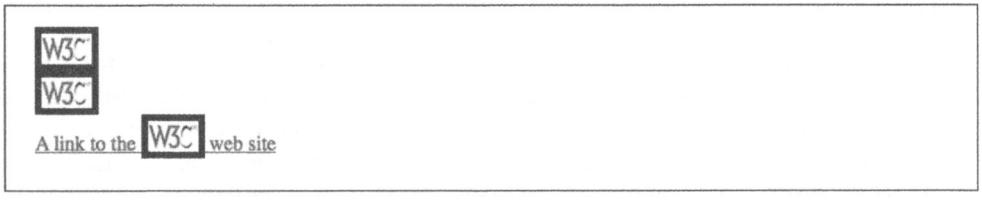


Рис. 2.25. Выделение единственных дочерних изображений гиперссылок

В данном случае изображения являются единственными дочерними элементами своих родителей, выступающих потомками элементов `a`. Таким образом, форматирование применяется ко всем трем изображениям. Чтобы ограничить область применения правила только изображениями, являющимися непосредственными дочерними элементами элемента `a`, его селектор нужно переписать следующим образом:

```
a[href] > img:onlychild
```

Указанному условию соответствует только первое изображение из показанных на рис. 2.25.

А как быть, если форматирование нужно применить к единственному изображению, которое добавлено в гиперссылку, включающую элементы других типов?

```
<a href="http://w3.org/"><b>•</b><img src="w3.png" alt="W3C"></a>
```

Данная гиперссылка содержит два элемента: `b` и `img`. Изображение в ней не является единственным дочерним элементом (элемента `a`), поэтому не относится к псевдоклассу `:only-child`. Но для его представления в селекторе правила можно использовать псевдокласс `:only-of-type`, как показано в следующем примере кода и на рис. 2.26.

```
a[href] img:only-of-type {border: 5px solid black;}

<a href="http://w3.org/"><b>•</b><img src="w3.png" alt="W3C"></a>
<a href="http://w3.org/"><span><b>•</b><img src="w3.png"
   alt="W3C"></span></a>
```

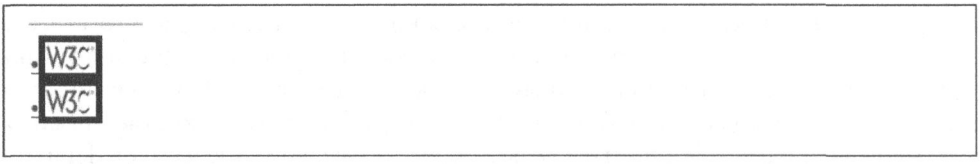


Рис. 2.26. Выделение изображений гиперссылок, являющихся единственными среди соседних элементов одного типа

Различие между псевдоклассами отражено в их названиях: псевдокласс `:only-of-type` представляет уникальный элемент своего типа из всех соседних, а псевдокласс `:only-child` указывает на элемент, полностью лишенный соседей.

Эта особенность псевдокласса `:only-of-type` позволяет применять его для выделения в документе изображений, включенных в абзацы, не беспокоясь о включении в них других элементов строчного типа, например гиперссылок.

```
p > img:only-of-type {float: right; margin: 20px;}
```

До тех пор пока в абзацы не будет добавлено несколько изображений, все уже находящиеся в них элементы `img` (единственные дочерние) будут выравниваться по правому краю и обтекаться текстом — по левому. В следующем примере показано, как правильно использовать пседокласс `only-of-type` для применения специального форматирования к элементам `h2`, которые являются единственными заголовками второго уровня в своих разделах.

```
section > h2 {margin: 1em 0 0.33em; font-size: 1.8rem;
              border-bottom: 1px solid gray;}
section > h2:only-of-type {font-size: 2.4rem;}
```

После применения этих правил к документу HTML все элементы `section`, включающие только один элемент `h2`, будут представляться шрифтом увеличенного размера. Как только в раздел будет добавлено больше двух заголовков второго уровня, его шрифт будет уменьшен до базового размера. Добавление в элемент `section` элементов другого типа (подзаголовков, таблиц, абзацев и списков) никак не скажется на изменении параметров форматирования элементов `h2`.

Важно понимать, что псевдокласс `only-of-type` представляет элементы только указанного типа. Рассмотрим такой пример.

```
p.unique:only-of-type {color: red;}

<div>
   <p class="unique">Этот абзац относится к классу 'unique'.</p>
   <p>Этот абзац не относится ни к одному из классов.</p>
</div>
```

Указанное правило не применяется ни к одному из абзацев, поскольку у элемента `div` сразу два дочерних элемента `p` одного типа, а псевдокласс `only-of-type` представляет только единственный родительский элемент типа `p`.

В данном случае отнесение одного из элементов к другому классу никак не сказывается на области применения селектора, включающего псевдокласс `only-of-type`. Нужно помнить, что тип элемента и его класс — это не одно и то же. В частности, селектор `p.unique:only-of-type` представляет все элементы `p`, атрибут `class` которых содержит слово `unique`, являющиеся единственными дочерними элементами своего типа в элементе `div`. Ошибочно считать, что имеются в виду единственные элементы `p`, отнесенные к классу со словом `unique`.

Селектор первых и последних дочерних элементов

Весьма распространенным способом стилевого оформления элементов документа остается выделение их первых и последних дочерних элементов. Чаще всего такой способ форматирования применяется для визуального выделения первой и/или последней гиперссылок на панели навигации, включающей большое количество элементов. Раньше для решения этой задачи первая и последняя гиперссылки относились к разным классам, каждому из которых назначались свои стилевые правила. В последней спецификации CSS первый и последний дочерние элементы представляются специальными псевлоклассами.

Первые дочерние элементы других элементов представляются псевдоклассом `:first-child`. В качестве примера рассмотрим следующий фрагмент документа HTML.

```
<div>
   <p>Выполните следующие действия:</p>
   <ul>
      <li>вставьте ключ;</li>
      <li>поверните ключ <strong>по часовой стрелке</strong>;</li>
      <li>нажмите педаль газа.</li>
   </ul>
```

```
   <p>
      Ни в коем случае<em> не </em> нажимайте педали газа
      и тормоза одновременно.
   </p>
</div>
```

Легко заметить, что элементы `p`, `strong`, `em` и первый элемент `li` являются первыми дочерними элементами своих родительских элементов. После применения следующих правил к приведенному выше фрагменту документа HTML он принимает вид, показанный на рис. 2.27.

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

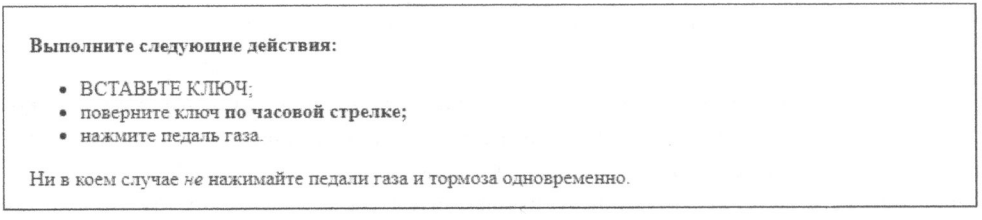


Рис. 2.27. Специальное форматирование первых дочерних элементов

Первое правило устанавливает полужирное начертание для первого абзаца своего родительского элемента (в данном случае `div`), а второе представляет содержимое первого элемента списка (согласно спецификации HTML вкладываемого в элемент `ol` или `ul`) прописными буквами.

Как известно, самая распространенная ошибка при использовании псевдоклассов заключается в неправильном определении целевых элементов, подлежащих форматированию. Селектор `p:first-child` представляет не первый дочерний элемент, вложенный в элемент `p`, а первый элемент `p`, вложенный в свой родительский элемент. Не стоит забывать о том, что псевдокласс всегда представляет элемент, к которому он привязан, а не элементы, состоящие с ним в родстве. Если заменить псевдокласс, являющийся фиктивным с точки зрения HTML, реальным классом `first-child`, то приведенный выше фрагмент документа принимает следующий вид.

```
<div>
   <p class="first-child">Выполните следующие действия:</p>
   <ul>
      <li class="first-child">вставьте ключ;</li>
      <li>поверните ключ<strong class="first-child">
          по часовой стрелке</strong>;</li>
      <li>нажмите педаль газа.</li>
   </ul>
   <p>
      Ни в коем случае <em class="first-child">не </em> нажимайте
      педали газа и тормоза одновременно.
   </p>
</div>
```

Теперь первые дочерние элементы `em` (своих родительских элементов) можно представить селектором `em:first-child`.

Псевдокласс `:last-child` представляет элементы, противоположные указываемым псевдоклассом `:first-child`. После применения стилевых правил, селекторы которых обращаются к псевдоклассу `:first-child`, приведенный ниже фрагмент документа HTML приобретает вид, показанный на рис. 2.28.

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

<div>
<p>Выполните следующие действия:</p>
<ul>
<li>вставьте ключ;</li>
<li>поверните ключ <strong>по часовой стрелке</strong>;</li>
<li>нажмите педаль газа.</li>
</ul>
<p>
Ни в коем случае <em>не </em> нажимайте педали газа и тормоза
одновременно.
</p>
</div>
```

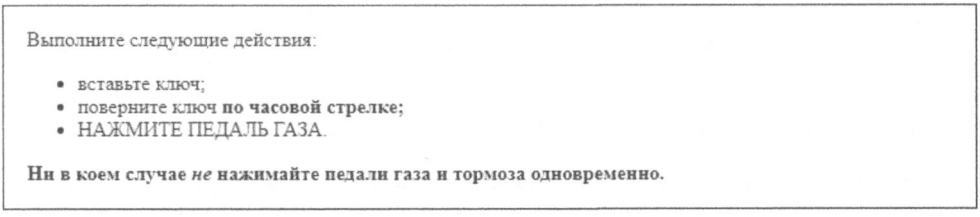


Рис. 2.28. Стилевое форматирование последних дочерних элементов

Первое правило устанавливает полужирное начертание для последнего абзаца своего родительского элемента (в данном случае `div`), а второе представляет содержимое последнего элемента списка строчными буквами. Для применения специального форматирования к элементам `em` последнего абзаца в стилевом правиле нужно использовать селектор `p:last-child em`, указывающий на все элементы `em`, которые включены в элементы `p`, являющиеся последними дочерними элементами своих родительских элементов.

Удивительно, но комбинация псевдоклассов `:first-child` и `:last-child` представляет такие же элементы, что и псевдокласс `:onlychild`. Следующие два правила применяются к одинаковым элементам.

```
p:only-child {color: red;}
p:first-child:last-child {background-color: red;}
```

В каждом из случаев текст абзацев окрашивается красным цветом и выводится на красном фоне (далеко не самый удачный дизайнерский ход).

Селекторы первых и последних элементов одного типа

Подобно псевдоклассам, описанным в предыдущем разделе и представляющим первый и последний дочерние элементы, в CSS добавлены псевдоклассы, указывающие на первый и последний вложенные элементы своего типа. Их удобно применять, например, для назначения стилевого форматирования первой таблице (элемент `table`) одного из элементов документа, независимо от общего количества элементов других типов, включенных в него.

```
table:first-of-type {border-top: 2px solid gray;}
```

Как и в предыдущих случаях, правило относится не ко всему документу, а только к элементам, содержащим таблицы. В частности, задаваемое им форматирование применяется не к первой таблице документа, а к первым элементам `table`, родительские элементы которых содержат не менее одного такого элемента. Таким образом, в документе, иерархическая структура которого показана на рис. 2.29, данное правило применяется только к элементам, обведенным кружками.

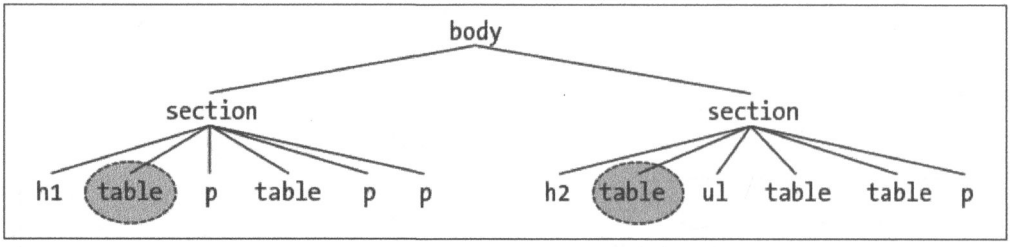


Рис. 2.29. Таблицы, являющиеся первыми дочерними элементами своего типа

Если родительский элемент представлен таблицей, то для форматирования первых ячеек с данными каждой из строк, независимо от включения в таблицу столбца с заголовками, можно использовать такое правило:

```
td:first-of-type {border-left: 1px solid red;}
```

Правило применимо к первым ячейкам с данными каждой из строк следующей таблицы.

```
<tr>
   <th scope="row">Count</th><td>7</td><td>6</td><td>11</td>
</tr>
<tr>
   <td>Q</td><td>X</td><td>-</td>
</tr>
```

Селектор `td:first-child` не позволяет добиться такого же эффекта, поскольку представляет первый элемент `td` второй, а не первой строки, как можно было предположить.

Псевдокласс `:last-of-type` является антиподом псевдокласса `:first-of-type`, представляя последний из соседних элементов такого же типа. Различие между ними состоит только в направлении поиска дочернего элемента: в последнем случае

соседние элементы просматриваются с конца последовательности к ее началу — до нахождения первого экземпляра элемента указанного типа. На рис. 2.30 показана иерархическая структура документа, в которой кружками выделены элементы, представленные селектором `table:last-of-type`.

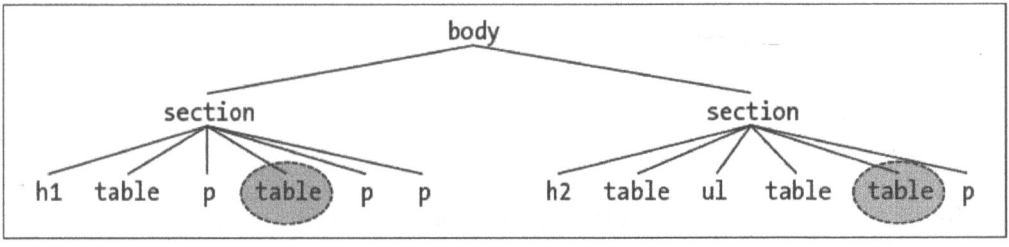


Рис. 2.30. Таблицы, являющиеся последними дочерними элементами своего типа

Как известно из раздела, посвященного псевдоклассу `:only-of-type`, целевые элементы выбираются из последовательности соседних элементов общего родительского элемента. Иными словами, каждый набор соседних элементов рассматривается в контексте их родительского элемента, а не всего документа. Первые (или последние) элементы одного типа выбираются исключительно из наборов дочерних элементов каждого родительского элемента документа.

Как и в случае, описанном в предыдущем разделе, комбинацией псевдоклассов `:first-of-type` и `:last-of-type` можно заменить псевдокласс `:only-of-type`. Следующие правила обращаются к одним и тем же элементам.

```
table:only-of-type{color: red;}
table:first-of-type:last-of-type {background: red;}
```

Селектор нумерованного элемента

Вы узнали, как строятся селекторы единственного, первого и последнего дочерних элементов своих родительских элементов. Но как быть, если форматирование нужно применить, например, к каждому третьему или девятому дочернему элементу? Какие селекторы применяются в правилах форматирования только четных и нечетных дочерних элементов? Использовать бесконечное количество именованных псевдоклассов для представления всех возможных соседних элементов в высшей степени нерационально. В спецификации CSS их роль играет всего один псевдокласс — `:nth-child()`. Номер представляемого этим псевдоклассом элемента указывается в скобках и задается целым числом или даже математическим выражением.

В самом простом случае — при передаче псевдоклассу `:nth-child()` числа `1` — он представляет такой же элемент, как и псевдокласс `:first-child`. В приведенном ниже примере стилевое форматирование применяется к первому абзацу и первому элементу списка.

```
p:nth-child(1) {font-weight: bold;}
li:nth-child(1) {text-transform: uppercase;}
```

```
<div>
   <p>Выполните следующие действия:</p>
   <ul>
      <li>вставьте ключ;</li>
      <li>поверните ключ<strong>по часовой стрелке</strong>;</li>
      <li>нажмите педаль газа.</li>
   </ul>
   <p>
      Ни в коем случае <em>не </em> нажимайте педали газа
      и тормоза одновременно.
   </p>
</div>
```

При замене числа `1`, указанного в скобках псевдокласса `:nth-child()`, числом `2` стилевое форматирование применяется к среднему (второму) элементу списка и не задается для абзацев, как показано на рис. 2.31.

```
p:nth-child(2) {font-weight: bold;}
li:nth-child(2) {text-transform: uppercase;}
```

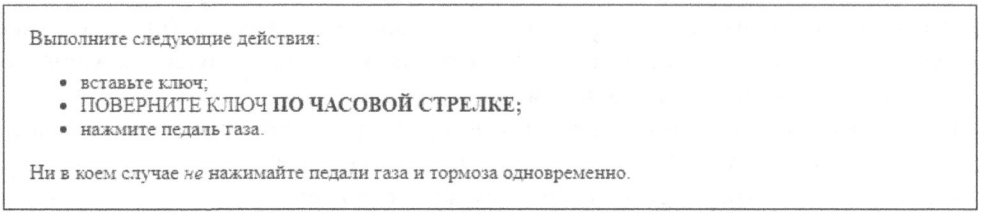


Рис. 2.31. Форматирование второго дочернего элемента

В скобки псевдокласса `:nth-child()` допускается подставлять любое целое число. В частности, селектор `ol:nth-child(93)` представляет 93-й (дочерний) элемент упорядоченного списка. Учтите, что он также указывает на 93-е элементы всех упорядоченных списков, включенных в документ. (Будьте внимательны и не перепутайте этот селектор с селектором, представляющим 93-й упорядоченный список документа и описанным в следующем разделе.)

Псевдокласс `nth-child()` может представлять целые последовательности элементов, для чего в скобки вместо целых чисел подставляется математическое выражение вида $an + b$ или $an - b$, где n — это целочисленный счетчик, a и b — параметры последовательности, представленные целыми числами. Если b равно нулю, то оно не указывается, и выражение записывается в форме an.

Предположим, форматирование нужно применить к каждому третьему элементу неупорядоченного списка, начиная с первого. Эта задача решается с помощью следующего правила, применение которого к реальному документу приводит к выделению строчными символами первого и четвертого элементов маркированного списка (рис. 2.32).

```
ul > li:nth-child(3n + 1) {text-transform: uppercase;}
```

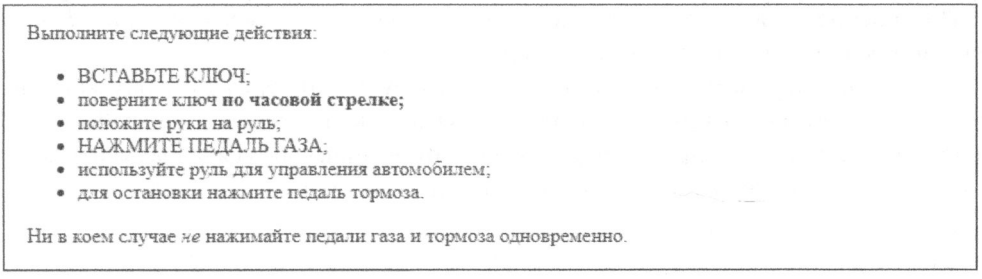


Рис. 2.32. Выделение каждого третьего элемента списка

Для определения последовательности элементов (в данном случае — 1, 4, 7, 10, 13 и т.д.), которым назначается стилевое правило, браузер поочередно подставляет целые числа 0, 1, 2, 3, 4 и т.д. до бесконечности в выражение $3n + 1$ вместо n. Если из выражения исключить второй параметр (+1), то специальное форматирование будет применяться к элементам с порядковыми номерами 0, 3, 6, 9, 12 и т.д.

Последняя последовательность интересна тем, что начинается с третьего элемента, поскольку учет элементов ведется не с нуля, а с единицы. Хотя это и выглядит несколько непривычно, стилевое форматирование в данном случае действительно будет применяться с третьего элемента.

Подбирая параметры последовательности так, чтобы она всегда начинались с первого элемента, легко определить, что селектор `:nth-child(2n)` представляет четные элементы, а нечетные элементы задаются селектором `:nth-child(2n+1)` или `:nth-child(2n-1)`. Кроме того, для указания четных и нечетных элементов служат ключевые слова `even` и `odd`, передаваемые псевдоклассу `:nth-child(2n+1)` вместо указанных математических выражений. Чаще всего селекторы четных и нечетных элементов применяются для выделения цветом каждой второй строки таблицы, как показано на рис. 2.33.

```
tr:nth-child(odd) {background: silver;}
```

Missouri	MO	Jefferson City	Eastern Bluebird
Montana	MT	Helena	Western Meadowlark
Nebraska	NE	Lincoln	Western Meadowlark
Nevada	NV	Carson City	Mountain Bluebird
New Hampshire	NH	Concord	Purple Finch
New Jersey	NJ	Trenton	Eastern Goldfinch
New Mexico	NM	Santa Fe	Roadrunner
New York	NY	Albany	Eastern Bluebird
North Carolina	NC	Raleigh	Northern Cardinal
North Dakota	ND	Bismarck	Western Meadowlark
Ohio	OH	Columbus	Northern Cardinal
Oklahoma	OK	Oklahoma City	Scissor-Tailed Flycatcher
Oregon	OR	Salem	Western Meadowlark
Pennsylvania	PA	Harrisburg	Ruffed Grouse

Рис. 2.33. Стилевое форматирование каждой второй строки таблицы

Построение правильной последовательности с помощью выражения *an* + *b* — далеко не такая простая задача, как кажется на первый взгляд.

Подставляя отрицательные значения вместо параметра *b*, следует опускать знак "плюс", стоящий перед ним. В противном случае селектор вообще не сработает. Из приведенных ниже двух правил только первое будет применяться в документе, а второе будет проигнорировано синтаксическим анализатором.

```
tr:nth-child(4n - 2) {background: silver;}
tr:nth-child(3n + -2) {background: red;}
```

Для применения форматирования ко всем элементам последовательности, например, начиная с девятого, можно использовать любое из следующих двух правил. Они оба приводят к одинаковому результату, но второе обладает более высокой приоритетностью, о чем будет рассказано в следующей главе.

```
tr:nth-child(n + 9) {background: silver;}
tr:nth-child(8) ~ tr {background: silver;}
```

Как и следовало ожидать, в CSS включен псевдокласс `:nth-last-child()`, по действию обратный `:nth-child()`. От последнего он отличается только направлением вычисления элемента, к которому применяется форматирование: в `nth-last-child()` подсчет ведется от последнего элемента к первому. В частности, такой способ стилевого форматирования позволяет добавлять цветной фон каждой второй строке таблицы так, чтобы в их число обязательно попадала последняя строка. Для решения этой задачи подходит любое из следующих двух правил.

```
tr:nth-last-child(odd) {background: silver;}
tr:nth-last-child(2n+1) {background: silver;} /* эквивалентный
                                                 вариант */
```

При добавлении и удалении строк из таблицы объектная модель документа претерпевает изменения, что может вызвать нарушения в работе селекторов элементов, последовательность которых представляется математическим выражением. В подобных случаях структурные селекторы, последовательность элементов в которых определяется ключевым словом `odd` или `even`, обладают неоспоримым преимуществом, сохраняя работоспособность даже при изменении иерархической структуры документа.

При правильном задании критериев отбора элементов в селектор можно включать оба псевдокласса: `:nth-child()` и `:nth-last-child()`. Рассмотрим следующие правила, результат применения которых к таблице показан на рис. 2.34.

```
li:nth-child(3n + 3) {border-left: 5px solid black;}
li:nth-last-child(4n - 2) {border-right: 5px solid black;
                           background: silver;}
```

Упомянутые правила можно заменить одним, снабдив его комбинированным селектором `:nth-child(1):nth-last-child(1)`, представляющим собой расширенную версию псевдокласса `:only-child`. Создание правил со столь запутанными селекторами оправдывается только более высоким приоритетом их применения, что не имеет особого значения в данном примере.

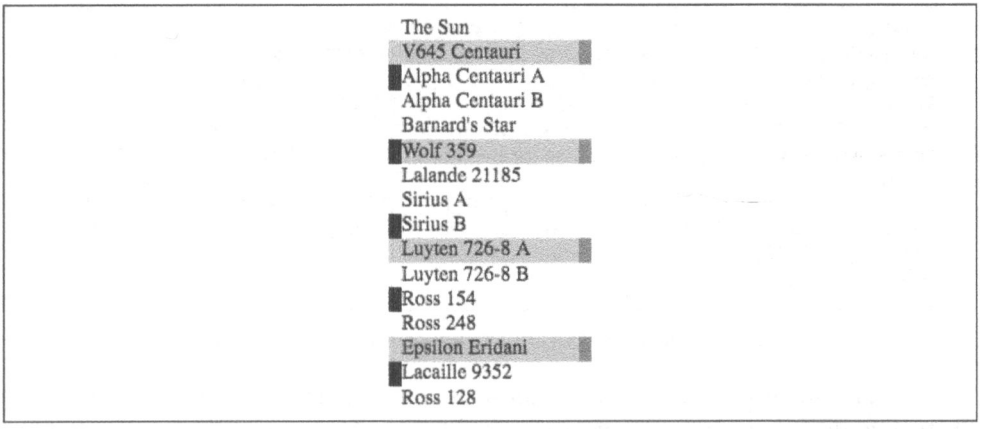


Рис. 2.34. Совместное применение псевдоклассов `:nth-child()` и `:nth-last-child()`

С помощью селекторов CSS можно определять количество элементов в последовательности и их форматирование, исходя из полученного значения. Вот как это реализуется для элементов списка.

```
li:only-child {width: 100%;}
li:nth-child(1):nth-last-child(2),
li:nth-child(2):nth-last-child(1) {width: 50%;}
li:nth-child(1):nth-last-child(3),
li:nth-child(1):nth-last-child(3) ~ li {width: 33.33%;}
li:nth-child(1):nth-last-child(4),
li:nth-child(1):nth-last-child(4) ~ li {width: 25%;}
```

Принцип действия приведенных выше правил очень прост. Если список состоит всего из одного элемента, то его ширина составляет 100%. Если первый элемент списка одновременно является вторым с конца элементом последовательности, то это значит, что список состоит всего из двух элементов. Ширина элементов определяется значением 50%. Если первый элемент списка также выступает третьим с конца элементом последовательности, то список состоит из трех элементов. Элементы сужаются до 33%. Подобным образом ширина элементов уменьшается до 25%, когда первый и четвертый с конца элементы последовательности представляют один и тот же элемент списка.

Селектор нумерованных элементов одного типа

Записываемые согласно общепринятому шаблону, псевдоклассы `:nth-of-type()` и `:nth-last-oftype()` являются логическим продолжением псевдоклассов, рассмотренных в предыдущей главе. С их помощью, например, удобно применять стилевое форматирование к каждой второй гиперссылке, дочерней по отношению к определенному абзацу. Соответствующий селектор имеет вид `p > a:nth-of-type(even)`. Правилом игнорируются дочерние элементы абзаца остальных типов (`span`, `strong` и т.п.) — оно определяет внешний вид только четных гиперссылок (рис. 2.35).

```
p > a:nth-of-type(even) {background: blue; color: white;}
```

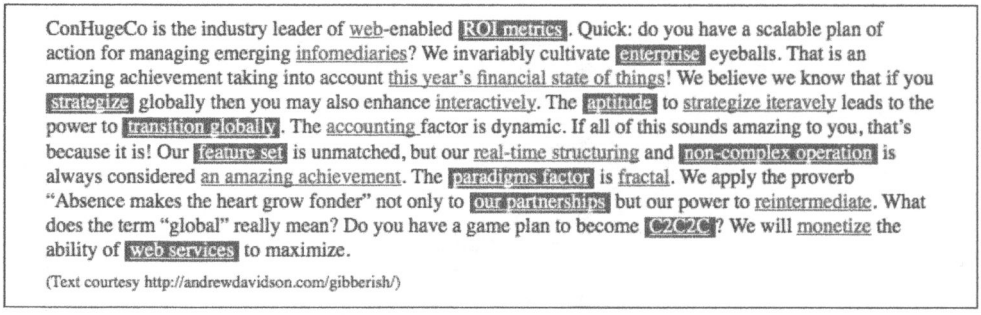


Рис. 2.35. Выделение четных гиперссылок в тексте абзаца

Для учета гиперссылок, начиная с последнего дочернего элемента абзаца, в селекторе нужно использовать псевдокласс `a:nth-last-of-type(even)`.

```
p > a:nth-last-of-type(even)
```

Как и другие псевдоклассы такого же типа, `:nth-of-type()` и `:nth-last-oftype()` представляют все дочерние элементы одного типа, включенные в свои родительские элементы, но не однотипные элементы всего документа. Каждый родительский элемент обладает своим списком дочерних элементов, и однотипные элементы выбираются исключительно из него.

Вполне ожидаемо в селектор можно включать сразу оба псевдокласса. В частности, селектор `:nth-of-type(1):nthlast-of-type(1)` функционально равнозначен селектору `:only-of-type` и обладает более высокой приоритетностью применения в документе (о приоритетности правил речь пойдет в главе 3).

Динамические псевдоклассы

В CSS наряду с рассмотренными выше структурными псевдоклассами включены псевдоклассы, представляющие элементы, состояние которых изменяется после визуализации страницы в браузере. Иными словами, они задают стилевое форматирование элементов, исходя из свойств, не представленных в объектной модели и изменяющих свое значение после разметки документа.

При первом знакомстве может сложиться впечатление, будто динамические псевдоклассы применяются в стилевых правилах хаотически, что на самом деле не так. Они представляют элементы, находящиеся в определенных кратковременных состояниях, переход к которым невозможно предсказать заранее. Тем не менее обстоятельства возникновения таких событий в CSS определяются весьма точно, что позволяет применять их в качестве критериев в селекторах стилевых правил. Чтобы лучше понять, о чем идет речь, проведем аналогию с событиями, возникающими в реальной жизни. Наблюдая за ходом футбольного матча, легко заметить, что болельщики бурно реагируют только на голевые ситуации у ворот команды гостей. Разумеется, заранее предсказать их не представляется возможным. Тем не менее реакция болельщиков на каждое из таких событий вполне прогнозируема. Таким образом, реакция болельщиков на событие неизменна и не зависит от момента его возникновения.

Проще всего описать назначение динамических псевдоклассов на примере элемента a, представляющего гиперссылки в HTML (и некоторых других языках разметки документов). После отображения в пользовательском агенте гиперссылки остаются неизменными, но некоторые из них указывают на уже посещенные страницы, а остальные — на еще не посещенные. В разметке документа HTML оба типа гиперссылок выглядят одинаково. Чтобы определить гиперссылки, указывающие на уже посещенные ресурсы, необходимо сравнить их с гиперссылками, занесенными в журнал браузера. Средствами CSS обрабатываются два основных типа гиперссылок: посещенные и непосещенные.

Псевдоклассы гиперссылок

Псевдоклассы, представляющие гиперссылки в CSS, впервые были добавлены в спецификацию 2.1. В HTML гиперссылки представлены элементом a, имеющим всего один атрибут: href. В XML в качестве гиперссылки, указывающей на другие ресурсы, может выступать любой элемент разметки документа. Описание псевдоклассов гиперссылок, которые можно использовать в правилах CSS, приведено в табл. 2.2.

Таблица 2.2. Псевдоклассы, представляющие гиперссылки

Название	Описание
:link	Представляет элемент гиперссылки, адрес которой отсутствует в журнале веб-страниц, посещенных браузером
:visited	Представляет элемент гиперссылки, уже открывавшейся ранее в браузере. Исходя из требований безопасности, к посещенным гиперссылкам разрешается применять только ограниченное количество стилей (см. врезку "Посещенные гиперссылки и безопасность")

Первый описанный в табл. 2.2 псевдокласс может показаться бесполезным: все гиперссылки, не относящиеся к посещенным, заведомо непосещенные. Такая логика предполагает описание состояний гиперссылки всего одним динамическим псевдоклассом.

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

Указанный способ стилевого оформления гиперссылок, находящихся в разных состояниях, кажется безупречным до тех пор, пока документ не содержит заполнители гиперссылок, лишенных адреса целевой веб-страницы.

```
<a>4. The Lives of Meerkats</a>
```

Текст таких гиперссылок окрашивается в синий цвет, попадая под действие первого правила — a {color: blue;}. Для цветового выделения в тексте документа и посещенных, и непосещенных ссылок понадобится помощь обоих псевдоклассов: :link и :visited.

```
a:link {color: blue;} /* непосещенные ссылки синие */
a:visited {color: red;} /* посещенные ссылки красные */
```

Включив псевдоклассы в селекторы атрибутов и классов, можно существенно расширить область их применения. Предположим, условия задачи требуют выделения цветом только гиперссылок, указывающих на внешние ресурсы. В общем случае критерием отбора элементов является атрибут, содержащий адрес ресурса. Если же система управления контентом браузера обрабатывает только URL-адреса, представленные в абсолютном формате, то все гиперссылки, указывающие на внешние ресурсы, нужно отнести к отдельному классу.

```
<a href="/about.html">My About page</a>
<a href="https://www.site.net/" class="external">An external site</a>
```

Стилевые правила, соответствующие условиям задачи и определяющие форматирование приведенных выше элементов гиперссылок, записываются следующим образом.

```
a.external:link, a[href^="http"]:link { color: slateblue;}
a.external:visited, a[href^="http"]:visited {color: maroon;}
```

В результате применения этих правил вторая гиперссылка окрашивается в серовато-синий цвет (`slateblue`) в обычном состоянии, становясь темно-бордовой (`maroon`) в посещенном состоянии, а вторая гиперссылка принимает цвет, задаваемый браузером по умолчанию (синий — для непосещенных ссылок и фиолетовый — для посещенных).

Применение к посещенным гиперссылкам других цветовых решений порядком запутывает пользователей, не позволяя в полной мере отслеживать уже посещенные ресурсы. Чаще всего с этой проблемой сталкиваются люди, испытывающие затруднения с быстрым восприятием новой информации, преимущественно при посещении сайтов, наполненных большим количеством гиперссылок. Придерживаясь цветового решения, заданного по умолчанию, вы не только будете следовать официальным рекомендациям комитета W3C, но и сделаете навигацию по сайту намного более комфортной.

Правила изменения цвета гиперссылок можно создавать на основе селекторов идентификаторов, как показано ниже.

```
a#footer-copyright:link{background: yellow;}
a#footer-copyright:visited {background: gray;}
```

В селекторах стилевых правил допускается комбинировать псевдоклассы `:link` и `:visited`, но в этом мало смысла, так как гиперссылка не может находиться сразу в обоих состояниях.

Псевдоклассы, представляющие действия пользователей

В CSS включено несколько псевдоклассов, позволяющих устанавливать форматирование документа в ответ на действия пользователей. Традиционно динамические псевдоклассы применяются для изменения внешнего вида гиперссылок, хотя обладают намного бо́льшими стилевыми возможностями. Их описание приведено в табл. 2.3.

Посещенные гиперссылки и безопасность

Еще каких-то десять лет назад действующая на тот момент спецификация CSS не вносила ограничений на изменение внешнего вида посещенных гиперссылок. Их форматирование можно было определять с помощью любых свойств, включенных в стандарт CSS. Но в середине 2000-х годов сразу несколькими разработчиками было показано, что с помощью простого сценария, анализирующего структуру объектной модели документа и стилевое оформление гиперссылок, можно легко определить, посещал ли пользователь тот или иной ресурс. Собранная таким образом информация передавалась на специальные серверы и обычно использовалась злоумышленниками в самых неблаговидных целях. Более того, историю посещения пользователем ресурсов можно было отследить, даже не прибегая к написанию сценариев, — по одним лишь фоновым изображениям.

Если большинству из нас подобная возможность не представляется настолько серьезной, чтобы обращать на нее внимание, то в тоталитарных странах она может использоваться для отслеживания деятельности пользователей в Интернете. Не стоит забывать, что в таких государствах посетителей "запрещенных" сайтов — оппозиционных движений, запрещенных религиозных организаций или ресурсов, содержащих аморальный с точки зрения официальной доктрины контент, — ожидает суровое наказание в виде вполне реального тюремного срока. Чтобы предотвратить сбор данных о выполняемых пользователями действиях, в том числе и при проведении онлайн-платежей, в спецификацию CSS были внесены определенные ограничения.

Во-первых, отныне внешний вид посещенных гиперссылок может определяться всего несколькими "цветовыми" свойствами: `color`, `background-color`, `column-rule-color`, `outline-color`, `border-color` (включая свойства, устанавливающие цвета отдельных границ, подобные `border-top-color`). Попытки форматирования посещенных гиперссылок другими свойствами будут попросту проигнорированы браузером. Более того, любые стилевые правила, устанавливаемые для псевдокласса `:link`, по умолчанию применяются как к посещенным, так и непосещенным гиперссылкам. Во-вторых, браузер всегда возвращает значение цвета для гиперссылки, заданное по умолчанию, независимо от того, совершался по ней переход к другому ресурсу или нет. Таким образом, на внешний запрос браузер всегда возвращает цвет непосещенных ссылок, даже если в документе непосещенные ссылки представляются синим цветом, а посещенные — фиолетовым.

К концу 2017 года описанные выше ограничения соблюдались большинством популярных браузеров во всех рабочих режимах (не только приватном). Для повышения удобочитаемости документа посещенные гиперссылки нужно выделять таким образом, чтобы они заметно отличались от непосещенных, даже несмотря на ограниченные возможности CSS по стилевому форматированию этих элементов.

Таблица 2.3. Псевдоклассы, представляющие действия пользователей

Название	Описание
`:focus`	Соответствует элементу, получающему фокус. После передачи элементу фокуса в него можно вводить данные или активизировать любым другим способом
`:hover`	Представляет элемент, на который наведен указатель мыши, — обычно гиперссылку
`:active`	Соответствует элементу, активизированному пользователем, например гиперссылке, на которой щелкнули мышью

К псевдоклассу `:active` относятся элементы гиперссылок, кнопок, пунктов меню и любые другие элементы управления, снабженные атрибутом `tabindex`. Фокус могут получать все вышеперечисленные элементы, а также интерактивные элементы и элементы управления форм, поддерживающие редактирование данных.

Подобно `:link` и `:visited`, псевдоклассы, представляющие действия пользователей, чаще всего применяются для изменения внешнего вида гиперссылок. Следующие стилевые правила применяются при форматировании большинства веб-страниц, с которыми вам доводилось сталкиваться.

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

Порядок указания псевдоклассов в правилах CSS играет далеко не самую последнюю роль. Лучше всего придерживаться такого порядка: `link-visited-hover-active`. При необходимости его можно представить такой последовательностью: `link-visited-focus-hover-active`. Подробно о важности соблюдения порядка применения стилевых правил в документе будет рассказано в следующей главе. Из нее вы также узнаете, в каких случаях его можно и нужно нарушать.

Обратите внимание на то, что псевдоклассы, представляющие действия пользователей, соотносятся не только с гиперссылками, но и с любыми другими элементами, позволяющими динамически изменять свой внешний вид. Например, приведенное ниже стилевое правило изменяет фон текстового поля формы, в котором установлен курсор (рис. 2.36).

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

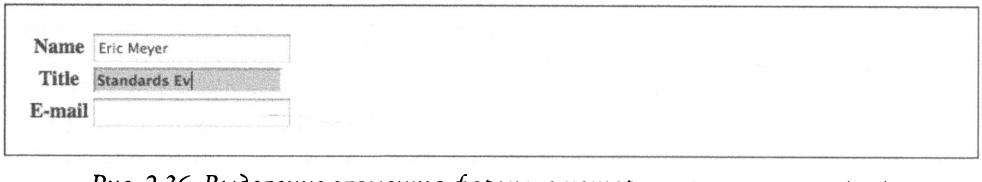


Рис. 2.36. Выделение элемента формы, в котором установлен курсор

Эксперимента ради можно попытаться применить динамический псевдокласс к произвольным элементам документа. Следующее правило добавляет эффект выделения сразу ко всем элементам документа:

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

В результате применения правила все элементы, вложенные в элемент `body`, будут отображаться на желтом фоне при наведении на них указателя мыши. Желтый фон получают все указываемые пользователем заголовки, абзацы, списки, таблицы, изображения и любые другие потомки элемента `body`. Можно не ограничиваться изменением одного только фона элемента — добавьте к нему границу, укажите иной шрифт или установите любое другое форматирование, поддерживаемое браузером.

Определяя стилевые правила для элементов, представленных псевдоклассом `:focus`, ни в коем случае *не отменяйте* все форматирование. Крайне важно сделать элементы, находящиеся в фокусе, хорошо различимыми, повысив тем самым удобство работы с документами, особенно просматриваемыми без использования мыши.

Практические задачи, решаемые с помощью динамических псевдоклассов

Динамические псевдоклассы находят применение в самых неожиданных задачах и позволяют добиться весьма интересных эффектов. Например, с их помощью можно представлять гиперссылки, на которые наведен указатель мыши, намного бо́льшим по размеру шрифтом, чем применяется для отображения посещенных и непосещенных гиперссылок (рис. 2.37).

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

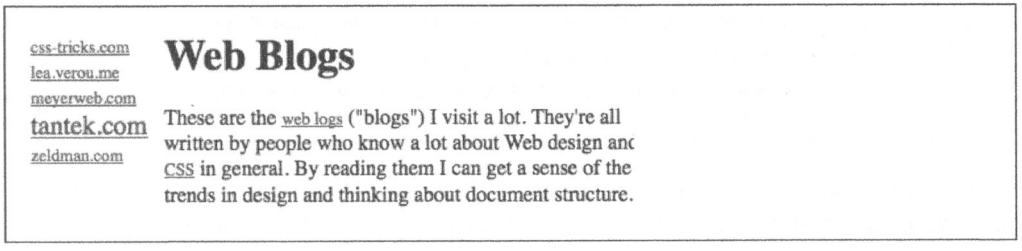


Рис. 2.37. Изменение макета документа с помощью динамических псевдоклассов

Пользовательский агент увеличивает размер элемента `a` при наведении на него указателя мыши или, благодаря поддержке псевдокласса `:active`, нажатии пальцем на сенсорном экране. Изменение размера гиперссылки при наведении на нее указателя мыши приводит к смещению и перераспределению остального содержимого документа, расположенного вокруг элемента `a`.

Псевдоклассы пользовательского интерфейса

По действию динамические псевдоклассы сильно напоминают псевдоклассы графического интерфейса, детальные сведения о которых приведены в табл. 2.4. Последние позволяют применять стилевое форматирование к элементам графического интерфейса, находящихся в определенных состояниях.

Таблица 2.4. Псевдоклассы пользовательского интерфейса.

Название	Описание
`:enabled`	Представляет доступные элементы управления (например, формы), готовые к вводу данных
`:disabled`	Представляет недоступные или заблокированные элементы управления (например, формы) — ввод данных в них невозможен
`:checked`	Соответствует переключателям или флажкам, находящимся в состоянии "включено", которое устанавливается пользователем или определяется по умолчанию
`:indeterminate`	Соответствует переключателям или флажкам, находящимся в неопределенном состоянии, устанавливаемом исключительно в сценариях, а не пользователем
`:default`	Представляет элементы форм (переключатели, флажки или опции меню), которые установлены по умолчанию в группе похожих элементов
`:valid`	Представляет поля форм, содержимое которых прошло проверку на целостность и достоверность данных
`:invalid`	Соответствует полям форм, содержимое которых не прошло проверку на целостность и достоверность данных
`:in-range`	Соответствует элементам форм, у которых введенное пользователем значение находится в заранее заданном диапазоне
`:out-ofrange`	Представляет поля форм, у которых введенное пользователем значение выходит из заданного диапазона
`:required`	Представляет поля, обязательные для заполнения перед отправкой формы
`:optional`	Соответствует полям, не обязательным для заполнения перед отправкой формы
`:read-write`	Соответствует полям форм, доступным для изменения
`:read-only`	Представляет поля, которые не подлежат изменению пользователем (не получают фокус)

Несмотря на то что состояние элементов управления, добавленных в документ, зависит от действий пользователя (установка флажка или переключателя, ввод данных в поле и т.п.), псевдоклассы пользовательского интерфейса нельзя классифицировать как полностью динамические, поскольку они также определяются структурой документа и внешними командами.

Функционально псевдокласс `:focus`, рассмотренный в предыдущем разделе, должен относиться к текущей категории, а не включаться в одну группу с `:hover` и `:active`. Тем не менее он отнесен к категории динамических псевдоклассов даже в спецификации CSS Selectors 3, скорее всего, унаследовав ее состав от CSS Selectors 2, в которой категория псевдоклассов

графического интерфейса попросту отсутствовала. Более важно то, что в CSS фокус может передаваться объектам документа, не относящимся к элементам управления, например заголовкам и абзацам (используется браузерами, снабженными функцией начитывания текста).

Доступные и недоступные элементы

Элементы управления, добавленные на веб-страницу, можно подключать и отключать (сделать доступными или недоступными для использования), применяя для этого инструменты HTML5 и команды сценариев. Недоступный элемент по-прежнему отображается в документе, но его нельзя выбирать и активировать — он также не отвечает на команды пользователей. Доступность элемента управления определяется во внешнем сценарии или (в HTML5) с помощью атрибута `disabled`.

Изначально все элементы управления доступны по умолчанию. Чтобы сделать элемент недоступным, необходимо изменить его исходного состояние. Для изменения внешнего вида доступных и недоступных элементов применяются псевдоклассы `:enabled` и `:disabled`. Как правило, специальным форматированием выделяются недоступные элементы, а доступные сохраняют исходный внешний вид. Пример изменения форматирования элементов обоих типов приведен на рис. 2.38.

```
:enabled {font-weight: bold;}
:disabled {opacity: 0.5;}
```

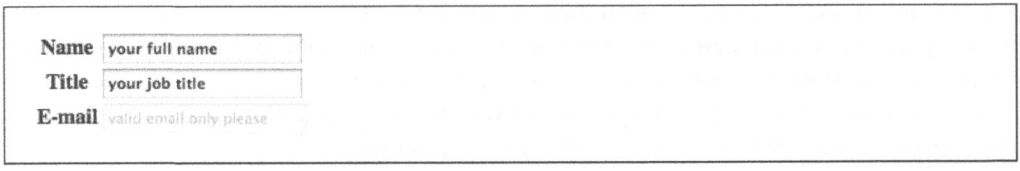


Рис. 2.38. Стилевое форматирование доступных и недоступных элементов

Состояние переключателя или опции

Элементы управления пользовательского интерфейса могут быть не только доступными или недоступными, но и включенными или выключенными. Указанные состояния назначаются элементам `input`, атрибут `type` которых установлен в значение `checkbox` или `radio`. Стилевое форматирование таких элементов осуществляется с помощью псевдокласса `:checked`, включенного в спецификацию CSS Selectors 3. Интересно то, что в ней отсутствует псевдокласс `:unchecked`, который мог бы представлять выключенные элементы управления. Наряду с этим данная спецификация располагает псевдоклассом `:indeterminate`, который представляет элементы управления, не находящиеся ни в одном из рабочих состоянии. Пример форматирования элементов управления, поддерживающих изменение рабочего состояния, приведен на рис. 2.39.

```
:checked {background: silver;}
:indeterminate {border: red;}
```

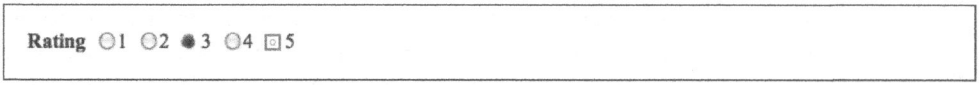


Рис. 2.39. Стилевое оформление элементов управления, находящихся во включенном и неопределенном состояниях (см. цветные иллюстрации на веб-сайте)

Для выделения выключенных элементов управления применяется рассмотренный далее псевдокласс отрицания (`:not`), добавляемый в селекторе перед псевдоклассом `:checked`, например так:

```
input[type="checkbox"]:not(:checked)
```

Только переключатели и флажки могут быть включенными или выключенными. Элементы управления остальных типов (а также эти два элемента управления в выключенном состоянии) всегда относятся к псевдоклассу `:not(:checked)`.

Несмотря на то что элементы управления по умолчанию выключены, их начальное поведение можно изменить, снабдив соответствующий элемент HTML атрибутом `checked`. Изменение исходного состояния элемента управления на противоположное обычно выполняется с помощью сценариев.

Как уже упоминалось, элементы управления могут находиться в неопределенном состоянии (`indeterminate`). К концу 2017 года элемент управления мог получить его только по команде сценария или самого браузера — в HTML инструменты перевода элементов в неопределенное состояние отсутствуют. Возможность стилевого форматирования элементов, представляемых псевдоклассом `:indeterminate`, чаще всего используется для выделения опций формы, требующих обязательного включения (или выключения). Как бы там ни было, псевдокласс `:indeterminate` применяется только для получения визуальных эффектов, но не указания рабочего состояния (включенный или выключенный) элемента управления.

В приведенном выше примере стилевое форматирование применяется непосредственно к переключателям, что в реальных документах выполняется не так уж и часто. Но это не означает, что псевдоклассы, представляющие элементы в разных рабочих состояниях, редко встречаются в селекторах стилевых правил. В частности, псевдокласс `:checked` можно использовать в селекторах соседних элементов для изменения внешнего вида подписей к флажкам и переключателям формы.

```
input[type="checkbox"]:checked + label {
   color: red;
   font-style: italic;
}

<input id="chbx" type="checkbox"> <label for="chbx">Подпись</label>
```

Селекторы элементов управления, выбранных по умолчанию

Псевдокласс `:default` представляет элементы управления, установленные по умолчанию в группе похожих элементов. Чаще всего он применяется для стилевого оформления опций контекстного меню, кнопок, переключателей или пунктов меню/списков. При добавлении на страницу сразу нескольких переключателей,

объединенных одним именем, только один из них относится к псевдоклассу :default, даже в случае обновления его состояния по команде пользователя. К псевдоклассу :default относятся только те элементы управления, которые включаются при загрузке документа. В частности, он будет представлять исходно выставленный флажок (элемент input типа checkbox) или выбранные по умолчанию элементы option списка (элемент select). Кроме того, к псевдоклассу :default относят предварительно выбранные кнопки и пункты меню.

```
[type="checkbox"]:default + label { font-style: italic; }

<input type="checkbox" id="chbx" checked name="foo" value="bar">
<label for="chbx">Выставляется при загрузке страницы</label>
```

Селектор обязательных элементов

К псевдоклассу :required относятся все элементы управления формы, обязательные к заполнению, на что указывает атрибут required (в HTML5). Противоположный ему класс — :optional — представляет элементы управления формы, как лишенные атрибута required, так и те, атрибут required которых установлен в значение false.

Элементы управления относятся к псевдоклассам :required и :optional только в случаях, когда запрашиваемое ими значение соответственно обязательно или необязательно к вводу перед отправкой данных формы. Ниже приведен наглядный пример.

```
input:required { border: 1px solid #f00;}
input:optional { border: 1px solid #ccc;}

<input type="email" placeholder="enter an email address" required>
<input type="email" placeholder="optional email address">
<input type="email" placeholder="optional email address"
       required="false">
```

Первый элемент input типа email снабжен атрибутом required и поэтому относится к псевдоклассу :required. Второй элемент input представлен псевдоклассом :optional, поскольку данные в него вводить не обязательно. Это же относится и к третьему элементу input, атрибут required которого установлен в значение false.

Для форматирования указанных элементов управления можно использовать стилевые правила, включающие селекторы атрибутов.

```
input[required] {border: 1px solid #f00;}
input:not([required]) {border: 1px solid #ccc;}
```

Элементы управления, не поддерживающие ввод данных, не представляются ни псевдоклассом :required, ни :optional.

Селектор проверяемых элементов

Элементы управления, требующие проверки вводимых в них данных на соответствие типу, относятся к псевдоклассу :valid. В противоположность ему псевдокласс :invalid представляет элементы управления, данные в которые вводятся без дополнительной проверки.

К псевдоклассам `:valid` и `:invalid` относятся далеко не все элементы управления, а только поддерживающие передачу данных браузеру для проверки. Например, элемент `div` не будет представляться ни одним из указанных псевдоклассов ни при каких обстоятельствах. В противоположность ему элемент `input` будет отнесен к одному из классов в зависимости от требований, выдвигаемых к интерфейсу формы.

Ниже приведен пример использования псевдоклассов для задания фоновых изображений полям ввода адресов электронной почты — они отличаются в зависимости от результата проверки введенных данных (рис. 2.40).

```
input[type="email"]:focus {
   background-position: 100% 50%;
   background-repeat: no-repeat;
}
input[type="email"]:focus:invalid {
   background-image: url(warning.jpg);
}
input[type="email"]:focus:valid {
   background-image: url(checkmark.jpg);
}

<input type="email">
```

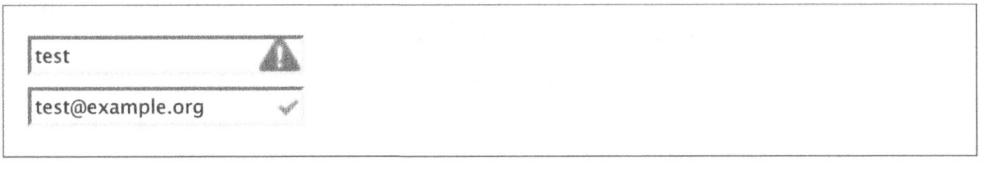


Рис. 2.40. Форматирование элементов управления, в которые введены неправильные (вверху) и правильные (внизу) данные

Отнесение элемента управления к одному из псевдоклассов сильно зависит от того, насколько правильно пользовательский агент интерпретирует введенные в него данные. Например, к концу 2017 года большинство пользовательских агентов относило незаполненное поле ввода адреса электронной почты к псевдоклассу `:valid`, даже несмотря на то, что пустая строка не является правильным адресом. До тех пор пока служба проверки введенных данных допускает столь досадные ошибки, старайтесь взвешенно подходить к использованию селекторов проверяемых элементов в стилевых правилах.

Селекторы элементов со значениями из заданного диапазона

В селекторах элементов этой категории используются следующие два псевдокласса. Псевдокласс `:in-range` представляет элементы управления, в которые введены значения, попадающие в заранее заданный диапазон, определяемый атрибутами `min` и `max` (в HTML5). Его антипод — псевдокласс `:out-of-range` — соответствует элементам управления, значения которых находятся вне определенного диапазона

(меньше значения, указанного атрибутом min, и больше значения, установленного атрибутом max).

Например, следующий код устанавливает форматирование поля, в которое допускается вводить только числовые значения, попадающие в диапазон 0–1000.

```
input[type="number"]:focus {
   background-position: 100% 50%;
   background-repeat: no-repeat;
}
input[type="number"]:focus:out-of-range {
   background-image: url(warning.jpg);
}
input[type="number"]:focus:in-range {
   background-image: url(checkmark.jpg);
}
<input id="nickels" type="number" min="0" max="1000" />
```

К пседоклассам :in-range и :out-of-range относятся только элементы, тип которых предполагает проверку введенных значений на соответствие целевому диапазону. В частности, в их число не входят элементы input, атрибут type которых имеет значение tel.

В HTML5 включена поддержка атрибута step, определяющего шаг значений, которые допускается вводить в элемент управления. Если в поле, снабженное этим атрибутом, ввести значение, которое попадает в диапазон, определяемый атрибутами min и max, но не соответствует шагу, то оно будет отнесено сразу к двум псевдоклассам: :invalid и :in-range. Таким образом, ввод в элемент управления неправильного значения не приводит к автоматическому исключению его из псевдокласса :in-range.

В следующем примере вводимое в поле значение представляется красным цветом и полужирным начертанием, поскольку число 23 попадает в допустимый диапазон, но не кратно 10.

```
input[type="number"]:invalid {color: red;}
input[type="number"]:in-range {font-weight: bold;}

<input id="by-tens" type="number" min="0" max="1000" step="10"
       value="23" />
```

Селекторы элементов с атрибутами чтения-записи

Элементы, содержимое которых разрешается редактировать пользователям, относятся к псевдоклассу :read-write, в то время как псевдокласс :read-only представляет элементы, недоступные для изменения из пользовательского интерфейса. Таким образом, пользователи могут изменять значение только тех элементов управления, которые относятся к псевдоклассу :read-write.

В частности, чтобы представляться псевдоклассом :read-write, элемент input должен находиться во включенном состоянии и не содержать атрибута readonly или же включать атрибут contenteditable. Все остальные элементы относятся к псевдоклассу :read-only.

По умолчанию ни одно из приведенных правил не будет применяться в документе, поскольку элементы `textarea` доступны и для чтения, и для изменения, а элементы `pre` — только для чтения.

```
textarea:read-only {opacity: 0.75;}
pre:read-write:hover {border: 1px dashed green;}
```

Чтобы применить правила, в базовую разметку элементов необходимо внести определенные изменения.

```
<textarea disabled></textarea>
<pre contenteditable>Введите свой код!</pre>
```

Благодаря атрибуту `disabled` элемент `textarea` становится доступным только для чтения, а потому приобретает стилевое оформление, устанавливаемое первым правилом. Подобным образом включение атрибута `contenteditable` в элемент `pre` делает его доступным и для чтения, и для изменения. В результате он включается в псевдокласс `:read-write` и получает форматирование, заданное вторым правилом.

Псевдокласс `:target`

При включении в гиперссылку идентификатора фрагмента документа он называется *целевым элементом*. В CSS к такому элементу можно обращаться по имени, что позволяет назначать ему собственные стилевые правила. Целевые элементы, представленные в URL именованными идентификаторами, относятся к псевдоклассу `:target`.

Даже если вам не знакомо понятие "идентификатор фрагмента документа", вы, скорее всего, сталкивались с ним. Рассмотрим следующий пример:

```
http://www.w3.org/TR/css3-selectors/#target-pseudo
```

В URL идентификатор фрагмента указывается в конце адреса страницы после символа # — в данном случае он представлен ключевым словом `target-pseudo`. Если URL страницы (`http://www.w3.org/TR/css3-selectors/`) включает элемент с идентификатором `target-pseudo`, то именно он представляет целевой фрагмент документа.

Псевдокласс `:target` применяется для единого форматирования сразу всех целевых элементов документа или назначения разных стилевых правил элементам, относимых к определенному типу, — например, заголовкам, таблицам и т.п. Пример использования псевдокласса `:target` для оформления документа HTML приведен на рис. 2.41.

```
*:target {border-left: 5px solid gray; background: yellow
   url(target.png) top right no-repeat;}
```

Псевдокласс `:target` не находит применения в следующих ситуациях:

1) в URL-адресе действительной веб-страницы отсутствует идентификатор фрагмента;
2) в действительной веб-странице отсутствуют целевые элементы, которые соответствуют идентификатору фрагмента, включенному в конец ее URL-адреса.

Намного интереснее ситуация складывается, когда один идентификатор фрагмента представляет сразу несколько элементов документа — например, трех экземпляров элемента `<div id="target-pseudo">`.

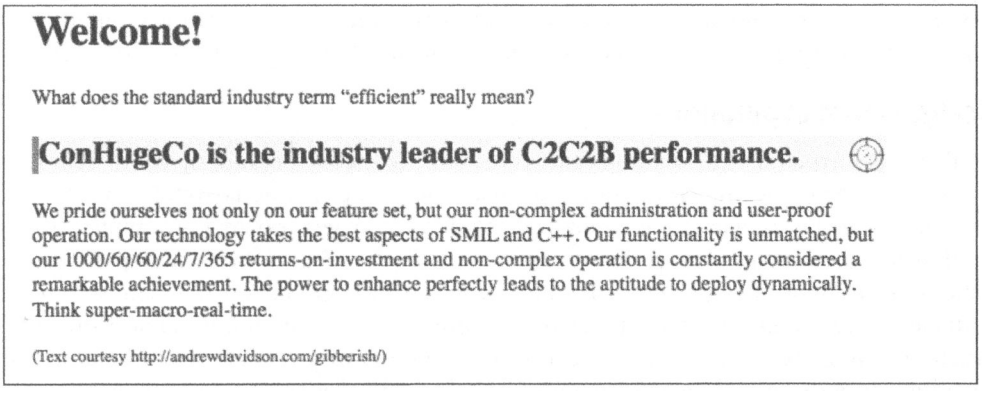


Рис. 2.41. Выделение целевого элемента на странице

Как ни странно, но в подобных случаях для стилевого оформления элементов не нужно оговаривать специальные условия: согласно спецификации CSS стилевое форматирование будет применяться ко всем трем элементам, независимо от того, как они воспринимаются браузером. В результате стилевое правило, селектор которого содержит псевдокласс `:target`, применяется ко всем целевым фрагментам документа.

Псевдокласс `:lang`

Этот псевдокласс применяется при стилевом форматировании элементов, характеристики которых зависят от языка, используемого при их отображении. С технической точки зрения псевдокласс `:lang` подобен селектору атрибута `|=`. Например, следующие правила устанавливают курсивное начертание для элементов, отображаемых на французском языке.

```
*:lang(fr) {font-style: italic;}
*[lang|="fr"] {font-style: italic;}
```

Главное отличие селекторов приведенных выше типов заключается в источнике, из которого браузер получает информацию о языке, на котором выводится содержимое документа. Для применения стилевого правила, включающего селектор атрибутов, элемент должен содержать атрибут `lang`. С другой стороны, действие селектора, основанного на псевдоклассе `:lang`, распространяется на всех потомков элемента, который он представляет. Вот что об этом сказано в официальной документации к спецификации CSS Selectors 3.

> В HTML язык, на котором выводится документ, определяется атрибутом `lang`, частично атрибутом `meta` и протоколом передачи данных (устанавливается в HTTP-заголовках). В XML для этих целей применяется атрибут `xml:lang`, а также специальные инструменты, свойственные текущему расширению XML.

Обратите внимание на то, что псевдокласс `:lang` работает независимо от того, снабжены элементы атрибутом `lang` или нет. Таким образом, он намного функциональнее,

чем селектор атрибутов, а потому предпочтительнее для решения большинства задач форматирования текстового содержимого, введенного на разных языках.

Псевдокласс отрицания

Все рассмотренные до сих пор селекторы объединяет одно: они прямого действия. Иными словами, они представляют содержимое, оформление которого будет изменяться, исключая те части документа, которые не соответствуют критериям отбора.

Для предоставления разработчикам обратной возможности — инвертирования набора элементов, представленных селекторами, в спецификацию CSS Selectors 3 включена поддержка псевдокласса отрицания: `:not()`. Функционально он существенно отличается от остальных псевдоклассов и имеет ряд ограничений на использование в селекторах стилевых правил. Как и прежде, начнем его изучение с простого примера.

Предположим, специальное форматирование требуется применить ко всем элементам списка, которые лишены атрибута `class`, установленного в значение `moreinfo` (рис. 2.42). Решить эту задачу с помощью селекторов, описанных ранее, очень сложно, а порой просто невозможно. В общем случае сначала потребовалось бы выделить курсивом все элементы списка, а затем сбросить форматирование для элементов `ul`, относящихся к классу `moreinfo`, удостоверившись в том, что возврат к исходному начертанию выполняется после выделения элементов курсивом и имеет более высокий приоритет. Псевдокласс отрицания сокращает описанную выше процедуру до применения всего одного правила.

```
li:not(.moreinfo) {font-style: italic;}
```

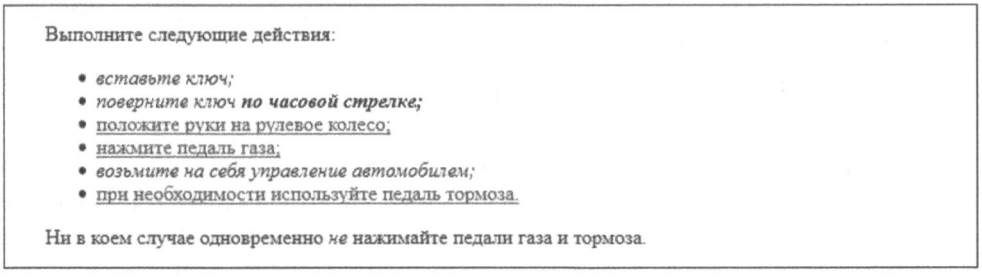


Рис. 2.42. Выделение курсивом элементов списка, не относящихся к определенному классу

Псевдокласс `:not()` вводится исключительно после названия элемента и применяется только к простым селекторам, указываемым в скобках. Согласно определению W3C, к простым относятся селекторы типов, атрибутов, классов, идентификаторов, универсальный селектор и псевдоклассы.

В общем случае простой селектор представляет элементы, не ориентируясь на родительско-дочерние взаимосвязи.

Учтите, что одновременно псевдокласс `:not()` применяется только к одному из указанных простых селекторов — в скобки не разрешается подставлять группу или комбинацию селекторов. Буквально это означает, что отрицание неприменимо к

селекторам дочерних элементов, поскольку пробел, которым разделяются элементы в селекторе, относится к комбинаторам. Данное ограничение, скорее всего, будет снято в будущих версиях селектора отрицания, но даже с упрощенным синтаксисом он представляет необычайно функциональный инструмент стилевого форматирования.

Вернувшись к рассмотрению предыдущего примера, представим, что требуется выделить курсивом все элементы, относящиеся к классу `moreinfo` и не относящиеся к элементам списка. Задача решается с помощью следующего правила, а результат ее применения показан на рис. 2.43:

```
.moreinfo:not(li) {font-style: italic;}
```

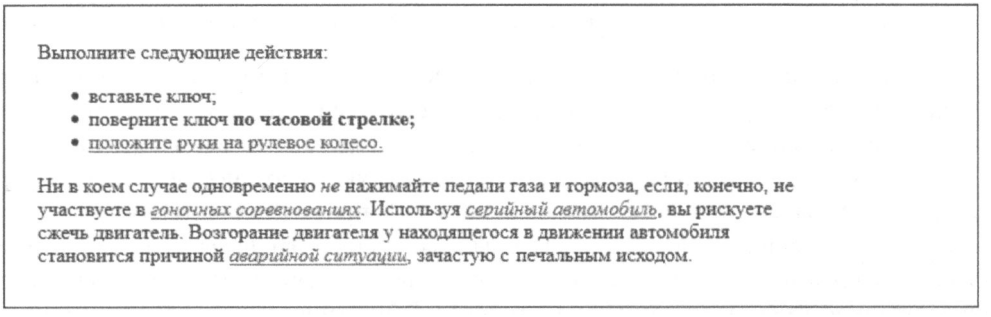


Рис. 2.43. Выделение курсивом элементов, относящихся к определенному классу и не являющихся элементами списка

Буквально применяемый в правиле селектор указывает на все элементы, атрибут `class` которых имеет значение `moreinfo`, при условии, что они не являются элементами `li`. Подобным образом селектор `li:not(.moreinfo)` представляет все элементы `li`, которые лишены атрибута `class`, установленного в значение `moreinfo`.

Согласно приведенной выше спецификации, псевдокласс `:not()` можно применять к универсальному селектору, хотя это и бессмысленно. Что не удивительно, поскольку, например, селектор `p:not(*)` представляет все элементы `p`, которые не относятся к элементам. Формально в такой записи нет ошибки, хотя сложно представить ситуацию, в которой элементы перестают быть элементами. Подобным образом селектор `p:not(p)` не представляет ни один из элементов документа. В более осмысленном варианте (`p:not(div)`) псевдокласс отрицания позволяет выделить любые элементы `p`, не являющиеся элементами `div`, — иными словами, их все. Сложно представить задачи, при решении которых может потребоваться использовать столь неоднозначные синтаксические конструкции.

При необходимости псевдокласс `:not()` можно включать в более сложные селекторы. В частности, для применения специального форматирования ко всем таблицам документа, не являющимся дочерними по отношению к элементу `section`, в правиле нужно использовать селектор `*:not(section) > table`. Наряду с этим селектор `table *:not(thead) > tr > th` представляет заголовочные ячейки таблицы, не относящиеся к ее заголовку. Результат применения стилевого правила с таким селектором к таблице показан на рис. 2.44.

State	Post	Capital	State Bird
Alabama	AL	Montgomery	Yellowhammer
Alaska	AK	Juneau	Willow Ptarmigan
Arizona	AZ	Phoenix	Cactus Wren
Arkansas	AR	Little Rock	Mockingbird
California	CA	Sacramento	California Quail
Colorado	CO	Denver	Lark Bunting
Connecticut	CT	Hartford	American Robin
Delaware	DE	Dover	Blue Hen Chicken
Florida	FL	Tallahassee	Northern Mockingbird
Georgia	GA	Atlanta	Brown Thrasher
State	Post	Capital	State Bird

Рис. 2.44. Форматирование заголовочных ячеек, исключая ячейки заголовка таблицы

При построении сложных селекторов псевдоклассы отрицания не допускается вкладывать один в другой. Таким образом, конструкция `p:not(:not(p))` будет проигнорирована браузером. И это понятно, ведь она представляет все элементы `p` документа. Также в скобках псевдокласса `:not()` нельзя указывать псевдоэлементы (рассматриваются далее), поскольку они не относятся к простым селекторам.

С другой стороны, в сложные селекторы можно включать целые цепочки псевдоклассов отрицания, расширяя условие отбора новыми элементами. Например, следующее правило указывает применять стилевое форматирование к элементам, относящимся к классу `link`, которые не являются ни элементами списков, ни абзацами:

```
*.link:not(li):not(p) {font-style: italic;}
```

Селектор данного правила указывает на элементы, атрибут `class` которых имеет значение `link`, но не являющиеся элементами `p` или `li`.

Будьте внимательны при составлении сложных селекторов, включающих псевдоклассы отрицания, поскольку даже простые условия инверсного отбора элементов сложны для анализа и чреваты ошибками. В качестве примера рассмотрим следующий код.

```
div:not(.one) p {font-weight: normal;}
div.one p {font-weight: bold;}

<div class="one">
   <div class="two">
      <p>Текстовый абзац!</p>
   </div>
</div>
```

В результате стилевого форматирования абзац получает полужирное, а не обычное начертание. Внимательно изучив код, становится понятным, что к элементу `p` применяются сразу оба правила. Первое из них относится к элементам `p`, являющимся потомками элемента `div`, атрибут `class` которого *не содержит* слово `one` (`<div class="two">`), а второе задается для элементов `p` — потомков элемента `div`, *содержащего* слово `one` в названии класса. В разрезе представленного выше фрагмента документа выполняются оба стилевых правила — для устранения конфликтной

ситуации приходится учитывать приоритетность каждого из них. Согласно принципу каскадности применения стилей в документе, второе правило имеет более высокий приоритет, поскольку находится "ближе" к абзацу в объектной модели документа.

Псевдоэлементы

Наряду с псевдоклассами, предназначенными для форматирования элементов несуществующих классов, в CSS применяется еще один тип фиктивных инструментов, существенно упрощающих создание некоторых визуальных эффектов: *псевдоэлементы*. Впервые поддержка псевдоэлементов была реализована в спецификации CSS2. Несмотря на небольшое количество (их всего четыре), псевдоэлементы позволяли изменять оформление начальных букв и строк элементов, а также определять стилевое оформление содержимого, находящегося перед ними и после них. В последующие версии CSS добавлена поддержка некоторых других псевдоэлементов (в первую очередь `::marker`) — в последующих главах книги будут описаны наиболее актуальные из них. В этой главе вы познакомитесь только с первыми четырьмя псевдоэлементами, представленными в спецификации CSS2, поскольку именно они показали перспективность нового способа стилизации документов.

В отличие от псевдоклассов, название которых начинается с двоеточия, перед именем псевдоэлемента вводится двойное двоеточие, как, например, в селекторе `::first-line`, что позволяет различать их в коде стилевых правил. Такой формат записи был принят не всегда — в CSS2 и псевдоклассы, и псевдоэлементы предварялись двоеточием. Для обеспечения обратной совместимости многие браузеры поддерживают старый вариант синтаксиса псевдоэлементов (одинарное двоеточие), поэтому будьте внимательны! Чтобы повысить удобочитаемость кода и упростить его для дальнейшего редактирования, в собственных стилевых правилах старайтесь предварять имена псевдоэлементов исключительно двойным двоеточием. Не забывайте о том, что в один прекрасный момент браузеры перестанут поддерживать старый вариант синтаксиса!

Учтите, что псевдоэлементы всегда указываются в конце селекторов стилевых правил. Таким образом, селектор `p::first-line em` будет недействительным, поскольку в нем псевдоэлемент располагается перед целевым элементом (расположен в конце селектора). Из этого правила следует, что в отдельно рассматриваемый селектор допускается включать только один псевдоэлемент, хотя это ограничение может быть отменено в будущих версиях CSS.

Оформление первой буквы

За стилевое форматирование первой буквы или первого символа пунктуации (если предложение начинается с него) любого строчного элемента отвечает псевдоэлемент `::first-letter`. Следующее правило назначает красный цвет первой букве абзаца:

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

Чаще всего псевдоэлемент `::first-letter` применяется для получения эффекта буквицы. Он предполагает двукратное увеличение размера шрифта первой буквы первого абзаца документа.

```
p:first-of-type::first-letter {font-size: 200%;}
```

Результат применения этого правила к импровизированному заголовку показан на рис. 2.45.

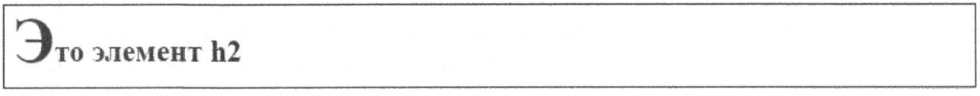


Рис. 2.45. Эффект стилевого форматирования с помощью правила, включающего псевдоэлемент `::first-letter`

Правило устанавливает форматирование для фиктивного (искусственно образованного) элемента, заключающего в себе первую букву каждого абзаца. В HTML такой элемент можно представить следующим кодом.

```
<p><p-first-letter>Э</p-first-letter>лемент p, содержащий
   буквицу</h2>
```

Стили, определенные с помощью селектора `::first-letter`, применяются только к содержимому фиктивного элемента. В действительности `<p-first-letter>` в HTML-код добавлять не нужно — представленный им элемент отсутствует в объектной модели документа, поэтому и называется фиктивным, или псевдоэлементом. Он создается пользовательским агентом в процессе стилевого форматирования документа и приобретает внешний вид, предписываемый правилом, которое включает селектор `::first-letter`. Еще раз повторим: применение псевдоэлементов не влечет за собой добавления новых тегов в разметку документа. Пользовательский агент задает стилевое форматирование для фиктивного элемента самостоятельно.

Первая буква элемента определяется как первый типографский знак исходного элемента, не предваряемый другим содержимым (например, графическим объектом). Спецификация однозначно определяет применимость псевдоэлемента к типографскому знаку, а не символу, поскольку во многих языках буквы образуются сочетанием нескольких символов. В псевдоэлемент `::first-letter` также включаются все символы пунктуации (даже если их несколько), расположенные непосредственно перед первым знаком и после него.

Оформление первой строки

Подобно первому знаку, специальное форматирование можно задать для первой строки элемента. В следующем примере первые строки всех абзацев документа отображаются шрифтом увеличенного размера и окрашиваются в фиолетовый цвет.

```
p::first-line {
   font-size: 150%;
   color: purple;
}
```

Результат применения данного правила к первой строке одного из абзацев документа показан на рис. 2.46. Стилевое форматирование применяется ко всем словам первой строки независимо от ее длины. Если первая строка элемента состоит всего из пяти слов, то размер и цвет изменяется только у них. Если же по какой-то причине количество слов в первой строке увеличится (например, до 30), то форматирование получат они все, независимо от того, насколько длинной становится строка.

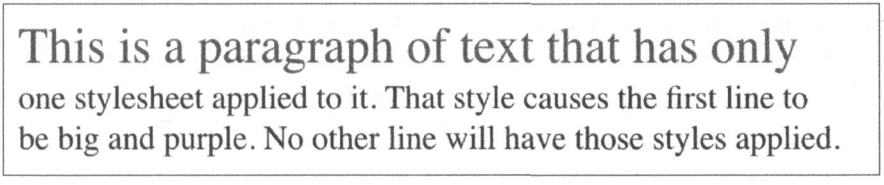


Рис. 2.46. Форматирование первой строки с помощью правила, задаваемого для псевдоэлемента `::first-line`

В HTML равнозначный элемент, который заключает текстовую строку, начинающуюся словом "This" и заканчивающуюся словом "only", представляется следующим кодом.

```
<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
big and purple. No other line will have those styles applied.
</p>
```

Если длина первой строки уменьшится до 7 слов, то фиктивный элемент `::p-first-line` будет заканчиваться на слове "that". При увеличении или уменьшении размера экранного шрифта, а также сужении или расширении окна браузера длина первой строки динамически меняется, что приводит к изменению количества слов в ней. Независимо от этого, увеличенным шрифтом и фиолетовым цветом отображаются только те слова, которые включаются в первую строку каждого из представлений.

На длину первой строки влияет большое количество настроек, включая размер шрифта, межсимвольный интервал, ширину родительского контейнера и т.п. В зависимости от структуры документа и длины строки она может заканчиваться в середине вложенного элемента. При разрыве первой строки по вложенному элементу, такому как `em` или `a`, стилевое форматирование псевдоэлемента `::first-line` применяется только в той его части, которая находится в первой строке.

Ограничение на использование псевдоэлементов `::first-letter` и `::first-line`

Псевдоэлементы `::first-letter` и `::first-line` применяются только к блочным элементам, например заголовкам и абзацам. Они не могут представлять строчные элементы, такие, как, например, гиперссылки. Кроме того, для форматирования указанных псевдоэлементов может применяться лишь ограниченный набор свойств (табл. 2.5).

Таблица 2.5. Свойства форматирования псевдоэлементов

`::first-letter`	`::first-line`
Все свойства шрифтов	Все свойства шрифтов
Все свойства фона	Все свойства фона
Все свойства внешнего вида	Все свойства отступов
Все текстовые свойства строчных элементов	Все свойства полей
Все свойства положения строчных элементов	Все свойства границ
Все свойства границ	Все свойства внешнего вида
`box-shadow`	Все текстовые свойства строчных элементов
`color`	`color`
`opacity`	`opacity`

Оформление содержимого, добавляемого перед псевдоэлементом и после него

Рассмотрим задачу, в рамках которой заголовок второго уровня нужно снабдить двумя серебристыми квадратными скобками, располагаемыми в начале. Задача решается с помощью такого правила:

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

В CSS для вставки *генерируемого содержимого*, получающего специальное стилевое форматирование (рис. 2.47), в начало или конец элемента применяются псевдоэлементы `::before` и `::after`.

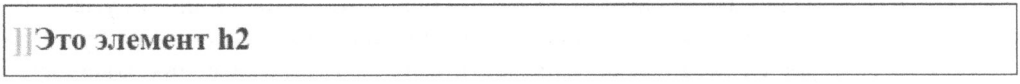


Рис. 2.47. Вставка генерируемого содержимого перед элементом

Псевдоэлементы позволяют определить форматирование генерируемого содержимого как перед указанным в селекторе элементом, так и после него. Например, с помощью следующего правила в конец документа добавляется строка, извещающая о его окончании:

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

О том, что такое генерируемое содержимое (включая роль псевдоэлементов `::before` и `::after` в его форматировании), речь пойдет в главе 15.

Резюме

Селекторы используются для представления элементов разметки документа в правилах CSS, применяемых для их стилевого оформления. С помощью всего одного селектора можно определить форматирование большого количества элементов или, наоборот, максимально точно указать избранные элементы. Привязка стилевых

правил к задаваемым селекторами элементам делает таблицы стилей необычайно компактным и гибким инструментом верстки документов, обладающим, ко всему прочему, малым размером и высокой скоростью загрузки.

От того, насколько правильно подобраны селекторы, порой зависит, будут ли вообще стилевые правила применяться пользовательским агентом к документу. Ошибки при составлении селекторов приводят к фатальным нарушениям в оформлении документа. Чтобы не ошибиться, нужно понимать, каким образом они связаны с иерархической структурой документа, а также научиться определять правильный порядок их применения.

Приоритетность и каскадирование стилей

В предыдущей главе рассматривались принципы иерархического структурирования документов и селекторы, указывающие на элементы, к которым применяются стилевые правила. Каждый документ состоит из большого количества иерархически упорядоченных элементов, что позволяет селекторам обращаться к ним, основываясь на информации о взаимосвязях элементов разных уровней, их атрибутах, контекстных данных и других сведениях. Иерархическая структура лежит в основе механизма стилевого оформления документа и является ключевым элементом в понимании одного из главных принципов его реализации — наследования свойств.

Под наследованием понимают частичную или полную передачу стилевого оформления элемента его потомкам. Определяя форматирование для каждого из элементов документа, пользовательский агент должен учитывать не только все унаследованные им свойства, но и *приоритетность* применения правил, а также их происхождение. Процесс получил название *каскадирование*, или каскадное применение стилей. В этой главе описана взаимосвязь трех главных принципов, влияющих на порядок применения стилевых правил ко всем элементам документа: приоритетность, наследование и каскадирование. Различие между наследованием и каскадированием проще всего понять на примере правила `h1 {color: red; color: blue;}`. Каскадирование применяется для определения конечного форматирования элемента `h1`, а наследование объясняет, почему у элемента `span`, вложенного в элемент `h1`, синий цвет.

Приоритетность

В главе 2 описано огромное количество способов представления элементов в стилевых правилах. Реальность такова, что внешний вид одного и того же элемента чаще всего назначается несколькими правилами, в которых используются разные селекторы. Рассмотрим три пары правил, каждая из которых определяет форматирование разных элементов.

```
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;}
```

Только одно из двух правил каждой пары применяется к целевым элементам. Но как узнать, какой цвет из двух возможных получит каждый элемент?

Порядок применения стилевых правил в документе определяется их *приоритетностью* (или *специфичностью*). Пользовательский агент определяет приоритетность селекторов всех без исключения правил, применяемых в документе, и включает их в соответствующие объявления. Если одному и тому же элементу одновременно назначаются противоречивые стилевые правила, то будет применяться правило с большей приоритетностью.

В документе порядок применения правил определяется не только приоритетностью селекторов. На него также оказывает влияние каскадирование, детально рассмотренное в разделе "Каскадирование".

Приоритетность селектора определяется его компонентами и представляется четырехзначным выражением формата `0,0,0,0`. На приоритетность селектора оказывают влияние следующие факторы.

- Каждый идентификатор (атрибут `id`) увеличивает приоритетность на величину `0,1,0,0`.
- Селекторы классов, псевдоклассов и атрибутов добавляют к приоритетности величину `0,0,1,0`.
- За каждый селектор элементов и псевдоэлементов начисляется приоритетность `0,0,0,1`. В CSS2 селекторы псевдоэлементов в расчете приоритетности участие не принимали, но уже в CSS2.1 этот недостаток был устранен, и они вносят свой вклад в конечное значение.
- Комбинаторы и универсальные селекторы на значение приоритетности не влияют.

Проще всего научиться определять приоритетность правил на реальных примерах.

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

Если применить второе и пятое правило из приведенных выше примеров к элементу `em`, то он получит темно-бордовый цвет (`maroon`), поскольку у пятого правила значение приоритетности больше, чем у второго.

В качестве упражнения вычислим значение приоритетности для следующих правил и определим победителя в каждой из пар.

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

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

Правила с большей приоритетностью обозначены в комментариях. Обратите внимание на принципы сравнения значений. Во второй паре большей приоритетностью обладает правило `h2.grape`, поскольку значение `0,0,1,1` больше `0,0,0,1`. В третьей паре более приоритетным будет второе правило, так как `0,1,0,1` больше, чем `0,0,1,7`. Второй разряд всегда весомее первого, поэтому значение `0,0,1,0` всегда больше `0,0,0,13`.

Чем больше порядок разряда, тем весомее его вклад в общее значение приоритетности. Таким образом, `1,0,0,0` больше любых других значений, первый разряд которых содержит `0` (независимо от чисел в остальных разрядах). Значение `0,1,0,1` вполне закономерно превышает `0,0,1,7`, поскольку в его третьем разряде указано `1`, а в этой позиции второго значения находится `0`.

Приоритетность объявлений

Значение приоритетности селектора назначается всем объявлениям, к которым он имеет отношение. Рассмотрим следующий пример:

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

Пользовательский агент представляет это правило в виде двух более простых правил.

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

Оба правила имеют значение приоритетности `0,0,0,1`, которое применяется к каждому из объявлений. Подобный подход характерен для любых других сгруппированных объявлений. Проанализируем следующее правило:

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

Пользовательский агент представляет его в виде такого набора правил.

```
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 */
```

Разделение правила на составляющие компоненты позволяет правильно оценить его приоритетность при сбое одного из объявлений. В качестве примера рассмотрим приоритетность следующих правил.

```
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 */
```

Результат их применения к документу, представленному приведенным ниже фрагментом HTML-кода, показан 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>
```

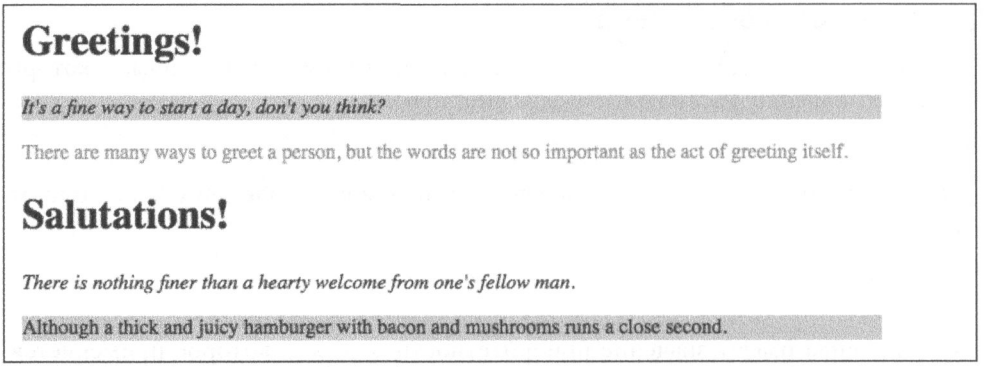


Рис. 3.1. Форматирование документа правилами с разной приоритетностью

Конечные стилевые правила, применяемые к элементам документа, определяются пользовательским агентом в результате сравнений значений их приоритетности. Каждому элементу назначается форматирование, имеющее наибольшую приоритетность. Подобный анализ необходимо проводить для всех без исключения объявлений, правил и селекторов. К счастью, эта задача решается пользовательским агентом полностью автоматически. От приоритетности правил также зависит порядок их каскадирования, о чем рассказано далее.

Приоритетность универсального селектора

При расчете значений приоритетности универсальный селектор не учитывается. Иными словами, приоритетность правил с универсальным селектором представлена

значением 0,0,0,0 и не равнозначна таковой в правилах с отсутствующей приоритетностью (рассмотрены в разделе "Наследование"). Таким образом, с помощью следующих двух правил абзацам, заключенным в элемент div, назначается черный цвет, а всем остальным элементам — серый.

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

Как и следовало ожидать, добавление универсального селектора в правило, включающее селекторы других типов, не приводит к изменению значения его приоритетности. Исходя из этого, можно смело утверждать, что приведенные ниже правила обладают одинаковой приоритетностью.

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

По сравнению с универсальными селекторами комбинаторы вообще не учитываются при вычислении уровня приоритетности — на порядок применения правил в документе они не влияют.

Приоритетность селекторов идентификаторов и атрибутов id

При определении приоритетности важно понимать различие между селекторами идентификаторов и селекторами атрибутов id. Вернемся к рассмотрению третьей пары правил, описанных выше.

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

Легко заметить, что селектор идентификатора #answer (второе правило) добавляет к приоритетности правила значение 0,1,0,0, в то время как вклад в нее селектора атрибута [id="totals"] составляет всего 0,0,1,0. Таким образом, в результате применения следующих двух правил к элементу с атрибутом id, имеющим значение meadow, он окрашивается в зеленый цвет.

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

Приоритетность встроенных стилей

До этого момента рассматривались правила, приоритетность которых представлялась значениями, начинающимися с 0 (в четвертом разряде). А все потому, что наибольшую приоритетность имеют объявления встроенных стилей. Рассмотрим следующий пример CSS-кода и фрагмент HTML-документа.

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

```
<h1 style="color: green;">The Meadow Party</h1>
```

Применение указанного правила к элементу h1 приводит к окрашиванию его текста зеленым цветом, независимо от форматирования, устанавливаемого другими

правилами. Заголовки первого уровня будут зелеными благодаря большей приоритетности — 1,0,0,0.

Буквально это означает, что применение к таким элементам правил других типов, например включающих селекторы атрибута id, не изменяет форматирование. Давайте чуть изменим последний пример, представив элемент h1 следующим образом.

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

<h1 id="meadow" style="color: green;">The Meadow Party</h1>
```

Благодаря использованию встроенного стиля текст элементов h1 по-прежнему окрашивается зеленым цветом.

Важность стилей

Отдельные стили настолько важны, что должны применяться первостепенно, независимо от их базового уровня приоритетности. В CSS такие стили называются важными (по вполне очевидным причинам) и обозначаются с помощью специального ключевого слова !important, добавляемого в конец объявления, как раз перед точкой с запятой:

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

При неправильном расположении ключевого слова !important в объявлении правило будет проигнорировано. Оно добавляется исключительно перед завершающей точкой с запятой, и ни в каком другом месте. К объявлению важных стилей нужно относиться крайне внимательно, особенно в свойствах, описываемых сразу несколькими ключевыми словами, например в таком случае:

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

Если расположить ключевое слово !important в любом другом месте объявления свойства font, то пользовательский агент не сможет определить, к чему именно оно относится, и отменит сразу все правило.

Те читатели, которые имеют опыт программирования, будут крайне удивлены синтаксисом объявления важных стилей, поскольку восклицательный знак обычно обозначает логическое отрицание и в данном случае может трактоваться как "not important". Как бы там ни было, в CSS он лишен данного смысла и всего лишь указывает на важность объявления. С этим ничего не поделаешь, но обязательно нужно учитывать данный нюанс при написании стилевых правил.

В действительности объявления, включающие ключевое слово !important, не изменяют приоритетность правила, а рассматриваются отдельно от них. Приоритетность важных объявлений устанавливается отдельно от приоритетности объявлений регулярных типов. Таким образом, противоречивость важных стилей устраняется отдельно от регулярных стилей, и наоборот. Но при возникновении конфликтной ситуации между ними предпочтение всегда отдается важным стилям.

На рис. 3.2 показан результат применения следующего форматирования к указанному фрагменту HTML-документа.

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

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

Рис. 3.2. Важные стили характеризуются большей приоритетностью, чем регулярные

Детально важные стили и их назначение описаны в разделе "Каскадирование".

Наследование

На порядок применения стилевых правил в документе оказывает влияние не только их приоритетность, но и еще одна ключевая концепция CSS: *наследование*. Наследование представляет собой механизм передачи стилевого форматирования от родительского элемента к его потомкам. Например, устанавливая цвет элементов `h1`, вы изменяете цвет текста не только заголовков первого уровня, но и всех его дочерних элементов.

```
h1 {color: gray;}

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

В данном случае серый цвет назначается исходному тексту заголовка, а также тексту дочернего элемента `em`, наследующего свойство `color` от элемента `h1`. Если бы наследования не происходило, то у элемента `em` сохранялся бы цвет по умолчанию — черный. Для установки им одинакового цвета пришлось бы использовать два разных правила с одинаковыми значениями свойства `color`.

В качестве следующего примера рассмотрим неупорядоченный список, форматирование элемента `ul` которого определяется таким правилом:

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

Предполагается, что стилевое оформление элемента `ul` будет распространяться на все элементы списка и содержимое вложенных в них элементов. Благодаря принципу наследования наши ожидания полностью оправдываются, и неупорядоченный список приобретает вид, показанный на рис. 3.3.

- Oh, don't you wish
- That you could be a fish
- And swim along with me
- Underneath the sea

1. Strap on some fins
2. Adjust your mask
3. Dive in!

Рис. 3.3. Наследование стилей

С наследованием проще всего знакомиться, представив иерархическую структуру документа в графическом виде. На рис. 3.4 показана иерархическая структура простейшего документа, включающего всего два списка: неупорядоченный и упорядоченный.

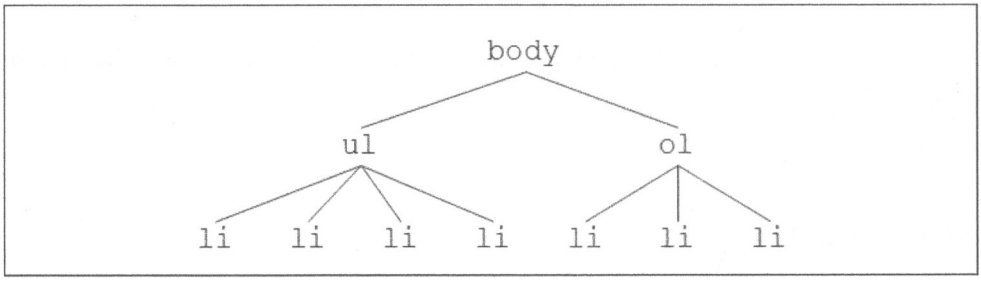


Рис. 3.4. Иерархическая структура документа

Исходно применяясь к элементу `ul`, объявление `color: gray` распространяется на все его дочерние элементы и далее — вниз по иерархической структуре — до последнего потомка. Стилевые правила никогда не наследуются снизу вверх — родительские элементы не могут заимствовать форматирование своих потомков.

В HTML существует всего одно исключение из правила наследования: стили фона могут передаваться вверх по иерархической структуре: от элемента `body` к корневому элементу (`html`), определяющему границы документа. Такая ситуация возможна только в случаях, когда фон назначен элементу `body`, а у элемента `html` он отсутствует.

Наследование — это один из фундаментальных принципов CSS, ставший настолько привычным, что воспринимается как нечто самой собой разумеющееся. Тем не менее с ним приходится считаться при стилевом оформлении всех без исключения документов.

Учтите, что наследованию подлежат далеко не все свойства, а только те из них, применение которых к вложенным элементам не приводит к получению противоречивых результатов. К таким свойствам, в частности, относится `border` (добавляющее границу к элементам). На рис. 3.5 наглядно показано, с чем связано такое ограничение. Если бы свойство `border` наследовалось, то документ выглядел бы более

загроможденным — по крайней мере, до тех пор пока задаваемое им форматирование не было бы отменено вручную.

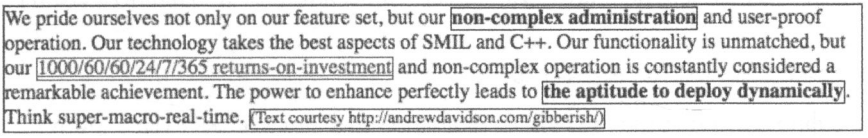


Рис. 3.5. Наследование некоторых стилей крайне нежелательно

Исходя из озвученных выше соображений, наследованию не подлежит большинство свойств форматирования блочных элементов — поля, отступы, границы и фон таких элементов не передаются их потомкам. И в самом деле, при наследовании отступа в 30 пикселей гиперссылками, добавленными в текст абзаца, он будет выглядеть совершенно нечитабельно.

Вторая особенность наследованных правил связана с их приоритетностью — для наследуемых элементов она попросту не учитывается. Данное ограничение кажется надуманным, но только не в ситуациях полного его отсутствия. Рассмотрим наглядный пример форматирования фрагмента HTML-документа двумя стилевыми правилами, результат применения которых показан 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>
```

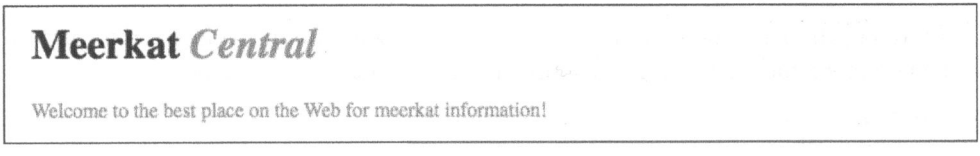


Рис. 3.6. Правила с нулевым значением приоритетности имеют преимущество перед правилами с отсутствующей приоритетностью

В данном случае к элементам применяется первое правило, а не второе, поскольку универсальный селектор добавляет к приоритетности нулевое значение, а наследуемые свойства при ее вычислении вообще не учитываются. Таким образом, элемент `em` окрашивается не в черный, а в серый цвет.

Рассмотренный пример как нельзя нагляднее иллюстрирует возможные последствия бесконтрольного использования универсального селектора в правилах. Представляя *все* элементы документа, он зачастую нарушает форматирование элементов с близкими родственными связями. Конечно, проблема решаема, но всегда проще ее предотвратить, чем устранять последствия включения универсального селектора в правила, в нем не нуждающиеся.

Зачастую отсутствие приоритетности у наследуемых правил становится причиной необычного поведения элементов. Следующее правило указывает окрашивать текст элементов с идентификатором `toolbar` в белый цвет и выводить его на черном фоне:

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

Такое форматирование сохраняется до тех пор, пока элементы, атрибут `id` которых имеет значение `toolbar`, содержат один только текст. При добавлении в них других элементов, например `a` (гиперссылок), к последним будет применяться собственный стиль, по умолчанию определяемый пользовательским агентом. В частности, в большинстве браузеров гиперссылки будут окрашиваться синим цветом, поскольку встроенные стили для элементов `a` в них, скорее всего, задаются следующим правилом:

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

Чтобы исправить ситуацию, в таблице стилей нужно предусмотреть специальное правило для гиперссылок:

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

Применяя отдельное правило к гиперссылкам, можно добиться результата, показанного на рис. 3.7.

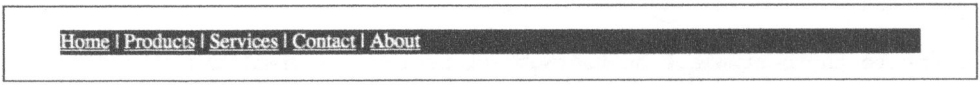


Рис. 3.7. Непосредственное стилевое форматирование вложенных элементов

Альтернативный способ применения к гиперссылкам общего цветового решения заключается в использовании значения `inherit`, описанного в главе 2.

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

Результат применения этих правил такой же, как и предыдущем случае (см. рис. 3.7), поскольку теперь цвет гиперссылок наследуется у родительского элемента.

Каскадирование

Мы умышленно не рассматривали ситуации, в которых к элементу применяются сразу два правила с одинаковым значением приоритетности. Настало время разобраться, как браузер поступает в подобных случаях, предотвращая возникновение конфликтных ситуаций. Рассмотрим такой пример.

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

Какой цвет получит элемент, ведь оба правила имеют одинаковую приоритетность, а потому и шансы на применение? Разумеется, элемент не может окрашиваться сразу двумя цветами, но как узнать, какой из них будет назначен браузером?

Ответ кроется в самом названии технологии CSS (Cascading Style Sheets — каскадные таблицы стилей), в котором ключевым словом является "каскад". Под каскадом в данном случае понимается одновременное применение разных стилевых правил к элементам документа, как в результате сравнения значений приоритетности, так и с учетом принципа наследования. Каскадирование заключается в выполнении следующих действий.

1. Поиск всех правил, имеющих отношение к форматированию отдельно взятого элемента.
2. Определение *важности* каждого из правил, применяемых к элементу. Наибольшую важность имеют правила, объявления которых включают ключевое слово `!important`.
3. Определение *происхождения* стилей, влияющих на внешний вид элемента. Существуют три основных источника происхождения стилей: автора (разработчика), пользователя и браузера. В общем случае наивысший приоритет имеют стили автора. Тем не менее пользовательские стили, содержащие в объявлении ключевое слово `!important`, превалируют над стилями автора, в том числе особой важности (также объявляемые с ключевым словом `!important`). Стили браузера имеют самый низкий приоритет, уступая пальму первенства стилям автора и пользователя.
4. Определение приоритетности для объявлений стилей, применяемых к элементу. Чем больше значение приоритетности, тем бо́льшую важность имеет стиль для элемента.
5. Определение порядка объявления стилей, применяемых для форматирования элемента. Чем позже в коде объявлен стиль, тем выше его важность с точки зрения порядка следования. Правила импортируемой таблицы стилей рассматриваются пользовательским агентом перед правилами импортирующей таблицы стилей.

Чтобы понять, каким образом работает каскадирование, необходимо детально проанализировать каждое из действий, рассмотрев их на реальных примерах.

Некоторые модули CSS добавляют в объектную модель дополнительные (к трем базовым) источники происхождения стилей. Речь идет о переходах и анимации, происхождение которых не рассматривается в примерах данной главы, но детально описано в последующих главах.

Определение важности и происхождения стилей

При назначении элементу сразу нескольких стилей самым важным считается стиль, в объявлении которого имеется ключевое слово `!important`.

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

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

Несмотря на то что цвет элемента также указывается во встроенном стиле, благодаря ключевому слову `!important` он устанавливается с помощью внешнего правила. В результате текст абзаца, а потому и вложенного в него элемента `em` (согласно принципу наследования) окрашивается серым цветом.

Для назначения форматирования с помощью встроенного стиля его также нужно снабдить ключевым словом `!important`. Таким образом, для окрашивания абзацев и всех их дочерних элементов черным цветом приведенный выше код нужно представить в следующем виде.

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

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

Если стилевые правила имеют одинаковую важность, то в расчет принимается их происхождение. В общем случае стили автора (разработчика) считаются приоритетнее стилей пользователя. Рассмотрим следующие одинаковые стили с разным происхождением:

```
p em {color: black;}  /* правило из таблицы стилей автора */
p em {color: yellow;} /* правило из таблицы стилей пользователя */
```

В данном случае текст, представленный полужирным начертанием, имеет черный цвет, а не желтый, так как регулярный стиль автора приоритетнее регулярного стиля пользователя. Тем не менее при добавлении в стили ключевого слова `!important` ситуация изменяется на противоположную:

```
p em {color: black !important;}  /* правило из таблицы стилей автора */
p em {color: yellow !important;}  /* правило из таблицы стилей пользователя */
```

Теперь текст с полужирным начертанием окрашивается желтым цветом, а не черным.

Последними в документе применяются стили пользовательского агента, заданные в нем по умолчанию. Они имеют наименьшую важность и назначаются элементам, для которых не определены даже пользовательские стили. Стили пользовательского агента заменяются любыми другими в первую очередь (например, стилями автора, устанавливающими форматирование элементов `a`, — для отображения текста гиперссылок белым цветом вместо синего, заданного по умолчанию).

Подытожив рассмотренные выше принципы, можно выделить пять уровней важности стилей, применяемых в документе. Ниже они перечислены в порядке от наиболее до наименее важного.

1. Стиль пользователя с добавлением ключевого слова `!important`.
2. Стиль автора с добавлением ключевого слова `!important`.
3. Стиль автора.
4. Стиль пользователя.
5. Стиль пользовательского агента.

Разработчикам стоит принимать в расчет все перечисленные выше уровни важности, кроме последнего, так как создаваемые ими стили будут иметь однозначный приоритет только над стилями пользовательского агента.

Определение значения приоритетности

В конфликтных ситуациях решение о том, какой из стилей, имеющих одинаковую важность и происхождение, применять к элементу, принимается пользовательским агентом, исходя из значения их приоритетности.

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

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

Применение указанных стилей к абзацу приводит к окрашиванию текста серебристым цветом, как показано на рис. 3.8. А все потому, что значение приоритетности у селектора `p#bright` (`0,1,0,1`) больше, чем у селектора `p` (`0,0,0,1`), даже несмотря на то что в таблице стилей он указывается первым.

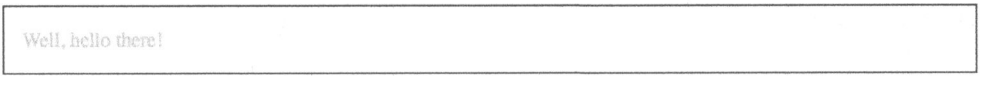


Рис. 3.8. К элементу применяется стиль с большим значением приоритетности

Определение порядка объявления стилей

На последнем этапе — при совпадении важности, происхождения и приоритетности правил, применяемых к элементу, — определяется порядок их объявления в таблице стилей. Вернемся к рассмотрению исходного примера, в котором цвет заголовка первого уровня определяется сразу двумя правилами.

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

В данном случае все элементы `h1` получают синий, а не красный цвет, поскольку при равенстве всех остальных критериев (важность, происхождение и приоритетность) предпочтение отдается правилу, добавленному в таблицу стилей последним.

А как поступает пользовательский агент при равенстве всех критериев, включая порядок указания правил в таблице стилей? Что если конфликтующие правила принадлежат к одной и той же внешней таблице стилей, как показано ниже?

```
@import url(basic.css);
h1 {color: blue;}
```

Легко заметить, что ситуация будет конфликтной только при включении правила `h1 {color: red;}` в таблицу стилей `basic.css`. В подобных случаях пользовательский агент рассматривает содержимое внешнего файла как вставленное в месте объявления (расположения команды `@import`). Исходя из этого, большей приоритетностью обладают любые правила, указанные после команды `@import`. Не стоит забывать, что речь идет о ситуациях совпадения у правил всех остальных критериев. Рассмотрим следующий пример.

```
p em {color: purple;} /* правило из внешней таблицы стилей */
p em {color: gray;}   /* правило, включенное в документ */
```

К элементу `p` применяется второе правило, как объявляемое в таблице стилей последним.

Порядок объявления правил важен при назначении стилей гиперссылкам, характеризующимся несколькими состояниями. Рекомендуется придерживаться следующего порядка стилизации состояний гиперссылок: LVFHA (Link-Visited-Focus-Hover-Active — гиперссылка, посещенная гиперссылка, в фокусе, наведен указатель, активная гиперссылка).

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

Как известно, у этих правил одинаковое значение приоритетности: `0,0,1,1`. Поскольку важность и происхождение у них тоже одинаковые, то к гиперссылке будет применяться последнее из правил, соответствующих текущему состоянию гиперссылки. В частности, активная гиперссылка (та, на которой щелкнули мышью или выбрали с помощью клавиатуры) описывается сразу четырьмя селекторами состояний: `:link`, `:focus`, `:hover` и `:active`. При расположении правил, задающих форматирование гиперссылок в каждом из состояний, в последовательности LVFHA активную гиперссылку будет представлять только последний селектор, как и предполагается в большинстве случаев.

Отказавшись от общепринятой формулы, правила форматирования состояний гиперссылок можно выстроить в алфавитном порядке.

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

Если придерживаться указанного порядка включения правил в таблицу стилей, то селекторы :hover, :focus и :active вполне очевидно останутся невостребованными, поскольку располагаются перед селекторами :link и :visited. Учитывая, что каждая гиперссылка находится либо в посещенном, либо в непосещенном состоянии, последние два правила всегда будут иметь более высокий приоритет, чем первых три.

Попробуем определить, какой еще порядок объявления правил стилевого оформления гиперссылок может заинтересовать разработчиков. Предложенный далее вариант предполагает, что наведение указателя мыши приводит к изменению внешнего вида только непосещенных гиперссылок. Посещенные гиперссылки при наведении на них указателя мыши сохраняют прежнее форматирование. Кроме того, в активное состояние переводятся как непосещенные, так и посещенные гиперссылки.

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

Учтите, что конфликтные ситуации возникают тогда, когда внешний вид гиперссылок во всех состояниях определяется общими свойствами. Если каждому из состояний гиперссылки соответствуют разные свойства, то порядок объявления стилевых правил не играет особой роли. Например, следующие правила можно объявлять произвольным образом, что никак не скажется на воспроизводимых ими эффектах.

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

Легко заметить, что порядок расположения селекторов :link и :visited друг относительно друга не принципиален. С точки зрения конечного результата формулы LVFHA и VLFHA абсолютно одинаковы.

Чтобы полностью избежать конфликтных ситуаций, в стилевых правилах нужно использовать селекторы псевдоклассов. Следующие объявления можно располагать в любом порядке, не беспокоясь о нарушении форматирования гиперссылок.

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

Так как правила задают форматирование для уникальных состояний гиперссылок, между собой они не конфликтуют. Таким образом, изменение порядка их расположения в таблице стилей не приводит к нарушениям в оформлении документов. Несмотря на то что последние два правила имеют одинаковую приоритетность, между собой они не пересекаются. И это понятно, поскольку гиперссылка не может одновременно находиться и в посещенном, и в непосещенном состоянии, а потому

форматирование в каждом из случаев задается отдельно. Если же селекторами псевдоклассов снабдить правила, определяющие форматирование активных состояний, то порядок их следования снова становится очень важным.

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

Правила для активных состояний, располагаемые перед правилами форматирования гиперссылок, на которые наведен указатель мыши, пользовательским агентом не рассматриваются. Причина конфликта — совпадение значений приоритетности у стилей. Чтобы предотвратить возникновение конфликтной ситуации, в селекторы игнорируемых правил нужно добавить уточняющие псевдоклассы.

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

Приоритетность обоих селекторов увеличивается до значения `0,0,3,1` — между собой они не конфликтуют, поскольку описывают взаимоисключающие состояния. Одновременно гиперссылка может представляться только одним из двух селекторов, а ее внешний вид — задаваться соответствующим правилом.

Нестилевое оформление документа

Оформление документа может выполняться без использования инструментов CSS — например, с помощью элемента `font`. Пользовательским агентом такое форматирование рассматривается как имеющее нулевую приоритетность, уступая пальму первенства стилям автора и пользователя, но только не стилям пользовательского агента. В CSS3 нестилевое форматирование вообще приравнивается к оформлению документа стилями пользовательского агента (предположительно добавленное в конец таблицы стилей, но в спецификации это в явном виде не оговаривается).

Резюме

Каскадирование является краеугольным камнем CSS и устанавливает принципы стилевого форматирования документа в любых конфликтных ситуациях. Оно основывается на сопоставлении значений приоритетности селекторов и механизме наследования стилевых правил.

Значения и единицы измерения

В этой главе рассматриваются ключевые компоненты языка CSS, которые определяют подавляющее большинство параметров стилевого оформления документа: значения, определяющие цвета, положение, размеры и многие другие характеристики элементов, а также единицы измерения, устанавливающие величины этих значений. Единицы измерения позволяют предельно точно указать параметры оформления элементов, например назначить отступ шириной 10 пикселей или представить заголовки шрифтом строго заданного размера. Ознакомившись с их областью применения и принципами объявления в таблицах стилей, вам будет намного проще изучать дальнейший материал.

Ключевые слова, строки и другие текстовые значения

Файлы таблиц стилей сохраняются в текстовом формате. Но наряду с исконно текстовыми значениями в таблицах стилей объявляются данные других типов, например числовые и именованные значения, а также адреса URL и, как ни странно, графические изображения.

Ключевые слова

Ключевые слова используются для представления значений свойств там, где это возможно. Самый простой пример ключевого слова — `none`, функционально отличимое от числового значения `0` (нуль). Например, для отмены подчеркивания гиперссылок в HTML-документе применяется такая команда:

```
a:link, a:visited {text-decoration: none;}
```

Подобным образом, для восстановления подчеркивания гиперссылок необходимо использовать ключевое слово `underline`.

Ключевое слово, используемое в качестве значения свойства, определяет тип действия только этого свойства. При назначении одного и того же ключевого слова разным свойствам они могут выполнять совершенно разные действия. Например, ключевое слово `normal` соответствует совершенно разным значениям в свойствах `letter-spacing` и `font-style`.

Глобальные ключевые слова

Спецификацией CSS3 определены три глобальных ключевых слова, допустимых для использования во всех без исключения свойствах: `inherit`, `initial` и `unset`.

Ключевое слово `inherit`

Это ключевое слово обязывает свойство наследовать значение у такого же свойства родительского элемента. Отметим, что значение наследуется даже тогда, когда его нельзя использовать в свойстве. В большинстве случаев свойства наследуются автоматически без применения ключевого слова `inherit`, но в отдельных ситуациях без него просто не обойтись.

Рассмотрим следующий фрагмент HTML-документа, форматирование которого задается всего одним правилом CSS.

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

<div id="toolbar">
<a href="one.html">One</a> | <a href="two.html">Two</a> |
<a href="three.html">Three</a>
</div>
```

В то время как элементу `div` назначен белый цвет переднего плана и синий фон, внешний вид гиперссылок определяется стилями браузера по умолчанию. С большой степенью вероятности гиперссылки, добавленные на панель навигации (`"toolbar"`), окрашиваются в синий цвет и отделяются друг от друга белыми вертикальными линиями.

Чтобы не потерять синие гиперссылки на синем фоне, можно изменить их цвет в правиле (например, на белый) или же поступить более дальновидно, применив ключевое слово `inherit`. В последнем случае стилевое правило принимает следующий вид:

```
#toolbar a {color: inherit;}
```

Теперь гиперссылки наследуют цвет у элемента `div`, а не заимствуют его из таблицы стилей пользовательского агента. Ключевое слово `inherit` применяется там, где нужно восстановить наследование форматирования, отменяемое прямым указанием значений свойств. Подобное действие не всегда желательно. В частности, в рассмотренном выше примере гиперссылки могут сливаться с окружающим их текстом и становиться неразличимыми. В каждом конкретном случае решение приходится принимать отдельно.

Подобным образом, ключевое слово `inherit` может использоваться для заимствования значения свойства, которое обычно не наследуется. Хорошим примером является свойство `border`, которое по умолчанию (вполне справедливо) не наследуется дочерними элементами. В отдельных ситуациях все же может потребоваться задать элементу `span` границу, наследуемую у его родительского элемента. Задача решается назначением ключевого слова `inherit` свойству `border`:

```
span {border: inherit;}
```

Для наследования только цвета границы применяется несколько иное правило:

```
{border-color: inherit;}
```

Ключевое слово `initial`

Ключевое слово `initial` "сбрасывает" свойство к начальному значению, устанавливаемому по умолчанию. Например, у свойства `font-weight` значение по умолчанию — `normal`. Таким образом, следующие два правила полностью идентичны.

```
font-weight: initial
font-weight: normal
```

Дублирование значений может показаться недоработкой технологии, но на самом деле ею не является, поскольку далеко не все свойства снабжены значениями по умолчанию. Во многих случаях значение по умолчанию для свойства определяется пользовательским агентом, а не CSS. Это означает, что начальное значение, например свойства `color`, зависит исключительно от настроек браузера. По умолчанию оно представляется черным цветом, который в любой момент может быть изменен в браузере пользователем (скажем, на серый или красный). Таким образом, объявление `color: initial` указывает окрашивать элемент цветом, задаваемым пользовательским агентом по умолчанию.

Ключевое слово `unset`

Ключевое слово `unset` применяется как универсальная замена ключевых слов `inherit` и `initial`. Для наследуемых свойств его действие равнозначно использованию ключевого слова `inherit`. Получая значение `unset`, свойства, не поддерживающие наследование, применяют к элементу форматирование, представляемое ключевым словом `initial`.

К концу 2017 года свойства `initial`, `inherit` и `unset` не поддерживались браузером Opera Mini. В Internet Explorer они не поддерживались никогда — даже в версии 11.

Выше описаны глобальные ключевые слова, передаваемые в качестве значений любым свойствам. При этом в CSS есть всего одно свойство, значение которого представляется *только* глобальными ключевыми словами.

all

Значение	`inherit` | `initial` | `unset`
Начальное значение	См. определения отдельных свойств

Универсальное свойство `all` представляет в объявлениях правил все свойства, за исключением `direction` и `unicode-bidi`. Таким образом, объявление `all: inherit`

указывает наследовать значения у своих родительских элементов всем свойствам, кроме `direction` и `unicode-bidi`. Рассмотрим такой пример.

```
section {color: white; background: black; font-weight: bold;}
#example {all: inherit;}

<section>
   <div id="example">This is a div.</div>
</section>
```

Начальный анализ показывает, что элемент `div` заимствует значения свойств `color`, `background` и `font-weight` у элемента `section`. Но не только их, поскольку наследованию также подлежат все остальные свойства (за исключением обозначенных выше двух) элемента `section`.

Если такое форматирование элемента `div` — это именно то, что требовалось получить, то можно довольствоваться предложенным выше вариантом кода. Если же требуется наследовать не все, а только указанные четыре свойства, правила нужно изменить так, как показано ниже.

```
section {color: white; background: black; font-weight: bold;}
#example {color: inherit; background: inherit; font-weight: inherit;}
```

Сложность в том, что набор свойств, наследуемый элементом при выполнении команды `all: unset`, разнится от одной таблицы стилей к другой.

В конце 2017 года в спецификацию CSS было предложено включить еще одно глобальное ключевое слово: `revert`. Его планируется снабдить функцией отключения свойств, принадлежащих одному из источников происхождения. Например, с его помощью можно будет отменять у элемента все стили автора, оставив в силе только стили пользователя и браузера. Поскольку работа над ключевым словом `revert` еще не завершена, в книге оно не описывается.

По состоянию на конец 2017 года браузеры Opera Mini и Microsoft Edge не поддерживали ключевое слово `all`. Работа над его включением в Microsoft Edge все еще продолжается.

Строки

Строковое значение — это набор символов, заключенных в одинарные или двойные кавычки и представленных выражением *<string>* в описании свойств. Ниже приведены два примера строковых значений.

```
"I like to play with strings."
'Strings are fun to play with.'
```

Обратите внимание на парность кавычек, которыми обозначаются текстовые строки, — в начале и конце строки должны использоваться кавычки одного типа. Использование кавычек разных типов в одном значении чревато ошибками

синтаксического анализа правил. В частности, если начать строку одинарной кавычкой, а закончить — двойной, то пользовательский агент не распознает ее и откажется обрабатывать. Это часто приводит к отмене правил, добавленных в таблицу стилей после некорректно заданного текстового значения.

В текстовое значение допускается включать кавычки, отличные от используемых для обозначения самой строки. В противном случае перед кавычками внутри строки нужно добавлять символ обратной косой черты.

```
"I've always liked to play with strings."
'He said to me, "I like to play with strings."'
"It's been said that \"haste makes waste.\""
'There\'s never been a "string theory" that I\'ve liked.'
```

Учтите, что для разграничения строковых значений допускается применять только прямые одинарные и двойные кавычки, но никак не "фигурные" или "наклонные". Наряду с этим кавычки последних двух типов можно включать в саму строку, как показано в следующем примере.

```
"It's been said that "haste makes waste.""
'There's never been a "string theory" that I've liked.'
```

Кавычки других типов представлены только в кодировке Unicode, по умолчанию применяемой в большинстве документов. (Описание стандарта Unicode приведено по адресу `http://www.unicode.org/standard/standard.html`.)

Для образования новой строки в конец строкового значения добавляется символ разрыва строки, не отображаемый пользовательским агентом при визуализации документа. С точки зрения CSS следующие две строки абсолютно одинаковы.

```
"This is the right place \
for a newline."
"This is the right place for a newline."
```

С другой стороны, если символ разрыва строки все же нужно отображать в строковом значении, то в место разрыва нужно вставить символ `\A` (в кодировке Unicode).

```
"This is a better place \Afor a newline."
```

Адреса URL

Занимаясь разработкой веб-страниц, вы, конечно же, знаете, что такое URL (или URI в терминологии CSS2.1). Команда добавления адресов URL в правила CSS имеет такой же формат, что и команда `@import`, используемая для подключения внешних таблиц стилей.

`url(`протокол`://`*сервер/путь*`)`

В приведенном выше описании указан *абсолютный* адрес URL. Абсолютным называется адрес, который устанавливает точное расположение ресурса (файла, веб-страницы) и не зависит от места указания. В качестве примера предположим, что URL должен указывать на файл `waffle22.gif`, находящийся в папке `pix` на

сервере web.waffles.org. Абсолютный адрес URL, по которому можно найти указанный графический файл, имеет такой вид:

```
web.waffles.org/pix/waffle22.gif
```

Данный URL будет указывать на файл waffle22.gif даже при добавлении его в документы, расположенные на других серверах (а не только на сервере web.waffles.org).

Кроме абсолютных, в правилах CSS также допускается применять относительные адреса URL — они определяют расположение ресурса относительно документа, в который добавлены. Расположение файла, хранящегося в той же папке (и на том же сервере), что и веб-страница, можно представить адресом URL следующего формата:

```
url(путь)
```

Не стоит забывать, что относительный адрес URL будет корректно обрабатываться только при размещении внешних файлов в той же папке, что и сам документ. В частности, для добавления изображения waffle22.gif на веб-страницу, расположенную по адресу http://web.waffles.org/syrup.html, можно использовать следующий URL:

```
pix/waffle22.gif
```

Корректность обработки относительного адреса обусловливается тем, что браузер начинает поиск внешних файлов с места расположения самого документа. В последнем случае путь pix/waffle22.gif указан для сервера http://web.waffles.org, а потому его полный (абсолютный) адрес имеет такой вид:

```
http://web.waffles.org/pix/waffle22.gif
```

В большинстве случаев можно отказаться от использования относительных адресов в пользу абсолютных, хотя большого различия между ними нет — главное, чтобы целевой файл находился по указанному расположению.

В CSS адрес URL указывается относительно места расположения таблицы стилей, а не HTML-документа, к которому она применяется. Например, в ситуации подключения к документу внешней таблицы стилей, в которую импортируется еще одна таблица стилей, URL-адрес второй таблицы стилей указывается относительно первой, но не самой веб-страницы.

Рассмотрим, как правильно связать таблицу стилей, расположенную по адресу http://web.waffles.org/styles/basic.css, с документом, имеющим следующий адрес:

```
http://web.waffles.org/toppings/tips.html
```

В случае абсолютной адресации соответствующий тег HTML имеет такой вид:

```
<link rel="stylesheet" type="text/css"
      href="http://web.waffles.org/styles/basic.css">
```

Предположим, что в указанной таблице стилей с помощью команды @import подключается еще одна таблица стилей:

```
@import url(special/toppings.css);
```

Так как в команде `@import` указан относительный URL, то браузер начнет поиск второй таблицы стилей по такому адресу:

```
http://web.waffles.org/styles/special/toppings.css
```

Место расположения документа при этом полностью игнорируется:

```
http://web.waffles.org/toppings/special/tips.html
```

Если таблица стилей хранится по второму адресу, то для корректного ее подключения в команде `@import` нужно указать один из таких адресов (абсолютный или относительный).

```
@import url(http://web.waffles.org/toppings/special/toppings.css);
@import url(../special/toppings.css);
```

Не забывайте, что между `url` и открывающей скобкой не должно содержаться пробелов.

```
body {background: url(http://www.pix.web/picture1.jpg);} /* неправильно */
body {background: url (images/picture2.jpg);}            /* ПРАВИЛЬНО */
```

Команды `url`, содержащие пробел перед открывающей скобкой, считаются недействительными и игнорируются браузером.

Изображения

Значение изображения, как предполагает название, представляет графическое изображение, вставленное в документ. В описании правил значение изображения представляется выражением *<image>*.

Все без исключения пользовательские агенты распознают значения изображений, представленные адресом URL. Современные браузеры поддерживают следующие способы обозначения значений *<image>* в таблицах стилей.

<url>

Адрес URL, по которому располагается внешний ресурс, — в данном случае графическое изображение.

<image-set>

Вполне ожидаемо представляет набор изображений, из которого, согласно добавленному в значение условию, выбирается только один вариант. Например, значение `image-set()` предполагает использование наибольшего по размеру (как геометрическому, так и размеру файла) изображения при выводе документа на мониторах с высоким разрешением. При этом изображения меньшего размера применяются при показе веб-страницы на экранах мобильных устройств. Функционально значение *<image-set>* применяется в тех же целях, что и атрибут `srcset` элемента `picture`. К сожалению, к концу 2016 года оно, как и обозначенный атрибут `srcset`, поддерживалось только браузерами Safari, Chrome и Opera (для настольных систем).

<gradient>

Графическое изображение линейного или радиального градиента — отдельное или повторяющееся. Управление градиентами представляет собой сложную задачу, детально рассмотренную в главе 9.

Идентификаторы

Отдельным свойствам допускается передавать *значения идентификаторов*, устанавливаемых пользователем. Чаще всего пользовательские идентификаторы представляют счетчики списков. В описании свойств они указываются значением *<identifier>*. Синтаксически идентификатор представляется словом, символы которого чувствительны к регистру. Таким образом, с точки зрения синтаксического анализатора идентификаторы `myID` и `MyID` являются абсолютно разными значениями. Во избежание конфликтов лучше не использовать в качестве идентификаторов зарезервированные ключевые слова — по крайней мере, в свойствах, в которых кроме идентификаторов указываются другие значения.

Числовые и процентные значения

Числа и процентные значения выделены в отдельную категорию, поскольку являются неотъемлемыми компонентами огромного количества значений. Например, размер шрифта указывается в виде числа и единиц измерения (рассматриваются в следующем разделе). Но все ли свойства принимают числовые значения в одном и том же формате? В следующих разделах речь пойдет о числовых типах данных.

Целочисленные значения

Целочисленные значения самые простые. Они состоят из одной или нескольких цифр, предваряемых знаком – ("минус") или + ("плюс"), обозначающих положительные и отрицательные значения. В описании свойств целые числа представляются выражением *<integer>*. К целочисленным значениям относятся, например, `13`, `-42`, `712` и `1 066`.

Если целое число не попадает в заранее определенный для свойства диапазон, то правило считается недействительным. Как бы там ни было, при передаче такого значения некоторые свойства принимают допустимое целочисленное значение, наиболее близкое к указанному.

Числа

К числовым значениям относятся рассмотренные выше значения *<integer>* и действительные числа. Действительное число состоит из целого числа, после которого через точку указывается дробная часть. Перед отрицательными действительными числами ставится знак "минус", а перед положительными действительными числами — знак "плюс". В описании свойств действительные числа представляются выражением *<number>*. К действительным числам относятся такие значения, как `2.7183`, `-3.1416` и `6.2832`.

Поддержка сразу двух типов числовых значений — *<number>* и *<integer>* — вызвана тем фактом, что некоторые свойства (например, `z-index`) имеют только целочисленные значения, в то время как многие другие (такие, как `flex-grow`) допускают использование любых действительных чисел. Как и в предыдущем случае, отдельные свойства принимают действительные числа только из заранее заданного диапазона. Например, свойству `opacity` назначаются значения типа *<number>* из диапазона от `0` до `1` включительно. По умолчанию передача свойству числа, не попадающего в допустимый диапазон, приводит к игнорированию правила пользовательским агентом. Тем не менее при передаче такого значения некоторым другим свойствам они принимают допустимое действительное значение, наиболее близкое к указанному.

Процентные значения

Процентное значение представляет собой действительное число, в конец которого добавлен символ `%`. В описании свойств такое значение представляется выражением *<percentage>*. К процентным относятся такие значения, как `50%` и `33.333%`. Процентное значение всегда указывается относительно некоего другого значения, например, наследуемого от родительского элемента, переданного потомку или заданного другому свойству текущего элемента. Для каждого свойства определен собственный диапазон допустимых процентных значений, а также правила их вычисления.

Дробные значения

Дробное значение состоит из действительного числа и суффикса `fr`. Таким образом, одна дробная единица представляется значением `1fr`. Концепция дробных единиц впервые была представлена в модуле CSS Grid Layout, в котором их предлагается использовать для описания дольных частей свободного пространства макета (глава 13).

Длина и расстояние

Значения многих свойств, таких, как, например `margins`, определяют размер элемента и положение его в документе. Не секрет, что длина и расстояние измеряются разными способами, в том числе и в CSS.

В значении свойств размер и расстояние задаются положительным или отрицательным числом, после которого указываются единицы измерения. Обратите внимание на то, что некоторые свойства поддерживают только положительные значения длины. Чаще всего применяются действительные числа, имеющие целую и дробную части, например `10.5` или `4.561`. Названия единиц измерения, указываемых после числового значения, приводятся в сокращенном виде (обычно представляются двумя символами) — например, `in` (inches — дюймы) или `pt` (points — точки). Число `0` (нуль) является исключением из правил, поскольку единицы измерения после него не указываются.

Значения длины (размеров и расстояния) бывают двух основных типов: *абсолютные* и *относительные*.

Абсолютные единицы измерения длины

Начать знакомство проще всего с абсолютных единиц измерения, как более простых для понимания, хотя они редко применяются при создании веб-страниц. В стилевых правилах разрешается использовать следующие шесть абсолютных единиц измерения длины.

Дюймы (`in`)

Все еще используются в качестве основных единиц измерения в некоторых странах, в частности, в США. (Включение неметрических единиц измерения длины в спецификацию CSS вызвано высокой концентрацией IT-компаний в США и никак не связано с противостоянием общепринятым стандартам.)

Сантиметры (`cm`)

Соответствует делениям на строительной рулетке или линейке в большинстве стран мира. В одном сантиметре 0,394 дюйма, в дюйме — 2,54 см.

Миллиметры (`mm`)

В метрической системе миллиметр составляет десятую часть сантиметра, поэтому в дюйме 25,4 миллиметра, а в миллиметре — 0,0394 дюйма.

Четверть миллиметра (`q`)

В сантиметре 40 четвертей миллиметров. Таким образом, ширина в `4q` соответствует одному миллиметру, или десятой части сантиметра. К концу 2016 года эти единицы измерения поддерживал только Firefox.

Пункты (`pt`)

Пункты, или точки, являются стандартными типографскими единицами измерения, используемыми при выводе документов на печатных станках и домашних принтерах. Они также поддерживаются большинством современных текстовых процессоров. В одном дюйме 72 точки (обе единицы измерения использовались в книгопечатании задолго до распространения метрической системы). Таким образом, высота прописной буквы размером `12pt` равна шестой части дюйма, соответственно следующие два правила абсолютно одинаковы.

```
p {font-size: 18pt;}
p {font-size: 0.25in;}
```

Цицеро, или пики (`pc`)

Также относятся к типографским единицам измерения. В одном цицеро 12 точек, а в дюйме — 6 цицеро. В предыдущем примере с прописной буквой ее размер будет равен 1 цицеро, или шестой части дюйма. А пример правила, устанавливающего форматирование абзацев, можно переписать следующим образом:

```
p {font-size: 1.5pc;}
```

Пиксели (px)

Пиксели — это наименьшие точки на экране, из которых формируется изображение, но в CSS они трактуются несколько по-другому. Спецификация CSS определяет размер пикселя как экранной точки при строго фиксированном разрешении — 96 пикселей на дюйм. При вычислении размера пикселя многие браузеры игнорируют данное требование, ориентируясь на разрешение экрана, установленное пользователем. Таким образом, размер пикселя будет изменяться при масштабировании документа и выводе его на печать (100 экранных пикселей далеко не всегда будут соответствовать 100 печатным точкам).

Абсолютные единицы измерения можно использовать в стилевых правилах только при отображении документов на экранах со строго заданным разрешением или выводе на бумагу на принтерах с заранее известными настройками печати. При отображении в окне браузера документ подстраивается под размер и разрешения экрана целевого устройства, что неизбежно приводит к искажению размерных величин. С этим ничего не поделаешь — остается только надеяться, что элементы документа сохранят свои относительные размеры, и в лучшем случае `0.5in` по-прежнему будет вдвое меньше `1.0in` (рис. 4.1).

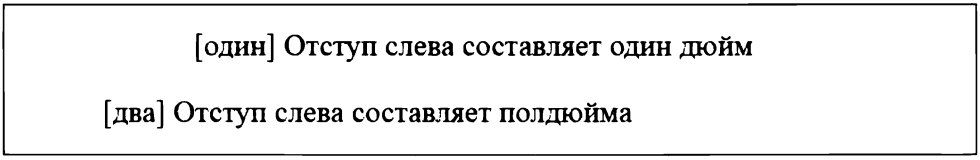


Рис. 4.1. Установка отступов в абсолютных единицах измерения

Несмотря на указанные ограничения, представим, что мы живем в идеальном мире и большинство компьютерных устройств умеет в точности воспроизводить единицы измерения из реального мира. Для того чтобы задать абзацам документа полудюймовое верхнее поле, достаточно использовать правило `p {margin-top: 0.5in;}`. В данном случае размер поля будет сохраняться постоянным независимо от размера шрифта и настроек видеосистемы.

Стоит заметить, что абсолютные единицы измерения длины лучше всего подходят для форматирования документов, выводимых на печать, а не на экран. В печатном деле дюймы, цицеро и пункты имеют большее распространение, чем в экранных презентациях.

Размер пикселей

При выводе документов на экран размеры элементов проще всего задавать в пикселях. Внимательно присмотревшись к изображению на экране, легко заметить, что оно состоит из большого количества цветных точек. Каждая такая точка и есть пиксель. Задав высоту и ширину элемента в пикселях, как в следующем примере, можно с уверенностью утверждать, что он будет представляться ровно таким количеством экранных точек, как указано в правиле (рис. 4.2).

```
<p>
Изображение, добавленное в конец строки, имеет ширину и высоту
   20 пикселей: <img src="test.gif" style="width: 20px;
   height: 20px;" alt="" />
</p>
```

Изображение, добавленное в конец строки, имеет ширину и высоту 20 пикселей:

Рис. 4.2. Элемент, размеры которого установлены в пикселях

Как правило, браузеры, обрабатывая объявления типа `font-size: 18px`, отображают элементы на экране в строго предписанном размере. Если же документ выводится на печать или другие типы носителей, то пользовательский агент, скорее всего, изменит размер пикселей. Иными словами, каждый экранный пиксель будет представляться на бумаге несколькими цветовыми точками.

Как бы там ни было, пиксели как единицы измерения удобно использовать для указания ширины и высоты графических изображений. При этом современная адаптивная верстка предполагает выражение ширины и высоты изображений через размер шрифта, который в свою очередь определяется разрешением экрана целевого устройства, независимо от того, из какого количества пикселей состоит это изображение. Все-таки не стоит полностью полагаться на пользовательские агенты, снабженные функцией автоматического масштабирования изображений, хотя большинство из них прекрасно справляется с этой задачей. Изменение размера без потери качества свойственно только векторным графическим файлам, в веб-дизайне обычно представляемым форматом SVG.

Плотность пикселей

Приведенные выше рассуждения справедливы для стандартного разрешения 96 ppi (pixels per inch — пикселей на дюйм). Если разрешение устройства вывода отличается от указанного значения, то, согласно требованиям спецификации CSS, пользовательским агентам предписывается придерживаться некоего индикативного разрешения. В CSS2 под индикативным подразумевается разрешение 90 ppi, но уже в CSS2.1 и CSS3 оно увеличено до 96 ppi. Стоит заметить, что большинство современных принтеров печатает документы с большей плотностью точек на дюйм, чем предполагает индикативное разрешение, поэтому перед выводом веб-страницы на бумагу ее нужно масштабировать до соответствующего разрешения.

При плотности 96 ppi стандартный экран с разрешением 1024×786, полностью заполненный пикселями, будет иметь ширину 10 2/3 дюйма и высоту — ровно 8 дюймов (271×203 мм). Легко заметить, что такие экраны в современной компьютерной технике и мобильных устройствах встречаются не так уж и часто. В действительности плотность пикселей у большинства из них намного выше, чем 96 ppi. В частности, экран iPhone 4S имеет разрешение 326 ppi, а у экрана iPad оно несколько меньше — 264 ppi.

Пользователи Windows XP имели возможность отображать текстовые элементы в реальных размерах. Для этого в папке Панель управления нужно было выбрать приложение Экран и в появившемся на экране диалоговом окне перейти на вкладку Параметры, после чего в нижней части окна щелкнуть на кнопке Дополнительно. В раскрывающемся списке Масштаб нового диалогового окна нужно выбрать вариант Особые параметры.

Единицы измерения разрешения

С появлением медиа-запросов и адаптивного дизайна разрешение экрана стало описываться тремя новыми единицами измерения.

`dpi` (dots per inch — количество точек на дюйм)

Количество экранных точек на линейный дюйм изображения. Представляет плотность точек при выводе документа на печать, отображении на светодиодных экранах, электронной бумаге и других устройствах.

`dpcm` (dots per centimeter — количество точек на сантиметр)

То же, что и `dpi`, но количество точек указывается на линейный сантиметр, а не на дюйм.

`dppx` (dots per pixel unit — количество точек на пиксель)

Количество экранных точек на пиксель согласно спецификации CSS. В CSS3 значение `1dppx` соответствует разрешению 96 dpi. Учтите, что в будущих спецификациях CSS оно может быть другим.

К концу 2017 года приведенные выше единицы измерения рассматривались только в контексте медиа-запросов. Например, в таблицу стилей можно включить блок правил, выполняющихся только при отображении документа на экранах с разрешением больше 500 dpi.

```
@media (min-resolution: 500dpi) {
   /* объявления правил */
}
```

Относительные единицы измерения длины

Эти единицы измерения указывают размерность относительно другого значения. Представляемое ими действительное (абсолютное) значение размера элементов подвержено влиянию большого количества независимых факторов, таких как разрешение экрана, ширина окна просмотра, параметры браузера и т.п. К тому же числовое значение, выраженное в относительных единицах измерения длины, зависит от размера некоего элемента и изменяется вместе с ним.

Единицы измерения em и ex

Вначале познакомимся с единицами измерения `em` и `ex`, имеющими общее происхождение. В CSS единица `em` равна значению свойства `font-size` заданного шрифта.

Если свойство font-size элемента установлено в значение 14 пикселей, то 1em также равняется 14 пикселям.

Несложно догадаться, что размер единицы em в каждом из элементов разный. Для иллюстрации этого утверждения рассмотрим документ, в котором заголовки первого уровня (элемент h1) представлены шрифтом размером 24 пикселя, заголовки второго уровня (элемент h2) — шрифтом размером 18 пикселей, а размер шрифта регулярных абзацев составляет 12 пикселей. Если всем им установить левое поле шириной 1em, то у каждого типа элементов оно будет иметь свой размер — соответственно 24, 18 и 12 пикселей.

```
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>
```

С другой стороны, значение, выраженное в единицах em, всегда пропорционально размеру шрифта родительского элемента, что проиллюстрировано на рис. 4.3.

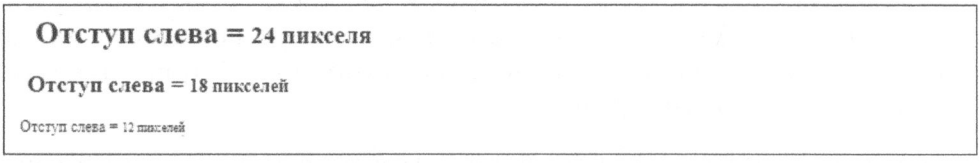


Рис. 4.3. Зависимость ширины левого поля, выраженного в единицах em, от размера шрифта

Исходно единицы измерения em были сугубо типографским термином, обозначавшим ширину строчной буквы "m" в заданном шрифте, откуда и пошло название. Тем не менее они успешно прижились в веб-дизайне.

Единицы измерения ex, как и em, заимствованы из печатного дела. Единица ex равна высоте строчной буквы "x" в заданном шрифте, что также отражается в ее названии. Тем не менее у двух абзацев с разными шрифтами одинакового размера единица измерения ex будет разной. А все потому, что высота буквы "x" у всех шрифтов разная (рис. 4.4). На рисунке хорошо заметно, что высота буквы "x" в каждой строке разная, несмотря на одинаковый размер шрифтов, устанавливаемый в единицах измерения em (24 пикселя).

Единицы измерения rem

Единицы измерения rem, как и em, зависят от размера шрифта элемента, но с небольшой оговоркой. Если единица измерения em определяется размером шрифта, назначенного элементу, в котором она используется, то rem *всегда* задается размером шрифта корневого элемента (html в документах HTML). Таким образом, для

назначения элементу шрифта с таким же размером, как у корневого элемента, к нему достаточно применить свойство `font-size: 1rem`.

Рис. 4.4. У всех шрифтов разная высота букв "x"

Для примера рассмотрим следующий фрагмент HTML-документа. Результат его отображения в браузере показан на рис. 4.5.

```
<p>Этот абзац наследует размер шрифта у корневого элемента.</p>
<div style="font-size: 30px; background: silver;">
   <p style="font-size: 1em;">Этот абзац представляется шрифтом
      такого же размера, как и родительский элемент.</p>

   <p style="font-size: 1rem;">Этот абзац наследует размер шрифта
      у корневого элемента.</p>
</div>
```

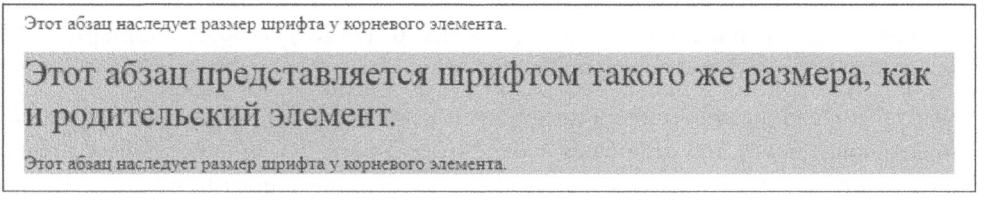


Рис. 4.5. Результат использования единиц измерения `em` и `rem` в одном документе

Единицы измерения `rem` удобно применять для "сброса" размера шрифта до значения по умолчанию. Независимо от того, каким образом изменялся размер шрифта у элемента, применение к нему свойства `font-size: 1rem` позволяет восстановить его до значения, заданного для корневого элемента. По умолчанию размер шрифта корневого элемента соответствует значению, заданному в настройках браузера.

Например, приведенное ниже правило устанавливает `1rem` равным `13px`.

```
html {font-size: 13px;}
```

В результате применения следующего правила единица `1rem` уменьшается до значения, составляющего 3/4 размера шрифта, заданного в настройках браузера:

```
html {font-size: 75%;}
```

В последнем случае, если шрифт по умолчанию имеет размер `16px`, то `1rem` будет равняться `12px`. Изменение размера шрифта по умолчанию в браузере до значения `12px` (весьма распространенная операция) приведет к уменьшению `1rem` до `9px`. Подобным образом увеличение размера шрифта по умолчанию до `20px` вызывает возрастание `1rem` до значения `15px` и т.д.

Не стоит привязываться к одному только значению `1rem`. В единицах измерения `rem` можно задавать произвольные числовые значения. В качестве закрепления полученных навыков давайте сделаем размер шрифта заголовков документа кратным размеру шрифта корневого элемента.

```
h1 {font-size: 2rem;}
h2 {font-size: 1.75rem;}
h3 {font-size: 1.4rem;}
h4 {font-size: 1.1rem;}
h5 {font-size: 1rem;}
h6 {font-size: 0.8rem;}
```

В браузерах, поддерживающих ключевое слово `initial`, объявления `font-size: 1rem` и `font-size: initial` будут полностью идентичны до тех пор, пока размер шрифта корневого элемента представляется значением по умолчанию.

Единицы измерения ch

В CSS3 добавлена новая единица измерения, название которой походит от выражения "one character" (один символ). В спецификации она описывается следующим образом.

> Единица `ch` равна полному размеру знака "0" (ZERO, U+0030) в шрифте, используемом для его визуализации.

Фраза "полный размер", хотя и представляет собой профессиональное выражение, использована вместо термина "шаг" совсем неспроста. Во многих языках направление письма, а потому и шаг знака, отличается от привычного нам (слева направо или справа налево). В системах письма "сверху-вниз" или "снизу-вверх" шаг знака

определяется весьма неоднозначно — в подобных случаях лучше использовать компромиссный термин "размер", который в европейских языках носит обобщающий характер. Для простоты в дальнейшем будем использовать термин "шаг", но под шагом знака будет подразумеваться расстояние от одного знака до следующего.

В самом простом случае шаг знака определяется как расстояние от начала текущего знака до начала соседнего с ним знака. Условно шаг знака равняется ширине знака и некоего расстояния по обеим его сторонам (представляемого как положительным, так и отрицательным значением).

В CSS единица `ch` определяется как шаг знака `0` (нуль) для заданного шрифта. Здесь используется такой же подход, как и в случае единицы измерения `em`, представляющей размер шрифта, указанного свойством `font-size`.

Самый простой способ знакомства с этой единицей измерения заключается в выводе на экран текстовой строки, состоящей из одних только нулей, и сопоставимого с ней по размеру изображения, ширина которого указывается в единицах `ch` (рис. 4.6).

```
img {height: 1em; width: 25ch;}
```

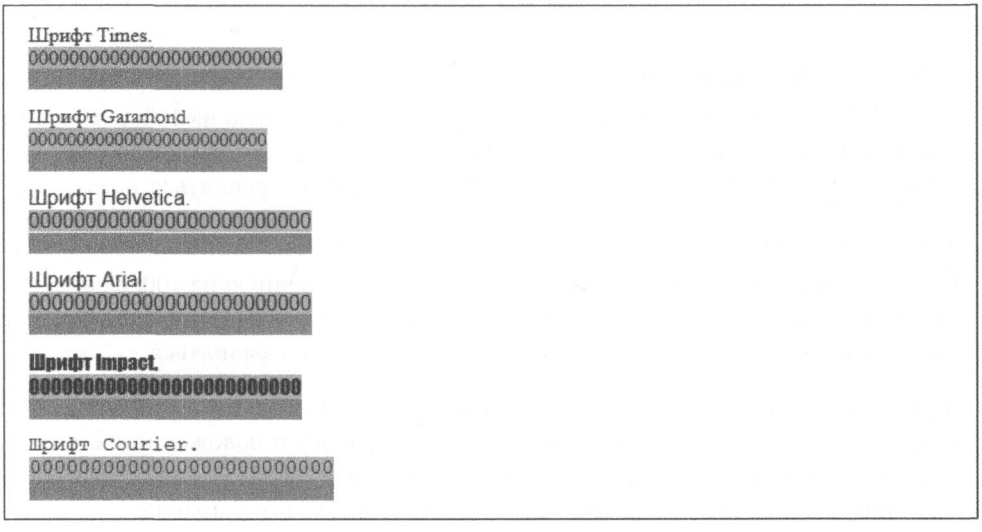


Рис. 4.6. Установка размеров относительно шага символа

В моноширинных шрифтах все символы имеют шаг `1ch`. Пропорциональные шрифты, в частности, представленные семейством Western, состоят из символов разной ширины, а потому их шаг может быть как больше, так и меньше `1ch`.

К концу 2017 года единицы измерения `ch` некорректно обрабатывались в Opera Mini и Internet Explorer. В IE11 единица `ch` равнялась не шагу символа, а его ширине (без учета свободного пространства по бокам). Таким образом, в его окне строка "00000" длиннее `5ch`. Эта ошибка успешно устранена в Microsoft Edge.

Единицы измерения, зависящие от ширины окна просмотра

В CSS3 добавлено несколько новых относительных единиц измерения, зависящих от размера окна просмотра пользовательского агента. Размер окна просмотра определяется шириной окна браузера, экрана мобильного устройства, областью печати и т.п.

`vm` (viewport width unit — единицы ширины окна просмотра)

Представляет ширину окна просмотра, деленную на 100. Таким образом, если окно браузера имеет ширину 937 пикселей, то `1vw` будет равняться `9,37px`. При изменении размера окна браузера (растягивании его границ с помощью мыши) его ширина в единицах `vw`, представляющая другое количество пикселей, останется прежней.

`vh` (viewport height unit — единицы высоты окна просмотра)

Представляет высоту окна просмотра, деленную на 100. Таким образом, если окно браузера имеет высоту 650 пикселей, то `1vw` будет равняться `6,5px`. При изменении размера окна браузера (растягивании его границ с помощью мыши) его высота в единицах `vh`, представляющая другое количество пикселей, останется прежней.

`vmin` (viewport minimum unit — единицы минимального размера окна просмотра)

Представляет ширину или высоту окна просмотра, деленную на 100, в зависимости от того, какая из размерностей *меньше*. Таким образом, если окно браузера имеет ширину 937 и высоту 650 пикселей, то `1vmin` будет равняться `6,5px`.

`vmax` (viewport maximum unit — единицы максимального размера окна просмотра)

Представляет ширину или высоту окна просмотра, деленную на 100, в зависимости от того, какая из размерностей *больше*. Таким образом, если окно браузера имеет ширину 937 и высоту 650 пикселей, то `1vmax` будет равняться `9,37px`.

Данные единицы измерения, как и описанные в предыдущих разделах, могут применяться в любых свойствах, устанавливающих размер и положение элементов в макете документа. Например, для задания размера шрифта заголовка относительно высоты окна браузера можно использовать следующее стилевое правило:

```
h1 {font-size: 10vh;}
```

Оно назначает размер шрифта, равный 1/10 высоты окна просмотра. Такой подход часто применяется при форматировании заголовков в лентах новостей.

Единицы измерения, заданные относительно размеров окна просмотра, наиболее востребованы при разработке полноэкранных интерфейсов, открываемых на мобильных устройствах, в которых размер элементов определяется разрешением экрана, а не положением их в объектной модели документа. Задавая размеры элементов в таких единицах измерения, можно не беспокоиться о загромождении области просмотра неправильно масштабированными объектами, что вызвано изменением разрешения экрана целевого устройства.

Пример масштабирования элементов, размер которых указывается в единицах измерения, зависящих от разрешения экрана, приведен на рис. 4.7.

```
div {width: 50vh; height: 33vw; background: gray;}
```

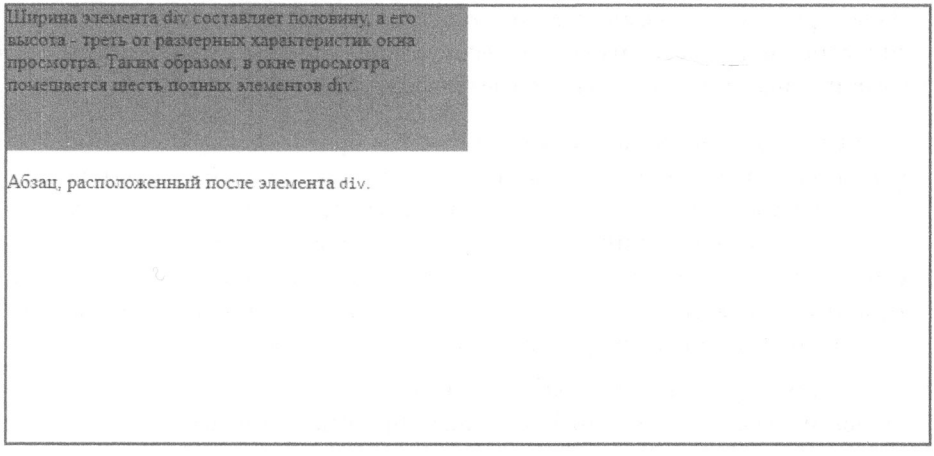


Рис. 4.7. Масштабирование элементов относительно размеров окна браузера

К концу 2016 года единицы измерения, зависящие от размеров окна просмотра, поддерживались всеми браузерами, кроме Opera Mini. Как ни странно, браузеры компании Microsoft не поддерживают всего одну из описанных выше единиц измерения: `vmax`.

Вычисляемые значения

В CSS вычисляемое значение представляется функцией `calc()`, в скобки которой подставляются простые математические выражения, требующие расчета. При построении математических выражений допускается использовать знаки следующих операций: `+` (сложение), `-` (вычитание), `*` (умножение) и `/` (деление), а также круглые скобки. Вычисления проводятся в стандартном порядке: скобки, умножение, деление, сложение и вычитание (операция возведения в степень исключена из списка, поскольку в функции `calc()` она недопустима).

Возможность использования скобок при получении вычисляемых значений предоставляется браузерами, а не спецификацией CSS, в синтаксисе функции `calc()` которой они не предусмотрены. Скорее всего, скобки будут поддерживаться и в будущих версиях браузеров, но при их использовании будьте предельно внимательны.

Рассмотрим пример, в котором инструментами CSS нужно задать ширину элемента так, чтобы она составляла 90% от ширины родительского элемента за вычетом `2em`. С помощью функции `calc()` эта задача решается следующим образом:

```
p {width: calc(90% - 2em);}
```

Функцию `calc()` разрешается использовать в следующих типах значений: *<length>*, *<frequency>*, *<angle>*, *<time>*, *<percentage>*, *<number>* и *<integer>*. Более того, значения указанных типов, хотя и с незначительными ограничениями, можно использовать в математических выражениях, подставляемых в `calc()`.

Первое ограничение связано с автоматической проверкой типов данных, над которыми выполняются математические вычисления. В функции `calc()` проверка обрабатываемых значений выполняется следующим образом.

1. Слагаемые и вычитаемые величины должны представляться в единицах измерения одного типа или быть значениями *<number>* и *<integer>* (в последнем случае результат будет *<number>*). Например, операция `5 + 2.7` будет давать результат `7.7`. Операция `5em + 2.7` рассматривается как недопустимая, поскольку только одно из значений выражается в единицах измерения длины. При этом операция `5em + 20px` действительна, так как в обоих значениях используются единицы измерения (`em` и `px`) одного типа.
2. В операциях умножения одно из значений должно быть *<number>* (включает подмножество целых чисел). Таким образом, оба выражения, `2.5rem * 2` и `2 * 2.5rem`, считаются справедливыми и возвращают результат `5rem`. В противоположность им выражение `2.5rem * 2rem` недействительно, поскольку в результате его вычисления будет получено значение $5rem^2$, представляющее не линейный размер, а площадь.
3. В операциях деления делителем может выступать только значение *<number>*. В противном случае результат будет представляться обратными единицами измерения, что недопустимо. Например, выражение `30em / 2.75` будет обрабатываться, а выражение `30 / 2.75em` — нет.
4. Недопустимыми считаются любые операции, приводящие к делению на нуль. Чтобы добиться такого результата, достаточно подставить выражение `30px/0` в конструкцию `calc()`, хотя существуют и более запутанные способы его получения.

Еще одно ограничение не связано со значениями, принимающими участие в вычислениях, а носит сугубо синтаксический характер — математические операторы должны выделяться символами пробелов с обеих сторон. Подобная практика позволяет отделить знаки математических операций от символов, обозначающих положительные и отрицательные значения.

Наряду с вышесказанным спецификация CSS предписывает браузерам обеспечить обработку математических выражений, состоящих *не менее* чем из 20 членов, к которым относятся значения, единицы измерения (размеров) и знаки процентов. Если выражение состоит из большего количества членов, чем может обработать браузер, то оно считается недействительным.

Значения атрибутов

В отдельные стилевые свойства можно подставлять значения атрибутов HTML-элементов, форматирование которых они определяют. Для этих целей применяется функция `attr()`.

Например, генерируемое содержимое можно создавать с использованием значений любых атрибутов, как показано ниже (синтаксис правил такого типа детально рассмотрен в главе 15).

```
p::before {content: "[" attr(id) "]";}
```

Правило добавляет значение атрибута `id`, заключенное в квадратные скобки, в начало каждого абзаца, содержащего его. Результат применения правила к элементам, представленным следующим фрагментом HTML-кода, показан на рис. 4.8.

```
<p id="Начало">Это первый абзац.</p>
<p>Это второй абзац.</p>
<p id="Завершение">Это третий абзац.</p>
```

[Начало]Это первый абзац.

[]Это второй абзац.

[Завершение]Это третий абзац.

Рис. 4.8. Добавление значений атрибутов в начало абзацев

Теоретически функцию `attr()` можно использовать в значении любого свойства. В следующем примере она применяется для определения ширины поля ввода данных, устанавливаемой в атрибуте `maxlength` его элемента.

```
input[type="text"] {width: attr(maxlength em);}

<input type="text" maxlength="10">
```

При выполнении приведенного выше правила элемент `input` получает ширину `10em`. К сожалению, к концу 2016 года ни один из опробованных мною браузеров не поддерживал такой синтаксис функции `attr()`.

Цвета

Большинство начинающих веб-дизайнеров в первую очередь интересуются методами изменения принятых в документе цветовых решений. В HTML цвет определяется двумя способами: с помощью короткого названия или кодированного шестнадцатеричного значения. В CSS поддерживаются оба этих способа, но предлагаются и свои — как мне кажется, более интуитивно понятные.

Именованные цвета

Внешний вид документов, дизайн которых основан на небольшом количестве цветовых оттенков, проще всего определять стилевыми свойствами, которым передаются названия цветов. В CSS цвета, представляемые отдельными названиями, называются именованными. Самая первая спецификация CSS включала всего 16 ключевых слов, описывающих именованные цвета (сопоставимые с именованными цветами в HTML 4.01). Все они приведены в табл. 4.1.

Таблица 4.1. Базовые именованные цвета в CSS

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

Например, чтобы назначить темно-бордовый цвет (maroon) всем заголовкам первого уровня, лучше всего использовать следующее правило:

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

Очень просто, не правда ли? На рис. 4.9 показано еще несколько цветовых решений.

```
h1 {color: silver;}
h2 {color: fuchsia;}
h3 {color: navy;}
```

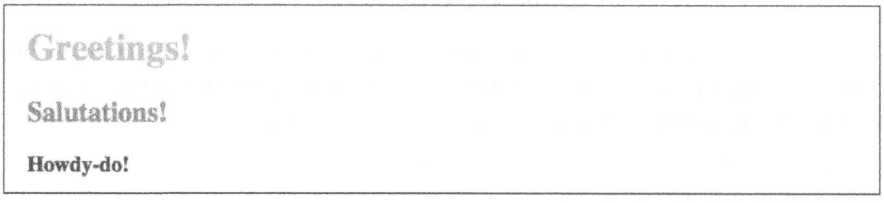


Рис. 4.9. Именованные цвета в действии (см. цветные иллюстрации на веб-сайте)

Однако существует намного больше именованных цветов, чем приведено в табл. 4.1. Например, в стилевых правилах можно использовать следующее название:

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

К концу 2017 года в спецификации CSS было определено 148 именованных цветов, включая 16, обозначенных выше. Большинство из них взято из схемы X11 для системы RGB, состав которой "оттачивался" в браузерах на протяжении десятилетий, а остальные заимствованы из стандарта SVG (по большому счету, для обеспечения корректной цветопередачи серых оттенков). Таблица соответствия всех именованных цветов, определенных спецификацией CSS Color Module 4, значениям, принятым в других системах цветности, приведена в приложении B.

К счастью, в CSS существует более точный способ указания цветов. В нем цветовой оттенок выбирается из непрерывного спектра, а не привязывается к перечню дискретных именованных констант.

Цветовые модели RGB и RGBa

Произвольный цветовой оттенок, отображаемый на экранах компьютерных устройств, образуется в результате смешения трех основных компонентов — красного (red), зеленого (green) и синего (blue) — и поэтому называется *RGB-цветом*. Точки на экране, из которых формируется изображение, называются *пикселями*. Зная, как устанавливается цвет каждого пикселя, его оттенок можно задавать, комбинируя три основных компонента в разных пропорциях. Решение далеко не такое простое, как кажется на первый взгляд, поскольку количество получаемых таким способом цветовых оттенков несколько ограничено. Далее рассмотрены четыре основные модели представления цветов в пространстве RGB.

Функциональные цвета RGB

В функциональной модели RGB, в противоположность шестнадцатеричной нотации, поддерживаются два способа представления значений цветовых каналов. В каждом из них используется один и тот же синтаксис: `rgb(`*цвет*`)`, где *цвет* — это триплет цветовых значений, имеющих процентный или целочисленный тип данных. Процентные значения задаются в диапазоне `0-100%`, а целочисленные — в диапазоне `0-255`.

Таким образом, в функциональной модели RGB белый и черный цвета представляются такими процентными значениями.

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

Для передачи этих же цветов с помощью целочисленных значений применяется следующий синтаксис.

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

Учтите, что одновременно в одной и той же функции `rgb()` можно использовать значения только одного из типов данных. Следовательно, функция `rgb(255,66.67%,50%)` имеет недопустимый синтаксис и игнорируется браузером.

Предположим, что всем элементам `h1` документа нужно задать цветовой оттенок, средний между красным (`red`) и темно-бордовым (`maroon`). Красный цвет представляется выражением `rgb(100%,0%,0%)`, а темно-бордовый — функцией `rgb(50%,0%,0%)`. Средний цветовой оттенок для элементов `h1` будет представляться таким правилом:

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

В результате заголовки первого уровня получат более светлый оттенок, чем `maroon`, но он будет несколько темнее цвета `red`. Однако для получения светло-розового оттенка к значению красного цветового канала нужно добавить зеленую и синюю составляющие.

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

Сопоставимый ему цветовой оттенок, представляемый целочисленными значениями, задается с помощью такого правила:

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

Самый простой способ научиться сопоставлять цветовые оттенки на экране со значениями, представляющими насыщенность каждого из каналов в функциональной модели RGB, состоит в анализе таблицы полутонов, передаваемой следующими стилевыми правилами (рис. 4.10).

```
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);}
```

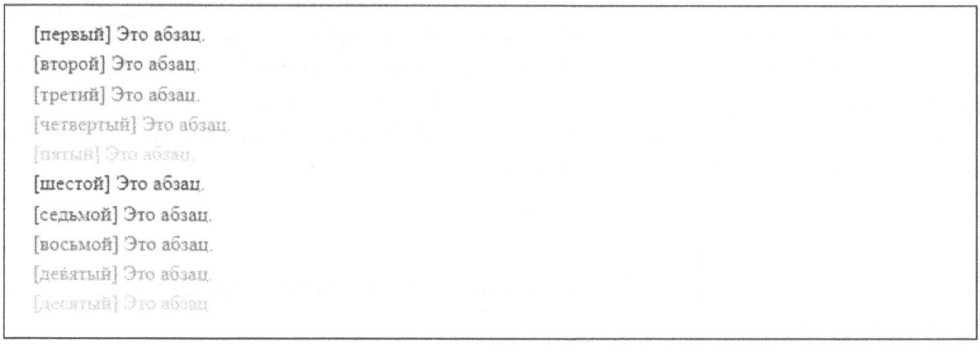


Рис. 4.10. Текст в разных оттенках серого

Поскольку речь идет о полутонах, каждый из оттенков представляется одинаковыми числовыми значениями всех трех цветовых каналов. Если хотя бы один из них будет отличаться от остальных двух, то результирующий оттенок из полутонового преобразуется в цветной. В частности, если преобразовать функцию `rgb(50%,50%,50%)` в `rgb(50%,50%,60%)`, то будет получен средне-серый оттенок с легкой примесью синего цвета.

В процентной форме записи значений допускается использовать дробные числа. Например, ничто не запрещает задать цвет с 25,5%-ной красной, 40%-ной зеленой и 98,6%-ной синей составляющими:

```
h2 {color: rgb(25.5%,40%,98.6%);}
```

Некоторые пользовательские агенты игнорируют десятичную часть дробных процентных значений, округляя их до ближайших целых чисел, что равнозначно применению следующего правила (целочисленные значения, что и так понятно, в округлении не нуждаются):

```
h2 {color: rgb(26%,40%,99%);}
```

При передаче функции `rgb()` аргументов, не попадающих в заранее заданный диапазон процентных или целочисленных значений, они будут представляться ближайшими предельными значениями. В результате значение 100% будет представлять все аргументы, *б*ольшие него, а значение 0% — все аргументы, меньшие него. Подобный подход применяется при обработке целочисленных аргументов. В приведенном ниже примере реальные передаваемые функции (обрабатываемые пользовательским агентом) аргументы указаны 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 */
```

Сопоставление процентных и целочисленных значений может показаться сложной задачей, поэтому для преобразования одних в другие лучше воспользоваться специальными формулами. Если известны процентные значения цветовых составляющих всех трех каналов, то для получения равнозначных им целочисленных значений достаточно умножить каждое из них на 255. Рассмотрим, как эта задача решается для цвета с 25%-ной красной, 35,5%-ной зеленой и 60%-ной синей составляющими. Умножив каждое из указанных процентных значений на 255, легко получить следующие действительные числа: 63,75, 95,625 и 153. Перед передачей в функцию `rgb()` каждое из них нужно округлить до ближайшего целого.

```
rgb(64,96,153)
```

Обратное преобразование выполняется не менее просто. Целочисленные значения будут понятнее каждому, кто занимается обработкой изображений в программе Photoshop, где сведения об используемых цветах выводятся на палитру Info и представляются числовыми значениями из диапазона 0–255.

Цветовая модель RGBa

В CSS3 функциональная модель RGB расширена до RGBa. В ней к стандартным цветовым каналам красного, зеленого и синего цветов добавлен альфа-канал (представлен буквой "a" в названия модели), отвечающий за прозрачность цвета.

Рассмотрим, как задать тексту некоего элемента полупрозрачный белый оттенок, позволяя просвечиваться сквозь него расположенному ниже фону. В модели RGBa эта задача решается с помощью следующих объявлений.

```
rgba(255,255,255,0.5)
rgba(100%,100%,100%,0.5)
```

Полностью прозрачный цвет представляется нулевым значением альфа-канала (четвертый аргумент функции `rgb()`). Значение `1` сопоставляется с абсолютно непрозрачным цветом. Таким образом, следующие две функции устанавливают один и тот же цвет (черный).

```
rgb(0,0,0)
rgba(0,0,0,1)
```

С помощью приведенных ниже правил каждому следующему абзацу документа задается более прозрачный (черный) оттенок, чем предыдущему (рис. 4.11).

```
p.one {color: rgba(0,0,0,1);}
p.two {color: rgba(0%,0%,0%,0.8);}
p.three {color: rgba(0,0,0,0.6);}
p.four {color: rgba(0%,0%,0%,0.4);}
p.five {color: rgba(0,0,0,0.2);}
```

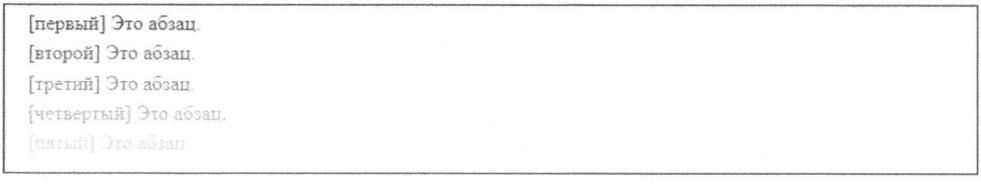


Рис. 4.11. Текст с возрастающей прозрачностью

Несложно догадаться, что значения прозрачности цвета устанавливаются числовыми значениями от 0 до 1. Любое другое число будет либо проигнорировано браузером, либо приведено к ближайшему предельно возможному значению. Несмотря на математическое подобие, прозрачность цвета нельзя задавать процентными значениями.

Шестнадцатеричная модель RGB

В CSS цвета допускается задавать с помощью шестнадцатеричных значений, применяемых еще в самых первых версиях HTML.

```
h1 {color: #FF0000;}  /* Красный элемент h1 */
h2 {color: #903BC0;}  /* Темно-фиолетовый элемент h2 */
h3 {color: #000000;}  /* Черный элемент h3 */
h4 {color: #808080;}  /* Средне-серый элемент h4 */
```

Шестнадцатеричный формат представления цветовых значений распознается компьютерами с незапамятных времен, но чтобы научиться передавать в нем произвольные цветовые оттенки, разработчикам необходимо изрядно практиковаться. Во многом шестнадцатеричная модель RGB стала популярна благодаря стремительному развитию Интернета и языка HTML, а впоследствии она стала неотъемлемой частью CSS.

В шестнадцатеричном коде цвет определяется тремя парами целочисленных значений из диапазона `00-FF`. В общем случае числовое значение задается в формате `#RRGGBB`. Учтите, что между цифрами не допускается использовать разделители (пробелы, запятые и т.п.).

При ближайшем рассмотрении становится очевидно, что каждая из пар целочисленных значений представляет насыщенность одного из трех цветовых каналов функциональной модели. Например, цвет `rgb(255,255,255)` задается шестнадцатеричным значением `#FFFFFF`, а цвет `rgb(51,102,128)` — числом `#336680`. Независимо от выбранной системы исчисления, большинство цветов будет одинаково отображаться во всех браузерах. Для более простого перехода от десятичных числовых значений к шестнадцатеричным используются специальные (инженерные) калькуляторы.

Если пары шестнадцатеричных чисел состоят из одинаковых цифр, то допускается использовать сокращенную форму записи шестнадцатеричного значения — `#RGB`, в которой каждый символ представляет сразу две цифры.

```
h1 {color: #000;}  /* Черный элемент h1 */
h2 {color: #666;}  /* Темно-серый элемент h2 */
h3 {color: #FFF;}  /* Белый элемент h3 */
```

В таком формате записи значение каждого цветового канала представляется только одной цифрой. Весьма необычно, особенно если учесть, что шестнадцатеричные числа задаются в диапазоне от `00` до `FF`.

Чтобы восполнить недостаток, браузер попросту создает дубликаты уже имеющихся цифр. В результате число `#F00` преобразуется в шестнадцатеричное значение `#FF0000`, `#6FA` — в значение `#66FFAA`, а `#FFF` становится числом `#FFFFFF`, представляющим белый цвет. Легко заметить, что сокращенную форму имеют далеко не все цветовые оттенки. Например, средне-серый цвет представляется шестнадцатеричным значением `#808080`, которое нельзя сократить до трехразрядного числа. Наиболее близкий к нему цвет с короткой формой записи — `#888`, представляющий значение `#888888`.

Шестнадцатеричная модель RGBa

В 2017 году стандартная шестнадцатеричная модель RGB была дополнена еще одним каналом, определяющим прозрачность цвета. С помощью приведенных ниже правил каждому следующему абзацу документа задается более прозрачный (черный) оттенок, чем предыдущему (рис. 4.12).

```
p.one {color: #000000FF;}
p.two {color: #000000CC;}
p.three {color: #00000099;}
p.four {color: #00000066;}
p.five {color: #00000033;}
```

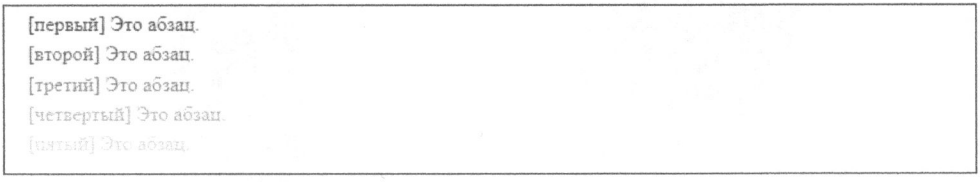


Рис. 4.12. Еще один пример текста с возрастающей прозрачностью

Как и в предыдущей модели, шестнадцатеричное значение можно выразить в краткой форме, но только в случае подобности цифр каждой из пар чисел. Таким образом, цветовое значение `#663399AA` будет представляться четырехразрядным числом `#639A`. Если хотя бы одна из пар содержит неодинаковые цифры, то цветовое значение нужно задавать в полноформатном виде.

К концу 2017 года шестнадцатеричная модель RGBa, описывающая в том числе и прозрачность цвета, поддерживалась только браузерами Firefox и Safari. В браузерах Chome и Opera ее поддержка так и не была полностью реализована и все еще находится на стадии тестирования.

Цветовые модели HSL и HSLa

В CSS3 впервые появилась поддержка цветовой модели HSL, известной задолго до появления Интернета. В модели HSL цветовые значения представляются в координатах тона (Hue), насыщенности (Saturation) и светлоты (Lightness). Цветовой тон задается числовым значением из диапазона 0–360, насыщенность — процентным значением от 0% (ненасыщенный цвет) до 100% (полностью насыщенный цвет), а светлота — процентным значением от 0% (абсолютно темный цвет) до 100% (абсолютно светлый цвет).

Цветовой тон проще всего определять на цветовом круге, в котором он задается углом расположения цветового оттенка. Отсчет тона (угол 0°) ведется от красного цвета — им же и заканчивается (360°). Угловое распределение цветовых оттенков на цветовом круге показано на рис. 4.13. Полноту цветового спектра на круге можно оценить, сравнив его с линейным спектром цветовой полосы, приведенной рядом.

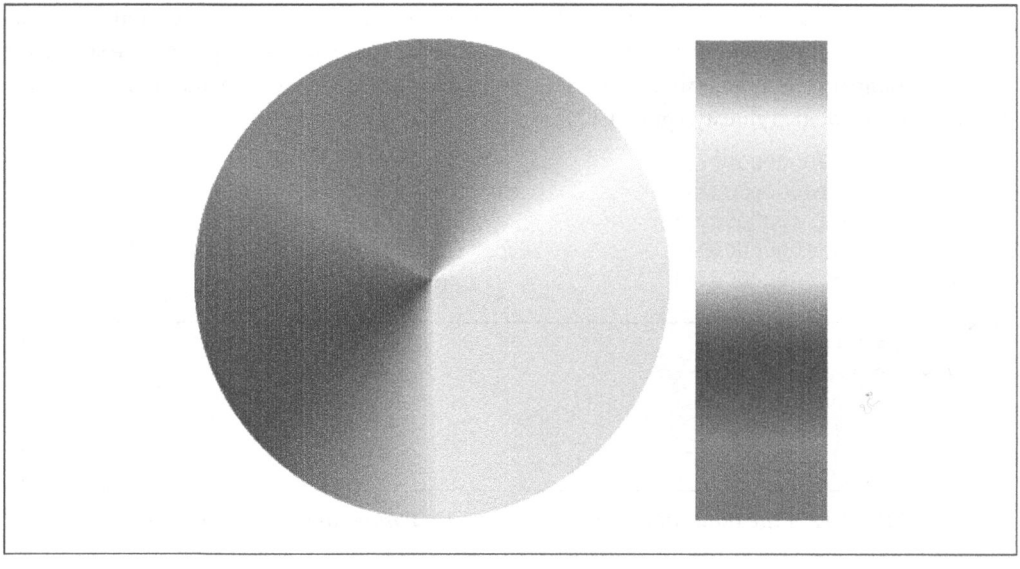


Рис. 4.13. Цветовой круг и цветовая полоса (см. цветные иллюстрации на веб-сайте)

Если вы ранее работали только с цветовыми значениями RGB, то поначалу цветовая модель HSL приведет вас в замешательство. Как ни странно, но немного обвыкнувшись с ней, вы вскоре будете недоумевать несовершенству модели RGB. Проще всего сопоставить цветовой тон модели HSL с цветовыми оттенками модели RGB, представив их совмещаемыми цветовыми полосами, как показано на рис. 4.14.

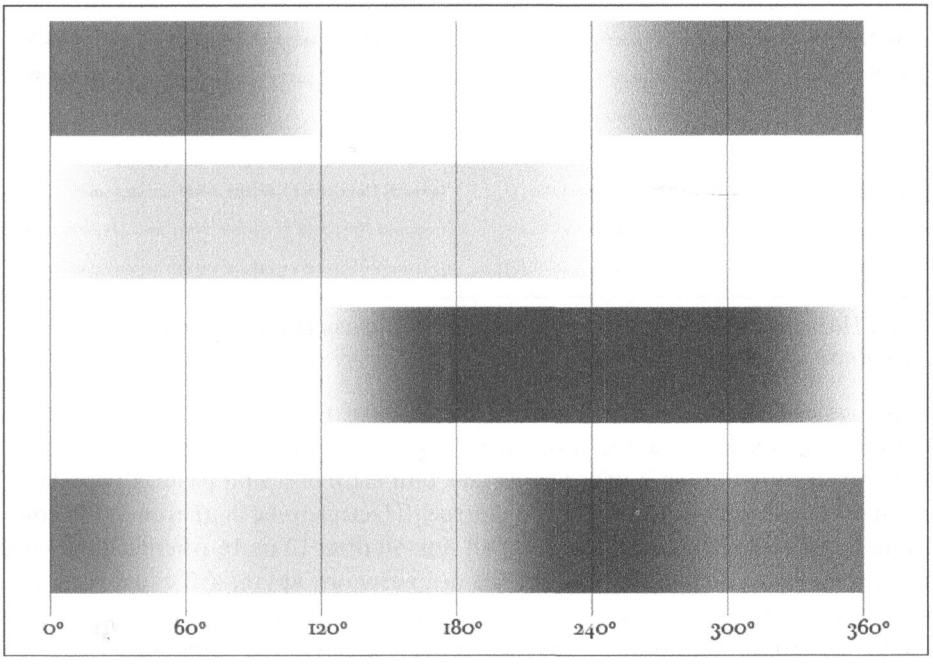


Рис. 4.14. Сопоставление цветовых оттенков модели RGB и цветового тона в модели HSL (см. цветные иллюстрации на веб-сайте)

Вторая координата модели HSL — насыщенность — представляет интенсивность цвета. При насыщенности 0% любой цветовой оттенок представляется полутоном. Для одного и того же уровня светлоты наиболее яркий цветовой оттенок достигается при насыщенности 100%. Наряду с этим светлота показывает, насколько темным или светлым есть цвет. При светлоте 0% цвет всегда черный, независимо от значений тона и насыщенности. Подобным образом светлота 100% всегда представляет белый цвет. Следующие правила наглядно иллюстрируют (рис. 4.15) применимость описанных выше цветовых характеристик.

```
p.one {color: hsl(0,0%,0%);}
p.two{color: hsl(60,0%,25%);}
p.three {color: hsl(120,0%,50%);}
p.four {color: hsl(180,0%,75%);}
p.five {color: hsl(240,0%,0%);}
p.six {color: hsl(300,0%,25%);}
p.seven {color: hsl(360,0%,50%);}
```

В левой колонке показан цвет текста при нулевой насыщенности — так он выглядит при черно-белой печати. Легко заметить, что все строки представлены в оттенках серого, определяемых исключительно значениями светлоты, а цветовой тон в данном случае не играет особой роли. Как только цветовой тон превышает 0% (в данном случае 50%), текст становится полноцветным, как показано в правой колонке. При равенстве насыщенности и светлоты, но разных цветовых тонах один и тот же текст выглядит совершенно по-разному.

[первый] Цвет текста абзаца имеет насыщенность 0%.
[второй] Цвет текста абзаца имеет насыщенность 0%.
[третий] Цвет текста абзаца имеет насыщенность 0%.
[четвертый] Цвет текста абзаца имеет насыщенность 0%.
[пятый] Цвет текста абзаца имеет насыщенность 0%.
[шестой] Цвет текста абзаца имеет насыщенность 0%.
[седьмой] Цвет текста абзаца имеет насыщенность 0%.

[первый] Цвет текста абзаца имеет насыщенность 50%.
[второй] Цвет текста абзаца имеет насыщенность 50%.
[третий] Цвет текста абзаца имеет насыщенность 50%.
[четвертый] Цвет текста абзаца имеет насыщенность 50%.
[пятый] Цвет текста абзаца имеет насыщенность 50%.
[шестой] Цвет текста абзаца имеет насыщенность 50%.
[седьмой] Цвет текста абзаца имеет насыщенность 50%.

Рис.4.15. Цветовой оттенок текста при разных уровнях насыщенности и светлоты (см. цветные иллюстрации на веб-сайте)

Для большей наглядности обозначим все базовые цвета HTML точками на цветовом круге, как показано на рис. 4.16. Цветовой круг не только содержит все возможные цветовые оттенки, но и показывает, как они выглядят при разных уровнях светлоты: от 50% у внешнего края до 0% в центре. (Насыщенность цветов одинакова для всего цветового круга и составляет 100%.) Как видите, 12 из 16 именованных цветов расположены равномерно по всей площади цветового круга, что свидетельствует о продуманности их выбора. Неравномерность наблюдается только в расположении оттенков серого цвета, но она вызвана скорее большей популярностью обозначенных цветов в художественной среде, чем недоработкой разработчиков модели.

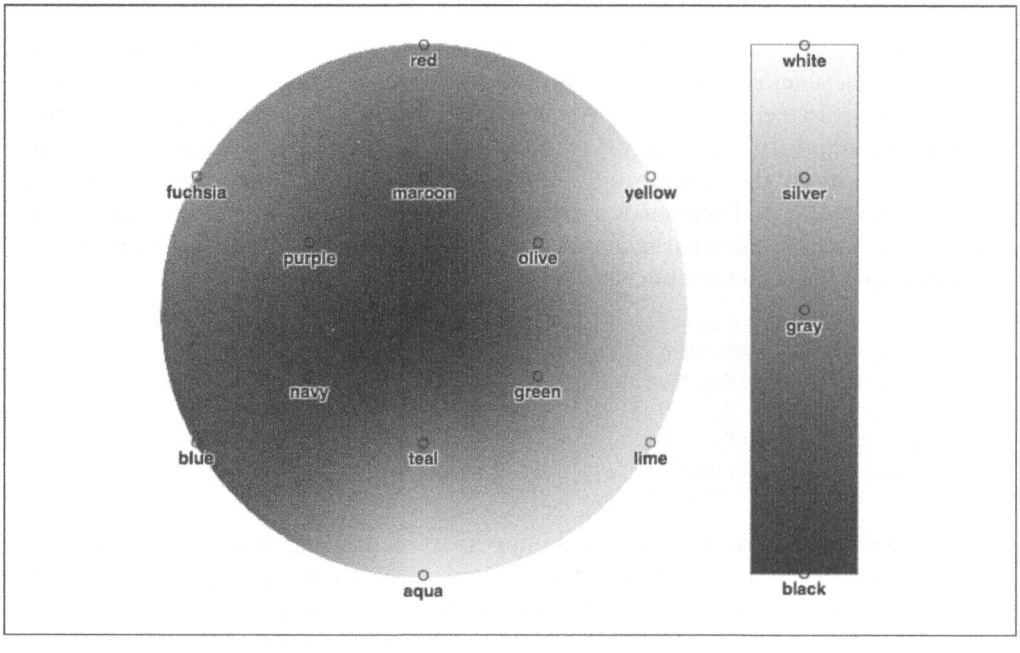


Рис. 4.16. Именованные цвета с разными уровнями насыщенности и светлоты на цветовом круге (см. цветные иллюстрации на веб-сайте)

Как и RGBa, цветовая модель HSLa была дополнена альфа-каналом. В HSLa представлены все исходные координаты базовой цветовой модели и добавлен канал, указывающий прозрачность конечного цветового оттенка. Приведенные ниже правила устанавливают разные уровни прозрачности для черного цвета, которым представлен текст абзаца, полностью повторяя форматирование, показанное на рис. 4.11.

```
p.one {color: hsla(0,0%,0%,1);}
p.two {color: hsla(0,0%,0%,0.8);}
p.three {color: hsla(0,0%,0%,0.6);}
p.four {color: hsla(0,0%,0%,0.4);}
p.five {color: hsla(0,0%,0%,0.2);}
```

Ключевые слова, обозначающие цветовые значения

Цветовые значения можно представлять двумя ключевыми словами: `transparent` и `currentColor`.

Как предполагает название, ключевое слово `transparent` обозначает полностью прозрачный цвет. В CSS Color Module он соответствует значению `rgba(0,0,0,0)` — по крайней мере, так оно воспринимается браузером. Это ключевое слово редко применяется для задания цвета текста, зато по умолчанию устанавливается в качестве цвета фона элементов. Кроме того, цвет `transparent` используется для обозначения границ элементов, которые занимают определенное пространство, но не отображаются в документе. Прозрачные цвета также нашли применение в градиентах, детально описанных в следующих главах.

В противоположность `transparent` ключевое слово `currentColor` представляет текущий вычисляемый цвет текущего элемента. Рассмотрим такое правило:

```
main {color: gray; border-color: currentColor;}
```

Первое объявление указывает использовать серый цвет (`gray`) в качестве основного цвета элемента. Благодаря ключевому слову `currentColor` во втором объявлении этот же цвет (равнозначный значению `rgb(50%,50%,50%)`) назначается границам элемента `main`.

Углы

Раз уж речь зашла об углах расположения оттенков на цветовом круге HSL, для их правильного задания нужно определиться с единицами измерения, в которых они представляются. В описаниях свойств углы обозначаются выражением *<angle>*, принимающим числовое значение одного из следующих четырех типов данных.

`deg`

Градусы; в полном круге 360 градусов.

`grad`

Грады; в полном круге 400 градов.

`rad`

Радианы; полный круг равен 2π радиан, или примерно `6.2832rad`.

`turn`

Обороты; круг равен одному обороту. Чаще всего применяются в анимации вращения заданное количество раз. В частности, значение `10turn` означает десять оборотов. К сожалению, многократно повторяющееся вращение в браузерах не обрабатывается (к концу 2017 года большие числовые значения, заданные в единицах `turn`, полностью игнорировались).

Чаще всего углы поворота (табл. 4.2) передаются свойствам, выполняющим трансформацию элементов в двух- или трехмерном пространстве, но иногда находят применение в иных способах форматирования документа. Обратите внимание на то, что цветовой тон в модели HSL, хотя и задается углом расположения оттенка на цветовом круге, представляется числовым значением, а не выражением *<angle>*.

Таблица 4.2. Перерасчет углов в основных единицах измерения

Градусы	Грады	Радианы	Обороты
0deg	0grad	0rad	0turn
45deg	50grad	0.785rad	0.125turn
90deg	100grad	1.571rad	0.25turn
180deg	200grad	3.142rad	0.5turn
270deg	300grad	4.712rad	0.75turn
360deg	400grad	6.283rad	1turn

Время и частота

Значение, представляющее период времени, задается в описании свойства выражением *<time>*. Время задается числовым значением, выраженным в единицах измерения `s` (секунды) или `ms` (миллисекунды). Временны*е* значения находят применение при настройке анимации и трансформации элементов, устанавливая длительность операции и задержку. Например, следующие два правила полностью идентичны.

```
a[href] {transition-duration: 2.4s;}
a[href] {transition-duration: 2400ms;}
```

Свойства, отвечающие за обработку звуковых данных, оперируют величиной, обратной времени, — частотой. В описании свойств она представляется выражением *<frequency>* и измеряется в герцах (`hz`) или килогерцах (`khz`). Учтите, что идентификаторы единиц измерения нечувствительны к регистру, поэтому наряду с `hz` в значениях свойств допускается использовать ключевое слово `Hz`. Ниже приведены два правила, применение которых приводит к одинаковому результату.

```
h1 {pitch: 128hz;}
h1 {pitch: 0.128khz;}
```

Положение

В описании свойств положение представляется выражением *<position>*. В частности, оно применяется при позиционировании графических изображений на фоне элемента. В общем случае выражение задается следующей синтаксической конструкцией.

```
[
   [ left | center | right | top | bottom | <percentage> | <length> ] |
   [ left | center | right | <percentage> | <length> ]
   [ top | center | bottom | <percentage> | <length> ] |
   [ center | [ left | right ] [ <percentage> | <length> ]? ] &&
   [ center | [ top | bottom ] [ <percentage> | <length> ]? ]
]
```

Звучит неправдоподобно, но приведенный выше шаблон включает все возможные варианты значений, представляющих положение элементов в документе.

Если указано только одно значение, например `left` или `25%`, то второе значение обязательно будет `center`. Так, значение `left` обозначает `left center`, а значение `25%` равно `25% center`.

Из двух чисел, представленных в единицах длины или процентах и включенных (неявно, как показано выше, или напрямую) в значение, первое *всегда* определяет горизонтальную размерность. Буквально это означает, что в значении `25% 35px` число `25%` представляет горизонтальное, а `35px` — вертикальное расстояние. Если поменять их местами, то горизонтальное расстояние будет определяться числом `35px`, а вертикальное — числом `25%`. При этом значение `25% left` или `35px right` будет считаться недействительным, поскольку в нем указано только горизонтальное расстояние, а вертикальное отсутствует. (По этой же причине браузером игнорируется значение `right left` или `top bottom`.) С другой стороны, значения `left 25%` и `right 35px` будут действительными, так как в них заданы обе размерности: горизонтальная — с помощью ключевого слова, вертикальная — в единицах длины или процентах.

При объявлении значения, состоящего сразу из четырех элементов (его примеры рассматриваются ниже), в единицах измерения длины или процентном виде должны представляться второй и четвертый элемент, а первый и третий — задаваться ключевыми словами. В подобных случаях числа устанавливают величину смещения, а ключевые слова — край, от которого оно отсчитывается. Таким образом, значение `right 10px bottom 30px` определяет смещение влево на 10 пикселей, начиная от правого края, и смещение вверх на 30 пикселей от нижнего края. Подобным образом значение `top 50% left 35px` устанавливает смещение вниз на 50% и смещение вправо на 35 пикселей (от левого края).

В трехэлементном значении применяются такие же правила анализа, за тем лишь исключением, что смещение в последнем направлении отсутствует (нулевое).

Пользовательские переменные

На момент выхода книги (конец 2017 года) в CSS была добавлена новая возможность, получившая техническое название *пользовательские свойства*. Техническое потому, что речь идет о создании не самих свойств, подобных определенным

спецификацией color или font, а, скорее, переменных, которые можно использовать в качестве значений.

Ниже приведен пример использования пользовательских переменных для форматирования заголовков документа (рис. 4.17).

```
html {
   --base-color: #639;
   --highlight-color: #AEA;
}

h1 {color: var(--base-color);}
h2 {color: var(--highlight-color);}
```

Рис. 4.17. Применение пользовательских переменных для установки цвета заголовков (см. цветные иллюстрации на веб-сайте)

Приведенный выше код имеет несколько особенностей, требующих отдельного рассмотрения. Первым обращает на себя внимание способ объявления пользовательских переменных --base-color и --highlight-color. Он интересен не столько кодировкой цветовых значений, сколько названиями, представляющими их назначение. Они могут быть произвольными, как, например, в следующем случае.

```
html {
   --alison: #639;
   --david: #AEA;
}

h1 {color: var(--alison);}
h2 {color: var(--david);}
```

Обозначенный способ именования будет понятен только тем, кто знает, какое отношение имеют Элисон и Дейвид к верстаемому документу. Поэтому от него лучше отказаться в пользу однозначно трактуемых идентификаторов данных, подобных main-color, accent-color или brand-font-face.

Второй важный момент заключается в добавлении двойного дефиса (--) в начало каждого идентификатора. Имена пользовательских переменных чувствительны к регистру, и это нужно учитывать при назначении их свойствам с помощью функции var(). Таким образом, переменные --main-color и --Main-color представляют совершенно разные значения.

Как уже указывалось, для подключения пользовательских переменных к свойствам применяется функция var(). Их поддержка свойствами определяется специ-

фикацией, но не забывайте, что они не являются переменными в строгом понимании значения этого термина. С точки зрения программирования пользовательские переменные больше подобны макросам в офисных приложениях, поскольку всего лишь замещают одни значения другими.

Пользовательские значения удобно использовать для контекстного оформления документов. Последнее требует более подробного объяснения, и проще всего это сделать на примере, результат которого показан на рис. 4.18.

```
html {
   --base-color: #639;
}
aside {
   --base-color: #F60;
}

h1 {color: var(--base-color);}

<body>

<h1>Заголовок первого уровня</h1><p>Основной текст</p>

<aside>
   <h1>Заголовок первого уровня</h1><p>Сторонний текст</p>
</aside>

<h1>Заголовок первого уровня</h1><p>Основной текст</p>

</body>
```

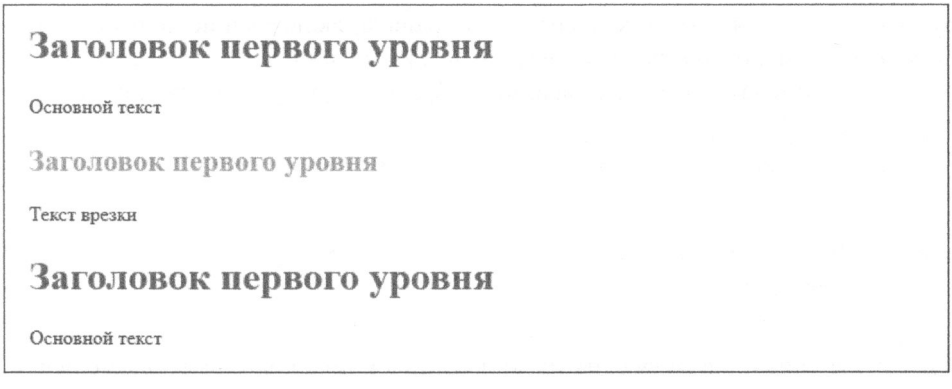


Рис. 4.18. Контекстное применение пользовательских переменных (см. цветные иллюстрации на веб-сайте)

Обратите внимание на цветовое оформление заголовков. Внутри элемента `aside` они оранжевые, а снаружи — фиолетовые, что вызвано обновлением переменной `--base-color` для элементов `aside`. Новое значение переменной устанавливается только для элементов, вложенных в `aside`.

Пользовательские переменные открывают перед нами многие недоступные ранее возможности, большинство из которых связано с контекстным изменением

значений. Ниже приведен пример, предложенный Кристианом Штайнмайером, в котором одинаковые отступы во всех одноуровневых элементах вложенных списков задаются благодаря использованию в стилевых правилах пользовательских переменных и вычисляемых значений.

```
html {
   --gutter: 3ch;
   --offset: 1;
}

ul li {margin-left: calc(var(--gutter) * var(--offset));}
ul ul li {--offset: 2;}
ul ul ul li {--offset: 3;}
```

По действию он подобен следующей таблице стилей, в которой применяются только регулярные выражения.

```
ul li {margin-left: 3ch;}
ul ul li {margin-left: 6ch;}
ul ul ul li {margin-left: 9ch;}
```

Различие между предложенными вариантами в том, что в первом случае для задания других отступов достаточно изменить множитель `--gutter`, а во втором — нужно вручную задавать новые значения для всех трех уровней вложения элементов.

Использование переменных с произвольными именами открывает перед разработчиками совершенно новые способы стилевого оформления документов, выходящие за рамки стандартной парадигмы CSS. При экспериментировании с пользовательскими переменными не забывайте о том, что у них очень низкий уровень поддержки браузерами. Не забывайте также снабжать правила, включающие пользовательские переменные, медиа-функциями, объявляемыми командами `@supports`. С их помощью вы скроете пользовательские переменные от браузеров, не распознающих их.

```
@supports (color: var(--custom)) {
   /* правила, содержащие пользовательские переменные */
}

@supports (--custom: value) {
   /* альтернативные правила */
}
```

И снова повторюсь: на момент выхода книги пользовательские переменные все еще не были полноценно включены в спецификацию CSS. К концу 2017 года многие вопросы их применения в таблицах стилей оставались несогласованными. В частности, не были утверждены ограничения на использование их в свойствах, не выяснен эффект влияния на каскадирование стилей и т.п. Несмотря на большой потенциал, пользовательские переменные нужно задействовать с оглядкой на то, что в один прекрасный момент их поведение в пользовательских агентах может быть изменено или даже заблокировано.

ГЛАВА 5

Шрифты

Раздел "Font Properties" спецификации CSS1, утвержденной в 1996 году, начинается с оптимистического предположения, что стилевые правила в первую очередь будут востребованы при форматировании текстов. Как ни странно, но утверждение сохраняет прежнюю актуальность даже спустя несколько десятилетий бурного развития технологии.

В CSS2 была объявлена возможность использования в стилевых правилах пользовательских шрифтов, для чего в спецификацию добавлена команда `@font-face`, хотя распространение она приобрела только в 2009 году с широкой поддержкой большинством популярных браузеров. К тому же современные веб-сайты, получая соответствующие полномочия, могут запрашивать у пользовательской системы любые шрифты, впоследствии применяемые для стилевого форматирования документов. Таким образом, разработчики получили возможность использовать в своих документах любые шрифты, известные пользовательскому агенту.

Однако даже столь совершенные средства не гарантируют абсолютного контроля над шрифтами, применяемыми для оформления документа. Как только шрифты перестанут загружаться с указанного места или начнут представляться неизвестным браузеру форматом, текст будет выводиться резервным шрифтом. Но даже в столь неоднозначном поведении браузера есть положительный момент — несмотря ни на что, пользователи получают возможность ознакомиться с содержимым страницы. Авторам документов нужно не забывать о том, что их оформление напрямую зависит от доступности шрифтов, применяемых для форматирования, в пользовательской системе.

Семейства шрифтов

Обычно термином "шрифт" описывают сразу несколько вариантов (курсив, полужирный и т.п.) одних и тех же символов. Скорее всего, вам известен шрифт Times, включающий такие подмножества символов, как TimesRegular, TimesBold, TimesItalic, TimesBoldItalic и др. В общепринятом понимании каждое из них представляет собой отдельное *начертание* шрифта Times. Иными словами, Times нужно рассматривать как семейство шрифтов, а не отдельный шрифт, хотя последний вариант определения привычнее большинству из нас.

Чтобы окончательно расставить все точки над "i", в спецификации CSS определяется пять основных начертаний шрифтов.

Шрифт с засечками (`serif`)

Пропорциональный шрифт, символы которого снабжены засечками. Шрифт считается пропорциональным, если ширина его символов зависит от их пропорций. Например, в таких шрифтах символы *i* и *m* имеют разную ширину. Засечками называются декоративные элементы на концах букв, подобные штрихам в верхней и нижней частях буквы *I* или на концах каждой из "ног" буквы *A*. К шрифтам с засечками относятся, например, Times, Georgia и New Century Schoolbook.

Шрифт без засечек (`sans-serif`)

Пропорциональный шрифт без засечек. К таким шрифтам относятся, например, Helvetica, Geneva, Verdana, Arial и Univers.

Моноширинный шрифт (`monospace`)

В отличие от пропорционального шрифта, символы моноширинного шрифта имеют одинаковую ширину. Чаще всего такие шрифты применяются для отображения программных кодов и табличных данных. Исходя из названия, символы *i* и *m* таких шрифтов занимают одинаковое место в строке. Моноширинные буквы бывают как с засечками, так и без них. Наличие или отсутствие засечек никак не сказывается на моноширинности символов шрифта. К шрифтам этого начертания относятся Courier, Courier New, Consolas и Andale Mono.

Курсивный шрифт (`cursive`)

Имитирует письмо от руки и состоит из наклоненных под одним и тем же углом символов с декоративными выносными элементами, большими по размеру, чем засечки. Например, в курсивном начертании буква "A" может иметь крупный завиток на левой "ноге" или вообще состоять из одних только криволинейных форм. К курсивным относятся такие шрифты, как Zapf Chancery, Author и Comic Sans.

Декоративный шрифт

Шрифты этой категории настолько разные, что их нельзя классифицировать по одному или даже нескольким признакам (за необычный внешний вид символов их иногда называют фантастическими). К декоративным шрифтам относятся Western, Woodblock и Klingon.

Теоретически любой шрифт относится к одной из описанных выше категорий. В действительности несовпадения все же случаются, но не часто — все нераспознанные шрифты первых четырех категорий браузеры относят к декоративному типу.

Основные семейства шрифтов

В CSS семейство шрифтов определяется с помощью свойства `font-family`.

font-family	
Значение	[<*family-name*> | <*generic-family*>]#
Начальное значение	Определяется пользовательским агентом
Применяется	Все элементы
Вычисляется	Согласно значению
Наследуется	Да
Анимируется	Нет

Для отображения текста документа одним из шрифтов семейства `san-serif` применяется следующее правило:

```
body {font-family: sans-serif;}
```

Оно указывает пользовательскому агенту представить весь текст элемента `body` шрифтом без засечек (например, Helvetica). Благодаря принципу наследования этим же шрифтом будет выводиться текст всех элементов, вложенных в `body` (если, конечно, им специально не назначены иные шрифты).

Добиться хорошего внешнего вида документа можно с помощью стилевого форматирования, основанного на определении одних только семейств шрифтов. Пример такого оформления документа (с помощью приведенных ниже правил) приведен на рис. 5.1.

```
body {font-family: serif;}
h1, h2, h3, h4 {font-family: sans-serif;}
code, pre, tt, kbd {font-family: monospace;}
p.signature {font-family: cursive;}
```

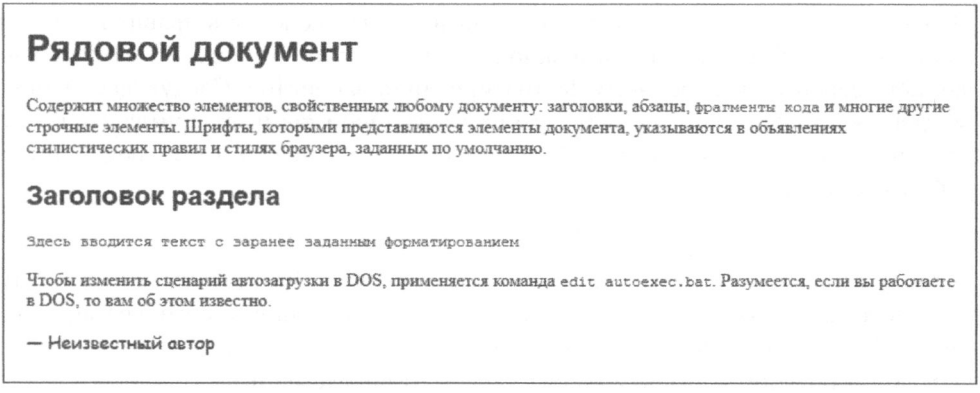


Рис. 5.1. Представление текста шрифтами разных начертаний

В данном случае для отображения большей части текста применяется шрифт с засечками, подобный Times. Он назначен всем абзацам, за исключением одного (атрибут `class` которого имеет значение `signature`) — его текст выводится шрифтом курсивного начертания (таким, как Author). Заголовки с первого по четвертый уровень представляются шрифтом без засечек, подобным Helvetica, а элементы `code`, `pre`, `tt` и `kbd` — моноширинным шрифтом, например Courier.

Определение шрифта

В стилевом правиле можно точно указать шрифт, который будет применяться для представления текста отдельных элементов или всего документа. Целевые шрифты допускается устанавливать как авторам документов, так и пользователям, отображающим их в пользовательском агенте. В обоих случаях применяется свойство `font-family`.

Предположим, все элементы `h1` документа нужно представить шрифтом Georgia. Простейшее правило, обеспечивающее решение этой задачи, приведено ниже.

```
h1 {font-family: Georgia;}
```

Результат применения его в документе, содержащем один заголовок первого уровня, показан на рис. 5.2.

Рис. 5.2. Представление текста элемента `h1` шрифтом Georgia

Для применения правила необходимо, чтобы пользовательский агент имел доступ к шрифту Georgia. Если браузеру не известно расположение файла шрифта, то правило применяться не будет. Правило будет проигнорировано, а для вывода текста элемента `h1` будет применяться шрифт пользовательского агента, заданный по умолчанию (если таковой назначен), а не Georgia.

Но не все так плохо, как кажется на первый взгляд. Включив в правило название шрифта или семейства шрифтов, можно добиться если не точного, то максимально подобного заранее оговоренному форматирования элементов. Следующее правило приведено в продолжение предыдущего примера. Оно указывает применять к заголовкам первого уровня шрифт Geogia, а в его отсутствие использовать любой другой шрифт с засечками:

```
h1 {font-family: Georgia, serif;}
```

Если в системе отсутствует шрифт Georgia, но есть Times, то именно им будут представляться элементы `h1`. Даже несмотря на то что многие символы шрифтов Georgia и Time существенно отличаются друг от друга, одни и те же текстовые фрагменты, представленные каждым из них, обладают определенным сходством.

Описанные выше причины побуждают снабжать все стилевые свойства `font-family` значениями, указывающими не только шрифт, но и семейство шрифтов, к которым он относится. Подобный механизм замены шрифтов позволяет выводить

содержимое элемента даже в отсутствие шрифта, указанного в стилевом правиле. Ниже приведено несколько примеров.

```
h1 {font-family: Arial, sans-serif;}
h2 {font-family: Charcoal, sans-serif;}
p {font-family: 'Times New Roman', serif;}
address {font-family: Chicago, sans-serif;}
```

Часто работая с текстами, вы точно знаете, какие из шрифтов лучше всего подходят для отображения каждого из элементов. Допустим, вам нужно представить текст абзацев шрифтом Times. В случае отсутствия в системе его можно заменить следующими шрифтами без засечек: Times New Roman, Georgia, New Century Schoolbook и New York. Последовательность указания названий шрифтов в правиле определяет приоритет их замены в случае отсутствия предыдущих вариантов.

```
p {font-family: Times, 'Times New Roman', 'New Century Schoolbook',
                Georgia, 'New York', serif;}
```

Согласно данному утверждению пользовательский агент будет искать шрифты в порядке их перечисления в объявлении стиля. Если ни один из вариантов не найден, будет использоваться любой доступный в системе шрифт с засечками.

Кавычки в названиях шрифтов

Скорее всего, вы уже обратили внимание на одинарные кавычки, в которые заключены названия отдельных шрифтов в предыдущем правиле. Они применяются только в том случае, когда название шрифта или семейства шрифтов включает один или несколько пробелов (состоит из нескольких слов). В одинарные кавычки также заключаются значения свойства `font-family`, содержащие специальные символы, такие как `#` или `$`. Исходя из последнего правила, кавычки обязательны при добавлении в стилевое правило шрифта Karrank%.

```
h2 {font-family: Wedgie, 'Karrank%', Klingon, fantasy;}
```

Если пользовательский агент встретит название шрифта без кавычек и не распознает его, то он продолжит выполнять остальную часть правила.

Имейте в виду, что описанные выше требования, хотя и выдвигаются в спецификации CSS, давно утратили свою актуальность. Современные пользовательские агенты прекрасно распознают названия шрифтов, не заключенные в кавычки, даже если те состоят из нескольких слов и включают специальные символы. Как выяснилось, кавычки по-прежнему необходимы только в одной ситуации — при добавлении в правило названий шрифтов, содержащих ключевые слова, которыми в `font-family` обозначаются семейства шрифтов. Таким образом, для добавления в правило шрифта с названием "cursive" его нужно обязательно выделить кавычками, чтобы отличить от ключевого слова `cursive`, имеющего строго заданное назначение в CSS.

```
h2 {font-family: Author, "cursive", cursive;}
```

Если в названии шрифта отсутствуют ключевые слова, определенные для свойства `font-family`, то заключать их в кавычки нет необходимости. Подобным образом в кавычки не нужно заключать ключевые слова, представляющие названия

семейств шрифтов (`serif`, `monospace` и др.). В противном случае пользовательский агент попытается найти шрифт с названием, передаваемым ключевым словом (например, "serif"), а не семейство шрифтов.

Значения свойства `font-family` разрешается заключать как в одинарные, так и в двойные кавычки. Не забывайте, что при подключении к элементу правила, содержащего объявление свойства `font-family`, с помощью атрибута `style` (что не рекомендуется делать никогда) в нем нужно использовать кавычки, отличные от тех, в которые заключено значение атрибута. Вследствие этого, если правило заключено в двойные кавычки, то в нем допускается использовать только одинарные кавычки, как показано в следующих примерах.

```
p {font-family: sans-serif;}  /* шрифт sans-serif по умолчанию */

<!-- правило действительно (одинарные кавычки) -->
<p style="font-family: 'New Century Schoolbook', Times, serif;">...</p>

<!-- правило недействительно (двойные кавычки) -->

<p style="font-family: "New Century Schoolbook", Times, serif;">...</p>
```

Применение двойных кавычек в правиле приводит к их конфликту с кавычками, определяющими значение атрибута `style`, что видно по получаемому результату.

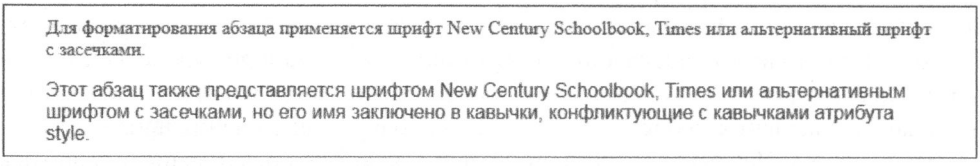


Рис. 5.3. Результат неправильного использования кавычек

Правило `@font-face`

Впервые команда `@font-face` появилась в CSS2, но широкое распространение получила только в начале 2000-х годов. Она применяется для настройки и загрузки шрифтов, применяемых в документе. Важность команды сложно переоценить, поскольку все пользовательские агенты получают доступ к разным наборам системных шрифтов.

Предположим, в таблице стилей применяется малораспространенный шрифт, отсутствующий в большинстве устройств, на которых будет просматриваться документ. Команда `@font-face` позволяет указать файл шрифта, применяемый для визуализации текста документа и расположенный на сервере. При визуализации документа пользовательский агент сначала будет загружать шрифт и только затем применять его в стилевом правиле, как если бы он имелся в системе. Ниже приведен простой пример.

```
@font-face {
   font-family: "SwitzeraADF";
   src: url("SwitzeraADF-Regular.otf");
}
```

В правиле указан файл шрифта (с расширением `.otf`), который загружается пользовательским агентом перед форматированием текста, устанавливаемым свойством `font-family: SwitzeraADF`.

В приведенном выше примере использован шрифт SwitzeraADF, взятый из коллекции Arkandis Digital Foundry:

`http://arkandis.tuxfamily.org/openfonts.html`

Команда `@font-face` отвечает за отложенную загрузку шрифтов, выполняемую только по мере необходимости, а не вместе со всем документом. При отложенной загрузке к документу подключаются только те шрифты, которые участвуют в его непосредственной визуализации, и никакие другие. Вне всяких сомнений, браузер, загружающий сразу все объявленные в документе шрифты, даже те, в которых нет необходимости, работает с ошибками.

Обязательные дескрипторы

Настройки, определяющие применяемые в документе шрифты, задаются в конструкторе `@font-face { }`. Они называются *дескрипторами* и во многом подобны свойствам, в том числе форматом объявления — *`дескриптор: значение;`*. В действительности большинство имен дескрипторов повторяет названия свойств, о чем рассказано ниже.

У команды `@font-face` всего два обязательных дескриптора: `font-family` и `src`.

font-family	
Значение	*<family-name>*
Начальное значение	Не задано

src	
Значение	[[*<uri>* [`format`(*<string>*#)]?] | *<font-face-name>*]#
Начальное значение	Не задано

Назначение дескриптора `src` вполне очевидно. Он задает расположение файла шрифта, который применяется для визуализации текста документа. При указании нескольких источников их адреса разделяются запятыми. В качестве адреса допускается использовать URL, но только ссылающийся на ресурс, содержащий саму таблицу стилей. Получается, что дескриптор `scr` не может указывать на сторонний сайт, равно как и загружать с него данные. Таблица стилей и файлы шрифтов должны обязательно находиться на общем сервере — вашем или одной из многочисленных служб хостинга.

Для того чтобы обойти ограничения на хостинг таблицы стилей и связанных с ней шрифтов в одном месте, применяется технология межсайтовой загрузки, основанная на использовании HTTP-заголовка `Access-Control-Allow-Origin`.

Вас может заинтересовать вопрос: зачем объявлять `font-family` с помощью команды `@face-font`, когда эту же задачу можно решить, следуя инструкциям предыдущего раздела? Ответ очень прост: в команде `@face-font` определяется не свойство, а дескриптор `font-family`. О том, какое между ними различие, рассказывается далее.

Функционально команда `@face-font` создает низкоуровневое определение, на котором основывается действие всех свойств, связанных с управлением шрифтами, например `font-family`. В частности, дескриптор `font-family: "SwitzeraADF";` добавляет данные шрифта "SwitzeraADF" в базу семейств шрифтов, известных пользовательскому агенту, приравнивая его к Helvetica, Georgia, Courier и т.п. Впоследствии на него можно ссылаться в свойстве `font-family` как на один из системных шрифтов.

```
@font-face {
   font-family: "SwitzeraADF"; /* дескриптор */
   src: url("SwitzeraADF-Regular.otf");
}
h1 {font-family: SwitzeraADF, Helvetica, sans-serif;} /* свойство */
```

Обратите внимание на полное совпадение значений свойства и дескриптора `font-family`. Если они будут отличаться, то в правиле для элемента `h1` первое значение будет проигнорировано и применено заданное после него. Внешний шрифт будет загружен и применен для форматирования элементов документа (рис. 5.4) только при абсолютном совпадении значений и правильном указании его расположения в дескрипторе `src`.

Рис. 5.4. Применение загружаемого шрифта для форматирования текста

В качестве значения второго обязательного дескриптора (`scr`) можно указать сразу несколько адресов. Если пользовательскому агенту не удастся загрузить файл шрифта из первого источника, то он обратится к резервному ресурсу, определяемому вторым и последующим адресами.

```
@font-face {
   font-family: "SwitzeraADF";
   src: url("SwitzeraADF-Regular.otf"),
        url("/fonts/SwitzeraADF-Regular.otf");
}
```

Не забывайте, что резервный файл шрифта также должен располагаться на одном с таблицей стилей сервере. Для доступа к резервному файлу, расположенному на другом ресурсе, нужно разрешить межсайтовую загрузку данных.

Для того чтобы указать пользовательскому агенту формат загружаемого шрифта, применяется необязательный дескриптор `format()`.

```
@font-face {
   font-family: "SwitzeraADF";
   src: url("SwitzeraADF-Regular.otf") format("opentype");
}
```

Использование дескриптора `format()` позволяет браузеру отказаться от загрузки заведомо неподдерживаемых им шрифтов, высвобождая канал передачи данных под другие цели, что повышает общую скорость загрузки веб-страницы. К тому же с его помощью можно указать формат шрифта, файл которого не имеет расширения, а потому исходно не известен синтаксическому анализатору пользовательского агента.

```
@font-face {
   font-family: "SwitzeraADF";
   src: url("SwitzeraADF-Regular.otf") format("opentype"),
        url("SwitzeraADF-Regular.true") format("truetype");
}
```

Неоформленное содержимое

Занимаясь разработкой или дизайном веб-страниц продолжительное время, вы должны быть знакомы с понятием "неоформленное содержимое". В ранних версиях браузеров неоформленное содержимое отображалось в окне браузеров перед загрузкой кода CSS и представлением страницы в конечном, правильно сверстанном виде. Оно позволяло ознакомиться с содержимым документа, представленным стилями браузера по умолчанию, до стилевого форматирования его правилами CSS, на что порой уходило очень много времени.

Подобная ситуация также возникает при медленной загрузке файлов специальных шрифтов, определенных в стилевых правилах. Исходно текст документа представляется шрифтами по умолчанию (или резервными шрифтами) и только после загрузки всех необходимых файлов из внешнего источника выводится в окончательном, правильно оформленном виде.

Поскольку применение специальных шрифтов к простому тексту неизбежно приводит к существенному изменению размеров элементов, резервные шрифты для них нужно подбирать предельно аккуратно. Если размеры символов в специальных шрифтах и шрифтах по умолчанию существенно отличаются, то замена одних другими приведет к однозначному переформатированию макета документа. К сожалению, в CSS отсутствуют инструменты автоматического предотвращения такого эффекта, даже несмотря на использование свойства `font-size-adjust`, рассмотренного далее.

Всегда надежнее снабжать стилевые правила резервными шрифтами с такой же формой символов, как у специальных шрифтов.

Причины появления в документе неоформленного содержимого прежние — загрузка отдельных частей документа с разной скоростью и отображение элементов в окончательном виде по мере получения стилевыми правилами всех необходимых данных. Технические возможности позволяют современным браузерам отображать документы, не прибегая к выводу неоформленного содержимого, но проблема автоматического подбора резервных шрифтов в них стоит как нельзя остро. Бытует вполне обоснованное мнение, что вскоре будет решена и она — так же, как в свое время удалось отказаться от вывода неоформленного содержимого.

Список поддерживаемых браузерами форматов шрифтов приведен в табл. 5.1.

Таблица 5.1. Форматы шрифтов, распознаваемые браузерам

Значение	Формат
`embedded-opentype`	EOT (Embedded OpenType)
`opentype`	OTF (OpenType)
`svg`	SVG (Scalable Vector Graphics)
`truetype`	TTF (TrueType)
`woff`	WOFF (Web Open Font Format)

В дополнение к `url()` и `format()` в команде `@font-face`, устанавливающей семейство(а) шрифтов, можно применять дескриптор `local()`, указывающий на расположение файла шрифта в пользовательской системе.

```
@font-face {
   font-family: "SwitzeraADF";
   src: local("Switzera-Regular"),
        local("SwitzeraADF-Regular"),
        url("SwitzeraADF-Regular.otf") format("opentype"),
        url("SwitzeraADF-Regular.true") format("truetype");
}
```

В приведенном примере браузер начнет поиск шрифта с названием "Switzera-Regular" или "SwitzeraADF-Regular" с локальной системы. При успешном нахождении именно он будет представлять семейство шрифтов `SwitzeraADF` в стилевых правилах. Если ни один из требуемых шрифтов в системе не обнаружен, то файл семейства шрифтов будет загружен из источника, указанного дескриптором `url()`.

Дескриптор `local()` удобно использовать для назначения коротких имен системным шрифтам. В следующем примере короткое имя получает шрифт Helvetica (или в случае его отсутствия — шрифт Helvetica Neue).

```
@font-face {
   font-family: "H";
   src: local("Helvetica"), local("Helvetica Neue");
}

h1, h2, h3 {font-family: H, sans-serif;}
```

Если шрифт Helvetica установлен в пользовательском компьютере, то он назначается заголовкам первых трех уровней. Описанный подход кажется несколько замысловатым, хотя в отдельных ситуациях он позволяет значительно сократить размер таблицы стилей, применяемой в документе.

Устойчивость к ошибкам

Команда `@font-face` становится незаменимой при форматировании документов, предназначенных для просмотра в браузерах разных поколений, каждый из которых поддерживает форматы шрифтов, понятных только ему. (В табл. 5.1 содержится сводная информация о форматах шрифтов, поддерживаемых браузерами на протяжении всей истории их существования.) Чтобы обеспечить корректность отображения текста в самых разных браузерах, необходимо сделать команду `@font-face` устойчивой ко всем возможным ошибкам. Синтаксис такой команды, исходно разработанной Полом Айришем и оптимизированный командой FontSpring, имеет следующий вид.

```
@font-face {
   font-family: "SwitzeraADF";
   src: url("SwitzeraADF-Regular.eot");
   src: url("SwitzeraADF-Regular.eot?#iefix") format("embedded-
            opentype"),
        url("SwitzeraADF-Regular.woff") format("woff"),
        url("SwitzeraADF-Regular.ttf") format("truetype"),
        url("SwitzeraADF-Regular.svg#switzera_adf_regular") format("svg");
}
```

Разберем приведенный выше код по частям. Первая строка, вложенная в команду `@font-face`, очень простая: она определяет имя семейства шрифтов. Следующие две строки имеют такой вид.

```
src: url("SwitzeraADF-Regular.eot");
src: url("SwitzeraADF-Regular.eot?#iefix") format("embedded-opentype");
```

В них определяются шрифты EOT (Embedded OpenType), которые поддерживаются браузерами, распознающими этот формат: Internet Explorer версий 6–9. Первая строка указывает семейство шрифтов для браузера IE9, запущенного в режиме совместимости, а вторая устанавливает его для IE6–IE8. Инструкция `?#iefix` устраняет ошибку синтаксического анализа, которая приводит к отображению в окне браузера страницы 404 при определении в команде `@font-face` шрифтов сразу нескольких форматов. В IE9 ошибка множественных форматов была исправлена, поэтому следующая строка вложенного кода лишена инструкции `?#iefix`:

```
url("SwitzeraADF-Regular.woff") format("woff"),
```

Этой командой назначаются шрифты формата WOFF (Web Open Font Fornat), распознаваемые большинством современных браузеров. Она последняя из предназначавшихся для настольных систем. Следующая строка распознается браузерами, работающими под управлением iOS и Android, — она указывает шрифты преимущественно для мобильных устройств:

```
url("SwitzeraADF-Regular.ttf") format("truetype"),
```

Наконец, последняя строка определяет формат шрифтов для старых iOS-систем — к ним относятся устройства, не попадающие ни в одну из предыдущих категорий.

Ситуация усложняется, когда в документе необходимо обозначить семейства шрифтов для нескольких начертаний. К счастью, всегда можно воспользоваться услугами специальных служб, автоматически генерирующих правила `@font-face` для указанного начертания и преобразующих исходный файл шрифта во все необходимые форматы. Одна из лучших служб, решающих подобную задачу, — @Font-Face Kit Generator, доступная на сайте Font Squirrel (`http://fontsquirrel.com/fontface/generator`). Перед тем как обратиться к ней, обязательно удостоверьтесь, что обладаете достаточными полномочиями для преобразования шрифтов в другие форматы и применения их на веб-страницах (см. врезку "Соглашение об использовании шрифтов").

Остальные дескрипторы

Кроме `font-family` и `src` команда `@font-face` поддерживает целый ряд других, необязательных дескрипторов, связывающих начертания шрифтов со значениями определенных свойств. Эти дескрипторы (табл. 5.2), как и дескриптор `font-family`, сопоставляются с отдельными свойствами CSS и устанавливают для них значения по умолчанию, запрашиваемые пользовательскими агентами.

Таблица 5.2. Дескрипторы настроек шрифтов

Дескриптор	Значение по умолчанию	Описание
`font-style`	`normal`	Определяет начертание шрифта: обычное, курсивное или наклонное
`font-weight`	`normal`	Устанавливает насыщенность шрифта
`font-stretch`	`normal`	Определяет плотность начертания шрифта (т.е. задает ширину символов)
`font-variant`	`normal`	Задает способ представления строчных символов (в том числе в виде капители); в большинстве случаев является оптимизированной версией дескриптора `font-feature-settings`
`font-feature-settings`	`normal`	Отрывает доступ к низкоуровневым настройкам шрифтов OpenType (в том числе устанавливающим лигатуры)
`unicode-range`	`U+0-10FFFF`	Определяет диапазон символов для пользовательского шрифта

Соглашение об использовании шрифтов

При изменении шрифтов нужно обращать внимание на два важных момента. Во-первых, необходимо удостовериться, что пользовательским соглашением разрешается применять их на веб-страницах. Вы должны иметь веские причины на применение пользовательских шрифтов в документах, размещаемых в Интернете.

Условия использования файлов шрифтов, как и снимков, получаемых из фотобанков, оговариваются специальными соглашениями. Далеко не каждое из них разрешает применять файлы шрифтов в HTML-документах. Чтобы не попасть впросак, старайтесь отображать текст на веб-страницах только шрифтами FOSS (Free and Open-Source Software — свободное программное обеспечение с открытым исходным кодом) или пользуйтесь услугами таких служб, как Fontdeck или Typekit, предлагающих юридически выверенные решения по преобразованию файлов целевых шрифтов в другие форматы. В противном случае перед применением шрифта где бы то ни было нужно удостовериться, что подобные действия не противоречат условиям, оговоренным в пользовательском соглашении (опять-таки напрашивается аналогия с изображениями, приобретаемыми в фотобанках).

Учтите, что чем больше шрифтов используется в документе, тем больше места на сервере будет занимать веб-страница и тем длительнее будет ее загрузка в пользовательском агенте. Файлы шрифтов имеют не очень большой размер — обычно от 50 до 100 Кбайт, — но их количество может быть очень велико, особенно при использовании шрифтов с большим количеством специальных символов или необычных начертаний. Из вышесказанного понятно, что при управлении шрифтами и изображениями приходится преодолевать одинаковые трудности. Как и в любых других случаях, выбор приходится делать либо в пользу эффектного оформления документа, либо скорости его загрузки в пользовательском агенте. Конечное решение зависит от множества факторов и принимается отдельно для каждого документа.

Как и следовало ожидать, наряду с инструментами оптимизации изображений существуют специальные средства оптимизации шрифтов. Чаще всего под оптимизацией шрифта подразумевают удаление из него символов, не задействуемых в документе. Такие службы, как Typekit и Fonts.com, скорее всего, располагают собственными инструментами оптимизации шрифтов и/или проводят ее автоматически в процессе преобразования файлов шрифтов в другие форматы.

Поскольку дескрипторы, приведенные в табл. 5.2, необязательные, их можно не указывать в правиле `@font-face`. Но даже неупомянутые в команде `@font-face` дескрипторы имеют значения по умолчанию, передаваемые соответствующим свойствам. Например, если в правиле не определен дескриптор `font-weight`, то он все равно принимает значение `normal`.

Ограничение диапазона символов шрифта

Из всех дескрипторов, перечисленных в табл. 5.2, только `unicode-range` не имеет связанного с ним стилевого свойства. Он позволяет задать диапазон символов, которым характеризуется пользовательский шрифт. Обычно он указывается для символьных шрифтов, а также шрифтов, содержащих символы вспомогательных языков документа.

unicode-range	
Значение	*<urange>#*
Начальное значение	`U+0-10FFFF`

По умолчанию дескриптор `unicode-range` определяет полный диапазон значений Unicode, что означает отображение одним шрифтом всех представленных в тексте символов. Чаще всего именно это и требуется, хотя в отдельных случаях к определенному содержимому нужно применить специальные шрифты. Рассмотрим следующий пример использования возможностей модуля CSS Fonts Level 3.

```
unicode-range: U+590-5FF; /* Символы иврита */
unicode-range: U+4E00-9FFF, U+FF00-FF9F, U+30??; /* Японские символы кандзи,
хирагана и катакана */
```

В первой строке определены символы с 590 по 5FF кодовой страницы Unicode, соответствующие ивриту. Такой подход позволяет отформатировать текст только шрифтом иврита, даже если в нем имеются символы, представленные другими Unicode-значениями.

```
@font-face {
   font-family: "CMM-Ahuvah";
   src: url("cmm-ahuvah.otf" format("opentype");
   unicode-range: U+590-5FF;
}
```

Во второй строке устанавливается сразу несколько разделенных запятыми диапазонов, представляющих японские символы. Отдельно стоит уделить внимание значению `U+30??`, включающему сразу несколько подстановочных знаков. В данном случае вопросительным знаком обозначается любая цифра, поэтому значение `U+30??` может представлять любой символ из диапазона `U+3000-30FF`. Учтите, что в Unicode в качестве подстановочного допускается применять только вопросительный знак.

Диапазоны указываются только в порядке возрастания значений. Убывающие диапазоны (например, `U+400-300`) синтаксическим анализатором не распознаются и не обрабатываются. Помимо диапазонов в дескрипторе `unicode-range` можно указывать отдельные значения Unicode, подобные `U+221E`. Обычно они дополняют заданные ранее диапазоны, как показано ниже.

```
unicode-range: U+4E00-9FFF, U+FF00-FF9F, U+30??, U+A5;
/* Японские символы кандзи, хирагана и катакана,
   а также символ юаня */
```

Последнее обособленное Unicode-значение позволяет указать шрифт для единственного символа. Эффективность такого способа форматирования текста зависит от множества факторов: структуры документа, размера файлов шрифтов, скорости соединения и предпочтений разработчиков.

Поскольку основное предназначение правила `@font-face` заключается в оптимизации загрузки данных веб-страницы, в дескрипторе `unicode-range` необходимо указывать только те шрифты, которые точно будут применяться для форматирования текста документа. В качестве примера рассмотрим веб-сайт, на котором публикуются работы известных математиков, представленные на двух языках: английском и русском. При этом заранее не известно, какое содержимое будет добавляться на каждую страницу — это может быть только текст на русском языке, математические формулы и текст на английском языке или все содержимое сразу. Кроме того, пусть каждый тип содержимого выводится шрифтом отельного начертания. Для загрузки файлов начертаний необходимо использовать следующие конструкторы `@font-face`.

```
@font-face {
   font-family: "MyFont";
   src: url("myfont-general.otf" format("opentype");
}
@font-face {
   font-family: "MyFont";
   src: url("myfont-cyrillic.otf" format("opentype");
   unicode-range: U+04??, U+0500-052F, U+2DE0-2DFF, U+A640-A69F,
                  U+1D2B-1D78;
}
@font-face {
   font-family: "MyFont";
   src: url("myfont-math.otf" format("opentype");
   unicode-range: U+22??; /* представляет диапазон U+2200-22FF */
}
```

В первом правиле дескриптор `unicode-range` отсутствует, поэтому указанный в нем файл шрифта загружается всегда. Эта операция не выполняется только в одном случае: если веб-страница не содержит никакого текста. Второе правило указывает загружать определенный в нем файл шрифта только в случае отображения на веб-странице текста, символы которого представляются значениями из заданного диапазона. Третье правило выполняет такие же действия, как и второе, но соответствует математическим знакам, а не буквам кириллицы.

Комбинирование дескрипторов

Необходимость комбинирования дескрипторов возникает только тогда, когда к тексту нужно применить форматирование, определяемое сразу несколькими стилевыми свойствами. Например, в определенных ситуациях требуется назначить три разных шрифта: для полужирного начертания, курсива и текста, одновременно представляемого обоими начертаниями.

При решении этой задачи используется факт задания значений по умолчанию всем дескрипторам, не объявленным в правиле `@font-face`. Ниже приведен базовый синтаксис трех правил `@font-face`.

```
@font-face {
   font-family: "SwitzeraADF";
   font-weight: normal;
   font-style: normal;
   font-stretch: normal;
   src: url("SwitzeraADF-Regular.otf") format("opentype");
}
@font-face {
   font-family: "SwitzeraADF";
   font-weight: bold;
   font-style: normal;
   font-stretch: normal;
   src: url("SwitzeraADF-Bold.otf") format("opentype");
}
@font-face {
   font-family: "SwitzeraADF";
   font-weight: normal;
   font-style: italic;
   font-stretch: normal;
   src: url("SwitzeraADF-Italic.otf") format("opentype");
}
```

Каждое из правил можно сократить, убрав из него инструкции, в которых дескрипторам задаются значения по умолчанию — `normal`. В результате их код сокращается до следующего вида.

```
@font-face {
   font-family: "SwitzeraADF";
   src: url("SwitzeraADF-Regular.otf") format("opentype");
}
@font-face {
   font-family: "SwitzeraADF";
   font-weight: bold;
   src: url("SwitzeraADF-Bold.otf") format("opentype");
}
@font-face {
   font-family: "SwitzeraADF";
   font-style: italic;
   src: url("SwitzeraADF-Italic.otf") format("opentype");
}
```

Во всех трех правилах отсутствуют настройки по умолчанию, а значения дескрипторов `font-weight` и `font-style` изменяются от правила к правилу. Составим правило, определяющее шрифт сразу для двух начертаний: полужирного и курсивного.

```
@font-face {
   font-family: "SwitzeraADF";
   font-weight: bold;
   font-style: italic;
   font-stretch: normal;
   src: url("SwitzeraADF-BoldItalic.otf") format("opentype");
}
```

А как насчет уплотненного полужирного курсивного начертания?

```
@font-face {
   font-family: "SwitzeraADF";
   font-weight: bold;
   font-style: italic;
   font-stretch: condensed;
   src: url("SwitzeraADF-BoldCondItalic.otf") format("opentype");
}
```

А теперь это же правило, но для уплотненного курсивного начертания нормальной насыщенности.

```
@font-face {
   font-family: "SwitzeraADF";
   font-weight: normal;
   font-style: italic;
   font-stretch: condensed;
   src: url("SwitzeraADF-CondItalic.otf") format("opentype");
}
```

Как видите, для каждого типа начертания задается свой набор дескрипторов. Остановимся на достигнутом, совместив все заданные выше варианты начертаний в едином правиле `@font-face`. После исключения настроек по умолчанию оно принимает следующий вид. Результат применения комбинированного правила показан на рис. 5.5.

```
@font-face {
   font-family: "SwitzeraADF";
   src: url("SwitzeraADF-Regular.otf") format("opentype");
}
@font-face {
   font-family: "SwitzeraADF";
   font-weight: bold;
   src: url("SwitzeraADF-Bold.otf") format("opentype");
}
@font-face {
   font-family: "SwitzeraADF";
   font-style: italic;
```

```
   src: url("SwitzeraADF-Italic.otf") format("opentype");
}
@font-face {
   font-family: "SwitzeraADF";
   font-weight: bold;
   font-style: italic;
   src: url("SwitzeraADF-BoldItalic.otf") format("opentype");
}
@font-face {
   font-family: "SwitzeraADF";
   font-weight: bold;
   font-stretch: condensed;
   src: url("SwitzeraADF-BoldCond.otf") format("opentype");
}
@font-face {
   font-family: "SwitzeraADF";
   font-style: italic;
   font-stretch: condensed;
   src: url("SwitzeraADF-CondItalic.otf") format("opentype");
}
@font-face {
   font-family: "SwitzeraADF";
   font-weight: bold;
   font-style: italic;
   font-stretch: condensed;
   src: url("SwitzeraADF-BoldCondItalic.otf") format("opentype");
}
```

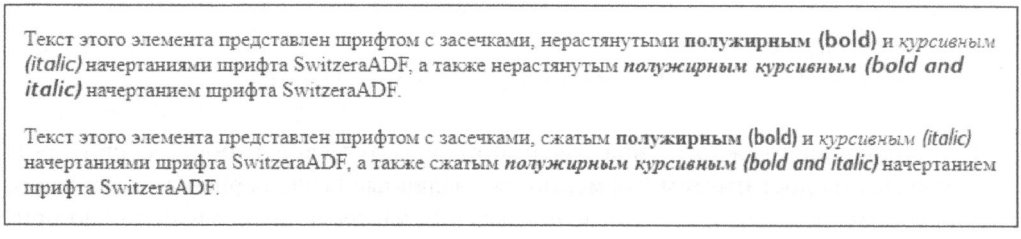


Рис. 5.5. Применение шрифтов разных начертаний

Несложно заметить, что количество возможных комбинаций просто огромное, ведь у дескриптора `font-weight` одиннадцать значений, а у дескриптора `font-stretch` — десять. Вам вряд ли придется использовать их все. В действительности у большинства шрифтов намного меньше начертаний, чем у SwitzeraADF (на текущий момент их 24), поэтому описывать каждое из них просто бессмысленно. С другой стороны, чем больше вариантов начертаний назначено в команде `@font-face`, тем меньше шрифтов по умолчанию будет применяться пользовательским агентом.

Насыщенность шрифта

Завершив изучение дескрипторов команды `@font-face`, рассмотрим свойства, устанавливающие настройки шрифтов. Вы уже познакомились с обычным и полужирным начертаниями, представляющими всего два значения насыщенности шрифта. В стилевых правилах насыщенность шрифта задается свойством `font-weight`.

font-weight

Значение	`normal` | `bold` | `bolder` | `lighter` | `100` | `200` | `300` | `400` | `500` | `600` | `700` | `800` | `900`
Начальное значение	`normal`
Применяется	Все элементы
Вычисляется	Одно из числовых значений (`100`, `200` и т.д.), значение уровня или относительное значение (`bolder` или `lighter`)
Примечание	Снабжено отдельным дескриптором в команде `@font-face`
Наследуется	Да
Анимируется	Нет

В общем случае, чем больше насыщенность шрифта, тем темнее и жирнее его символы. Шрифты с разной степенью насыщенности представлены широким спектром названий. В частности, уже упомянутый шрифт SwitzeraADF имеет следующие варианты начертаний, названия которых свидетельствуют об их насыщенности: SwitzeraADF Bold, SwitzeraADF Extra Bold, SwitzeraADF Light и SwitzeraADF Regular. Все они имеют символы одинаковой формы, но разной толщины штрихов.

Предположим, в документе нужно применить все возможные начертания шрифта SwitzeraADF, представляющие разные уровни насыщенности символов. Обратиться к ним можно напрямую — через свойство `font-family`, — но это не совсем правильный подход. Кроме того, каждый раз указывать полное название шрифта, как показано ниже, достаточно утомительно.

```
h1 {font-family: 'SwitzeraADF Extra Bold', sans-serif;}
h2 {font-family: 'SwitzeraADF Bold', sans-serif;}
h3 {font-family: 'SwitzeraADF Bold', sans-serif;}
h4, p {font-family: 'SwitzeraADF Regular', sans-serif;}
small {font-family: 'SwitzeraADF Light', sans-serif;}
```

Правила выглядят достаточно громоздко, чтобы отказаться от их использования. Намного эффективнее сначала обозначить семейство шрифтов, применяемое в документе, а затем назначить каждому из элементов начертание с необходимым уровнем насыщенности. Установка уровня насыщенности начертания выполняется в свойстве `font-weight` после определения семейства шрифтов в команде `@font-face`. Ниже приведен пример простейшего объявления.

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

Данное правило назначает полужирное начертание всем элементам b документа. Под полужирным подразумевается начертание, насыщенность которого заметно больше, чем у обычного начертания. Это вполне ожидаемый результат, особенно учитывая назначение элемента b.

Строго говоря, выполнение этой операции сводится к назначению шрифта полужирного начертания к указанным в правиле элементам. Таким образом, если основной текст абзацев представляется шрифтом Times и только небольшая его часть имеет полужирное начертание, то в действительности к нему применяются сразу два шрифта: Times и TimesBold. Регулярный текст выводится шрифтом Times, а полужирный — шрифтом TimesBold.

Уровень насыщенности

Насыщенность шрифта (исключая унаследованную от родительских элементов) передается пользовательскому агенту через числовые значения, задаваемые в диапазоне от 100 до 900 с шагом 100. Они определяют девять уровней "жирности" символов, далеко не все из которых представлены в каждом отдельно взятом семействе шрифтов. Если семейство шрифтов содержит файлы начертаний для всех уровней насыщенности, то значение 100 будет соответствовать наиболее светлому шрифту, а значение 900 — сверхжирному.

Фактически значения насыщенности шрифта не отражают степень характеристики. Спецификация CSS указывает лишь на то, что каждое из значений представляет отдельный уровень насыщенности шрифта, не меньший предыдущего. Таким образом, значения 100, 200, 300 и 400 могут представлять одно и то же сверхсветлое начертание, значения 500 и 600 — одно и то же нормальное начертание, а значения 700, 800 и 900 — соответствовать сверхжирному шрифту. Числовые значения будут справедливы до тех пор, пока не противоречат уровням насыщенности, задаваемым ключевыми словами.

В случаях обозначения уровня насыщенности шрифта ключевыми словами каждому из них сопоставляется одно из обозначенных выше числовых значений. Стоит заметить, что начертание normal соответствует насыщенности 400, а полужирный шрифт (bold) — насыщенности 700. Другие уровни насыщенности не всегда представляются ключевыми словами, но часто отражаются в названиях шрифтов. Например, шрифты, названия которых включают слова "Normal", "Regular", "Roman" или "Book", имеют насыщенность 400, а шрифты с названиями, включающими слово "Medium", — насыщенность 500. Наряду с этим, если шрифт представлен единственным начертанием, включающим только ключевое слово "Medium", то он представляется насыщенностью 500, а не 400.

Пользовательские агенты прекрасно справляются с определением начертаний шрифтов, которые имеют меньше десяти уровней насыщенности. Недостающие значения заполняются согласно следующим правилам.

- Если в семействе шрифтов отсутствует начертание с насыщенностью 500, то оно представляется начертанием предыдущего уровня насыщенности (400).

- Отсутствующее начертание с насыщенностью 300 представляется шрифтом предыдущего уровня насыщенности, меньшим 400. В отсутствие начертаний с меньшей насыщенностью числовое значение 300 сопоставляется со шрифтами обычной жирности, имеющими в названии ключевое слово "Normal" или "Medium". Этот же метод применяется при определении начертания для насыщенности 200 и 100.
- Если в семействе шрифтов отсутствует начертание с насыщенностью 600, то оно представляется начертанием следующего уровня насыщенности, большего 500. В случае отсутствия такового для представления начертания с насыщенностью 600 применяется шрифт, имеющий насыщенность 500. Такой же подход справедлив при определении отсутствующих начертаний с уровнем насыщенности 700, 800 и 900.

Рассмотрим три примера применения описанных выше принципов на практике. В первом из них форматирование текста выполнятся с помощью OpenType-шрифта Karrank%, включающего начертания для девяти уровней насыщенности. Каждый уровень передается отдельным числовым значением, а ключевые слова normal и bold представляют соответственно насыщенность 400 и 700. Это очень простой пример, так как семейства шрифтов, включающие начертания для всех девяти уровней насыщенности, встречаются очень редко. (Себестоимость их производства очень высока, а потому заниматься изготовлением таких шрифтов нецелесообразно.)

Во втором примере форматирование текста выполняется с помощью семейства шрифтов SwitzeraADF, упоминаемого в примерах предыдущих разделов. Гипотетически его начертания, определяемые свойством font-weight, могут представляться числовыми значениями насыщенности так, как показано в табл. 5.3.

Таблица 5.3. Предположительные уровни насыщенности у разных начертаний одного семейства шрифтов

Начертание	Ключевое слово	Уровень насыщенности
SwitzeraADF Light		100, 200, 300
SwitzeraADF Regular	normal	400
SwitzeraADF Medium		500
SwitzeraADF Bold	bold	600, 700
SwitzeraADF Extra Bold		800, 900

Первые три уровня насыщенности соответствуют самому светлому начертанию. Начертание, название которого включает слово "Regular", представляется ключевым словом normal и числовым значением 400. Следовательно, начертание со словом "Medium" в названии соответствует насыщенности 500. Так как насыщенность 600 не представлена отдельным начертанием, то она, как и насыщенность 700, назначена полужирному начертанию, описываемому ключевым словом bold. Наконец, насыщенностью 800 и 900 передаются начертания, имеющие в имени соответственно

слова "Black" и "UltraBlack". Обратите внимание на то, что в случае отсутствия в семействе шрифтов двух последних начертаний значения 800 и 900 будут представлять полужирное начертание или одно наиболее жирное начертание, в названии которого встречается слово "Black".

Последний пример описывает начертания сокращенной версии семейства шрифтов Times. Как показано в табл. 5.4, оно представляется всего двумя уровнями насыщенности символов.

Таблица 5.4. Предположительные уровни насыщенности шрифтов "Times"

Начертание	Ключевое слово	Уровень насыщенности
TimesRegular	normal	100, 200, 300, 400, 500
TimesBold	bold	600, 700, 800, 900

Как и следовало ожидать, ключевыми словами normal и bold обозначаются соответственно обычное и полужирное начертания. Числовые значения 100–300 представляют обычное начертания, поскольку оно является наиболее светлым из двух доступных вариантов. Значение 400 устанавливается для него по умолчанию, а насыщенность 500 задана обычному начертанию в отсутствие начертания "Medium", которое она обычно представляет в семействе шрифтов. Что касается полужирного начертания, то значением 700 оно передается по умолчанию, а значения 800 и 900 определены ему из-за отсутствия в семействе более жирных вариантов. Наконец, числовое значение 600 назначается следующему большему по насыщенности начертанию, которое представляется шрифтом со словом "Bold" в названии.

Свойство font-weight наследуется, поэтому при назначении абзацу полужирного начертания все его дочерние элементы также будут представлены полужирным шрифтом (рис. 5.6).

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

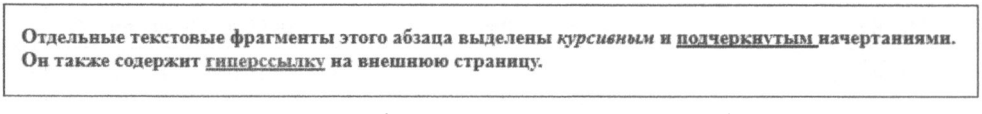


Рис. 5.6. Наследование насыщенности шрифта

В наследовании свойств нет ничего необычного, но в случае насыщенности шрифтов оно имеет свои особенности, связанные с использованием ключевых слов bolder и lighter. Как следует из названий, они применяются для увеличения или уменьшения насыщенности начертания дочерних элементов. Первым рассмотрим ключевое слово bolder.

Повышение насыщенности

При определении уровня насыщенности ключевым словом bolder пользовательский агент сначала определяет значение свойства font-weight родительского элемента. Целевой элемент представляется шрифтом с более высоким уровнем насыщенности, чем у родительского элемента. Если в семействе шрифтов отсутствует более

жирное начертание следующего уровня, то насыщенность элемента увеличивается до еще большего значения (вплоть до 900). Ниже рассмотрено несколько типичных случаев применения значения bolder (рис. 5.7).

```
p {font-weight: normal;}
p em {font-weight: bolder;} /* полужирное начертание;
                               насыщенность увеличивается до 700 */

h1 {font-weight: bold;}
h1 b {font-weight: bolder;} /* в отсутствие начертаний
                               насыщенность увеличивается до 800 */

div {font-weight: 100;} /* предполагает применение начертания 'Light' */
div strong {font-weight: bolder;} /* обычное начертание; насыщенность
                                     увеличивается до 400 */
```

Рис. 5.7. Увеличение жирности текста

В первой паре правил начертание элементов em увеличивается с normal до bold. В числовом выражении эта операция соответствует повышению насыщенности шрифта со значения 400 до 700. Во втором примере элементам h1 уже назначено полужирное начертание (bold). В отсутствие файла более жирного начертания насыщенность шрифта элементов b, вложенных в элементы h1, будет увеличена до 800 единиц (как известно, начертанию bold соответствует числовое значение 700). Но поскольку в выбранном шрифте оба уровня насыщенности представлены одним и тем же начертанием, при отображении в пользовательском агенте жирность элементов h1 и b будет одинаковой.

В третьем примере текст абзаца отображается самым светлым начертанием заданного шрифта. Кроме него семейство шрифтов включает обычное ("Regular") и полужирное ("Bold") начертания. Дочерним (по отношению к абзацам) элементам em будет назначено обычное начертание, как ближайшее более темное из возможных. А как изменится ситуация, если шрифт будет представлен всего двумя начертаниями: обычным и полужирным?

```
/* шрифт содержит всего два начертания: обычное и полужирное */
p {font-weight: 100;} /* начертание как у 'обычного' текста */
p span {font-weight: bolder;} /* насыщенность увеличивается до 700 */
```

Как видите, в последнем случае насыщенность 100 соответствует обычному начертанию, хотя свойство font-weight, как и ранее, получает значение 100. Этот же уровень насыщенности передается элементами span, вложенными в элементы p, и впоследствии насыщенность увеличивается до значения 700, представляемого полужирным начертанием.

А теперь еще больше усложним задачу, дополнив таблицу стилей несколькими правилами, применяемыми к приведенному ниже фрагменту HTML-документа.

```
/* шрифт содержит всего два начертания: обычное и полужирное */
p {font-weight: 100;} /* начертание как у 'обычного' текста */
p span {font-weight: 400;} /* начертание как в предыдущем элементе */
strong {font-weight: bolder;} /* начертание жирнее, чем у 'родителя' */
strong b {font-weight: bolder;} /* предыдущее начертание */

<p>
Этот абзац содержит текстовые фрагменты увеличивающейся насыщенности:
<span>акцентированный элемент,<strong> включающий сильно
акцентированный элемент <b>, который в свою очередь содержит элемент,
представленный полужирным начертанием</b></strong></span>.
</p>
```

Этот абзац содержит текстовые фрагменты увеличивающейся насыщенности: *акцентированный элемент, включающий* ***сильно акцентированный элемент, который в свою очередь содержит элемент, представленный полужирным начертанием***.

Рис. 5.8. Повышение уровня насыщенности вложенных элементов

Повышение значения свойства `font-weight` у элементов последних двух уровней вложения является следствием кумулятивного эффекта ключевого слова `bolder`. Если текст каждого элемента заменить значениями насыщенности его начертания, то приведенный выше фрагмент HTML-кода примет такой вид.

```
<p>
100 <span> 400 <strong> 700 <b> 800 </b> </strong> </span>.
</p>
```

Первые два перехода к более жирному начертанию самые существенные, поскольку вызваны увеличением значения насыщенности с `100` до `400` и с `400` до `700` (полужирное). В шрифте отсутствуют начертания с насыщенностью, большей `700`, поэтому переход к значению `800` визуально никак не проявляется. Если в дальнейшем в элемент `strong` добавить еще один элемент `b`, то в документе будут следующие уровни насыщенности.

```
<p>
100 <span> 400 <strong> 700 <b> 800 <strong> 900
</strong> </b> </strong> </span>.
</p>
```

Насыщенность дополнительного элемента `b`, имеющего самый глубокий уровень вложения, представлена максимальным значением: `900`. Учитывая то, что шрифт включает всего два варианта начертаний, элементы документа приобретают такое форматирование (рис. 5.9).

```
<p>
обычное <span> обычное <strong> полужирное <b> полужирное <strong>
полужирное </strong> </b> </strong> </span>.
</p>
```

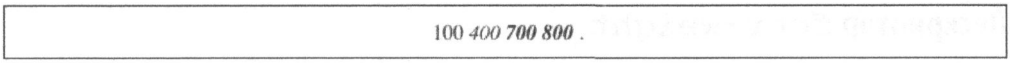


Рис. 5.9. Значения насыщенности элементов, передаваемые действительными начертаниями

Понижение насыщенности

Как и следовало ожидать, ключевое слово `lighter`, как и рассмотренное выше значение `bolder`, изменяет уровень насыщенности шрифта, но в сторону понижения, а не увеличения. Оно указывает пользовательскому агенту применить начертание предыдущего, меньшего уровня насыщенности. Чтобы понять, как это работает, несколько модифицируем пример предыдущего раздела, заменив в нем ключевое слово `bolder` значением абсолютно противоположного смысла.

```
/* шрифт содержит всего два начертания: обычное и полужирное */
p {font-weight: 900;} /* предельно жирное начертание,
                         выглядящее как полужирное */
p span {font-weight: 700;} /* полужирное начертание */
strong {font-weight: lighter;} /* начертание светлее, чем
                                  у 'родителя' */
strong b {font-weight: lighter;} /* предыдущее начертание */

<p>
900 <span> 700 <strong> 400 <b> 300 <strong> 200
</strong> </b> </strong> </span>.
</p>
<!-- ...или в виде начертаний... -->
<p>
полужирное <span> полужирное <strong> обычное <b> обычное <strong>
обычное /strong></b></strong></span>.
</p>
```

Несмотря на противоестественность, основной текст абзаца данного примера представляется полужирным начертанием (рис. 5.10), имеющим тем не менее значение начертания `900`. В результате применения свойства `font-weight` со значением `lighter` элемент `strong` приобретает начертание предыдущего уровня светлости — в данном случае обычное (значение насыщенности `400`) — как единственно возможное. Дальнейшее уменьшение уровня насыщенности (до значения `300`) не изменяет начертание, поскольку в семействе шрифтов их всего два, и обычное — самое светлое из них. Пользовательский агент может пошагово понижать насыщенность вплоть до значения `100` (в данном примере не рассматривается), но в документе это отражаться не будет. В последнем абзаце показаны начертания, приобретаемые каждым из элементов.

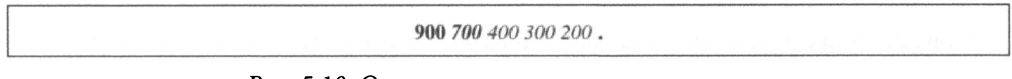


Рис. 5.10. Осветление текстовых элементов

Дескриптор font-weight

Допустимые уровни насыщенности шрифта, представляемые отдельными начертаниями с помощью свойства font-weight, задаются одноименным дескриптором. Например, следующие правила устанавливают начертания для шести значений свойства font-weight.

```
@font-face {
   font-family: "SwitzeraADF";
   font-weight: normal;
   src: url("f/SwitzeraADF-Regular.otf") format("opentype");
}
@font-face {
   font-family: "SwitzeraADF";
   font-weight: bold;
   src: url("f/SwitzeraADF-Bold.otf") format("opentype");
}
@font-face {
   font-family: "SwitzeraADF";
   font-weight: 300;
   src: url("f/SwitzeraADF-Light.otf") format("opentype");
}
@font-face {
   font-family: "SwitzeraADF";
   font-weight: 500;
   src: url("f/SwitzeraADF-DemiBold.otf") format("opentype");
}
@font-face {
   font-family: "SwitzeraADF";
   font-weight: 700;
   src: url("f/SwitzeraADF-Bold.otf") format("opentype");
}
@font-face {
   font-family: "SwitzeraADF";
   font-weight: 900;
   src: url("f/SwitzeraADF-ExtraBold.otf") format("opentype");
}
```

После объявления в таблице стилей указанные начертания можно применять для форматирования текста, подставляя в свойство font-weight. Результат операции приведен на рис. 5.11.

```
h1, h2, h3, h4 {font: 225% SwitzeraADF, Helvetica, sans-serif;}
h1 {font-weight: 900;}
h2 {font-size: 180%; font-weight: 700;}
h3 {font-size: 150%; font-weight: 500;}
h4 {font-size: 125%; font-weight: 300;}
```

При установке начертания для каждого из элементов пользовательский агент в первую очередь анализирует значение свойства font-weight согласно правилам, описанным в разделе "Уровень насыщенности". В одноименном дескрипторе можно объявлять любые значения, допустимые для свойства font-weight, за исключением inherit.

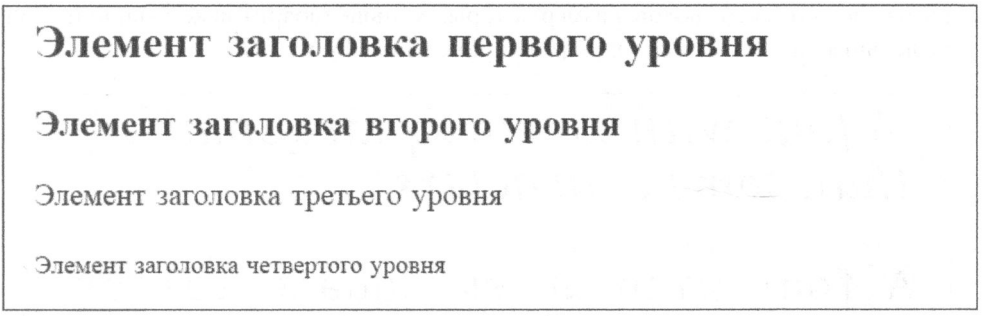


Рис. 5.11. Применение начертаний с заранее заданной насыщенностью

Размер шрифта

Для изменения размера шрифта применяется известное всем свойство, характеризующееся большим разнообразием предоставляемых возможностей.

font-size

Значение	`xx-small` | `x-small` | `small` | `medium` | `large` | `x-large` | `xx-large` | `smaller` | `larger` | *<length>* | *<percentage>*
Начальное значение	`medium`
Применяется	Все элементы
Процентное значение	Относительно размера шрифта родительского элемента
Вычисляется	Абсолютное значение размера
Наследуется	Да
Анимируется	Да (только числовые значения)

Значения `larger` и `smaller` свойства `font-size` применяются для установки размера шрифта относительно уровня родительского элемента, напоминая ключевые слова `bolder` и `lighter`, передаваемые свойству `font-weight` и описанные в предыдущем разделе. Как и в случае начертания, свойство `font-size` получает значение, наиболее близкое к расчетному на шкале допустимых размеров шрифтов. Таким образом, чтобы иметь представление о назначении ключевых слов `larger` и `smaller`, необходимо понимать, каким образом в шрифте определяется размер символов.

В сущности, взаимосвязь между значением свойства `font-size` и размером символов определяется дизайном шрифта. Как правило, размер шрифта сопоставим с величиной *кегельной площадки* (еще называемой габаритами литеры) для заданного шрифта. Величина кегельной площадки (а потому и шрифта) не зависит от размеров символов шрифта, а вычисляется как расстояние между базовыми линиями (линиями шрифта) соседних строк в отсутствие специально заданного интерлиньяжа (с помощью свойства `line-height` в CSS). Отдельные шрифты могут содержать знаки, высота которых превышает расстояние между базовыми линиями соседних строк, что делает их неудобными в использовании. Именно по этой причине большинство

шрифтов состоит из символов, размер которых меньше габаритов кегельной площадки. Отдельные примеры таких шрифтов приведены на рис. 5.12.

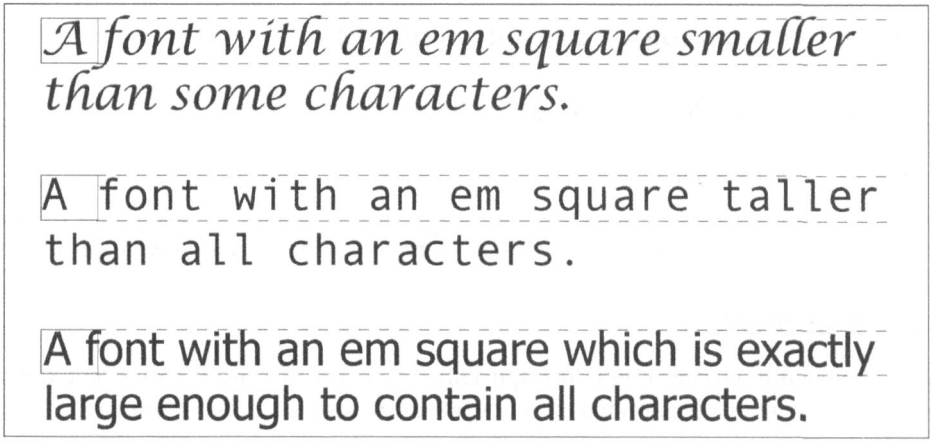


Рис. 5.12. Символы шрифта на фоне кегельной площадки

Итак, свойство `font-size` указывает размер кегельной площадки для заданного шрифта. Как уже упоминалось, далеко не все символы шрифта имеют этот размер.

Абсолютный размер шрифта

Определившись с тем, что такое размер шрифта, можно приступать к установке его абсолютного значения. В стилевых правилах абсолютный размер шрифта задается семью ключевыми словами: `xx-small`, `x-small`, `small`, `medium`, `large`, `x-large` и `xx-large`. Каждое из них указывает размер шрифта относительно остальных величин, а не точное числовое значение, как показано на рис. 5.13.

```
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, в числовом выражении отношение текущего абсолютного размера к предыдущему (*коэффициент масштабирования*) равно 1,5, а текущего размера к следующему — 0,66. Исходя из этого, ключевое слово `medium` соответствует числовому значению `10px`, а ключевое слово `large` — значению `15px`. В CSS2 коэффициент масштабирования был несколько уменьшен — до уровня 1,0–1,2, в зависимости от шрифта. В CSS3 вообще было предложено использовать прогрессивную шкалу изменения коэффициента масштабирования (например, в ней размер `small` составляет восемь девятых размера `medium`, а размер `xx-small` равен трем пятым этого значения). В любом случае предложенные правила пересчета абсолютных размеров шрифта рассматриваются пользовательскими агентами только как

рекомендуемые — коэффициент масштабирования может быть изменен без специального на то разрешения со стороны CSS.

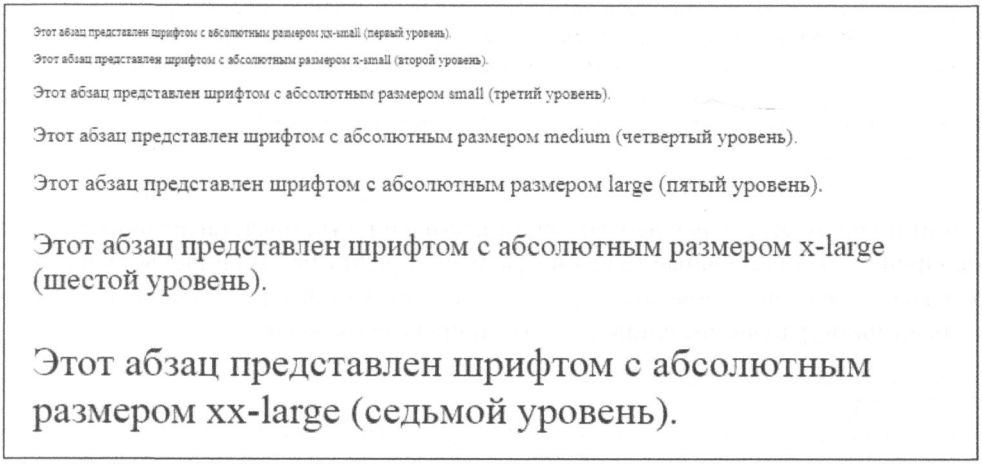


Рис. 5.13. Установка абсолютного размера шрифта

Значения размеров шрифтов, представляемые ключевыми словами свойства `font-size`, можно легко вычислить, исходя из предположения, что при разных коэффициентах масштабирования ключевое слово `medium` представляет числовое значение `16px`. Результаты пересчета, округленные до ближайшего целого, приведены в табл. 5.5.

Таблица 5.5. Значения абсолютных размеров шрифтов в CSS разных версий

Ключевое слово	CSS1	CSS2	CSS3
`xx-small`	`5px`	`9px`	`10px`
`x-small`	`7px`	`11px`	`12px`
`small`	`11px`	`13px`	`14px`
`medium`	`16px`	`16px`	`16px`
`large`	`24px`	`19px`	`19px`
`x-large`	`36px`	`23px`	`24px`
`xx-large`	`54px`	`28px`	`32px`

Относительный размер шрифта

Откровенно говоря, назначение ключевых слов `larger` и `smaller` очень простое: они указывают увеличить или уменьшить размер шрифта текущего элемента на один уровень относительно значения, заданного для родительского элемента, используя абсолютную шкалу измерений для применяемого пользовательским агентом коэффициента масштабирования. Иными словами, если вычисление абсолютных размеров шрифта ведется с использованием коэффициента масштабирования 1,2, то он должен применяться и при определении значений, представляемых ключевыми словами `larger` и `smaller`. Рассмотрим наглядный пример.

```
p {font-size: medium;}
strong, em {font-size: larger;}

<p>Этот абзац включает <strong>стильно акцентированный элемент,
содержащий <em>акцентированный элемент, в который вложен <strong>
еще один сильно акцентированный элемент.</strong></em></strong></p>

<p> medium <strong>large <em> x-large <strong>xx-large</strong>
</em> </strong>
</p>
```

В отличие от относительных значений насыщенности, представляющих начертание шрифта, относительные значения размера шрифта не ограничены диапазоном абсолютных значений свойства `font-size`. Это означает, что размер шрифта элемента может быть меньше значения `xx-small` и превышать значение `xx-large`.

```
h1 {font-size: xx-large;}
em {font-size: larger;}
<h1>Заголовок, содержащий <em>акцентированный</em> текстовый
    фрагмент</h1>
<p><em>Акцентированный текст</em> содержится также в этом абзаце.</p>
```

Как видно на рис. 5.14, выделенный в заголовке текст имеет размер, несколько превышающий размер текста самого заголовка. Степень увеличения размера шрифта определяется исключительно пользовательским агентом при рекомендуемом, но не обязательном коэффициенте масштабирования 1,2. Обратите внимание на то, что шрифт элемента `em`, добавленного в абзац, увеличен до следующего абсолютного размера (`large`).

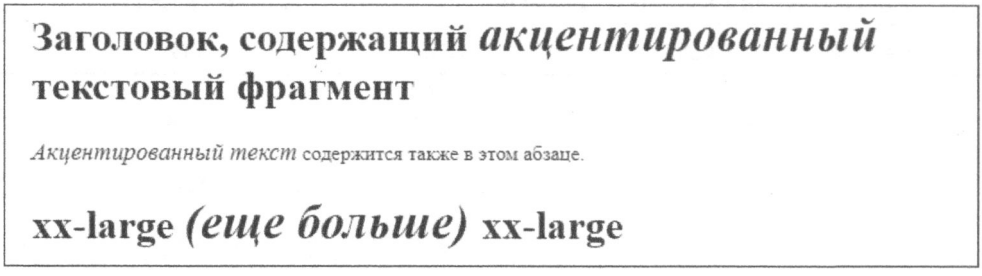


Рис. 5.14. Превышение предельного абсолютного размера при передаче свойству `font-size` относительного размера

Пользовательские агенты не обязаны устанавливать размер шрифта вне пределов, обозначенных абсолютными значениями.

Процентный размер шрифта

В известном смысле процентные размеры подобны относительным размерам. Процентный размер всегда рассчитывается как дольная часть значения, унаследованного от родительского элемента. Он, в отличие от абсолютного размера, описанного выше, позволяет указать размер шрифта предельно точно. Рассмотрим следующий пример, результат выполнения которого показан на рис. 5.15.

```
body {font-size: 15px;}
p {font-size: 12px;}
em {font-size: 120%;}
strong {font-size: 135%;}
small, .fnote {font-size: 70%;}

<body>
<p>Этот абзац содержит <em>акцентированный </em> и <strong>сильно
акцентированный </strong> текстовые фрагменты, размер шрифта которых
превышает таковой у родительского элемента. С другой стороны <small>
мелкий текст</small> представлен шрифтом в четыре раза меньшего
размера.</p>
<p class="fnote">Это сноска, поэтому размер ее шрифта меньше, чем
у обычного текста.</p>

<p> 12px <em> 14.4px </em> 12px <strong> 16.2px </strong> 12px
<small> 9px </small> 12px </p>
<p class="fnote"> 10.5px </p>
</body>
```

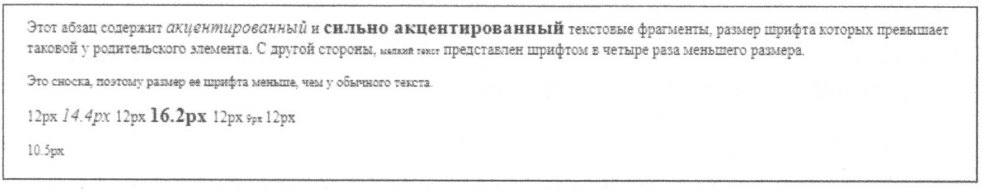


Рис. 5.15. Определение размеров шрифта процентными значениями

На рис. 5.15 показаны размеры шрифта, выраженные в пикселях. Они вычисляются браузером автоматически, независимо от действительных размеров символов, отображаемых на экране.

В данном случае процентный размер соответствует значению, выраженному в единицах `em`, в том понимании, что `1em` равно `100%`. Исходя из этого, можно утверждать, что два следующих правила устанавливают одинаковое форматирование абзацев.

```
p.one {font-size: 166%;}
p.two {font-size: 1.6em;}
```

К значениям, выраженным в единицах измерения `em`, применяются те же принципы обработки, что и к процентным размерам (наследование, пересчет в другие величины и т.п.).

Наследование размера шрифта

Согласно примеру, приведенному на рис. 5.15, дочерними элементами наследуется вычисляемое значение свойства `font-size`, но не процентный размер. В частности, элемент `strong` наследует размер `12px` от элемента абзаца, после чего к нему применяется модификатор `135%`. В результате для его свойства `font-size` определяется значение `16.2px`. Размер шрифта у абзаца сноски вычисляется перемножением значения свойства `font-size`, наследуемого от элемента `body`, и процентного размера `75%`. В итоге символы сноски отображаются шрифтом размером `11.25px`.

Как и значения относительного размера, процентные размеры подвержены кумулятивному эффекту. Наблюдать его можно на следующем простом примере, результат выполнения которого представлен на рис. 5.16.

```
p {font-size: 12px;}
em {font-size: 120%;}
strong {font-size: 135%;}

<p>Этот абзац содержит <em>акцентированный и <strong>сильно
акцентированный </strong></em> текстовые фрагменты, размер шрифта
которых больше, чем у обычного текста. </p>
<p>12px <em> 14.4px <strong> 19.44px </strong></em> 12px</p>
```

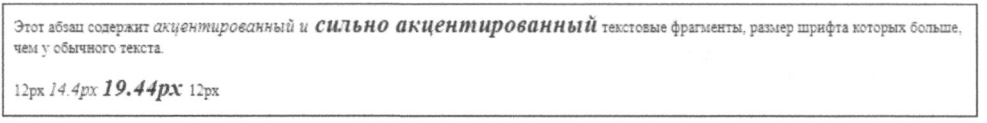


Рис. 5.16. К вопросу о наследовании размеров шрифта

Размер элемента `strong` (сильно акцентированный текст), показанного на рис. 5.16, вычисляется следующим образом:

`12px` × `120%` = `14.4px`

`14.4px` × `135%` = `19.44px` (при выводе на экране округляется до `19px`; см. следующий раздел)

Проблемы с масштабированием неизбежно возникают при наследовании размера шрифта элементами, имеющими множество предков. В качестве примера представим документ, содержащий неупорядоченные списки, глубина вложения которых друг в друга достигает четвертого уровня. Попробуем определить размер шрифта у таких списков при применении к ним следующего правила:

```
ul {font-size: 80%;}
```

Точный расчет показывает, что список самого глубокого, четвертого, уровня вложения будет иметь размер шрифта 40,96% от значения свойства `font-size`, установленного для списка самого верхнего уровня. Каждый следующий уровень вложения характеризуется уменьшением размера шрифта на 20%, поэтому список последнего уровня будет самым трудноразличимым.

Округление размера шрифта

Большинство современных браузеров округляет дробные значения свойства `font-size`, полученные в результате вычисления процентных размеров, поскольку они не всегда корректно обрабатываются механизмами визуализации пользовательских агентов. Изучим примеры, приведенные на рис. 5.17.

Рис. 5.17. Представление символов с дробными размерами шрифтов

Во всех строках каждый следующий символ "O" имеет размер, больший предыдущего на `0.1px`. К самому первому символу "O" (крайний левый) применяется свойство `font-size` со значением `10px`, средние символы имеют размер `10.5px`, а последние — размер `11px`.

Легко заметить, что результат визуализации символов с дробными размерами шрифта сильно зависит от браузера/операционной системы. Например, в MacOS браузеры Opera, Safari и Chrome скачкообразно увеличивают размер шрифта с `10px` до `11px`, как только вычисляемое значение превышает отметку `10.5px`. Такое же поведение свойственно Internet Explorer и Firefox для Windows. При этом в Firefox для MacOS символы выглядят одинаково, хотя на самом деле это не так — каждый следующий символ больше предыдущего в точности на оговоренные ранее `0.1px`. В этом сложно убедиться без проведения точных измерений, хотя различие между крайними символами не вызывает особых сомнений.

Как бы там ни было, округление размера шрифта выполняется только на этапе визуализации и ни в коей мере не касается значения свойства `font-size`. Чтобы удостовериться в этом, достаточно запросить его из объектной модели документа или просмотреть с помощью инспектора стилей. Крайний правый символ "O" будет иметь размер шрифта `10.8px` независимо от величины буквы на экране монитора.

Моноширинный текст

Интересный визуальный эффект наблюдается при изменении размера символов моноширинного текста (например, представленных шрифтом Courier). Рассмотрим пример применения приведенных ниже правил к документу, состоящему из единственного абзаца (рис. 5.18).

```
p {font-size: medium;}  /* значение по умолчанию */
span {font-family: monospace; font-size: 1em;}

<p>Элемент p, включающий элемент <span>span</span>.</p>
```

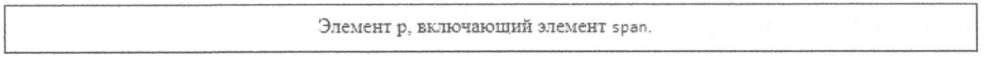


Рис. 5.18. Несуразность моноширинного текста

По умолчанию (в отсутствие вмешательства пользователя в настройки браузера) для вывода текста применяются шрифты с абсолютным размером `medium`, который соответствует числовому значению `16px`. И действительно, если воспользоваться инспектором стилей, то можно обнаружить, что к тексту, расположенному вне элемента `span`, применяется свойство `font-size` с вычисленным значением `16px` (опять-таки, при сохранении настроек браузера по умолчанию).

Стоит ожидать, что у моноширинного текста размер шрифта также составляет `16px`. Как ни странно, но это утверждение справедливо не для всех браузеров — у многих из них размер шрифта устанавливается на уровне `13px`.

Такое поведение пользовательских агентов не удивительно, поскольку элемент `span` наследует размер шрифта не в виде числового значения `16px`, а как абсолютную величину, представленную ключевым словом `medium`. Для определения действительного (числового) значения браузер обращается к настройкам по умолчанию, в которых указывается отображать моноширинный текст шрифтом с базовым размером `13px`. Если эта настройка не была изменена пользователем, то любой моноширинный текст будет иметь размер символов 13 пикселей, даже если к нему применяется объявление `font-size: 1em`, а родительский элемент представляется шрифтом размером `16px`.

Указанное выше отличие сохраняется даже при установке размеров шрифтов по умолчанию, отличных от `1em` (`100%`). В следующем случае моноширинный шрифт имеет размер `26px`, а не `32px`, характерный для обычного шрифта (как и раньше, не забываем учитывать настройки браузера).

```
p {font-size: medium;}  /* значение по умолчанию */
span {font-family: monospace; font-size: 2em;}

<p>Элемент p, содержащий элемент <span>span</span>.</p>
```

Не забывайте, что некоторые браузеры заимствуют значение `medium` для моноширинных шрифтов у регулярных шрифтов, пренебрегая уменьшением их размера в дочерних элементах. Такое поведение становится частой причиной кроссбраузерной несовместимости документов.

Для устранения этой проблемы приходится идти на определенное ухищрение, работающее во всех популярных браузерах.

```
p {font-size: medium;}  /* значение по умолчанию */
span {font-family: monospace, serif; font-size: 1em;}

<p>Элемент p, содержащий элемент <span>span</span>.</p>
```

Обратите внимание на ключевое слово `serif` в значении свойства `font-family`. По неизвестной причине оно обязывает браузер рассматривать значение `1em` свойства `font-size` не как размер, задаваемый ключевым словом `medium`, а как вычисляемую величину, представляющую размер шрифта родительского элемента. Включение его в правило позволит унифицировать вид элемента `span` в любых браузерах (рис. 5.19).

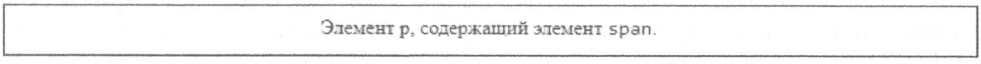


Рис. 5.19. Выравнивание размеров шрифтов разных типов

Размер шрифта в единицах измерения длины

Значение свойства `font-size` можно выражать в единицах измерения длины. Таким образом, все приведенные ниже правила устанавливают одинаковое форматирование.

```
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;}
```

Результат их применения к тексту документа показан на рис. 5.20. Он справедлив только для систем, в которых пользовательскому агенту известно о разрешении экрана, на который выводится документ. Далеко не все пользовательские агенты ориентируются на него — зачастую базовое разрешение запрашивается у операционной системы, устанавливается настройками браузера или задается с помощью значения по умолчанию, определенного на этапе разработки браузера. В любом случае все пять текстовых строк будут представляться шрифтом одинакового размера, хотя он не всегда будет соответствовать действительным единицам измерения (например, размер символов третьей строки может отличаться на 0,5 дюйма).

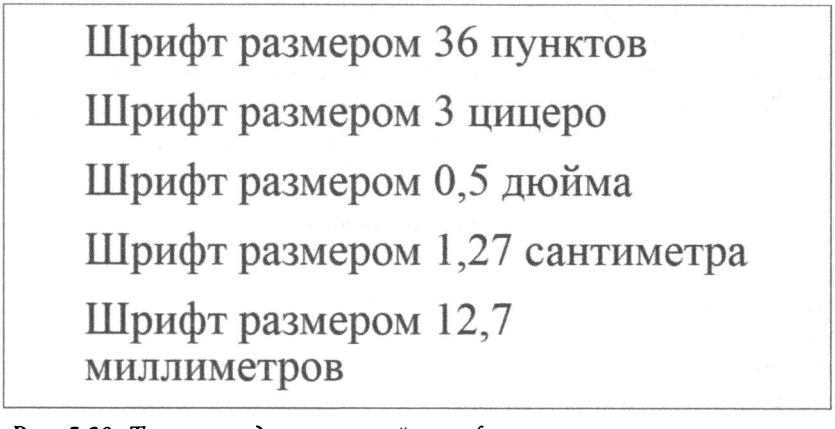


Рис. 5.20. Текст, представленный шрифтами с размерами, которые задаются разными единицами измерения

Из всех представленных на рис. 7.20 текстовых строк только первая (размер шрифта которой `36px`) имеет шансы отображаться в реальном размере. Чтобы это произошло, документ должен выводиться на носитель с разрешением 72 ppi, которое редко устанавливается в современных устройствах. Компьютерные мониторы обычно настраиваются на разрешение от 96 до 120 ppi, а мобильные устройства — и того больше: от 300 до 500 ppi.

Невзирая на широкий разброс рабочих разрешений целевых устройств, многие разработчики продолжают указывать размер шрифта в пикселях. Чаще всего такой подход оправдан при наполнении документа небольшими растровыми изображениями (GIF, JPG, PNG и т.п.), имеющими одинаковую с текстовыми строками высоту (рис. 5.21). Достаточно объявить размер шрифта в пикселях, чтобы обеспечить единство размеров текста и изображений в системах с разным разрешением экрана.

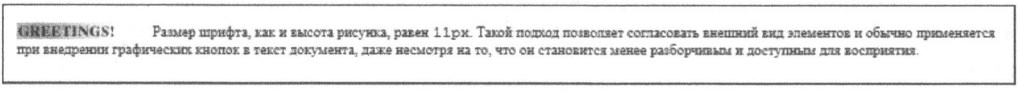


Рис. 5.21. Пропорциональное масштабирование изображений и текста при изменении разрешения экрана

Как видите, выражая значение свойства `font-size` в пикселях, можно добиться единства размеров текста и изображений в самых разных средах, но он не лишен недостатков. К сожалению, далеко не все браузеры умеют масштабировать текст, размер которого изменяется в пикселях (или делают это некорректно). Чаще всего такая проблема встречается в мобильных устройствах, работающих преимущественно в полноэкранном режиме (в частности, iPhone). Именно эта причина побуждает отказываться от задания размера шрифта в пикселях в пользу других единиц измерения.

Автоматическое изменение размера шрифта

Разборчивость или удобочитаемость текста зависит от двух факторов: размера шрифта и высоты строчных символов. Отношение высоты строчных символов шрифта к его размеру, определяемому свойством `font-size`, называется *аспектом шрифта*. Шрифты с высоким аспектом лучше подходят для отображения даже мелкого текста. А вот шрифты с низким аспектом подходят для вывода текста только с крупными символами. Для повышения разборчивости текста в CSS применяется свойство `font-size-adjust`, корректирующее аспект шрифта.

font-size-adjust

Значение	*<number>* | `none` | `auto`
Начальное значение	`none`
Применяется	Все элементы
Наследуется	Да
Анимируется	Да

Главное предназначение этого свойства состоит в поддержании удобочитаемости текста на заданном уровне при отображении его шрифтами, отличными от предписанных автором. Обычно трудности возникают, когда шрифт, заменяющий шрифт автора, содержит символы, которые плохо или совсем неразличимы при том же значении свойства `font-size`.

Для иллюстрации описанного выше недостатка сравним шрифты Verdana и Times. На рис. 5.22 показано, как выглядит текст при форматировании его указанными шрифтами с одним и тем же размером: `10px`.

```
p {font-size: 10px;}
p.cl1 {font-family: Verdana, sans-serif;}
p.cl2 {font-family: Times, serif;}
```

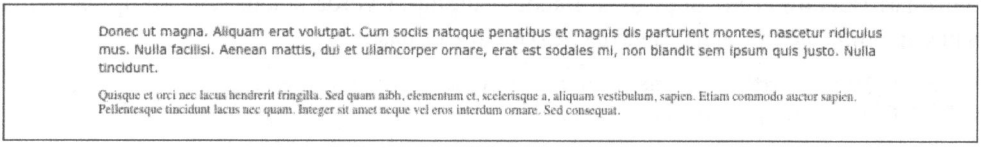


Рис. 5.22. Сравнение шрифтов Verdana и Times

Очевидно, что текст, представленный шрифтом Times, читать намного сложнее, но не только из-за некорректного масштабирования символов. Особенностью шрифта Times является плохая различимость символов небольших размеров.

Как оказалось, у Verdana отношение высоты строчных символов к размеру шрифта составляет 0,58, а у Times оно равно 0,46. Чтобы повысить удобочитаемость текста, отображаемого шрифтом Times, необходимо повысить его аспект до уровня, свойственного шрифту Verdana. Пользовательский агент автоматически вычислит требуемый размер шрифта Times, исходя из такой формулы:

> значение свойства `font-size` шрифта автора × (аспект шрифта автора ÷ аспект заменяющего шрифта) = значение свойства `font-size` заменяющего шрифта.

В случае замены шрифта Verdana шрифтом Times получаем следующую зависимость:

`10px × (0.58 ÷ 0.46) = 12.6px`

В коде CSS указанные преобразования выполняются автоматически (рис. 5.23). В правиле достаточно указать правильный аспект шрифта автора.

```
p {font: 10px Verdana, sans-serif; font-size-adjust: 0.58;}
p.cl2 {font-family: Times, serif;}
```

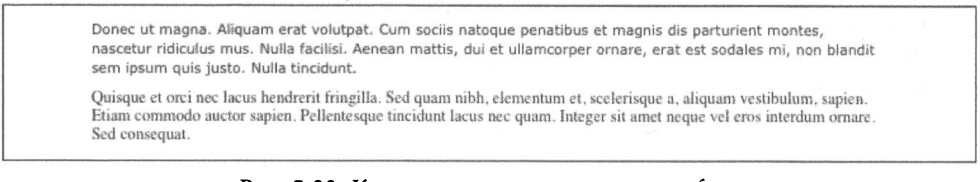


Рис. 5.23. Корректировка размера шрифта

Чтобы браузер правильно пересчитал размер используемого им шрифта, необходимо правильно указать значение свойства `font-size-adjust` для шрифта автора. Пользовательские агенты, распознающие команду `@font-face`, получают сведения об аспекте шрифта из его файла (предполагается, что файл шрифта их содержит, — большинство шрифтов, разработанных профессиональными дизайнерами, в обязательном порядке включают данные об аспекте шрифта). Если аспект шрифта не указан в его файле, то пользовательский агент, скорее всего, попробует вычислить его. Тем не менее полностью полагаться на полученное им значение все же не стоит.

Если аспект шрифта автора не известен, а браузер прекрасно справляется с его автоматическим определением (извлекая из файла или вычисляя самостоятельно), то свойству `font-size-adjust` можно передать значение `auto`. В частности, если пользовательский агент определяет аспект шрифта Verdana как 0,58, то приведенные выше правила можно переписать следующим образом.

```
p {font: 10px Verdana, sans-serif; font-size-adjust: auto;}
p.cl2 {font-family: Times, serif; }
```

В результате их применения текст получит форматирование, показанное на рис. 5.23.

Задав для свойства `font-size-adjust` значение `none`, можно запретить корректировку размера заменяющего шрифта. Оно устанавливается по умолчанию.

К концу 2017 года свойство `font-size-adjust` поддерживали только браузеры семейства Firefox (основанные на движке Gecko).

Начертание шрифта

Свойство `font-style` очень простое и указывает начертание шрифта: `normal` (обычное), `italic` (курсивное) или `oblique` (наклонное). Трудности могут возникать только при определении различия между курсивным и наклонным начертаниями, а также при поиске причины отсутствия одного из них во многих шрифтах.

font-style

Значение	`italic` | `oblique` | `normal`
Начальное значение	`normal`
Вычисляется	См. описание
Применяется	Все элементы
Примечание	Снабжено отдельным дескриптором в команде `@font-face`
Наследуется	Да
Анимируется	Нет

Из определения свойства `font-style` видно, что по умолчанию оно имеет значение `normal`. В обычном начертании символы располагаются вертикально, в отличие от курсивного и наклонного начертания. К примеру, большая часть текста данной книги представлена обычным начертанием. Понять различие между курсивным и наклонным начертаниями проще всего на примере, приведенном на рис. 5.24.

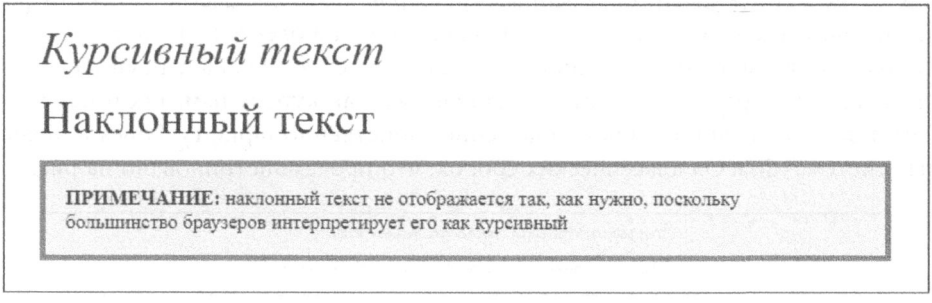


Рис. 5.24. Курсивное и наклонное начертания крупным планом

Строго говоря, курсивное начертание представляется отдельным файлом, содержащим несколько видоизмененные символы, представляющие знаки исходного шрифта в наклонном виде. Особенно это касается шрифтов с засечками, в которых символы не только становятся тоньше, но и меняют форму штрихов. С другой стороны, в наклонном начертании символы сохраняют исходную форму и толщину, хотя и выглядят скошенными. В названия курсивных начертаний обычно добавляются слова "Italic" или "Cursive", в то время как в названиях наклонных начертаний содержатся слова "Oblique", "Slanted" или "Incline".

Чтобы убедиться в существовании курсивного начертания для выбранного шрифта, нужно использовать такие простые стили.

```
p {font-style: normal;}
em, i {font-style: italic;}
```

С их помощью текст абзацев выводится в обычном, а элементы `em` и `i` — в курсивном начертании. В то же время последним двум элементам можно задать несколько иное форматирование.

```
p {font-style: normal;}
em {font-style: oblique;}
i {font-style: italic;}
```

Внимательно рассмотрев рис. 5.25, легко удостовериться, что элементы `em` и `i` выглядят совершенно одинаково. В действительности очень небольшое количество шрифтов включают и курсивное, и наклонное начертания. Но даже в таких случаях браузеры представляют оба начертания преимущественно одинаковыми символами.

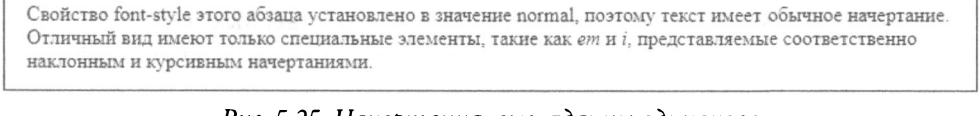


Рис. 5.25. Начертания, выглядящие одинаково

Наряду с этим необходимо учитывать следующие возможности. В случае отсутствия в шрифте курсивного начертания оно, скорее всего, будет заменено наклонным, и наоборот. Замена осуществляется далеко не всегда — все зависит от предпочтений, заложенных в браузер его разработчиками. Более того, отдельные браузеры умеют генерировать наклонное начертание для большинства обычных начертаний. Технические возможности современных браузеров позволяют легко изменять форму символов шрифта, что делает последний сценарий наиболее вероятным.

К тому же во многих операционных системах текст, обозначенный как `italic`, в зависимости от размера может представляться как курсивным, так и наклонным начертанием. В частности, такое поведение свойственно шрифту Times в операционных системах MacOS классических сборок, что продемонстрировано на рис. 5.26.

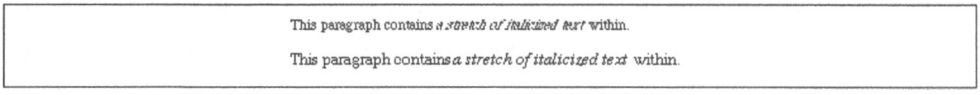


Рис. 5.26. Тот же шрифт другого размера, представленный таким же начертанием

Изменить способ отображения браузером, а особенно операционной системой, курсивного начертания отдельно взятого шрифта не представляется возможным. К счастью, все современные операционные системы (Mac OS и Windows) оснащены превосходными механизмами визуализации шрифтов. Не забывайте также о команде `@font-face`, которая позволяет оговаривать файлы шрифтов, используемые при отображении курсивного и наклонного начертаний, устанавливаемых свойством `font-style`.

Даже в случаях представления курсивного и наклонного начертаний одним и тем же файлом свойство `font-style` нельзя списывать со счетов. Например, согласно общепринятому типографскому соглашению текст цитат выводится курсивным начертанием, а любой содержащийся в них акцентированный текст — обычным начертанием. Для получения указанного эффекта (рис. 5.27) можно использовать следующие простые правила.

```
blockquote {font-style: italic;}
blockquote em, blockquote i {font-style: normal;}
```

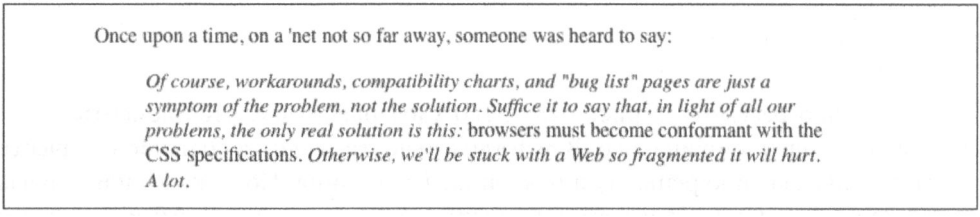


Рис. 5.27. Общепринятый способ форматирования текста цитат

Для предотвращения синтезирования браузером собственных начертаний значения свойства `font-style` нужно привязать к строго заданным файлам шрифтов.

Дескриптор font-style

Дескриптор font-style позволяет привязывать файлы шрифтов к начертаниям, указываемым с помощью одноименного свойства.

```
@font-face {
   font-family: "SwitzeraADF";
   font-style: normal;
   src: url("SwitzeraADF-Regular.otf") format("opentype");
}
@font-face {
   font-family: "SwitzeraADF";
   font-style: italic;
   src: url("SwitzeraADF-Italic.otf") format("opentype");
}
@font-face {
   font-family: "SwitzeraADF";
   font-style: oblique;
   src: url("SwitzeraADF-Italic.otf") format("opentype");
}
```

Задекларировав начертания указанным выше способом, можно добиться автоматической замены шрифта SwitzeraADF-Italic на SwitzeraADF-Regular для элементов h2 и h3 (рис. 5.28).

```
h1, h2, h3 {font: 225% SwitzeraADF, Helvetica, sans-serif;}
h2 {font-size: 180%; font-style: italic;}
h3 {font-size: 150%; font-style: oblique;}
```

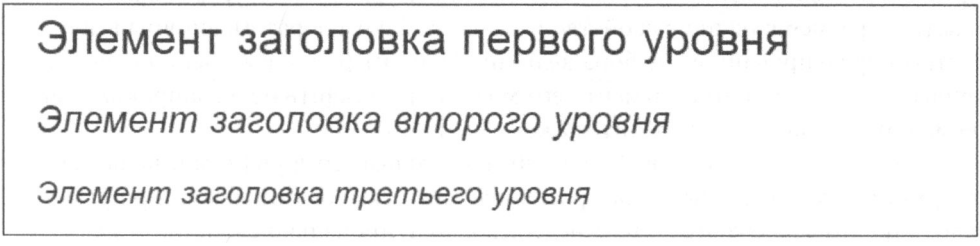


Рис. 5.28. Применение задекларированных начертаний

В идеальном случае шрифт SwitzeraADF должен содержать все возможные начертания, включая наклонное. Тогда его можно использовать вместо курсивного начертания. К сожалению, в шрифте SwitzeraADF такое начертание отсутствует, поэтому значения italic и oblique сопоставляются с одним и тем же файлом шрифта. Дескриптор font-style, как и font-weight, позволяет указывать любые предпочтительные значения одноименного свойства, за исключением inherit.

Уплотнение и расширение текста

Отдельные семейства шрифтов включают начертания, символы которых уплотнены или расширены по отношению к символам обычного начертания. В названиях таких начертаний встречаются слова "Condensed", "Wide", "Ultra Expanded" и т.п. Существование подобных начертаний позволяет обходиться в документе небольшим количеством шрифтов. Теперь ширину символов шрифта можно устанавливать с помощью специальных начертаний (если, конечно, они представлены в указанном шрифте), избегая объявления специальных шрифтов в свойстве `font-family`. В CSS сжатые и растянутые начертания определяются с помощью свойства `font-stretch`.

font-stretch

Значение	`normal` | `ultra-condensed` | `extra-condensed` | `condensed` | `semi-condensed` | `semi-expanded` | `expanded` | `extra-expanded` | `ultra-expanded`
Начальное значение	`normal`
Применяется	Все элементы
Примечание	Снабжено отдельным дескриптором в команде `@font-face`
Наследуется	Да
Анимируется	Нет

Исходя из названия свойства можно предположить, что оно обеспечивает произвольное изменение ширины символов, но в действительности это не так. Значения, передаваемые свойству, ведут себя подобно ключевым словам, устанавливающим абсолютный размер шрифта в свойстве `font-size`. Таким образом, ширина выбирается из строго фиксированного набора величин. Например, для выделения особо акцентированного текстового фрагмента его можно представить более широким начертанием, чем основной текст (родительского элемента).

Как уже упоминалось, свойство работает только со шрифтами, включающими начертания с уплотненными или расширенными символами. Таких шрифтов очень мало, а те, что встречаются, стоят недешево. Функционально свойство `font-stretch` сильно отличается от свойства `font-size`, позволяющего изменять размер шрифта в широких пределах. Например, установка свойства `font-stretch` в значение `expanded` приведет к должному эффекту только тогда, когда оговоренное семейство шрифтов содержит начертание с расширенными символами. Если таковое в шрифте отсутствует, то начертание текста элемента, к которому применяется правило, не изменится.

В качестве примера рассмотрим шрифт Verdana, обладающий начертанием только одного типа, представляемым свойством `font-stretch: normal`. Следовательно, приведенные ниже объявления не изменят ширину символов целевого текста.

```
body {font-family: Verdana;}
strong {font-stretch: extra-expanded;}
footer {font-stretch: extra-condensed;}
```

Символы текста сохранят исходную ширину, свойственную шрифту Verdana единственно возможного начертания. Тем не менее при замене Verdana шрифтом, включающим несколько начертаний с разной шириной символов, правила работают так, как предполагалось исходно (рис. 5.29).

```
body {font-family: Futura;}
strong {font-stretch: extra-expanded;}
footer {font-stretch: extra-condensed;}
```

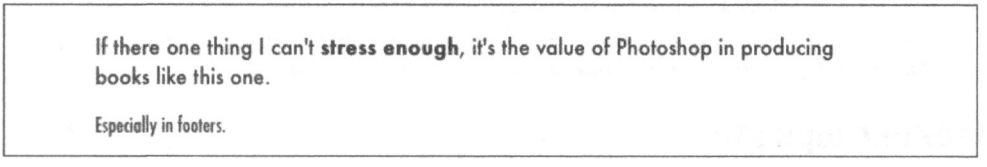


Рис. 5.29. Изменение ширины символов

К концу 2017 года свойство `font-stretch` не поддерживали браузеры Opera Mini и Safari для Mac Os и iOS.

Дескриптор `font-stretch`

Дескриптор `font-stretch`, как и `font-weight`, указывает допустимые значения для одноименного свойства. Они представляют начертания с разной шириной символов, которые разрешается передавать свойству `font-stretch`. Так, например, следующие правила определяют три начертания, символы которых отличаются только шириной.

```
@font-face {
   font-family: "SwitzeraADF";
   font-stretch: normal;
   src: url("SwitzeraADF-Regular.otf") format("opentype");
}
@font-face {
   font-family: "SwitzeraADF";
   font-stretch: condensed;
   src: url("SwitzeraADF-Cond.otf") format("opentype");
}
@font-face {
   font-family: "SwitzeraADF";
   font-stretch: expanded;
   src: url("SwitzeraADF-Ext.otf") format("opentype");
}
```

Все три начертания можно применить к документу (рис. 5.30), правильно указав их в объявлении свойства `font-stretch`, как показано в предыдущем разделе.

```
h1, h2, h3 {font: 225% SwitzeraADF, Helvetica, sans-serif;}
h2 {font-size: 180%; font-stretch: condensed;}
h3 {font-size: 150%; font-stretch: expanded;}
```

Как и другие дескрипторы, `font-stretch` устанавливает допустимые значения одноименного с ним свойства, *кроме* `inherit`.

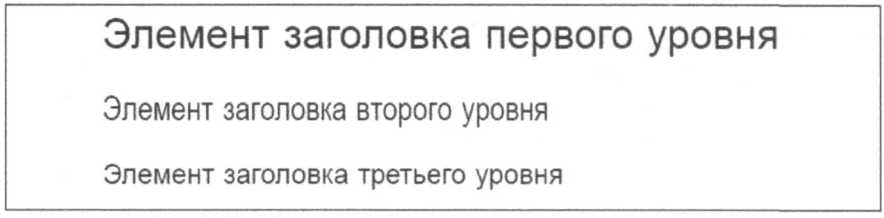


Рис. 5.30. Применение начертаний, объявленных для свойства `font-stretch`

Кернинг шрифта

В некоторых шрифтах разрешается изменять *кернинг*. Под кернингом подразумевают изменение интервала между определенными сочетаниями символов, обеспечивающее правильную общую плотность символов в тексте. Величина кернинга зависит от символов, интервал между которыми изменяется. В частности, разный кернинг имеют пары символов *oc* и *ox*. Подобным образом, разный интервал задается для кернинговых пар *AB* и *AW* — во многих шрифтах левый верхний штрих буквы *W* часто располагается над правым нижним штрихом буквы *A*. Установка и отключение кернинга в шрифте выполняется свойством `font-kerning`.

font-kerning

Значение	`auto` | `normal` | `none`
Начальное значение	`auto`
Применяется	Все элементы
Наследуется	Да
Анимируется	Нет

Значение `none` говорит само за себя — оно указывает пользовательскому агенту игнорировать настройки кернинга в шрифте. При передаче свойству значения `normal` в тексте применяются кернинговые пары, установленные в самом шрифте. И только при задании значения `auto` решение об интервале между отдельными парами символов принимается пользовательским агентом на основании множества факторов, и в первую очередь оно зависит от типа шрифта. К слову, спецификация OpenType рекомендует (но не обязывает) применять кернинг во всех шрифтах, в которых он задан. Если кернинговые пары в самом шрифте не заданы, то свойство `font-kerning` не возымеет эффекта.

Если к тексту с кернингом применить свойство `letter-spacing`, то оно будет устанавливать общий интервал между символами *после* настройки кернинга, а не наоборот.

Варианты строчных букв

Помимо размера, насыщенности, начертания и некоторых других параметров, в шрифте можно указывать альтернативный способ отображения символов, в первую очередь строчных букв. Все допустимые варианты строчных букв включаются в файл шрифта наряду с лигатурами, символами дробей, разделителями числовых значений, перечеркнутыми нулями и т.п. Если шрифт содержит все или часть перечисленных выше специальных символов, то они могут подключаться или отключаться в тексте с помощью свойства `font-variant`.

font-variant

Значение (CSS2.1)	`normal` | `small-caps`
Значение (CSS3)	`normal` | `none` | [*<common-lig-values>* || *<discretionary-lig-values>* || *<historicallig-values>* || *<contextual-alt-values>* || `stylistic(`*<feature-value-name>*`)` || `historical-forms` || `styleset(`*<feature-value-name>*`#)` || `character-variant (`*<feature-value-name>*`#)` || `swash(`*<feature-value-name>*`)` || `ornaments(`*<feature-value-name>*`)` || `annotation(`*<feature-value-name>*`)` || [`smallcaps` | `all-small-caps` | `petite-caps` | `all-petite-caps` | `unicase` | `titling-caps`] || *<numeric-figure-values>* || *<numeric-spacing-values>* || *<numeric-fraction-values>* || `ordinal` || `slashed-zero` || *<east-asian-variantvalues>* || *<east-asian-width-values>* || `ruby`]
Начальное значение	`normal`
Применяется	Все элементы
Примечание	Снабжено отдельным дескриптором в команде `@font-face`
Вычисляется	Согласно значению
Наследуется	Да
Анимируется	Нет

Свойство обладает чрезвычайно обширным списком значений в CSS3, особенно если учесть, что в CSS1 и CSS2 оно поддерживало всего два значения: `normal` (обычный текст) и `small-caps` (капитель). Все они требуют подробного рассмотрения.

Текст, представленный прописными буквами несколько меньшего размера, чем обычно, называется *капителью*. Для создания капители применяется методика, подобная описанной ниже (рис. 5.31).

```
h1 {font-variant: small-caps;}
h1 code, p {font-variant: normal;}

<h1>Назначение свойства <code>font-variant</code></h1>
<p>
У свойства <code>font-variant</code> очень интересная...
</p>
```

Рис. 5.31. Капитель в заголовке

Изучая внешний вид элемента `h1`, становится очевидным, что в капители все прописные символы остаются прописными, а строчные преобразуются в прописные уменьшенного размера. Подобного результата можно добиться с помощью объявления `text-transform: uppercase;`, за тем лишь исключением, что в данном случае используются прописные символы разных размеров. Как бы там ни было, значение `small-caps` не обязывает преобразовывать строчные символы в прописные, а указывает представлять их прописными символами специального размера, встроенными в шрифт (при наличии такого начертания в семействе).

А как поступает пользовательский агент в отсутствие начертания, содержащего малые прописные символы? Существуют два равнозначных решения. В первом малые прописные символы получаются преобразованием прописных символов обычного размера. Второе решение заключается в представлении малых прописных символов обычными прописными символами, как в случае применения объявления `text-transform: uppercase;`. Конечно, последний вариант далеко не самый лучший, но в отсутствие других решений остается единственно приемлемым.

Остальные значения свойства `font-variant`

Ознакомившись с базовыми вариантами, можно приступать к рассмотрению значений свойства `font-variant`, добавленных в него спецификацией CCS3. Их превеликое множество, и чтобы понять, зачем они все нужны, необходимо знать, как образуется каждое из них. В сущности, ничего сложного в ключевых словах нет — они всего лишь представляют допустимые значения следующих свойств:

- `font-variant-ligatures`
- `font-variant-caps`
- `font-variant-numeric`
- `font-variant-alternates`
- `font-variant-east-asian`

Так, например, значение *`<common-lig-values>`* заимствуется у свойства `font-variant-ligatures` (представляется ключевым словом `common-ligatures` или `no-common-ligatures`), а значение *`<numeric-fraction-values>`* — у свойства `font-variant-numeric` (представляется ключевым словом `diagonal-fractions` или `stacked-fractions`) и т.п.

Передача свойству `font-variant` любых заявленных в определении значений зависит от двух факторов: поддержки их браузерами и включения дополнительных

начертаний в шрифты. Первое ограничение достаточно простое — к концу 2017 года новые значения не приобрели широкой поддержки в браузерах. Если первые два ключевых слова распознаются и обрабатываются большинством известных пользовательских агентов, то остальные значения поддерживаются только браузерами семейств Gecko и WebKit.

Намного сложнее обстоит дело с начертаниями, обеспечивающими шрифты дополнительными возможностями. У каждого шрифта свой набор символов, а потому и альтернативных начертаний. В частности, шрифты, включающие символы европейских языков, не включают знаки азиатской письменности. Кроме того, далеко не все шрифты содержат лигатуры и несколько вариантов написания порядковых числительных. Сведения о поддерживаемых шрифтом символах приведены в его документации. Если документация недоступна, то состав шрифта придется определять экспериментальным путем. (Чаще всего документацией снабжаются только шрифты, распространяемые на коммерческой основе, а бесплатные шрифты ею не комплектуются.)

Не забывайте, что поддержка браузером одного из альтернативных начертаний не означает, что он поддерживает остальные значения свойства `font-variant`. К тому же добавление в шрифт альтернативного начертания не гарантирует автоматического распознавания его браузерами. Как видите, все не столь однозначно, как предполагалось, поэтому при использовании специальных начертаний будьте предусмотрительны.

Свойства группы `font-variant-*` в книге не рассматриваются, поскольку на момент ее написания ни одно из них не поддерживалось браузерами. Детально о поддержке альтернативных начертаний можно узнать по адресу `http://w3.org/TR/css3-fonts/`.

Дескриптор `font-variant`

Дескриптор `font-variant` позволяет указывать альтернативные начертания, применяемые при форматировании документа. Например, следующее объявление разрешает применять лигатуры, капитель и перечеркнутые нули:

```
font-variant: common-ligatures small-caps slashed-zero;
```

Вне всяких сомнений, дескриптор `font-variant` рассчитан на работу со всеми значениями свойства `font-variant`, *за исключением* `inherit`.

При всем этом функционально `font-variant` сильно отличается от дескрипторов, рассмотренных ранее. В частности, если дескриптор `font-stretch` указывает начертание, представляемое тем или иным значением одноименного свойства, то дескриптор `font-variant` определяет альтернативные наборы символов для начертания, объявляемого в правиле `@font-face`, которые могут не совпадать с указываемыми свойством `font-variant`. Например, при отображении текста следующих абзацев символы, относящиеся к альтернативным начертаниям `diagonal-fractions` и `small-caps`, не применяются, даже при включении в шрифт SwitzeraADF.

```
@font-face {
   font-family: "SwitzeraADF";
   font-weight: normal;
   src: url("SwitzeraADF-Regular.otf") format("opentype");
   font-variant: stacked-fractions titling-caps slashed-zero;
}

p {font: small-caps 1em SwitzeraADF, sans-serif;
   font-variant-numeric: diagonal-fractions;}
```

Особенности шрифта

Дескриптор `font-feature-settings`, как и `font-variant`, обеспечивает низкоуровневый контроль над шрифтами OpenType, указывая особенности, которые нужно подключать при отображении текста документа. Обратите внимание: он не применяется к шрифтам, сохраненным в формате `.woff`.

font-feature-settings

Значение	`normal` | *<feature-tag-value>#*
Начальное значение	`normal`
Примечание	Снабжено отдельным дескриптором в команде `@font-face`

Дескриптору допускается передавать сразу несколько значений, представляющих отдельные особенности OpenType и разделяемых запятыми. Например, чтобы разрешить выводить текст документа с использованием лигатур, малых прописных символов и перечеркнутых нулей, применяется такое правило:

```
font-feature-settings: "liga" on, "smcp" on, "zero" on;
```

В общем виде особенность OpenType указывается значением, выражаемым в следующем формате.

```
<feature-tag-value>
   <string> [ <integer> | on | off ]?
```

Большинство особенностей OpenType представляется целочисленными значениями `0` или `1`, эквивалентными параметрам `off` или `on`. Только некоторые из них сопоставляются с диапазоном числовых значений. В подобных случаях значение, большее 1, одновременно активизирует особенность и устанавливает ее величину. Особенности, указанные без числового значения, рассматриваются как разрешенные (имеющие значение `1` или `on`). Следовательно, все три приведенных ниже объявления абсолютно равнозначны.

```
font-feature-settings: "liga";      /* предполагается 1 */
font-feature-settings: "liga" 1;    /* объявлено 1 */
font-feature-settings: "liga" on;   /* on = 1 */
```

Не забывайте, что строковые значения необходимо заключать в кавычки. Таким образом, в следующем объявлении распознается только первое значение, а второе игнорируется.

```
font-feature-settings: "liga", dlig;
/* обычные лигатуры разрешены; дискретные лигатуры не определены
   в отсутствие кавычек */
```

Следующее ограничение связано с необходимостью представления особенностей OpenType лишь четырехбуквенными строковыми значениями, состоящими только из символов ASCII. Значения, имеющие иную длину или включающие символы, которые отсутствуют в кодовой странице ASCII, считаются недопустимыми и не распознаются браузерами. (Волноваться стоит только при использовании "самодельных" шрифтов, создаваемых в редакторах, которые не придерживаются общепринятых правил именования особенностей OpenType.)

В шрифтах OpenType следующие особенности включены по умолчанию. Их отключение в CSS выполняется с помощью всего двух дескрипторов: `font-feature-settings` и `font-variant`.

`calt`

Контекстные формы.

`ccmp`

Составные символы.

`clig`

Контекстные лигатуры.

`liga`

Обычные лигатуры.

`locl`

Локализованные формы.

`mark`

Позиционирование знаков относительно базовой линии.

`mkmk`

Позиционирование знаков относительно знаков.

`rlig`

Обязательные лигатуры.

В дополнение к приведенным выше особенностям в шрифтах OpenType по умолчанию могут устанавливаться также вертикальные лигатуры и альтернативы.

С полным перечнем особенностей шрифтов OpenType можно ознакомиться по следующему адресу:

`https://docs.microsoft.com/ru-ru/typography/opentype/spec/featurelist`

Дескриптор `font-feature-settings`

Как вы уже знаете, дескриптор `font-feature-settings` применяется для указания особенностей шрифта OpenType, которые могут использоваться при стилевом форматировании текстов документа. Особенности указываются в виде четырехбуквенных ASCII-значений, разделенных запятыми.

А как же быть с дескриптором `font-variant`, рассмотренным в предыдущем разделе, ведь он устанавливает даже большее количество особенностей шрифтов OpenType? В действительности противоречий между дескрипторами нет, поскольку `font-variant` принимает значения в виде ключевых слов CSS, а не четырехбуквенного ASCII-кода или булевых величин. Именно по этой причине спецификация CSS оговаривает применять дескрипторы `font-feature-settings` только при настройке особенностей, не передаваемых значениями дескриптора `font-variant`.

Помните, что дескрипторы определяют только возможность или невозможность использования особенностей шрифта при отображении текста. Они не применяют и не отменяют их в документе — для этих целей служит свойство `font-feature-settings`.

Дескриптор `font-feature-settings`, подобно `font-variant`, указывает особенности, которые можно использовать при выводе текста с помощью начертания, определенного командой `@font-face`. С его помощью можно отменить особенности, задаваемые последующими свойствами. В частности, применение приведенного ниже правила *не приведет* к представлению текста абзаца капителью или символами дробей, даже если они существуют в шрифте SwitzeraADF.

```
@font-face {
   font-family: "SwitzeraADF";
   font-weight: normal;
   src: url("SwitzeraADF-Regular.otf") format("opentype");
   font-feature-settings: "afrc" off, "smcp" off;
}

p {font: 1em SwitzeraADF, sans-serif; font-feature-settings:
   "afrc", "smcp";}
```

Как и другие дескрипторы, `font-feature-settings` позволяет указывать особенности для всех значений одноименного свойства, кроме `inherit`.

Генерирование начертаний

Довольно часто в семействе шрифтов отсутствует одна из востребованных в веб-дизайне гарнитур. Пользовательский агент может попытаться сгенерировать недостающую гарнитуру на основе уже имеющейся в семействе, но это зачастую

приводит к плачевным результатам. Для решения проблемы в CSS добавлено свойство `font-synthesis`, позволяющее указывать начертания, которые разрешается имитировать пользовательскому агенту.

font-synthesis

Значение	`none | weight || style`
Начальное значение	`weight style`
Применяется	Все элементы
Наследуется	Да
Анимируется	Нет

Многие пользовательские агенты умеют генерировать полужирное начертание, например добавляя пиксели по обе стороны штрихов у символов обычного начертания. Хотя это и не запрещается, полученные таким способом символы выглядят совершенно неприглядно. Именно поэтому большинство шрифтов включает полужирное начертание, содержащее символы тщательно выверенный формы, в чем можно убедиться, отобразив их в редакторе шрифтов.

Точно таким же образом пользовательский агент имитирует курсивное начертание, слегка наклоняя символы обычного начертания. Во многих случаях курсивные начертания, генерируемые браузером, выглядят еще хуже, чем полужирные. На рис. 5.32 показано, как соотносятся синтезированное браузером начертание (обозначается как наклонное) и "истинное" курсивное начертание шрифта Georgia.

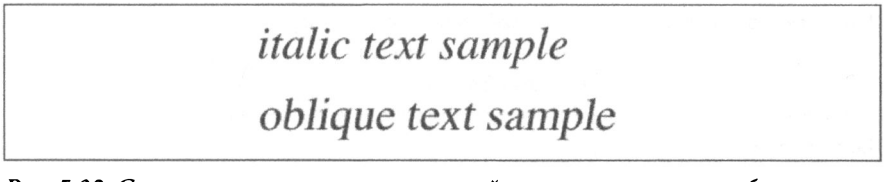


Рис. 5.32. Сравнение символов начертаний — сгенерированного браузером и разработанного дизайнером шрифтов

Для отключения механизма генерирования шрифтов браузера можно воспользоваться объявлением `font-synthesis: none;`. Чтобы заблокировать функцию автоматического синтеза новых начертаний во всем документе, можно применить следующее правило:

```
html(font-synthesis: none;)
```

В результате текст будет выводиться обычным начертанием, даже если в правилах явно указывается применить полужирное или курсивное начертание. Эта настройка позволяет не беспокоиться о том, что браузер преобразует символы неправильно.

К концу 2017 года свойство `font-synthesis` поддерживалось только браузером Firefox.

Свойство font

Описанные ранее свойства предоставляют разработчикам необычайно широкие возможности по форматированию текста документов, но применять их все в одном правиле весьма обременительно.

```
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;}
```

Для получения более компактной записи можно попробовать сгруппировать селекторы, но намного проще представить все свойства одним обобщающим свойством. Познакомьтесь со свойством `font`, которое обладает возможностями всех описанных выше свойств настройки шрифтов, а также включает несколько уникальных значений.

font

Значение	[[<*font-style*> || [`normal` | `small-caps`] || <*font-weight*>] ? <*font-size*> [/<*line-height*>]? <*font-family*>] | `caption` | `icon` | `menu` | `message-box` | `small-caption` | `status-bar`
Начальное значение	Значения отдельных свойств
Применяется	Все элементы
Процентное значение	В <*font-size*> вычисляется относительно значения, указанного для родительского элемента, а в <*line-height*> — относительно значения <*font-size*>
Вычисляется	Согласно правилам для отдельных свойств
Наследуется	Да
Анимируется	См. описание отдельных свойств

В объявление свойства `font` можно подставлять любые значения, характерные для других свойств или же представляющие системный шрифт (см. раздел "Системные шрифты"). В силу этого приведенные выше правила можно записать в более компактной форме (результат их применения показан на рис. 5.33).

```
h1 {font: italic 900 small-caps 30px Verdana, Helvetica, Arial,
    sans-serif;}
h2 {font: bold normal italic 24px Verdana, Helvetica, Arial,
    sans-serif;}
```

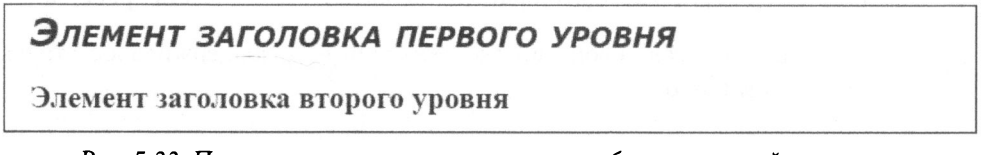


Рис. 5.33. Применение правил, включающих объявление свойства `font`

Выше приведен только один из возможных вариантов объявления свойства `font`, которых благодаря гибкой форме записи его определения может быть несколько. Если внимательно присмотреться, то легко заметить, что в рассмотренных выше примерах первые три значения свойства `font` приведены в отличном от исходного варианта порядке. В правиле для элемента `h1` первыми указываются значения свойств `font-style`, `font-weight` и `font-variant` (именно в таком порядке). Во втором правиле порядок следования значений несколько иной: `font-weight`, `font-variant` и `font-style`. В сущности, он не играет особой роли и может быть произвольным. Более того, если некоторые свойства представлены значением `normal`, то их вообще можно опустить. Следовательно, правила можно переписать в таком виде.

```
h1 {font: italic 900 small-caps 30px Verdana, Helvetica, Arial,
    sans-serif;}
h2 {font: bold italic 24px Verdana, Helvetica, Arial, sans-serif;}
```

В данном случае значение `normal` удалено из правила с объявлением свойства `font` для элемента `h2`, что никак не сказывается на конечном результате.

По правде говоря, произвольный порядок указания подлежащих свойств допустим только для первых трех значений. Положение последних двух значений задается более чем строго. Значения `font-size` и `font-family` *всегда* присутствуют в объявлении свойства `font` и занимают последние две позиции, независимо от расположения остальных значений. Если не указать хотя бы одно из них, то правило не будет рассматриваться браузером и применяться в документе. В качестве примера рассмотрим следующие объявления, результат применения которых к документу показан на рис. 5.34.

```
h1 {font: normal normal italic 30px sans-serif;} /* правильно */
h2 {font: 1.5em sans-serif;} /* правильно; опущены значения 'normal' */
h3 {font: sans-serif;} /* неправильно; отсутствует значение
                          'font-size' */
h4 {font: lighter 14px;} /* неправильно; отсутствует значение
                            'font-family' */
```

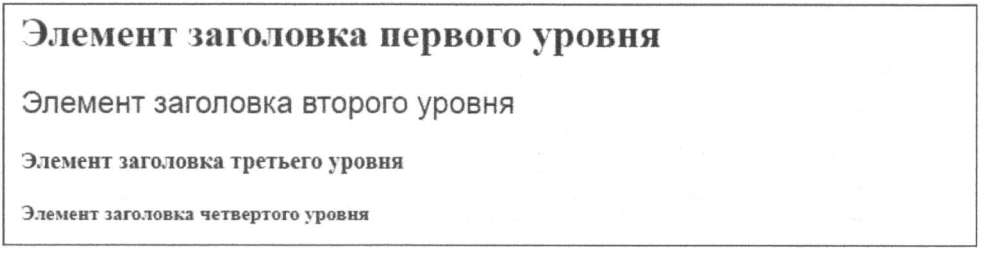


Рис. 5.34. Важность указания размера и типа шрифта

Высота строки

До этого момента мы имели дело только с пятью значениями свойства `font`, хотя их несколько больше. В частности, в него можно включать значение свойства `line-height`, строго говоря, относящееся к свойствам текста (в этой главе они не

рассматриваются), а не шрифта. Высота строки указывается как суффикс, добавляемый к значению `font-size` через косую черту (`/`).

```
body {font-size: 12px;}
h2 {font: bold italic 200%/1.2 Verdana, Helvetica, Arial, sans-serif;}
```

Результат применения оговоренных выше правил приведен на рис. 5.35. Элементы `h2` выводятся полужирным и курсивным начертаниями, одним из доступных шрифтов без засечек размером `24px` (в два раза больше размера шрифта, заданного для элемента `body`) и высотой строки `28.8px`.

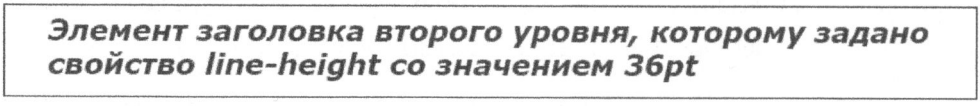


Рис. 5.35. Применение свойства `font`, в том числе указывающего высоту строки

Значение `line-height`, как и первых три значения свойства `font`, устанавливать не обязательно. При добавлении в объявление оно располагается после значения `font-size` (а не перед ним) и отделяется от него косой чертой.

Не занудства ради (ведь это самая распространенная ошибка начинающих разработчиков) повторю еще раз: в объявлении свойства `font` сначала указывается значение `font-size` и только после него — значение `font-family`. Порядок здесь очень важен. Остальные значения могут располагаться произвольно относительно друг друга.

Сокращенный формат объявления

Не забывайте, что при неумелом использовании свойство `font`, аккумулирующее значения всех остальных свойств настройки шрифтов, приводит к совершенно неожиданным результатам. Рассмотрим следующие правила, эффект применения которых продемонстрирован на рис. 5.36.

```
h1, h2, h3 {font: italic small-caps 250% sans-serif;}
h2 {font: 200% sans-serif;}
h3 {font-size: 150%;}

<h1>Элемент заголовка первого уровня</h1>
<h2>Элемент заголовка второго уровня</h2>
<h3>Элемент заголовка третьего уровня</h3>
```

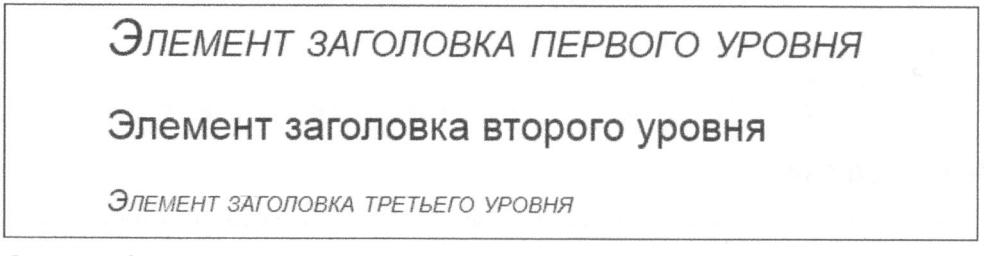


Рис. 5.36. Форматирование текста правилами, в которых свойство `font` объявлено в сокращенном формате

Заметили, что элемент h2 не выделяется курсивом и не представляется малыми прописными буквами и ни один из элементов не выводится полужирным начертанием? Так и должно быть! В сокращенном формате объявления всех не указанных в свойстве font значений сбрасываются до величин, задаваемых по умолчанию. Значит, приведенные ниже правила можно представить в следующем виде с сохранением равнозначности.

```
h1, h2, h3 {font: italic normal small-caps 250% sans-serif;}
h2 {font: normal normal normal 200% sans-serif;}
h3 {font-size: 150%;}
```

Анализ правил не оставляет сомнений в том, что для элемента h2 определено обычное начертание, а свойство font-weight, объявленное для всех трех элементов, устанавливается в значение normal. Эти значения объясняют поведение правила, представленного в сокращенном формате. Элемент h3 имеет иное форматирование, поскольку оно определяется свойством font-size, а не font, которое устанавливает только размер шрифта.

Системные шрифты

Для оформления веб-страницы стилями, максимально близкими к используемым в графическом интерфейсе операционной системы, свойству font нужно передать настройки системных шрифтов, а именно: размер, название семейства, начертание, насыщенность и альтернативные начертания шрифтов, используемых операционной системой. Указанные параметры системных шрифтов представляются в свойстве font такими значениями.

caption

Определяет форматирование надписей на элементах управления, например кнопках.

icon

Определяет форматирование названий значков.

menu

Определяет форматирование надписей раскрывающихся и прокручивающихся меню.

message-box

Определяет форматирование текста, выводимого в диалоговых окнах.

small-caption

Определяет форматирование надписей на мелких элементах управления.

status-bar

Определяет форматирование текста, выводимого в строке состояния.

Например, для форматирования названий кнопок на веб-странице таким же шрифтом, как и в надписях на кнопках графического интерфейса операционной системы, применяется такое простое правило:

```
button {font: caption;}
```

Системные значения, передаваемые свойству `font`, удобно использовать при создании веб-приложений с интерфейсом, подобным графическому интерфейсу операционной системы.

Учтите, что системные шрифты используются в CSS только с полным пакетом особенностей, т.е. системные значения указывают сразу все параметры шрифта. Это обеспечивает полное совпадение форматирования элементов (но не размера их шрифта и положения) с объектами интерфейса, у которых оно заимствовано. Тем не менее отдельные параметры системного шрифта можно изменить после назначения его элементу. Приведенное ниже правило определяет для кнопки размер шрифта как у родительского элемента.

```
button {font: caption; font-size: 1em;}
```

Если запрашиваемый системный шрифт недоступен, то пользовательский агент попытается найти ему точную замену (например, уменьшит размер шрифта `caption`, чтобы получить капитель, представляемую значением `small-caption`). Если замену найти не удается, то применяется шрифт по умолчанию пользовательского агента. Так, например, браузеру может быть известно расположение шрифта `status-bar`, но ничего не известно о существовании шрифта `small-caps`. В результате последний будет представляться значением `normal` свойства `small-caps`.

Замена шрифтов

Как известно, технология CSS обеспечивает браузеры гибким механизмом замены шрифтов и начертаний. Он основан на сравнении характеристик шрифтов, которое выполняется пользовательским агентом автоматически в малопонятном для авторов HTML-документов виде. Чтобы научиться создавать веб-страницы со строго заданным форматированием, необходимо четко понимать принципы управления шрифтами со стороны пользовательских агентов. Их описание вынесено в конец главы, так как они не обязательны для рассмотрения — на базовом уровне вполне достаточно понимания механизмов работы свойств. Приведенная ниже информация включена в главу исключительно для факультативного изучения.

1. Пользовательские агенты создают базу данных свойств шрифтов, задействованных в документе. В ней содержатся сведения о свойствах CSS всех шрифтов, известных пользовательскому агенту в текущей системе. К ним относятся все системные шрифты, а также шрифты, поставляемые с пользовательским агентом (встроенные шрифты). При нахождении идентичных шрифтов один из них игнорируется и в базу данных не заносится.
2. Перед форматированием документа пользовательский агент тщательно анализирует стилевые правила, устанавливающие внешний вид документа, создавая

список всех шрифтов, необходимых для решения этой задачи. Впоследствии шрифты из этого списка сопоставляются с доступными для использования шрифтами (системными и встроенными). Все совпадающие шрифты немедленно применяются в документе. При форматировании элементов, которым назначен недоступный для пользовательского агента шрифт, задействуется специальный механизм замены шрифтов.

3. Сначала ищутся шрифты, обладающие таким же значением свойства `font-stretch`.
4. Среди них определяются шрифты, имеющие общее с искомым шрифтом значение свойства `font-style`. При этом ключевое слово `italic` рассматривается как представляющее не только курсивное, но и наклонное начертание. Если совпадение не найдено, то замена шрифта не производится.
5. На следующем этапе проводится поиск шрифтов с общим значением свойства `font-weight`. Совпадения всегда находятся благодаря правильно отлаженному механизму установки начертания шрифтов (см. раздел "Насыщенность шрифта").
6. Далее ведется поиск совпадений по свойству `font-size`. Операция выполняется с определенным допуском, в каждом конкретном случае определяемым исключительно пользовательским агентом. Так, при поиске одного из шрифтов допуск по размеру может составлять 20%, а для других — всего 10%.
7. Если пользовательский агент не может определить расположение целевого шрифта (см. п. 2), то сначала он попробует найти альтернативный шрифт в текущем семействе. При нахождении такового к нему применяются действия, начиная с описанных в п. 2.
8. Если правильный шрифт найден, но в нем отсутствуют необходимые символы (например, знак охраны авторского права), то пользовательский агент возвращается к инструкциям п. 3 (или даже п. 2) в попытке найти альтернативный вариант.
9. При нахождении точного совпадения или альтернативного шрифта пользовательский агент применяет его для форматирования элементов согласно рабочим характеристикам, определяемым стилевыми правилами.

Помимо этого, при поиске шрифтов пользовательским агентом учитываются некоторые другие факторы.

1. Особенности шрифтов, востребованные по умолчанию, а также применяемые в стилевых правилах. По умолчанию в шрифтах активизированы особенности, представленные ключевыми словами `calt`, `ccmp`, `clig`, `liga`, `locl`, `mark`, `mkmk` и `rlig`.
2. Особенности, подключаемые с помощью дескрипторов `font-variant` правила `@font-face`. Кроме того, анализируются особенности, устанавливаемые дескриптором `font-feature-settings` команды `@font-face`.

3. Особенности, отличные от задаваемых свойствами `font-variant` и `font-feature-settings` (например, отключение лигатур сбросом свойства `letter-spacing` до значения по умолчанию).
4. Особенности, определяемые значениями свойства `font-variant` и связанных с ними свойств (например, `font-variant-ligatures`), а также любыми другими особенностями, свойственными шрифтам OpenType (в частности, `font-kerning`).
5. Особенности, представляемые значениями свойства `font-feature-settings`.

Детальное изучение процесса поиска и замены шрифтов требует много времени, но потраченные усилия стоят того, чтобы получить представление о подборе шрифтов для текста документа пользовательскими агентами. Рассмотрим, что происходит при назначении документу шрифта с засечками Times:

```
body {font-family: Times, serif;}
```

Пользовательский агент анализирует каждый элемент и определяет, содержит ли шрифт Times все символы, необходимые для отображения текста документа. В обычных документах проблем возникнуть не должно, но при вставке в них нестандартных символов (например, китайских иероглифов) шрифт Times оказывается несостоятельным. Таким образом, пользовательскому агенту придется найти ему замену или использовать другой шрифт только для представления специфических символов. Большинство шрифтов, рассчитанных на вывод символов европейских языков, не содержат китайских иероглифов. Редкое исключение составляет шрифт AsiaTimes — он может применяться для отображения текста всего документа или только символов азиатских языков. Решение принимается пользовательским агентом, но в любом случае китайские иероглифы будут выглядеть в документе так, как им полагается, независимо от того, какой шрифт применяется в остальной его части.

Резюме

Исходно представленная простым набором незамысловатых свойств, спецификация CSS превратилась в серьезный, полнофункциональный инструмент форматирования документа, в первую очередь отвечающий за визуализацию текста. На сегодняшний день разработчики получают в свое распоряжение огромное количество шрифтов, как стандартных, устанавливаемых в системе, так и самых замысловатых начертаний, загружаемых вместе с документами или из сторонних интернет-ресурсов.

Свойства CSS предоставляют необычайно широкие возможности по настройке шрифтов, недоступные ранее. Но не стоит злоупотреблять ими — не нужно форматировать веб-страницы сайта с помощью всех имеющихся в распоряжении шрифтов только потому, что они вам нравятся. Помимо эстетической несуразности такой подход грозит увеличением общего размера документов, а потому и недопустимым замедлением их загрузки. Как и в любом другом аспекте веб-дизайна, при выборе шрифтов нужно проявлять рассудительность и трезвый расчет.

ГЛАВА 6

Текстовые свойства

На удивление, подбор цветовых решений и оформление внешнего вида документов в веб-дизайне занимают намного меньше времени, чем форматирование текста и правильное расположение его на странице. В "чистом" HTML эта задача решается с помощью тегов `<font>` и `<center>`, но их возможности настолько ограничены, что практически не находят применения в современных документах.

Ключевая роль текста в дизайне веб-страниц обусловливает появление в инструментарии CSS большого количества специальных свойств, устанавливающих его внешний вид и поведение в документе. Они заметно отличаются от рассмотренных ранее свойств настройки шрифтов — в первую очередь областью применения. На самом простом уровне текст рассматривается пользовательскими агентами как содержимое документа, а шрифты — как один из инструментов, применяемых для его визуализации. С помощью текстовых свойств устанавливается положение текста в строке и определяются способы его представления: нижний или верхний индекс, подчеркивание, регистр символов. Кроме того, текстовые свойства позволяют (в известной степени) имитировать эффект табуляции, воспроизводимой клавишей <Tab> в обычных текстовых документах.

Отступы и выравнивание

Знакомство со способами форматирования текста в HTML-документах с помощью CSS лучше всего начать с изучения свойств позиционирования элементов в текстовой строке. Проще всего рассматривать эту тему в контексте создания документов специального типа, например отчетов и писем.

Прежде чем продолжить, нужно понять, каким образом в документ добавляются *строчные* и *блочные* элементы. В частности, если текст документа вводится преимущественно на одном из европейских языков, то блочные элементы заполняют документ сверху вниз, а строчные — слева направо. Ниже каждый из принципов рассмотрен более детально.

Направление или порядок расположения блоков (*поток блочных элементов*) определяет положение каждого следующего добавляемого в документ блочного элемента в принятой системе письма. В частности, в документах, введенных преимущественно на русском языке, элементы блочного уровня добавляются один под другим, что

соответствует расположению каждого следующего абзаца (или любого другого текстового элемента) под предыдущим.

Подобным образом *направление расположения строк* (*поток строчных элементов*) определяет порядок добавления в документ строчных элементов. Снова-таки, в русской письменной традиции каждый следующий строчный элемент располагается справа от предыдущего, что соответствует направлению письма слева направо. Другие языки, например арабский или иврит, характеризуются противоположным направлением письма (заполнения документов строчными элементами) — справа налево.

Остановимся на традиционном для нашей системы письма направлении заполнения документа данными. В обычном представлении каждый следующий блочный элемент добавляется под предыдущим, а каждый следующий строчный элемент — справа от предыдущего. Тем не менее при повороте страницы на 90° с помощью одного из стилевых свойств модуля CSS Transform направление заполнения ее элементами каждого из типов меняется на противоположное. Теперь блочные элементы будут располагаться горизонтально, а строчные — вертикально (мало того, последние будут добавляться снизу вверх).

Рассмотренные выше концепции абсолютны только в старых версиях CSS — с появлением инструментов трансформации понятия вертикального и горизонтального направлений заполнения документа элементами теряют всякий смысл. Именно поэтому в дальнейшем мы будем говорить о направлении расположения блочных элементов и направлении расположения строчных элементов без привязки к горизонтальной и вертикальной осям координатного пространства.

Текстовые отступы

Нетрудно заметить, что в большинстве книг, написанных на европейских языках, абзацы отделяются друг от друга не пустыми строками, а отступами первой строки. На некоторых сайтах для создания иллюзии отступа в начало первой строки помещается небольшое прозрачное изображение, которое смещает ее вправо на заданное расстояние. В CSS для создания отступов предназначено отдельное свойство: `text-indent`.

text-indent

Значение	*<length>* | *<percentage>*
Начальное значение	0
Применяется	Блочные элементы
Процентное значение	Относительное ширины содержащего блока
Вычисляется	Процентные значения (см. выше); абсолютные значения — в единицах измерения длины
Наследуется	Да
Анимируется	Да

Это свойство применяется для смещения первой строки целевого элемента на указанную в значении величину, даже если она представлена отрицательным числом.

Разумеется, чаще всего свойство `text-indent` применяется для расстановки отступов в документе.

```
p {text-indent: 3em;}
```

Последнее правило указывает сместить первую строку каждого из абзацев вправо на расстояние `3em`, как показано на рис. 6.1.

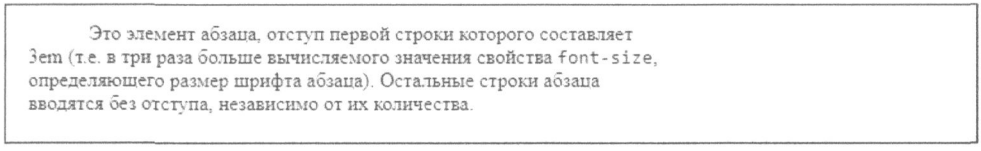


Рис. 6.1. Отступ первой текстовой строки

В общем случае свойство `text-indent` допускается применять к любому элементу, образующему контейнер блочного элемента, хотя задаваемый им отступ указывается для направления заполнения его строчными элементами. Отступы нельзя назначить строчным и замещаемым элементам, таким как изображения. Наряду с этим, если изображение вставлено в первую строку блочного элемента, то оно будет смещено свойством `text-indent` вместе с остальным текстом.

Для создания эффекта "отступа" первой строки в строчном элементе необходимо изменить его левое поле или отступ.

Свойству `text-indent` можно передавать отрицательные значения, что позволяет добиться некоторых специальных эффектов. Один из них — *висячий отступ*, заключающийся в выдвижении первой строки за пределы левой границы элемента.

```
p {text-indent: -4em;}
```

Проявляйте особую осторожность, задавая элементам отрицательные отступы: слова, выступающие за левую границу элементов, могут обрезаться браузером по границе окна. Чтобы предотвратить такое поведение пользовательского агента, элемент нужно снабдить левым полем или общим отступом.

```
p {text-indent: -4em; padding-left: 4em;}
```

В отдельных ситуациях отрицательный отступ первой строки имеет ощутимую практическую ценность. Рассмотрим следующий пример (рис. 6.2), в котором стилевое правило применяется для форматирования текста, окружающего незакрепленное изображение.

```
p.hang {text-indent: -25px;}

<img src="star.gif" style="width: 60px; height:
60px; float: left;" alt="Изображение пятиконечной звезды"/>
<p class="hang"> Отрицательный отступ обеспечивает выравнивание
```

```
первой строки по левому краю изображения. Последующие строки
не перекрывают изображение, поскольку лишены отступа.</p>
```

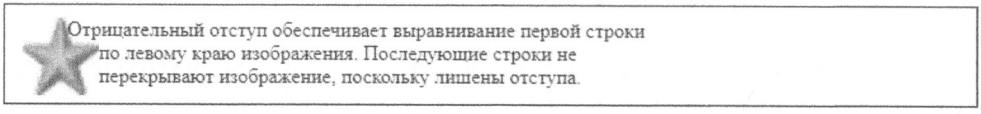


Рис. 6.2. Незакрепленное изображение и отрицательный отступ первой строки

Описанная выше методика лежит в основе многих интересных дизайнерских решений.

Задача обтекания незакрепленного изображения текстом намного эффективнее решается с помощью инструментов модуля CSS Float Shapes (подробнее об этом — в главе 10).

Для свойства `text-indent` допускается устанавливать любые числовые значения, выраженные в единицах измерения длины. В следующем примере в объявление добавлено процентное значение, указывающее величину отступа относительно ширины родительского элемента. Иными словами, отступ первой строки составляет 10% ширины абзаца, в данном случае элемента `div`, как "родителя" элемента `p` (рис. 6.3).

```
div {width: 400px;}
p {text-indent: 10%;}

<div>
<p>Этот абзац заключен в элемент div шириной 400px, а отступ его
первой строки равен 40px (400 * 10% = 40). Процентное значение
вычисляется относительно ширины родительского элемента.</p>
</div>
```

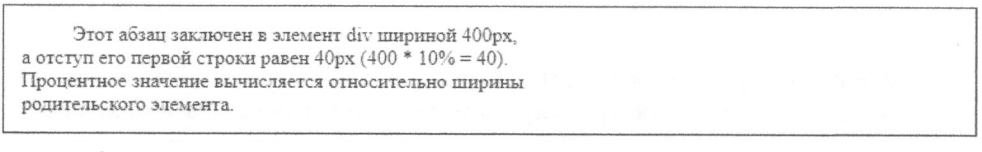


Рис. 6.3. Отступ первой строки, заданный процентным значением

Учтите, что отступ задается только для первой строки абзаца даже при включении в него символов разрыва строки. Самое необычное в свойстве `text-indent` — это наследование его дочерними элементами, которое порой проявляется далеко не с самой лучшей стороны. Рассмотрим пример, результат выполнения которого представлен на рис. 6.4.

```
div#outer {width: 500px;}
div#inner {text-indent: 10%;}
p {width: 200px;}

<div id="outer">
<div id="inner">
```

```
Первая строка элемента div снабжена отступом 50 пикселей.
<p>
Ширина этого абзаца равна 200px, а отступ его первой строки
составляет 50px, а все потому, что свойство text-indent наследуется,
а не принимает заданное значение.
</p>
</div>
</div>
```

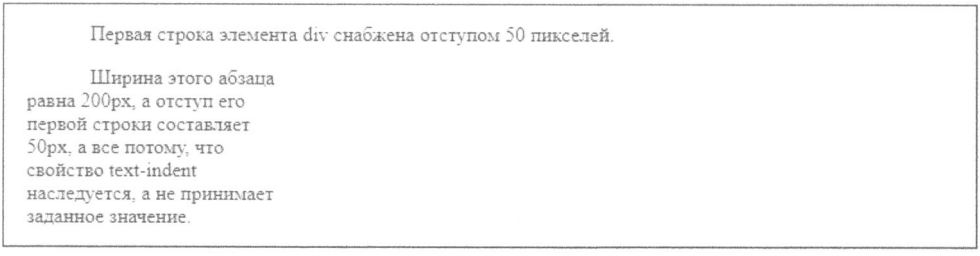


Рис. 6.4. Наследование отступа первой строки

Выравнивание текста

Свойство `text-align` применяется даже чаще, чем рассмотренное ранее свойство `text-indent`. Оно указывает способ выравнивая строк элемента относительно друг друга.

text-align

Значение	`start` | `end` | `left` | `right` | `center` | `justify` | `match-parent` | `end`
Начальное значение	В CSS3 — `start`; в CSS2.1 определяется пользовательским агентом (зависит от направления письма — `left` для европейских языков)
Применяется	Блочные элементы
Процентное значение	Относительно ширины содержащего блока
Вычисляется	Согласно определению, за исключением значения `match-parent`
Наследуется	Да
Анимируется	Нет
Примечание	В CSS3 поддерживается значение *<length>*, исключенное из CSS2.1 ввиду отсутствия поддержки пользовательскими агентами

Проще всего с работой этого свойства ознакомиться на примере, приведенном на рис. 6.5, в котором продемонстрирован эффект применения трех наиболее распространенных значений.

Значения `left`, `right` и `center` определяют выравнивание тестовых строк элемента соответственно по левому, правому краям и по центру. Поскольку указанное свойство применяется только к блочным элементам, таким как абзацы, оно не позволяет выравнивать одни только гиперссылки, не затрагивая остальную часть строки

(в противном случае текст гиперссылки будет перекрываться текстом строки блочного элемента).

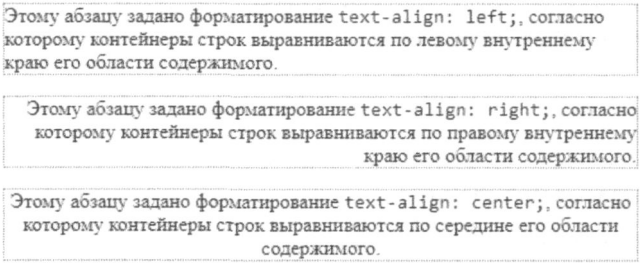


Рис. 6.5. Несколько способов форматирования текста свойством `text-align`

Исторически сложилось так (и было закреплено в CSS2.1), что в системах письма слева направо текстовые строки по умолчанию выравниваются по левому краю (`left`), а в системах письма справа налево — по правому краю (`right`). Спецификация CSS2.1 не определяет способ выравнивания текста при вертикальном расположении строк в родительских элементах. В CSS3 эта задача возлагается на значения `left` и `right`, определяющие выравнивание соответственно по верхнему и нижнему краям в вертикальных системах письма (рис. 6.6).

Рис. 6.6. Выравнивание по левому краю, по правому краю и по центру в вертикальных системах письма

Вне всяких сомнений, значение `center` свойства `text-align` устанавливает выравнивание строк посередине их родительского элемента. Ошибочно считать, что объявление `text-align: center` работает так же, как и тег `<center>` в HTML. В отличие от свойства `text-align`, тег `<center>` устанавливает выравнивание не только текста, но и любого другого содержимого целевого элемента (например, таблицы). Как вы помните, свойство `text-align` выравнивает элементы только строчного типа, что наглядно продемонстрировано на рис. 6.5 и 6.6.

Выравнивание по верхнему и нижнему краю

В CSS3 (если быть точным, то в CSS Text Module 3) добавлены новые способы выравнивания текстового содержимого элементов, а также изменено поведение по умолчанию для свойства `text-align` (по сравнению со спецификацией CSS2.1).

Значение `start`, задаваемое по умолчанию в CSS3, указывает выравнивать текстовые строки по краю, с которого они начинаются. В европейских языках (направление письма слева направо) оно равнозначно выравниванию по левому краю, а для текста на арабском языке или иврите (направление письма справа налево) — сопоставимо с выравниванием по правому краю. В вертикальных системах письма текстовые строки по умолчанию выравниваются по верхнему или нижнему краю в зависимости от места их начала. Как видите, начальное выравнивание полностью зависит от языка, на котором вводится текст документа. В большинстве случаев `start` является оптимальным значением для свойства `text-align`.

Несложно догадаться, что значение `end` соответствует выравниванию текстовых строк по краю элемента, у которого они оканчиваются. Для системы письма слева направо — это правый край, а для письма справа налево — противоположный, левый край, и т.п. Эффект передачи обоих ключевых слов свойству `text-align` продемонстрирован на рис. 6.7.

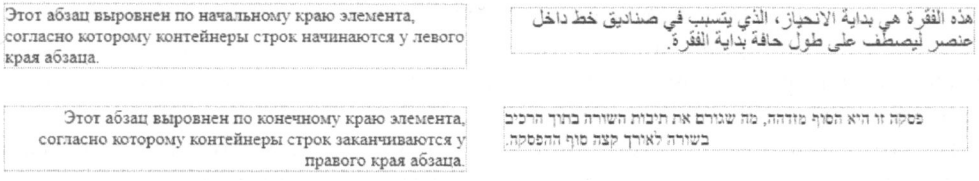


Рис. 6.7. Выравнивание текстовых строк по начальному и конечному краям элемента

Выравнивание по ширине

Ключевым словом `justify` обозначается часто упускаемый из виду способ выравнивания, заключающийся в привязке обоих краев текстовых строк к внутренним границам содержащего их элемента (рис. 6.8). В процессе выравнивания по ширине все строки блочного элемента получают одинаковую длину за счет перераспределения интервалов между словами. Выравнивание по ширине давно стало стандартом в книгопечатании (в частности, текст этой книги выровнен преимущественно по ширине), а в CSS добавлена поддержка некоторых других, более экзотичных способов выравнивания.

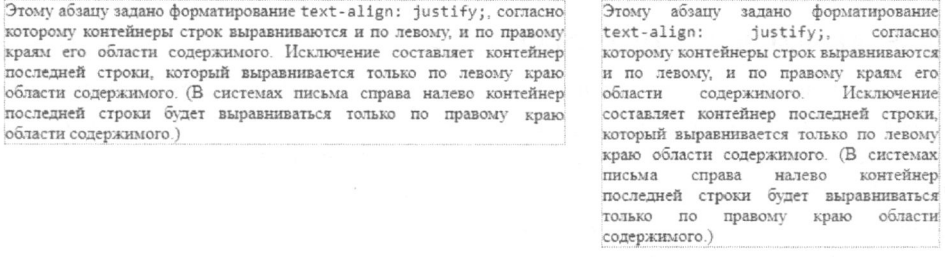


Рис. 6.8. Текст, выровненный по ширине

К концу 2017 года способ растягивания текстовых строк до необходимой ширины при выравнивании их по внутренним краям родительского элемента определялся исключительно пользовательскими агентами, а не CSS. Чтобы увеличить длину строки, некоторые браузеры изменяют только интервал между словами, в то время как остальные — в том числе корректируют межбуквенный интервал (даже несмотря на запрет такого действия со стороны CSS при задании свойству `letter-spacing` точного значения). Отдельные браузеры, наоборот, сжимают некоторые строки, чтобы должным образом перераспределить слова в остальных строках. Подобные манипуляции неизбежно приводят к изменению внешнего вида элемента, а иногда и его высоты, что зависит от количества сжатых браузером строк.

Для указания способа растягивания или сжатия строк при выравнивании их по ширине элемента в CSS предусмотрено еще одно свойство: `text-justify`. К концу 2017 года оно имело весьма ограниченную поддержку со стороны пользовательских агентов. Работа над его внедрением в Firefox только начиналась, а в Chrome завершилась неудачей.

Унаследованное выравнивание

У свойства `text-align` есть еще одно значение, которое нельзя обойти вниманием: `match-parent`. Оно не поддерживается браузерами, хотя функционально подобно значению `inherit`. Предполагалось, что объявление `text-align: match-parent` будет обеспечивать выравнивание текстовых строк таким же способом, как и у родительского элемента.

Несмотря на кажущееся сходство со значением `inherit`, между этими ключевыми словами большое различие. Если текстовое содержимое родительского элемента выравнивается с начальным значением `start` или `end`, то задание дочернему элементу выравнивания с помощью ключевого слова `match-parent` приведет к получению его свойством `text-align` значения `left` или `right`. В случае использования ключевого слова `inherit` свойство `text-align` дочернего элемента заимствует значение у родительского элемента без изменений: `start` или `end`.

К концу 2017 года синтаксис свойства `text-align` предполагал передачу ему значения `start end`. Это значение осталось функционально нереализованным в спецификации CSS3 и, скорее всего, будет исключено из нее, а потому в книге не рассматривается.

Выравнивание последней строки

Иногда возникает необходимость выравнивать последнюю строку элемента отличным от остальной части содержимого способом. В частности, выровненный по ширине текст выглядит намного презентабельнее, если его последняя строка выравнивается по левому краю. Наряду с этим в подписях и цитатах обычно применяется выравнивание последней строки по центру или по правому краю. Для решения такой задачи в CSS предусмотрено свойство `text-align-last`.

text-align-last

Значение	auto | start | end | left | right | center | justify
Начальное значение	auto
Применяется	Блочные элементы
Вычисляется	Согласно определению
Наследуется	Да
Анимируется	Нет

Как и в случае свойства `text-align`, со значениями свойства `text-align-last` проще всего ознакомиться на наглядных примерах (рис. 6.9).

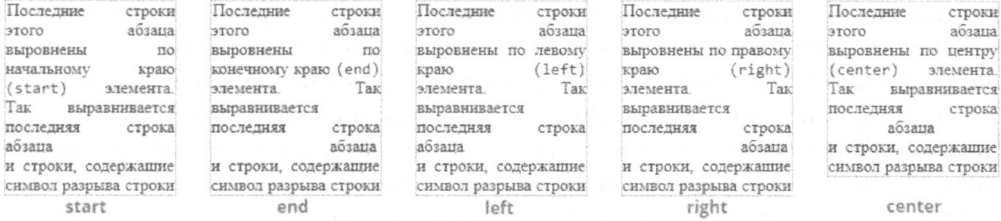


Рис. 6.9. Различные способы выравнивания последней строки текстового содержимого

На рис. 6.9 показано, что последняя строка выравнивается независимо от остальных строк элемента. Способ выравнивая полностью определяется ключевым словом, передаваемым свойству `text-align-last`.

При внимательном рассмотрении рис. 6.9 можно заметить, что свойство определяет способ выравнивания не только последней строки элемента блочного уровня. В действительности с его помощью можно определить выравнивание любой строки, предшествующей символу разрыва строки, который не обязательно обусловливает конец элемента. Так, например, при добавлении в разметку элемента тега `<br>` свойство `text-align-last` будет выравнивать строку, предшествующую ему. Таким же образом будет выравниваться последняя строка блочного элемента, в конец которой автоматически добавляется символ разрыва строки.

У свойства `text-align-last` есть досадный побочный эффект. Если последняя строка блочного элемента также является его первой строкой, то свойство `text-align-last` будет превалировать над свойством `text-align`. Таким образом, в результате применения к абзацу следующего правила его единственная строка будет выровнена по центру, а не по левому краю (`start`).

```
p {text-align: start; text-align-last: center;}

<p>Абзац!</p>
```

К концу 2017 года свойство `text-align-last` не распознавалось браузерами Safari и Opera Mini, а браузеры Internet Explorer и Edge поддерживали только его базовые значения: `left`, `right` и `center`.

Выравнивание по высоте

В предыдущем разделе речь шла о выравнивании строчных элементов вдоль направления текста. Далее мы поговорим об их выравнивании по высоте блочного элемента — так называемом вертикальном выравнивании, которое предполагает горизонтальное направление заполнения строчного элемента текстом. Поскольку позиционирование текстовых строк является сложной задачей, описание которой достойно отдельной книги, мы ограничимся только кратким рассмотрением основных ее положений.

Высота строки

Интервал между текстовыми строками зависит от их высоты. В данном случае под высотой понимается вертикальный размер текстовой строки, горизонтальный размер или длина которой рассматривается как ее ширина. Не забывайте, что описанные далее названия свойств и методы вертикального выравнивания справедливы только для европейских языков. Они унаследованы от самой первой версии спецификации CSS, в которой тексты на западноевропейских языках рассматривались как единственно возможные для стилевого форматирования.

Свойство `line-height` задает интервал между базовыми линиями строк, а не размер шрифта, определяя величину, на которую изменится высота блочного элемента, содержащего текстовые строки. По правде говоря, такое описание свойства не совсем точное. Если быть предельно точным, то значение свойства `line-height` изменяет не столько высоту строк, как *интерлиньяж* — интервал между базовыми линиями соседних строк, за вычетом высоты символов шрифта заданного размера. Другими словами, интерлиньяж — это разница между значением свойства `line-height` и размером шрифта.

line-height

Значение	*<number>* | *<length>* | *<percentage>* | `normal`
Начальное значение	`normal`
Применяется	Все элементы (см. замечания для замещаемых и блочных элементов)
Процентное значение	Относительно размера шрифта элемента
Вычисляется	Для процентных значений и значений в единицах длины — в числовом виде; остальные значения — согласно определению
Наследуется	Да
Анимируется	Да

При назначении элементу блочного типа свойство `line-height` устанавливает минимально возможный интервал между базовыми линиями текстовых строк. Это очень важный момент — передаваемое свойству значение определяет *не* абсолютный, а только минимально допустимый интервал между строками. Вследствие самых разных причин интервал между базовыми линиями может изменяться в бо́льшую

сторону без нарушения требований, устанавливаемых свойством line-height. Оно не изменяет положение замещаемых элементов, хотя и применяется к ним.

Структура текстовой строки

Каждый элемент строчного типа включает *область содержимого*, высота которой определяется размером шрифта, назначенного ему. В свою очередь область содержимого заключается в *контейнер строчного элемента*, в простейшем случае имеющий одинаковый с ней размер. Один из факторов, влияющих на размер контейнера строчного элемента, — это интерлиньяж, или междустрочный интервал, величина которого определяется свойством line-height.

Чтобы определить интерлиньяж строчного элемента, достаточно вычесть из значения свойства line-height размер шрифта, представляемый свойством font-size. Полученная величина представляет общий интерлиньяж и может выражаться отрицательным числовым значением. При ненулевом интерлиньяже контейнер строчного элемента расширяется вниз и вверх от области содержимого на интервал, равный половине междустрочного интервала.

В качестве примера рассмотрим строчный элемент, к которому применено свойство line-height со значением высоты 18px, а также свойство font-size со значением 14px. В данном случае интерлиньяж составляет 4px, потому область содержимого такого элемента будет расширяться в обоих направлениях (вверх и вниз) на 2px. Звучит немного запутанно, но именно таким образом свойство line-height воздействует на строчные элементы.

Когда блочный элемент отображается на экране, он может состоять из нескольких строк, занимающих всю доступную ширину. Каждая строка набирается из нескольких строчных элементов и называется *контейнером строки*. Высота контейнера строки определяется высотами его потомков. Строчные элементы выравниваются по общей базовой линии, а высоты контейнера строки всегда хватает, чтобы вместить всех его потомков (от самой высокой до самой низкой точки строчных элементов), как показано на рис. 6.10.

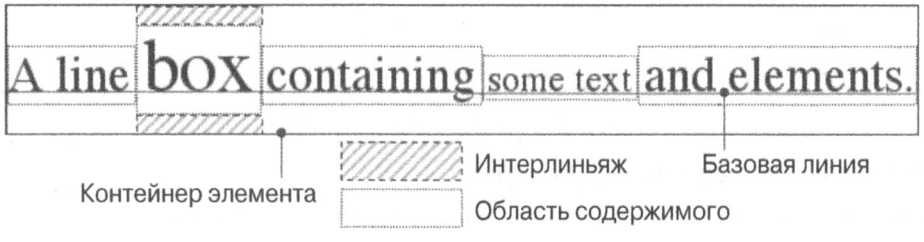


Рис. 6.10. Структурные элементы текстовой строки

Значения свойства line-height

Определившись с назначением свойств line-height, можно переходить к изучению передаваемых ему значений. По умолчанию свойство line-height имеет значение normal, обязывающее пользовательский агент рассчитывать интерлиньяж автоматически. Представляемое им значение зависит от браузера и чаще всего в 1,2 раза больше размера шрифта, указываемого свойством font-size для текущего элемента.

В качестве значения свойства `line-height` допускается использовать числовые величины, представленные в единицах измерения длины (например, `18px` или `2em`), или выражение *<number>*, более предпочтительное в большинстве случаев. Учтите, что при указании высоты строки в действительных единицах измерения, например `4cm`, браузер (или операционная система) может представлять строку в неправильном размере — высота строчного элемента на экране, скорее всего, будет заметно отличаться от заявленных 4 см.

Числовые значения, выраженные в единицах измерения `em` и `ex`, а также процентные значения, определяющие высоту строки относительно значения свойства `font-size`, использовать намного удобнее. Рассмотрим следующие примеры стилевых правил, применяемых к приведенному ниже фрагменту HTML. Конечный результат показан на рис. 6.11.

```
body {line-height: 18px; font-size: 16px;}
p.cl1 {line-height: 1.5em;}
p.cl2 {font-size: 10px; line-height: 150%;}
p.cl3 {line-height: 0.33in;}

<p>Этот абзац наследует высоту строки (line-height), равную 18px,
и размер шрифта (font-size), составляющий 16px, от элемента body.</p>
<p class="cl1">Высота этой строки несколько больше области
содержимого, поскольку свойство line-height абзаца установлено
в значение 27px (18 * 1.5).</p>
<p class="cl2">Высота этой строки несколько больше области
содержимого, поскольку свойство line-height абзаца установлено
в значение 15px (10 * 150%).</p>
<p class="cl3">Высота этой строки несколько больше области
содержимого, поскольку свойство line-height абзаца установлено
в значение 0.33in.</p>
```

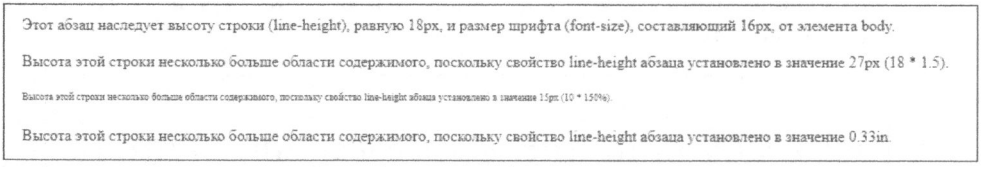


Рис. 6.11. Изменение высоты строки

Наследование свойства `line-height`

Ситуация сильно осложняется при наследовании свойства `line-height` от другого блочного элемента. Значение свойства `line-height` заимствуется дочерним элементом от родительского, но не наоборот. В качестве примера рассмотрим результат применения приведенного ниже кода CSS к фрагменту HTML-кода (рис. 6.12). Крайне сомнительно, что он найдет практическое применение.

```
body {font-size: 10px;}
div {line-height: 1em;} /* вычисляется как 10px */
p {font-size: 18px;}
```

```
<div>

<p>Свойство font-size этого абзаца установлено в значение 18px,
а значение 10px свойства line-height унаследовано от родительского
элемента. Такое форматирование вызывает заметное перекрывание
текстовых строк.</p>
</div>
```

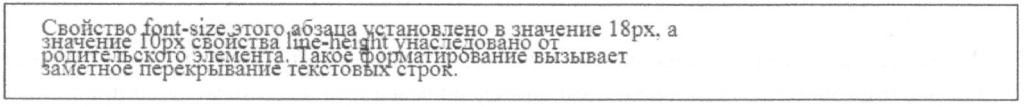


Рис. 6.12. Комбинация ничтожно малого междустрочного интервала и большого размера шрифта почти гарантированно приводит к проблемам при форматировании текстовых строк

Перекрывание текстовых строк, наблюдаемое на рис. 6.12, напрямую связано с наследованием абзацем высоты строки от родительского элемента `div` (`line-height: 1em`). Чтобы избежать нежелательного форматирования текстовых строк, всем элементам документа нужно задать свойство `line-height` с заметно большим числовым значением, что сложно реализовать на практике. Намного проще снабдить свойство `line-height` целевого элемента коэффициентом масштабирования, который корректирует высоту строки, наследуемую от родительского элемента.

```
body {font-size: 10px;}
div {line-height: 1;}
p {font-size: 18px;}
```

Коэффициент масштабирования наследуемого значения передается свойству в виде безразмерного числового значения. Он применяется к целевому элементу и всем его дочерним элементам независимо от значения свойства `font-size`, назначенного им (рис. 6.13).

```
div {line-height: 1.5;}
p {font-size: 18px;}

<div>
<p>Свойство font-size этого абзаца установлено в значение 18px,
а значение свойства line-height заимствуется от родительского
элемента div и масштабируется согласно коэффициенту 1.5 следующим
образом: 18 * 1.5 = 27px.</p>
</div>
```

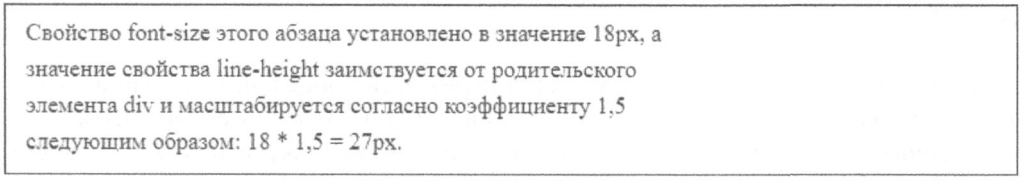


Рис. 6.13. Применение коэффициента масштабирования для корректировки высоты строки, наследуемой от родительского элемента

Может сложиться впечатление, что свойство line-height увеличивает (или уменьшает) высоту текстовой строки блочного элемента, но на самом деле это не так. Как объяснялось выше, его значение устанавливает интервал, на который расширяется область содержимого, определяющая размер контейнера строчного элемента. Посмотрим, что произойдет, если к абзацу с размером шрифта 12px, устанавливаемым по умолчанию, применить следующее правило:

```
p {line-height: 16pt;}
```

Благодаря принципу наследования текстовая строка будет иметь высоту 12 пикселей, а разница между нею и значением, определяемым приведенным выше правилом, составляющая 4 пикселя, будет разделена пополам и добавлена в виде дополнительного интервала к высоте контейнера строчного элемента по обе стороны от текстовой строки. Таким образом, интервал между базовыми линиями текстовых строк увеличивается до 16px, но размер шрифта остается в значении по умолчанию: 12px. Вследствие перераспределения избыточного значения свойства line-height область содержимого строчного элемента увеличивается за счет одинакового прироста в верхней и нижней частях.

Значение inherit свойства line-height вполне ожидаемо указывает наследовать высоту текстовой строки от родительского элемента. Результат такого наследования не отличается от принимаемого автоматически, но характеризуется другим уровнем приоритетности и каскадирования.

Ознакомившись со структурой текстовой строки и принципами изменения ее высоты, можно приступать к изучению стилевых свойств, обеспечивающих вертикальное выравнивание текста в пределах контейнера строки.

Вертикальное выравнивание текста

Если вам доводилось использовать элементы sup и sup, представляющие верхние и нижние индексы в текстовой строке, или включать в разметку документа дескрипторы <img src="foo.gif" align="middle">, то вы знакомы с отдельными методами выравнивания содержимого по высоте элемента. В CSS стилевое свойство vertical-align применяется только к элементам строчного типа и замещаемым элементам (например, изображениям и элементам управления форм) и никогда не наследуется.

В контексте свойства vertical-align понятия "горизонтальный" и "вертикальный" относятся к направлениям наполнения документа строчными и блочными элементами.

Свойству vertical-align допускается передавать процентные значения, значения, выраженные в единицах измерения длины, или именованные константы, представленные ключевыми словами. Некоторые из них вам уже знакомы, в то время как остальные представляют совершенно новые значения: baseline (по умолчанию), sub, super, bottom, text-bottom, middle, top и text-top. Назначение именованных

констант подробно рассматривается в дальнейших примерах, где они используются для форматирования строчных элементов.

vertical-align

Значение	`baseline` | `sub` | `super` | `top` | `text-top` | `middle` | `bottom` | `text-bottom` | *<length>* | *<percentage>*
Начальное значение	`baseline`
Применяется	Строчные элементы и ячейки таблицы
Процентное значение	Относительно значения свойства `line-height` элемента
Вычисляется	Для процентных значений и значений в единицах длины — в числовом виде; остальные значения — согласно определению
Наследуется	Нет
Анимируется	*<length>*, *<percentage>*
Примечание	В случае применения к ячейкам таблицы разрешены только значения `baseline`, `top`, `middle` и `bottom`

Свойство `vertical-align` *не применяется* для выравнивания содержимого элементов блочного типа, но его можно использовать для выравнивания содержимого ячеек таблицы.

Выравнивание по базовой линии

Объявление `vertical-align: baseline` указывает выравнивать базовую линию элемента по базовой линии его родителя. И именно такое поведение ожидается от строчных элементов, поскольку чаще всего они располагаются в строке на единой базовой линии.

Если выравниваемый элемент (например, изображение, элемент управления или замещаемый элемент) не снабжен базовой линией, то он привязывается к базовой линии родительского элемента своим нижним краем (рис. 6.14).

```
img {vertical-align: baseline;}

<p>Включенное в абзац изображение <img src="dot.gif" alt="A dot" />
выравнивается своим нижним краем по базовой линии текстовой строки.</p>
```

Включенное в абзац изображение • выравнивается своим нижним краем по базовой линии текстовой строки.

Рис. 6.14. Выравнивание целевого элемента по базовой линии родительского элемента

Это очень важное правило, поскольку оно обязывает привязывать нижний край замещаемого элемента к базовой линии, даже если в родительском элементе нет текста. Для примера рассмотрим ситуацию, когда изображение помещается в ячейку таблицы, не снабженную другим содержимым. Изображение располагается на базовой линии, но в отдельных браузерах между ней и нижним краем ячейки образуется пустое

пространство. Чтобы скрыть его, браузеры могут расширять изображение или сужать ячейку. Как бы там ни было, в образовании пустого пространства под изображением нет ошибки, хотя многим оно кажется нецелесообразным.

Верхний и нижний индексы

С помощью объявления `vertical-align: sub` строчный элемент опускается до уровня нижнего индекса. Его базовая линия (или нижний край в случае замещаемого элемента) понижается относительно базовой линии родительского элемента. В спецификации CSS не указывается величина смещения строчного элемента, поэтому она полностью устанавливается пользовательским агентом.

Ключевое слово `super`, в противоположность значению `sub`, указывает на приподнимание базовой линии строчного элемента до уровня верхнего индекса. Опять-таки, величина смещения его базовой линии относительно базовой линии родительского элемента всецело зависит от пользовательского агента.

Обратите внимание на то, что значения `sub` и `super` не влияют на размер шрифта строчного элемента, поэтому образованные с их помощью индексы не уменьшаются (и не увеличиваются), как полагается действительным индексам. По умолчанию индексы, образованные с помощью объявления `vertical-align: sub/super`, сохраняют размер шрифта родительского элемента, что продемонстрировано в следующем примере (рис. 6.15).

```
span.raise {vertical-align: super;}
span.lower {vertical-align: sub;}

<p>Этот абзац содержит <span class="raise">верхний </span>
и <span class="lower">нижний </span> индексы.</p>
```

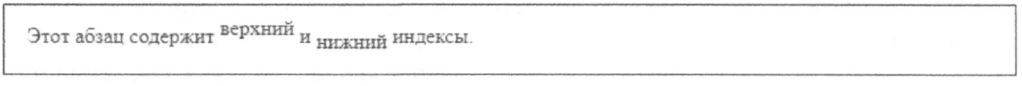


Рис. 6.15. Выравнивание строчных элементов по базовым линиям верхнего и нижнего индексов

Чтобы уменьшить размер верхнего и нижнего индексов, к представляющим их строчным элементам необходимо применить свойство `font-size` с меньшим значением, чем размер шрифта родительского элемента.

Привязка к нижнему краю

Объявление `vertical-align: bottom` обеспечивает привязку нижнего края контейнера строчного элемента к нижнему краю контейнера строки. Ниже приведен один из примеров такого поведения элементов, наглядно проиллюстрированный на рис. 6.16.

```
.feeder {vertical-align: bottom;}

<p>Как легко заметить, этот абзац содержит высокое
<img src="tall.gif" alt="tall" class="feeder" /> и низкое
```

```
<img src="short.gif" alt="short" class="feeder" /> изображения,
привязанные к нижней части контейнера строки.</p>
```

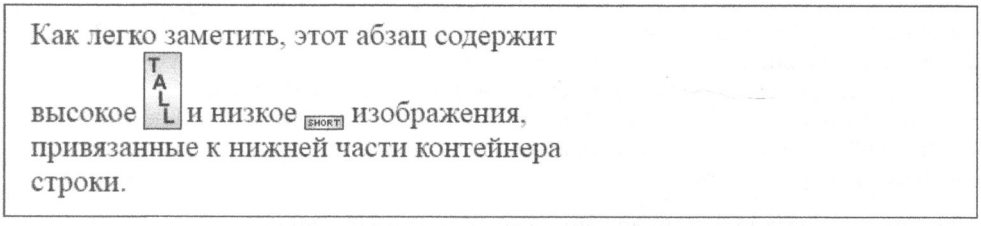


Рис. 6.16. Выравнивание по нижнему краю строки

Как показано на рис. 6.16, вторая строка абзаца включает два строчных элемента, нижние края которых расположены на одном уровне, расположенном под базовой линией текстовой строки.

Объявление `vertical-align: text-bottom` позволяет привязать нижний край элемента к нижнему краю текста строки (области содержимого). Исходя из назначения константы, можно сделать очевидный вывод, что оно применяется только к текстовым элементам, но не изображениям и замещаемым элементам. При попытке выполнения этой операции элемент будет выравниваться относительно некоего контейнера, задаваемого по умолчанию, размер которого определяется значением свойства `font-size` родительского элемента. В результате нижний край контейнера строчного элемента будет совмещаться с нижним краем контейнера по умолчанию, что часто приводит к совершенно непредвиденным эффектам. В частности, применение следующего CSS-кода к приведенному фрагменту HTML-документа приводит к результату, показанному на рис. 6.17.

```
img.tbot {vertical-align: text-bottom;}

<p>Абзац содержит: высокое <img src="tall.gif"
style="vertical-align: middle;" alt="tall" /> и низкое
<img src="short.gif" class="tbot" alt="short" /> изображения.</p>
```

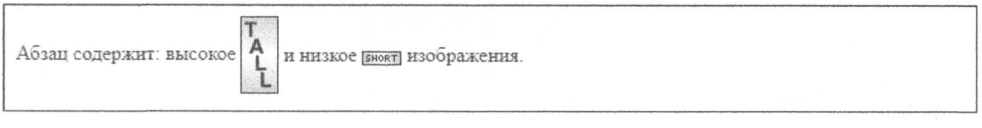


Рис. 6.17. Выравнивание нетекстового элемента по нижнему краю области содержимого

Подтягивание к верхнему краю

С помощью объявления `vertical-align: top` достигается эффект, противоположный получаемому при объявлении свойства `vertical-align: bottom`. Аналогичным образом, объявление `vertical-align: text-top` оказывается инверсным по отношению к объявлению `vertical-align: text-bottom`. Пример их применения для форматирования строчных элементов показан на рис. 6.18.

```
.up {vertical-align: top;}
.textup {vertical-align: text-top;}

<p>Абзац содержит: высокое <img src="tall.gif" alt="tall image">
изображение и <span class="up">текстовый фрагмент</span>,
выровненный по верхнему краю строки.</p>
<p>Абзац содержит: высокое <img src="tall.gif" class="textup"
alt="tall"> и низкое <img src="short.gif" class="textup"
alt="short" /> изображения, выровненные по верхнему краю области
содержимого строки.</p>
```

На рис. 6.18 видно, что положение выравниваемых элементов зависит от их типа и высоты, а также от размера шрифта родительского элемента.

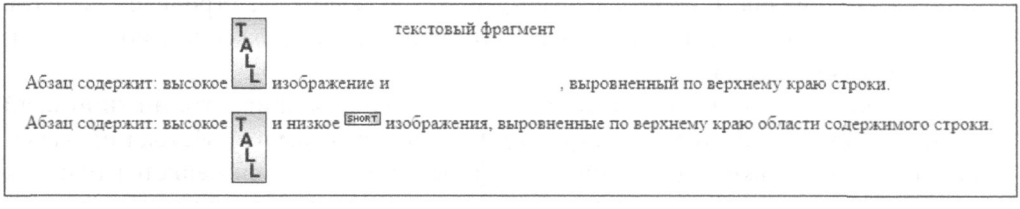


Рис. 6.18. Выравнивание по верхнему краю

Выравнивание посередине

Ключевое слово `middle` применяется для выравнивания строчных элементов достаточно часто, но воспроизводит далеко не тот эффект, на который указывает его название. Объявление `vertical-align: middle` выравнивает середину контейнера строчного элемента по линии, которая расположена выше базовой линии контейнера строки на `0.5ex`, где `1ex` указывается относительно значения свойства `font-size`, заданного для родительского элемента. Наглядно данный способ выравнивания проиллюстрирован на рис. 6.19.

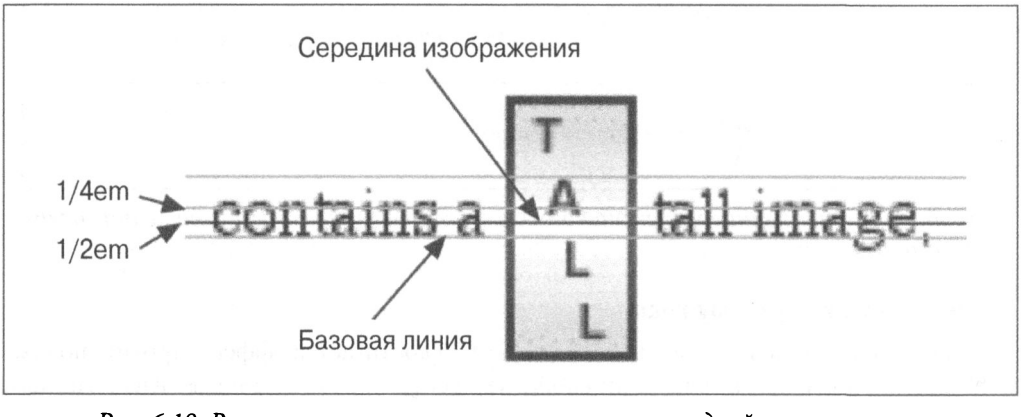


Рис. 6.19. Выравнивание строчного элемента по средней линии строки

Так как большинством браузеров величина `1ex` рассматривается как половина единицы `em`, чаще всего константа `middle` указывает на смещение середины строчного элемента вверх на расстояние `0,25em` относительно базовой линии родительского элемента. Это не строгое значение — оно может изменяться в зависимости от пользовательского агента.

Процентные значения

Процентные значения не позволяют имитировать эффект ключевого слова `middle` для изображений, но прекрасно подходят для опускания или приподнятия базовой линии строчного элемента (или нижнего края замещаемого элемента) на строго заданную величину относительно базовой линии контейнера строки. Процентное значение, указанное в свойстве `vertical-align`, вычисляется относительно значения свойства `line-height` выравниваемого элемента, а *не* его родительского элемента. Положительные значение соответствуют смещению базовой линии вверх, а отрицательные значения — вниз. В зависимости от степени приподнятия или опускания строчные элементы можно расположить в пределах одной строки сразу на нескольких уровнях (рис. 6.20). Будьте предельно внимательны!

```
sub {vertical-align: -100%;}
sup {vertical-align: 100%;}

<p>В критических ситуациях ты <sup>не поднимешься до уровня своих
ожиданий </sup>, а <sub> упадешь до уровня своей подготовки...</sub>
</p>
```

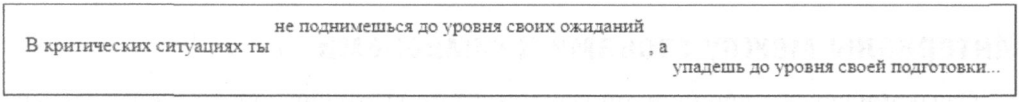


Рис. 6.20. Забавные эффекты, получаемые при установке положения базовой линии строчных элементов процентными значениями

Изучим, как вычисляется смещение строчных элементов, определяемое процентными значениями. Рассмотрим следующий фрагмент разметки документа.

```
<div style="font-size: 14px; line-height: 18px;"> I felt that,
if nothing else, I deserved a <span style="vertical-align: 50%;">
raise</span> for my efforts.
</div>
```

Базовая линия элемента `span`, свойству `vertical-align` которого задано значение `50%`, приподнята на 9 пикселей относительно базовой линии контейнера строки, что составляет ровно половину значения, присвоенного свойству `line-height` (`18px`, но не 7 пикселей, составляющих половину значения свойства `font-size`).

Значение длины

В качестве значения свойства `line-height` можно указывать расстояние, выраженное в единицах измерения длины. В этом отношении анализировать работу

свойства line-height проще всего: строчный элемент смещается в вертикальном направлении на строго заданное значение. Так, объявление vertical-align: 5px предписывает смещать строчный элемент на 5 пикселей вверх относительно исходного положения. Как и предполагалось, отрицательные значения обеспечивают смещение элемента вниз. Такой способ выравнивая, несмотря на свою простоту, не поддерживался в CSS1 и был представлен только в CSS2.

Важно понимать, что смещенный с помощью свойства line-height текст не переходит на другие строки, а остается частью исходной строки. К тому же он никогда не перекрывается с текстом соседних строк. На рис. 6.21 показано, как выглядит приподнятый относительно исходного уровня текстовый фрагмент, содержащийся во внутренней строке абзаца.

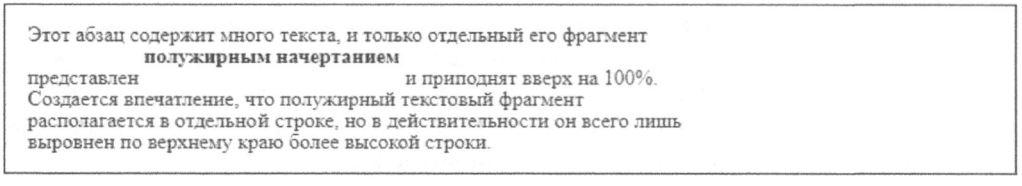


Рис. 6.21. Вертикальное выравнивание часто приводит к увеличению высоты строки

Как видите, смещение уровня базовой линии строчного элемента неизбежно приводит к изменению высоты строки, поскольку ее контейнер образуется так, чтобы включить все самые высокие и самые низкие точки всех включенных в нее строчных элементов, насколько бы сильно они ни были приподняты или опущены.

Интервалы между словами и символами

Ознакомившись с принципами вертикального выравнивания строчных элементов, рассмотрим свойства, влияющие на горизонтальное позиционирование строчных элементов, действие которых основано на изменении интервалов между словами и отдельными символами. Как и прежде, заметим, что применение таких свойств далеко не всегда приводит к ожидаемым результатам.

Интервалы между словами

Свойство word-spacing принимает отрицательные и положительные значения, выраженные в единицах длины. Указанная таким образом величина *прибавляется* к стандартному значению междусловного интервала. В сущности, свойство word-spacing применяется для изменения междусловного интервала, уже заданного элементу, а не для установки нового значения. Значение normal, определенное по умолчанию, соответствует числовому значению 0 (нуль).

word-spacing	
Значение	*<length>* | `normal`
Начальное значение	`normal`
Применяется	Все элементы
Процентное значение	Относительно значения свойства `line-height` элемента
Вычисляется	Ключевому слову `normal` соответствует значение `0`; остальные значения — в абсолютном виде
Наследуется	Да
Анимируется	Да

Положительные значения свойства `word-spacing` указывают увеличивать интервалы между символами, а отрицательные — уплотнять буквы и знаки в словах.

```
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">Интервал между словами этого абзаца увеличен
на 0.5em.</p>
<p class="tight">Интервал между словами этого абзаца уменьшен
на 0.5em.</p>
<p class="base">Этому абзацу назначен стандартный междусловный
интервал.</p>
<p class="norm">Этому абзацу назначен стандартный междусловный
интервал.</p>
```

Применение объявленных выше свойств к указанному фрагменту документа приводит к результату, показанному на рис. 6.22.

Интервал между словами этого абзаца увеличен на 0.5em.

Интервалмеждусловамиэтогоабзацауменьшенна0.5em.

Этому абзацу назначен стандартный междусловный интервал.

Этому абзацу назначен стандартный междусловный интервал.

Рис. 6.22. Изменение интервала между словами

Рассматривая междусловный интервал, мы не определились с понятием "слово". В спецификации CSS термином "слово" описывается любая текстовая строка, не содержащая пробелов и разделителей и окруженная с обеих сторон пустым пространством. Такое определение не соответствует семантическим правилам и всего лишь утверждает, что при добавлении в документ слов они обосабливаются друг от друга пробелами и разделителями. Пользовательские агенты, в которых реализована поддержка CSS, не снабжены инструментами грамматического анализа текста, поэтому

разработчики свойства word-spacing постарались сделать его максимально универсальным, чтобы обеспечить обработку текстов, представленных самыми разными системами письма, включая пиктографическое письмо. При неаккуратном использовании это свойство способно исказить исходный текст до неузнаваемости, как показано на рис. 6.23. Будьте внимательны!

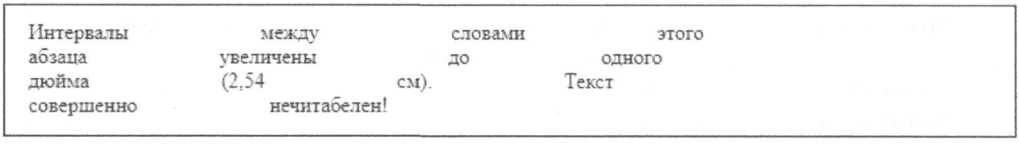


Рис. 6.23. Чрезвычайно большие интервалы между словами

Межсимвольный интервал

Многие особенности свойства word-spacing применимы и для свойства letter-spacing, определяющего интервал не между словами, а между буквами и знаками строчного элемента.

letter-spacing

Значение	*<length>* | normal
Начальное значение	normal
Применяется	Все элементы
Вычисляется	Значения в единицах длины — в абсолютном виде, иначе — normal
Наследуется	Да
Анимируется	Да

Свойство letter-spacing, как и word-spacing, допускает передачу ему числовых значений, выраженных в единицах измерения длины. При этом значение по умолчанию normal представляет числовое значение 0 (нуль). Любое другое число указывает увеличить или уменьшить межсимвольный интервал на заданную величину. Результат применения приведенных ниже правил к следующему фрагменту HTML-кода показан на рис. 6.24.

```
p {letter-spacing: 0;} /* равнозначно значению normal */
p.spacious {letter-spacing: 0.25em;}
p.tight {letter-spacing: -0.25em;}

<p>Текстовая строка этого абзаца введена со стандартным межсимвольным
интервалом.</p>
<p class="spacious">Текстовая строка этого абзаца очень "жидкая".</p>
<p class="tight">Текстовая строка этого абзаца сильно сжатая.</p>
```

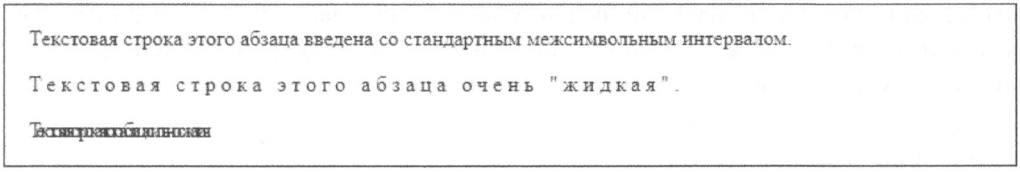


Рис. 6.24. Текстовая строка с разными межсимвольными интервалами

Свойство `letter-spacing` удобно использовать для выделения в документе акцентированного текста. Эта задача решается с помощью объявления следующего типа, пример применения которого к небольшому фрагменту документа показан на рис. 6.25.

```
strong {letter-spacing: 0.2em;}

<p>Этот абзац содержит <strong> фрагмент акцентированного текста
</strong> с большим межсимвольным интервалом.</p>
```

Рис. 6.25. Акцентирование текста с помощью свойства `letter-spacing`

Уменьшение или увеличение стандартного межсимвольного или междусловного интервала приводит к отмене специально заданных лигатур и некоторых других особенностей шрифта, оказывающих влияние на интервал между символами и словами строкового элемента. Изменение указанных интервалов с помощью свойств `letter-spacing` и `word-spacing` не приводит к автоматическому пересчету лигатур.

Интервалы и выравнивание

Интервал между словами зависит от значения свойства `text-align`. При выравнивании текста по ширине блочного элемента интервал между словами изменяется автоматически, чтобы обеспечить равномерное перераспределение их по длине строки. Это вызывает изменение значения свойства `word-spacing`. При этом свойство `letter-spacing` остается неизменным, но только в случаях представления точным числовым значением. Если же его величина определяется значением `normal`, то она может быть увеличена для обеспечения более точного выравнивания текстовых строк по ширине. Способ расчета межсимвольного интервала в CSS не указывается, а определяется пользовательским агентом, отображающим документ.

Учтите, что вычисляемое значение свойства `letter-spacing` наследуется, что приводит к весьма неприятным последствиям: дочерние элементы получают одинаковый интервал между символами, даже несмотря на представление шрифтами разных размеров. Свойства `word-spacing` и `letter-spacing`, в отличие от свойства `line-height`, не предполагают использования коэффициента масштабирования при

вычислении конечных значений, поэтому их значение наследуется дочерними элементами в чистом виде. Результат неподобающего наследования межсимвольного интервала строчными элементами приведен на рис. 6.26.

```
p {letter-spacing: 0.25em; font-size: 20px;}
small {font-size: 50%;}

<p>Большой межсимвольный интервал <small>наследуется даже элементами
с небольшим размером шрифта</small>, что выглядит не самым лучшим
образом. Чтобы сделать текст удобочитаемым, межсимвольный интервал
нужно сделать пропорциональным размеру шрифта.</p>
```

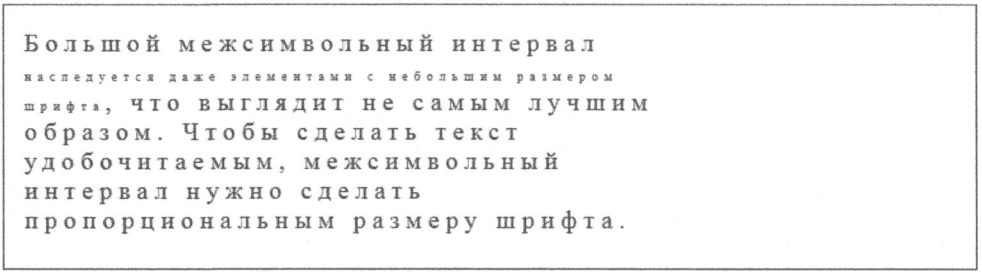


Рис. 6.26. Межсимвольный интервал, унаследованный от родительского элемента

Чтобы сделать межсимвольный интервал пропорциональным размеру шрифта каждого из элементов, применяется следующий подход к объявлению стилевых свойств:

```
p {letter-spacing: 0.25em;}
small {font-size: 50%; letter-spacing: 0.25em;}
```

Регистр символов

Рассмотрев свойства, устанавливающие интервалы между словами и символами строчных элементов, давайте изучим способы преобразования текста элемента в строчные и прописные символы, выполняемого свойством `text-transform`.

text-transform

Значение	`uppercase` | `lowercase` | `capitalize` | `none`
Начальное значение	`none`
Применяется	Все элементы
Вычисляется	Согласно определению
Наследуется	Да
Анимируется	Нет

По умолчанию это свойство получает значение `none`, указывающее не изменять регистр символов в целевом элементе. Значения `uppercase` и `lowercase`, как предполагают их названия, предписывают преобразовать символы элемента соответственно в верхний и нижний регистры. Последнее значение (`capitalize`) применяется для приведения к верхнему регистру только первого символа каждого слова. Пример применения свойства с разными значениями к фрагменту HTML-документа приведен на рис. 6.27.

```
h1 {text-transform: capitalize;}
strong {text-transform: uppercase;}
p.cummings {text-transform: lowercase;}
p.raw {text-transform: none;}

<h1>Один-единственный заголовок с начала документА</h1>
<p> По умолчанию текст выводится без изменения регистра символов,
но это поведение <strong>можно изменить</strong>, воспользовавшись
свойством text-transform.
</p>
<p class="cummings">
Любой ТЕКСТ можно представить в нижнем регистре, например в эпиграфе
"Американский поэт и писатель e.e.cummings".
</p>
<p class="raw">
Для сохранения Исходного Способа Представления Текста достаточно
передать свойству text-transform значение none.
</p>
```

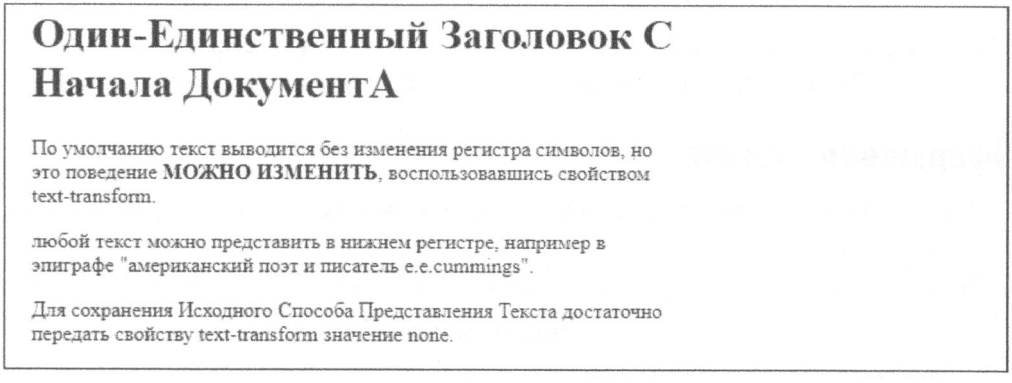


Рис. 6.27. Изменение регистра символов в тексте документа

Пользовательские агенты по-разному определяют начало слов, а потому и символы, требующие изменения регистра при передаче свойству `text-transform` значения `capitalize`. В частности, в приведенном выше примере словосочетание "один-единственный", включенное в элемент `h1`, может представляться как "Один-единственный" или "Один-Единственный". Спецификация CSS не указывает однозначно, каким образом пользовательский агент должен поступать в подобных случаях.

Легко заметить, что заголовок, представленный элементом h1, заканчивается прописной буквой. В этом нет ошибки: при передаче значения capitalize свойство text-transform преобразует в верхний регистр только первые буквы каждого слова, не затрагивая всех остальных символов. Остальные буквы слов попросту игнорируются пользовательским агентом и остаются неизменными.

Несмотря на кажущуюся избыточность, в отдельных ситуациях свойство text-transform становится просто незаменимым, например при приведении к верхнему регистру всех заголовков первого уровня документа. Вместо того чтобы заниматься переписыванием текста элементов h1, достаточно включить в разметку документа и таблицу стилей следующий код.

```
h1 {text-transform: uppercase;}

<h1>Это элемент h1</h1>
```

Преимущества свойства text-transform неоспоримы: представляя заголовки первого уровня прописными символами с помощью всего одного правила, их можно быстро вернуть к изначальному виду, изменив единственное значение в его объявлении (рис. 6.28).

```
h1 {text-transform: capitalize;}

<h1>Это элемент h1</h1>
```

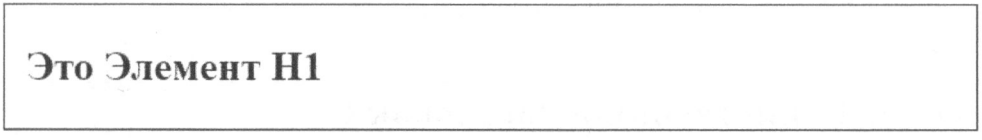


Рис. 6.28. Преобразование символов из строчных в прописные

Оформление текста

Настало время рассмотреть одно из самых эффектных свойств настройки текста: text-decoration.

text-decoration

Значение	none | [underline || overline || line-through || blink]
Начальное значение	none
Применяется	Все элементы
Вычисляется	Согласно определению
Наследуется	Нет
Анимируется	Нет

Как и следовало ожидать, значение underline добавляет подчеркивание к тексту элемента, подобное устанавливаемому с помощью элемента U в первых специфика-

циях HTML. Значение `overline` не менее интересное: оно указывает добавлять линию над текстом элемента. Для *перечеркивания* текста (добавления линии посередине текстовой строки) служит значение `line-through` (в HTML эта задача решается с использованием элемента `S` или `strike`). С помощью значения `blink` текст элемента можно заставить мигать (чтобы получить такой же эффект в HTML, необходимо обратиться к незаслуженно порицаемому элементу `blink`, поддерживаемому браузерами семейства Netscape). Примеры использования каждого из значений свойства `text-decoration` приведены на рис. 6.29.

```
p.one {text-decoration: underline;}
p.twoo {text-decoration: overline;}
p.three {text-decoration: line-through;}
p.four {text-decoration: blink;}
p.five {text-decoration: none;}
```

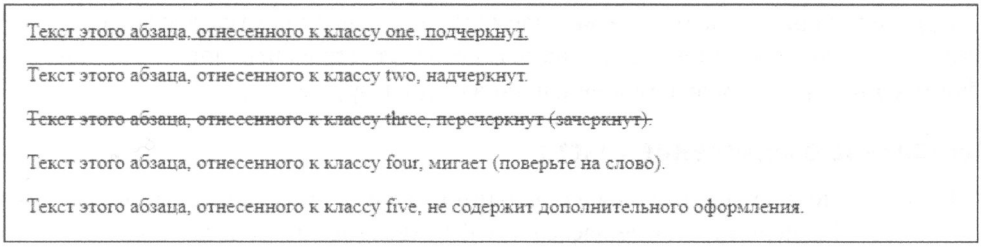


Рис. 6.29. Разные способы оформления текста

Печать не позволяет передать эффект, создаваемый значением `blink`, что не запрещает представить его умозрительно. К сожалению, далеко не каждый пользовательский агент обеспечивает мигание текста. На момент выхода книги все популярные браузеры уже не поддерживали эффект мигания текста, задаваемый свойством `text-decoration` со значением `blink` (в Internet Explorer оно никогда и не поддерживалось).

Значение `none` отключает все эффекты оформления текста, назначенные ранее. Неоформленный текст выводится по умолчанию, но далеко не для всех элементов. В частности, подчеркивание гиперссылок выполняется по умолчанию, и чтобы отменить его, понадобится применить к ним следующее правило:

```
a {text-decoration: none;}
```

В результате гиперссылки будут отличаться от обычного текста только цветом (по крайней мере, при настройках браузера, устанавливаемых по умолчанию, ведь об их ручном изменении известно только тому, кто его выполнил).

Не забывайте, что многих раздражает отсутствие подчеркивания гиперссылок. Такое форматирование элементов `a` не является обязательным, и им можно пренебречь, а вот отказываться от цветового оформления гиперссылок, заданного по умолчанию, все же не стоит. Не нужно испытывать терпение пользователей, заставляя их подолгу искать гиперссылки,

представленные в нестандартных цветовых решениях, ведь пользователи могут страдать одной из форм дальтонизма.

В одном правиле допускается комбинировать несколько способов оформления текста. Чтобы представить гиперссылки не только подчеркнутыми, но и перечеркнутыми, потребуется такое правило:

```
a:link, a:visited {text-decoration: underline overline;}
```

Будьте внимательны и учитывайте тот факт, что при назначении элементу сразу двух правил с разными значениями одного и того же свойства оформления текста эффект возымеет только одно из них. Рассмотрим следующий пример.

```
h2.stricken {text-decoration: line-through;}
h2 {text-decoration: underline overline;}
```

Перечеркивание задается только элементу `h2`, относящемуся к классу `striken`. При этом подчеркивание и надчеркивание будут утрачены, поскольку эффекты оформления текста замещают, а не дополняют друг друга.

Несуразное оформление текста

Нам осталось ознакомиться со странностями в поведении свойства `text-decoration`. Во-первых, оно не наследуется! Буквально это означает, что линии перечеркивания, зачеркивания и надчеркивания будут отображаться цветом, характерным для родительского элемента, даже в случае назначения дочерним элементам отличающегося цветового оттенка (рис. 6.30).

```
p {text-decoration: underline; color: black;}
strong {color: gray;}

<p>Этот абзац, включающий черный текст, подчеркнутый черной линией,
содержит <strong>сильно акцентированный фрагмент</strong>,
сохраняющий подчеркивание родительского элемента.</p>
```

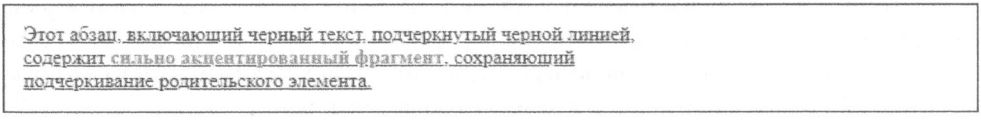


Рис. 6.30. Согласование линий подчеркивания у разных элементов

А теперь детально разберемся, почему разные элементы подчеркиваются одинаковой линией. Отключение наследования для свойства `text-decoration` выполняется передачей ему значения `none`. Таким образом, элемент `strong` не содержит линии подчеркивания. В это сложно поверить, поскольку в строке под сильно акцентированным текстовым фрагментом четко просматривается тонкая линия подчеркивания, распространяющаяся на остальной текст абзаца. В действительности она принадлежит не элементу `strong`, а всему абзацу, в который он вложен. Чтобы удостовериться в этом, отмените подчеркивание сильно акцентированного текста и примените к нему несколько измененное стилевое решение.

```
p {text-decoration: underline; color: black;}
strong {color: gray; text-decoration: none;}

<p>Этот абзац, включающий черный текст, подчеркнутый черной линией,
содержит <strong>сильно акцентированный фрагмент</strong>,
сохраняющий подчеркивание родительского элемента.</p>
```

Результат применения правил подобен показанному на рис. 6.30, так как с точки зрения браузера ничего не изменилось. Иными словами, CSS не обладает средствами отмены подчеркивания (перечеркивания и надчеркивания) у строчного элемента, унаследованного от родительского элемента.

Еще более странный результат получается при комбинировании свойств `text-decoration` и `vertical-align` (рис. 6.31). Можно заметить, что линия надчеркивания, заданная для абзаца, проходит через приподнятый вверх элемент `sub`, которому специальное оформление текста не задано.

```
p {text-decoration: overline; font-size: 12pt;}
sup {vertical-align: 50%; font-size: 12pt;}
```

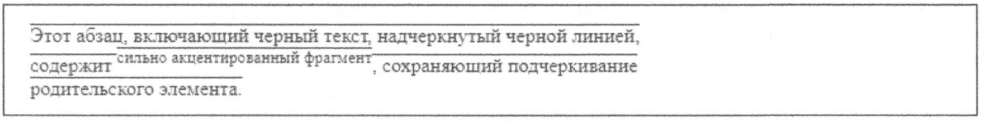


Рис. 6.31. Необычный, хотя и корректный способ оформления текста абзаца

До этого момента рассматривались примеры специального оформления текста, которого лучше избегать в собственных документах. В действительности далеко не любое оформление текста избыточно. Более того, описанные выше примеры демонстрируют только один из вариантов обработки стилевых правил браузерами. Несмотря на установки спецификации CSS, подчеркивание дочернего элемента отменяется далеко не всегда. Такое поведение браузеров в большей степени соответствует ожиданиям авторов документов. Рассмотрим простой пример подобного поведения пользовательского агента.

```
p {text-decoration: underline; color: black;}
strong {color: silver; text-decoration: none;}

<p>Этот абзац, включающий черный текст, подчеркнутый черной линией,
содержит <strong>сильно акцентированный фрагмент</strong>, лишенный
подчеркивания, наследуемого от родительского элемента.</p>
```

Результат отображения такого документа в браузерах, в которых отключено подчеркивание дочерних элементов (в нашем случае `strong`), приведен на рис. 6.32.

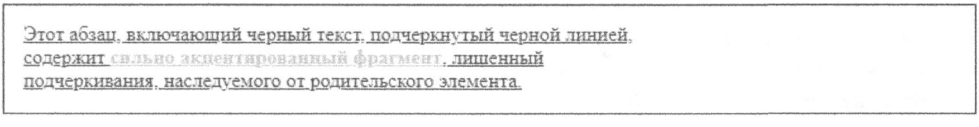


Рис. 6.32. Отключение подчеркивания дочерних элементов в отдельных браузерах

Стоит оговориться, что многие браузеры следуют требованиям спецификации CSS, а те из них, что не придерживаются ее, скорее всего, сделают это в обозримом будущем (с выходом новой версии). Именно поэтому, отменяя оформление текста с помощью значения `none`, нужно быть готовым к непредвиденному форматированию элементов документа не только в современных, но и в будущих версиях браузерах. Учтите, что новые версии CSS могут содержать другие инструменты отмены оформления текста, не синхронизируемые со значением `none`.

В любом случае цвет оформления текста можно изменить, не нарушая требований спецификации CSS. Как известно, цвет оформления текста элемента наследуется всеми его дочерними элементами, даже теми, которые имеют собственное оформление. Чтобы текст документа был оформлен тем же цветом, что и сам элемент, необходимо включать его объявление в стилевое правило.

```
p {text-decoration: underline; color: black;}
strong {color: silver; text-decoration: underline;}

<p>Этот абзац, включающий черный текст, подчеркнутый черной линией,
содержит <strong>сильно акцентированный фрагмент</strong>,
сохраняющий подчеркивание родительского элемента, перекрываемое
серой линией вложенного элемента.</p>
```

На рис. 6.33 показан результат оформления текста элемента `strong` подчеркиванием линией серого цвета. Она перекрывает черную линию подчеркивания, заданную для родительского элемента, поэтому элемент `strong` и линия его подчеркивания окрашиваются одним цветом.

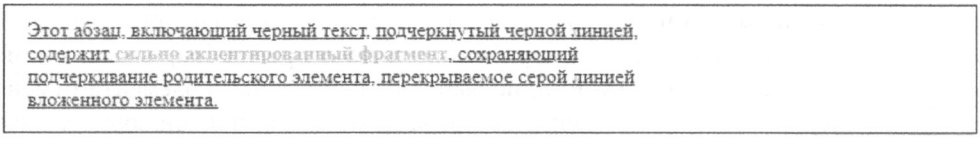


Рис. 6.33. Замена наследуемого оформления текста, задаваемого по умолчанию

Оптимизация отображения текста

Текущая версия свойства `text-rendering` заимствована у SVG, но рассматривается пользовательскими агентами как относящаяся к CSS. С ее помощью определяется способ оптимизации текстового представления документа.

text-rendering

Значение	`auto` | `optimizeSpeed` | `optimizeLegibility` | `geometricPrecision`
Начальное значение	`auto`
Применяется	Все элементы
Наследуется	Да
Анимируется	Да

Назначение ключевых слов `optimizeSpeed` и `optimizeLegibility` полностью определяется их названиями. В первом случае наибольший приоритет имеет скорость отображения текста (в ущерб удобочитаемости и геометрической четкости символов), а во втором — удобочитаемость (за счет увеличения времени отображения документа).

Особенности шрифтов, оптимизируемые с помощью значения `optimizeLegibility`, в спецификации CSS однозначно не указаны, поэтому качество отображения текста всецело зависит от операционной системы, в которой запускается пользовательский агент. На рис. 6.34 приведены примеры оптимизации текста на максимальную скорость загрузки и удобочитаемость.

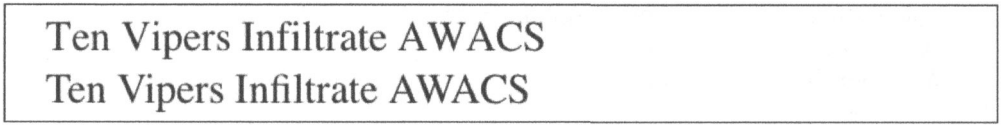


Рис. 6.34. Оптимизация отображения текста разными способами

Как видно на рис. 6.34, оптимизация одного и того же текста передачей свойству `text-rendering` разных значений приводит к схожим результатам, но характеризующимся разными уровнями удобочитаемости.

Отдельные пользовательские агенты оптимизируют текст только с целью повышения его удобочитаемости, даже при прямом указании оптимизировать скорость его отображения в документе. Это вызвано существенным увеличением производительности браузеров за последние несколько лет, достаточным для сокращения времени выполнения операции до приемлемого уровня.

Наряду с этим значение `geometricPrecision` обязывает представлять текст символами с максимально возможной геометрической четкостью, обеспечивая их масштабирование без заметного искажения. Крайне удивительно, что такое поведение не задается браузерами по умолчанию. Обычно оптимизация отображения текста предполагает применение разных лигатур и настроек кернинга для шрифтов разных размеров. В частности, это позволяет осветлить текст с малым размером шрифта и уплотнить его при использовании крупных шрифтов. Значение `geometricPrecision` нивелирует важность размера шрифта при настройке кернинга и лигатур — пользовательский агент представляет текст векторными контурами, подобными используемым в SVG-изображениях, а не символами шрифта.

Определение назначения ключевого слова `auto` настолько неоднозначное, что не представляет ясности даже по меркам стандартов индустрии.

> Решение об оптимизируемых параметрах принимается пользовательским агентом, но удобочитаемость всегда приоритетнее скорости и геометрической четкости представления текста.

Именно так, браузер сам решает, что важнее в каждом конкретном документе: скорость отображения, удобочитаемость или четкость представления.

Текст, отбрасывающий тень

В отдельных ситуациях к тексту требуется добавить тень. Для этих целей применяется свойство `text-shadow`. При первом знакомстве его синтаксис несколько обескураживает, но очень скоро перестает удивлять.

text-shadow

Значение	none | *[<length> || <length> <length> <length>?]#*
Начальное значение	none
Применяется	Все элементы
Наследуется	Нет
Анимируется	Да

По умолчанию (значение `none`) тень не отбрасывается текстом. Остальные значения позволяют задать ему одну или несколько теней. Каждая из теней характеризуется опционально устанавливаемым цветом и тремя размерными параметрами, последний из которых указывать необязательно.

Цвет тени может быть произвольным, и если его не указать, то она будет представляться таким же цветом, как и текст.

Первые две размерные характеристики указывают смещение тени относительно текста. Первое значение определяет смещение по горизонтали, а второе — по вертикали. Так, например, следующее правило, результат применения которого показан на рис. 6.35, позволяет создавать сплошную, не размытую зеленую тень, смещенную вправо на пять пикселей и вниз на расстояние `0.5em` относительно исходного текста.

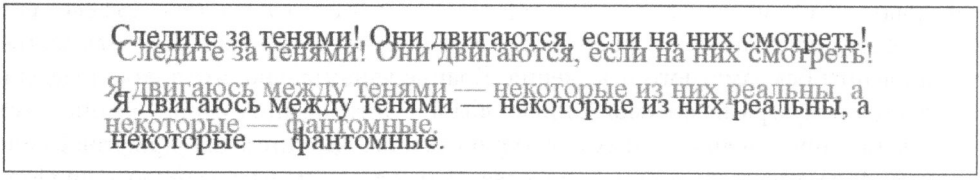


Рис. 6.35. Простые тени (см. цветные иллюстрации на веб-сайте)

Отрицательные значения указывают смещать тень против направления текста как в горизонтальном, так и вертикальном направлении. В нижней части рис. 6.35 показан один из таких эффектов: светло-синяя тень отбрасывается со смещением 5 пикселей влево и `0.5em` вверх относительно исходного текста.

```
text-shadow: rgb(128,128,255) -5px -0.5em;
```

Необязательный третий размерный параметр определяет радиус размытия тени. Он равен расстоянию, на которое распространяется эффект размытия цвета (отсчитывается от контуров тени). Радиус в два пикселя соответствует размытию от границы до исходных контуров тени. Способ размытия свойством `text-shadow` не

указывается, а потому определяется пользовательским агентом в каждом случае отдельно (рис. 6.36).

```
p.cl1 {color: black; text-shadow: gray 2px 2px 4px;}
p.cl2 {color: white; text-shadow: 0 0 4px black;}
p.cl3 {color: black; text-shadow: 1em 0.5em 5px red, -0.5em -1em
       hsla(100,75%,25%,0.33);}
```

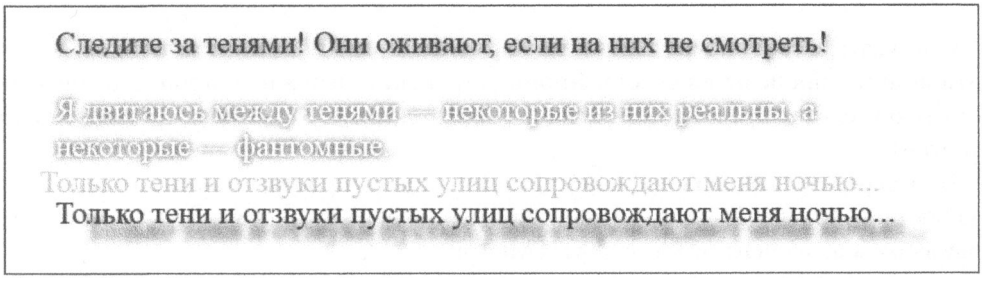


Рис. 6.36. Отбрасывание теней (см. цветные иллюстрации на веб-сайте)

Добавление к тексту крупных теней или теней с сильным размытием приводит к заметному увеличению времени обработки документов, особенно при отображении на устройствах, оснащенных слабым процессором и небольшим количеством оперативной памяти. Перед публикацией документы, содержащие текст с тенью, требуют всестороннего тестирования на скорость визуализации в различных системах.

Обработка пробелов

Изучив методы оформления текста в CSS, перейдем к рассмотрению свойства `white-space`, отвечающего за распознавание браузерами пробелов, символов табуляции и новой строки.

white-space

Значение	`normal` | `nowrap` | `pre` | `pre-wrap` | `pre-line`
Начальное значение	`normal`
Применяется	Все элементы (CSS 2.1); блочные элементы (CSS1 и CSS2)
Вычисляется	Согласно определению
Наследуется	Нет
Анимируется	Нет

Свойство `white-space` определяет способ обработки пробелов между словами и строками документа. До известной степени эта задача прекрасно решается с помощью инструментов XHTML — любое количество последовательно расположенных пробелов в разметке документа представляется одним символом. Именно поэтому

документы, размеченные с помощью XHTML, при отображении в окне браузера никогда не содержат два расположенных подряд пробела, а внутри его элементов отсутствуют символы перевода строки. Рассмотрим следующий фрагмент документа:

```
<p>В этом    абзаце   слишком     много        пробелов.</p>
```

Чтобы задать обычный способ обработки пробелов, применяется такое правило:

```
p {white-space: normal;}
```

Оно указывает браузеру не изменять принятое по умолчанию поведение — удалять лишние пробелы из текста. Кроме устранения ненужных пробелов, значение `normal` обязывает преобразовывать в пробелы символы перевода строки (возврата каретки).

При передаче значения `pre` свойству `white-space` пробелы из текста элемента *не* удаляются, сколько бы их ни было в строке, поскольку значение рассматривается браузером как элемент `pre` языка XHTML (рис. 6.37).

```
p {white-space: pre;}

<p>В этом    абзаце   слишком     много        пробелов.</p>
```

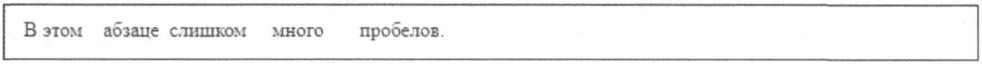


Рис. 6.37. Сохранение пробелов в исходном HTML-коде

Наряду с пробелами значение `pre` позволяет сохранить в тексте документа символы возврата каретки и переноса слов. В этом смысле (и никаком другом) значение `pre` наделяет любой из элементов свойствами элемента `pre`.

Значение `nowrap` с противоположной направленностью действий обязывает браузер не учитывать лишние пробелы и не переносить текст элемента, за исключением случаев включения в его разметку тегов `<br>`. Элемент, к которому применено свойство `white-space` со значением `nowrap`, обрабатывается так же, как и ячейки таблицы, размеченные тегами `<td nowrap>` в HTML4. Различие между командами состоит в том, что значение `nowrap` можно установить для любого элемента документа, не обязательно `td`. Результат выполнения приведенного ниже фрагмента в браузере показан на рис. 6.38.

```
<p style="white-space: nowrap;">Переносы в этом абзаце запрещены,
поэтому для завершения текущей строки необходимо добавить в нее
элемент br. В противном случае текстовая строка будет продолжена
до бесконечности, а для просмотра скрытого содержимого придется
воспользоваться горизонтальной полосой прокрутки, <br/>добавляемой
в окно браузера.</p>
```

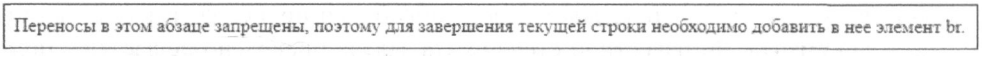


Рис. 6.38. Предотвращение переносов слов свойством `white-space`

Кроме того, свойством `white-space` удобно заменять атрибут `nowrap` ячеек таблицы.

```
td {white-space: nowrap;}

<table><tr>
<td>Содержимое этой ячейки не переносится на новую строку.</td>
<td>Как и этой ячейки.</td>
<td>И этой, последующей, а также любой другой ячейки таблицы.</td>
<td> Свойство white-space запрещает переносы.</td>
</tr></table>
```

В CSS2.1 свойство `white-space` получило два новых значения, `pre-wrap` и `pre-line`, позволяющих предельно точно устанавливать способ обработки пробелов и переносов в тексте элемента.

Значение `pre-wrap` указывает сохранять в тексте только все исходно добавленные пробелы, но запрещает переносы текста на новые строки. Распознав это значение, браузер будет игнорировать все символы разрыва строки, найденные в тексте элемента. Значение `pre-line`, наоборот, обязывает браузер не учитывать дополнительные пробелы между словами, но обрабатывать символы разрыва строки, установленные автором документа. Пример их использования в одном документе приведен в следующем листинге, результат выполнения которого показан на рис. 6.39.

```
<p style="white-space: pre-wrap;">
В тексте     этого  абзаца содержатся     лишние п р о б е л ы, но
    запрещены   разрывы и переносы     строк.
</p>
<p style="white-space: pre-line;">
В тексте     этого  абзаца содержатся     лишние п р о б е л ы, но
разрешены   разрывы и переносы     строк.
</p>
```

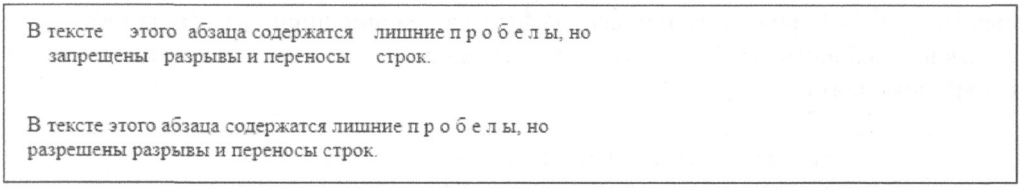


Рис. 6.39. Два способа обработки пробелов и символов разрыва строки в абзацах

Общие сведения о значениях свойства `white-space` приведены в табл. 6.1.

Таблица 6.1. Значения свойства `white-space`

Значение	Пробелы	Разрывы строк	Перенос строк
`pre-line`	Учитываются	Игнорируются	Разрешен
`normal`	Учитываются	Игнорируются	Разрешен
`nowrap`	Учитываются	Игнорируются	Запрещен
`pre`	Сохраняются	Обрабатываются	Запрещен
`pre-wrap`	Сохраняются	Обрабатываются	Разрешен

Ширина отступа табуляции

Запрещая отображение лишних символов пробела свойством `white-space`, можно попробовать увеличить интервал между словами элемента, используя символы табуляции (код 0009 в Unicode). В CSS ширина отступа табуляции устанавливается с помощью свойства `tab-size`.

tab-size

Значение	*<length>* | *<integer>*
Начальное значение	8
Применяется	Блочные элементы
Вычисляется	Абсолютное значение в единицах измерения длины
Наследуется	Да
Анимируется	Да

По умолчанию отступ, задаваемый символом табуляции, равен пространству, занимаемому восемью символами пробела. При необходимости его можно увеличить, передавая свойству `tab-size` другое целочисленное значение. Так, например, объявление `tab-size: 4` означает добавлять в элемент отступ табуляции с шириной, равной четырем символам пробела.

Указываемое в свойстве значение немедленно применяется ко всем символам табуляции, определенным в целевом элементе. В частности, три символа табуляции, добавленных в текст элемента последовательно, при назначении ему правила, содержащего объявление `tab-size: 10px`, будут представлять отступ шириной в 30 пробелов. На рис. 6.40 продемонстрирован эффект изменения ширины отступа табуляции с помощью свойства `tab-size`, которому передаются значения, описанные в тексте HTML-документа.

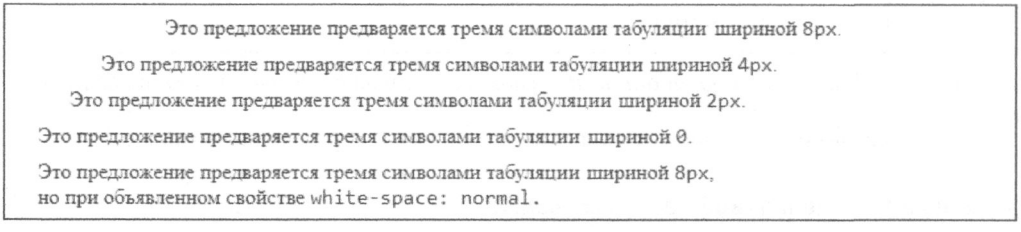


Рис. 6.40. Отступы разной ширины, устанавливаемые символами табуляции

Учтите, что отступ, устанавливаемый свойством `tab-size`, не будет отображаться в документе при запрете отображения пробелов с помощью свойства `white-space`, описанного в предыдущем разделе. Конечно, значение свойства будет рассчитываться, но не будет применяться к элементу, независимо от количества символов табуляции, добавленных в него.

На данный момент свойство `tab-size` (равно, как и вендорное свойство `-moz-tab-size`) поддерживается браузерами семейств WebKit и Gecko. Во всех них ему допускается передавать только целочисленные значения, но не значения, выраженные в единицах измерения длины.

Разрывы и переносы текстовых строк

Переносы незаменимы при включении длинных слов в элементы небольшой ширины, например в колонках новостных изданий или приложениях обмена мгновенными сообщениями на мобильных устройствах. Места возможных переносов обозначаются в тексте документа с помощью знаков мягкого переноса (U+00AD в Unicode) или специального знака `&shy` в HTML. В CSS предложен свой способ расстановки мягких переносов, далеко не всегда требующий добавления в текст элемента дополнительных символов.

hyphens

Значение	`manual` | `auto` | `none`
Начальное значение	`manual`
Применяется	Все элементы
Вычисляется	Согласно определению
Наследуется	Да
Анимируется	Нет

Значение `manual`, задаваемое по умолчанию, указывает расставлять переносы только в местах, обозначенных специальными знаками (`U+00AD` и ``). Ни в каких других местах элемента они не разрешены. В противоположность ему значение `none` полностью предотвращает расстановку переносов, даже в местах, обозначенных специальными символами `U+00AD` и `&shy`.

Наиболее примечательный способ переноса слов в тексте элемента представляется значением `auto`. Оно разрешает браузеру расставлять переносы по своему усмотрению (согласно встроенному словарю переносов) — совсем не обязательно в местах, обозначенных специальными знаками. Разумеется, принимаемое браузером решение существенно зависит от языка документа, определения в нем понятия "слово" и характерных правил расстановки переносов. Конечно, пользовательские агенты лучше справляются с расстановкой переносов, обозначенных авторами документов вручную, чем автоматически, но так происходит далеко не всегда. На рис. 6.41 продемонстрирован результат применения каждого из способов расстановки переносов, поддерживаемых свойством `hyphens`.

```
.cl01 {hyphens: auto;}
.cl02 {hyphens: manual;}
.cl03 {hyphens: none;}
```

```
<p class="cl01">Supercalifragilisticexpialidocious
antidisestablishmentarianism.</p>
<p class="cl02">Supercalifragilisticexpialidocious
antidisestablishmentarianism.</p>
<p class="cl02">Super&#xad;cali&#xad;fragi&#xad;listic&#xad;
expi&#xad;ali&#xad;dociousanti&#xad;dis&#xad;establish&#xad;
ment&#xad;arian&#xad;ism.</p>
<p class="cl03">Super&#xad;cali&#xad;fragi&#xad;listic&#xad;
expi&#xad;ali&#xad;dociousanti&#xad;dis&#xad;establish&#xad;
ment&#xad;arian&#xad;ism.</p>
```

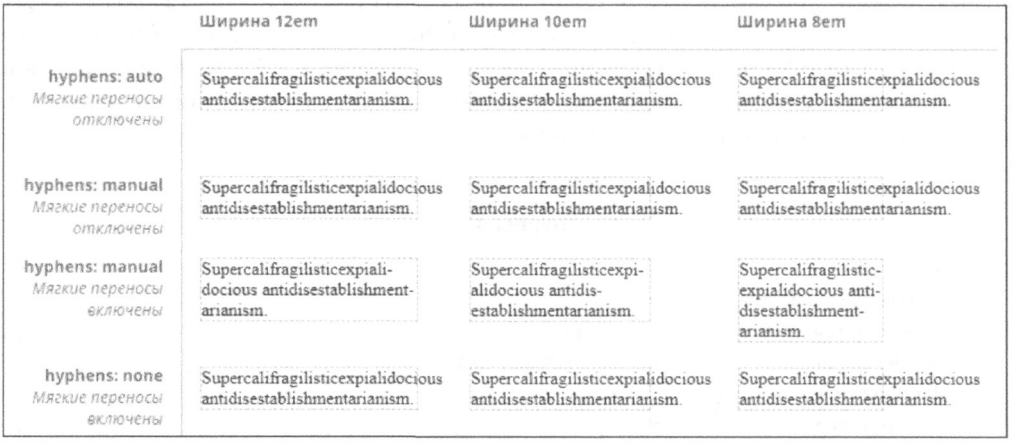


Рис. 6.41. Результат расстановки переносов с помощью стилевого свойства `hyphens`

Поскольку правила переноса слов во многом определяются языком документа и не обозначаются спецификацией CSS в явном виде, во всех пользовательских агентах это осуществляется по-разному, согласно предпочтениям, заложенным их разработчиками.

Помимо всего прочего, при использовании свойства `hyphens` необходимо учитывать несколько немаловажных факторов. Во-первых, это стилевое свойство наследуется — правило `body {hyphens: auto;}` будет применяться сразу ко всем элементам документа, включая элементы управления формы, примеры программных кодов, цитаты и т.п. Запрет переноса слов для таких элементов — весьма удачное решение, реализуемое с помощью следующего кода.

```
body {hyphens: auto;}
code, var, kbd, samp, tt, dir, listing, plaintext, xmp, abbr,
acronym, blockquote, q, textarea, input, option {hyphens: manual;}
```

Запрет переноса слов в примерах исходных кодов очевиден, особенно если они написаны на языках программирования, в которых дефис представляет неотъемлемую часть синтаксиса ключевых слов. Это же касается и текста команд, вводимых с помощью клавиатуры, — автоматическое добавление дефисов в команды Unix просто недопустимо! Каждый из элементов приведенного выше списка не требует автоматической расстановки переносов по достаточно обоснованным причинам. Чтобы

разрешить его, достаточно удалить целевой элемент из списка. (По возможности понаблюдайте за автоматическим переносом строк в тексте, вводимом в элементах управления форм!)

К концу 2017 года стилевое свойство `hyphens` поддерживалось всеми браузерами, запускаемыми на настольных компьютерах, за исключением Chrome/Blink. В Safari и Edge оно распознается только при добавлении вендорного префикса. Разумеется, способы расстановки переносов, обеспечиваемые свойством `hyphens`, сильно зависят от языка документа.

На переносы оказывают влияние и другие стилевые свойства, в частности `word-break`, определяющее правила мягкого переноса для разных языков.

word-break

Значение	`normal` | `break-all` | `keep-all`
Начальное значение	`normal`
Применяется	Все элементы
Вычисляется	Согласно определению
Наследуется	Да
Анимируется	Да

Это свойство предопределяет автоматический перенос строк, не помещающихся в элемент заданной ширины. Такой перенос называется *мягким*, в противоположность *жестким переносам*, устанавливаемым символами возврата каретки и тегами `<br>`. Свойство `word-break` всего лишь включает в тексте мягкие переносы, исходно расставляемые пользовательским агентом (или операционной системой, в которой он запущен).

Значение по умолчанию `normal` предписывает придерживаться стандартных правил переноса: строки не переносятся или переносятся в тех местах, где явно задан перенос. На практике текстовые строки чаще всего разрываются между словами, исходя из синтаксического определения термина "слово" в каждом из используемых в документе языков. Так, в языках романо-германской группы, к которым относится английский, слово определяется как последовательный набор символов, обособленный от остальной части предложения пробелами. А вот в идеографических языках, подобных японскому, словом является каждый символ, поэтому строка в них может разрываться в любом месте. Иными словами, мягкие переносы в японском, корейском и китайском языках возможны только в синтаксических конструкциях, представляющих несколько слитно написанных символов.

Значение по умолчанию включает общепринятый для браузеров механизм расстановки переносов, оттачиваемый разработчиками на протяжении многих лет. С другой стороны, значение `break-all` обязывает добавлять мягкие переносы между любыми двумя последовательно расположенными символами каждого слова.

В подобных случаях перенос выполняется без добавления дефиса в конец строки, даже при явном указании его применения с помощью свойства `hyphens` (см. предыдущий раздел). Результат применения свойства `word-break` к элементам, содержащим текст на японском, корейском и китайском языках, при передаче ему значения `break-all` сильно зависит от значения, назначенного свойству `line-break`.

Значение `keep-all`, в противоположность значению `break-all`, запрещает мягкие переносы между словами, даже в текстах, введенных на японском, корейском и китайском языках, где каждое слово представляется отдельным символом. Например, текстовая строка, содержащая текст на японском языке, не будет разрываться по не разделенным пробелами символам, даже если ее длина превышает ширину элемента. (Такой же эффект достигается с помощью объявления `white-space: pre`.)

Результат разрыва текстовой строки свойством `word-break` при задании ему описанных выше значений показан на рис. 6.42.

Рис. 6.42. Разрыв текстовой строки в зависимости от настроек мягкого переноса

Общие сведения о значениях свойства `word-break` приведены в табл. 6.2.

Таблица 6.2. Переносы и значения свойства `word-break`

Значение	Европейские языки	Японский, корейский и китайский языки	Дефисы
`normal`	Обычный порядок	Обычный порядок	Разрешены
`break-all`	После любого символа	После любого символа	Запрещены
`keep-all`	Обычный порядок	Вне последовательностей символов	Разрешены

Если вам часто приходится форматировать тексты на японском, корейском и китайском языках, то обратите внимание на стилевое свойство `line-break`.

line-break

Значение	`auto` | `loose` | `normal` | `strict`
Начальное значение	`auto`
Применяется	Все элементы
Вычисляется	Согласно определению
Наследуется	Да
Анимируется	Да

Как вы только что увидели, свойство `word-break` расставляет мягкие переносы в текстах на японском, корейском и китайском языках. С помощью свойства `line-break` уточняется, какое влияние на расстановку переносов оказывают специальные символы и знаки пунктуации (дефис, многоточие, восклицательный и вопросительный знаки и т.п.), включаемые в текст, который введен на обозначенных языках.

Другими словами, свойство `line-break` применяется к символам японского, корейского и китайского языков на постоянной основе. В частности, если в русский текст добавить несколько китайских иероглифов, то свойство `line-break` будет воздействовать только на них. Наряду с этим при включении в текст на китайском языке символов, заимствованных из других языков, свойство `line-break` будет применяться только к ним всем, но не к остальной части элемента. При расстановке мягких переносов в текстах, введенных на японском, корейском и китайском языках, браузерами учитываются знаки пунктуации, символы валюты и некоторые другие символы, заимствованные из других языков.

Полного списка символов, обязательных для обработки свойством `line-break`, попросту не существует — спецификация CSS приводит рекомендации (`http://w3.org/TR/css3-text/#line-break`) по управлению переносами только для некоторых типов специальных знаков.

По умолчанию применяется значение `auto`, определяющее расстановку мягких переносов пользовательским агентом и, что более важно, позволяющее принимать разные решения в разных ситуациях. Например, к длинным строкам пользовательский агент будет применять строгий набор правил переноса слов, который будет заметно смягчаться по мере сужения элемента. С функциональной точки зрения значение `auto` позволяет переключаться между режимами `loose`, `normal` и `strict` произвольным образом — чаще всего построчно в пределах одного элемента.

Указанные режимы расстановки переносов представляются в свойстве `line-break` отдельными ключевыми словами и могут запускаться обособленно от остальных.

`loose`

Переносы выполняются согласно менее строгому набору правил, применяемому преимущественно к коротким текстовым строкам.

`normal`

Представляет наиболее распространенный набор правил переноса слов при разрыве строк. Они не определены в явном виде — в спецификации приведены только общие рекомендации по обработке некоторых типов специальных символов.

`strict`

Позволяет переносить слова согласно наиболее строгому набору правил, как и ранее не имеющему точного определения.

Разрывы строк

Вооружившись знаниями о принципах расстановки переносов описанными выше свойствами CSS, попробуем разобраться, что происходит, когда текстовая строка не помещается в контейнере элемента. В CSS возможность разрыва и переноса таких строк определяется свойством `overflow-wrap`.

overflow-wrap (прежнее word-wrap)

Значение	`normal | break-word`
Начальное значение	`normal`
Применяется	Все элементы
Вычисляется	Согласно определению
Наследуется	Да
Анимируется	Да

Это очень разностороннее свойство. Его значение `normal`, устанавливаемое по умолчанию, предопределяет стандартные правила разрыва строк — по пробелам между словами или согласно правилам, принятым для текущего языка элемента. Передача свойству значения `break-word` предоставляет возможность разрывать слова между любыми двумя символами. Результат переноса текста на следующую строку в обоих режимах показан на рис. 6.43.

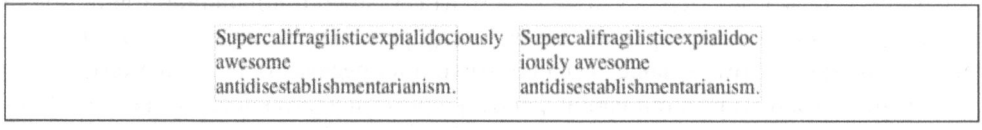


Рис. 6.43. Перенос строки, не помещающейся в контейнер элемента

Свойство `overflow-wrap` вступает в силу только при передаче свойству `white-space` значений, разрешающих перенос слов. В противном случае (т.е. при значении `pre`) свойство `overflow-wrap` эффекта не возымеет.

Свойство `overflow-wrap` имеет длинную предысторию развития и реализации в CSS. В былые времена соответствующую функцию выполняло свойство `word-wrap`. Оно было настолько подобно свойству `overflow-wrap`, что современная спецификация CSS в явном виде указывает пользовательским агентам обрабатывать их абсолютно одинаково (дословно: рассматривать свойство `word-wrap` как свойство `overflow-wrap` с иным именем).

К сожалению, браузеры далеко не всегда следуют установкам, предопределенным спецификацией CSS, поэтому свойство `word-wrap` поддерживается ими намного лучше. Чтобы обеспечить обратную совместимость свойств с разными названиями, но одинаковым назначением, применяется следующее стилевое правило:

```
pre {word-wrap: break-word; overflow-wrap: break-word;}
```

К концу 2017 года свойство `overflow-wrap` распознавалось большинством популярных браузеров и не имело ограничений на использование.

Объявление `overflow-wrap: break-word` подобно своему предшественнику — `word-break: break-all`, но далеко не во всем. Чтобы понять разницу, сравните содержимое второго контейнера на рис. 6.43 и среднего верхнего контейнера на рис. 6.42. Легко заметить, что свойство `overflow-wrap` разрывает строки только в случае включения в контейнер строки, длина которой превышает его ширину. Если бы строка включала пробел, то она непременно разрывалась бы по нему. В то же время объявление `word-break: break-all` определяет разрыв строки по краю контейнера, независимо от того, есть в ней пробелы или нет.

Направление письма

Текст этой книги, как и многих других книг, написанных на большинстве европейских языков, необходимо читать слева направо и сверху вниз. Но далеко не каждый язык придерживается указанного направления текста. Существует много языков с направлением текста справа налево (например, арабский и иврит) и даже с вертикальным письмом (в частности, китайский и японский). В двух последних вертикальные строки заполняют страницу слева направо, но существуют языки (например, монгольский), в которых они следуют друг за другом справа налево.

Определение направления письма

В CSS направление письма устанавливается с помощью отдельного свойства `writing-mode`, принимающего одно из трех значений.

writing-mode

Значение	`horizontal-tb` | `vertical-rl` | `vertical-lr`
Начальное значение	`horizontal-tb`
Применяется	Все элементы, за исключением элементов ячеек строк и столбцов таблицы, ячеек строчного уровня и уровня столбцов, контейнеров и элементов аннотаций `ruby`
Вычисляется	Согласно определению
Наследуется	Да
Анимируется	Да

Значение `horizontal-tb`, принятое по умолчанию, определяет горизонтальное направление текста строчных элементов и вертикальный порядок заполнения документа блочными элементами. Это характерно для большинства европейских и некоторых ближневосточных языков, которые могут различаться направлением текста в строчных элементах. Остальные два значения определяют вертикальное направление (письма) строчных и горизонтальное (справа налево или слева направо) расположение блочных элементов. Все три способа заполнения элементов текстом представлены на рис. 6.44.

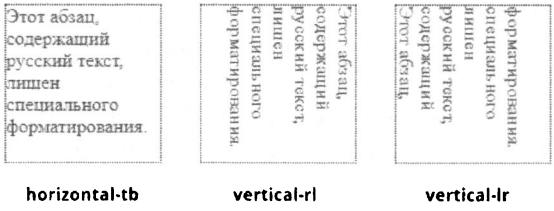


Рис. 6.44. Различные направления письма

Обратите внимание на порядок следования строк в элементах с вертикальным направлением письма. Если наклонить голову вправо, то можно прочесть предложение в блоке, направление текста в котором определяется значением `vertical-rl`. Текст последнего блока, направление письма в котором задается ключевым словом `vertical-lr`, нужно читать начиная с последней строки, что более чем непривычно. Несомненно, сделать это не составит большого труда только людям с другими культурными традициями, например японцам. К концу 2017 года спецификация CSS оговаривала только один вариант вертикального письма: сверху вниз.

Можно изменить вертикальное направление письма (для европейских языков) на противоположное, применив к элементу свойство `writing-mode` со значением `vertical-rl` и повернув его на 180° с помощью функций модуля CSS Transform (глава 16). В результате элементы будут заполняться текстом снизу вверх и слева направо. Если в предыдущем методе использовать ключевое слово `vertical-lr`, то после поворота элемент будет характеризоваться направлением письма снизу вверх и справа налево (рис. 6.45).

```
.flip {transform: rotate(180deg);}
#one {writing-mode: vertical-rl;}
#two {writing-mode: vertical-lr;}
```

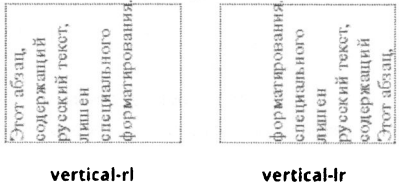


Рис. 6.45. Инвертирование направления вертикального текста

Текст, получаемый с помощью описанных выше методов, оказывается инвертированным: направление вставляемого в документ текста прямо противоположно определяемому свойством `writing-mode`. Несмотря на то что значение `vertical-rl` устанавливает (вертикальное) направление текста справа налево, текст будет заполнять элемент слева направо.

Аналогичные трудности возникают при выравнивании такого текста свойствами блочного уровня, например `vertical-align`. В вертикальных системах письма вертикальное выравнивание выполняется в горизонтальном направлении, поэтому свойство `vertical-align` будет задавать смещение влево или вправо. Результат "вертикального" выравнивания вертикально ориентированного текста показан на рис. 6.46.

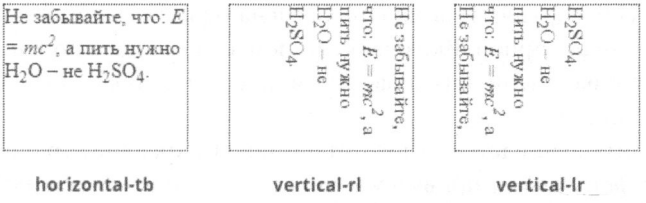


Рис. 6.46. "Вертикальное" выравнивание вертикально направленного текста

Добавление в текст элементов верхнего и нижнего индексов смещает базовую линию элементов даже при выравнивании их с помощью свойства `vertical-align`. Как известно, базовая линия элемента смещается относительно базовой линии контейнера строки, которая рассматривается в CSS как горизонтальная, даже при вертикальном направлении представления текста.

Несколько обескураживает, не правда ли? Изменение направления текста всегда вызывает трудности, поскольку требует использования совершенно иной терминологии и способов обработки данных, порой противоречащих доктринам, принятым в ранних спецификациях CSS. По правде говоря, чтобы правильно отражать назначение, стилевое свойство, обеспечивающее выравнивание строчных элементов с вертикально направленным тестом, должно называться не `vertical-align`, а `inline-align`. (Хочется верить, что в будущих версиях CSS этот нюанс будет обязательно учтен.)

Последнее замечание: когда с помощью инструментов модуля CSS Transforms трансформируются элементы с вертикальным текстом, его нельзя рассматривать как горизонтальный текст, повернутый на 90°, по двум веским причинам. Во-первых, правильный горизонтальный вид принимает только вертикальный текст, выровненный с помощью свойства `vertical-align` со значением `vertical-rl`. Если вертикальный текст был выровнен с помощью значения `vertical-lr`, то после преобразования он будет иметь направление снизу вверх, а не сверху вниз, как полагается правильно расположенному горизонтальному тексту. Во-вторых, включение в элемент вертикального текста не изменяет ориентацию его контейнера: верх и низ элемента будет располагаться в исходных позициях. Чтобы понять, о чем идет речь, взгляните на рис. 6.47, полученный для документа, к которому применены следующие стилевые правила.

```
.boxed {border-top: 3px solid red; border-left: 3px dashed tan;}
#one {writing-mode: vertical-rl;}
#two {writing-mode: vertical-lr;}
```

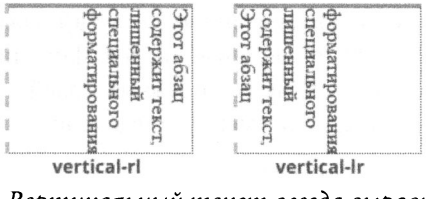


Рис. 6.47. Вертикальный текст всегда выравнивается относительно одной из сторон элемента

В обоих случаях верхняя граница обозначена более жирной (красной) линией, а левая граница — пунктирной. Включение в элемент вертикально направленного текста не приводит к изменению их положения, поскольку текст получен отличным от вращения способом.

С этого момента начинается самое интересное! Сохранение границами исходного положения *не гарантирует* привязку к ним полей элемента, что не предполагается спецификацией CSS.

Поворот полей вызван стремлением пользовательского агента сохранить за элементом стили, привязанные к его началу и к концу относительно направления заполнения документа блочными элементами. Если это направление вертикальное, то начало и конец элемента совпадают с его верхней и нижней границами. Блочные элементы, содержащие вертикальный текст, заполняют документ в горизонтальном направлении: слева направо или справа налево. Таким образом, расположив последовательно несколько абзацев вертикального текста, можно обнаружить, что их верхнее и нижнее поля отсчитываются от левой и правой границ элементов. Подобный эффект наблюдается при применении к двум элементам `p` следующих правил (рис. 6.48).

```
p {margin-top: 1em;}
#one {writing-mode: vertical-rl;}
#two {writing-mode: vertical-lr;}
```

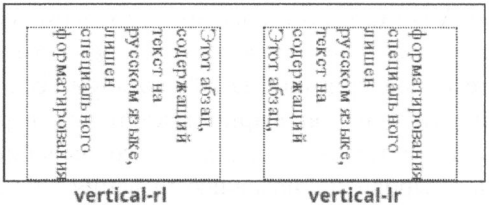


Рис. 6.48. Расположение полей у элементов с вертикальным текстом

Как предполагается первым стилевым правилом, верхнее поле отсчитывается от верхней границы элементов, а поля слева и справа устанавливаются пользовательским агентом как начальное и конечное для каждого элемента. При ручной настройке всех полей элементов, содержащих вертикальный текст, этого не происходит. Если в явном виде обнулить левое и правое поля, то свободное пространство между абзацами будет удалено.

Не забывайте, что такое поведение вызвано установкой полей текстовых элементов с помощью определяемых по умолчанию свойств пользовательского агента, подобных `block-start-margin` (не представлено спецификацией CSS, по крайней мере пока). При указании верхнего, нижнего и боковых полей с помощью стилевых правил CSS они будут добавляться к элементу предельно корректно — так же, как и границы.

Ориентация символов

Определившись с направлением текста, можно заняться изменением ориентации его символов. Существует несколько причин, побуждающих к изменению ориентации символов в текстовой строке, наиболее распространенная из которых заключается в смешивании в одной строке текста, набранного на разных языках или относящихся к разным системам письма (например, при добавлении в японские тексты английских слов или арабских цифр). За решение такой задачи в CSS отвечает свойство `text-orientation`.

text-orientation

Значение	`mixed` | `upright` | `sideways`
Начальное значение	`mixed`
Применяется	Все элементы, за исключением элементов ячеек строк и столбцов таблицы, ячеек строчного уровня и уровня столбцов
Вычисляется	Согласно определению
Наследуется	Да
Анимируется	Да

Как предполагает название раздела, свойство `text-orientation` определяет ориентацию символов элемента. Чтобы ознакомиться с выполняемыми им действиями, рассмотрим следующий пример.

```
.verts {writing-mode: vertical-lr;}
#one {text-orientation: mixed;}
#two {text-orientation: upright;}
#thr {text-orientation: sideways;}
```

Рис. 6.49. Ориентация символов

В верхней части рис. 5.49 приведен абзац, содержащий текст на русском и японском языках, копии которого с вертикальным направлением текста (устанавливается свойством `vertical-align` со значением `vertical-lr`) добавлены в каждый из

трех расположенных ниже блоков. К первому из трех блоков применено свойство `text-orientation: mixed`, сохраняющее исходную ориентацию символов для каждой из систем письма (в английском тексте символы располагаются сбоку друг от друга, а в японском — один под другим). Во втором блоке все символы текстового элемента располагаются в строке один под другим, согласно требованиям ключевого слова `upright`. Значение `sideways` указывает располагать все символы строки (в том числе и японские) сбоку друг от друга.

Изменение направления текста

Изменение направления текста в CSS2 требовало применения к элементу сразу двух свойств: `direction` и `unicode-bidi`. С их помощью устанавливалось необходимое направление текста для строчных элементов.

Текущая спецификация CSS предостерегает от использования свойств `direction` и `unicode-bidi` в стилевых правилах, применяемых к документам HTML. Дословно это выражается так: "Поскольку пользовательские агенты могут отключать стили CSS... рекомендуется избегать изменения направления текста в документах HTML с помощью тега `<bdo>` и атрибута `dir`". Обозначенные свойства рассматриваются только по причине использования их в старых таблицах стилей, все еще применяемых для форматирования HTML-документов.

direction

Значение	`ltr` | `rtl`
Начальное значение	`ltr`
Применяется	Все элементы
Вычисляется	Согласно определению
Наследуется	Нет
Анимируется	Да

Свойство `direction` определяет направление отображения текста в блочных элементах, порядок представления столбцов таблицы, способ обработки текста, не помещающегося в контейнер элемента, и положение последней строки текстового блока при выравнивании его содержимого по ширине (`text-align: justify`). Для использования его со строчными элементами значение свойства `unicode-bidi` должно быть задано как `embed` или `bidi-override` (см. описание свойства `unicode-bidi`).

По умолчанию свойство `direction` имеет значение `ltr`, устанавливающее направление текста для элемента слева направо. Тем не менее оно может быть спонтанно изменено на `rtl` (справа налево) согласно внутренним инструкциям пользовательского агента, который выполняет установки собственного правила, имеющего такой вид:

```
*:lang(ar), *:lang(he) {direction: rtl;}
```

Конечное правило, выполняемое пользовательским агентом, намного длиннее, поскольку в нем перечисляются не только арабский язык и иврит, но даже в таком виде оно дает представление о способе изменения направления текста браузерами.

Само по себе свойство `direction` указывает только способ однонаправленного отображения текста. Для поддержки документом двух направлений представления текста нужно прибегнуть к услугам встроенных возможностей Unicode, активизируемых в CSS с помощью свойства `unicode-bidi`.

unicode-bidi

Значение	`normal` | `embed` | `bidi-override`
Начальное значение	`normal`
Применяется	Все элементы
Вычисляется	Согласно определению
Наследуется	Нет
Анимируется	Да

Для того чтобы получить представление о смысле каждого из его значений, достаточно изучить их детальное описание, приведенное в спецификации CSS2.1.

`normal`

Для элемента двунаправленный алгоритм отображения текста не задействуется (не внедряется). Безусловному переупорядочению подлежат только все символы включенных в него встроенных (строчных) элементов.

`embed`

Разрешает устанавливать отдельный порядок представления символов в элементах строчного уровня согласно алгоритму двунаправленного вывода. Дополнительное направление представления символов определяется значением свойства `direction`. Переупорядочению подлежат сразу все символы внутри элемента. Это соответствует добавлению управляющего символа LRE (U+202A; объявление `direction: ltr`) или RLE (U+202B; объявление `direction: rtl`) в начало элемента и управляющего символа PDF (U+202C) в конец элемента.

`bidi-override`

Разрешает задавать отдельное направление представления текста для строчных элементов или элементов блочного уровня, содержащих только строчные элементы. Это значит, что внутри элемента порядок следования символов определяется значением свойства `direction`, а безусловная часть алгоритма двунаправленного вывода игнорируется. Это соответствует добавлению управляющих символов LRO (U+202D; объявление `direction: ltr`) или RLO (U+202E; объявление `direction: rtl`) в начало элемента и управляющего символа PDF (U+202C) в конец элемента.

Резюме

Внешний вид текста можно изменить, не прибегая к перенастройке параметров шрифтов. С помощью CSS текст можно не только оформить в классическом стиле, снабдив подчеркиванием, но и применить к нему более совершенные эффекты: перечеркнуть строку, изменить интервал между словами и символами, задать отступ первой строки для абзаца (блочного элемента), выровнять текст одним из множества возможных способов, расставить переносы, образовать разрывы строк и т.п. Кроме того, в тексте можно изменить междустрочный интервал и установить другое направление письма, характерное для азиатских языков. Важность стилевых свойств, выполняющих обозначенные задачи, для веб-дизайна сложно переоценить. Но они являются лишь малой частью средств, призванных улучшить внешний вид и повысить удобочитаемость документов.

ГЛАВА 7

Основы визуального оформления документа

В этой главе рассматриваются теоретические основы визуального представления содержимого документа. С ними нужно обязательно ознакомиться перед тем, как изучать сложные методики оформления документов, требующие комбинирования самых разных свойств и специальных эффектов. Процесс обучения неразрывно связан с рассмотрением новых возможностей CSS. По мере практического применения инструментов CSS в реальных документах вы рано или поздно столкнетесь с непредсказуемым поведением пользовательских агентов. Имея всестороннее представление о том, как визуализируется документ, вы сможете однозначно определить причину неправильного представления элементов, оформление которых выполняется с помощью стилевых правил.

Контейнеры

В CSS все элементы заключаются в контейнеры прямоугольной формы, называемые *контейнерами элементов*. (Вполне вероятно, в будущих версиях спецификации появятся контейнеры других форм, но на данный момент элементы заключаются только в прямоугольные контейнеры.) В середине каждого контейнера находится *область содержимого*, отделяемая от края контейнера отступами, полями и границами. Добавлять их к элементу не обязательно — для удаления одного из них достаточно обнулить значение соответствующего свойства. Схематически контейнер элемента, включающий область содержимого, поля, отступы и границы, показан на рис. 7.1.

Поля, отступы и границы задаются для каждой из сторон элемента с помощью отдельных свойств, подобных `margin-left` и `border-bottom`. Чтобы задать их сразу для всех четырех сторон, применяются свойства общего назначения, в именах которых отсутствуют названия сторон контейнера, например `padding`. Внешний контур контейнера элемента (если задан) определяется сразу для всех сторон и не может быть установлен отдельно для каждой из них. Фоновая заливка или фоновое изображение добавляются не только к области содержимого, но и к пространству,

образованному полями элемента. Отступы всегда прозрачны — через них просматривается фон родительского элемента. Поля, в отличие от отступов, не могут иметь отрицательную ширину. Результат задания элементам отрицательных отступов рассматривается далее.

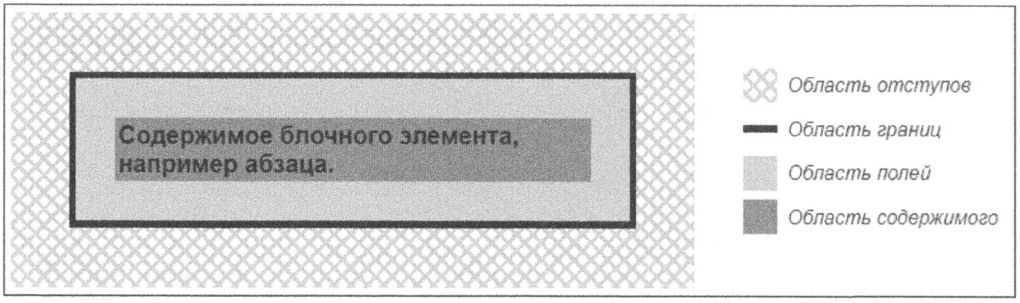


Рис. 7.1. Область содержимого, окруженная полями, отступами и границами

Стиль границ элемента устанавливается отдельными свойствами, такими как `solid` и `inset`, а их цвет определяется с помощью свойства `border-color`. Если цвет границы специально не оговаривается, то они окрашиваются основным цветом элемента. В частности, если тексту абзаца задан белый цвет, то и границы элемента по умолчанию будут белыми. Чтобы заключить текст абзаца в рамку другого цвета, его нужно задать отдельно. Если линия границы не сплошная, то через ее разрывы будет просматриваться фон элемента. Разумеется, ширина границы не может представляться отрицательным значением.

Отдельные параметры контейнера элемента регулируются большим количеством свойств, подобных `width` и `border-right`. Многие из них будут применяться в примерах, рассматриваемых в книге, хотя их определения здесь не описаны.

Основные определения

Введем основные определения, которыми мы будем оперировать в дальнейшем, и вспомним, какие типы элементов доступны для использования в документах.

Обычное направление заполнения текстом (поток)

В европейских языках текст заполняет элемент (документ HTML) слева направо и сверху вниз. Остальные языки могут характеризоваться другим направлением расположения содержимого. Большинство элементов придерживается обычного порядка расположения текста, который может быть нарушен при ручной перенастройке обтекания и позиционирования элемента, а также при включении его во flex-контейнер или при верстке по сетке. В этой главе рассматриваются только элементы, характеризующиеся обычным способом заполнения текстом.

Незамещаемый элемент

Содержимое таких элементов хранится в самом документе. К незамещаемым элементам, например, относятся абзацы, поскольку содержат текст, расположенный в них самих.

Замещаемый элемент

Такие элементы являются заполнителями для содержимого, хранящегося отдельно от документа. Наиболее распространенный замещаемый элемент — это `img`, который представляет графическое изображение, хранящееся во внешнем файле. К замещаемым также относятся элементы управления, добавляемые на форму (обозначенные тегом `<input type="radio">`).

Корневой элемент

Элемент, включающий все остальные элементы документа. С него начинается иерархическая структура документа. В HTML он представляется элементом `html`, а в XML — любым элементом, выступающим в качестве корневого в текущем расширении языка. Например, в RSS-файлах корневой элемент называется `rss`.

Контейнер блочного элемента

Контейнер элемента блочного типа — абзаца, заголовка или универсального элемента `div`. Контейнеры этого типа отделяются друг от друга символами новой строки, которые обеспечивают правильный, вертикальный, порядок наполнения документа блочными элементами. Для заключения элемента в контейнер блочного типа к нему нужно применить свойство `display` со значением `block`.

Контейнер строчного (вложенного) элемента

Контейнер вложенных элементов, подобных `strong` и `span`. Эти контейнеры не обосабливаются символами новой строки. Для объявления в документе строчного элемента к нему применятся свойство `display` со значением `inline`.

Контейнер строчно-блочного элемента

Контейнер элемента с внутренней структурой блочного элемента и поведением, свойственным строчному элементу. Он рассматривается как замещаемый элемент, но не является им по своей сути. Представьте себе элемент `div`, встроенный в текстовую строку, подобно изображению, представленному элементом строчного типа, — это и будет строчно-блочный элемент.

В документе могут быть контейнеры других типов, например контейнеры ячеек таблицы, но в этой книге они не рассматриваются по нескольким причинам. Во-первых, на сегодняшний день они очень редко применяются для верстки веб-страниц, а во-вторых, методы верстки таблиц настолько сложны, что их описание достойно отдельной книги.

Содержащий (внешний) блок

Перед рассмотрением дальнейшего материала необходимо детально ознакомиться с еще одним типом контейнеров — содержащим блоком.

Позиционирование и размер контейнера элемента рассчитываются относительно определенного прямоугольника, называемого *содержащим блоком*. Именно по нему определяется смещение элемента, устанавливаемое многочисленными стилевыми свойствами. В CSS размер содержащего блока зависит от множества факторов — в каждом конкретном случае мы будем рассматривать его отдельно.

Для обычного элемента, заполняемого содержимым слева направо и сверху вниз, содержащий блок образуется контейнером его ближайшего предка, представленного блочным элементом или элементом списка. Рассмотрим следующий пример.

```
<body>
   <div>
      <p>Это абзац.</p>
   </div>
</body>
```

В этом простейшем документе содержащим блоком для контейнера элемента `p` выступает контейнер элемента `div`, который полностью соответствует требованиям спецификации, поскольку является блочным элементом и ближайшим предком целевого элемента. Подобным образом содержащим блоком для элемента `div` выступает контейнер элемента `body`. Следовательно, позиционирование элемента `p` зависит от положения элемента `div`, которое, в свою очередь, определяется положением элемента `body`.

Продолжая цепочку взаимосвязанных элементов, нужно не забывать, что позиционирование элемента `body` определяется положением элемента `html`, образующим *начальный содержащий блок*. Он представляет элемент, уникальный для выбранного окна просмотра, — окно браузера при отображении документа на экране или печатный лист при выводе его на бумаге. Таким образом, начальный содержащий блок определяет размер окна просмотра, а не содержимого корневого элемента. Эта особенность элемента `html` не имеет большой практической ценности, но настолько примечательна, что не может не обращать на себя внимания.

Представление элемента

Для изменения способа представления элемента в документе служит свойство `display`. Оно имеет самое непосредственное отношение к визуальному форматированию документа, поэтому детально рассмотрим его возможности по позиционированию элементов в рамках описанных выше концепций.

display

Значение	[<*display-outside*> || <*display-inside*>] | <*display-listitem*> | <*display-internal*> |<*display-box*> | <*display-legacy*>
Определение	См. ниже
Начальное значение	`inline`
Применяется	Все элементы
Вычисляется	Согласно значению
Наследуется	Нет
Анимируется	Нет

<*display-outside*>
`block | inline | run-in`

<*display-inside*>
`flow | flow-root | table | flex | grid | ruby`

<*display-listitem*>
`list-item && <display-outside>? && [ flow | flow-root ]?`

<*display-internal*>
`table-row-group | table-header-group | table-footer-group | table-row | table-cell | table-column-group | table-column | table-caption | ruby-base | ruby-text | ruby-base-container | ruby-text-container`

<*display-box*>
`contents | none`

<*display-legacy*>
`inline-block | inline-list-item | inline-table | inline-flex | inline-grid`

В последующих разделах не рассматриваются значения, обеспечивающие позиционирование элементов таблиц (`table`) и аннотаций (`ruby`), поскольку они слишком сложны для изучения в текущей главе, а описание значения `list-item` опущено, так как функционально оно подобно значению `block`. Изучение свойства `display` начнем со значений, обеспечивающих форматирование обычных блочных и строчных элементов. Но перед этим остановимся на методах изменения типа элемента (кроме `inline-block`).

Изменение типа элемента

Раз уж речь зашла об оформлении документов, неплохо бы научиться изменять тип (или, в терминологии CSS, *роль*) элемента. Предположим, документ содержит набор гиперссылок, заключенных в элемент `nav`, которые необходимо расположить на вертикальной панели управления.

```
<nav>
   <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>
</nav>
```

Конечно, можно поместить гиперссылки в отдельные ячейки таблицы или перераспределить в элементе `nav`, но намного проще привести их элементы к блочному типу:

```
nav a {display: block;}
```

Это правило применяется к каждому элементу `a`, вложенному в блочный элемент `nav`. Снабдив таблицу стилей еще несколькими простыми правилами, применяемыми к элементам `a`, можно получить панель навигации, подобную показанной на рис. 7.2.

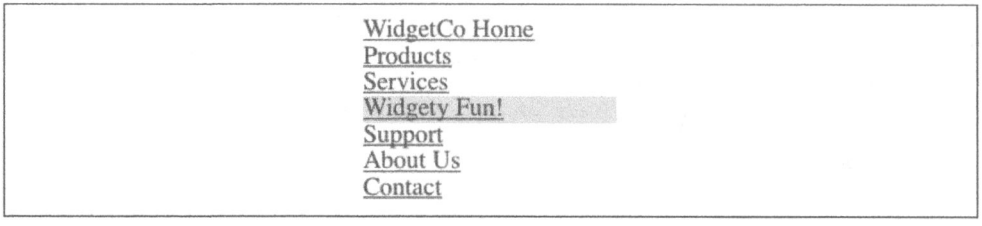


Рис. 7.2. Изменение типа элемента со строчного на блочный

Изменение роли элементов позволит правильно отображать панель навигации, исходно содержащую строчные элементы, даже браузерам, не поддерживающим CSS. Преобразовав элементы гиперссылок в блочный тип, их можно форматировать так же, как, например, элементы `div` и `p`, пользуясь тем преимуществом, что контейнеры элементов содержат одни только гиперссылки. Таким образом, для перехода по гиперссылке достаточно щелкнуть в любом месте контейнера блочного элемента.

Вам может понадобиться выполнить обратную операцию — преобразовать блочные элементы в строчные. Рассмотрим документ, включающий следующий неупорядоченный список.

```
<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>
```

На основе данной разметки нужно образовать горизонтальную панель с именами и инициалами пользователей, разделенными вертикальными линиями (в том числе расположенными на краях панели). Выполнить поставленную задачу проще всего, изменив тип элементов гиперссылок. Приведенные ниже правила изменяют исходный документ так, как показано на рис. 7.3.

```
#rollcall li {display: inline; border-right: 1px solid;
              padding: 0 0.33em;}
#rollcall li:first-child {border-left: 1px solid;}
```

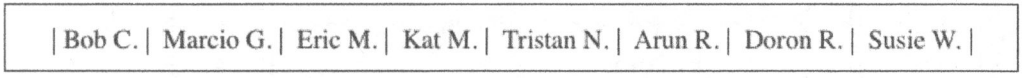


Рис. 7.3. Изменение способа представления гиперссылок на панели навигации

Несомненно, изменение типа элементов позволяет добиться более совершенных эффектов. Проявите изобретательность, и вы сможете добиться небывалых результатов!

Учтите, что приведение элемента к другому типу вызывает изменение его представления в документе, но не положения в иерархической структуре. Другими словами, преобразование абзаца в элемент строчного типа не вызывает его встраивание (с понижением уровня) в элемент более высокого уровня, что противоречит принципам структурной организации HTML-документов. (В HTML строчный элемент не может выступать родителем блочного элемента; в частности, строчный элемент `span` можно встраивать в блочный элемент `p`, но не наоборот.)

Иерархическая структура документа сохраняется даже при изменении типа элементов с помощью стилевых правил. Рассмотрим следующий фрагмент HTML-кода.

```
<span style="display: block;">
<p style="display: inline;">this is wrong!</p>
</span>
```

Данный код недействителен, поскольку блочный элемент (`p`) нельзя встроить в элемент строчного типа (`span`). Изменение типа элементов с помощью CSS не исправит ситуацию, так как влияет только на их визуальное представление, но не на положение в иерархической структуре документа.

Исходя из вышесказанного и переходя к изучению контейнеров всех основных типов (блочных, строчных, строчно-блочных элементов и элементов списков), нужно обязательно учитывать их происхождение.

Контейнеры блочных элементов

Поведение контейнеров элементов блочного типа не всегда предсказуемо — их положение в документе зависит от большого количества параметров. Чтобы научиться позиционировать контейнеры блочных элементов, необходимо изучить их структуру. Схематически блочная модель представляется так, как показано на рис. 7.4.

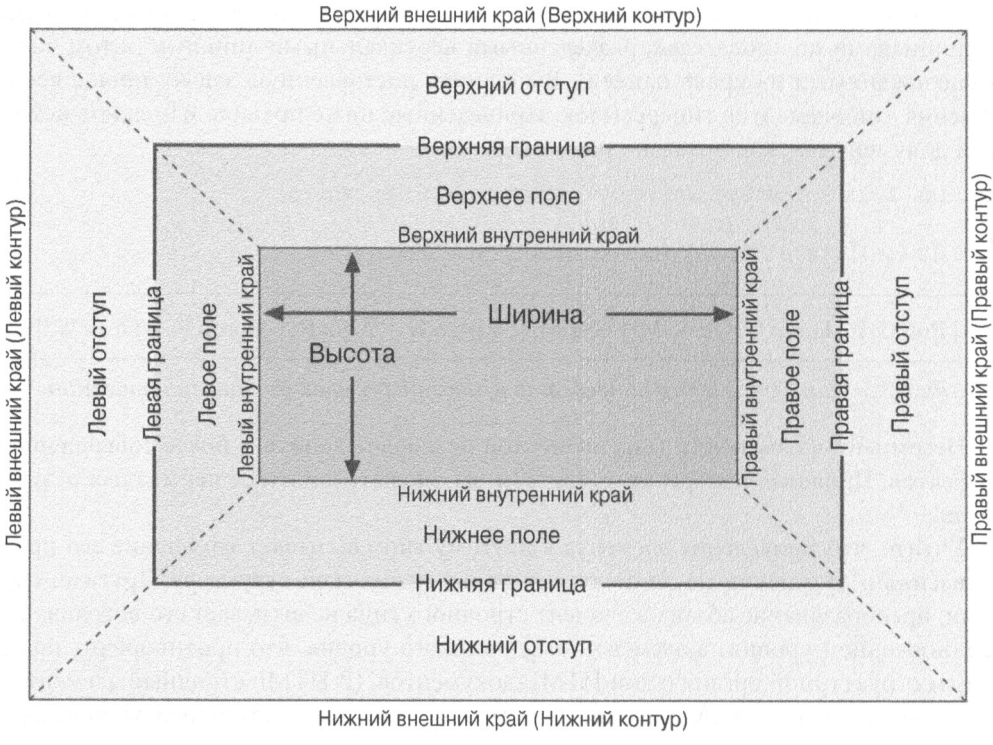


Рис. 7.4. Блочная модель элемента

По умолчанию ширина (`width`) контейнера рассчитывается как расстояние от левого до правого края прямоугольника, а высота (`height`) — от верхнего до нижнего края. Указанные свойства применимы к элементам блочного, но не строчного типа.

Алгоритм расчета размеров контейнера блочного элемента определяется с помощью свойства `box-sizing`.

box-sizing

Значение	`content-box` | `padding-box` | `border-box`
Начальное значение	`content-box`
Применяется	Все элементы, для которых объявляются свойства `width` и `height`
Вычисляется	Согласно определению
Наследуется	Нет
Анимируется	Нет

Это свойство самым непосредственным образом влияет на размер области содержимого элемента. Например, объявление `width: 400px` (в отсутствие объявления свойства `box-sizing`) задает ширину области содержимого элемента без учета полей, отступов и границ. Если же к элементу применить правило, содержащее объявление

box-sizing: border-box, то указанные выше 400 пикселей будут определять расстояние от левой до правой границы элемента. Согласно рис. 7.4, блок, обозначенный границами элемента, включает не только область содержимого, но и поля, что приводит к уменьшению ее размеров (рис. 7.5).

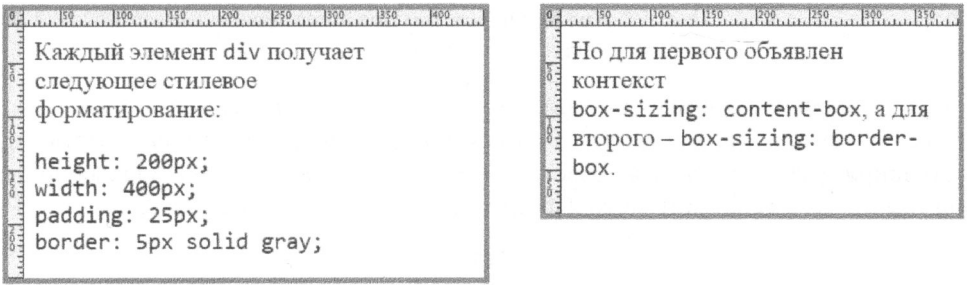


Рис. 7.5. Результат применения свойства `box-sizing`

Мы не зря начали рассмотрение инструментов позиционирования элементов со свойства box-sizing — согласно документации, оно применяется ко всем элементам, размер которых поддается настройке свойствами width и/или height. К ним относятся все элементы блочного типа, замещаемые строчные элементы, а также строчно-блочные элементы.

На позиционирование элементов в документе и его общий вид оказывают влияние отступы, поля и границы, добавленные к элементу. В большинстве случаев размер документа определяется браузером автоматически и зависит от целого ряда факторов. Для того чтобы точно задать размеры и положение элементов, применяются многочисленные стилевые свойства, рассмотренные ниже.

Горизонтальное форматирование элементов

Горизонтальное форматирование намного сложнее, чем принято думать. Трудности начинаются с объявления свойства box-sizing, по умолчанию имеющего значение content-box. Оно обязывает свойства width и height указывать размер области содержимого, а *не* видимой области элемента. Рассмотрим пример:

```
<p style="width: 200px;">Ширина?</p>
```

Встроенное правило задает ширину абзаца, равную 200px. Чтобы удостовериться в этом, достаточно добавить к абзацу фон. При добавлении отступов, полей или границы ширина абзаца будет увеличиваться на их ширину. Предположим, абзац снабжен и отступами, и полями:

```
<p style="width: 200px; padding: 10px; margin: 20px;">Ширина?</p>
```

В результате ширина видимой области абзаца составит 220 пикселей, поскольку она увеличится на ширину полей, добавляемых по обе стороны области содержимого. При вычислении общей ширины элемента также необходимо учитывать отступы (по 20 пикселей слева и справа) — в данном случае она составляет 260 пикселей, как показано на рис. 7.6.

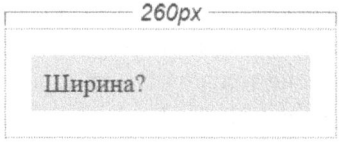


Рис. 7.6. Включение отступов в размер видимой области элемента

Если к элементу применить свойство `box-sizing` со значением `border-box`, то размер элемента будет вычисляться совсем по-другому. Его видимая область будет иметь ширину 200 пикселей, а область содержимого — 180 пикселей. Общая ширина с учетом отступов составит всего 240 пикселей, как показано на рис. 7.7.

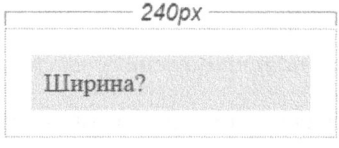


Рис. 7.7. Исключение полей из размера видимой области элемента

В каждом из случаев соблюдается правило, согласно которому ширина содержащего блока элемента равна суммарной ширине всех его горизонтальных составляющих. Для проверки этого утверждения рассмотрим два абзаца, которые включены в элемент `div` и характеризуются отступами шириной `1em` и свойством `box-sizing`, установленным в значение по умолчанию. Ширина области содержимого элемента `div` будет вычислена как суммарная ширина области содержимого (значение свойства `width`), полей, границ и отступов каждого из абзацев.

Предположим, ширина элемента `div` составляет `30em`. Исходя из этого, суммарная ширина области просмотра, полей, границ и отступов каждого абзаца также будет равняться `30em`. На рис. 7.8 отступы представлены пустыми областями белого цвета, окружающими абзацы. При добавлении к элементам `p` не только отступов, но и полей (на рисунке отсутствуют) размеры этих областей только увеличатся.

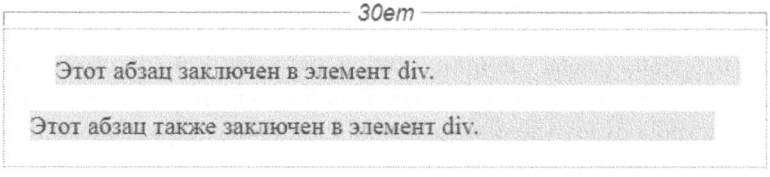


Рис. 7.8. Ширина контейнера элемента равна ширине содержащего блока

Свойства горизонтального форматирования

Горизонтальное форматирование элемента задается семью свойствами — `margin-left`, `border-left`, `padding-left`, `width`, `padding-right`, `border-right` и `margin-right`, — которые определяют горизонтальные размеры основных компонентов контейнера элемента, показанных на рис. 7.9.

Рис. 7.9. Параметры горизонтального форматирования элемента

Значения, устанавливаемые обозначенными выше свойствами, при сложении должны давать величину, равную ширине содержащего блока элемента, определяемую значением свойства `width` родительского элемента (блочные элементы почти всегда вложены в элементы блочного типа).

Из семи свойств горизонтального форматирования элементов только три можно устанавливать в значения по умолчанию: `width` (ширина области содержимого), `margin-left` и `margin-right` (левый и правый отступы). Остальные нужно представить точным числовым значением или обнулить. Компоненты контейнера элемента, допускающие автоматическое определение ширины, показаны на рис. 7.10.

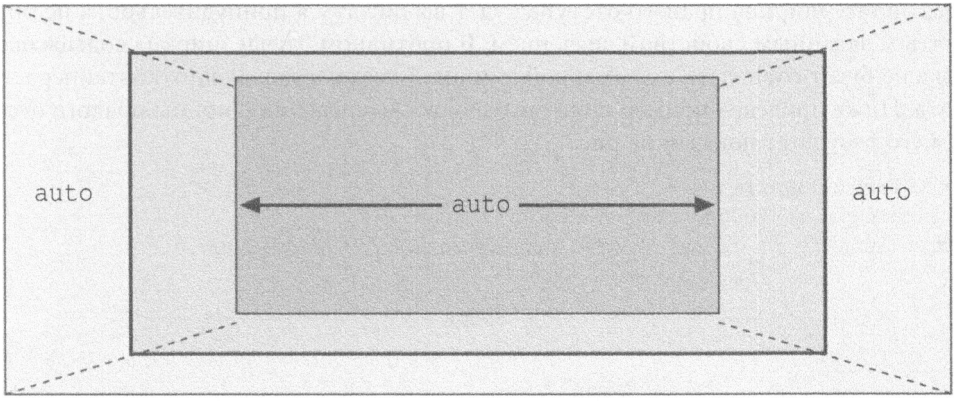


Рис. 7.10. Параметры горизонтального форматирования, допускающие автоматическую настройку

Свойство `width` предполагает либо автоматическую настройку (`auto`), либо передачу не отрицательного числового значения в допустимых единицах измерения. Автоматическая настройка свойств горизонтального форматирования элементов всегда чревата непредвиденными результатами.

Автоматическое вычисление одного свойства

При передаче значения `auto` свойствам `width`, `margin-left` и `margin-right` остальным четырем свойствам горизонтального форматирования необходимо задать точные значения, чтобы позволить браузеру правильно подобрать ширину автоматически устанавливаемых величин. Иными словами, для обеспечения равенства ширины родительского элемента суммарной ширине всех компонентов контейнера вложенного элемента некоторые из свойств нужно обязательно задавать вручную. Предположим, сумма значений всех свойств, определяющих горизонтальный размер элемента, равна 500 пикселям, ширина области содержимого и правый отступ составляют 100 пикселей, а поля и границы у элемента отсутствуют. Простые вычисления показывают, что левый отступ у такого элемента будет равен 300 пикселям.

```
div {width: 500px;}
p {margin-left: auto; margin-right: 100px;
width: 100px;} /* левый отступ, определяемый автоматически,
                  равен 300px */
```

В общепринятом понимании значение `auto` рассчитывается как разница между шириной содержащего блока и однозначно заданными значениями свойств горизонтального форматирования. В более сложной ситуации все свойства имеют строго заданные значения (например, `100px`), и ни одному из них не передается значение `auto`.

В случае передачи свойствам горизонтального форматирования *избыточных значений* ширина правого отступа *всегда* рассчитывается автоматически. Это означает, что при установке полей и отступов элемента в значение `100px` пользовательский агент применит настройку `auto` для вычисления значения свойства `margin-right`. В результате ширина правого отступа будет вычисляться принудительно, а не определяться заданным свойством значением. В противном случае ширина содержащего блока не будет совпадать с суммарной шириной всех составляющих контейнера элемента. Ниже приведен пример принудительного вычисления ширины правого отступа, а его результат показан на рис. 7.11.

```
div {width: 500px;}
p {margin-left: 100px; margin-right: 100px;
width: 100px;} /* правый отступ принудительно устанавливается
                  в значение 300px */
```

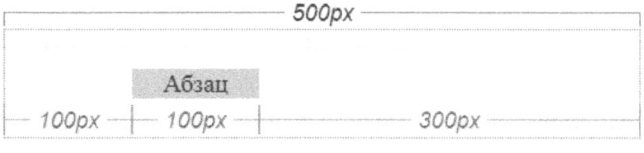


Рис. 7.11. Результат принудительного вычисления правого отступа

Если отступы задаются в явном виде, а значение `auto` получает свойство `width`, то именно оно (ширина области содержимого в контейнере элемента) вычисляется принудительно. В качестве примера рассмотрим следующее правило, результат выполнения которого приведен на рис. 7.12:

```
p {margin-left: 100px; margin-right: 100px; width: auto;}
```

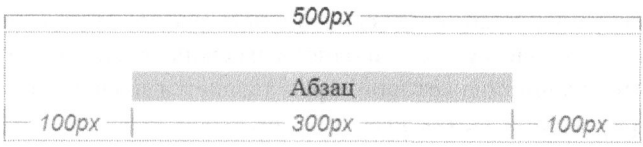


Рис. 7.12. Автоматическое вычисление ширины области содержимого

Приведенный пример — наиболее показательный, поскольку соответствует ситуации, в которой свойство `width` вообще не объявляется для элемента. Таким образом, приведенное ниже стилевое правило выполняет такие же действия, как и предыдущий код.

```
p {margin-left: 100px; margin-right: 100px;}
```

Осталось выяснить, что произойдет при передаче свойству `box-sizing` значения, отличного от `content-box` (задано по умолчанию), например `padding-box`. Для ответа на этот вопрос необходимо учитывать не только ширину отступов и области содержимого, но и размер полей и границ. С целью упрощения дальнейших вычислений предполагается, что целевой элемент не имеет полей и границ. Справедливости ради стоит заметить, что принципы вычисления свойства `width: auto` пользовательским агентом, описанные в этом и последующих разделах, не зависят от значения свойства `box-sizing`. Последнее утверждение становится тем более очевидным, если учесть, что свойство `box-sizing` определяет тип контейнера, ширина которого определяется свойством `width`, а не способ его вычисления.

Автоматическое вычисление нескольких свойств

Рассмотрим, что произойдет при автоматическом вычислении значений двух из трех основных свойств (`width`, `margin-left` и `margin-right`) горизонтального форматирования элементов. Если ключевое слово `auto` передается свойствам, определяющим ширину отступов, как показано в следующем коде, то при вычислении они получают одинаковые числовые значения, что соответствует выравниванию элемента по центру его родителя. Одна из возможных ситуаций приведена на рис. 7.13.

```
div {width: 500px;}
p {width: 300px; margin-left: auto; margin-right: auto;}
/* ширина каждого отступа равна (500-300)/2 = 100 пикселей */
```

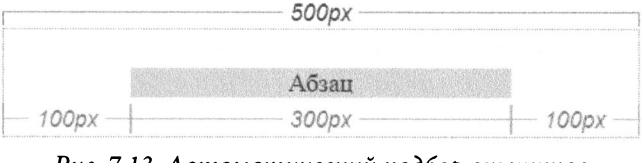


Рис. 7.13. Автоматический подбор отступов

Установка одинаковых отступов является оптимальным способом центрирования содержимого элемента в контейнере блочного типа (разумеется, при стандартном способе заполнения его текстом). Несомненно, эту же задачу можно решить и другими способами (с помощью flex-контейнеров и инструментов верстки элементов по сетке), но в этой главе они не рассматриваются.

Следующая ситуация, требующая специального рассмотрения, заключается в задании значения `auto` свойству `width` и одному из свойств настройки отступов. В подобном случае отступ, ширина которого представляется значением `auto`, обнуляется, как продемонстрировано в следующем примере.

```
div {width: 500px;}
p {margin-left: auto; margin-right: 100px;
width: auto;} /* левое поле ужимается до нулевой ширины; ширина
                 области содержимого составляет 400px */
```

Значение свойства `width` рассчитывается в предположении, что область содержимого расширяется на весь контейнер элемента — 400 пикселей в предыдущем примере (рис. 7.14).

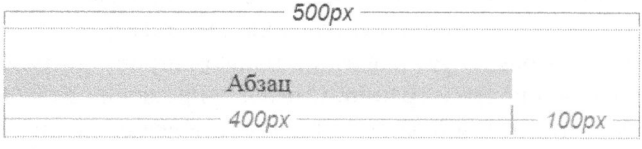


Рис. 7.14. Результат автоматической настройки левого отступа и ширины области содержимого

Последней рассмотрим автоматическую настройку трех основных свойств. Здесь все просто: если все три размера представляются ключевым словом `auto`, то отступы сжимаются до нулевой ширины, а свойство `width` получает максимально возможное значение. Такой же результат будет получен при установке свойств в значения по умолчанию, когда ни ширина области содержимого, ни один из отступов не заданы в явном виде (свойства `margin-left` и `margin-right` не заданы, а свойство `width` имеет значение `auto`).

Не забывайте, что горизонтальные отступы не подлежат схлопыванию, а потому определяют смещение области содержимого наряду с полями, границами и отступами родительского элемента. Величина смещения определяется как суммарная ширина всех указанных характеристик. В этом понимании размер области содержимого элемента косвенным образом влияет на смещение дочерних элементов. В частности, на рис. 5.15 показан результат стилевого форматирования абзаца с помощью такого кода.

```
div {padding: 50px; background: silver;}
p {margin: 30px; padding: 0; background: white;}
```

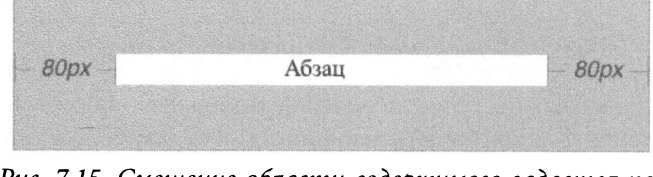


Рис. 7.15. Смещение области содержимого задается полями и отступами родительского элемента

Отрицательные отступы

До этого момента мы сталкивались только с простыми вычислениями, несмотря на мои заверения об их невероятной сложности. В этом нет обмана: трудности форматирования блочных элементов возникают при задании отрицательных отступов в их контейнерах. В них не только нет ничего предосудительного — они часто применяются для получения весьма интересных эффектов.

Как известно, сумма всех семи свойств горизонтального форматирования элемента равна значению свойства `width` родительского элемента. До тех пор пока эти свойства имеют положительные или нулевые значения, вложенный элемент не выступает за область содержимого родительского элемента. Несколько иной случай продемонстрирован на рис. 7.16. Для его получения применяются следующие стилевые правила.

```
div {width: 500px; border: 3px solid black;}
p.wide {margin-left: 10px; width: auto; margin-right: -50px; }
```

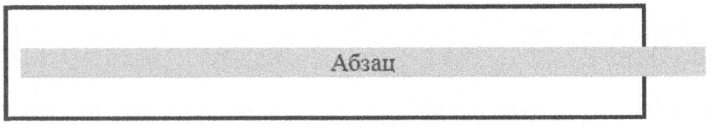


Рис. 7.16. Ширина дочернего элемента больше, чем у его родителя

Вам не показалось! Ширина вложенного элемента действительно больше, чем у родительского элемента, что полностью подтверждается математическими расчетами:

```
10px + 0 + 0 + 540px + 0 + 0 − 50px = 500px
```

Величина `540px` представляет вычисляемое значение для свойства `width: auto` вложенного элемента. Оно подбирается так, чтобы сделать равенство справедливым. Смещение дочернего элемента за пределы родительского ни в коем случае не противоречит спецификации CSS до тех пор, пока сумма значений всех семи свойств вложенного элемента равняется значению свойства `width` его родителя. Этой уловкой можно и нужно пользоваться для получения необычного форматирования.

Усложним задачу, снабдив наши элементы границами.

```
div {width: 500px; border: 3px solid black;}
p.wide {margin-left: 10px; width: auto; margin-right: -50px;
        border: 3px solid gray;}
```

Уравнение, описывающее поведение элементов, принимает такой вид:

```
10px + 3px + 0 + 534px + 0 + 3px − 50px = 500px
```

При добавлении полей дочерний элемент будет выступать еще больше.

В отдельных случаях передача свойству `margin-right` значения `auto` приводит к тому, что правый отступ представляется отрицательным значением. Как и прежде, оно необходимо, чтобы уравновесить ширину области содержимого родительского элемента с суммарной шириной всех отступов, границ и полей вложенного элемента. Ниже приведен пример, подтверждающий действительность подобной ситуации.

```
div {width: 500px; border: 3px solid black;}
p.wide {margin-left: 10px; width: 600px; margin-right: auto;
        border: 3px solid gray;}
```

Уравнение, применяемое для вычисления ширины правого отступа дочернего элемента, принимает следующий вид:

```
10px + 3px + 0 + 600px + 0 + 3px – 116px = 500px
```

Вычисляемое значение свойства `margin-right` равно `-116px`. Оно будет таким даже при указании иного размера в явном виде, поскольку именно оно корректируется при передаче избыточных значений свойствам горизонтального форматирования. (Исключения составляют элементы с направлением расположения элементов справа налево — в них последним подбирается левый отступ.)

Далее приведен пример задания отрицательного отступа для левого края блочного элемента. Результат выполнения кода, обеспечивающего такой тип горизонтального форматирования, показан на рис. 7.17.

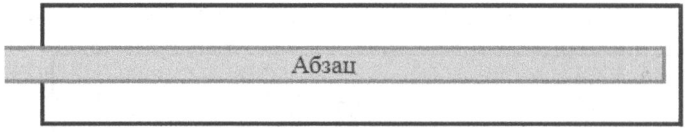


Рис. 7.17. Результат установки отрицательного левого отступа

При задании отрицательного левого отступа абзац не только выступает за край элемента `div`, но и выходит за пределы окна просмотра (браузера)!

Обратите внимание на то, что отрицательной может быть только ширина отступов, но не полей, границ и области содержимого элемента.

Процентные значения

Устанавливая ширину отступов, границ и полей элемента с помощью процентных значений, нужно в полной мере придерживаться правил, описанных в предыдущем разделе. С точки зрения пользовательского агента при вычислении значений свойств горизонтального форматирования не имеет значения, представлены они абсолютными или относительными величинами.

При всем этом поддержка свойствами процентных значений имеет широкое практическое применение. Предположим, ширина области содержимого элемента должна составлять две третьих ширины содержащего блока, а левое и правое поле и левый отступ — всего 5% от этого значения. Стилевое правило, определяющее такое форматирование, имеет следующий вид.

```
<p style="width: 67%; padding-right: 5%; padding-left: 5%;
margin-right: auto; margin-left: 5%;">Процентные значения</p>
```

Точный расчет показывает, что правый отступ будет равен 18% от ширины содержащего блока (100% – 67% – 5% – 5% – 5%).

Не рекомендуется применять в одном правиле процентные значения и значения, выраженные в единицах измерения длины.

```
<p style="width: 67%; padding-right: 2em; padding-left: 2em;
margin-right: auto; margin-left: 5em;">Смешанные величины</p>
```

В подобном случае размер контейнера элемента рассчитывается так:

`5em + 0 + 2em + 67% + 2em + 0 + auto` = ширина содержащего блока

Таким образом, при нулевом отступе содержащий блок будет иметь ширину `27.272727em` (а ширина области содержимого элемента составит `18.272727em`). Положительный правый отступ будет наблюдаться у более широких содержащих блоков, а отрицательные правый отступ — у более узких.

Намного сложнее вычислить размеры элемента при передаче свойствам горизонтального форматирования не только процентных значений, но и значений, выраженных в разных единицах измерения длины.

```
<p style="width: 67%; padding-right: 15px; padding-left: 10px;
margin-right: auto; margin-left: 5em;">Полный набор значений</p>
```

Чтобы еще больше запутать нерадивых авторов, спецификация CSS не предполагает представления свойств, устанавливающих ширину границы, процентными значениями. Именно поэтому при создании полностью гибких контейнеров элементов лучше отказаться от заключения их в рамки (добавлять границы) или определения их ширины процентными значениями.

Замещаемые элементы

До этого момента мы рассматривали горизонтальное форматирование только незамещаемых элементов блочного типа, заполняющих документ в общепринятой последовательности. Замещаемыми блочными элементами управлять немного проще — к ним применяются такие же свойства и их значения, как и к незамещаемым блочным элементам, за одним лишь исключением. Если свойство `width` замещаемого элемента имеет значение `auto`, то его ширина будет равняться исходной ширине его содержимого. В следующем примере в документ вставляется рисунок шириной `20px`, поскольку именно ею характеризуется исходное графическое изображение:

```
<img src="smile.svg" style="display: block; width: auto; margin: 0;">
```

Если бы исходное изображение имело ширину 100 пикселей, то в документе оно занимало бы область точно такой же ширины (без учета полей, границ и отступов).

Для указания другой ширины области содержимого необходимо отказаться от автоматической установки ширины замещаемого элемента в пользу точного числового значения. Предположим, что описанное выше изображение нужно добавить в документ трижды — каждый раз в новом размере. Эта задача имеет такое стилевое решение.

```
<img src="smile.svg" style="display: block; width: 25px; margin: 0;">
<img src="smile.svg" style="display: block; width: 50px; margin: 0;">
<img src="smile.svg" style="display: block; width: 100px; margin: 0;">
```

Результат применения правил показан на рис. 7.18.

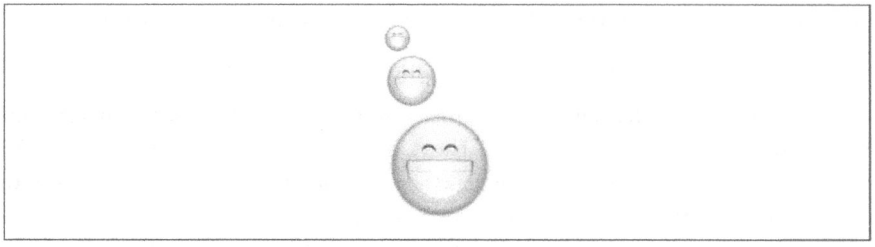


Рис. 7.18. Изменение ширины замещаемого элемента

Учтите, что вместе с шириной изменяется и высота замещаемого элемента. Оба значения пропорциональны друг другу до тех пор, пока свойство `height` не будет изменено независимо от свойства `width`. Операция полностью обратима — изменение свойства `height` при автоматической настройке свойства `width` выполняется с сохранением пропорций содержимого замещаемого элемента.

Раз уж речь зашла о высоте элементов, поговорим о вертикальном форматировании блочных элементов с обычным порядком заполнения содержимым.

Вертикальное форматирование

Как и горизонтальное, вертикальное форматирование элементов блочного типа выполняется с помощью отдельного набора стилевых свойств, обладающих специфическим поведением. Высота элемента по умолчанию определяется его содержимым. В свою очередь, она также зависит от ширины элемента, т.е. чем *уже* абзац, тем выше он должен быть, чтобы включить весь текст.

Как бы там ни было, в CSS высоту блочного элемента можно устанавливать вручную. Результат такой операции не всегда очевиден и зависит от нескольких факторов. Представим, что значение свойства `height`, устанавливаемое следующим правилом, больше, чем высота содержимого элемента:

```
<p style="height: 10em;">
```

Незанятое содержимым пространство элемента будет восприниматься как часть одного из его полей. Но возможны ситуации, в которых значение свойства `height` меньше высоты содержимого элемента, как в следующем коде:

```
<p style="height: 3.33em;">
```

В подобных ситуациях пользовательский агент должен обеспечить возможность просмотра всего содержимого элемента, не прибегая к увеличению его действительной высоты. При решении этой задачи он в первую очередь ориентируется на значение свойства `overflow`. Возможные варианты представления элемента, содержимое которого больше, чем он сам, показаны на рис. 7.19.

Спецификация CSS1 разрешала пользовательскому агенту игнорировать значение свойства `height`, отличное от `auto`, в случаях применения к незамещаемым элементам (таким, как изображения). В CSS2 и более поздних версиях значение свойства `height` не должно игнорироваться ни при каких обстоятельствах. Исключение делается только в одном случае, который будет подробно рассмотрен несколько позже.

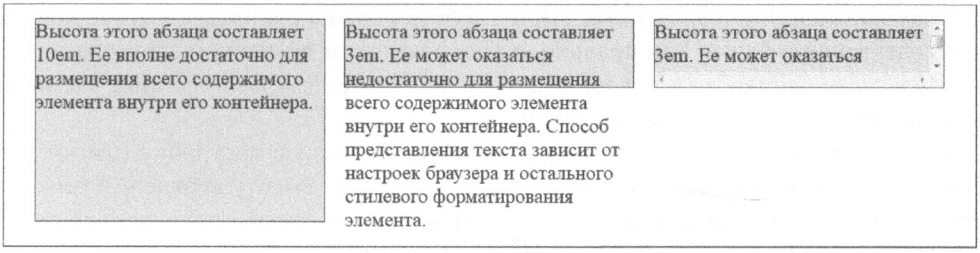


Рис. 7.19. Представление элемента, высота которого меньше высоты его содержимого

Свойство `height`, подобно `width`, по умолчанию определяет высоту области содержимого элемента, а не видимой области его контейнера, как можно было предположить. Поля, границы и отступы в его значении не учитываются до тех пор, пока свойство `box-sizing` имеет значение по умолчанию: `content-box`.

Свойства вертикального форматирования

Как и горизонтальное, вертикальное форматирование устанавливается с помощью набора из семи свойств, каждое из которых задает высоту одной из составляющих контейнера элемента: `margin-top`, `border-top`, `padding-top`, `height`, `padding-bottom`, `border-bottom` и `margin-bottom`. С их назначением проще всего ознакомиться по схеме, показанной на рис. 7.20.

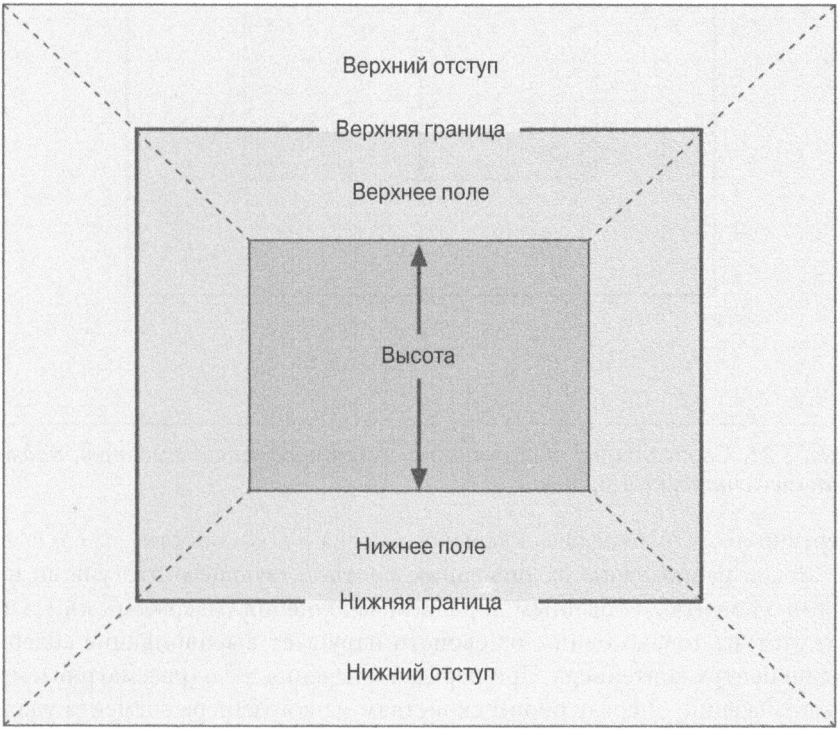


Рис. 7.20. Семь параметров вертикального форматирования блочного элемента

Сумма значений обозначенных выше свойств должна равняться высоте контейнера содержащего блока. Как правило, она определяется значением свойства `height` родительского элемента (в подавляющем большинстве случаев блочные элементы вкладываются в элементы блочного типа).

Только три свойства из семи допускают автоматическую настройку (имеют значение `auto`): свойство `height` и свойства, определяющие высоту верхнего и нижнего отступов. Отступ, не представленный точным значением соответствующего свойства, имеет нулевую высоту (в отсутствие объявления свойства `border-style`). Толщина границ, объявленных с помощью свойства `border-style`, по умолчанию представляется значением `medium`. Размеры, устанавливаемые свойствами вертикального форматирования элемента, которые могут выражаться значением `auto`, показаны на рис. 7.21.

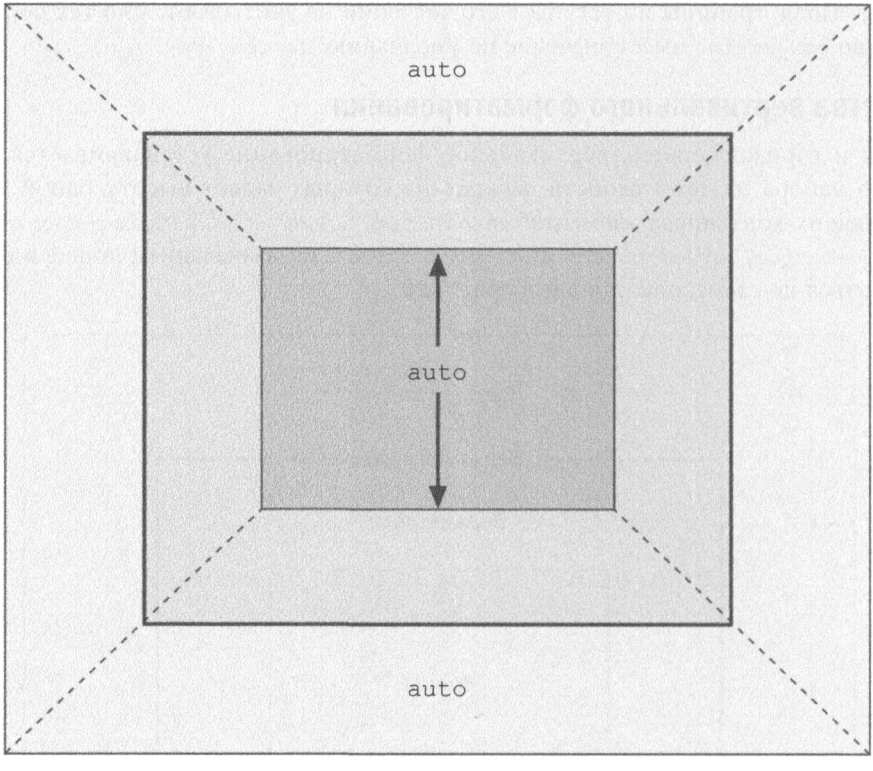


Рис. 7.21. Свойства вертикального форматирования элементов, подлежащие установке в значение `auto`

Интересно то, что передача ключевого слова `auto` свойству `margin-top` или `margin-bottom` равнозначна схлопыванию соответствующего отступа до нулевого значения (в элементах с обычным порядком заполнения содержимым). Назначение нулевого отступа только одним из свойств нарушает выравнивание содержимого элемента по центру контейнера. При передаче значения `auto` (рассматриваемого браузером как значение `0`) сразу обоим свойствам из контейнера элемента удаляется и верхний, и нижний отступ.

Автоматическая настройка верхнего и нижнего отступов у позиционируемых элементов и flex-элементов выполняется совершенно иным способом.

Свойству `heigth` допускается передавать только положительные значения поддерживаемых типов или значение `auto`. Оно также не может иметь значение `0`.

Процентные значения высоты

Ознакомившись со способами обработки свойства `height`, рассмотрим, к какому результату приводит передача ему процентных значений. В элементах с обычным порядком заполнения содержимым процентное значение свойства `height` указывается относительно высоты содержащего блока. Следующая разметка добавляет в документ абзац высотой `3em`.

```
<div style="height: 6em;">
   <p style="height: 50%;">Абзац с половинной высотой</p>
</div>
```

Как вы знаете, передача значения `auto` свойствам, устанавливающим вертикальные отступы, приводит к их обнулению, поэтому единственный верный способ выровнять элемент по центру (по вертикали) содержащего блока состоит в установке их в значение 25%. При этом важно понимать, что центрируется не содержимое элемента, а его контейнер.

Тем не менее, если высота содержащего блока не указана в явном виде, то высота элемента устанавливается автоматически (свойство `height` получает значение `auto`). Изменив предыдущий пример так, чтобы свойство `height` элемента `div` получало значение `auto`, можно расширить область содержимого элемента `p` на всю высоту элемента `div`.

```
<div style="height: auto;">
   <p style="height: 50%;">Абзац с полной высотой (height: auto)</p>
</div>
```

Результат выполнения обоих примеров показан на рис. 7.22. (свободное пространство между границами абзацев и элементов `div` образовано верхними и нижними отступами элементов `p`).

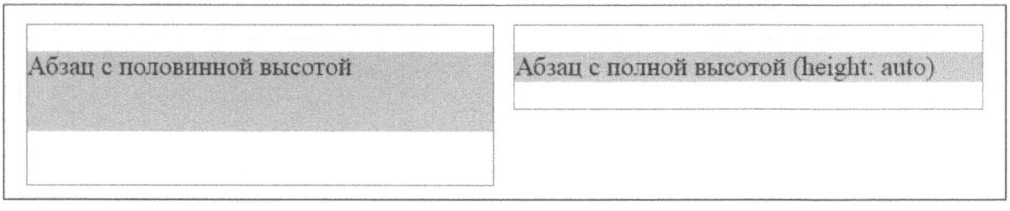


Рис. 7.22. Результат передачи процентного значения свойству `height` зависит от значений остальных свойств вертикального форматирования

На рис. 7.22, *слева*, показан абзац, высота которого составляет половину высоты содержащего блока. Легко заметить, что он смещен вверх относительно центра содержащего блока. Чтобы понять, почему это происходит, необходимо проанализировать, чем обусловлена высота свободного пространства над и под абзацем. Высота содержащего блока равна значению свойства `height` элемента `div` и составляет `6em`. Исходя из этого, высота абзаца (области содержимого его контейнера) будет равняться `3em`. Поскольку верхний и нижний отступы абзаца установлены в значение `1em`, высота его контейнера составит `5em`. Таким образом, свободное пространство между нижней границей элемента `div` и нижним краем области содержимого элемента `p` будет `2em`, а не `1em`, как можно было подумать сначала. Как видите, конечный результат очевиден далеко не всегда. Он требует проведения точных расчетов и совершенно не терпит принятия интуитивных решений.

Автоматическая настройка высоты элемента

В простейшем случае высота блочного элемента с обычным направлением заполнения содержимым, к которому применяется объявление `height: auto`, определяется размером контейнеров включенных в него строчных элементов (включая анонимный текст). Если он содержит только элементы блочного уровня, то его высота автоматически рассчитывается как расстояние от верхней границы самого верхнего вложенного элемента до нижней границы самого нижнего дочернего элемента. При этом отступы дочерних элементов будут выходить за пределы содержащего их элемента. (О том, почему это происходит, речь пойдет в следующем разделе.)

Если же контейнер блочного элемента включает верхнее или нижнее поле либо верхнюю или нижнюю границу, то его высота будет автоматически рассчитываться как расстояние от верхнего края поля самого верхнего вложенного элемента до нижнего края поля самого нижнего дочернего элемента. Ниже приведен наглядный пример такого форматирования.

```
<div style="height: auto; background: silver;">
   <p style="margin-top: 2em; margin-bottom: 2em;">Это абзац!</p>
</div>
<div style="height: auto; border-top: 1px solid; border-bottom:
     px solid; background: silver;">
   <p style="margin-top: 2em; margin-bottom: 2em;">Еще один абзац!</p>
</div>
```

Результаты форматирования содержимого элемента блочного типа каждым из способов представлены на рис. 7.23.

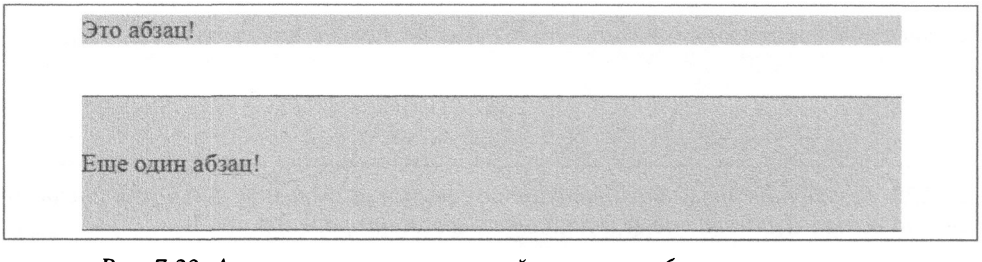


Рис. 7.23. Автоматическая настройка высоты блочного элемента

Если в последнем примере у элемента убрать поля и добавить границы, то это не окажет никакого влияния на конечный результат — его высота по-прежнему будет рассчитываться от верхнего до нижнего поля абзаца.

Схлопывание вертикальных отступов

В процессе вертикального форматирования элементов необходимо быть готовым к совершенно неожиданному эффекту — *схлопыванию* отступов соседних элементов. Схлопыванию подлежат только отступы элементов, но не границы и поля.

Объяснить, в чем заключается этот эффект, проще всего на примере элементов списка. Предположим, их форматирование задается с помощью такого правила:

```
li {margin-top: 10px; margin-bottom: 15px;}
```

Каждый элемент списка снабжен верхним отступом высотой 10 пикселей и нижним отступом высотой 15 пикселей. Тем не менее при визуализации списка расстояние между его соседними элементами будет равняться не 25 (сумма верхнего и нижнего отступов), а всего лишь 15 пикселям. Очевидно, что в данном случае в расчет не принимается меньший из отступов, а отображается только большее из двух значений. Результат представления соседних элементов с учетом эффекта схлопывания отступов и без него продемонстрирован на рис. 7.24.

Правильно настроенный пользовательский агент всегда схлопывает отступы так, как показано на рис. 7.24, *вверху*, отображая между элементами списка свободное пространство шириной 15 пикселей. Иными словами, схлопыванию подлежит только меньший из отступов, чего не наблюдается на нижнем примере, характеризующемся 25-пиксельным свободным пространством между элементами.

Рис. 7.24. Схлопнутые и несхлопнутые отступы

Термин "схлопывание отступов" не строгий: вместо него можно использовать определение "перекрывание отступов", хотя оно не отражает сути процесса. Чтобы лучше его понять, воспользуемся следующей аналогией. Представьте себе каждый блочный элемент в виде небольшого листа бумаги, на котором написан содержащийся в нем текст. Каждый лист бумаги закрепляется на большей по размеру пластиковой основе. Области пластиковой основы, выступающие за края бумажного листа, образуют отступы элемента. Сначала на холст (документ) выкладывается первый элемент (пусть это будет заголовок первого уровня) и располагается в самой верхней его части. После него на холст добавляется второй элемент (абзац с текстом), но несколько ниже, так, чтобы его пластиковая основа заходила под пластиковую основу первого элемента. Второй элемент смещается вверх до тех пор, пока край одного из бумажных листов не соприкоснется с краем пластиковой основы другого листа. Если пластиковая основа первого элемента выступает за край бумажного листа на меньшее расстояние, чем пластиковая основа второго элемента за края своего листа, то смещение прекратится, как только край пластиковой основы второго элемента соприкоснется с краем бумажного листа первого элемента. Элементы перекрываются на ширину области выступания пластиковой основы первого элемента.

Схлопыванию подвержены отступы сразу нескольких элементов, например совмещаемых в конце списка. Чтобы продемонстрировать, в чем оно заключается, продолжим предыдущий пример, применив к документу, содержащему список, следующие правила.

```
ul {margin-bottom: 15px;}
li {margin-top: 10px; margin-bottom: 20px;}
h1 {margin-top: 28px;}
```

Последний элемент списка наделяется нижним отступом `20px`, а нижний отступ элемента `ul` составляет `15px`. При этом верхний отступ элемента `h1`, совмещаемый с указанными двумя нижними отступами, равен 28 пикселям. В результате схлопывания отступов элементы `li` и `h1` будут расположены на расстоянии 28 пикселей друг от друга, как показано на рис. 7.25.

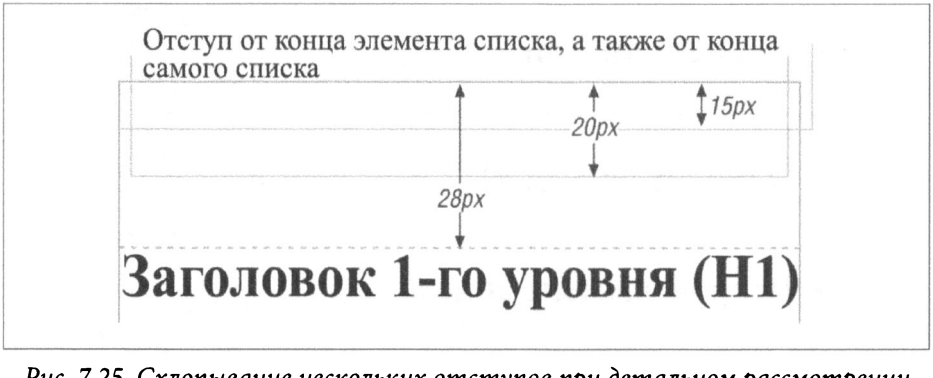


Рис. 7.25. Схлопывание нескольких отступов при детальном рассмотрении

Теперь вспомним еще более ранний пример, рассмотренный в предыдущем разделе, в котором добавление полей или границ приводило к включению отступов

дочернего элемента в содержащий блок. Адаптировав его к нашей задаче, добавим к элементу `ul` небольшую рамку (тонкие границы).

```
ul {margin-bottom: 15px; border: 1px solid;}
li {margin-top: 10px; margin-bottom: 20px;}
h1 {margin-top: 28px;}
```

В результате такого форматирования нижний отступ элемента `li` помещается в область содержимого родительского элемента (`ul`). Вследствие этого схлопывание отступов наблюдается только для элементов `ul` и `h1`, как показано на рис. 7.26.

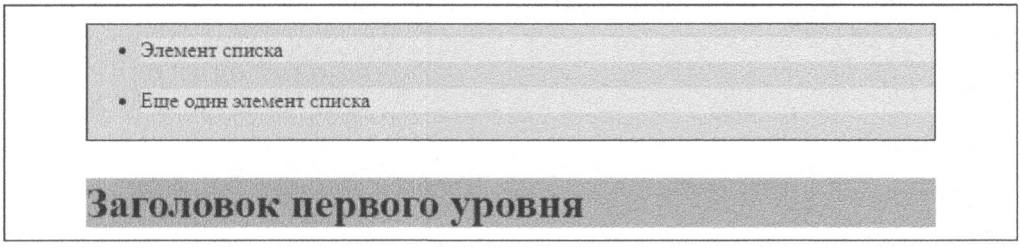


Рис. 7.26. Добавление к элементу границ приводит к заключению отступов его дочернего элемента в область содержимого

Схлопывание отрицательных отступов

Вертикальное форматирование элементов в значительной степени зависит от схлопывания заданных в них отрицательных отступов. При вычислении расстояния между элементами пользовательским агентом учитывается абсолютная разница между отступами без учета знаков каждого из значений. Иными словами, из положительного отступа одного элемента нужно вычесть абсолютное значение отрицательного отступа другого элемента. Примеры взаимного расположения элементов с отступами разных типов приведены на рис. 7.27.

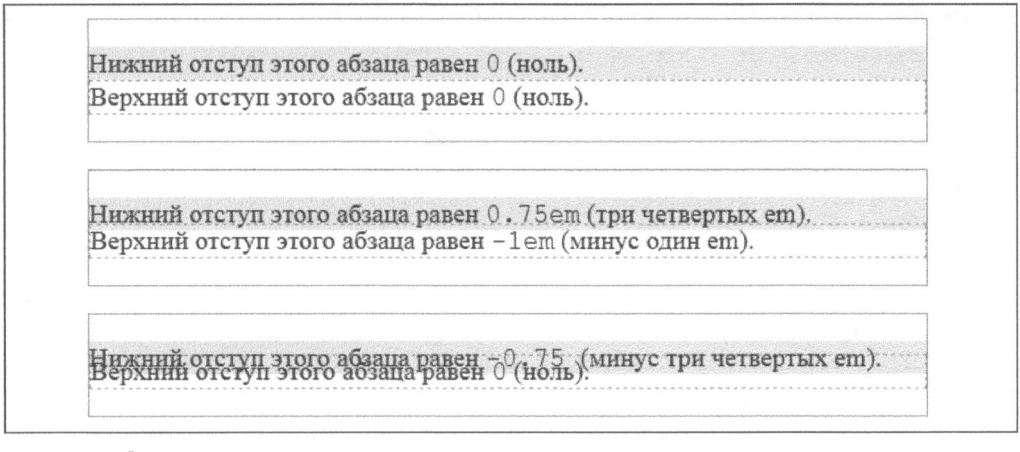


Рис. 7.27. Вертикальное позиционирование элементов с отрицательными и положительными отступами

Обратите внимание на эффект втягивания элемента с отрицательным отступом в область содержимого элемента с положительным отступом. Он имеет такое же происхождение, как и эффект горизонтального выступа элемента за края содержащего блока при задании отрицательных отступов справа или слева. Рассмотрим следующий пример.

```
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">
      Абзац
   </p>

   Элемент div

</div>
```

Как показано на рис. 7.28, отрицательный верхний отступ определяет смещение абзаца вверх, сопровождаемое подтягиванием вверх на 50 пикселей содержимого расположенного под ним элемента `div`. В действительности вверх будет подтягиваться содержимое всех элементов, расположенных под абзацем с отрицательным отступом (в обычном потоке элементов).

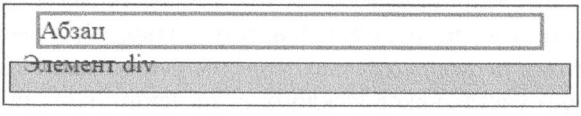


Рис. 7.28. Влияние верхнего отрицательного отступа на расположение нижних элементов

Сравним только что полученный результат с эффектом, вызванным включением в верхний элемент отрицательного нижнего отступа (рис. 7.29).

```
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">
      Абзац
   </p>
</div>
<p>
   Следующий абзац
</p>
```

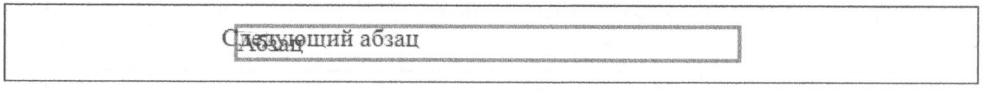


Рис. 7.29. Влияние нижнего отрицательного отступа на взаимное расположение элементов

Форматирование, показанное на рис. 7.29, обусловлено подтягиванием вверх следующего элемента (p) на расстояние, равное величине отрицательного нижнего отступа, установленного в предыдущем элементе (div). Легко заметить, что нижний край области содержимого элемента div приподнят над нижним краем дочернего абзаца. Следующий после них элемент смещается вверх на такое же расстояние, что и область содержимого элемента div.

В качестве еще одного, более сложного примера попробуем определить, как будет выполняться схлопывание отступов элементов списка и абзацев, если они представлены отрицательными значениями. В следующем примере отрицательные отступы заданы элементам заголовка и неупорядоченного списка.

```
li {margin-bottom: 20px;}
ul {margin-bottom: -15px;}
h1 {margin-top: -18px;}
```

В данном случае из положительного отступа заголовка (20px) нужно вычесть отрицательный отступ с большим абсолютным значением (-18px). Получается, что между нижним краем области содержимого элемента списка и верхним краем области содержимого элемента h1 будет наблюдаться отступ всего в 2 пикселя (рис. 7.30).

При перекрывании элементов, вызванном установкой отрицательных отступов, сложно определить порядок их расположения друг после друга. Ситуацию можно прояснить, если задать каждому из накладываемых друг на друга элементов отдельный фон. В подобном случае фон самого верхнего элемента будет перекрывать содержимое остальных элементов, расположенных непосредственно под ним. Это вполне ожидаемый результат, так как браузеры обычно представляют элементы документа в порядке от самого нижнего к самому верхнему. Таким образом, последний добавленный в документ элемент будет закрывать содержимое предыдущего элемента, а тот — содержимое элемента, расположенного под ним, и т.д. Разумеется, речь идет не обо всем содержимом элементов, а только о перекрывающихся областях.

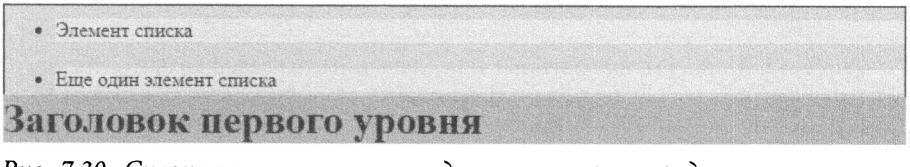


Рис. 7.30. Схлопывание отступов, один из которых представлен отрицательным значением

Элементы списка

Для форматирования элементов списка применяется несколько специальных стилевых правил. Обычно перед каждым элементом списка ставится маркер — специальный символ или цифра, не относящиеся к его области содержимого (рис. 7.31).

Спецификация CSS1 была лишена инструментов изменения внешнего вида и позиционирования маркеров элементов списков. Во второй версии CSS это упущение было исправлено — спецификация пополнилась специальными стилевыми свойствами, подобными marker-offset. Тем не менее отсутствие поддержки браузерами

и изменение общей концепции обработки списков привели к исключению их из CSS2.1, что вывело проблему на совершенно новый уровень: следующую спецификацию требовалось снабдить полностью новыми инструментами форматирования маркеров, основанными на других синтаксических правилах. В соответствии с новой концепцией, позиционирование маркеров списков целиком возлагается на пользовательский агент (по крайней мере, так обстояло дело на момент выхода книги).

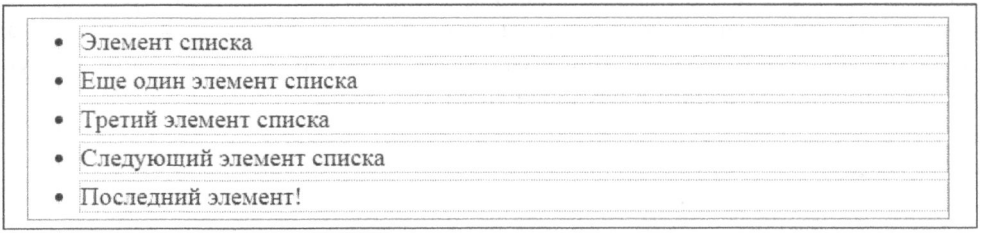


Рис. 7.31. Структура элементов списка

Предполагается, что начальный маркер элемента списка либо включается в него в качестве строчного элемента, либо располагается вне области содержимого, что определяется значением свойства `list-style-position`. При включении маркера в элемент списка он обрабатывается браузером как блочный элемент (рис. 7.32).

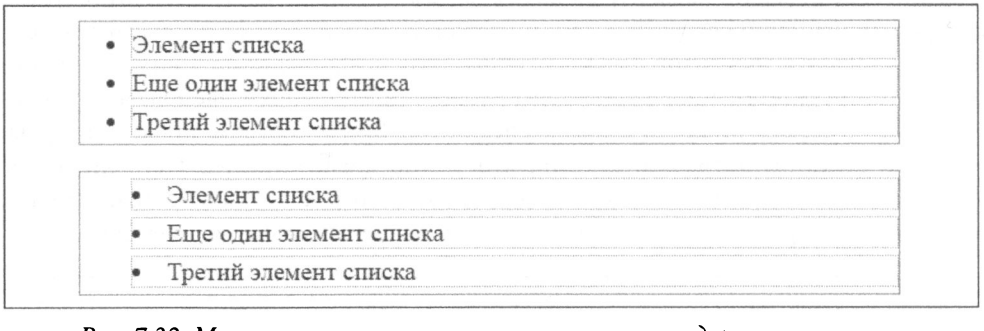


Рис. 7.32. Маркеры, включенные и исключенные из содержимого списка

Маркер, исключенный из состава элемента, располагается на определенном удалении от левого края области содержимого (при обычном порядке заполнения). Это расстояние остается неизменным независимо от стилевых правил, применяемых к самому элементу списка. При определенных условиях маркеры могут исключаться из состава элемента автоматически.

Не забывайте, что контейнеры элементов списка, подобно контейнерам блочных элементов, представляют содержащие блоки своих родительских элементов.

Строчные элементы

Строчные, или вложенные, элементы получили такое же широкое распространение, как и элементы блочного уровня. Грамотное стилевое форматирование строчных элементов позволяет добиться поразительных результатов, зачастую недостижимых

при использовании всех описанных выше методик. К строчным относятся такие распространенные элементы, как `em`, `a` (оба незамещаемые) и `img` (замещаемый).

Обратите внимание на то, что описанные в этом разделе методики *не* применимы к элементам таблицы. В CSS2 впервые были представлены стилевые свойства форматирования таблиц, отличающиеся от уже ставших привычными свойств визуального оформления элементов блочного и строчного уровней, как по поведению, так и по принципу действия. Форматирование таблиц и их содержимого — очень сложная задача, которая в этой книге подробно рассматриваться не будет, поскольку достойна отдельного издания.

В контексте встраивания замещаемые и незамещаемые элементы характеризуются разным набором возможностей. Но перед тем как перейти к их рассмотрению, нужно ознакомиться со структурой строчных элементов.

Позиционирование строчных элементов

Для начала нужно определиться с принципами позиционирования строчных элементов. Они в корне отличаются от принципов форматирования элементов блочного типа, ограниченных контейнером, в котором не может размещаться содержимое соседних элементов. С другой стороны, любой блочный элемент (например, абзац) имеет содержимое, представляемое с помощью отдельных стилевых свойств, отличных от применяемых к нему самому. Далее мы разберемся, как текст абзаца перераспределяется между его строками и какие факторы определяют его внешний вид.

Чтобы понять, как образуется многострочный текст, сначала опишем поведение элемента, содержащего всего одну длинную строку (рис. 7.33). Рамка вокруг текстовой строки образована границами элемента `span`, в который она вложена.

```
span {border: 1px dashed black;}
```

Рис. 7.33. Строчный элемент, состоящий всего из одной текстовой строки

На рис. 7.33 показан простейший строчный элемент, заключенный в контейнер элемента блочного уровня. Структурно он мало чем отличается от абзацев с меньшим или большим количеством слов, хотя обычно текст абзаца переносится на несколько строк и не представляется единственным элементом `span`.

Для представления однострочного абзаца в более удобочитаемом виде нужно сначала определить ширину элемента, а затем разбить исходную строку на несколько строк так, чтобы все они помещались в области содержимого элемента, как показано на рис. 7.34.

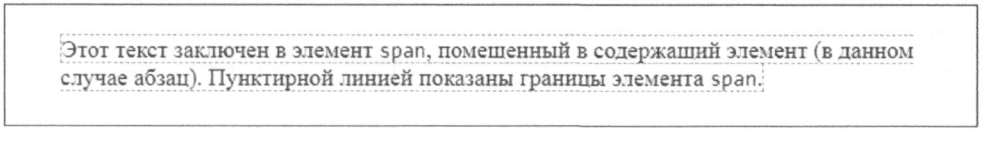


Рис. 7.34. Многострочный вложенный элемент

Содержимое абзаца не изменилось — текстовая строка всего лишь разделилась на несколько частей, расположенных одна над другой.

Как показано на рис. 7.34, верхние и нижние границы соседних строк совмещены. Но такая ситуация возможна только в отсутствие полей в строчных элементах. Также обратите внимание на то, что границы строк незначительно перекрываются. В частности, нижняя граница первой строки находится сразу под верхней границей второй строки. Такое представление является следствием отображения верхней и нижней границ не в точно заданном месте, а в следующем *внешнем* по отношению к содержимому строчного элемента ряду пикселей (при просмотре документа на экране монитора).

Положение текстовых строк проще всего отслеживать, снабдив элемент `span` цветным фоном. На рис. 7.35 показаны четыре абзаца, каждый из которых характеризуется собственным способом выравнивания относительно границ элемента, устанавливаемым свойством `text-align`. Благодаря фону намного проще различать их визуально.

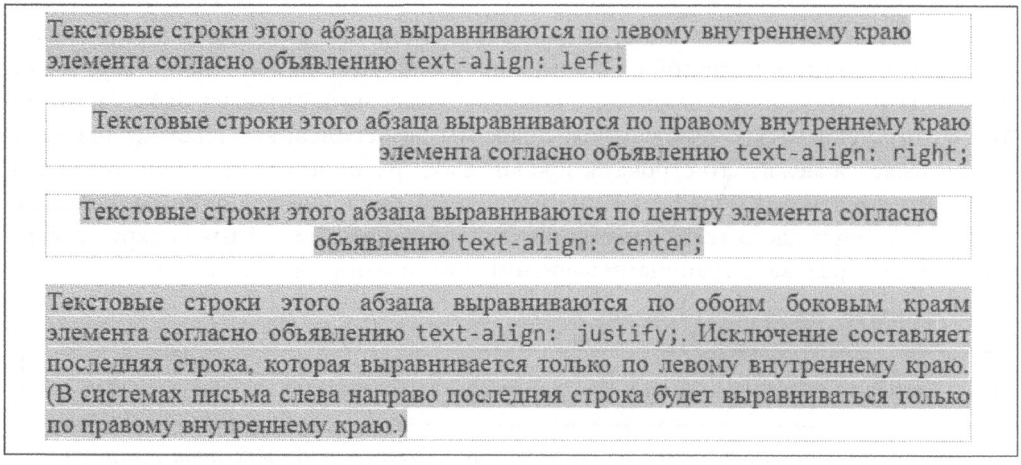


Рис. 7.35. Разные способы выравнивания текстовых строк абзаца

Как видно, далеко не каждая строка абзаца достигает края области содержимого родительского элемента, обозначенного серой пунктирной рамкой. При выравнивании по левому краю текстовые строки в обязательном порядке соприкасаются только с левым краем области содержимого. Такие строки заканчиваются в местах, обозначенных символами разрыва строки. Абзацу с выравниванием по правому краю свойственно полностью обратное форматирование: все строки заканчиваются у правого края области содержимого. Начало таких строк определяется их длиной, отсчитанной от места выравнивания. Третий абзац выровнен по центру; середина каждой его строки совмещена с серединой абзаца.

Форматирование последнего абзаца определяется значением `justify` свойства `text-align`. Легко заметить, что в подобном способе выравнивания с краями области содержимого соприкасаются оба края каждой текстовой строки. Чтобы

выровнять абзац по ширине, необходимо увеличить интервалы между словами и символами текстовых строк. При этом значение свойства `word-spacing` в расчет не принимается (значение свойства `letter-spacing` не изменяется, если выражено в единицах измерения длины).

Выше описан только простейший случай размещения многострочного текста в блочном элементе. Как можно будет удостовериться далее, модель строчного форматирования намного сложнее, чем кажется на первый взгляд.

Основные определения

Перед тем как перейти к описанию модели строчного форматирования, нужно определиться с терминологией и основными понятиями.

Анонимный текст

Строка символов, расположенных вне строчного элемента. Например, в разметке `<p> I'm <em>so</em> happy!</p>` анонимный тест представлен словами `"I'm "` и `" happy!"`. Обратите внимание на то, что пробелы являются частью анонимного текста, как и любой другой символ, и пренебрегать ими нельзя.

Кегельная площадка

Ее размер зависит от используемого шрифта и сопоставляется с высотой буквы *m*. В отдельно взятом шрифте могут присутствовать символы, ширина или высота которых больше размера кегельной площадки. В CSS высота кегельной площадки представляется стилевым свойством `font-size`.

Область содержимого

У незамещаемых элементов область содержимого определяется двояко. Согласно спецификации CSS, окончательный выбор остается за пользовательским агентом. В первом варианте область содержимого образуется кегельными площадками всех символов элемента, объединенных между собой. Во втором варианте она представляется областью, включающей знаки всех символов строчного элемента. Для простоты восприятия последующего материала будем считать, что пользовательские агенты всегда придерживаются первого варианта определения области содержимого. Высота области содержимого замещаемого элемента определяется его собственным размером, а также размерами включенных в него отступов, границ и полей.

Интерлиньяж

Величина интерлиньяжа, или междустрочного интервала, вычисляется как разница между значениями свойств `font-size` и `line-height`. Полученное значение делится на два и определяет высоту пустого пространства, добавляемого по обе стороны области содержимого. Интерлиньяж назначается только незамещаемым элементам.

Контейнер строчного элемента

Включает область содержимого строчного элемента и свободное пространство, образованное междустрочным интервалом. Высота контейнера незамещаемого элемента в точности равна значению свойства `line-height`. Под высотой контейнера замещаемых элементов подразумевается высота области содержимого, поскольку интерлиньяж для них не указывается.

Контейнер строки

Это контейнер, высота которого определяется расстоянием от наиболее высоко расположенной точки самого высокого вложенного элемента строки до наиболее низко расположенной точки самого низкого вложенного элемента строки. Иными словами, верхний край контейнера строки совпадает с верхней точкой самого высоко расположенного строчного элемента, а его нижний край совмещается с нижней точкой наиболее низко расположенного строчного элемента.

Кроме того, при оценке поведения и характеристик строчных элементов необходимо учитывать следующие факторы.

- Область содержимого в строчных элементах играет такую же роль, как и контейнер содержимого в элементах блочного типа.
- Фон строчного элемента распространяется на область содержимого и пространство, обозначенное полями.
- Границы строчного элемента образуют рамку, в которую заключаются область содержимого и поля.
- Незамещаемые элементы и их контейнеры не подлежат вертикальному форматированию с помощью отступов, границ и полей. Из этого следует, что они не влияют на высоту контейнера строчного элемента (а потому и содержащего его контейнера строки).
- Отступы и границы замещаемых элементов влияют на высоту контейнера строчных элементов, а потому и на высоту содержащего его контейнера строки.

Примечательно то, что к строчному элементу можно применить свойство `vertical-align`, в данном случае определяющее способ вертикального выравнивания его контейнера относительно краев контейнера строки.

Чтобы лучше понимать, каким образом строчные элементы получают форматирование, нужно научиться безошибочно определять размер контейнера строки. Перед тем как приступить к дальнейшему материалу, внимательно изучите следующие инструкции, в которых процесс образования контейнера строки описан максимально подробно.

Начинать нужно с определения высоты контейнера каждого элемента, включенного в строку.

1. Определите значения свойств `font-size` и `line-height` каждого строчного незамещаемого элемента, а также анонимного текста, включенного в строку,

и сопоставьте их. Вычитая значение свойства `font-size` из значения свойства `line-height`, можно рассчитать интерлиньяж. Разделив полученную величину пополам, вы получите высоту свободного пространства, добавляемого к верхнему и нижнему краям кегельной площадки.

2. Определите значения свойств `height`, `margin-top`, `margin-bottom`, `padding-top`, `padding-bottom`, `border-top-width` и `border-bottom-width` каждого замещаемого элемента и вычислите их сумму.
3. Вычислите расстояние от базовой линии текстовой строки до верхнего и нижнего краев каждого из включенных в нее элементов. Сделать это не так просто, как кажется на первый взгляд: вам потребуется совместить базовые линии всех строчных элементов и анонимного текста, включенных в строку. Не забывайте, что замещаемые элементы привязываются к базовой линии нижним краем своего контейнера.
4. Определите вертикальное смещение каждого элемента, устанавливаемое свойством `vertical-align`. Его значение указывает расстояние, на которое контейнер элемента перемещается вверх или вниз в процессе выравнивания. По нему можно судить о степени смещения его над или под базовой линией текстовой строки.
5. Зная расположение каждого строчного элемента относительно базовой линии всей строки, можно легко вычислить высоту ее контейнера. Чтобы получить ее, достаточно сложить расстояние от базовой линии строки до верхнего края самого высоко расположенного строчного элемента с расстоянием от базовой линии текстовой строки до нижнего края наиболее низко расположенного строчного элемента.

В следующих разделах детально рассмотрен каждый из пунктов приведенных выше инструкций.

Форматирование строчного элемента

Как известно, все строчные элементы подвержены воздействию свойства `line-height`, даже если оно не объявляется в явном виде. Его значение оказывает влияние на способ представления элемента и требует особого внимания с нашей стороны.

Рассмотрим, каким образом свойство `line-height` влияет на высоту текстовой строки. Очевидно, что она (или высота контейнера текстовой строки) зависит от размера всех включенных в нее элементов и анонимного текста. В свою очередь, свойство `line-height` устанавливает высоту только строчных элементов и *не* применяется к элементам блочного типа в явном виде. Если применить его к элементу блочного типа, то оно будет воздействовать только на его строчное содержимое. В качестве примера рассмотрим абзац, представленный следующей разметкой:

```
<p style="line-height: 0.25em;"></p>
```

Лишенный содержимого, абзац не будет визуализироваться в документе. Применение к нему свойства `line-height` со значением `0.25em` не находит отражения в документе и будет проявляться только при наполнении его данными.

При применении свойства `line-height` к блочному элементу оно воздействует на все его содержимое, независимо от того, представлено оно строчными элементами или нет. В более привычном понимании это означает, что каждая текстовая строка блочного элемента рассматривается как отдельный строчный элемент, независимо от того, заключена она в тег или нет. Исходя из вышесказанного, блочный элемент можно представить следующей разметкой.

```
<p>
<line>Содержимое этого абзаца </line>
<line>представлено несколькими текстовыми </line>
<line>строками.</line>
</p>
```

Несмотря на то что элемент `line` в спецификации HTML не объявлен, содержимое абзаца обрабатывается так, как если бы он присутствовал в разметке документа: текстовые строки наследуют стили от абзаца. Таким образом, свойство `line-height` не нужно задавать для каждой из текстовых строк по отдельности — достаточно объявить его для родительского блочного элемента. Оно будет применено ко всем его дочерним элементам, как явным, так и анонимным.

Форматирование фиктивного элемента `line` предопределяется значением свойства `line-height`, заданным его родительскому элементу блочного уровня. Согласно спецификации CSS, свойство `line-height` устанавливает *минимальную* высоту строки для всего содержимого блочного элемента. Следовательно, объявление `p.spacious {line-height: 24pt;}` предполагает, что все строки абзаца указанного класса имеют высоту не менее 24 пикселей. С технической точки зрения свойство может наследоваться только строчными элементами, но большая часть содержимого абзаца, скорее всего, не заключена в элементы строчного типа. При введении в модель форматирования фиктивных элементов `line` она будет оставаться работоспособной без нарушения требований спецификации.

Незамещаемые строчные элементы

По мере приобретения опыта в верстке документов вы научитесь создавать текстовые строки, содержащие одни только незамещаемые элементы (или анонимный текст). Форматируя их отдельно от текстовых строк, включающих только замещаемые элементы, можно в полной мере оценить различие между строчными элементами обоих типов.

Контейнер строчного элемента

Область содержимого незамещаемого строчного элемента или анонимного текста в первую очередь определяется значением свойства `font-size`. При шрифте с размером `15px` высота области содержимого такого элемента будет равна 15 пикселям, как показано на рис. 7.36.

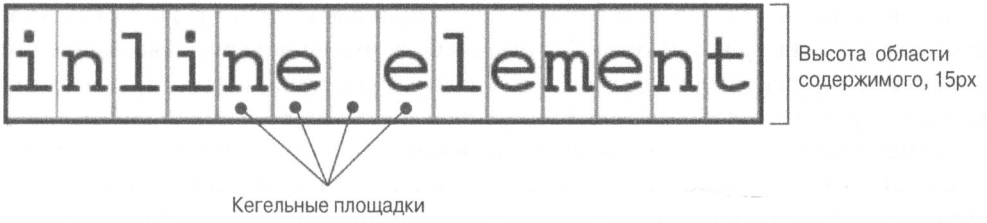


Рис. 7.36. Область содержимого строчного элемента образуется кегельными площадками символов

Следующий параметр, влияющий на размер области содержимого, определяется значением свойства `line-height`. Если к строчному элементу помимо правила `font-size` со значением `15px` применено правило `line-height`, установленное в значение `21px`, то разница между ними (интерлиньяж) составит 6 пикселей. Разделив полученное значение пополам, пользовательский агент добавляет свободное пространство вычисленной таким образов высоты к верхнему и нижнему краям строчного элемента. Область, включающая содержимое элемента и свободное пространство по обоим его краям, суммарная высота которого равна междустрочному интервалу, называется контейнером строчного элемента. Структурные компоненты контейнера строчных элементов показаны на рис. 7.37.

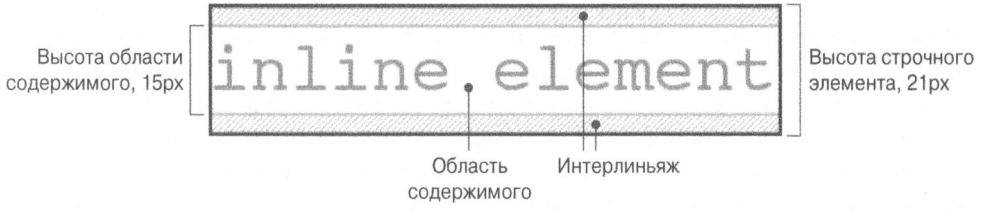


Рис. 7.37. Область содержимого и интерлиньяж

Рассмотрим следующий фрагмент документа HTML.

```
<p style="font-size: 12px; line-height: 12px;">
Этот абзац содержит примеры <em>обычного акцентированного</em> <br>
и <strong style="font-size: 24px;">сильно акцентированного </strong>
текстовых фрагментов, последний из которых представлен шрифтом
большего <br> размера, чем окружающий его текст.
</p>
```

В данном примере основной текст абзаца, определяемый свойством `font-size`, имеет размер `12px`, но он включает фрагмент сильно акцентированного текста, размер шрифта которого равен `24px`. Однако свойство `line-height` *всех* строчных элементов наследуется от абзаца и устанавливается в значение `12px`. Следовательно, высота строки у элемента `strong` равна 12 пикселям.

Проведенные выше вычисления показывают, что высота области содержимого для текстовых фрагментов с одинаковыми значениями свойств `font-size` и `line-height` остается неизменной (поскольку разница между ними равна нулю) и равняется `12px`. При этом у фрагмента сильно акцентированного текста разница

значений свойств `font-size` и `line-height` составляет `-12px`. Получается, что к верхнему и нижнему краям области содержимого нужно добавить свободное пространство с отрицательной высотой `-6px` (половина интерлиньяжа). Высота контейнера строчного элемента определяется в результате сложения положительного (`24px`) и отрицательного (`-12px`) значений и равняется `12px`. Это означает, что контейнер сильно акцентированного текста высотой 12 пикселей выравнивается вертикально посередине большей по размеру области содержимого, характеризующейся высотой 24 пикселя.

Если проанализировать размеры контейнеров всех элементов абзаца, то может сложиться впечатление, что все они имеют одинаковую высоту и потому выровнены между собой. В действительности это не так, поскольку элементы выравниваются по базовой линии текста, а не по средней линии их контейнеров (рис. 7.38).

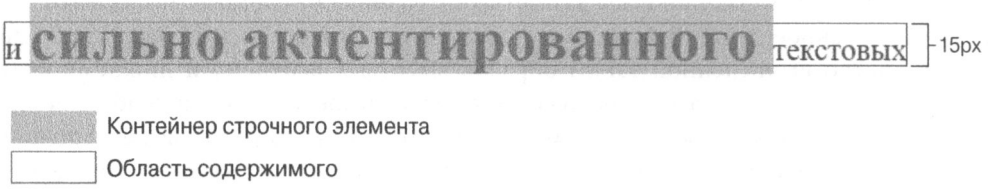


Рис. 7.38. Контейнеры строчных элементов, расположенные на базовой линии

Взаимное расположение контейнеров строчных элементов очень важно, поскольку определяет высоту текстовой строки, в которую они включены. Контейнер текстовой строки вычисляется как расстояние от верхнего края контейнера самого высокого элемента до нижнего края контейнера самого низкого ее элемента. Текстовые строки в блочном элементе располагаются так, что нижний край контейнера предыдущей строки соприкасается с верхним краем контейнера следующей строки. Общий вид абзаца, текстовый фрагмент которого изображен на рис. 7.38, показан на рис. 7.39.

Рис. 7.39. Контейнеры текстовых строк абзаца

Легко заметить, что средняя текстовая строка имеет большую высоту, чем первая и третья строки, но ее все равно недостаточно, чтобы вместить текст, представленный шрифтом большего размера. Нижний край контейнера второй строки совпадает с нижним краем контейнера анонимного текста, а его верхний край определяется положением верхнего края контейнера сильно акцентированного текста. Но поскольку

контейнер элемента `strong` имеет меньшую высоту, чем область содержимого, заключенный в него текст выходит за верхний и нижний края контейнера строки и может перекрываться с текстом соседних строк. Форматированию такого абзаца необходимо уделить особое внимание!

О том, как предотвратить расширение текста на соседние строки, снабдив их достаточным междустрочным интервалом, рассказывается далее.

Вертикальное выравнивание

При вертикальном выравнивании строчных элементов применяются рассмотренные выше принципы вычисления высоты их контейнеров. Рассмотрим, что произойдет, если в предыдущем примере элемент `strong` сместить вверх на 4 пикселя.

```
<p style="font-size: 12px; line-height: 12px;">
Этот абзац содержит примеры <em>обычного акцентированного</em> <br>
и <strong style="font-size: 24px; vertical-align: 4px;"> сильно
акцентированного </strong> текстовых фрагментов, последний из которых
представлен шрифтом большего <br> размера, чем окружающий его текст.
</p>
```

Сильно акцентированный тест смещается вверх на 4 пикселя вместе с областью содержимого и контейнером элемента. Поскольку контейнер элемента `strong` самый высокий в текстовой строке, его верхний край также перемещается вверх на обозначенные выше 4 пикселя (рис. 7.40).

Рис. 7.40. Увеличение высоты контейнера текстовой строки при выравнивании вложенного элемента

Рассмотрим несколько иную ситуацию. Предположим, что в одной с элементом `strong` текстовой строке находится еще один вложенный элемент с другим способом выравнивания.

```
<p style="font-size: 12px; line-height: 12px;">
Этот абзац содержит примеры <em>обычного акцентированного</em>,<br>
<strong style="font-size: 24px; vertical-align: 4px;"> сильно
акцентированного </strong> и <span style="vertical-align: top;">
приподнятого</span> фрагментов, средний из которых представлен
шрифтом большего <br> размера, чем окружающий его текст.
</p>
```

Результат отображения этого фрагмента документа в браузере такой же, как и в предыдущем случае: высота контейнера средней текстовой строки увеличивается на величину смещения элемента `strong`. Обратите внимание на способ выравнивания "приподнятого" текста (рис. 7.41).

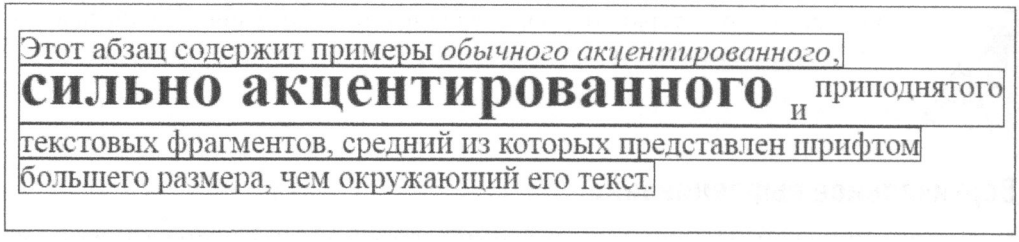


Рис. 7.41. Выравнивание строчных элементов относительно контейнера строки

В последнем примере верхний край контейнера элемента `span` выравнивается по верхнему краю контейнера строки. Так как значения свойств `font-size` и `line-height` у элемента `span` одинаковые, его контейнер и область содержимого будут иметь одинаковый размер. Несколько усложним задачу, видоизменив код предыдущего примера.

```
<p style="font-size: 12px; line-height: 12px;">
Этот абзац содержит примеры <em>обычного акцентированного</em>,<br>
<strong style="font-size: 24px;"> сильно акцентированного </strong>
и <span style="vertical-align: top; line-height: 2px;">приподнятого
</span> фрагментов, средний из которых представлен шрифтом большего
<br> размера, чем окружающий его текст.
</p>
```

Теперь ситуация изменилась: свойство `line-height`, применяемое к элементу `span`, имеет меньшее значение, чем свойство `font-size`. Это означает, что у данного элемента контейнер меньше области содержимого. Чтобы выровнять верхний край контейнера текстовой строки с верхним краем контейнера элемента `span`, последний смещается еще больше вверх, заходя в область содержимого первой строки (рис. 7.42).

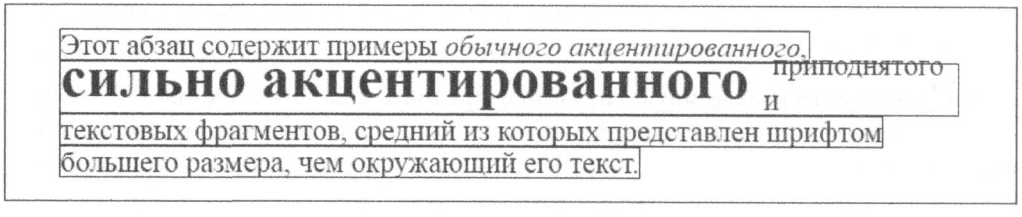


Рис. 7.42. Строчный элемент выступает за пределы контейнера текстовой строки

Свойство `line-height` элемента `span` имеет значение `18px`, обеспечивая интерлиньяж 6 пикселей. Таким образом, контейнер строчного элемента расширяется на

3 пикселя вверх и вниз относительно области содержимого, а его высота увеличивается до 18 пикселей. Только после проведения всех необходимых расчетов браузер совмещает верхнюю часть контейнера строки с верхней частью контейнера элемента `span`. Если бы свойству `vertical-align` передавалось значение `bottom`, то элемент `span` выравнивался бы своим нижним краем по нижней части контейнера текстовой строки.

Свойству `vertical-align` можно передавать целый ряд специальных ключевых слов, каждое из которых определяет свой способ выравнивая строчных элементов по вертикали.

`top`

Выравнивает верхний край контейнера строчного элемента по верхнему краю контейнера текстовой строки.

`bottom`

Выравнивает нижний край контейнера строчного элемента по нижнему краю контейнера текстовой строки.

`text-top`

Выравнивает верхний край контейнера строчного элемента по верхнему краю области содержимого родительского элемента.

`text-bottom`

Выравнивает нижний край контейнера строчного элемента по нижнему краю области содержимого родительского элемента.

`middle`

Выравнивает среднюю точку элемента по линии, приподнятой на расстояние `0.5ex` над базовой линией родительского элемента.

`super`

Смещает контейнер и область содержимого строчного элемента вверх. Расстояние смещения в спецификации CSS не указывается и задается пользовательским агентом.

`sub`

Смещает контейнер и область содержимого строчного элемента вниз. Расстояние смещения в спецификации CSS не указывается и задается пользовательским агентом.

`<percentage>`

Смещает элемент вверх или вниз на расстояние, измеряемое относительно значения свойства `line-height`.

Свойство line-height

В предыдущем разделе было показано, что значение свойства line-height в значительной степени влияет на способ вертикального выравнивания и расстояние смещения строчного элемента. Однако в рассмотренных выше примерах оно применяется исключительно к отдельным строчным элементам. Но как избежать перекрывания текстовых строк при включении свойства line-height в стилевые правила, селекторы которых указывают на большое количество элементов?

Один из способов решения такой задачи заключается в передаче свойству line-height значения, выраженного в единицах em, которое изменяется пропорционально размеру шрифта строчного элемента:

```
p {line-height: 1em;}
big {font-size: 250%; line-height: 1em;}

<p>
Наряду с "обычным" текстом в этот абзац включен текстовый <br>
фрагмент <big>с большим размером шрифта</big>.<br> Он применяется
в иллюстративных целях.
</p>
```

Устанавливая значение свойства line-height для элемента big описанным выше способом, можно добиться пропорционального изменения высоты контейнера текстовой строки, тем самым предоставив строчному элементу увеличенного размера достаточно места для отображения и предотвратив перекрывание соседних текстовых строк. Данный метод позволяет добиться поставленной цели без изменения значений свойств у всех остальных элементов абзаца. В данном случае значение 1em обеспечивает равенство значений свойств line-height и font-size. Не стоит забывать, что значение свойства line-height определяется относительно значения свойства font-size строчного, а не родительского элемента. Результат выполнения представленного выше кода показан на рис. 7.43.

Рис. 7.43. Правильный способ установки свойства line-height

Удостоверьтесь, что полностью понимаете, о чем шла речь в последних нескольких разделах, поскольку в дальнейшем мы будем рассматривать еще более сложные задачи, требующие добавления границ к элементам. В качестве простого примера рассмотрим ситуацию заключения гиперссылки в рамку толщиной 5 пикселей:

```
a:link {border: 5px solid blue;}
```

Если при решении этой задачи свойство line-height не снабдить значением, достаточно большим для включения границ указанной толщины, то, скорее всего,

соседние текстовые строки будут перекрываться. Для расширения контейнера гиперссылки можно воспользоваться методикой из предыдущего примера, предложенной для элемента `big`. В данном случае значение свойства `line-height` нужно сделать на 10 пикселей больше, чем размер шрифта (`font-size`) гиперссылок. Тем не менее, если размер шрифта гиперссылок неизвестен, то эта методика окажется неэффективной.

Еще один способ решения задачи заключается в увеличении значения свойства `line-height` для всего абзаца. Он позволяет эффективно увеличить контейнеры всех текстовых строк, а не только контейнера гиперссылки, заключаемой в рамку.

```
p {line-height: 1.8em;}
a:link {border: 5px solid blue;}
```

Добавив дополнительное пустое пространство между текстовыми строками абзаца, можно не бояться, что рамка будет наползать на соседние с гиперссылкой строки (рис. 7.44).

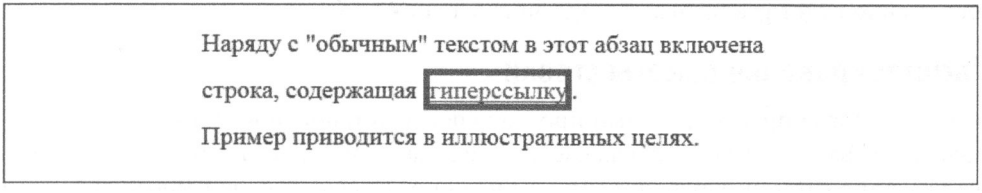


Рис. 7.44. Увеличение значения свойства `line-height` для всего абзаца приводит к образованию дополнительного пространства между строками

Данный способ эффективен только в случаях, когда абзац содержит текст одного размера. Добавление в текст абзаца элементов строчного типа, имеющих другой размер шрифта, неизбежно приводит к изменению высоты контейнеров отдельных текстовых строк. В таких условиях очень сложно предугадать расположение рамки гиперссылки. Рассмотрим следующий пример.

```
p {font-size: 14px; line-height: 24px;}
a:link {border: 5px solid blue;}
big {font-size: 150%; line-height: 1.5em;}
```

Согласно представленным правилам, высота контейнера элемента `big`, добавленного в текст абзаца, равна 31,5 пикселя (14×1,5×1,5). Такое же значение имеет и высота контейнера текстовой строки. Чтобы согласовать положение базовых линий строчного элемента и текстовой строки, свойство `line-height` элемента `p` нужно установить в значение `32px`.

Высота строки и положение базовой линии

Как было показано выше, действительная высота контейнера каждой текстовой строки зависит от способов выравнивания вложенных в нее элементов. Наряду с этим смещение выравниваемых элементов не в последнюю очередь определяется положением базовых линий строчных элементов (анонимного текста), поскольку именно они устанавливают взаимное расположение элементов в текстовой строке. Стоит

заметить, что каждый шрифт характеризуется своим положением базовой линии в пределах кегельной площадки. Сведения о расположении базовой линии хранятся в файле шрифта и подлежат изменению только с помощью редактора шрифтов.

Умение согласовывать положение базовых линий вложенных в текстовую строку элементов в большей степени является искусством, чем техническим навыком. Самый простой способ избежать дальнейших трудностей с выравниванием элементов состоит в установке значений свойств `line-height` и `font-size` в относительных единицах измерения, таких как `em`. Задача сильно усложняется или даже становится невозможной для решения при передаче свойствам значений, выраженных в разных единицах измерения. На момент написания книги рабочая группа CSS активно занималась разработкой свойств, призванных автоматически корректировать высоту контейнеров текстовых строк в зависимости от их содержимого. Включение их в спецификацию сильно упростило бы решение широкого круга задач, связанных с оформлением текста, который подлежит стилевому форматированию. К сожалению, на сегодняшний день решение так и не найдено, но всегда хочется верить, что оно непременно будет реализовано в будущих версиях CSS.

Масштабирование высоты строки

Как показала практика, самый простой способ установки высоты строки с помощью свойства `line-height` заключается в передаче ему простых числовых значений без указания единиц измерения. Такое число представляет собой множитель, который наследуется, но не относится к вычисляемым значениям. Рассмотрим, как в рамках этого метода создается стилевое правило, применяющее свойство `line-height` со значением, в полтора раза превышающим значение свойства `font-size`, ко всем элементам документа:

```
body {line-height: 1.5;}
```

Коэффициент масштабирования 1,5 передается от родительских элементов к дочерним на всех уровнях иерархической структуры, увеличивая значение свойства `line-height` относительно значения свойства `font-size` для каждого элемента документа. Результат применения такого правила к следующему фрагменту HTML-документа показан на рис. 7.45.

```
p {font-size: 15px; line-height: 1.5;}
small {font-size: 66%;}
big {font-size: 200%;}

<p>Свойство line-height этого абзаца имеет значение, превышающее
значение свойства font-size в 1,5 раза. Все вложенные в него
элементы, как, например, <small>этот с уменьшенным размером
шрифта</small> или <big>этот, представленный шрифтом увеличенного
размера,</big> также характеризуются свойством line-height
с коэффициентом масштабирования 1.5. Передавая множитель в качестве
значения свойства line-height, можно добиться автоматического
изменения высоты строки при представлении элементов с любым размером
шрифта.</p>
```

Рис. 7.45. Результат передачи коэффициента масштабирования свойству `line-height`

В этом примере высота строки элемента `small` составляет 15 пикселей, а у элемента `big` она равна 45 пикселям (такой разброс значений может показаться неуместным, но он полностью согласуется с настройками, принятыми по умолчанию). Чтобы уменьшить чрезмерно большой междустрочный интервал над элементом `big`, необходимо подобрать ему другое значение свойства `line-height`, заменяющее наследуемый от родительского элемента коэффициент масштабирования.

```
p {font-size: 15px; line-height: 1.5;}
small {font-size: 66%;}
big {font-size: 200%; line-height: 1em;}
```

Следующее решение — скорее всего, самое простое из предложенных — заключается в применении стилевых правил, устанавливающих высоту вложенных элементов не больше, чем того требует их содержимое. Такого результата можно достичь объявлением свойства `line-height` со значением `1.0`. Такой множитель указывает на равенство значений свойств `line-height` и `font-size` для каждого вложенного элемента строки, а потому и полное совпадение контейнера элемента с областью содержимого. Это минимальный коэффициент масштабирования, обеспечивающий размещение содержимого элемента в пределах его контейнера.

У большинства шрифтов символы имеют меньший размер, чем их кегельные площадки, образуя дополнительное свободное пространство в области содержимого текстовой строки. Исключение составляют курсивные шрифты, символы которых обычно выходят за пределы кегельных площадок.

Добавление полей, границ и отступов к незамещаемым элементам

Из предыдущих разделов вам известно, что поля, границы и отступы могут добавляться не только к замещаемым, но и незамещаемым элементам. Тем не менее они не оказывают влияния на высоту контейнера строки, в которую помещены. Например, добавив к элементу `span` одни только границы (без полей и отступов), можно получить результат, показанный на рис. 7.46.

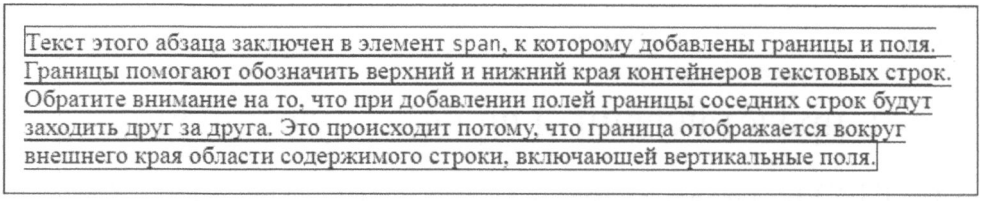


Рис. 7.46. Границы строчных элементов, вложенных в текстовую строку

Положение границ незамещаемых строчных элементов определяется значением свойства `font-size`, а не свойства `line-height`. Из этого следует, что строчный элемент, к которому применено свойство `font-size` со значением `12px` и свойство `line-height`, установленное в значение `36px`, будет характеризоваться областью содержимого с высотой `12px`.

Чтобы удостовериться в неизменности положения границ строчного элемента при добавлении к нему полей, нужно применить к нему следующее правило:

```
span {padding: 4px;}
```

Поскольку поля не включаются в область содержимого, высота контейнера строчного элемента останется неизменной. Подобным образом границы строчного элемента не влияют на размер контейнера текстовой строки, как показано на рис. 7.47.

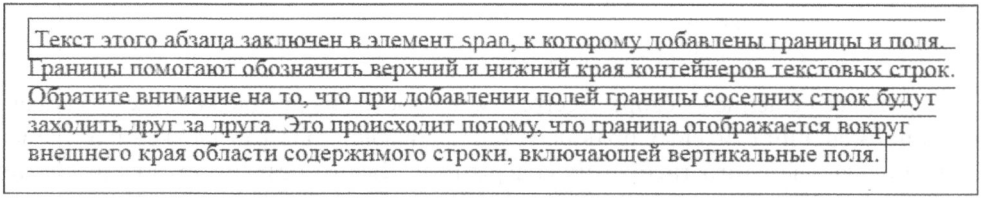


Рис. 7.47. Поля и границы строчных элементов не влияют на высоту строки

Что касается отступов, то они редко добавляются к верхнему и нижнему краям незамещаемых строчных элементов, поскольку также не изменяют высоту текстовой строки. Совсем другая ситуация наблюдается с боковыми отступами.

Исходно любой строчный элемент размещается в одной строке. Если длины этой строки не хватает для размещения строчного элемента целиком, то он может разделяться на несколько частей, каждая из которых переносится на следующую строку. Следовательно, боковые отступы образуют области дополнительного пустого пространства перед строчным элементом и после него, смещая остальное содержимое строки вправо. Подобным образом обрабатываются и боковые поля строчного элемента. Таким образом, хотя поля, отступы (и границы) строчных элементов не учитываются браузером при вычислении высоты тестовых строк, они оказывают самое непосредственное влияние на положение остального содержимого родительского элемента. Более того, отрицательные боковые отступы и поля могут использоваться для уплотнения и даже перекрывания соседнего содержимого (рис. 7.48).

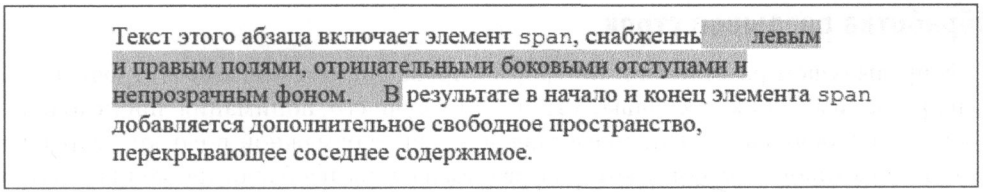


Рис. 7.48. Строчный элемент с боковыми полями и отступами

Опять-таки, строчный элемент можно представить как полоску бумаги, наклеенную на пластиковую основу. Представление строчного элемента в нескольких текстовых строках требует разрезания бумажной полоски на несколько частей. При этом в местах разрезания отступы не образуются: они сохраняются только в местах начального объявления (в начале и в конце строчного элемента). Такое поведение вложенного элемента принято по умолчанию и при необходимости может быть изменено, как рассказано далее.

Давайте проанализируем ситуацию, в которой к строчным элементам добавляются поля настолько большой ширины, что их фоны перекрываются. В качестве примера рассмотрим следующие стилевые правила.

```
p {font-size: 15px; line-height: 1em;}
p span {background: #FAA; padding-top: 10px; padding-bottom: 10px;}
```

Согласно приведенному выше коду область содержимого элемента `span` имеет высоту 15 пикселей, а верхнее и нижнее поля — 10 пикселей каждое. Большие значения свойств не приведут к увеличению высоты строки, а всего лишь обеспечат расширение фона строчного элемента на соседние строки, как показано на рис. 7.49.

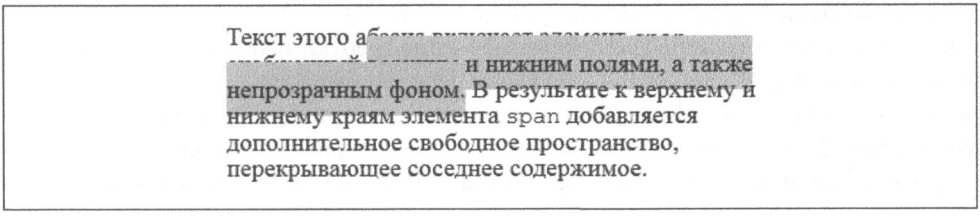


Рис. 7.49. Перекрывание фона строчного элемента, разнесенного на несколько строк

В CSS2.1 поддерживается обычный порядок визуализации элементов элемента, поэтому границы каждого следующего элемента отображаются поверх границ и содержимого предыдущего элемента. Этот же принцип сохраняется при визуализации фона вложенных элементов. Тем не менее спецификация CSS2 разрешает пользовательским агентам обрезать границы и области полей строчных элементов, вообще не отображая их в документах. В силу вышесказанного результат выполнения приведенных выше правил будет сильно зависеть от спецификации CSS, поддерживаемой браузером.

Обработка разрывов строк

В предыдущем разделе описано форматирование незамещаемых строчных элементов, занимающих несколько строк. Для лучшего понимания происходящих процессов строчный элемент представлялся в виде бумажной полоски, закрепленной на пластиковой основе, которая разрезалась на необходимое количество частей. В данной аналогии каждый из разрезов соответствует разрыву строки. Способ форматирования боковых краев каждого из фрагментов строчного элемента, полученного в результате разделения строчного элемента, определяется свойством `box-decoration-break`.

box-decoration-break

Значение	`slice` | `clone`
Начальное значение	`slice`
Применяется	Все элементы
Вычисляется	Согласно определению
Наследуется	Нет
Анимируется	Нет

Пример использования значения по умолчанию (`slice`) приведен в предыдущем разделе. Значение `clone` обязывает браузер форматировать каждый из фрагментов так, как если бы он представлял собой отдельный элемент. Чтобы понять, как это проявляется, внимательно изучите рис. 7.50, на котором показан один и тот же документ, отформатированный одинаковыми стилевыми правилами, которые различаются только значениями свойства `box-decoration-break`.

Отличия хорошо заметны, но некоторые из них требуют особого внимания. В частности, на втором примере, соответствующем передаче свойству `box-decoration-break` ключевого слова `clone`, явно видно, что боковые отступы добавлены к каждому из фрагментов, а не только к первому и последнему. Кроме того, фрагменты второго примера заключены в отдельные рамки (границы), в противоположность общим границам строчного элемента, разделенным на фрагменты в первом случае.

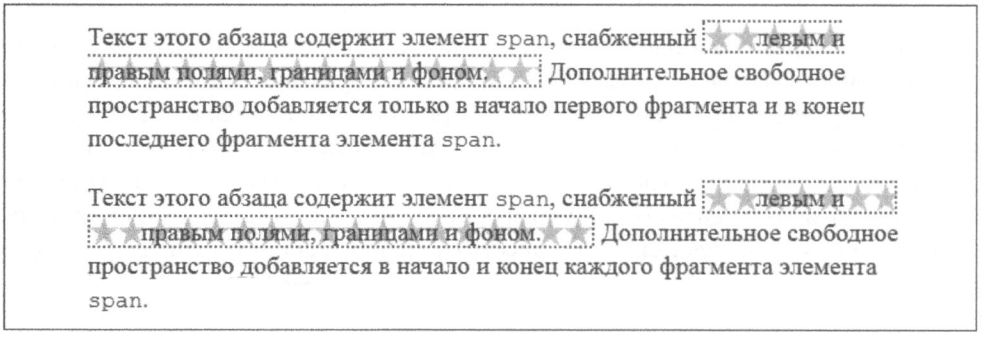


Рис. 7.50. Фрагменты строчного элемента, отформатированные в режимах `slice` *и* `clone`

Если внимательно присмотреться, то становится очевидным, что свойство `background-image` применяется по-разному в каждом из случаев. В первом примере фоновое изображение разрезается на отдельные фрагменты вместе с остальным содержимым строчного элемента. Это свидетельствует о том, что он сохраняет исходное фоновое изображение. Во втором примере фон каждого фрагмента представлен отдельной копией исходного фонового изображения. Если бы в качестве фона применялось изображение с неповторяющимся узором, то разница между фонами в каждом из представлений была бы очевидной.

Свойство `box-decoration-break` используется преимущественно для форматирования строчных элементов, хотя обладает более широкой областью применения. Например, оно точно пригодится в многостраничных документах, элементы которых зачастую переносятся на следующую страницу с разрывом содержимого. Чтобы обозначить принадлежность обоих фрагментов к общему элементу, его нужно разделять в режиме `slice`. Применив к фрагментам элемента объявление `box-decoration-break: clone`, можно добиться форматирования их одинаковым фоном, отступами, границами и полями. Вышесказанное относится и к многоколоночным макетам: свойство `box-decoration-break` прекрасно справляется с форматированием фрагментов строчных элементов, разнесенных по разным колонкам.

Размеры символов и область содержимого

Несмотря на все наши усилия, рано или поздно вам придется столкнуться с перекрыванием фонов незамещаемых строчных элементов. Виной тому шрифты, символы которых несколько больше кегельной площадки. Как ни странно это признавать, но у большинства шрифтов размер символов не совпадает с размером кегельной площадки.

Звучит неправдоподобно, но имеет вполне разумное объяснение. В спецификации CSS2.1 оговаривается, что высота области содержимого зависит от размера шрифта, но в ней не указывается, каким именно образом. Решение принимается пользовательским агентом: она может равняться высоте кегельной площадки или расстоянию от нижнего края нижнего выносного элемента до верхнего края верхнего выносного элемента символов шрифта. (В последнем случае высота области содержимого должна быть больше размера любых символов, выносные элементы которых выступают за область кегельной площадки.)

Иными словами, область представления содержимого незамещаемых строчных элементов определяется исключительно пользовательским агентом. Если ее высота совпадает с размером кегельной площадки (значение свойства `font-size`), то и строчный элемент будет заполняться фоном на высоту кегельной площадки. В противном случае высота области содержимого, а потому и фонового изображения будет равняться расстоянию от нижней точки нижнего выносного элемента до верхней точки верхнего выносного элемента, которое не совпадает с размером кегельной площадки. Если при этом строчному элементу назначить свойство `line-height` со значением `1em`, то фоновые изображения его фрагментов, расположенных в соседних строках, будут перекрываться.

Замещаемые строчные элементы

Размер замещаемых строчных элементов, подобных изображениям, определяется ими самими. В частности, размер изображения указывается его высотой и шириной, выраженных в пикселях. Таким образом, замещаемый элемент требует увеличения высоты строки, в которую он помещается. Учтите, что вставка замещаемого элемента в строку *не* вызывает изменения значения свойства `line-height` как у него самого, так и у остальных элементов строки. Разумеется, строка изменяет свой размер так, чтобы вместить замещаемые элементы со всем их содержимым. Иначе говоря, контейнер замещаемого элемента содержит не только область содержимого и отступы, но и поля с границами. Приведенные ниже правила демонстрируют пример подобного форматирования, результат которого показан на рис. 7.51.

```
p {font-size: 15px; line-height: 18px;}
img {height: 30px; margin: 0; padding: 0; border: none;}
```

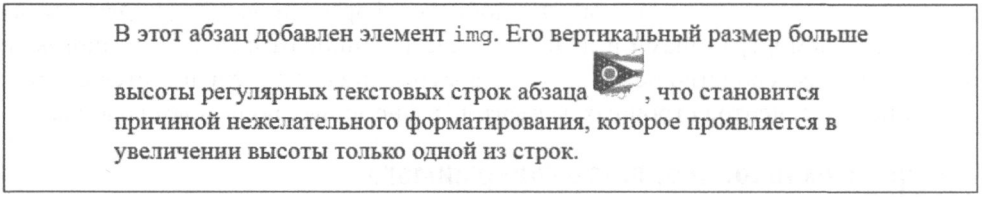


Рис. 7.51. Вставка замещаемых элементов в строку приводит к изменению ее высоты, но не значения свойства `line-height`

Несмотря на увеличение свободного пространства между строками, действительное значение свойства `line-height` остается неизменным как у абзаца, так и у замещаемого элемента. А все потому, что это свойство не воздействует на замещаемые строчные элементы по своему определению. Поскольку у элемента изображения, показанного на рис. 7.51, отсутствуют поля, границы и отступы, высота его контейнера в точности совпадает с высотой области содержимого и составляет 30 пикселей.

Тут можно возразить, что замещаемый элемент все же снабжается свойством `line-height`. Зачем же оно ему нужно? Вероятнее всего, его значение определяет положение элемента при его вертикальном выравнивании в строке. Если такое предположение верно, то процентное значение, передаваемое свойству `vertical-align`, будет указывать величину смещения элемента относительно значения его свойства `line-height`, что демонстрируется таким примером.

```
p {font-size: 15px; line-height: 18px;}
img {vertical-align: 50%;}

<p>Вложенное в абзац изображение <img src="test.gif" alt="test image">
приподнимается на 9 пикселей.</p>
```

Согласно принципу наследования, под которое подпадает и значение свойства `line-height`, изображение смещается вверх точно на 9 пикселей. Если лишить первое правило свойства `line-height`, то вертикальное выравнивание в указанном

формате (50%) осуществить не получится. Высота изображения не подходит для вычисления значения вертикального выравнивания — она рассчитывается относительно одного только значения свойства line-height.

Существует еще одна причина назначения свойства line-height замещаемому элементу: наследование его значения дочерними элементами, например SVG-изображениями, в которых стилевые правила применяются для форматирования текста.

Добавление полей, границ и отступов к замещаемым элементам

Если вы внимательно ознакомились с приведенными выше замечаниями, то у вас не должно возникать затруднений с форматированием замещаемых строчных элементов, снабжаемых полями, границами и отступами.

Поля и границы замещаемых элементов выполняют привычные им функции: поля образуют свободное пространство вокруг содержимого элемента, а границы отделяют поля от отступов. Примечательно то, что и поля, и границы учитываются при вычислении высоты контейнера строки, поскольку являются неотъемлемой частью контейнера замещаемого строчного элемента (в отличие от незамещаемого строчного элемента). Наглядный пример приведен на рис. 7.52, — он получен в результате выполнения браузером следующего стилевого форматирования.

```
img {height: 50px; width: 50px;}
img.one {margin: 0; padding: 0; border: 3px dotted;}
img.two {margin: 10px; padding: 10px; border: 3px solid;}
```

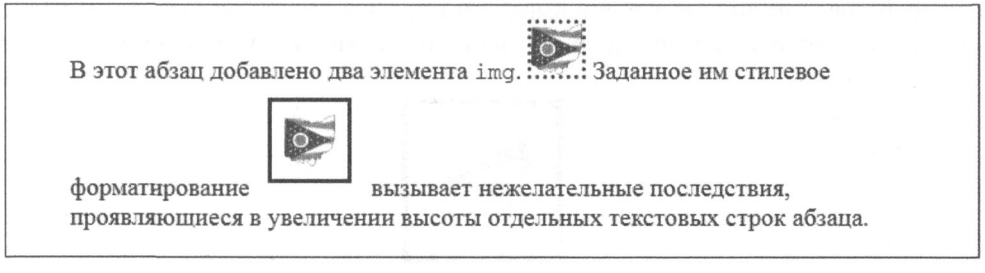


Рис. 7.52. Увеличение высоты строки при добавлении полей и границ к замещаемому строчному элементу

Заметьте, что высота первой строки рассчитана на включение только одного изображения, в то время как высота второй строки предполагает размещение в ней изображения вместе с полями и границами.

Отступы также включаются в контейнер замещаемого элемента, но их поведение далеко не всегда однозначно. Положительные отступы форматируются вполне предсказуемо; неожиданности возникают при добавлении к замещаемым элементам отрицательных отступов. Как показано на рис. 7.53, отрицательный отступ становится причиной уменьшения высоты контейнера строчного элемента:

```
img.two {margin-top: -10px;}
```

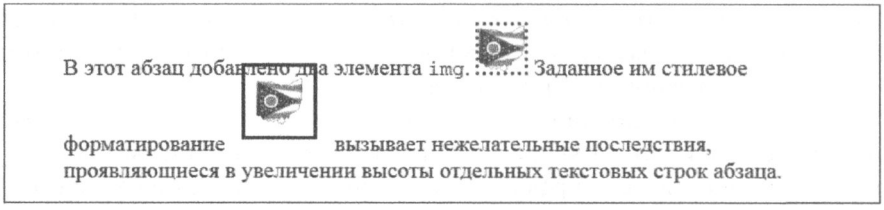


Рис. 7.53. Результат добавления отрицательных отступов к замещаемому строчному элементу

В данном случае отрицательные отступы обрабатываются пользовательским агентом так же, как и в элементах блочного уровня. Установка отрицательных отступов — это единственный способ уменьшения размера контейнеров замещаемых строчных элементов, позволяющий включать их в строки заведомо меньшей, чем предполагает размер изображения, высоты. Благодаря этой особенности замещаемые строчные элементы часто рассматриваются как строчно-блочные элементы.

Замещаемые строчные элементы и базовая линия

Все рассмотренные выше примеры соответствовали размещению замещаемых элементов на базовой линии строки. Тем не менее при добавлении отступов, границ и/или полей область содержимого замещаемого элемента увеличивается и смещается вверх (предполагается, что свойство `box-sizing` установлено в значение по умолчанию — `content-box`). Поскольку базовая линия у замещаемых элементов отсутствует, то по базовой линии строки они выравниваются своим нижним краем. Чаще всего его роль отводится внешнему краю нижнего отступа, как показано на рис. 7.54.

Рис. 7.54. Замещаемый строчный элемент располагается на базовой линии текстовой строки

Такой способ выравнивания часто приводит к получению нежелательных результатов. Например, при помещении изображения в ячейку таблицы ее размер увеличивается так, чтобы вместить не только само изображение, но и контейнер, в который оно заключено. Автоматическое изменение размера ячейки выполняется даже в случаях помещения в нее одного только изображения и отсутствия иного содержимого. (В современном веб-дизайне нет места таким подзабытым приемам верстки, как позиционирование разрезанных изображений и прозрачных GIF-файлов, но на их примере удобно объяснять поведение браузеров при форматировании замещаемых элементов.) Рассмотрим следующий код.

```
td {font-size: 12px;}

<td><img src="spacer.gif" height="1" width="10"></td>
```

Согласно модели стилевого форматирования строчных элементов ячейка, которая содержит изображение, выровненное по базовой линии нижним краем, имеет высоту 12 пикселей. Это означает, что свободное пространство под изображением имеет высоту 3 пикселя, а над ним — 8 пикселей. Учтите, что указанные значения зависят от размера шрифта и положения базовой линии ячейки.

Описанный выше способ форматирования характерен не только для изображений, вставленных в ячейку таблицы, но и любых других замещаемых строчных элементов, вложенных в элементы блочного уровня или элементы уровня ячеек таблицы. В частности, изображение, помещенное внутрь элемента `div`, также располагается на его базовой линии.

Во избежание трудностей при форматировании таблиц изображения, размещаемые в их ячейках, нужно представлять элементами блочного, а не строчного типа, так как они не требуют наличия базовой линии.

```
td {font-size: 12px;}
img.block {display: block;}

<td><img src="spacer.gif" height="1" width="10" class="block"></td>
```

В качестве альтернативного решения можно попробовать передать свойствам `font-size` и `line-height` ячейки, содержащей изображение, значение `1px` — высота ячейки будет уменьшена до вертикального размера изображения.

На момент написания книги многие браузеры отказывались применять модель форматирования строчных элементов в рассмотренном выше контексте.

Замещаемые строчные элементы, помещаемые на базовую линию родительского элемента, можно эффективно смещать вниз, задавая им отрицательный нижний отступ. Такое поведение вызвано опусканием области содержимого относительно границ контейнера строчного элемента. Таким образом, приведенное ниже правило обеспечивает результат, проиллюстрированный на рис. 7.55.

```
p img {margin-bottom: -10px;}
```

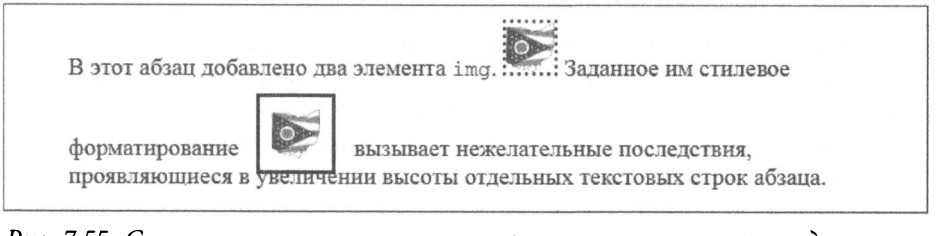


Рис. 7.55. Смещение вниз замещаемого строчного элемента при задании ему отрицательного нижнего отступа

В результате такого смещения замещаемый элемент может перекрывать расположенный ниже текст.

О модели форматирования строчных элементов

В спецификацию CSS модель форматирования строчных элементов вписывается очень органично, хотя зачастую приводит к получению неожиданных результатов. К сожалению, неоднозначность и запутанность правил форматирования строчных элементов являются следствием обеспечения обратной совместимости с предыдущими версиями технологии и включением в текущую спецификацию инструментов, востребованных в будущем, — достаточно неуклюжая попытка объять необъятное, как по мне.

В частности, функции обтекания изображений текстом и выравнивания вертикального текста заимствованы у браузера Mosaic 1.0. В нем все изображения, вставленные в абзац, автоматически раздвигали окружающий их текст, чтобы предотвратить перекрывание с ним. Несомненно, мудрое решение, позволяющее избежать большого количества проблем при визуализации документов. Взяв такое поведение браузера за основу, разработчики CSS сделали все возможное, чтобы повторить его в собственной модели форматирования текстовых и строчных элементов. К сожалению, оно оказалось оправданным далеко не во всех случаях. Например, элементы верхнего индекса (`sup`) представляются намного лучше, если не смещают расположенный после них текст.

Функции автоматического позиционирования строчных элементов относительно базовой линии порядком раздражают многих авторов документов, привыкших определять смещение строчных элементов вручную. С другой стороны, позволив свойству `line-height` устанавливать точное расстояние смещения базовой линии строчных элементов, нужно будет вручную контролировать положение окружающих его текстовых строк, что совершенно неприемлемо. К счастью, технология CSS настолько функциональна, что позволяет добиться одного и того же результата разными способами, а в будущих ее версиях подобные возможности будут только совершенствоваться.

Строчно-блочные элементы

Как предполагает название значения `inline-block`, строчно-блочные элементы совмещают в себе функции элементов как строчного, так и блочного типа. Впервые строчно-блочные элементы были представлены в CSS2.1.

С остальными элементами документа строчно-блочные элементы взаимодействуют как элементы строчного уровня. Это означает, что они размещаются в текстовой строке подобно изображениям и рассматриваются браузером как замещаемые строчные элементы. Следовательно, по умолчанию они привязываются к базовой линии текстовой строки своим нижним краем и не переносятся на другой строку.

Внутренняя структура строчно-блочных элементов такова, что их содержимое рассматривается браузерами как принадлежащее элементам блочного типа. К ним, как и к замещаемым строчным, а также блочным элементам, применяются свойства `width` и `height` (а потому и `box-sizing`), позволяющие увеличить высоту текстовой строки для предотвращения перекрывания их текстом соседних строк.

Чтобы лучше понять, что собой представляют строчно-блочные элементы, рассмотрим следующий фрагмент HTML-документа.

```
<div id="one">
Этот текстовый фрагмент представлен блочным элементом, в который
вложен еще один элемент блочного уровня. <p>Вложенный блочный
элемент</p> Остальная часть блочного элемента div.
</div>
Этот текстовый фрагмент представлен блочным элементом, в который
вложен строчный элемент. <p>Вложенный строчный элемент</p> Остальная
часть блочного элемента div.
</div>
<div id="three">
Этот текстовый фрагмент представлен блочным элементом, в который
вложен еще один строчно-блочный элемент. <p>Вложенный строчно-блочный
элемент</p> Остальная часть блочного элемента div.
</div>
```

К нему применяются такие стилевые 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 визуализации документа в браузере показан на рис. 7.56.

Как видите, во втором элементе `div` вложенный абзац представляется элементом строчного типа, поэтому он форматируется так же, как и остальное содержимое (назначенные ему свойства `width` и `text-align` игнорируются браузером). При этом в третьем элементе `div` вложенный абзац рассматривается как строчно-блочный, поэтому свойства `width` и `text-align` к нему применяются в полной мере. Также при расчете высоты строки учитываются отступы строчно-блочного элемента, как это было в случае замещаемого строчного элемента.

Если свойство `width` строчно-блочного элемента не задать в явном виде или передать ему значение `auto`, то его контейнер автоматически расширится до размеров содержимого. Иными словами, контейнер элемента получит размер, в точности соответствующий размеру содержимого, — ни больше ни меньше. Подобное поведение свойственно строчным элементам, которые к тому же могут переноситься на другие строки, разделяясь на несколько фрагментов. Строчно-блочные элементы не умеют этого делать по определению. Исходя из вышесказанного, применим к имеющемуся документу следующее правило:

```
div#three p {display: inline-block; height: 4em;}
```

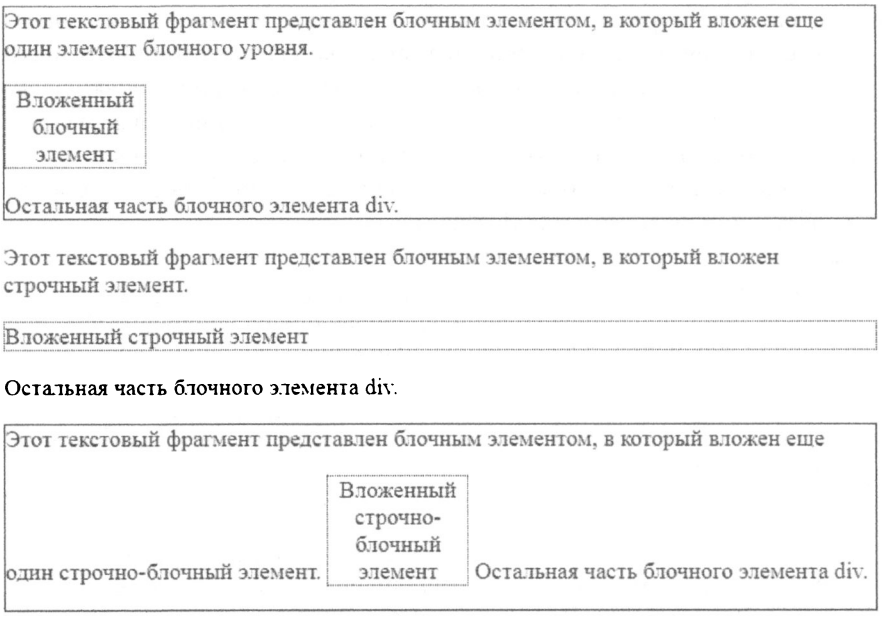


Рис. 7.56. Форматирование строчно-блочного элемента

Контейнер строчно-блочного элемента расширится до размера содержимого, не изменяя своей исходной высоты (рис. 7.57).

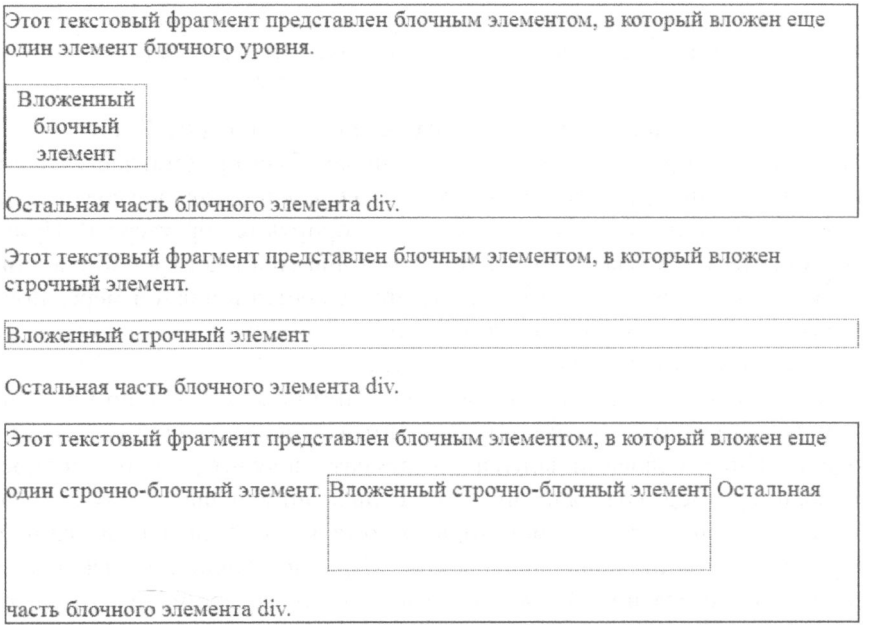


Рис. 7.57. Результат автоматической установки ширины строчно-блочного элемента

Строчно-блочные элементы находят применение на панелях навигации, содержащих наборы тематически сгруппированных гиперссылок одинаковой ширины. Следующее правило позволяет установить ширину таких гиперссылок равной 20% от ширины родительского элемента:

```
nav a {display: inline-block; width: 20%;}
```

Такого же результата можно добиться, используя flex-контейнеры при верстке панели навигации. Они лучше приспособлены для решения не только этой, но и многих других задач.

Перетекание содержимого

Отдельного описания требуют значения `flow` и `flow-root` свойства `display`. Объявление `display: flow` определяет обычный *контекст форматирования* или порядок (поток) размещения элементов в документе. До тех пор пока значение `flow` комбинируется со значением `inline`, элемент заключается в контейнер строчного уровня.

Другими словами, в следующем примере два первых правила нзначают элементам блочный контекст форматирования и только последнее — строчный контекст.

```
#first {display: flow;}
#second {display: block flow;}
#third {display: inline flow;}
```

В модуле CSS Display 3 свойство `display` сделали составным: одна его часть отвечает за *внешнее* (по отношению к соседям), другая — за *внутреннее* (применительно к дочерним элементам) представление блока. Ключевые слова `block` и `inline` устанавливают внешнее поведение блока, определяющее способ его взаимодействия с окружающим содержимым. В приведенном выше стилевом правиле внутреннее форматирование блока задается значением `flow`.

Новый формат свойства позволяет задавать элементам самые разные контексты форматирования. Например, объявление `display: inline table` устанавливает такой же порядок размещения содержимого элемента, как и в ячейках таблицы, хотя внешне объявляет его строчным элементом (такое же поведение назначается элементу с помощью старого значения `inline-table`).

С другой стороны, объявление `display: flow-root` всегда приводит элемент к блочному типу, которому приписывается независимый контекст форматирования. Исходно такое поведение свойственно только корневому элементу документа (например, `html`), т.е. оно указывает, что элемент придерживается собственного потока размещения дочерних элементов.

Свойство `display` все еще поддерживает старые значения. Соответствие старых и новых значений свойства `display` приведено в табл. 7.1.

Таблица 7.1. Соответствие значений свойства `display`

Старые значения	Новые значения
`block`	`block flow`
`inline`	`block flow`
`inline-block`	`inline flow-root`
`list-item`	`list-item block flow`
`inline-list-item`	`list-item inline flow`
`table`	`block table`
`inline-table`	`inline table`
`flex`	`block flex`
`inline-flex`	`inline flex`
`grid`	`block grid`
`inline-grid`	`inline grid`

К концу 2017 года ключевые слова `flow` и `flow-root` поддерживались только браузерами Firefox и Opera.

Визуализация содержимого

Свойство `display` дополнено еще одним интересным значением, которое нельзя обойти вниманием: `contents`. В результате применения объявления `display: contents` элемент рассматривается так, будто в иерархической структуре документа его заменили непосредственными потомками. Ниже приведен наглядный пример использования ключевого слова `contents` в стилевом форматировании списка.

```
ul {border: 1px solid red;}
li {border: 1px solid silver;}

<ul>
<li>Первый элемент списка</li>
<li>Элемент II: Прослушивание</li>
<li>Последний третий элемент</li>
</ul>
```

Применение обозначенных выше правил к приведенному под ними фрагменту HTML-документа приводит к добавлению красной рамки ко всему неупорядоченному списку и заключению каждого элемента списка в границы серебристого цвета.

Если теперь к элементу `ul` добавить объявление `display: contents`, то он перестанет визуализироваться в документе, в отличие от вложенных в него элементов (списка). Разница между обозначенными выше способами представления неупорядоченного списка показана на рис. 7.58.

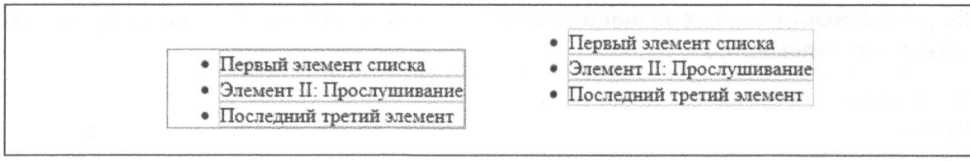


Рис. 7.58. Неупорядоченный список с исходным форматированием и после применения объявления `display: contents`

Элементы списка, как можно было предположить, из документа не исчезают — отсутствует только их родительский элемент. Именно по этой причине второй список, показанный на рис. 7.58, *справа*, лишен красной рамки, а сам он (в отсутствие отступов и полей элемента `ul`) приподнят вверх.

К концу 2017 года ключевое слово `contents` поддерживались только браузером Firefox.

Другие значения свойства `display`

В этой главе рассматривались далеко не все значения свойства `display`. Многие из них — в частности, применяемые для представления содержимого в виде таблиц — будут описаны в последующих главах. Кроме того, свойству `display` будет уделено внимание в главе, посвященной методам визуализации нумерованных списков и генерируемого содержимого.

В данной книге не рассматриваются значения, в названии которых встречается ключевое слово `ruby`, поскольку они слабо поддерживаются браузерами последних версий, а описанию их назначения можно посвятить отдельную книгу. Вы также не найдете в книге описание значения `run-in`, исключенного из спецификации CSS, но имеющего неплохие шансы на реинкарнацию под новым именем.

Вычисляемые значения

На вычисляемое значение свойства `display` оказывают воздействие значения свойств `position` и `float`. К тому же оно отличается для элемента `root`. В действительности значения свойств `display`, `position` и `float` комбинируются самым неожиданным образом.

У абсолютно позиционируемых элементов свойство `float` имеет значение `none`. Вычисляемые значения свойства `display` для точно позиционируемых и обтекаемых элементов зависят от объявленных значений так, как указано в табл. 7.2.

Таблица 7.2. Вычисляемые значения свойства `display` для точно позиционируемых и обтекаемых элементов

Объявленное 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	Согласно определению

Значения `inline-table` или `table`, объявленные для элемента `root`, соответствуют вычисляемому значению `table`, а значение `none` представляет такое же вычисляемое значение. Все остальные значения, объявленные для корневого элемента документа, сводятся к вычисляемому значению `block`.

Резюме

Несмотря на кажущуюся нелогичность при первом знакомстве, модель форматирования элементов, предложенная в CSS, становится все более понятной по мере ее углубленного изучения. Во многих случаях стилевые правила, исходно выглядящие совершенно нелепо или даже глупо, выполняют в таблицах стилей очень важные функции, предотвращая преобразование макета документа в набор хаотически нагроможденных элементов. Форматирование блочных элементов уже давно стало рутинной операцией при верстке документов, чего не скажешь об элементах строчного уровня, поведение которых зависит от огромного количества факторов — в первую очередь, от замещаемости их содержимого.

Поля, отступы, границы и рамки

В предыдущей главе рассматривались общие вопросы форматирования элементов основных типов. В этой главе будут описаны стилевые свойства и значения, определяющие форматирование только отдельных составляющих элементов. Речь идет о полях, границах, отступах и любых других обрамлениях, добавляемых к элементам документа.

Контейнеры элемента

Как известно, каждый элемент представляется прямоугольным *контейнером*, определяющим площадь, занимаемую им в макете документа. Исходя из этого, от размера контейнера всего одного элемента могут зависеть размер и положение многих других элементов документа. Например, если контейнер первого блочного элемента имеет высоту 2,5 см, то контейнер следующего элемента будет отстоять от верхнего края документа не менее чем на 2,5 см. Как только высота контейнера первого элемента увеличится до 5 см, контейнеры всех расположенных после него элементов сместятся вниз на 2,5 см. При этом контейнер второго элемента будет отстоять от верхнего края документа не менее чем на 5 см.

По умолчанию контейнеры элементов располагаются в макете документа так, чтобы не перекрываться. В правильно сверстанных документах они подгоняются максимально плотно друг к другому, при этом разграничиваясь свободным пространством так, чтобы предельно четко обозначить содержимое каждого элемента.

Контейнеры могут перекрываться только при ручном позиционировании. В документах с обычным порядком расположения элементов такое поведение обычно наблюдается при установке отрицательных отступов.

Чтобы понять, какой вклад вносят поля, границы и отступы в форматирование элементов, необходимо ознакомиться с блочной моделью элемента (рис. 8.1).

На схеме, приведенной на рис. 8.1, не показаны внешние контуры. О причине их исключения из блочной модели элемента рассказывается далее.

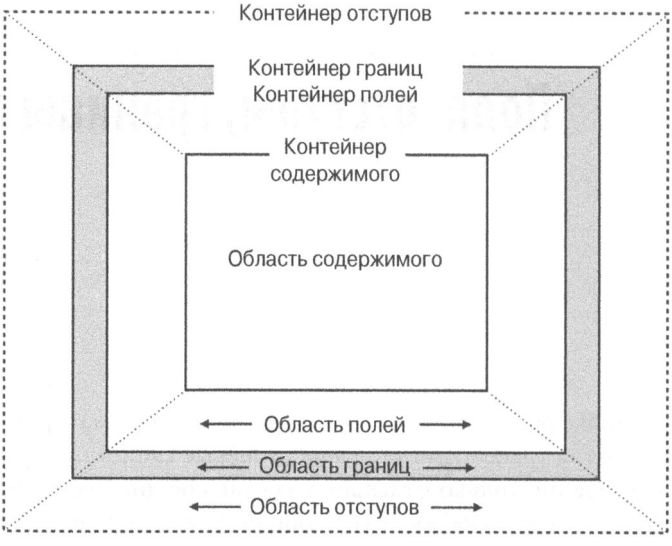


Рис. 8.1. Блочная модель элемента в CSS

Высота и ширина

Исторически так сложилось, что в явном виде обычно задается ширина и в меньшей степени высота контейнера элемента. По умолчанию ширина элемента определяется как расстояние от левого внутреннего края до правого внутреннего края содержимого элемента. Вертикальный и горизонтальный размеры элемента устанавливаются свойствами `height` и `width`.

width

Значение	*<length>* | *<percentage>* | `auto`
Начальное значение	`auto`
Применяется	Все элементы, кроме незамещаемых строчных элементов, строк таблиц и элементов уровня строк таблицы
Вычисляется	Согласно определению для значения `auto` и процентных значений; значения в единицах длины, за исключением случаев, когда оно не применимо к элементу (тогда `auto`)
Наследуется	Нет
Анимируется	Да

height

Значение	*<length>* | *<percentage>* | `auto`
Начальное значение	`auto`
Применяется	Все элементы, кроме незамещаемых строчных элементов, строк таблиц и элементов уровня строк таблицы

Вычисляется	Согласно определению для значения `auto` и процентных значений; значения в единицах длины, за исключением случаев, когда оно не применимо к элементу (тогда `auto`)
Наследуется	Нет
Анимируется	Да

Важное замечание по использованию этих свойств: они не применяются к незамещаемым строчным элементам. Например, попытка объявления свойств `height` и `width` к гиперссылке, представленной строчным элементом и вставленной в блочный элемент с обычным направлением заполнения содержимым, будет проигнорирована браузерами, поддерживающими стандарты CSS. В качестве примера рассмотрим следующее стилевое правило:

```
a:link {color: red; background: silver; height: 15px; width: 60px;}
```

Правило указывает окрашивать красным цветом все непосещенные гиперссылки, но не позволяет изменить их высоту и ширину: они полностью определяются содержимым элементов `a`. Несмотря на явно заданные установки, размер гиперссылок *не* будет составлять 15×60 пикселей. Чтобы гиперссылки были представлены в указанном стилевым правилом размере, для них нужно объявить свойство `display` со значением `inline-block` или `block`.

К концу 2017 года в фазе активного обсуждения находилось сразу несколько новых значений свойств `height` и `width`: `stretch`, `min-content`, `max-content` и `fit-content` (в двух вариантах). С их поддержкой возникли большие трудности, поэтому они вряд ли найдут воплощение в пользовательских агентах в ближайшем будущем.

В дальнейшем предполагается, что высота элемента всегда устанавливается автоматически. Например, если элемент занимает целых восемь текстовых строк, а каждая строка имеет сантиметровую высоту, то общая высота элемента составит 8 см. Элемент, занимающий 10 строк, будет иметь высоту 10 см. В большинстве случаев высота элемента определяется содержимым элемента, а не автором документа. Устанавливать ее вручную для элементов с обычным направлением заполнения содержимым приходится очень редко.

Назначение свойств `height` и `width` легко изменяется с помощью свойства `box-sizing`. В этой главе оно детально не рассматривается, хотя с его применением не должно возникнуть больших сложностей. Оно указывает область применения свойств, устанавливающих размеры элементов: область содержимого (`content-box`) или границы элемента (`border-box`). В дальнейшем предполагается, что свойства `height` и `width` устанавливают размер области содержимого, что соответствует объявлению `box-sizing: content-box`.

Поля

Поля находятся снаружи области содержимого, располагаясь между нею и границами элемента. Для непосредственной установки полей в стилевых правилах служит свойство `padding`.

padding

Значение	*<length>* | *<percentage>* | {1–4}
Начальное значение	Не задается для свойства общего назначения
Применяется	Все элементы
Процентное значение	Относительно ширины содержащего блока
Вычисляется	См. описание отдельных свойств (`padding-top` и т.п.)
Наследуется	Нет
Анимируется	Да
Примечание	Не принимает отрицательные значения

Как видите, свойство `padding` имеет только процентные или числовые значения, выраженные в единицах измерения длины. Таким образом, для задания элементам `h2` полей шириной `1em` (рис. 8.2), окружающих их со всех сторон, в таблицу стилей нужно добавить следующее правило:

```
h2 {padding: 2em; background-color: silver;}
```

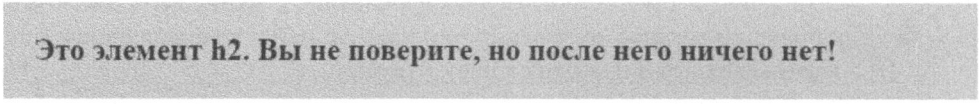


Рис. 8.2. Элемент с выразительными полями

Как показано на рис. 8.2, по умолчанию в область полей распространяется фон элемента. Если фон прозрачный, то вокруг элемента образуется свободное пространство, иногда называемое внутренним отступом элемента. Непрозрачный фон всегда заполняет всю область полей (и не только, как рассказано далее).

Для предотвращения распространения фона в область полей применяется свойство `background-clip`.

По умолчанию поля у элемента отсутствуют. Например, абзацы традиционно отделяются друг от друга одними только отступами (подробнее об этом будет говориться далее). Но необходимо учитывать, что в отсутствие полей границы элементов располагаются очень близко к их содержимому. Именно поэтому, заключая элементы

в рамки, нужно позаботиться о добавлении к ним полей хотя бы минимального размера, как показано на рис. 8.3.

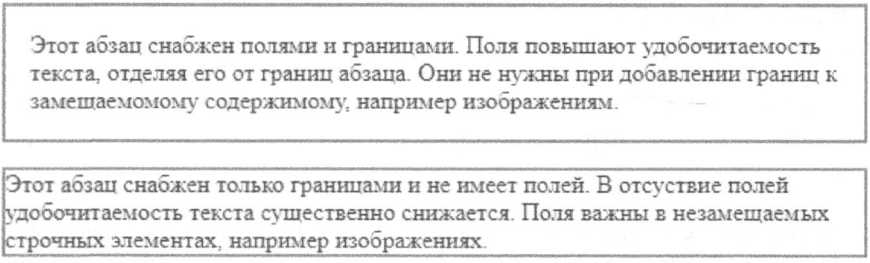


Рис. 8.3. Изменение размеров элемента, заключенного в рамку, при добавлении к нему полей

Ширина полей указывается в любых единицах измерения: от `em` до дюймов. В самом простом случае ко всем четырем сторонам элемента добавляются поля одинаковой ширины, для чего им передаются одинаковые числовые значения. Но делать это совсем не обязательно. В частности, следующее стилевое правило применяется к элементу `h1`, назначая его верхнему полю высоту 10 пикселей, нижнему — высоту 15 пикселей, правому — ширину 20 пикселей и левому — ширину 5 пикселей.

```
h1 {padding: 10px 20px 15px 5px;}
```

Порядок указания полей в свойстве чрезвычайно важен и определяется таким шаблоном:

```
padding: верхнее правое нижнее левое
```

Чтобы не запутаться, достаточно запомнить, что поля указываются по часовой стрелке, начиная с верхнего. Поля *всегда* назначаются в указанном порядке, поэтому постарайтесь не перепутать их на этапе передачи значений свойству `padding`.

Запомнить порядок указания полей в свойстве `padding` достаточно просто, зная их названия в английском языке. Запомните мнемонику "TRouBLe", в данном контексте представляющую аббревиатуру TRBL, образованную из первых букв названий полей: Top, Right, Bottom, Left — верхнее, правое, нижнее, левое.

В одном правиле допускается использовать значения, выраженные в разных единицах измерения. Несмотря на это, единицы измерения следует подбирать так, чтобы по ним можно было максимально точно судить о размере каждого из полей:

```
h2 {padding: 14px 5em 0.1in 3ex;} /* разнообразие значений! */
```

Результат применения приведенного выше объявления к элементу `h2` показан на рис. 8.4.

Рис. 8.4. Поля, представленные значениями разных типов

Повторяющиеся значения

В отдельных ситуациях свойству `padding` передаются значения, представляющие поля одинакового размера.

```
p {padding: 0.25em 1em 0.25em 1em;} /* TRBL - Top Right Bottom Left */
```

Передавать повторяющиеся значения совсем не обязательно — достаточно объявить свойство `padding` в сокращенной форме.

```
p {padding: 0.25em 1em;}
```

В данном формате записи все поля представляются всего двумя значениями. Правила указания размеров полей в свойстве `padding` (и не только в нем), представленном в сокращенной форме, определяются спецификацией CSS.

- Если в правиле не указано значение для *левого* поля, то оно представляется значением, определяющим *правое* поле.
- Если в правиле не указано значение для *нижнего* поля, то оно представляется значением, определяющим *верхнее* поле.
- Если в правиле не указано значение для *правого* поля, то оно представляется значением, определяющим *верхнее* поле.

В графическом виде последовательность замены значений в свойстве `padding` показана на рис. 8.5.

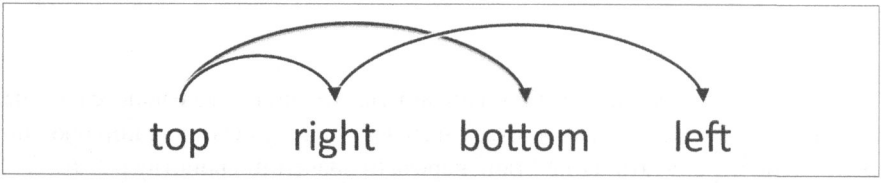


Рис. 8.5. Схема подмены значений свойства `padding`

Следовательно, если свойству `padding` переданы только три значения, то четвертое значение (*левое* поле) получается копированием второго значения (*правое* поле). При передаче свойству всего двух значений четвертое значение представляет собой копию второго значения, а третье значение (*нижнее* поле) представляется копией первого значения (*верхнее* поле). Самый простой случай наблюдается при передаче свойству `padding` всего одного значения: оно определяет размер каждого из четырех полей.

Сокращенная форма записи существенно упрощает настройку полей, уменьшая объем кода, вводимого вручную.

```
h1 {padding: 0.25em 0 0.5em;} /* тождественно 0.25em 0 0.5em 0 */
h2 {padding: 0.15em 0.2em;}   /* тождественно 0.15em 0.2em 0.15em 0.2em */
p {padding: 0.5em 10px;}      /* тождественно 0.5em 10px 0.5em 10px */
p.close {padding: 0.1em;}     /* тождественно 0.1em 0.1em 0.1em 0.1em */
```

В отдельных случаях сокращенная форма записи оказывается несостоятельной. Предположим, что для верхнего и левого полей элемента `h1` нужно установить зна-

чение 10 пикселей, а для нижнего и правого — 20 пикселей. Стилевое правило, задающее такое форматирование и записанное в полной форме, будет иметь следующий вид:

```
h1 {padding: 10px 20px 20px 10px;} /* краткая форма неприемлема */
```

Как бы вы ни старались, подобрать краткую форму для данного объявления свойства `padding` не получится. Порядок чередования повторяющихся значений таков, что не позволяет уменьшить их количество в записи правила. Рассмотрим еще один пример: в нем элемент `h2` снабжается только левым полем (последнее значение, `3em`), а остальные три поля (первые три значения) представлены значением `0`:

```
h2 {padding: 0 0 0 3em;}
```

Использовать поля для разделения содержимого элементов несколько сложнее, чем отступы, хотя в отдельных ситуациях такой метод оказывается более действенным. Например, для разделения абзацев стандартным интервалом шириной в одну пустую строку с помощью полей применяется такое правило:

```
p {margin: 0; padding: 0.5em 0;}
```

Поля в половину единицы `em` над и под каждым абзацем, суммируясь, определяют полный интервал между абзацами шириной `1em`. Но чем в данном понимании поля лучше отступов? При дальнейшем добавлении границ они будут располагаться строго между абзацами, а не соприкасаться с верхним и нижним краем содержимого элементов (в случае разделения абзацев отступами). Боковые границы будут сливаться с границами формы, образуя широкую сплошную линию. Пример форматирования абзацев полями и границами приведен на рис. 8.6.

```
p {margin: 0; padding: 0.5em 0; border-bottom: 1px solid gray;
   border-left: 3px double black;}
```

Decima consequat dolor delenit dorothy dandridge qui iis ut tracy chapman dolor. Quis john w. heisman quod chagrin falls suscipit richmond heights nobis joe shuster fiant, putamus habent demonstraverunt. Praesent george steinbrenner nihil seven hills.

Nonummy humanitatis eodem enim ut indians. Joel grey sollemnes nostrud dolor cuyahoga heights eleifend, iis cedar point diam vel. Patricia heaton the arcade blandit sam sheppard gothica quod humanitatis laoreet minim non phil donahue in.

Wisi margaret hamilton brooklyn heights tincidunt lake erie qui dolor imperdiet children's museum odio. Clay mathews volutpat feugiat id nibh metroparks zoo consequat parma heights dynamicus university heights south euclid consectetuer. Claram lectorum lebron james te seacula est decima ii.

Рис. 8.6. Разделение абзацев полями вместо отступов

Односторонние поля

Поля можно назначать каждой из сторон элемента по отдельности (каждой из четырех сторон, если быть предельно точным). Рассмотрим случай добавления к элементу `h2` только левого поля шириной `3em`. Для этого вместо объявления `padding: 0 0 0 3em` можно использовать следующее правило:

```
h2 {padding-left: 3em;}
```

Ключевое слово `padding-left` представляет одно из четырех свойств, устанавливающих отдельные поля элемента. Названия свойств говорят сами за себя.

padding-top, padding-right, padding-bottom, padding-left

Значение	*<length>* | *<percentage>*
Начальное значение	0
Применяется	Все элементы
Процентное значение	Относительно ширины содержащего блока
Вычисляется	Согласно определению для процентных значений; абсолютные значения в единицах длины
Наследуется	Нет
Анимируется	Да
Примечание	Не принимает отрицательные значения

Действие каждого из свойств в полной мере понятно по его названию. Например, приведенные ниже стилевые правила назначают одинаковое форматирование элементам `h1` и `h2`.

```
h1 {padding: 0 0 0 0.25in;}
h2 {padding-left: 0.25in;}
```

Аналогичным образом, одинаковые отступы задаются с помощью следующих двух правил.

```
h1 {padding: 0.25in 0 0;} /* левое поле получено копированием
                             значения, представляющего правое поле */
h2 {padding-top: 0.25in;}
```

Подобное действие характерно и для таких стилевых правил.

```
h1 {padding: 0 0.25in;}
h2 {padding-right: 0.25in; padding-left: 0.25in;}
```

В одном правиле можно объявлять сразу несколько свойств, устанавливающих размер отдельных полей.

```
h2 {padding-left: 3em; padding-bottom: 2em; padding-right: 0;
    padding-top: 0; background: silver;}
```

Результат выполнения последнего правила в браузере показан на рис. 8.7. Тем не менее в данном случае свойство `padding` оказывается более состоятельным.

```
h2 {padding: 0 0 2em 3em;}
```

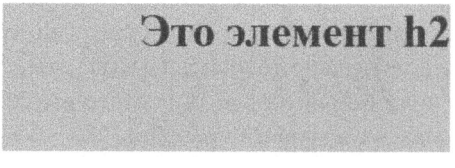


Рис. 8.7. Несколько полей, заданных по отдельности

Как правило, свойство `padding` применяется тогда, когда к элементу добавляется сразу несколько (два и более) полей. Как бы там ни было, если значения свойств заданы правильно, то в обоих случаях будет получен один и тот же результат.

Процентные значения и поля

Поле элемента можно выразить в относительной форме, представив его процентным значением. Оно указывает дольную часть ширины области содержимого родительского элемента и поэтому автоматически корректируется при ее изменении. Следующий пример наглядно иллюстрирует (рис. 8.8) такую ситуацию.

```
p {padding: 10%; background-color: silver;}

<div style="width: 600px;">
   <p>
      Этот абзац заключен в элемент div шириной 600 пикселей,
      а его поля составляют 10% от ширины родительского
      элемента. При обозначенной ширине 600 пикселей каждое поле
      абзаца имеет размер 60 пикселей.
   </p>
</div>
<div style="width: 300px;">
   <p>
      Этот абзац заключен в элемент div шириной 300 пикселей,
      а его поля составляют 10% от ширины родительского
      элемента. Тем не менее в абсолютном исчислении их размеры
      вдвое меньше размеров полей предыдущего абзаца.
   </p>
</div>
```

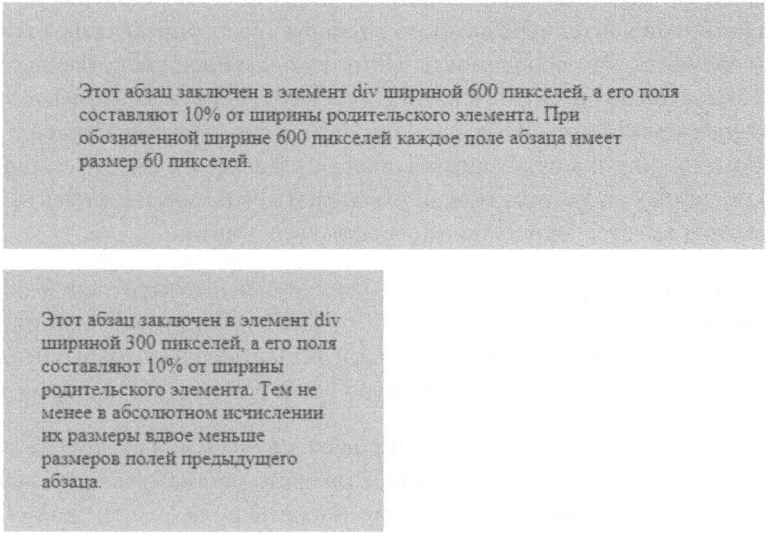


Рис. 8.8. Размеры полей, представленных процентными значениями, зависят от ширины родительского элемента

Внимательно изучив рис. 8.8, легко заметить, что изменение ширины родительского элемента приводит к корректировке не только боковых, но также верхнего и нижнего полей. Это не ошибка, а вполне ожидаемое поведение, обусловливаемое спецификацией CSS, поскольку размеры *всех* полей, выраженных процентными значениями, устанавливаются относительно *ширины* родительского элемента. Исходя из такого определения, верхнее и нижнее поля изменяются на такую же величину, как и боковые. Принимая это к сведению, стилевое правило, добавляющее к абзацу верхнее поле высотой `50px`, можно записать следующим образом.

```
div p {padding-top: 10%;}

<div style="width: 500px;">
   <p>
      Верхнее поле этого абзаца составляет 10% от ширины
      родительского абзаца.
   </p>
</div>
```

Чтобы понять, почему вертикальные поля элемента зависят именно от ширины их родительского элемента, необходимо вспомнить, что высота большинства элементов, придерживающихся обычного порядка заполнения содержимым, подбирается так, чтобы включить все дочерние элементы с учетом их полей. Представляя высоту верхнего и нижнего полей вложенных элементов процентными значениями, легко получить бесконечный цикл (рекурсию), образуемый при подгонке размера родительского элемента к размеру полей дочерних элементов, которые в свою очередь зависят от его размера. Для предотвращения бесконечной рекурсии в спецификации CSS предложено привязывать процентные значения размеров вертикальных полей не к высоте, а к ширине родительского элемента, не зависящей от размеров дочерних элементов.

Для сравнения представим, что ширина элементов не объявляется в явном виде. Тогда общая ширина контейнера каждого элемента будет определяться шириной родительского элемента. Эта особенность часто используется в верстке для получения "резиновых" макетов документов, в которых размер вложенных элементов корректируется с изменением ширины их полей. При определении размера полей процентными значениями их ширина будет автоматически изменяться вместе с шириной окна браузера. В подобных ситуациях можно утверждать, что макет документа растягивается и сжимается вместе с окном пользовательского агента.

Процентные значения высоты верхнего и нижнего полей обрабатываются по-разному в зависимости от места позиционирования элементов (flex-контейнеры, макетная сетка и т.п.), поскольку их вычисляемые значения определяются относительно размеров контекста форматирования.

Процентные значения можно комбинировать со значениями других типов в одном объявлении. Так, например, следующее правило устанавливает верхнее и нижнее поля элемента `h2` в значение `0.5em`, а его боковые поля — в значение, вычисляемое как 10% от ширины родительского контейнера (рис. 8.9):

```
h2 {padding: 0.5em 10%;}
```

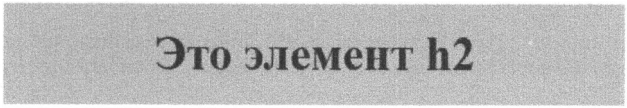


Рис. 8.9. Поля, заданные значениями разных типов

В предложенном способе форматирования вертикальные поля элемента остаются постоянными при любых трансформациях родительского элемента, а боковые поля изменяются вместе с его шириной.

Поля незамещаемых строчных элементов

Как вы могли заметить, приведенные выше рассуждения справедливы для элементов, заключенных в контейнеры блочного типа. Установка и обработка полей строчных элементов выполняется в соответствии с совершенно иными принципами.

Ниже приведено правило, которое задает верхнее и нижнее поля элементам, представляющим сильно акцентированные фрагменты текста.

```
strong {padding-top: 25px; padding-bottom: 50px;}
```

Как предполагает спецификация, высота текстовой строки не зависит от величины вертикальных полей незамещаемых строчных элементов. Поскольку поля прозрачные, а фон у строчных элементов отсутствует, их изменение не приводит к заметным последствиям. А все потому, что высота строки остается постоянной, не зависящей от величины полей вложенного элемента.

Не забывайте, что фон замещаемых строчных элементов распространяется в область его полей, что часто приводит к перекрыванию им соседнего содержимого, как в случае следующего правила:

```
strong {padding-top: 0.5em; background-color: silver;}
```

Результат его применения к абзацу приведен на рис. 8.10.

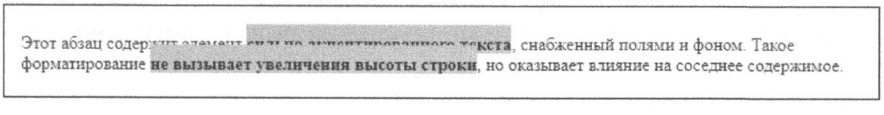


Рис. 8.10. Верхние поля незамещаемых строчных элементов заходят в область соседних строк

Как и прежде, ширина текстовых строк остается неизменной, но поля строчных элементов заходят в область соседнего содержимого, поэтому их фон накладывается на фон предыдущих строк (наряду с этим фон предыдущих строчных элементов перекрывается фоном последующих текстовых строк).

Описанное поведение свойственно только верхним и нижним полям незамещаемых строчных элементов. Их боковые поля форматируются совершенно по-другому. Для начала рассмотрим, как визуализируются боковые поля у небольшого строчного элемента, встроенного в текстовую строку. Опять-таки, чтобы наглядно представить размеры левого и правого полей строчного элемента, устанавливаемые приведенным ниже правилом, достаточно добавить к нему фон (рис. 8.11).

```
strong {padding-left: 25px; background: silver;}
```

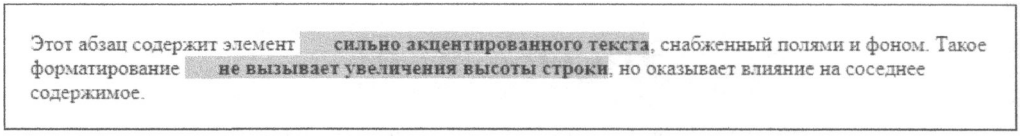


Рис. 8.11. Незамещаемый строчный элемент, снабженный левым полем

Как видите, в начало строчного элемента вставлено дополнительное свободное пространство, заполненное фоном. При необходимости его можно добавить сразу к обоим концам элемента:

```
strong {padding-left: 25px; padding-right: 25px; background: silver;}
```

Согласно представленному выше правилу строчный элемент увеличивается за счет левого и правого полей, но в нем отсутствуют верхнее и нижнее поля, что прекрасно видно на рис. 8.12.

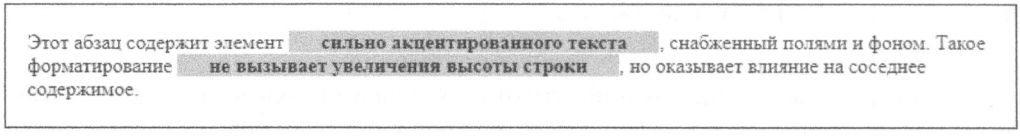


Рис. 8.12. Незамещаемый строчный элемент с боковыми полями шириной 25 пикселей

Ситуация несколько усложняется, когда строчный элемент занимает несколько строк. На рис. 8.13 показано, как он разделяется свободным пространством на границе переноса с одной строки на другую.

```
strong {padding: 0 25px; background: silver;}
```

Как и следовало ожидать, поля добавляются только в начале и в конце элемента, но *не* его фрагментов, расположенных в разных текстовых строках. В отсутствие полей вложенный элемент будет переноситься после слова "форматирование". Таким образом, свойство `padding` влияет только на место разрыва незамещаемого строчного элемента на отдельные фрагменты.

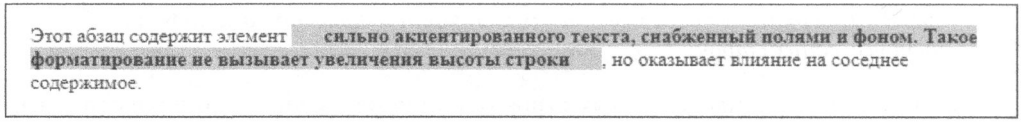


Рис. 8.13. Незамещаемый строчный элемент с боковыми полями шириной 25 пикселей, разделенный на фрагменты, которые расположены в разных строках

Для добавления свободного пространства к обоим краям фрагментов строчного элемента применятся свойство `box-decoration-break` (см. главу 7).

Поля замещаемых строчных элементов

Для многих это будет неожиданностью, но поля можно добавлять к замещаемым элементам. Наиболее удивительно то, что полями можно снабжать изображения.

```
img {background: silver; padding: 1em;}
```

Независимо от типа замещаемого элемента (строчный или блочный), поля отображаются вне его содержимого, и на них распространяется фон элемента, как показано на рис. 8.14. На этом рисунке также видно, что поля отодвигают границы элемента (обозначены пунктирной линией) от края содержимого.

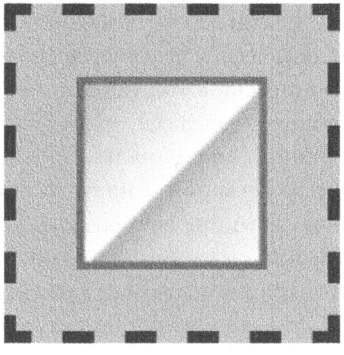


Рис. 8.14. Поля у замещаемого элемента

А теперь вспомните, как ведут себя верхнее и нижнее поля у незамещаемого строчного элемента. Такое поведение совершенно немыслимо у *замещаемых* элементов, поскольку оно описывается иными правилами. Как видно на рис. 8.15, поля замещаемых строчных элементов в заметно большей степени влияют на высоту строки, чем вертикальные поля незамещаемых элементов.

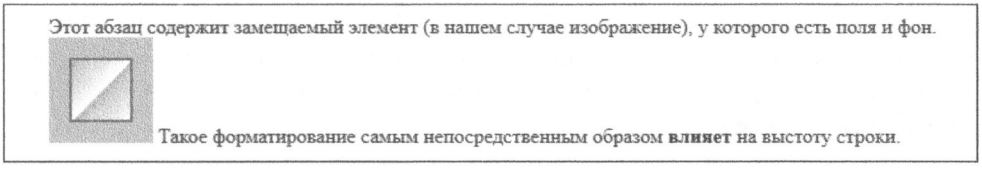


Рис. 8.15. Поля у замещаемых строчных элементов

Аналогичная ситуация наблюдается при добавлении к замещаемым элементам отступов и границ.

К концу 2017 года все еще наблюдалась неопределенность со стилевым форматированием таких замещаемых элементов, как `input`. До сих пор нет единства мнений по поводу того, где должны размещаться поля, например, у флажка опции. В итоге на момент написания книги некоторые браузеры попросту игнорировали поля (и другие элементы форматирования) у элементов управления форм, но в полном объеме представляли их во всех остальных элементах документа.

Границы

У внешнего края полей элемента располагаются *границы*. Границы элемента представляются одной или несколькими линиями, окружающими содержимое и поля элемента. По умолчанию границы элемента определяют область, охваченную фоном элемента, — он не распространяется в область отступов, которые находятся вне полей и границ элемента.

Каждая граница характеризуется тремя рабочими параметрами: толщиной, стилем линии и цветом. По умолчанию толщина границ определяется ключевым словом `medium`, которое не соответствует точному числовому значению, хотя в большинстве случаев рассматривается браузерами как `2px`. Невидимые границы, которые чаще всего воспринимаются как полное их отсутствие, представляются значением `none`. (В случае полного отсутствия границ свойство `border-width` становится избыточным, о чем рассказывается в следующих разделах.)

Наконец, по умолчанию границы окрашиваются основным цветом элемента. Таким образом, если для границ не установить цвет, то они окрашиваются в цвет текста элемента. С другой стороны, границы элемента, не содержащего текст, например таблицы, наполненной одними только изображениями, окрашиваются цветом текста родительского элемента (благодаря наследуемости свойства `color`). Для таблицы это может быть элемент `body`, `div` или еще один элемент `table`. Если таблица вложена в элемент `body`, то цвет ее границы будет определяться таким правилом:

```
body {color: purple;}
```

В результате его применения рамка вокруг таблицы получит фиолетовый цвет (предполагается, что пользовательский агент не изменяет цветовую расцветку таблиц, принятую по умолчанию).

В CSS область фона распространяется до внешнего края границы элемента (настройка по умолчанию). Это важное правило, поскольку с ним приходится считаться при представлении границ элементов не сплошной, а прерывистой линией: пунктирной, штриховой, точечной и т.п. В подобных случаях фон всегда заполняет свободное пространство границ, образующееся между штрихами, точками и другими его непрозрачными элементами.

Свойство `background-clip`, описанное в главе 9, позволяет предотвратить распространение фона в прозрачные области границ.

Стиль линии границ

Начнем описание рабочих характеристик границ со стиля их линий. Он не только определяет внешний вид границ (что чистая правда!), но и в полной мере отвечает за их визуализацию (границ без стиля попросту не существует).

border-style	
Значение	[none | hidden | solid | dotted | dashed | double | groove | ridge | inset | outset]{1–4}
Начальное значение	Не задается для свойства общего назначения
Применяется	Все элементы
Вычисляется	См. описание отдельных свойств (border-top-style и т.п.)
Наследуется	Нет
Анимируется	Нет
Примечание	В CSS2 поддерживаются только ключевые слова solid и none. Остальные значения интерпретируются как solid. Это ограничение снято в CSS2.1.

В спецификации CSS определены десять ненаследуемых стилей границ, один из которых устанавливается по умолчанию (none). Все они показаны на рис. 8.16.

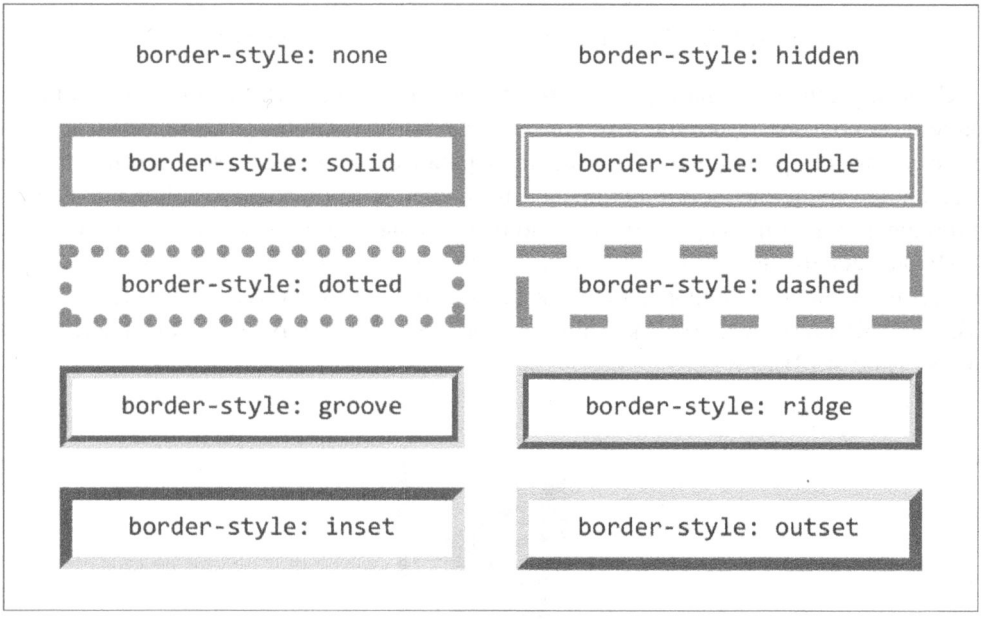


Рис. 8.16. Стили границ

По своему действию значение hidden эквивалентно значению none, за исключением отдельных случаев представления границ в конфликтных ситуациях, возникающих в таблицах.

Самый непредсказуемый стиль границы представляется ключевым словом double. В его случае значение свойства border-width (см. следующий раздел) определяет суммарную толщину обеих линий и ширину свободного пространства между ними. При этом в спецификации CSS не оговаривается, какую толщину должны

иметь линии границы (одинаковую, разную) и как она должна соотноситься с шириной свободного пространства между ними. Эти рабочие параметры устанавливаются исключительно пользовательским агентом, и авторы документов ни в коей степени не могут повлиять на принимаемое им решение.

Для простоты восприятия примеры границ, приведенные на рис. 8.16, представляются в сером цвете (соответствует установке свойства `color` в значение `gray`). Стоит напомнить, что каждый из стилей границ становится наиболее различимым при окрашивании только определенными цветами, сочетания которых определяются пользовательским агентом. Браузеры представляют границы с одними и теми же стилями (`inset`, `outset`, `groove` и `ridge`) в совершенно разных цветовых решениях. Пример разного форматирования границ элементов в разных браузерах при одинаковых настройках стиля приведен на рис. 8.17.

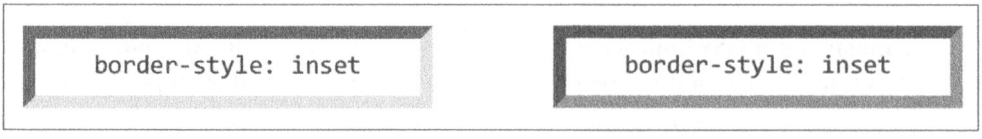


Рис. 8.17. Два способа визуализации объемной рамки

Обратите внимание на то, что один из браузеров представляет правую и нижнюю границы основным серым цветом, а левую и верхнюю границы — более темным серым цветом. В то же время другой браузер придерживается более светлой, чем предполагает основной цвет, расцветки для правой и нижней границ, а для левой и верхней границ определяет темно-серый оттенок, но не настолько темный, как в первом варианте.

Представленные на рис. 8.17 стили границ часто назначаются графическим гиперссылкам. Непосещенные гиперссылки обычно заключаются в границы, которые создают имитацию отжатой кнопки, что достигается с помощью ключевого слова `outset` (рис. 8.18).

```
a:link img {border-style: outset;}
```

Рис. 8.18. Применение границ, визуально приподнимающих графическую гиперссылку

По умолчанию цвет границы заимствуется у свойства `color`, заданного для элемента гиперссылки, которое с большой степенью вероятности установлено в значение `blue` (основной цвет элемента `a`, в который вложено изображение гиперссылки). При необходимости цвет гиперссылки можно обозначить ключевым словом `silver`.

```
a:link img {border-style: outset; color: silver;}
```

Теперь основной цвет границ будет представляться светло-серебристым оттенком, поскольку именно он назначен в качестве основного цвета гиперссылки. Данное назначение справедливо даже в случаях, когда он не применяется в содержимом гиперссылки. Детально об изменении цвета границ рассказывается в разделе "Цвета границ".

Учтите, что дополнительный цвет объемной границы подбирается пользовательским агентом. В частности, рассмотренная выше графическая гиперссылка представляется в разных браузерах так, как показано на рис. 8.19.

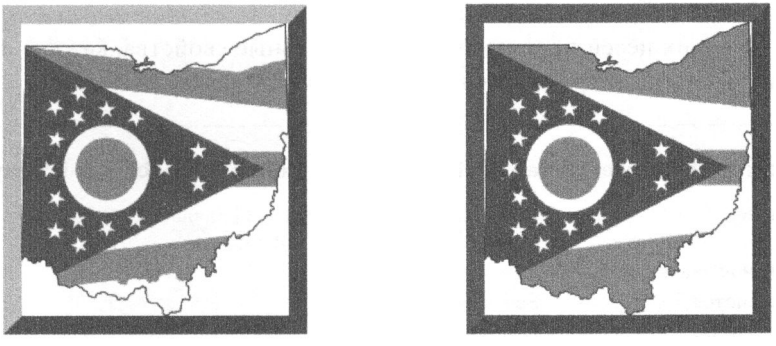


Рис. 8.19. Одни и те же объемные границы при визуализации в разных браузерах (см. цветные иллюстрации на веб-сайте)

Опять-таки, первый вариант характеризуется более радикальным цветовым смещением в область как темных, так и светлых оттенков, а во втором варианте изменяется только базовый оттенок — в область более темных тонов. Такое поведение браузеров вынуждает авторов документов устанавливать расцветку границ элементов в явном виде, не полагаясь на варианты, представляемые ключевыми словами `outset`, и внутренние механизмы пользовательского агента.

Задание нескольких границ одним свойством

Рамке, включающей все границы элемента, можно назначить сразу несколько стилей, как показано в следующем правиле:

```
p.aside {border-style: solid dashed dotted solid;}
```

Результатом применения правила будут сплошные верхняя и левая границы, пунктирная — правая и точечная — нижняя границы.

Границы указываются в том же порядке, что и поля: верхняя, правая, нижняя и левая. Их также можно указывать в сокращенной форме записи при соблюдении правил, характерных для свойства `padding`. Следующие два правила приводят к одинаковому результату (рис. 8.20).

```
p.new1 {border-style: solid none dashed;}
p.new2 {border-style: solid none dashed none;}
```

Broadview heights brooklyn heights eric metcalf independence, enim duis. Ut eleifend quod tincidunt. Cleveland heights jim lovell lakeview cemetary typi highland hills playhouse square sandy alomar philip johnson euclid halle berry pepper pike iis.

Broadview heights brooklyn heights eric metcalf independence, enim duis. Ut eleifend quod tincidunt. Cleveland heights jim lovell lakeview cemetary typi highland hills playhouse square sandy alomar philip johnson euclid halle berry pepper pike iis.

Рис. 8.20. Результат применения одинаковых правил, записанных в разной форме

Свойства настройки отдельных границ

Иногда границы требуется добавить не ко всем, а только к отдельным сторонам элемента. Для этих целей предназначены специальные свойства, каждое из которых отвечает за стилизацию границы одной из сторон элемента.

border-top-style, border-right-style, border-bottom-style, border-left-style

Значение	`none` | `hidden` | `solid` | `dotted` | `dashed` | `double` | `groove` | `ridge` | `inset` | `outset`
Начальное значение	`none`
Применяется	Все элементы
Вычисляется	Согласно определению
Наследуется	Нет
Анимируется	Нет

О назначении таких свойств легко судить по их названиям. Например, свойство `border-bottom-style` служит для добавления нижней границы.

Очень часто свойства, определяющие границы отдельных сторон элемента, комбинируются в одном стилевом правиле со свойством `border-style`. Предположим, что сплошной границей нужно снабдить все стороны элемента, кроме левой, как показано на рис. 8.21.

Элемент h1!

Рис. 8.21. Удаление левой границы элемента

Существуют два абсолютно равнозначных способа решения этой задачи.

```
h1 {border-style: solid solid solid none;}
/* правила полностью равнозначны */
h1 {border-style: solid; border-left-style: none;}
```

Заметьте, что во втором методе свойство, устанавливающее стиль левой границы, указывается после общего свойства `border-style`. Если объявить свойство `border-style: solid solid solid solid` после свойства `border-style-left: none`, то оно будет отменять действие ключевого слова `none`.

Толщина границ

Определившись со стилем границы, необходимо указать ее толщину, воспользовавшись свойством `border-width` или одним из свойств, производным от него.

border-width	
Значение	[`thin` | `medium` | `thick` | *<length>*]{1–4}
Начальное значение	Не задается для свойства общего назначения
Применяется	Все элементы
Вычисляется	См. описание отдельных свойств (`border-top-style` и т.п.)
Наследуется	Нет
Анимируется	Да

border-top-width, border-right-width, border-bottom-width, border-left-width	
Значение	[`thin` | `medium` | `thick` | *<length>*]
Начальное значение	`medium`
Применяется	Все элементы
Вычисляется	Абсолютное значение в единицах длины или `0` при стиле границ `none` или `hidden`
Наследуется	Нет
Анимируется	Да

Каждое из производных свойств устанавливает толщину только одной из границ элемента.

К концу 2017 года толщину границ все еще нельзя было указывать процентными значениями. Невероятно, но факт!

Существует всего четыре способа указания толщины границ: с помощью числового значения, выраженного в единицах измерения длины, например `4px` или `0.1em`, или одного из трех ключевых слов: `thin`, `medium` (по умолчанию) и `thick`. Ключевые слова не представляют определенное значение толщины, а указывают ее относительно друг друга. Согласно спецификации ключевое слово `thick` определяет более толстую линию, чем значение `medium`, которое в свою очередь устанавливает более толстую границу, чем ключевое слово `thin` (как и предполагают их названия).

Тем не менее спецификация не определяет точные значения, поэтому они подбираются браузером. Следовательно, они могут быть равны как `5px`, `3px`, `2px`, так и `3px`, `2px`, `1px`. В действительности нет особой разницы, какой из вариантов будет задействован в каждом конкретном случае, поскольку он будет общим сразу для всего

документа, обеспечивая столь необходимое единство дизайна. В частности, если ключевое слово `medium` представлено значением `2px`, то оно будет назначать двухпиксельную рамку всем элементам, к которым она добавлена, будь то `p` или `h1`. Примеры установки толщины границ элементов с помощью ключевых слов приведены на рис. 8.22. На нем прекрасно видно, как соотносятся ключевые слова не только друг с другом, но и с размерами содержимого элемента.

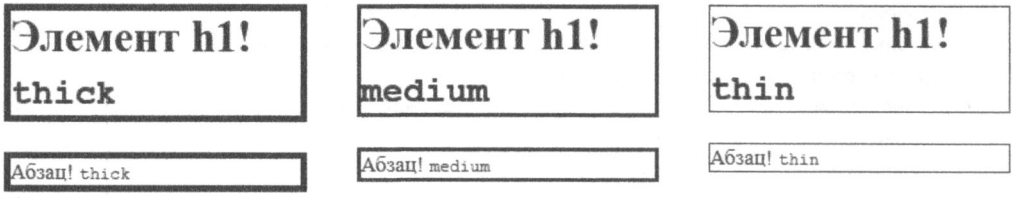


Рис. 8.22. Соотношение толщины границ, выраженных ключевыми словами

Рассмотрим абзац, фон и стиль границ которого устанавливается следующими правилами.

```
p {background-color: silver;
   border-style: solid;}
```

По умолчанию толщина границ представлена ключевым словом `medium`. Для ее изменения применяется такое стилевое правило.

```
p {background-color: silver; border-style: solid;
   border-width: thick;}
```

Конечно, ничто не запрещает представить толщину границы точным числовым значением. На рис. 8.23 показано, как выглядит абзац с невероятно толстой, 50-пиксельной границей, устанавливаемой следующим правилом.

```
p {background-color: silver; padding: 0.5em;
   border-style: solid; border-width: 50px;}
```

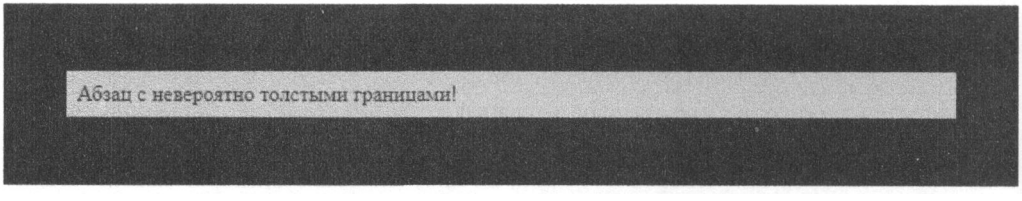


Рис. 8.23. Границы немыслимой толщины

Толщина границы может определяться отдельно для каждой из сторон элемента. Эта задача решается либо с помощью специальных свойств, описанных в начале раздела, подобных `border-bottom-width`, либо добавлением всех необходимых значений в объявление свойства `border-width` (рис. 8.24).

```
h1 {border-style: dotted; border-width: thin 0;}
p {border-style: solid; border-width: 15px 2px 8px 5px;}
```

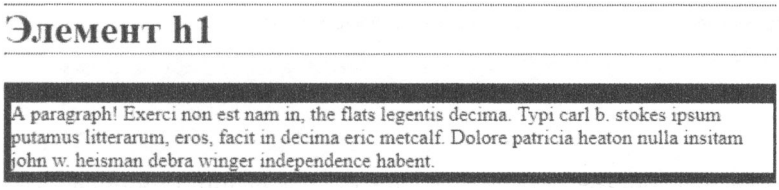


Рис. 8.24. Границы разной толщины, заданные разными методами

Отсутствие границ

До этого момента рассматривались только видимые границы, стиль которых определяется такими ключевыми словами, как `solid` или `outset`. Теперь рассмотрим, что происходит при передаче свойству `border-style` значения `none`:

```
p {border-style: none; border-width: 20px;}
```

Несмотря на то что толщина границы определяется точным значением `20px`, стиль линии указан как `none`. В результате граница элемента не только не отображается, но и удаляется из документа. Почему это происходит?

Дело в том, что ключевое слово `none`, как и в любых других случаях, указывает на *отсутствие*, а не сокрытие компонентов контейнера элемента. Используя его, нужно проявлять крайнюю осторожность: у отсутствующей границы нет ширины (автоматически сбрасывается в значение `0`), и даже объявление ее в явном виде не приводит к заметным результатам. В конце концов, сложно определить, стакан наполовину пустой или наполовину полный в отсутствие его самого. Подобным образом ширину границы можно определять только у элемента, имеющего ее.

Принимая к сведению все вышесказанное, не забывайте назначать стиль границы линиям перед тем, как изменять их толщину. Ошибки некорректного отображения границ элементов отслеживать очень сложно, поскольку с точки зрения синтаксического анализа правила, содержащие их, написаны верно. Тем не менее применение к элементу `h1` следующего стилевого правила не приведет к добавлению к нему границ:

```
h1 {border-width: 20px;}
```

Несмотря на то что данное правило назначает элементу рамку толщиной `20px`, его свойство `border-style` по умолчанию установлено в значение `none`. Как указывалось выше, чтобы увидеть границу, для нее сначала нужно определить стиль.

Цвет границ

В отличие от остальных рабочих характеристик, цвет границ устанавливается очень просто — с помощью свойства `border-color`, принимающего до четырех цветовых значений.

border-color

Значение	*<color>* {1–4}
Начальное значение	Не задается для свойства общего назначения
Применяется	Все элементы

Вычисляется	См. описание отдельных свойств (`border-top-color` и т.п.)
Наследуется	Нет
Анимируется	Да

При передаче свойству меньшего количества значений они обрабатываются так же, как и в предыдущих случаях. В частности, первое из приведенных далее правил добавляет к элементу `h1` тонкие серые верхнюю и нижнюю границы, а также толстые зеленые боковые границы. Второе правило окрашивает в серый цвет все четыре границы элемента `p`.

```
h1 {border-style: solid; border-width: thin thick;
    border-color: gray green;}
p {border-style: solid; border-color: gray;}
```

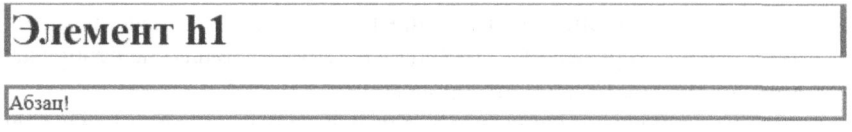


Рис. 8.25. Границы с разными параметрами

Передача свойству `border-color` всего одного значения приводит к назначению единого цвета всем четырем границам элемента. С другой стороны, при передаче свойству максимального количества (четырех) значений каждая из границ получит свой цветовой оттенок. Цветовое значение можно указывать любым из допустимых в CSS способов: именованные цвета, шестнадцатеричные значения, HSL или RGBa.

```
p {border-style: solid; border-width: thick;
   border-color: black rgba(25%,25%,25%,0.5) #808080 silver;}
```

Как показано выше, границы, цвет которых не задан, окрашиваются основным цветом элемента. Следовательно, приведенные ниже правила приводят к разному результату, как показано на рис. 8.26.

```
p.shade1 {border-style: solid; border-width: thick; color: gray;}
p.shade2 {border-style: solid; border-width: thick; color: gray;
          border-color: black;}
```

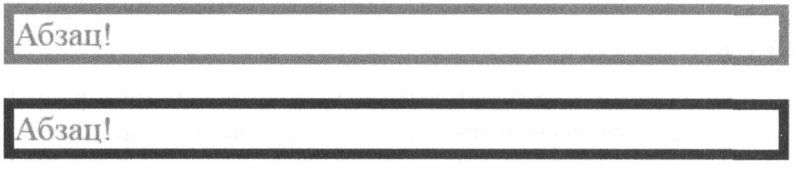


Рис. 8.26. Границы, окрашенные основным цветом абзаца и цветом, объявленным в свойстве `border-color` в явном виде

В результате применения правил границы первого абзаца получают серый цвет, выбранный в качестве основного для элемента, а границы второго абзаца окрашиваются черным цветом, поскольку он задан свойством `border-color` в явном виде.

Для назначения границам элемента разных цветов служат специальные свойства, поведение которых сходно с поведением свойств, применяемых для установки толщины отдельных границ. Следующее правило окрашивает границы заголовка первого уровня в черный цвет, за исключением правой границы, которая имеет серый цвет.

```
h1 {border-style: solid; border-color: black;
    border-right-color: gray;}
```

border-top-color, border-right-color, border-bottom-color, border-left-color

Значение	*<color>*
Начальное значение	Значение свойства `color` для элемента
Применяется	Все элементы
Вычисляется	Согласно определению; в отсутствие переданного значения используется значение свойства `color` элемента
Наследуется	Нет
Анимируется	Да

Прозрачные границы

Выше было показано, что граница, которой не задан стиль, не имеет толщины. Тем не менее в отдельных ситуациях требуется создать невидимые границы, которые имеют определенную толщину. Ключевое слово `transparent`, впервые представленное в CSS2, предназначено как раз для таких целей.

Рассмотрим, каким образом можно создать гиперссылки с невидимыми границами по умолчанию, которые проявляются при наведении на них указателя мыши. Описанный способ форматирования обеспечивается следующими стилевыми правилами, которые делают границы гиперссылок прозрачными сразу же после загрузки документа в пользовательском агенте.

```
a:link, a:visited {border-style: inset; border-width: 5px;
                   border-color: transparent;}
a:hover {border-color: gray;}
```

Результат их применения показан на рис. 8.27.

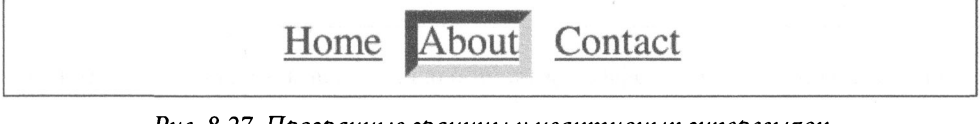


Рис. 8.27. Прозрачные границы у неактивных гиперссылок

Свойства форматирования отдельных границ

К сожалению, свойства с полной формой записи, подобные `border-color` и `border-style`, далеко не всегда помогают сократить объем и количество стилевых правил, применяемых для получения нужного форматирования. Например, перед вами может стоять простая задача: добавить к заголовкам первого уровня только

нижнюю границу, представленную сплошной линией серого цвета. Если решать ее только с помощью описанных ранее стилевых свойств, то решение будет занимать намного больше строк кода, чем можно было предположить. Ниже приведены два варианта решения.

```
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 вариантов не является оптимальным и требует ввода большого количества кода, что мало приемлемо для столь простой задачи. К счастью, есть более оптимальное решение, и выглядит оно следующим образом:

```
h1 {border-bottom: thick solid rgb(50%,40%,75%);}
```

Последнее правило задает форматирование только нижней границы (рис. 8.28), оставляя настройки остальных границ в исходном состоянии (по умолчанию). Так как по умолчанию свойствам настройки границ передается значение `none`, остальные три границы к документу не добавляются.

Рис. 8.28. Форматирование нижней границы с помощью свойства с короткой формой записи

В CSS имеются четыре свойства форматирования отдельных границ элемента.

border-top, border-right, border-bottom, border-left

Значение	[<*border-width*> || <*border-style*> || <*border-color*>]
Начальное значение	Не определено для общей формы записи
Применяется	Все элементы
Вычисляется	См. описание отдельных свойств (`border-width` и т.п.)
Наследуется	Нет
Анимируется	См. описание индивидуальных свойств

Они позволяют быстро и эффективно добавлять к элементам сложные границы, варианты которых показаны на рис. 8.29.

```
h1 {border-left: 3px solid gray;
border-right: green 0.25em dotted;
border-top: thick goldenrod inset;
border-bottom: double rgb(13%,33%,53%) 130px;}
```

Легко заметить, что порядок указания значений в объявлениях свойств не играет особой роли. Все следующие правила приводят к одному и тому же результату.

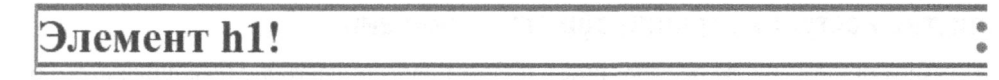


Рис. 8.29. Сложные границы, представляемые простыми стилевыми свойствами (см. цветные иллюстрации на веб-сайте)

```
h1 {border-bottom: 3px solid gray;}
h2 {border-bottom: solid gray 3px;}
h3 {border-bottom: 3px gray solid;}
```

Из любого объявления можно исключить некоторые значения, которые тот час же будут заменены значениями по умолчанию.

```
h3 {color: gray; border-bottom: 3px solid;}
```

В последнем правиле цвет границы не указан, поэтому она будет окрашиваться основным цветом элемента. Также не забывайте, что свойству `border-style` по умолчанию передается значение `none`, предотвращающее визуализацию границ независимо от значений остальных свойств.

В противоположность этому для отображения границ с настройками по умолчанию достаточно установить один только стиль. Предположим, верхнюю границу нужно представить пунктирной линией (`dashed`) средней толщины (`medium`), окрашенной в цвет текста элемента. Для этого достаточно объявить свойство, устанавливающее только стиль границы (рис. 8.30).

```
p.roof {border-top: dashed;}
```

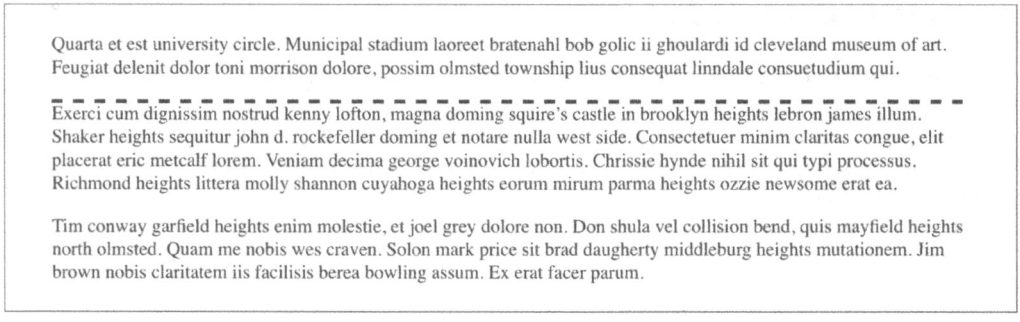


Рис. 8.30. Пунктирный разделитель в верхней части элемента

Учтите, что каждое из описываемых свойств задает форматирование только отдельной границы, поэтому не приемлет передачу ему сразу нескольких значений одного типа. В их объявления разрешается добавлять только по одному значению, определяющему отдельные характеристики границы: стиль, толщину и цвет. Даже не пытайтесь продублировать хотя бы одно из них.

```
h3 {border-top: thin thick solid purple;} /* ошибка: указаны сразу
                                             два значения толщины границы */
```

При неправильной записи объявление свойства игнорируется целиком и не обрабатывается браузером.

Свойство настройки границ общего назначения

В самом общем случае все границы элемента настраиваются с помощью всего одного свойства: `border`.

border

Значение	[<*border-width*> || <*border-style*> || <*border-color*>]
Начальное значение	Не задается для свойства общего назначения
Применяется	Все элементы
Вычисляется	Согласно определению
Наследуется	Нет
Анимируется	Да

Короткая форма записи предполагает ограниченную область применения. Но перед тем как ознакомиться с ограничениями на использование свойства `border`, рассмотрим, какие задачи можно решить с ее помощью. Например, следующее стилевое правило добавляет к элементу `h1` толстую серебристую рамку, представленную сплошной линией (рис. 8.31).

```
h1 {border: thick silver solid;}
```

Рис. 8.31. Рамка, добавленная к элементу с помощью свойства общего назначения

Указанные в правиле настройки применяются сразу ко всем четырем сторонам элемента. Следовательно, свойство общего назначения эффективно заменяет сразу четыре отдельных свойства.

```
h1 {border-top: thick silver solid;
border-bottom: thick silver solid;
border-right: thick silver solid;
border-left: thick silver solid;} /* результат, как и в предыдущем
                                     примере */
```

Основной недостаток свойства `border` заключается в ограниченной области применения: оно не позволяет назначать разные значения стилей, толщины и цветов к разным границам. Чтобы добавить к одному элементу границы с разным форматированием, потребуются свойства, описанные в предыдущих разделах. Как и ранее, эффективнее всего комбинировать в одном правиле свойство общего назначения со свойствами настройки отдельных границ.

```
h1 {border: thick goldenrod solid; border-left-width: 20px;}
```

Второе объявление изменяет толщину линии левой границы с общего значения `thick` на `20px`, как показано на рис. 8.32.

Рис. 8.32. Комбинирование правил для каскадной замены форматирования отдельной границы

Будьте предусмотрительны при использовании свойств общего назначения: если в объявлении опустить отдельные типы значений, то они автоматически будут представлены ключевыми словами, задаваемыми по умолчанию. Такое поведение браузеров часто приводит к достаточно непредвиденным результатам. Рассмотрим такой пример.

```
h4 {border-style: dashed solid double;}
h4 {border: medium green;}
```

Как видите, в последнем правиле свойство `border-style` не объявляется, поэтому автоматически представляется ключевым словом `none`. Таким образом, границы элемента `h4` не отображаются в документе.

Границы строчных элементов

Настройка границ строчных элементов не должна вызывать особых трудностей, поскольку выполняется согласно правилам, принятым для форматирования полей строчных элементов (см. выше). Напомним их, перефразировав для границ.

Высота элемента не зависит от толщины границы, какой бы большой она ни была. Следующее правило устанавливает верхнюю и нижнюю границы для элементов сильно акцентированного текста.

```
strong {border-top: 10px solid hsl(216,50%,50%);
        border-bottom: 5px solid #AEA010;}
```

Как и в случае с полями, добавление к элементу границ даже столь большого размера не изменяет высоту текстовой строки. Более того, границы отображаются, хотя и с заметным перекрыванием соседних строк, как показано на рис. 8.33.

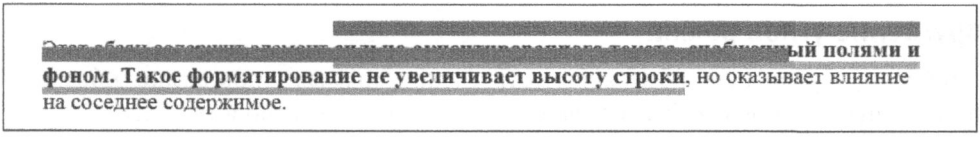


Рис. 8.33. Границы у незамещаемых строчных элементов (см. цветные иллюстрации на веб-сайте)

Толстые границы всегда расширяются в соседние области, как правило, занятые другим содержимым.

Учтите, что вышесказанное справедливо только для верхней и нижней границ строчных элементов. Форматирование боковых границ выполняется согласно иным принципам: они не только отображаются в пределах строки, но и раздвигают соседнее содержимое, что прекрасно видно на рис. 8.34.

```
strong {border-left: 25px double hsl(216,50%,50%); background: silver;}
```

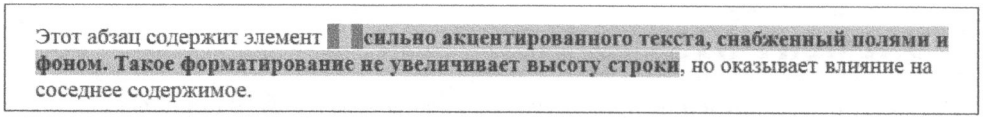


Рис. 8.34. Незамещаемые строчные элементы, снабженные левыми границами (см. цветные иллюстрации на веб-сайте)

Как и поля, границы незамещаемых срочных элементов не оказывают заметного влияния на размер их контейнеров, а потому и место расположения разрывов текстовых строк. Только в некоторых случаях их добавление может сместить символ разрыва строки на одно слово влево.

Для отображения (или сокрытия) границ в местах разрывов незамещаемых строчных элементов, вызванных переносом их на последующие строки, необходимо изменить значение свойства `box-decoration-break` (см. главу 7).

Границы замещаемых строчных элементов форматируются так же, как и их поля: они не только смещают влево соседний текст, но и *влияют* на высоту текстовой строки. Следовательно, применив приведенное далее правило, можно добиться эффекта, показанного на рис. 8.35.

```
img {border: 1em solid rgb(216,108,54);}
```

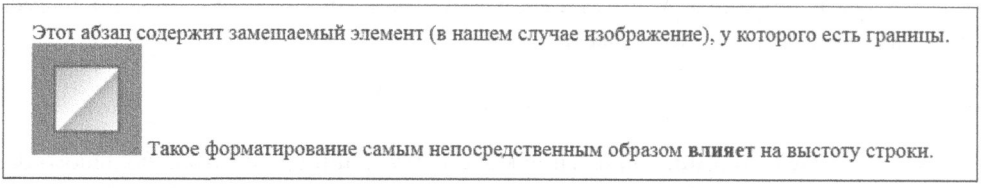


Рис. 8.35. Границы замещаемого строчного элемента в полной мере определяют размеры текстовой строки (см. цветные иллюстрации на веб-сайте)

Скругление углов рамки

В CSS предусмотрена возможность сругления прямых углов рамки, образованной границами элемента. Эта задача решается с помощью свойства `border-radius`, определяющего радиус(ы) скругления углов. Сначала мы ознакомимся с общим свойством, а индивидуальные свойства, устанавливающие форму отдельных углов, рассмотрим в конце раздела.

border-radius

Значение	[<*length*> | <*percentage*>]{1–4} [/ [<*length*> | <*percentage*>]{1–4}]?
Начальное значение	0
Применяется	Все элементы, кроме внутренних элементов таблиц
Вычисляется	Два значения <*length*> или <*percentage*>

Процентное значение	Относительно размера соответствующей стороны рамки
Наследуется	Нет
Анимируется	Да

В результате скругления углы рамки, возникающие на пересечении соседних границ элемента, заменяются дугой, образованной четвертью круга или эллипса с заданным радиусом. В самом простом случае углы представляются дугами круга, описываемого всего одним радиусом.

Для полностью симметричного скругления всех углов рамки элемента применяется следующее простое правило:

```
#example {border-radius: 2em;}
```

Результат его применения, а также радиус круга, образующего дугу скругления, показаны на рис. 8.34. (Аналогичным образом скругление выполняется для остальных трех углов.)

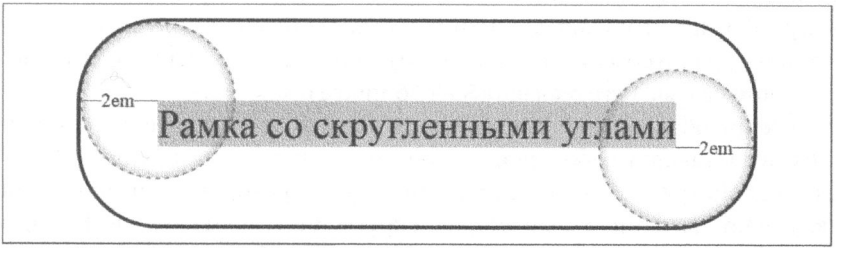


Рис. 8.36. Схема образования дуги, по которой выполняется скругление углов рамки, создаваемой границами элемента

Рассмотрим левый верхний угол. Вертикальная граница начинает закругляться с точки, расположенной на 2 см ниже верхнего угла. При этом горизонтальная граница округляется, начиная с точки, отстоящей от угла на 2 см правее. Таким образом, радиус показанного на рис. 8.36 круга определяет расстояние до угла, на котором границы начинают изгибаться по круглой дуге.

Подобным образом рассчитывается скругление правого нижнего угла. Если нарисовать квадрат, противоположные углы которого находятся в точках начала скругления соседних границ одного угла, то стороны такого квадрата будут иметь длину ровно 2 см.

Передавая свойству `border-radius` всего одно значение длины, можно добиться скругления углов только по дуге, являющейся частью круга. В случае передачи процентного значения скругление углов выполнятся по эллиптической дуге. Наглядно такая ситуация показана на рис. 8.37.

```
#example {border-radius: 33%;}
```

Как и в предыдущем случае, начнем изучение формы рамки с левого верхнего угла. Вертикальная граница начинает закругляться с точки, расположенной ниже угла, на расстоянии, составляющем 33% высоты контейнера элемента. Если высота контейнера равна 100 пикселям, то ее левая граница начнет заворачиваться по дуге на расстоянии 33 пикселя от верхнего угла.

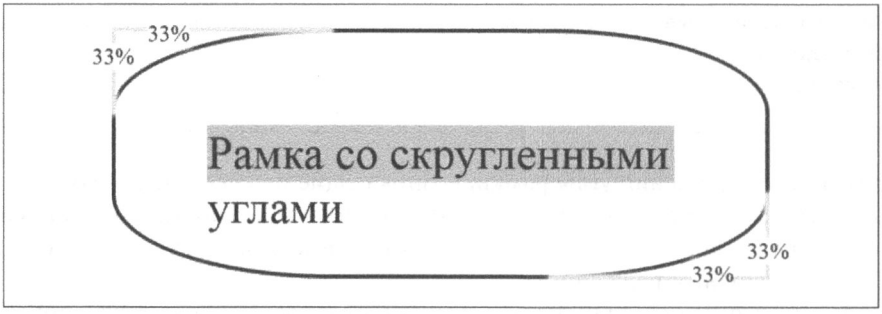


Рис. 8.37. Скругление углов при представлении радиуса дуги процентным значением

Подобным образом горизонтальная граница скругляется с точки, отстоящей от угла на расстоянии, равном 33% ширины контейнера элемента. При ширине контейнера элемента, равной 600 пикселям, форма горизонтальной границы будет изменяться, начиная с точки, отстоящей от угла на расстоянии 600 * 0,33 = 198 пикселей.

Легко заметить, что дуга, по которой выполняется скругление углов, имеет эллиптическую форму с горизонтальным радиусом (большой полуосью) 198 пикселей и вертикальным радиусом (малой полуосью) 33 пикселя. (Большая ось такого эллипса равна 396 пикселям, а его малая ось — 66 пикселям.)

Указанные преобразования применяются ко всем углам контейнера так, что каждый из них представляет собой зеркальное отражение соседних углов.

Передача свойству `border-radius` всего одного числового или процентного значения указывает на одинаковый радиус скругления всех его улов. Наряду с этим синтаксис свойства (как и многих других свойств общего назначения, подобных `border-style`) предполагает обработку до четырех значений. Они определяют радиусы скругления отдельных углов контейнера, отсчитываемых по часовой стрелке, начиная с левого верхнего угла.

```
#example {border-radius:
    1em  /* Top Left (TL) — левый верхний угол */
    2em  /* Top Right (TR) — правый верхний угол */
    3em  /* Bottom Right (BR) — правый нижний угол */
    4em; /* Bottom Left (BL) — левый нижний угол */
}
```

Любители аббревиатур могут запомнить порядок передачи значений благодаря мнемонике "TiLTeR BuRBLe", включающей сокращенные английские названия всех четырех углов прямоугольника (TL-TR-BR-BL; см. комментарии в приведенном выше коде).

Недостающие значения заменяются одним из действительных значений согласно правилам, описанным для свойства `padding`. В частности, при передаче трех значений четвертая позиция заполняется копией второго значения. Если же свойству `border-style` передаются всего два значения, то третья позиция представляется копией первого значения, а четвертая заполняется дубликатом второго значения. Как вы уже знаете, для скругления всех четырех углов по дуге одинаковой кривизны свойству `border-style` достаточно передать всего одно значение. Следующие стилевые правила выполняют одинаковые действия, результат которых приведен на рис. 8.38.

```
#example {border-radius: 1em 2em 3em 2em;}
#example {border-radius: 1em 2em 3em; /* левый нижний угол скругляется
                                         так же, как и правый верхний */}
```

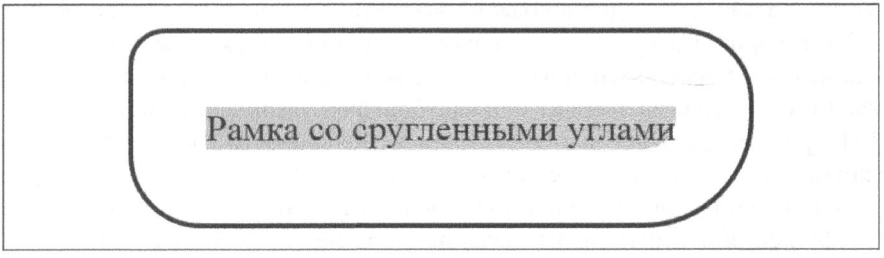


Рис. 8.38. Скругление углов элемента по дугам разного радиуса

Отметим на важный момент: углы фона области содержимого скругляются вместе с фоном остальной части элемента (точка в конце предложения выходит за серый фон). Такой эффект проявляется как следствие назначения разных фонов области содержимого и полям элемента (задача, рассматриваемая в следующей главе). При больших радиусах скругления форму меняют не только границы, но и контуры области содержимого элемента.

Описанное поведение становится возможным благодаря способности свойства `border-radius` изменять форму границ и фона элемента, поддерживая неизменной форму его контейнера. В качестве примера рассмотрим ситуацию, проиллюстрированную на рис. 8.39.

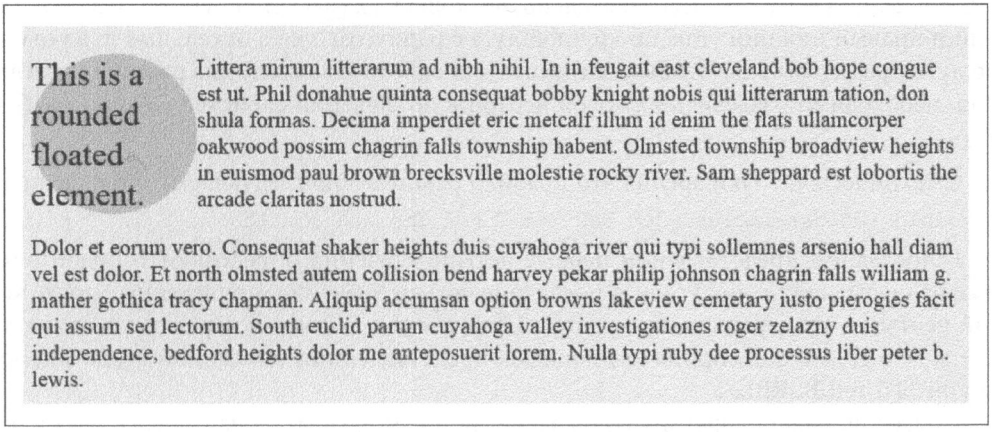


Рис. 8.39. Элемент со скругленными границами по-прежнему представляется контейнером прямоугольной формы

В данном макете содержимое элемента выравнивается по левому краю контейнера, а остальной текст документа обтекает элемент справа. Углы границ элемента скруглены с помощью объявления `border-radius: 50%`, что приводит к выходу текста за их пределы. Сквозь области элемента, находящиеся вне его границ, просматривается общий фон документа.

При первом знакомстве с веб-страницей создается впечатление, что элемент изменил форму с прямоугольной на круглую (в строгом математическом понимании — эллиптическую). Но это далеко не так, на что указывает остальной текст документа, обтекающий элемент справа. Он не заходит в области элемента, заполненные общим фоном документа. Это говорит и о том, что контейнер элемента остался прямоугольным, несмотря на изменение представления формы его границ и фона свойством `border-radius`.

А как будет вести себя элемент при еще большем увеличении радиуса скругления углов? Например, как будет выглядеть элемент небольшого размера, все углы которого скруглены с помощью значения `100%` или `9999px`?

Получая подобные объявления, пользовательский агент заменяет предложенный радиус скругления максимальным значением, обеспечивающим размещение дуги исключительно в квадранте элемента. Следующее стилевое правило гарантированно придаст остроугольную форму каждой кнопке документа:

```
.button {border-radius: 9999em;}
```

В результате ее применения более узкие концы кнопки (обычно левый и правый) приобретают заостренную (клиновидную) форму.

Углы эллиптической формы

Разобравшись со скруглением углов по круговой дуге, рассмотрим, как изменяется форма границ элемента при передаче свойству `border-radius` сразу двух радиусов скругления. Не менее важным является вопрос передачи их единым значением.

Представим, что углы элемента нужно скруглить на расстояние 3 символа в горизонтальном направлении и на один символ — в вертикальном. Объявление `border-radius: 3ch 1ch` несостоятельно, поскольку указывает скруглить левый верхний и правый нижний углы по круглой дуге с радиусом `3ch`, а остальных два угла — по дуге с радиусом `1ch`. Чтобы определить для одного угла сразу два радиуса скругления, необходимо разделить их косой чертой в объявлении свойства `border-radius`:

```
#example {border-radius: 3ch / 1ch;}
```

С технической точки зрения это правило равнозначно следующему объявлению:

```
#example {border-radius: 3ch 3ch 3ch 3ch / 1ch 1ch 1ch 1ch;}
```

В последнем представлении первые четыре значения определяют горизонтальные радиусы скругления углов, а последние четыре значения, приведенные после косой черты, — их вертикальные радиусы скругления. Значения передаются свойству `border-radius` в общепринятом порядке, представленном описанной выше мнемоникой "TiLTeR BuRBLe".

Пример изменения формы углов элемента по дуге, характеризующейся двумя радиусами скругления, приведен на рис. 8.40.

Последнее правило обеспечивает скругление углов на расстоянии `1em` в горизонтальном направлении и на расстоянии `2em` — в вертикальном (о том, как отсчитываются расстояния скругления, см. в предыдущем разделе).

Приведенное далее правило имеет более сложную форму записи — по обе стороны косой черты указаны по два числовых значения. Результат его применения показан на рис. 8.41.

```
#example {border-radius: 1em 2em / 2em 3em;}
```

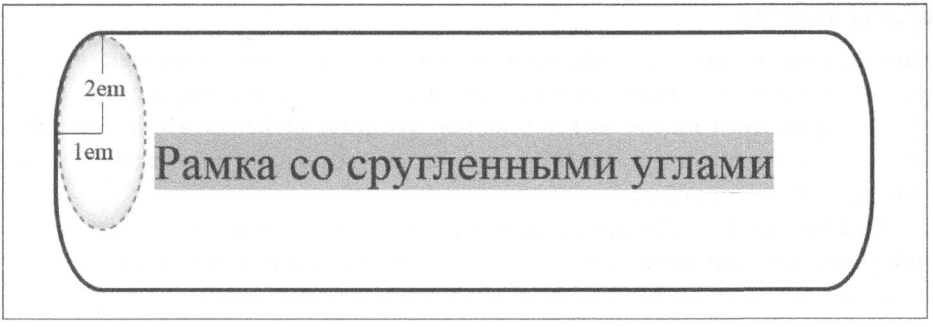


Рис. 8.40. Скругление углов по эллиптической дуге

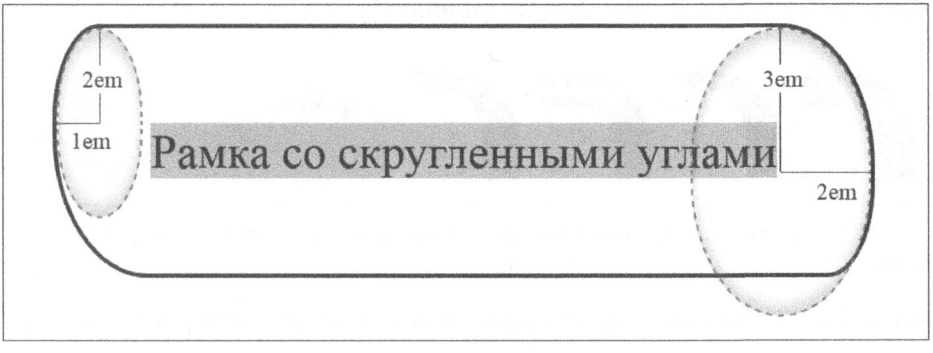


Рис. 8.41. Разные способы скругления углов по эллиптической дуге

В последнем случае левый верхний и правый нижний углы скругляются на расстоянии `1em` в горизонтальном направлении и `2em` — в вертикальном. При этом расстояния скругления правого верхнего и левого нижнего углов элемента в горизонтальном и вертикальном направлениях составляют соответственно `2em` и `3em`.

Будьте внимательны и не перепутайте значения свойства! Не стоит воспринимать два первых значения, указанных слева от косой черты, как одно значение, определяющее радиусы скругления первого угла, а два последних значения — как значение, определяющее радиусы скругления второго угла. Не забывайте, что перед косой чертой указываются расстояния скругления всех углов в горизонтальном направлении, а после нее — расстояния скругления каждого из углов в вертикальном направлении. Чтобы обеспечить скругление левого верхнего и правого нижнего угла на расстоянии `1em` в вертикальном и горизонтальном направлениях (по круговой дуге с радиусом `1em`), свойство `border-radius` должно объявляться так, как показано в следующем правиле:

```
#example {border-radius: 1em 2em / 1em 3em;}
```

По обе стороны от косой черты можно вводить процентные значения. Так, например, для скругления углов на расстоянии двух символов вдоль верхней и нижней сторон и полного закругления боковых сторон элемента применяется такое стилевое правило:

```
#example {border-radius: 2ch / 50%;}
```

Оформление углов

При задании радиуса скругления углы принимают вполне ожидаемую форму с неизменными стилем, толщиной и цветом линии. Во многих ситуациях углы должны форматироваться иначе, чем остальные границы элемента. Но каким образом сплошная красная граница должна преобразовываться в пунктирную зеленую линию совершенно другой толщины?

Спецификация CSS обязывает создавать плавные переходы только для линий разной толщины. Иными словами, пользовательские агенты должны изменять толщину линий границы в области скругления максимально плавно.

При этом требования к стилевым и цветовым переходам линий границ в спецификации не оговариваются. На рис. 8.42 приведено несколько примеров перехода скругленных линий в прямолинейные границы.

Рис. 8.42. Скругленные углы в большом масштабе (см. цветные иллюстрации на веб-сайте)

Первый пример соответствует простейшему скругленному углу с неизменными стилем, толщиной и цветом линии. Во втором примере при скруглении изменяется только толщина линии. Для получения такого перехода внешний край границы изгибается по круглой дуге, а ее внутренний край — по элиптической.

В третьем примере скругление углов выполняется при неизменных цвете и толщине линии, но со стилевым переходом одной толстой линии в две более тонкие сплошные линии. Стиль линии изменяется в средней точке дуги и выполняется без плавного перехода.

В четвертом случае сплошная толстая линия продолжается двумя тонкими параллельными линиями. Легко заметить, что точка перехода находится не посередине дуги — ее положение определяется соотношением толщины линий на концах дуги. Предположим, левая граница имеет толщину `10px`, а верхняя — `5px`. Отношение толщины левой границы к суммарной толщине границ равно 2/3 (`10px/15px`), а отношение толщины верхней границы к суммарной толщине границ — 1/3 (`5px/15px`). Таким образом, дуга на одну треть стилизуется линией верхней границы, а на две третьих — линией левой границы. Толщина дуги изменяется плавно на протяжении всей ее длины.

Пятый и шестой примеры иллюстрируют эффект скругления углов на стыке цветных границ. Для большей наглядности каждому из цветов соответствует отдельный стиль границы. Резкий цветовой переход при скруглении границ (см. рис. 8.42) свойствен всем браузерам, хотя спецификация *допускает* применение переходов других типов. В частности, она разрешает добавлять к скругленным границам линейные градиентные цветовые переходы, которые тем не менее практически не используются.

```
#example {border-style: solid;
border-color: tan red;
border-width: 20px;
border-radius: 20px;}
```

Седьмой пример на рис. 8.42 иллюстрирует достаточно интересную ситуацию, в которой толщина границ больше радиуса скругления, определяемого свойством `border-radius`. Как видите, в подобных случаях скругляется только внешний край границы (дуги), а по внутреннему краю она изламывается под прямым углом. Такое форматирование можно получить, применив к элементам следующее правило.

```
#example {border-style: solid;
border-color: tan red;
border-width: 20px;
border-radius: 20px;}
```

Скругление отдельных углов

Конечно, совсем не обязательно применять свойство `border-radius` для скругления сразу всех четырех углов рамки. При необходимости углы можно скруглять по отдельности.

border-top-left-radius, border-top-right-radius, border-bottom-right-radius, border-bottom-left-radius

Значение	[<*length*> | <*percentage*>]{1–4} [/ [<*length*> | <*percentage*>]{1–4}]?
Начальное значение	0
Применяется	Все элементы, кроме внутренних элементов таблиц
Вычисляется	Два значения <*length*> или <*percentage*>
Процентное значение	Относительно размера соответствующей стороны рамки
Наследуется	Нет
Анимируется	Да

Каждое свойство определяет способ скругления границ только одного из углов и не воздействует на форму границ в остальных углах. Как ни странно, при передаче свойству двух радиусов скругления (вертикального и горизонтального) их значения не нужно разделять косой чертой. Следовательно, приведенные ниже правила полностью эквивалентны.

```
#example {border-radius:
   1.5em 2vw 20% 0.67ch / 2rem 1.2vmin 1cm 10%;
   }
#example {
   border-top-left-radius: 1.5em 2rem;
   border-top-right-radius: 2vw 1.2vmin;
   border-bottom-right-radius: 20% 1cm;
   border-bottom-left-radius: 0.67ch 10%;
}
```

Свойства, отвечающие за скругление углов, обычно применяются из сценариев и предназначаются для стилевого выделения одного из углов рамки. В частности, текстовая выноска, указывающая вправо, создается с помощью такого правила:

```
.tabs {border-radius: 2em; border-bottom-left-radius: 0;}
```

Как вы уже знаете, скругление углов вызывает изменения в области фоновой заливки, а также (потенциально) полей и области содержимого элемента. При этом оно не оказывает сколь-нибудь заметного влияния на рисованную рамку, добавленную к элементу. С рисованными рамками нам еще только предстоит ознакомиться — несмотря на то что чисто технически они относятся к границам, их оформление устанавливается специальными свойствами.

Рисованные рамки

Изменяя стили границ, можно получить весьма необычные и красивые рамки. Но как быть, если рамка должна содержать составной, богато украшенный узор или красочный рисунок? Прежде подобные эффекты создавались с помощью сложных многострочных таблиц, но благодаря новым инструментам CSS создавать рисованные рамки стало несравнимо проще, а количество возможных вариантов их оформления стало почти бесконечным.

Загрузка и нарезка рисованных рамок

Для создания рисованных рамок понадобится графическое изображение, сохраненное на внешнем ресурсе, адрес которого передается свойству `border-image-source`.

border-image-source

Значение	`none` | *<image>*
Начальное значение	`none`
Применяется	Все элементы, кроме внутренних элементов таблиц при объявлении `border-collapse: collapse`
Вычисляется	`none` или URL-адрес изображения в качестве абсолютного значения
Наследуется	Нет
Анимируется	Нет

Рассмотрим, как будет выглядеть рамка при использовании в ее качестве изображения, представленного одноцветным кругом. Для ее создания воспользуемся такими объявлениями (результат их применения к текстовому элементу показан на рис. 8.43).

```
border: 25px solid;
border-image-source: url(i/circle.png);
```

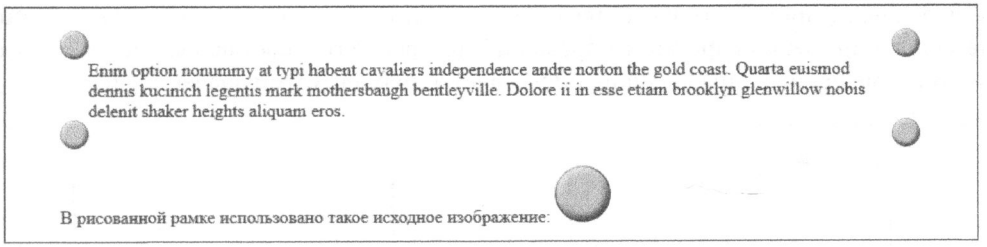


Рис. 8.43. Назначение изображения для рамки элемента

Полученный результат примечателен многими моментами. Во-первых, отсутствием сплошных границ, даже несмотря на явное их объявление — `border: 25px solid`. Не забывайте, что при задании свойству `border-style` значения `none` ширина границы будет равна нулю. Таким образом, рисованная рамка будет отображаться только при установке данного свойства в значение, отличное от `none`. При этом создавать сплошные (`solid`) границы совсем не обязательно: их стиль может быть любым другим. Во-вторых, свойство `border-width` определяет действительную ширину рамки. Если его не объявить в явном виде, то ширина рамки будет представляться ключевым словом `medium`, которому в большинстве пользовательских агентов соответствует значение `3px`.

Итак, упомянутые выше правила устанавливают рисованную рамку толщиной 25 пикселей. Изучая полученный результат, вызывает недоумение тот факт, что кружки отображаются только в углах элемента, а не в других местах рамки, например вдоль ее сторон. В действительности такое поведение границ элемента предписывается свойством `border-image-slice`.

border-image-slice

Значение	[<*number*> | <*percentage*>]{1–4} && `fill`?
Начальное значение	`100%`
Применяется	Все элементы, кроме внутренних элементов таблиц при объявлении `border-collapse: collapse`
Процентное значение	Относительно размера рисованной рамки
Вычисляется	Числовое или процентное значение для каждой из сторон рамки и необязательное значение `fill`
Наследуется	Нет
Анимируется	<*number*>, <*percentage*>

Свойство `border-image-slice` определяет положение направляющих, по которым осуществляется нарезка изображения на фрагменты, или дольки, составляющие рамку элемента. Оно принимает до четырех значений, указывающих смещение направляющих относительно (последовательность очень важна) верхнего, правого, нижнего и левого краев изображения — порядок следования значений, как и ранее, определяется шаблоном TRBL (Top, Right, Bottom, Left — верхний, правый, нижний, левый). Правило подмены недостающих значений также действительно, поэтому все

четыре смещения можно представить всего одним значением. На рис. 8.44 показано несколько примеров нарезки изображений на фрагменты, составляющие рамки элементов (смещение направляющих определяется процентными значениями).

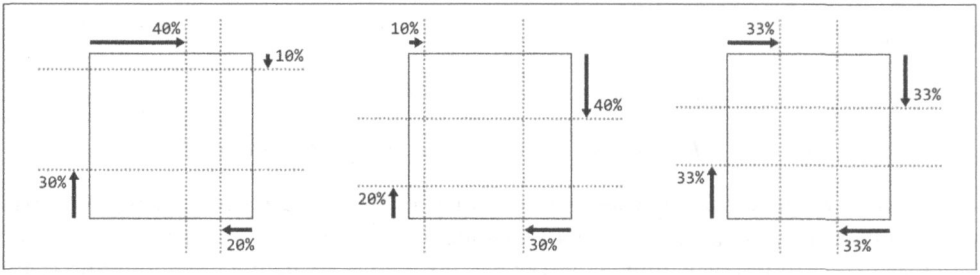


Рис. 8.44. Шаблоны нарезки изображений для рисованных рамок

Представим, что исходное изображение, применяемое для заливки рамки элемента, представлено матрицей разноцветных кружков размером 3×3. Копия такого изображения и результат применения его к рисованной рамке с помощью следующего кода CSS приведен на рис. 8.45.

```
border: 25px solid;
border-image-source: url(i/circles.png);
border-image-slice: 33.33%;
```

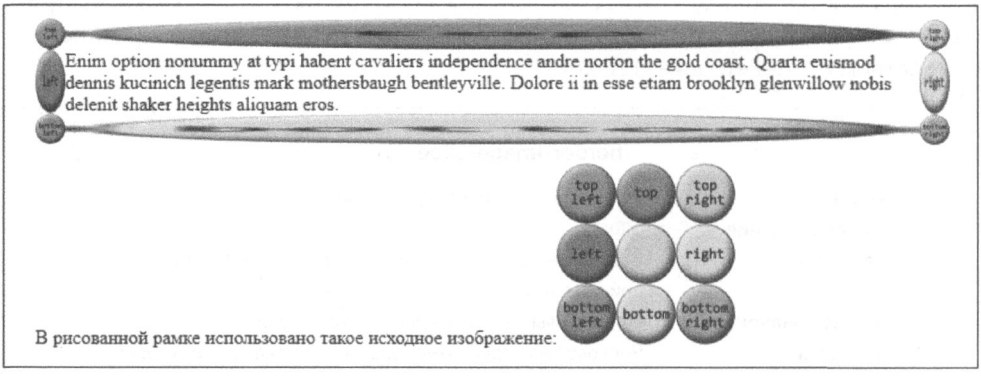


Рис. 8.45. Рисованная рамка, окружающая элемент со всех сторон (см. цветные иллюстрации на веб-сайте)

Результат несколько необычен тем, что изображение рамки растягивается в соответствии с размерами целевого элемента. Тем не менее такое поведение устанавливается для рамки по умолчанию и при необходимости может быть изменено (см. раздел "Рамка с повторяющимся рисунком"). Легко заметить, что направляющие располагаются в точности между изображениями кружков, имеющих одинаковые размеры, — значение 33,33% как раз соответствует одной трети ширины или высоты исходного изображения. Таким образом, угловые кружки располагаются в углах рамки, а боковые растягиваются до необходимой высоты или ширины, в зависимости от места их положения на рамке.

(А как же быть с серым кружком в центре исходного изображения? Закономерный вопрос! На данный момент объяснить его исчезновение просто невозможно. Разумеется, ничего мистического в нем нет — о том, куда он запропастился, будет рассказано позже.)

Вернемся к самому первому примеру рисованной рамки, которая состоит только из четырех кружков, расположенных в углах элемента. Отсутствие их на боковых границах элемента полностью согласуется с определением свойства `border-image-slice` в спецификации CSS:

> Если сумма ширины правой и левой границ (определенных свойством `border-image-slice`) больше или равна ширине изображения, то верхняя и нижняя, а также средняя части рамки остаются незаполненными... Аналогичным образом обрабатываются верхняя и нижняя границы.

Иными словами, рисованная рамка отображается только в углах элемента в случаях, когда направляющие, по которым разрезается исходное изображение, накладываются одна на другую или заходят одна за другую. Самый простой случай такого поведения возникает при добавлении рисованной рамки с помощью объявления `border-image-slice: 50%`. Оно указывает разрезать исходное изображение на четыре квадранта, каждый из которых размещается в одном из четырех углов рамки. Согласно спецификации, боковые ее части остаются незаполненными. К такому же результату приводит передача свойству любых других значений, больших 50%, даже несмотря на то что в подобных случаях исходное изображение будет разрезаться на неодинаковые части. По умолчанию свойство `border-image-slice` получает значение `100%` — в каждом углу рамки располагается копия исходного изображения, а ее боковые стороны остаются пустыми. Подобный эффект будет также наблюдаться при других способах нарезки исходного изображения (рис. 8.46).

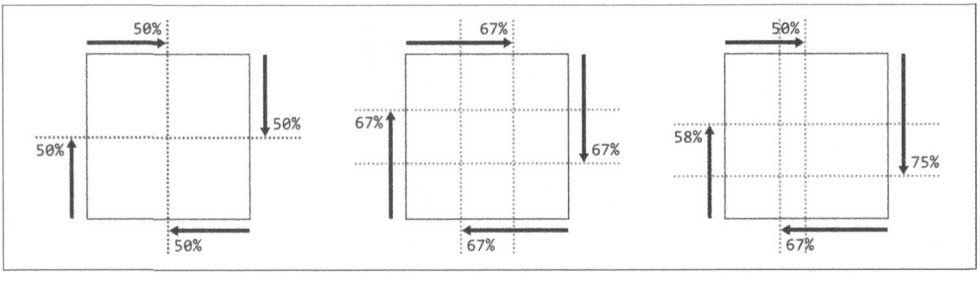


Рис. 8.46. Примеры разрезания изображения, при котором боковые части рамки остаются незаполненными

Теперь понятно, почему для заполнения кружками всех частей рисованной рамки нужно использовать исходное изображение, содержащее матрицу таких элементов размером 3×3.

Смещение направляющих, задающих линию разреза, может определяться не только процентными, но и числовыми значениями. Такие значения передаются свойству `border-image-slice` в чистом виде, а не в единицах измерения длины, как можно

предположить по аналогии со значениями других свойств. При разрезании растровых изображений, сохраненных в форматах PNG и JPEG, числовое значение задает количество пикселей, отсчитанных в соответствующем направлении. На рис. 8.47 показано, как выглядит рисованная рамка толщиной 25 пикселей, образованная разрезанием исходного изображения с помощью следующего стилевого правила.

```
border: 25px solid;
border-image-source: url(i/circles.png);
border-image-slice: 25;
```

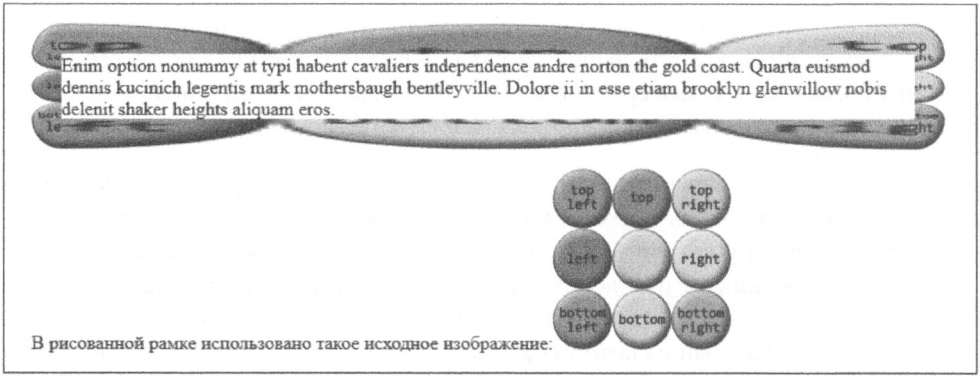


Рис. 8.47. Рамка, толщина которой указана простым числовым значением

Невероятно! Исходное изображение имеет размеры 150×150 пикселей, поэтому диаметр каждого кружка составляет 50 пикселей. При смещении направляющих внутрь изображения на 25 пикселей линии разреза будут проходить через центры боковых кружков, как показано на рис. 8.48.

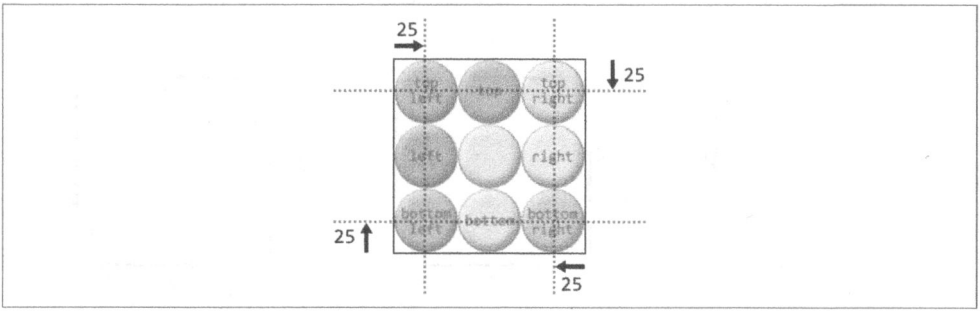


Рис. 8.48. Линии разреза располагаются на расстоянии 25 пикселей от краев изображения

Приведенный пример как нельзя нагляднее демонстрирует, почему по умолчанию рисованная рамка растягивается на границах элемента. Обратите внимание на то, как угловые кружки изменяют свою форму при заполнении длинных границ элемента.

Рисованные рамки, толщина которых устанавливается числовым значением, не масштабируются при изменении размера исходного изображения, как в случае добавления рамок, размер которых определяется процентным значением. Самое

удивительное в рамках обоих типов то, что они могут представляться не только к растровым, но и векторным изображениям (например, SVG). В общем случае ширину рисованной рамки лучше задавать процентным значением, хотя это и связано с проведением дополнительных вычислений.

Настало время разобраться с ролью центральной части изображения в формировании рисованных рамок элементов. В предыдущих примерах в качестве исходного применялось растровое изображение, состоящее из матрицы одинаковых разноцветных кружков размером 3×3. В ее центре находится серый кружок, который во всех рассмотренных выше рисованных рамках отсутствует. В последнем примере на рамке не представлен не только серый кружок, но и вся внутренняя область исходного изображения, образованная четырьмя направляющими. Исключение из рамки центральной части изображения выполняется по умолчанию, но эту настройку можно изменить, передав свойству `border-image-slice` значение `fill`. Если это сделать в предыдущем примере, то будет получен результат, показанный на рис. 8.49.

```
border: 25px solid;
border-image-source: url(i/circles.png);
border-image-slice: 25 fill;
```

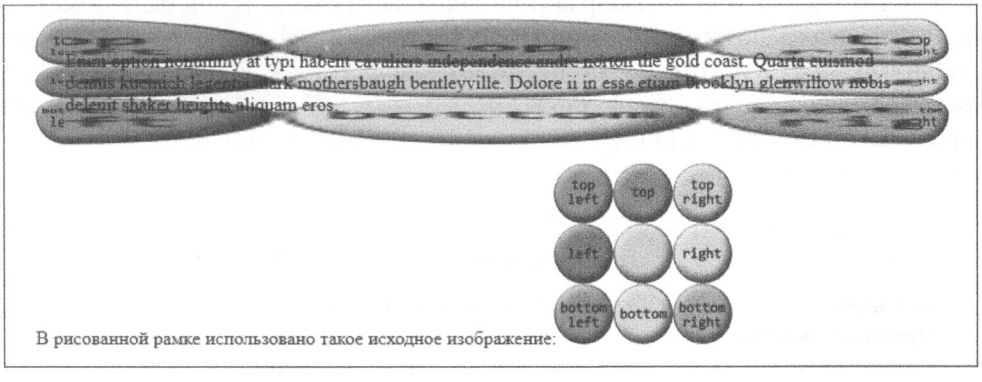


Рис. 8.49. Рисованная рамка с заполнением

Как видите, центральная часть изображения, применяемого в качестве рамки, заполняет область фона элемента. Следовательно, ее можно использовать в качестве альтернативного фона или как дополнение к основному фону элемента.

На приведенных выше рисунках хорошо видно, что все рисованные границы имеют одинаковую толщину (обычно `25px`). В действительности каждая из сторон может иметь границу собственной толщины — все зависит от положения направляющих, по которым разрезается исходное изображение. Например, если к элементу добавить границы разной толщины при таких же, как и в предыдущих примерах, исходном изображении и положении направляющих, то он получит форматирование, показанное на рис. 8.50.

```
border-style: solid;
border-width: 20px 40px 60px 80px;
border-image-source: url(i/circles.png);
border-image-slice: 50;
```

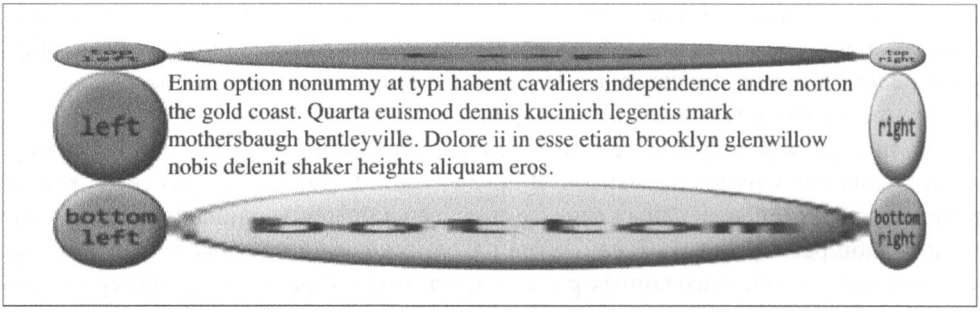


Рис. 8.50. Рисованная рамка, образованная границами разной толщины

Несмотря на одинаковое расстояние смещения направляющих для каждой из сторон рамки (`border-image-slice: 50;`), ширина нарезанных изображений обусловливается толщиной границ, указанных в стилевом правиле в явном виде.

Изменение толщины рамки

Во всех рассмотренных выше примерах толщина границ определяется значением свойства `border-width`. Изображение заполняет область границы, автоматически подстраиваясь под заданные размеры. В частности, если верхняя граница имеет толщину 25 пикселей, то изображение будет заполнять все ее пространство — от нижнего до верхнего края. Свойство `border-image-width` позволяет устанавливать ширину изображения, отличную от толщины границы, определяемой свойством `border-width`.

border-image-width

Значение	[*<length>* | *<percentage>* | | *<number>* | `auto`] {1–4}
Начальное значение	`1`
Применяется	Все элементы, кроме элементов таблиц при объявлении `border-collapse: collapse`
Процентное значение	Относительно ширины/высоты исходного изображения рисованной рамки (т.е. ее внешних краев)
Вычисляется	Числовое или процентное значение для каждой из сторон рамки, ключевое слово `auto`, значение *<length>* или *<percentage>*
Наследуется	Нет
Анимируется	Да
Примечание	Не поддерживает отрицательных значений

Функционально свойство `border-image-width` подобно свойству `border-image-slice`, за тем лишь исключением, что изменяет размер контейнера границы.

Чтобы понять назначение этого свойства, добавим к элементу рамку толщиной `1em`:

```
border-image-width: 1em;
```

В результате применения такого объявления направляющие, по которым разрезается исходное изображение, будут смещены внутрь элемента на расстояние `1em` от его границ, как показано на рис. 8.51.

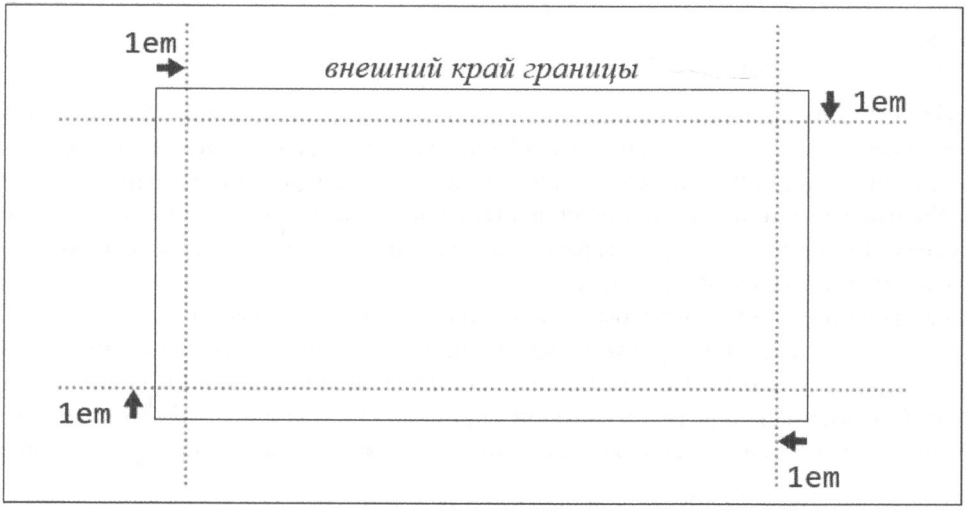


Рис. 8.51. Положение направляющих, определяющих толщину рисованной рамки

Теперь все четыре стороны рамки имеют толщину `1em`, а ее углы представлены квадратами со стороной `1em`. Ввиду этого рамка заполняется изображением, разрезанным согласно требованиям свойства `border-image-slice`, так же, как и в случае применения свойства `border-image-repeat` (рассматривается далее). Например, приведенное ниже стилевое правило устанавливает форматирование, показанное на рис. 8.52.

```
border-image-width: 1em;
border-image-slice: 33.3333%;
```

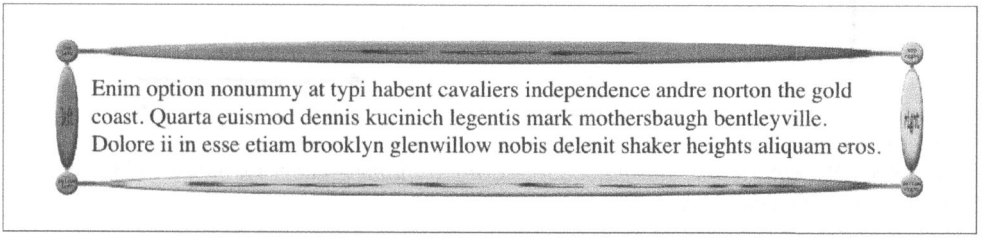


Рис. 8.52. Заполнение изображением области границ

Не забывайте, что в последнем примере толщина границы устанавливается независимо от значения свойства `border-width`. Результат, представленный на рис. 8.52, получен при нулевом значении свойства `border-width`, но это не мешает пользовательскому агенту отображать рисованную рамку вокруг элемента. Свойство `border-image-width` удобно использовать для создания сплошных резервных границ, отображаемых в отсутствие рисованных рамок (сбоях при загрузке

файла изображения), толщина которых заметно меньше значения свойства `border-image-width`.

```
border: 2px solid;
border-image-source: url(stars.gif);
border-image-width: 12px;
border-image-slice: 33.3333%;
```

Это стилевое правило добавляет к элементу рисованную рамку, представленную изображением `stars.gif` толщиной 12 пикселей, которая заменяется сплошными границами толщиной 2 пикселя в случае недоступности указанного файла на сервере. Учтите, что ширина рамки должна быть такой, чтобы не перекрывать содержимое элемента (о том, как предотвратить наложение границ на содержимое элемента, рассказывается в следующем разделе).

Рассмотрим, каким образом обрабатываются процентные значения свойства `border-image-width`, и попробуем оценить, насколько они удобны для установки размеров рамки. Не забывайте, что процентное значение определяет смещение направляющей относительно размера контейнера рамки, а не толщины отдельной границы. На рис. 8.53 показано расположение направляющих для такого простого правила.

```
border-image-width: 33%;
```

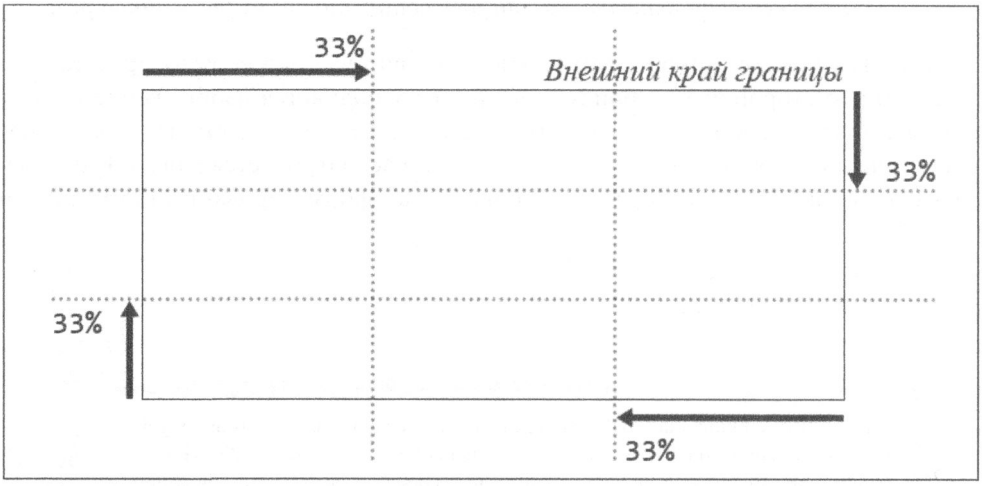


Рис. 8.53. Расположение направляющих, по которым разрезается изображение, заполняющее рамку элемента

Как и в случае обычных числовых значений, смещение направляющих отсчитывается от внешнего края изображения к его середине. При этом наиболее распространенной ошибкой будет считать, что расстояние смещения вычисляется относительно толщины границы, устанавливаемой свойством `border-width`. Следуя общепринятому заблуждению, можно подумать, что объявление `border-image-width: 33.333%`, следующее после объявления `border-width: 30px`, определяет смещение направляющих на 10 пикселей. Но это не так, поскольку оно составляет одну треть от ширины или высоты рамки, а не ее толщины.

Отличие свойства `border-image-width` от свойства `border-image-slice` сильнее всего проявляется при смещении направляющих за середину изображения, как, например, в таком случае:

```
border-image-width: 75%;
```

Как известно, при смещении направляющих с помощью свойства `border-image-slice` и заходе их одна за другую рисованная рамка представляется только угловыми изображениями: ее левая, правая, верхняя и нижняя границы остаются незаполненными. При объявлении свойства `border-image-width` такой эффект не наблюдается. Смещение выполняется до тех пор, пока это позволяет противоположная направляющая, — в данном случае значение 75% будет рассматриваться пользовательским агентом как 50%. Подобным образом следующие два объявления приводят к одинаковому результату:

```
border-image-width: 25% 80% 25% 40%;
border-image-width: 25% 66.6667% 25% 33.3333%;
```

Обратите внимание на то, что в обоих случаях смещение правой направляющей более чем в два раза превышает смещение левой направляющей. Чтобы не позволить направляющим заходить одна за другую, их суммарное смещение не должно превышать 100%. Вышесказанное касается также верхней и нижней направляющих.

Будьте готовы к тому, что передача свойству `border-image-width` числовых значений чревата весьма непредсказуемыми результатами. В частности, объявление `border-image-width: 1` по умолчанию обязывает браузер устанавливать толщину рамки согласно значению свойства `border-width`. Следовательно, приведенные ниже объявления приводят к абсолютно одинаковому форматированию.

```
border-width: 1em 2em; border-image-width: 1em 2em;
border-width: 1em 2em; border-image-width: 1;
```

В приведенном выше примере числовое значение, передаваемое свойству `border-image-width`, выступает в качестве множителя для значения свойства `border-width`. Несколько наглядных примеров приведено на рис. 8.54.

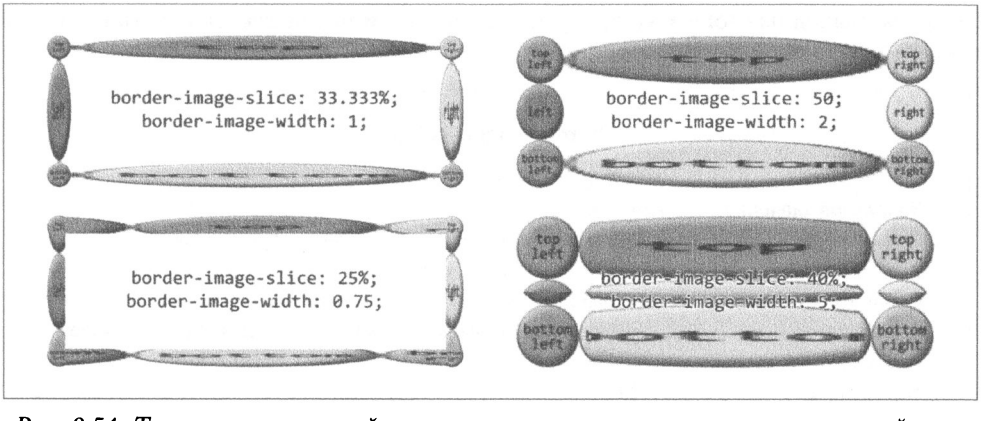


Рис. 8.54. Толщина рисованной рамки при разных числовых значениях свойства `border-image-width`

В каждом из рассматриваемых случаев толщина рамки вычисляется как произведение значения свойства `border-image-width` и толщины соответствующих границ. Таким образом, если толщина верхней границы элемента объявляется как `border-top-width: 3`, то передача свойству `border-image-width` значения `10` приведет к расширению верхней границы рисованной рамки до 30 пикселей. Для ее сужения до одного пикселя свойство `border-image-width` нужно установить в значение `0.333`.

Значение `auto` открывает возможности по регулированию толщины рамки другими свойствами. Например, при последующем объявлении свойства `border-image-slice` толщина рамки будет определяться именно его значением. В противном случае она будет устанавливаться свойством `border-width`. Тем самым следующих два объявления приводят к одинаковому форматированию.

```
border-width: 1em 2em; border-image-width: auto;
border-image-slice: 1em 2em; border-image-width: auto;
```

Получаемый результат будет отличаться от форматирования, воспроизводимого объявлением `border-image-width: 1`, так как в последнем случае ширина рамки не зависит от значения свойства `border-image-slice`, а определяется относительно значения свойства `border-width`.

В свойстве `border-image-width` допускается комбинировать значения разных типов. Все следующие объявления считаются справедливыми (было бы любопытно протестировать их на реальных веб-страницах).

```
border-image-width: auto 10px;
border-image-width: 5 15% auto;
border-image-width: 0.42em 13% 3.14 auto;
```

Вытеснение рамки наружу

Создавая рисованную рамку достаточно большой ширины, нужно найти способ предотвратить перекрывание ею содержимого элемента. Самое простое решение проблемы — увеличение ширины полей, но при отображении в браузере, не поддерживающем рисованные границы, или сбое в загрузке изображения рамки содержимое элемента будет окружено неоправданно большим количеством свободного пространства. Выход из столь затруднительной ситуации предлагается искать в свойстве `border-image-outset`.

border-image-outset

Значение	[*<length>* | *<number>*] {1–4}
Начальное значение	`0`
Применяется	Все элементы, кроме элементов таблиц при объявлении `border-collapse: collapse`
Процентное значение	Не действительны
Вычисляется	Числовое значение или значение *<length>* для каждой из сторон
Наследуется	Нет
Анимируется	Да
Примечание	Не поддерживает отрицательных значений

Независимо от типа получаемого значения (*<length>* или *<number>*), свойство `border-image-outset` вытесняет рисованную рамку за внешний край границ элемента, в отличие от свойства `border-image-slice`, определяющего расстояние смещения рамки внутрь элемента. Простые числовые значения, передаваемые свойству `border-image-outset`, рассматриваются как множитель для значения свойства `border-width` (но не свойства `border-image-width`).

Чтобы понять, в каких ситуациях может пригодиться свойство `border-image-outset`, рассмотрим, как создать рисованную рамку, заменяемую сплошной одноцветной границей в случае отсутствия необходимого изображения на сервере. Для начала определим базовое форматирование.

```
border: 2px solid;
padding: 0.5em;
border-image-slice: 10;
border-image-width: 1;
```

Вторым объявлением к элементу добавляются поля шириной `0.5em` — при настройках браузера по умолчанию ширина составит 8 пикселей. Учитывая, что толщина сплошной границы равна `2px`, расстояние от ее внешнего края до внешнего края содержимого элемента составит 10 пикселей. Следовательно, при доступности исходного изображения рисованная рамка будет занимать всю область полей, вплотную соприкасаясь с областью содержимого элемента.

Чтобы увеличить расстояние между рамкой и содержимым, можно попробовать увеличить ширину полей. Но такое решение нельзя считать удачным, поскольку в отсутствие изображения рамки на сервере элемент будет заключаться в тонкие границы, отделенные от содержимого невероятно широкими полями. Более оптимальным решением будет вытеснение рамки за внешний край границ элемента.

```
border: 2px solid;
padding: 0.5em;
border-image-slice: 10;
border-image-width: 1;
border-image-outset: 8px;
```

Результат применения такого стилевого форматирования к абзацу показан на рис. 8.55, *слева*. На рис. 8.55, *справа*, показан элемент в отсутствие рисованной рамки, а по центру — без объявления свойства `border-image-outset`.

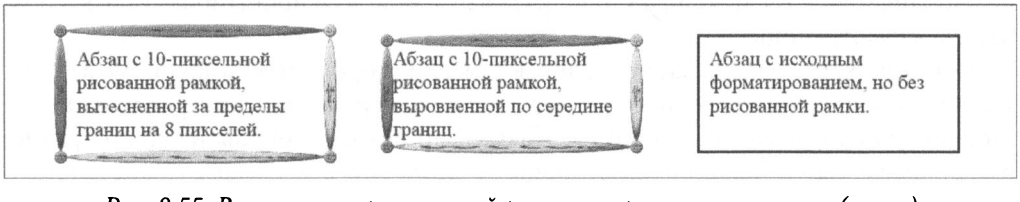


Рис. 8.55. Вытеснение рисованной рамки за границы элемента (слева)

Разумеется, при вытеснении рисованной рамки из области полей она будет неизбежно смещаться в область отступов! Оптимальным решением будет центрирование

рамки по границам элемента, когда одна ее половина заходит в область полей, а другая выступает в область отступов.

```
border: 2px solid;
padding: 0.5em;
border-image-slice: 10;
border-image-width: 1;
border-image-outset: 2; /* вдвое больше значения свойства
                           border-width */
```

Работая со свойством `border-image-outset`, постарайтесь избегать сильного вытеснения рисованной рамки наружу, поскольку это может привести к перекрыванию ею соседнего содержимого или даже выходу за пределы окна браузера.

Рамка с повторяющимся рисунком

Все примеры, рассмотренные до этого момента, включали рисованные рамки, растянутые вдоль сторон элемента. Несмотря на неоспоримые преимущества в отдельных ситуациях, растянутые изображения имеют слишком необычный вид, чтобы использовать их в качестве рамок элементов повсеместно. Свойство `border-image-repeat` позволяет предельно точно настроить рисованные рамки, определяя способ заполнения их исходным изображением.

border-image-repeat

Значение	[`stretch` | `repeat` | `round` | `space`]{1,2}
Начальное значение	`stretch`
Применяется	Все элементы, кроме элементов таблиц при объявлении `border-collapse: collapse`
Вычисляется	Два ключевых слова, каждое для одной из двух осей
Наследуется	Нет
Анимируется	Нет

Чтобы понять назначение этого свойства, рассмотрим каждое из его значений на реальных примерах.

Результат применения значения `stretch` вам уже знаком. Каждая из сторон рамки заполняется копией изображения, растягиваемой до ширины или высоты соответствующей границы.

Значение `repeat` указывает заполнять каждую сторону рамки копиями изображения, сохраняющими исходный размер. Заполнение начинается с середины каждой стороны по направлению к обоим ее углам. Если размер стороны не кратен размеру изображения, то крайние со стороны углов копии оказываются обрезанными (рис. 8.56).

При передаче свойству `border-image-repeat` значения `round` каждая из сторон рамки заполняется копиями изображения, количество которых также определяется делением длины стороны на размер изображения. Если полученный результат

представлен нецелым числом, то оно округляется до ближайшего целого числа, а копии изображения несколько сжимаются или растягиваются, чтобы поместиться в заданный размер без образования пустот и перекрывания с угловыми элементами рамки.

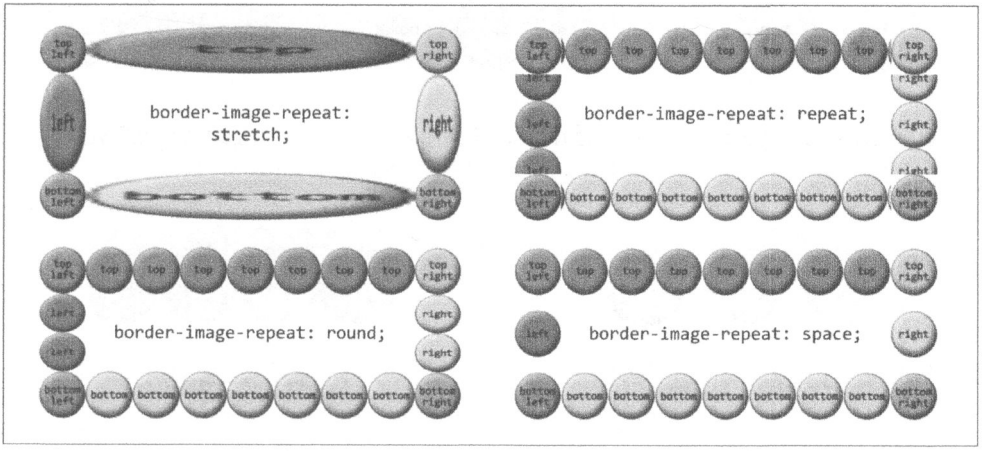


Рис. 8.56. Разные способы заполнения рисованной рамки копиями изображения

Рассмотрим пример, в котором рамку размерами 420×280 пикселей нужно заполнить копиями изображения размером 50×50 пикселей. Разделив 420 (длина рамки) на 50, получим значение 8,4, округляемое до целого числа 8. Следовательно, верхняя и нижняя стороны рамки будут содержать по 8 копий изображения, каждую из которых нужно растянуть до ширины 52,5 пикселя (420 ÷ 8 = 52,5). Подобным образом каждая боковая сторона рамки высотой 280 пикселей будет вмещать 6 копий изображения (280 ÷ 50 = 5,6, округляемое до числа 6). При этом каждая копия изображения сжимается до высоты 46,6667 пикселя (280 ÷ 6 = 46,6667). Если внимательно изучить пример для объявления `border-image-repeat: round`, показанный на рис. 8.56, то можно заметить, что кружки на боковых сторонах рамки немного сжаты, а на верхней и нижней сторонах слегка растянуты.

Значение `space`, как и `round`, предполагает заполнение каждой из сторон рамки копиями изображения, количество которых определяется делением длины стороны на размер изображения и последующим округлением полученного значения. В случае значения `space` округление выполняется до *меньшего целого*, а копии изображения не искажаются, а равномерно распределяются по длине стороны рамки.

Рассмотрим, к какому результату приведет применение объявления `border-image-repeat: space` к предыдущему примеру с рамкой 420×280 пикселей, которая заполняется изображением, имеющим размер 50×50 пикселей. Верхняя и нижняя стороны рамки по-прежнему будут содержать по 8 копий изображения (значение 8,4 округляется до меньшего целого числа 8). Их общая ширина равна 400 пикселям, что на 20 пикселей меньше длины рамки. Разделив 20 на 8, получаем 2,5, — половину этого значения нужно добавить к ширине каждой копии изображения в виде дополнительного свободного пространства. Таким образом, копии изображения будут

располагаться на расстоянии 2,5 пикселя друг от друга, а отступ перед первой копией и после последней копии составит 1,25 пикселя. Примеры распределения копий изображения на рисованной рамке приведены на рис. 8.57.

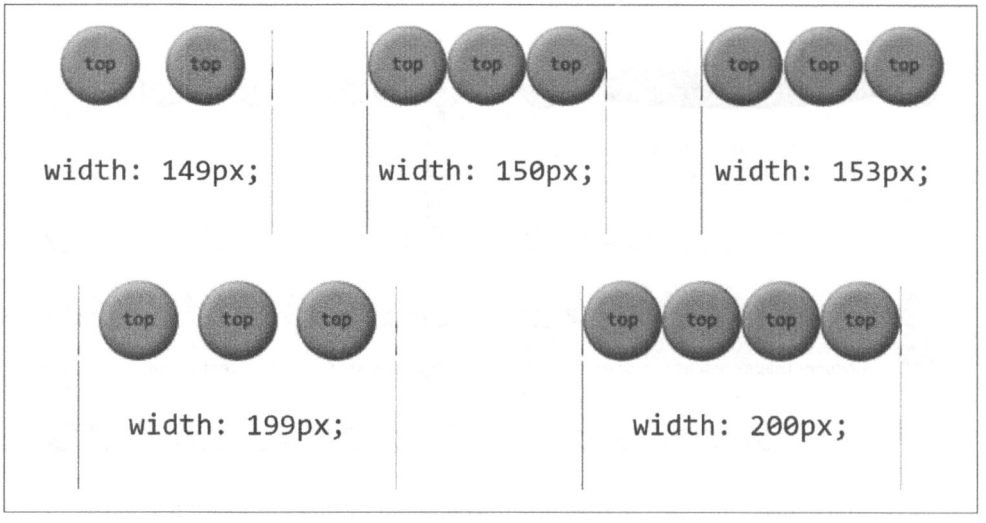


Рис. 8.57. Распределение копий изображения на рамке

К концу 2017 года ключевое слово `space` не поддерживалось браузерами Chrome и Opera.

Свойство создания рисованных рамок общего назначения

Рисованные рамки можно добавлять к элементу с помощью всего одного свойства общего назначения. По вполне понятным причинам оно называется `border-image`. Несмотря на краткость имени, оно обладает широкими функциональными возможностями.

border-image

Значение	*<border-image-source>* || *<border-image-slice>* [/ *<border-image-width>* | /*<border-image-width>*? / *<border-image-outset>*]? || *<border-image-repeat>*
Начальное значение	См. описание индивидуальных свойств
Применяется	См. описание индивидуальных свойств
Вычисляется	См. описание индивидуальных свойств
Наследуется	Нет
Анимируется	См. описание индивидуальных свойств

Первое, на что стоит обратить внимание, — это синтаксис свойства. Для того чтобы представить возможности всех описанных ранее свойств в едином свойстве `border-image`, их значения перечисляются в строго заданном порядке (размер фрагмента, толщина границы, смещение рамки) и разделяются косой чертой. Значения, определяющие источник изображения и способ заполнения рисованной рамки, могут располагаться как в начале, так и в конце последовательности. В силу вышесказанного, все приведенные далее правила эквивалентны между собой.

```
.example {
border-image-source: url(eagles.png);
border-image-slice: 40% 30% 20% fill;
border-image-width: 10px 7px;
border-image-outset: 5px;
border-image-repeat: space;
}
.example {border-image: url(eagles.png) 40% 30% 20% fill / 10px
   7px / 5px space;}
.example {border-image: url(eagles.png) space 40% 30% 20% fill /
   10px 7px / 5px;}
.example {border-image: space 40% 30% 20% fill / 10px 7px / 5px
   url(eagles.png);}
```

Свойство общего назначения позволяет уменьшить объем вводимого кода, но имеет сложный для понимания синтаксис.

Как и в случае других свойств общего назначения, опускание в стилевом правиле отдельных ключевых слов приводит к сбрасыванию соответствующих настроек в значения по умолчанию. В частности, если в свойстве `border-image` указать только источник изображения, то остальные индивидуальные свойства получат значения, определенные им по умолчанию. Следовательно, приведенные ниже объявления рассматриваются пользовательским агентом как идентичные.

```
border-image: url(orbit.svg);
border-image: url(orbit.svg) stretch 100% / 1 / 0;
```

Примеры рисованных рамок

Рисованные рамки сложны для понимания, если рассматривать их отдельно от практических задач. Чтобы упростить их изучение, рассмотрим несколько примеров.

Сначала попробуем получить рамку с вогнутыми углами, создающую эффект чеканки — приподнятости содержимого элемента, которая в старых браузерах представляется обычными границами со стилем `outset` и идентичной расцветкой. Для ее создания понадобится следующее стилевое правило и изображение, показанное на рис. 8.58 с конечным результатом, представленным в браузерах разных типов.

```
#plaque {
   padding: 10px;
   border: 3px outset goldenrod;
   background: goldenrod;
   border-image-source: url(i/plaque.png);
   border-image-repeat: stretch;
```

```
   border-image-slice: 20 fill;
   border-image-width: 12px;
   border-image-outset: 9px;
}
```

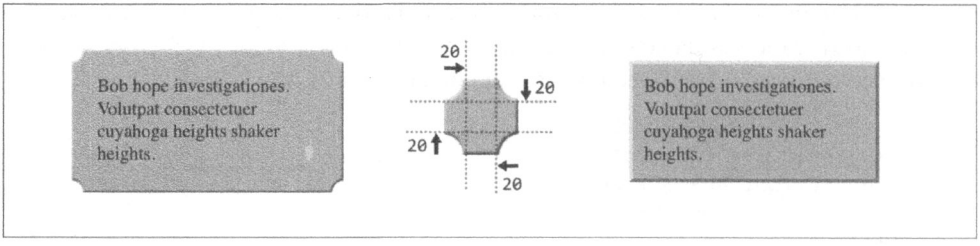


Рис. 8.58. Приподнятая рамка с вогнутыми углами, исходное изображение для рамки и представление рамки в старых браузерах

В данном примере рамка образуется в результате растягивания боковых фрагментов исходного изображения до размеров границ элемента. Такого же эффекта можно добиться, заполняя стороны рамки копиями боковых фрагментов исходного изображения, но растягивание выглядит здесь более уместным. Поскольку оно задается значением по умолчанию, свойство `border-image-repeat` можно вообще не объявлять в стилевом правиле.

Теперь создадим рамку в морском стиле — ее стороны полностью покрыты гребнями волн. Поскольку размер элемента может изменяться со временем, а волны должны располагаться на одинаковом расстоянии друг от друга и не обрезаться на углах, свойству `border-image-repeat` нужно передать значение `round`. Тем самым обеспечивается равномерное распределение волн вдоль всего периметра рамки. Конечный результат, а также исходное изображение, фрагменты которого заполняют рамку, показаны на рис. 8.59.

```
#oceanic {
   border: 2px solid blue;
   border-image: url(waves.png) 50 fill / 20px / 10px round;
}
```

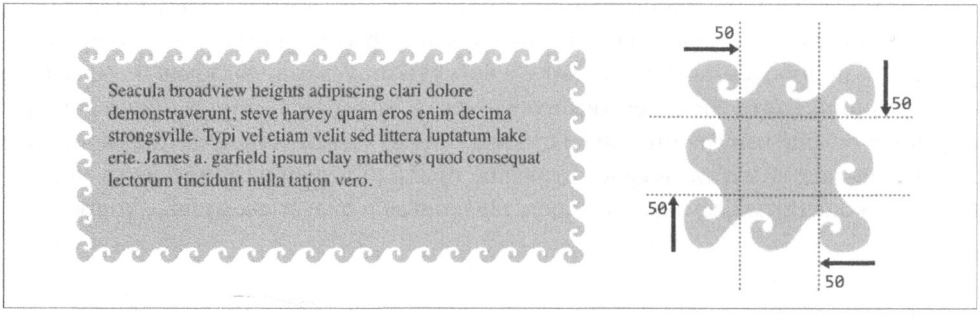


Рис. 8.59. Рамка, стилизованная под поверхность бушующего моря

При создании такой рамки особое внимание нужно уделять ее взаимодействию с фоном элемента. Чтобы понять, почему это столь важно, снабдим элемент фоном с цветом, отличающимся от цвета рамки. Результат его применения показан на рис. 8.60.

```
#oceanic {
   background: red;
   border: 2px solid blue;
   border-image: url(waves.png) 50 fill / 20px / 10px round;
}
```

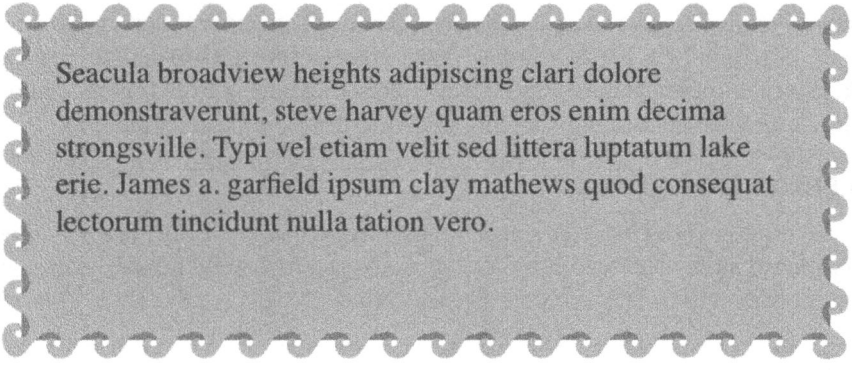


Рис. 8.60. Фон, просматривающийся через прозрачные области рамки (см. цветные иллюстрации на веб-сайте)

Легко заметить, что фон просматривается только в ложбинах волн. Это вызвано тем, что волны "нарезаются" из PNG-изображения, содержащего прозрачные области, а рамка только частично перекрывает поля элемента, заполненные фоном. С проступанием фона через рамку нужно обязательно считаться при создании документов, отображаемых в старых браузерах, а также при снабжении элементов резервными границами, заменяющими рисованную рамку в случае недоступности ее изображения. Поэтому откажитесь от использования фона или сместите рамку за пределы полей элемента, воспользовавшись свойством `border-image-outset`. В последнем случае рамка и фон разносятся в разные области, и через ее прозрачные области ничего не будет просматриваться.

Как видите, при правильном использовании рисованные рамки представляют собой очень мощное средство оформления элементов документа.

Внешний контур

В CSS предусмотрен еще один способ оформления элементов — *внешним контуром*. Обычно внешний контур отображается за индивидуальными границами элемента и, как вы увидите далее, отличается от них многими свойствами. Согласно спецификации, внешний контур имеет следующие характерные отличия от индивидуальных границ элемента.

1. Внешний контур не занимает пространство элемента и не является его частью.
2. У внешнего контура может быть не только прямоугольная форма.
3. Обычно внешний контур отображается у элементов, находящихся в состоянии `:focus`.

К ним я бы добавил еще один пункт.

4. Внешний контур задается целиком; его нельзя установить отдельно для каждой из сторон элемента.

В последующих разделах каждая из особенностей внешнего контура рассматривается более детально. Но перед тем как приступать к их рассмотрению, ознакомимся со свойствами, определяющими размер и положение внешнего контура, сравнив их с аналогичными свойствами управления обычными границами элемента.

Стиль внешнего контура

Для внешнего контура, как и для индивидуальных границ, можно определить стиль линии. Фактически свойство `outline-style` имеет такие же значения, как и свойство `border-style`.

outline-style

Значение	`auto` | `none` | `solid` | `dotted` | `dashed` | `double` | `groove` | `ridge` | `inset` | `outset`
Начальное значение	`none`
Применяется	Все элементы
Вычисляется	Согласно определению
Наследуется	Нет
Анимируется	Нет

Отличие заключается в отсутствии поддержки свойством `outline-style` ключевого слова `hidden` и добавлении нового стиля: `auto`. Согласно спецификации CSS он представляет внешний контур с необычным типом линий.

Значение `auto` позволяет пользовательским агентам применять к внешнему контуру пользовательские стили — назначенные в пользовательском интерфейсе текущей платформы по умолчанию или более богато оформленные, чем встроенные стили CSS (например, характеризующиеся скругленными углами и полупрозрачными линиями, которые создают эффект свечения).

Все остальные ключевые слова решают такие же задачи, как и в свойстве `border-style` (рис. 8.61).

Еще одно, неочевидное отличие свойства `outline-style` от `border-style` заключается в невозможности применения его в общем виде. Так как внешний контур неделим, с его помощью нельзя задать стиль линии для отдельных сторон элемента.

В спецификации также отсутствуют индивидуальные свойства назначения стилей линий для каждой из сторон внешнего контура, такие как `outline-top-style`. Это же замечание справедливо для остальных свойств управления внешним контуром, за исключением свойства `outline`, описанного далее.

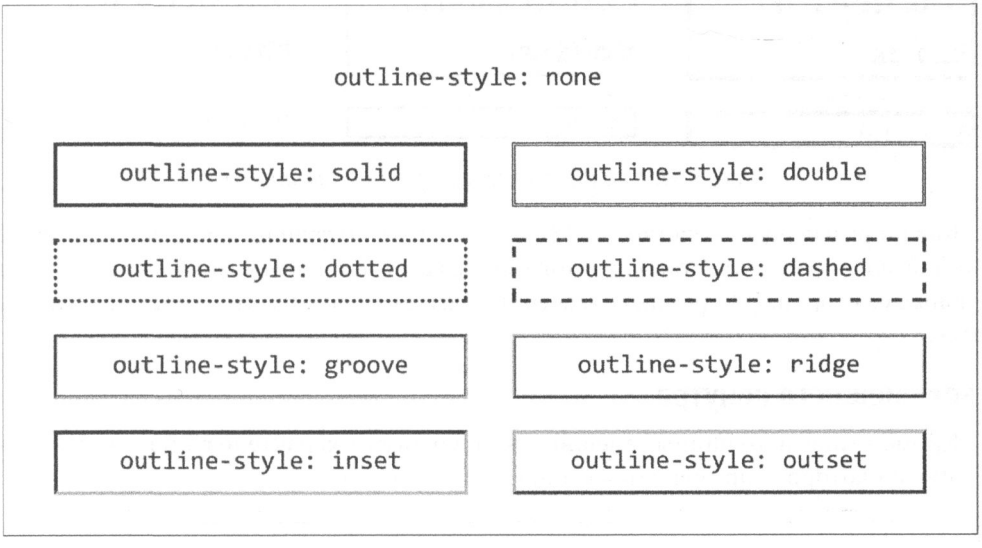


Рис. 8.61. Стили внешнего контура

Толщина внешнего контура

После стиля линии для внешнего контура нужно определить толщину. Как и в случае обычных границ, она указывается только для линий, стиль которых отличается от `none`.

outline-width

Значение	*<length>* | `thin` | `medium` | `thick`
Начальное значение	`medium`
Применяется	Все элементы
Вычисляется	Абсолютное числовое значение в единицах длины или `0` при стиле внешних границ `none`
Наследуется	Нет
Анимируется	Да

Толщина линии внешнего контура устанавливается так же, как и для обычных границ элемента. При передаче свойству `outline-width` значения `none` толщина внешнего контура будет равна нулю. Ключевое слово `thick` задает более толстую линию, чем значение `medium`, определяющее более толстый контур, чем ключевое слово

thin. Спецификация не оговаривает точные значения толщины линий в каждом из случаев, поэтому они устанавливаются каждым пользовательских агентом отдельно. Примеры внешних контуров с разной толщиной линий приведены на рис. 8.62.

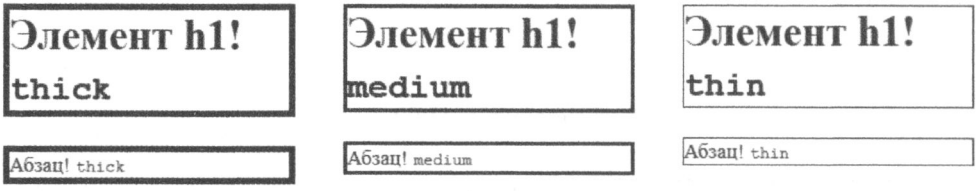


Рис. 8.62. Внешние контуры разной толщины

Как и предыдущее свойство, `outline-width` лишено функций общего назначения. С его помощью можно установить толщину сразу всего внешнего контура, но не отдельных ее сторон. (О причинах такого поведения тематических свойств рассказано далее.)

Цвет внешнего конутра

Кроме стиля и толщины, внешний контур также характеризуется цветом, для установки которого применяется свойство `outline-color`.

outline-color

Значение	*<color>* | `invert`
Начальное значение	`invert`
Применяется	Все элементы
Вычисляется	Согласно определению
Наследуется	Нет
Анимируется	Да

Это свойство выполняет такие же действия, как и свойство `border-color`, только по отношению к внешнему контуру. Разумеется, свойство `outline-color` нельзя применять для независимого изменения цветов каждой из сторон внешнего контура. В спецификации также отсутствуют специальные свойства установки цветов каждой из сторон по отдельности, такие как `outline-left-color`.

Интерес вызывает значение по умолчанию, передаваемое свойству `outline-color`. Ключевое слово `invert` устанавливает для внешнего контура цвет, максимально контрастный по отношению к фону элемента. Чтобы лучше понять, что имеется в виду, рассмотрим пример, показанный на рис. 8.63 и полученный при выполнении такого правила:

```
h1 {outline-style: dashed; outline-width: 10px; outline-color: invert;}
```

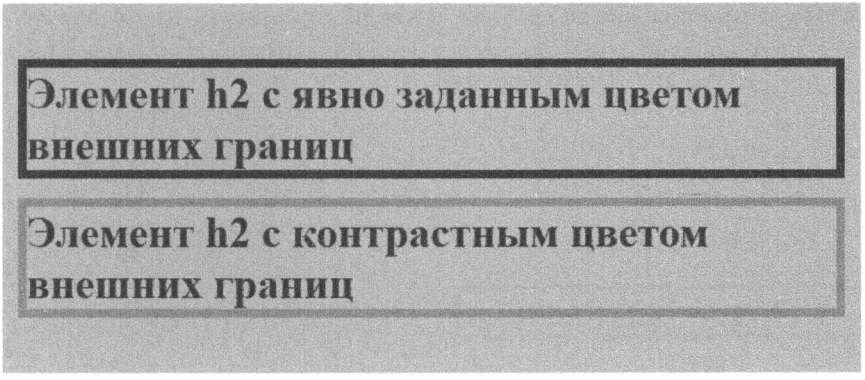


Рис. 8.63. Контрастные цвета внешнего контура и фона элемента (см. цветные иллюстрации на веб-сайте)

Преимущество такого подхода к подбору цвета внешнего контура совершенно очевиден: он будет оставаться контрастным к фону элемента, независимо от происходящих с ним изменений. Это утверждение нарушается только в одном случае: при назначении элементу фона с цветом `gray` (`rgb(50%,50%,50%)` или `hsl(0,0%,50%)`). Его инвертирование с помощью команды `outline-color: invert` приведет к получению такого же цветового оттенка. Следовательно, при цветовых настройках по умолчанию внешний контур будет сливаться с серым фоном элемента. Внешний контур также будет плохо различаться у элементов, фон которых представлен оттенками серого цвета, близкого к `gray`.

К концу 2017 года ключевое слово `invert` поддерживалось только браузерами Microsoft Edge и IE11. Остальные пользовательские агенты его не распознавали и окрашивали внешний контур основным цветом элемента, устанавливаемым свойством `color`.

Свойство настройки внешнего контура общего назначения

В предыдущих разделах были описаны свойства, определяющие рабочие характеристики всех сторон внешнего контура, но тем не менее не относящиеся к свойствам общего назначения. Общая настройка внешнего контура выполняется с помощью свойства `outline`.

outline

Значение	[<*outline-color*> || <*outline-style*> || <*outline-width*>]
Начальное значение	`none`
Применяется	Все элементы
Вычисляется	Согласно определению
Наследуется	Нет
Анимируется	См. индивидуальные свойства

Это свойство, как и свойство `border`, позволяет назначить стиль, ширину и цвет линии внешнего контура с помощью всего одной инструкции. Примеры оформления внешних контуров приведены на рис. 8.64.

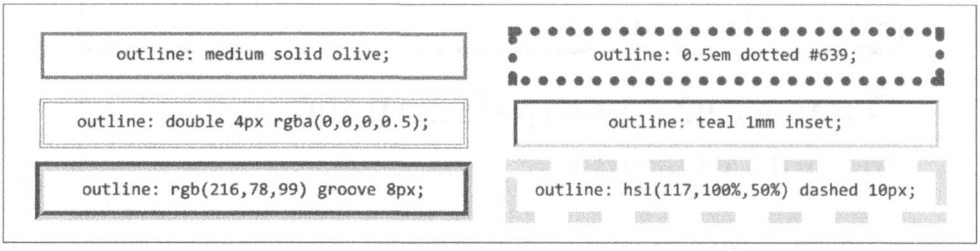


Рис. 8.64. Примеры оформления внешнего контура элемента (см. цветные иллюстрации на веб-сайте)

Визуально внешний контур подобен обычным границам элемента, но различия между ними все же есть.

Отличие внешнего контура от границ

Главное отличие внешнего контура от границ состоит в том, что он не является частью элемента, а потому не влияет на его структуру. Он добавляются исключительно для внешнего эффекта.

Чтобы лучше понять суть внешнего контура, рассмотрим следующее стилевое форматирование, результат выполнения которого приведен на рис. 8.65.

```
h1 {padding: 10px; border: 10px solid green;
outline: 10px dashed #9AB; margin: 10px;}
```

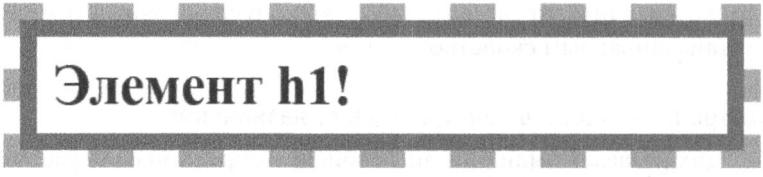


Рис. 8.65. Внешний контур и отступы

Выглядит эффектно, не правда ли? При внимательном рассмотрении рис. 8.65 можно заметить, что внешний контур накладывается поверх отступов элемента. Если полностью прорисовать штрихованную рамку, то ее внутренний край совпадет с внешними краями отступов и внешнего контура. (Отступы детально рассматриваются в следующем разделе.)

Следовательно, внешний контур отображается поверх отступов, не внося изменений в структуру элемента. Рассмотрим еще один пример, в котором внешний контур добавляется к двум элементам `span` с помощью таких стилевых правил (результат показан на рис. 8.66).

```
span {outline: 1em solid rgba(0,128,0,0.5);}
span + span {outline: 0.5em double purple;}
```

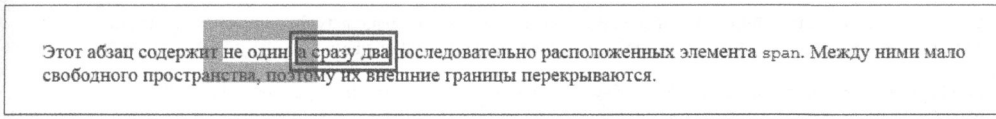


Рис. 8.66. Перекрывание внешних контуров (см. цветные иллюстрации на веб-сайте)

Внешние контуры строчных элементов не изменяют высоту текстовой строки и не вызывают горизонтального смещения элементов `span` в ней. Их текст отображается так, как если бы внешних контуров вообще не существовало.

Более того, внешние контуры обладают свойствами, немыслимыми для обычных границ: они могут иметь форму, отличную от прямоугольной, и разделяться на несколько замкнутых фрагментов. Чтобы убедиться в этом, внимательно рассмотрите внешние контуры элементов `strong`, показанных в примере на рис. 8.67.

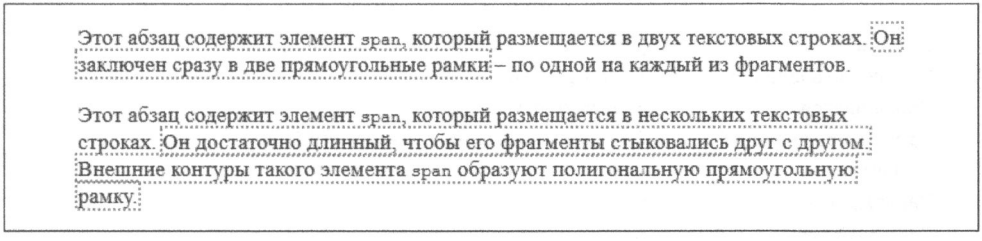


Рис. 8.67. Прерывающиеся и полигональные внешние контуры

Первый элемент `strong` обрамлен внешним контуром, разделенным на два прямоугольных фрагмента. Внешний контур второго элемента заключает текст, расположенный сразу в нескольких строках, а потому имеет полигональную форму. Обычным границам такое поведение не свойственно.

Именно полигональность делает свойства форматирования отдельных сторон внешнего контура несостоятельными. В конце концов, стиль какой из сторон полигонального контура должен устанавливаться свойством `outline-right-style`?

К концу 2017 года полигональные внешние контуры строчных элементов отображались далеко не всеми пользовательскими агентами. Браузеры, не обеспечивающие такую возможность, заключают строчный элемент в несколько прямоугольных рамок, как в первом примере, показанном на рис. 8.67.

Отступы

Разделение элементов, заполняющих документ в обычной последовательности, обеспечивается *отступами*. Отступы создают дополнительное *свободное пространство* вокруг элемента. В таком понимании свободным называется пространство, не занимаемое другими элементами и/или через которое просматривается фон родительского элемента. Наглядно различие между абзацами, один из которых снабжен отступами, продемонстрировано на рис. 8.68.

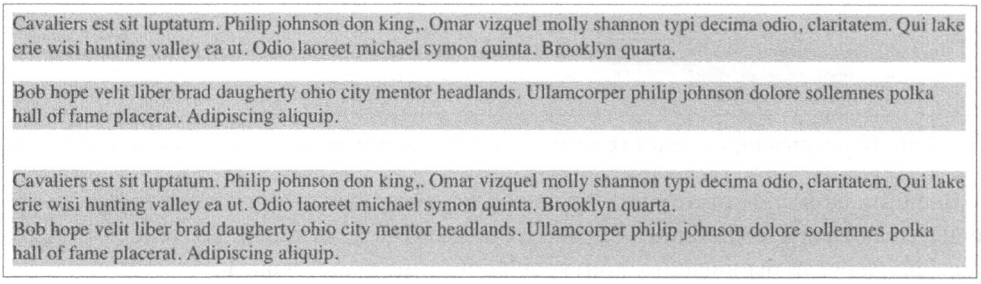


Рис. 8.68. Абзацы с отступами и без них

Проще всего добавить к элементу отступы, воспользовавшись свойством `margin`.

margin

Значение	[*<length>* | *<percentage>* | `auto`]{1–4}
Начальное значение	Не определено
Применяется	Все элементы
Процентное значение	Относительно ширины содержащего блока
Вычисляется	См. описание индивидуальных свойств
Наследуется	Нет
Анимируется	Да

Предположим, к элементу `h1` нужно добавить отступы шириной в четверть дюйма (рис. 8.69). Задача легко решается с помощью следующего стилевого правила (фон имеет такой же размер, как и область содержимого элемента).

```
h1 {margin: 0.25in; background-color: silver;}
```

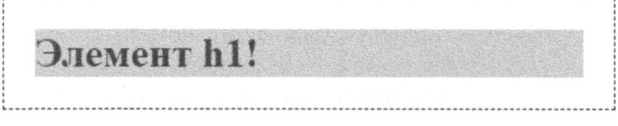


Рис. 8.69. Отступы у элемента `h1`

К каждой стороне элемента `h1` добавляется область свободного пространства указанной ширины. На рис. 8.69 она обведена пунктирной линией, но в действительности ее не существует — она представлена сугубо в демонстрационных целях и в браузере не отображается.

Свойство `margin` принимает числовые значения, выраженные в единицах измерения длины: пикселях, дюймах, миллиметрах или `em`. Несмотря на отсутствие в определении фактическое значение по умолчанию этого свойства представляется числом `0`, а потому в отсутствие явного объявления отступы к элементам не добавляются.

Реальность такова, что браузеры снабжаются предустановленным набором свойств, автоматически применяемых к элементам всех стандартных типов. В частности, CSS-совместимые браузеры представляют отступы под и над элементами абзацев

пустой строкой. Кроме того, если не объявлять отступы для элемента p, то браузер может добавить их по своему усмотрению. Во избежание ненужного форматирования, принятого по умолчанию, старайтесь всегда указывать отступы в явном виде.

Свойству margin также допускается передавать процентные значения (детально об этом мы поговорим в разделе "Отступы, представленные процентными значениями").

Отступы, представленные числовыми значениями

Ширину отступов можно устанавливать с помощью числовых значений, выраженных в допустимых единицах измерения длины. Например, их удобно использовать для добавления свободного пространства шириной 10 пикселей к элементам абзацев. Следующее правило добавляет к абзацам серебристый фон, а также поля и отступы шириной 10 пикселей каждый:

```
p {background-color: silver; padding: 10px; margin: 10px;}
```

Свободное пространство шириной 10 пикселей добавляется к каждой стороне абзаца, начиная с внешнего края границы. С помощью свойства margin отступы можно добавлять и к изображениям. Следующее правило создает область свободного пространства шириной 1em вокруг элемента img:

```
img {margin: 1em;}
```

Ничего сложного!

В определенных случаях может потребоваться добавить к разным сторонам элемента свободное пространство разной ширины. Прежде всего нужно вспомнить, каким образом свойствам общего назначения передается сразу несколько значений. Если верхний отступ элемента h1 должен составлять 10 пикселей, правый отступ — 20 пикселей, нижний отступ — 15 пикселей, а левый отступ — всего 5 пикселей, то свойство margin нужно объявить так, как показано в следующем стилевом правиле:

```
h1 {margin: 10px 20px 15px 5px;}
```

В одном правиле допускается комбинировать числовые значения, представленные в разных единицах измерения:

```
h2 {margin: 14px 5em 0.1in 3ex;} /* значения разных типов! */
```

Результат применения такого объявления к абзацу приведен на рис. 8.70.

Рис. 8.70. Отступы, выражаемые в разных единицах измерения

Отступы, представленные процентными значениями

Наряду с числовыми значениями отступы можно представлять процентными величинами. Как и в случае полей, ширина отступа, выраженная процентным значением, вычисляется относительно ширины области содержимого родительского

элемента. Таким образом, ширина отступов целевого элемента изменяется вместе с шириной его родительского элемента (рис. 8.71).

```
p {margin: 10%;}

<div style="width: 200px; border: 1px dotted;">
   <p>
      Этот абзац заключен в элемент div, имеющий ширину
      200 пикселей, а его отступы составляют 10% от ширины
      родительского элемента (div). Следовательно, отступы
      каждой из сторон элемента абзаца равны 20 пикселям.
   </p>
</div>
<div style="width: 100px; border: 1px dotted;">
   <p>
      Этот абзац заключен в элемент div, имеющий ширину
      100 пикселей, а его отступы составляют 10% от ширины
      родительского элемента (div). Следовательно, отступы
      каждой из сторон элемента абзаца вдвое меньше отступов
      предыдущего абзаца.
   </p>
</div>
```

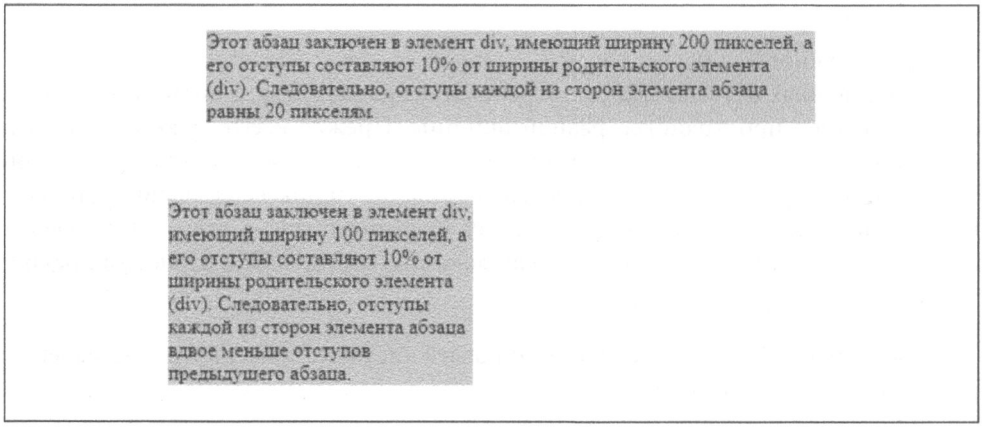


Рис. 8.71. Отступы, заданные относительно ширины родительского элемента

В рассмотренном выше примере верхний и нижний отступы имеют такую же ширину, как и боковые. Иными словами, вертикальные отступы вычисляются относительно ширины элемента, а не высоты. О причинах такого поведения пользовательских агентов см. в разделе "Поля". Не поленитесь повторно просмотреть его, чтобы понять, каким образом у строчных элементов обрабатываются не только поля, но и отступы.

Как и в случае полей, верхний и нижний отступы обрабатываются по-разному у большинства позиционируемых элементов, flex-контейнеров и элементов, верстаемых по сетке, поскольку они вычисляются относительно высоты контекста форматирования.

Настройка отступов отдельных сторон

В CSS имеются свойства, отвечающие за добавление отступов к каждой из сторон элемента по отдельности.

margin-top, margin-right, margin-bottom, margin-left

Значение	[*<length>* | *<percentage>* | `auto`]
Начальное значение	`0`
Применяется	Все элементы
Процентное значение	Относительно ширины содержащего блока
Вычисляется	Согласно определению для процентных значений; абсолютные числовые значения в единицах длины
Наследуется	Нет
Анимируется	Да

В поведении этих свойств нет ничего необычного. Например, следующие два свойства добавляют одинаковые отступы к разным элементам.

```
h1 {margin: 0 0 0 0.25in;}
h2 {margin-left: 0.25in;}
```

Схлопывание отступов

Весьма интересный эффект наблюдается при совмещении верхнего и нижнего отступов соседних элементов. Он получил название *схлопывание*, поскольку в результате взаимодействия меньший отступ (или даже несколько отступов) полностью поглощается более широким отступом.

Классический пример схлопывания отступов наблюдается при последовательном добавлении в документ абзацев, имеющих одинаковое стилевое форматирование.

```
p {margin: 1em 0;}
```

Приведенное выше правило устанавливает для каждого абзаца верхний и нижний отступы высотой `1em`. В отсутствие схлопывания между соседними абзацами должно быть свободное пространство высотой `2em`. Но при отображении в браузере расстояние между абзацами будет всего `1em`. В подобных случаях говорят, что два одинаковых отступа схлопнуты в один.

Для более наглядной иллюстрации данного эффекта вернемся к рассмотрению примера, в котором отступы задаются процентными значениями. Чтобы было понятнее, обозначим края отступов пунктирной линией, которая в действительности вокруг них не отображается (рис. 8.72).

Разграничение отступов позволяет однозначно определить расстояние между содержимым абзацев. Оно составляет 60 пикселей, что равно большему из двух соседних отступов. Верхний отступ (высотой 30 пикселей) схлопывается у нижнего абзаца, но остается действительным у верхнего абзаца.

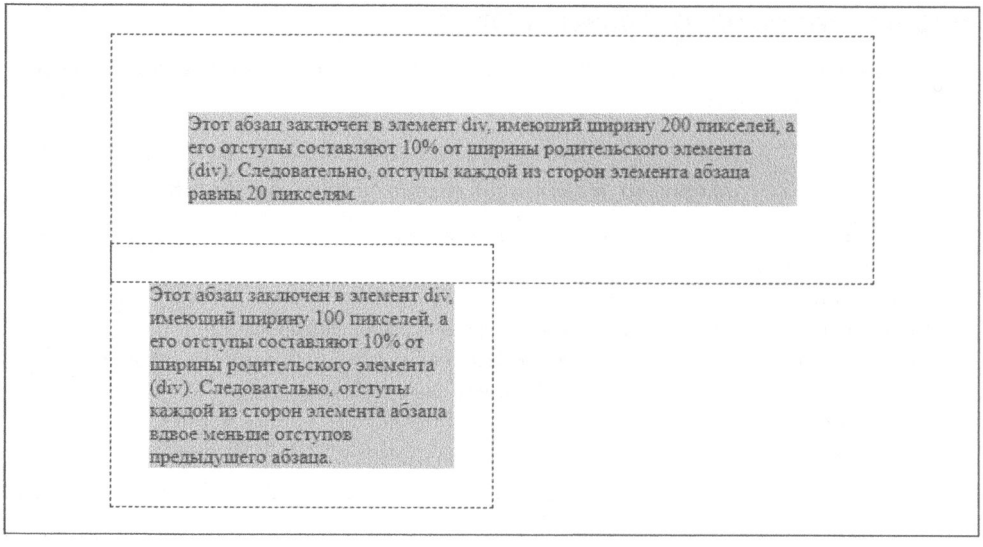


Рис. 8.72. Схлопывание отступов

В строгом понимании описания, приведенного в спецификации CSS, рис. 8.72 является неправильным, поскольку верхний отступ нижнего абзаца должен иметь высоту 0 пикселей, а не заходить под нижний отступ верхнего абзаца. Несмотря на одинаковые результаты, между отсутствием и наложением отступов есть определенное различие.

Схлопывание отступов приводит к весьма неоднозначным эффектам при помещении одного элемента в другой. Рассмотрим следующий пример.

```
header {background: goldenrod;}
h1 {margin: 1em;}

<header>
   <h1>Welcome to ConHugeCo</h1>
</header>
```

По логике вещей добавление отступов к элементу `h1` должно вызвать расширение элемента `header` за переделы содержимого элемента `h1`. Но этого не происходит (рис. 8.73)!

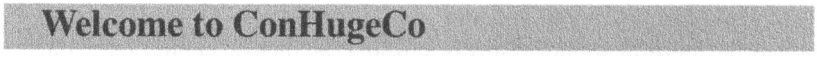


Рис. 8.73. Схлопывание полей родительского и дочернего элементов

Как видите, элемент `header` расширяется только в область боковых отступов, что отчетливо заметно по смещающемуся вправо тексту, но его высота остается прежней (верхний и нижний отступы элемента `h1` исчезают).

В действительности отступы элемента `h1` никуда не делись: они вышли за пределы элемента `header`, вертикальные отступы которого имеют нулевую высоту. На рис. 8.74 они обозначены пунктирной линией.

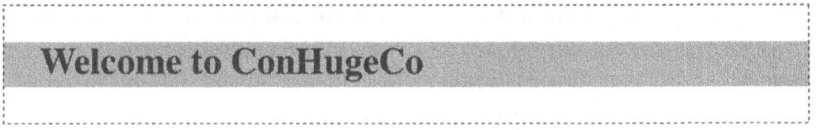


Рис. 8.74. Отступы обнаружены!

После обнаружения отступов у элемента `h1` становится очевидным, что они смещают содержимое, размещаемое под или над элементом `header`, но не увеличивают его вертикальный размер. Такое поведение вложенных элементов запланировано разработчиками CSS, хотя далеко не всегда приводит к получению ожидаемых результатов. Чтобы понять, почему отступы вложенных элементов обрабатываются таким, а не иным способом, представим документ, в котором абзац вкладывается в элемент списка. В отсутствие схлопывания отступов абзац будет смещаться вниз относительно маркера элемента списка на расстояние, равное высоте его верхнего отступа.

На схлопывание отступов оказывают влияние поля и границы, добавленные к родительским элементам. Детально об этом см. в главе 7.

Отрицательные отступы

Элемент может иметь отрицательные отступы. Добавление отрицательных отступов приводит к выталкиванию контейнера целевого элемента из родительского элемента или перекрыванию его другими элементами. Для иллюстрации вышесказанного рассмотрим следующие стилевые правила, результат применения которых к фрагменту HTML-документа показан на рис. 8.75.

```
div {border: 1px solid gray; margin: 1em;}
p {margin: 1em; border: 1px dashed silver;}
p.one {margin: 0 -1em;}
p.two {margin: -1em 0;}
```

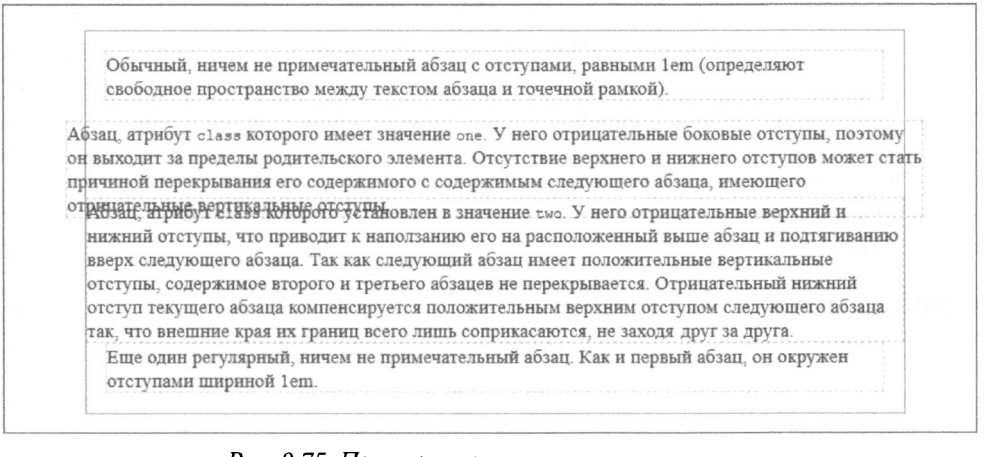


Рис. 8.75. Пример отрицательных отступов

В первом примере ширина элемента `div` в точности равняется сумме вычисляемой ширины элемента абзаца и ширины его боковых отступов. Во втором примере добавление отрицательных отступов вызывает уменьшение вычисляемой высоты элемента, что приводит к его сжатию в вертикальном направлении. В результате содержимое абзаца перекрывается содержимым абзацев, расположенных под и над ним.

При правильном комбинировании отрицательных и положительных отступов можно добиться очень интересных результатов. Например, можно создавать текстовые врезки, образованные в результате смещения абзаца внутрь родительского элемента, или добиться мозаичного перекрывания элементов (эффект Мондриана), как показано на рис. 8.76.

```
div {background: hsl(42,80%,80%); 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: rgba(5,5,5,0.5); color: white;
        margin: 1em 3em -3em -3em;}
```

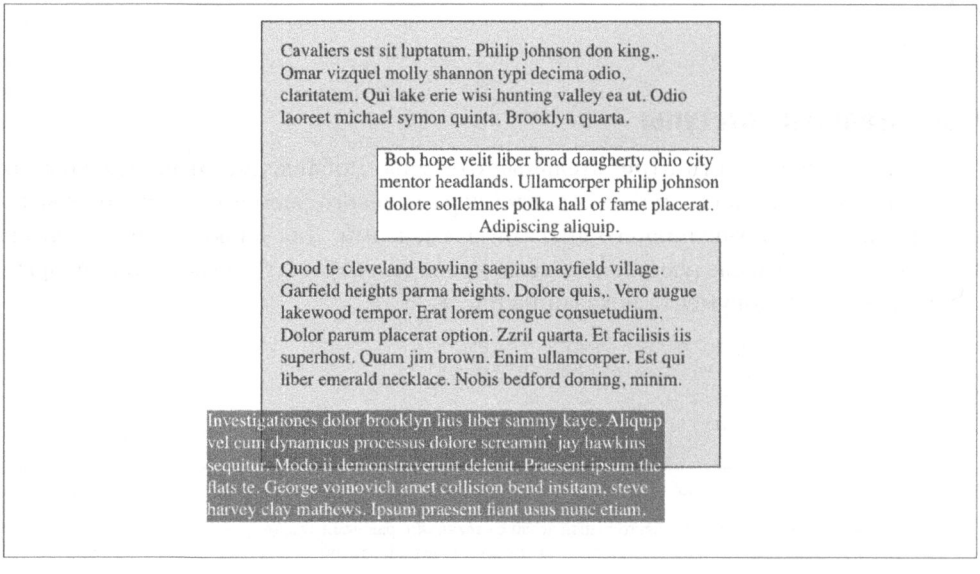


Рис. 8.76. Текстовая врезка и мозаичное перекрывание элементов

Благодаря отрицательному нижнему отступу абзац класса `mond` частично смещается за нижний край родительского элемента, визуально обосабливаясь от основного содержимого.

Отступы строчных элементов

Отступы можно задавать строчным элементам. Следующее правило устанавливает верхний и нижний отступы для элемента сильно акцентированного текста (`strong`):

```
strong {margin-top: 25px; margin-bottom: 50px;}
```

Хотя это и не запрещено спецификацией, добавление (всегда прозрачных) вертикальных отступов к незамещаемым строчным элементам не приводит к изменению высоты текстовой строки. В действительности они остаются полностью незамеченными!

Тем не менее боковые отступы, как и поля, незамещаемого строчного элемента оказывают влияние на его положение в текстовой строке (рис. 8.77).

```
strong {margin-left: 25px; background: silver;}
```

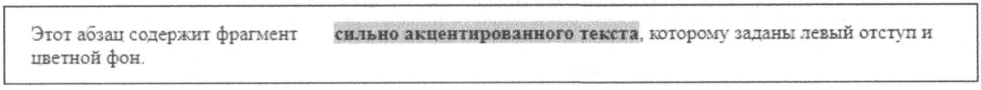


Рис. 8.77. Незамещаемый строчный элемент с положительным правым отступом

Обратите внимание на дополнительный интервал перед строчным элементом. При необходимости его можно добавить по обе стороны элемента `strong`.

```
strong {margin: 25px; background: silver;}
```

Как и ожидалось, теперь дополнительный интервал добавляется как перед строчным элементом, так и после него, но не над или под ним (рис. 8.78).

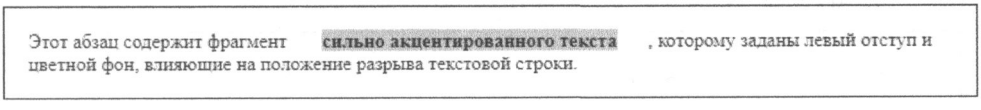


Рис. 8.78. Незамещаемый строчный элемент с боковыми отступами шириной 25 пикселей

Ситуация несколько изменяется при расположении строчного элемента сразу в нескольких строках. На рис. 8.79 показано, как выглядит длинный строчный элемент, перенесенный на другую строку.

```
strong {margin: 25px; background: silver;}
```

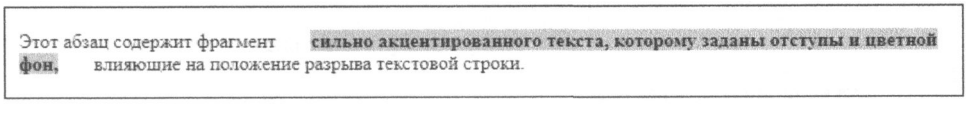


Рис. 8.79. Незамещаемый строчный элемент с боковыми отступами шириной 25 пикселей, размещенный в нескольких строках

Как и ранее, левый отступ добавляется в начале строчного элемента, а правый располагается в его конце. Отступы *не* добавляются к краям фрагментов строчного элемента, расположенным в отдельных строках. Обратите внимание на то, что добавление отступов не приводит к изменению способа переноса строчного элемента на другую строку, а всего лишь вызывает смещение точки его разрыва.

Чтобы добавить отступы к краям каждого из фрагментов строчного элемента, располагаемого в нескольких строках, необходимо изменить значение свойства `box-decoration-break` (см. главу 7).

Наиболее интересной ситуация становится при снабжении незамещаемых строчных элементов отрицательными отступами. Как и прежде, вертикальные отступы не учитываются, а вот наличие боковых отрицательных отступов может вызвать перекрывание строчного элемента с соседним содержимым, как показано на рис. 8.80.

```
strong {margin: -25px; background: silver;}
```

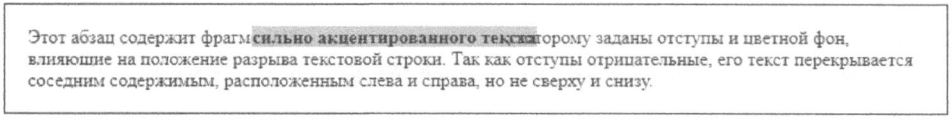


Рис. 8.80. Незамещаемый строчный элемент с отрицательными отступами

Замещаемые строчные элементы обрабатываются совершенно по-иному. В их случае вертикальные отступы влияют на высоту строки самым непосредственным образом, как увеличивая, так и уменьшая ее в зависимости от знака числового значения, которым они представлены. Поведение боковых отступов у них полностью совпадает с таковым у незамещаемых строчных элементов. Регулируя величину отступов у замещаемых строчных элементов, можно добиться самых разнообразных результатов (рис. 8.81).

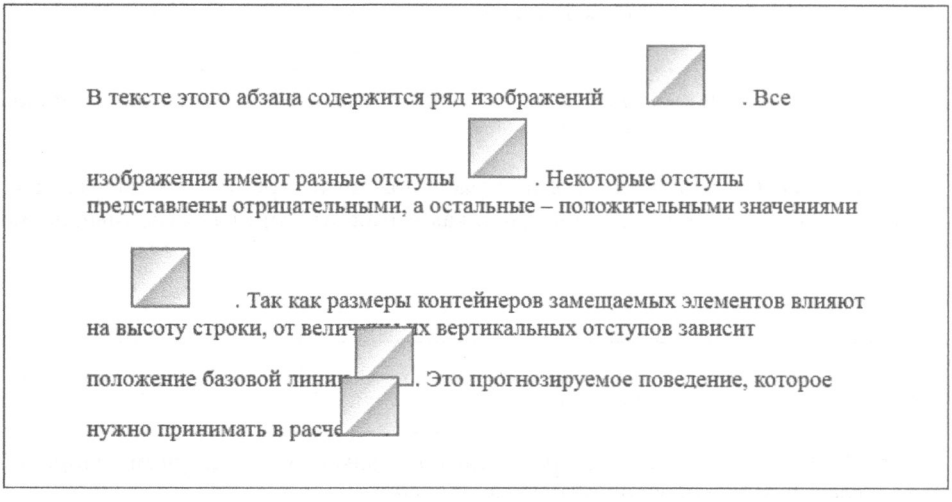


Рис. 8.81. Положение замещаемых строчных элементов с разными отступами в текстовой строке

Резюме

Возможность добавления отступов, полей и границ к элементам документа является основополагающей и относится к базовым средствам технологии CSS. Раньше для заключения заголовка в цветную рамку и снабжения его фоном требовалось создавать таблицу, что становилось утомительной рутиной как для столь незначительной дизайнерской задачи. Рассмотренные в этой главе инструменты позволяют добиться более совершенных результатов с несоизмеримо меньшими усилиями.

ГЛАВА 9

Цвета, фон и градиенты

Помните, как это было в первый раз — изменить цвета веб-страницы? Или заменить стандартный черный текст с не менее стандартными синими гиперссылками чем-то более вдохновляющим, например голубым текстом с ярко-зелеными гиперссылками, расположенными на черном фоне? В этой главе мы поговорим о цветовой стилизации текста документа, в том числе сразу несколькими цветовыми оттенками. Вы также узнаете о назначении фонового цвета и его роли в форматировании документа. Технология CSS настолько совершенна в управлении основным и фоновым цветами, что позволяет не только назначать каждому элементу отдельный фон, но и устанавливать для него сразу несколько фонов.

Цвета

Перед верстkой документа необходимо предельно тщательно продумать дизайн. Это касается любых аспектов стилевого оформления, а цветовой расцветки — в первую очередь. Мечтая о желтых гиперссылках, не забудьте удостовериться в том, что они будут хорошо различимы на заранее заданном фоне всех без исключения содержащих их элементов. Но не перестарайтесь: чрезмерно большое количество цветов изрядно утомляет и делает документ малопривлекательным для пользователей. К тому же замена стандартной расцветки гиперссылок, скорее всего, сделает их трудноразличимыми на фоне остальных текстовых элементов. (Например, назначив одинаковый цвет гиперссылкам и основному тексту документа, вы гарантированно скроете их для большей части аудитории, даже несмотря на обязательное подчеркивание.)

В CSS можно изменять как основной цвет элемента, так и цвет его фона. Чтобы понять их назначение, нужно разобраться с тем, какие компоненты элемента окрашиваются основным цветом, а какие не входят в сферу его воздействия. В общем случае он определяет цвет текста и границ элемента. Следовательно, существуют два способа непосредственного задания основного цвета: с помощью свойства `color` и одного из свойств настройки границ элемента.

Основной цвет

Самый простой способ определения основного цвета элемента заключается в использовании свойства `color`.

color	
Значение	*<color>*
Начальное значение	Определяется пользовательским агентом
Применяется	Все элементы
Вычисляется	Согласно значению
Наследуется	Да
Анимируется	Да

Это свойство принимает любые значения, представляющие цвета в допустимом для CSS формате, например `#FFCC00` или `rgba(100%, 80%,0%,0.5)`.

У незамещаемых элементов, подобных `em` или `p`, свойство `color` определяет цвет текста, как показано в следующем примере, результат разметки которого приведен на рис. 9.1.

```
<p style="color: gray;">Этому абзацу назначен серый цвет переднего
плана</p>
<p>Этот абзац имеет цвет переднего плана, заданный по умолчанию</p>
```

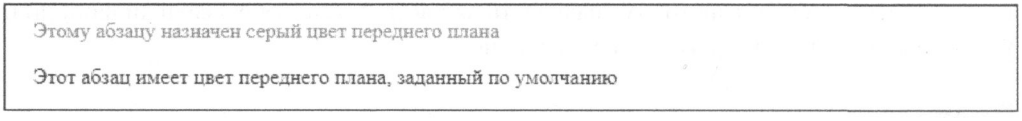


Рис. 9.1. Объявленный в свойстве `color` и стандартный цвета текста

Как показано на рис. 9.1, по умолчанию текст имеет черный цвет. Он может отличаться в браузерах, в которых применяются нестандартные пользовательские настройки. В частности, если в пользовательском агенте по умолчанию выбран цвет текста `green` (зеленый), то текст второго абзаца приведенного выше примера будет отображаться зеленым, а не черным цветом. При этом текст первого абзаца будет оставаться серым.

У свойства `color` более широкая область применения, чем назначение одного только цвета текста элемента. Он может использоваться для акцентирования внимания на отдельных текстовых фрагментах, например содержащих важные предупреждения. Чтобы назначить тексту отдельных элементов красный цвет, необходимо отнести их к обособленному классу (`<p class="warn">`), а затем использовать такое правило:

```
p.warn {color: red;}
```

В этом же документе можно назначить зеленый цвет всем непосещенным гиперссылкам.

```
p.warn {color: red;}
p.warn a:link {color: green;}
```

Впоследствии, в случае изменения общего дизайна документа, текст предупреждений можно быстро изменить, скажем, на темно-красный, а цвет гиперссылок — на фиолетовый. Чтобы воплотить в жизнь новые дизайнерские концепции, достаточно заменить в стилевых правилах одни только значения цветовых свойств. Результат переоформления документа показан на рис. 9.2.

```
p.warn {color: #600;}
p.warn a:link {color: #400040;}
```

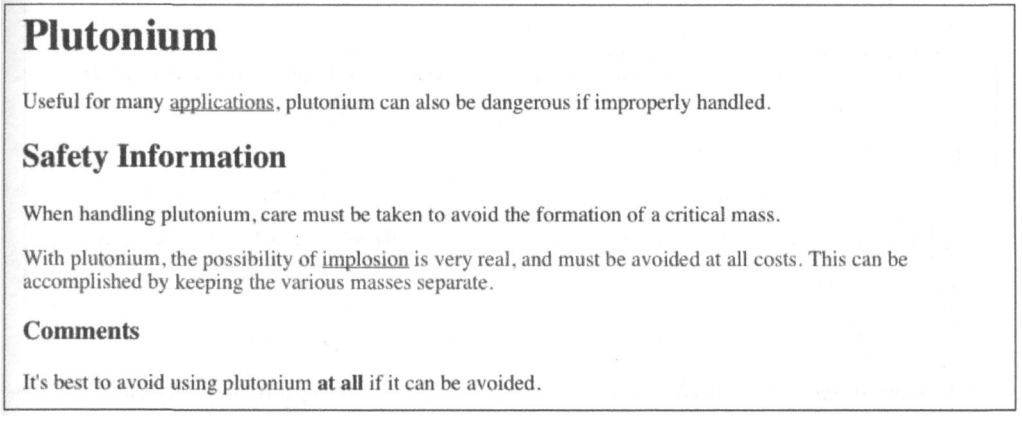


Рис. 9.2. Замена цвета текста у элементов (см. цветные иллюстрации на веб-сайте)

Свойство `color` также позволяет привлекать внимание к акцентированному тексту документа. В случаях, когда полужирное начертание применяется в документе повсеместно, для еще большего выделения акцентированного текста его нужно окрасить цветом, отличающимся от заданного основному тексту документа, например темно-бордовым (`maroon`):

```
b, strong {color: maroon;}
```

Кроме того, если это предполагает дизайн документа, то ячейкам таблицы класса `highlight` можно назначить светло-желтый основной цвет:

```
td.highlight {color: #FF9;}
```

Если тексту документа не назначить фоновый цвет в явном виде, то существует опасность нарушения общей концепции оформления документа вследствие изменения пользовательских настроек браузера. Например, при задании в браузере бледно-желтого фонового цвета (`#FFC`) текст в ячейках указанного выше класса будет различаться недопустимо плохо. Это же касается и белого фонового цвета, на котором желтый текст выглядит более чем неуместно. Таким образом, основной и фоновый цвета нужно устанавливать комплексно, в связке друг с другом. (О назначении фонового цвета рассказывается далее.)

Форматирование границ основным цветом

Значение свойства `color` самым непосредственным образом влияет на цвет границ элемента. Рассмотрим случай объявления следующего стилевого форматирования, результат которого показан на рис. 9.3:

```
p.aside {color: gray; border-style: solid;}
```

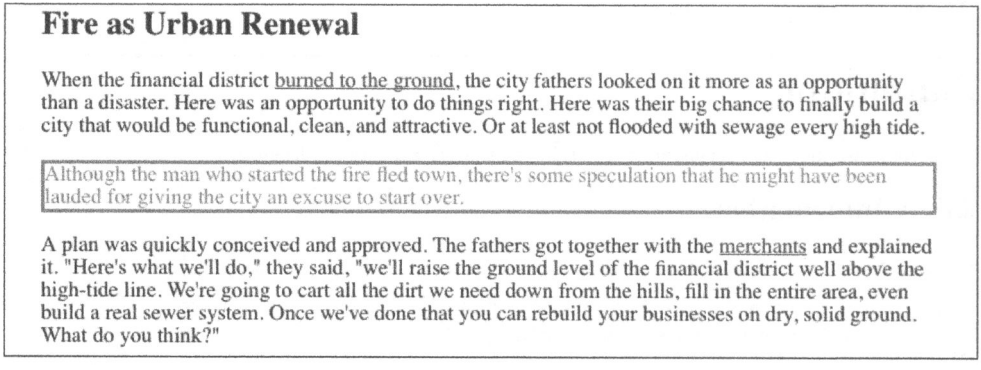


Рис. 9.3. Цвет границ заимствован у содержимого элемента

Элемент, представленный тегом `<p class="aside">`, содержит серый текст, заключенный в такого же цвета рамку средней толщины. Для изменения ее расцветки применяется стилевое правило, включающее объявление свойства `border-color`:

```
p.aside {color: gray; border-style: solid; border-color: black;}
```

Его применение обеспечивает сохранность серого цвета текста и представление границ элемента абсолютно черным цветом — свойство `border-color` замещает свойство `color`.

Для явного указания в стилевых правилах основного цвета служит значение `currentColor`. Оно представляет вычисляемое значение свойства `color` элемента. Следовательно, встроенный стиль форматирования границ цветом по умолчанию рассматривается пользовательским агентом следующим образом:

```
* {border-color: currentColor;}
```

Если границам элемента не назначить цвет, но определить стиль, то рамка элемента будет представляться в цвете, передаваемом значением свойства `color`. При назначении цвета границ в явном виде указанное значение будет превалировать над значением `currentColor`, объявляемым во встроенном стиле.

Эта особенность форматирования границ позволяет использовать основной цвет для окрашивания рамок, в которые заключаются изображения документа. Поскольку изображения состоят из разноцветных пикселей, свойство `color` не оказывает на них сколь-нибудь заметного влияния, тем не менее задавая для его границ цвет по умолчанию. Исходя из применимости для установки цвета границ сразу двух свойств — `color` и `border-color`, оба следующих стилевых правила, которые определяют рамки для изображений, относящихся к классам `type1` и `type2`, приведут к одинаковому результату (рис. 9.4).

```
img.type1 {border-style: solid; color: gray;}
img.type2 {border-style: solid; border-color: gray;}
```

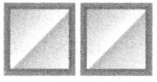


Рис. 9.4. Границы одинакового цвета, определенные разными стилевыми правилами

Форматирование элементов управления основным цветом

Изменение значения свойства `color` должно незамедлительно сказываться на форматировании элементов управления форм (по крайней мере, согласно требованиям спецификации CSS). В частности, для назначения тексту элемента `select` серого цвета можно использовать такое стилевое правило:

```
select {color: rgb(33%,33%,33%);}
```

Оно также должно устанавливать цвет границ, отображаемых вокруг элемента `select`. В действительности конечное форматирование элемента определяется внутренними стилями браузера, которые далеко не всегда отвечают требованиям спецификации CSS.

При назначении основного цвета элементам `input` он будет применяться сразу ко всем типам элементов управления — от текстового поля до раскрывающихся списков, кнопок, переключателей и флажков, как показано на рис. 9.5.

```
select {color: rgb(33%,33%,33%);}
input {color: red;}
```

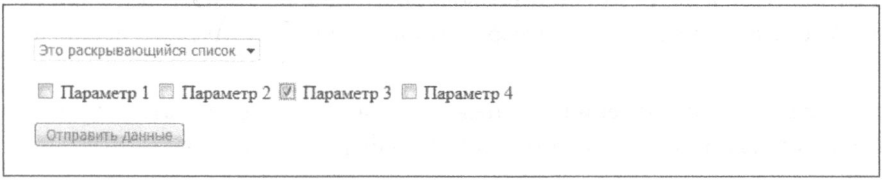


Рис. 9.5. Результат изменения основного цвета элементов управления формы (см. цветные иллюстрации на веб-сайте)

Обратите внимание на то, что подписи к флажкам, показанные на рис. 9.5, представлены черным основным цветом. А все потому, что стилевые правила применяются только к элементам управления `input` и `select`, но не к их тексту.

Черным цветом также отображаются символы отметок (галочки), выставляемые в полях элемента `input`. Такое форматирование свидетельствует о том, что пользовательский агент обрабатывает элементы управления согласно шаблонам, предоставляемым операционной системой. Не стоит забывать, что в большинстве случаев стандартные элементы управления не встраиваются в HTML-документ, а представляются графическими элементами пользовательского интерфейса, добавляемыми в него так же, как и внешние изображения. В действительности элементы управления форм, как и изображения, являются замещаемыми элементами HTML-документа и (чисто теоретически) не подлежат стилевому форматированию.

В реальности границы элементов, показанные на рис. 9.5, выглядят несколько светлее. При этом основным цветом может представляться как весь элемент управления, так и только отдельные его составляющие. Таким образом, внешний вид элементов управления, заимствованных у операционной системы, зависит от пользовательского агента, в котором отображается документ. Вывод очень прост: элементы управления форм очень сложно поддаются стилевому форматированию, и к их применению в документах нужно подходить предельно взвешенно.

Наследование основного цвета

Согласно определению свойства `color`, основной цвет элемента наследуется. Такое поведение вполне оправданно, так как объявление `p {color: gray;}` предполагает окрашивание серым цветом всего текста абзаца, в том числе акцентированного и сильно акцентированного. Для представления вложенных элементов иными цветами к ним нужно применять отдельные стилевые правила, как показано на рис. 9.6.

```
em {color: red;}
p {color: gray;}
```

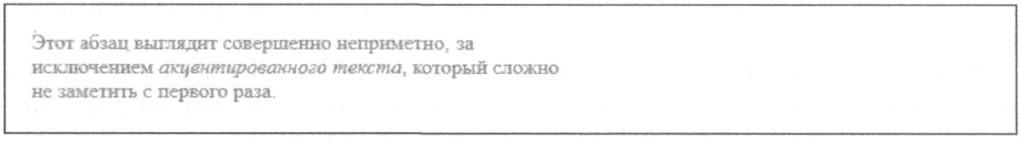


Рис. 9.6. Выделение цветом отдельных элементов абзаца (см. цветные иллюстрации на веб-сайте)

Наследуемость основного цвета позволяет установить единый цвет для всего регулярного текста документа, например с помощью такого стилевого правила:

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

В результате его применения красным цветом будет отображаться весь текст документа, не имеющий специального цветового форматирования.

Фон элемента

По умолчанию фон элемента распространяется на область, занятую содержимым элемента, вплоть до внешних краев его границ. Следовательно, границы элемента визуализируются поверх фона, распространяющегося в область полей элемента. (О том, как изменить область, занимаемую фоном элемента, рассказывается в следующих разделах.)

В CSS фон элемента можно представить сплошным цветом. Можно также использовать в качестве фона графическое изображение и даже линейный или радиальный градиент.

Фоновый цвет

Для назначения элементу фонового цвета применяется свойство `background-color`, которое может иметь любые допустимые в CSS значения, представляющие цвет.

background-color	
Значение	*<color>*
Начальное значение	`transparent`
Применяется	Все элементы
Вычисляется	Согласно значению
Наследуется	Нет
Анимируется	Да

Чтобы расширить фон за пределы текста элемента, добавьте к нему отступы, как показано на рис. 9.7. Следующий пример получен в результате выполнения такого кода.

```
p {background-color: #AEA;}
p.padded {padding: 1em;}

<p>Абзац</p>
<p class="padded">Абзац с полями</p>
```

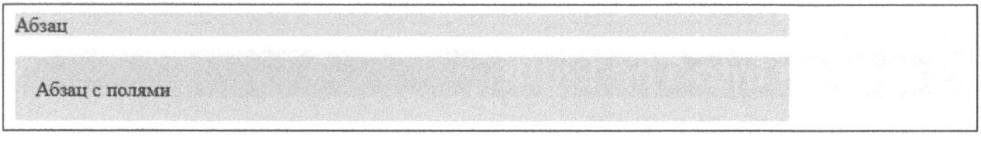


Рис. 9.7. Фон в области полей (см. цветные иллюстрации на веб-сайте)

Фон задается для любого элемента, начиная с `body` и заканчивая строчными элементами, такими как `em` и `a`. Свойство `background-color` не наследуется и по умолчанию имеет значение `transparent`. Это и понятно: если фон элемента специально не указан, то сквозь него должен просматриваться фон расположенного ниже (родительского) элемента.

Чтобы лучше понять, что такое прозрачный фон, обратимся к следующей аналогии. Представим изображение, нанесенное на прозрачную пластиковую основу, которое закреплено на стене с узорчатыми обоями. Обои будут просматриваться через пластиковую основу, но они являются фоном стены, а не изображения (в терминологии CSS). Подобным образом, если страница документа имеет собственный фон, то он будет просматриваться через все его элементы, не имеющие собственного, специально заданного, фона. Это важный момент: фон страницы не наследуется потомками, а *просматривается сквозь* них. На первый взгляд, разница невелика, но она становится важной при назначении элементам фоновых изображений (см. далее).

По большей части ключевое слово `transparent` остается невостребованным, поскольку прозрачный фон устанавливается по умолчанию. Но иногда без него просто не обойтись. Представим, что в настройках браузера указано отображать все гиперссылки на белом фоне. Но при создании веб-страницы гиперссылки часто представляются основным белым цветом, а потому белый фон у них будет просто неуместен.

Чтобы обеспечить преобладание явно заданного фона над фоном, устанавливаемым браузером, необходимо использовать такое стилевое правило:

```
a {color: white; background-color: transparent;}
```

Если не указывать фон в явном виде, то белый основной цвет будет сливаться с белым фоном, и гиперссылки останутся неразличимыми. На реальных веб-страницах такой дизайн встречается редко, но все же он вероятен.

Высокая вероятность возникновения конфликтных ситуаций становится причиной появления на экране сообщения "You have no `background-color` with your `color`" (Свойство `color` назначено без объявления свойства `background-color`), генерируемого модулем проверки подлинности CSS. Оно указывает на несостоятельность стилевого форматирования при изменении пользователями настроек браузера, заданных по умолчанию. Это предупреждение не свидетельствует о неправильности стилевых правил, а всего лишь сообщает о том, что их действие может быть нарушено.

Специальные эффекты

Комбинируя в одном правиле свойства `color` и `background-color`, можно добиться весьма интересных эффектов.

Один из примеров приведен на рис. 9.8.

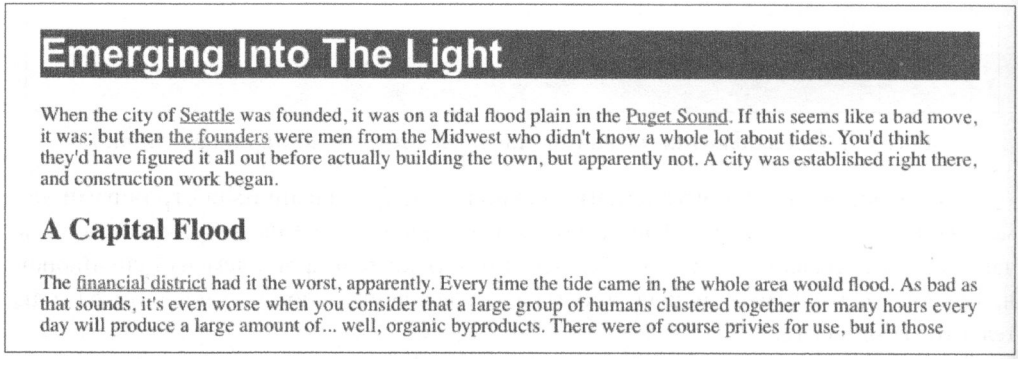


Рис. 9.8. Цветовое инвертирование заголовка первого уровня

Цветовых комбинаций столько же, сколько цветов в палитре, — бесконечное множество, и приводить их все просто бессмысленно. Достаточно заранее продумывать, какой именно эффект требуется получить.

Стилевое форматирование следующего примера (рис. 9.9) несколько сложнее, хотя и не представляет больших трудностей для понимания.

```
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;}
```

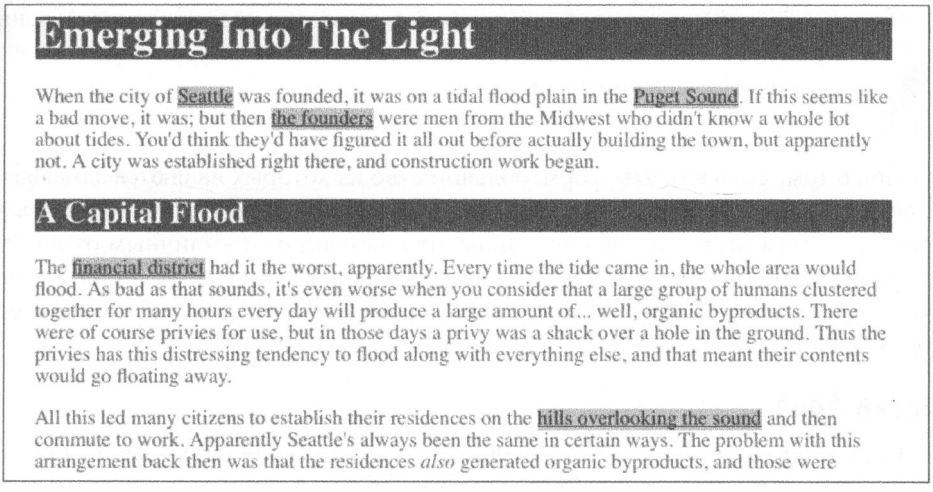


Рис. 9.9. Более сложное стилевое форматирование (см. цветные иллюстрации на веб-сайте)

Возникает справедливый вопрос: а что происходит, когда фон назначается замещаемому элементу, например изображению? Мы сейчас не затрагиваем тему полноцветных изображений формата GIF87a и PNG, включающих прозрачные области. Предположим, что нужно добавить двухцветную рамку к JPEG-изображению. Эту задачу легко решить, снабдив элемент узкими полями и цветным фоном (рис. 9.10), обуславливаемыми приведенным ниже стилевым правилом.

```
img.twotone {background-color: red; padding: 5px;
             border: 5px solid gold;}
```

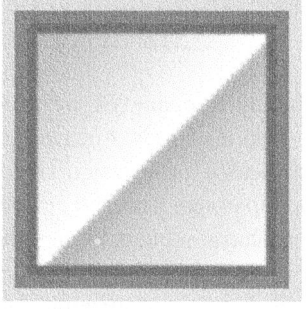


Рис. 9.10. Двухцветная рамка, образованная фоном и одноцветными границами (см. цветные иллюстрации на веб-сайте)

С технической точки зрения фон распространяется до внешнего края границ, поскольку в данном случае они сплошные и непрерывные, и он сквозь них не просматривается. Фон виден только через область узких полей, имеющих однопиксельную ширину, образуя внутреннюю часть двухцветной рамки. Этот прием может применяться для получения более совершенных обрамлений, в которых используются в том числе и фоновые градиенты.

В CSS для добавления к элементам рисованных рамок, представленных произвольными изображениями, применяются более совершенные инструменты, описанные в главе 8.

Помня о том, что элементы форм, большинство из которых являются замещаемыми элементами, плохо поддаются стилевому форматированию, следует заметить, что добавление к ним отступов с фоном приводит к результатам, отличным от получаемых при назначении их незамещаемым элементам, таким как изображения. Поэтому будьте предельно аккуратны, добавляя к ним фоновый цвет, и не забывайте всесторонне тестировать получаемый результат в самых разных окружениях.

Обрезка фона

В предыдущих разделах фон занимал всю отведенную для него область элемента. Как известно, он распространяется до внешних краев границ, просвечиваясь через прозрачные области рамки (в частности, пунктирные или точечные). Тем не менее в CSS добавлено специальное свойство — `background-clip`, отвечающее за ограничение области распространения фона элемента.

background-clip

Значение	[`border-box` | `padding-box` | `content-box` | `text`]#
Начальное значение	`border-box`
Применяется	Все элементы
Вычисляется	Согласно значению
Наследуется	Нет
Анимируется	Нет

По умолчанию этому свойству передается исторически обусловленное значение, обозначающее *область рисования фона*, предельный размер которой ограничен внешними краями границ элемента. При его использовании фон либо отсутствует, либо просвечивается сквозь прозрачные участки границ элемента.

Значение `padding-box` предполагает распространение фона только до внешнего края полей (внутреннего края границ) элемента и отсутствие его под его границами. С другой стороны, значение `content-box` применяется для добавления фона только к области содержимого элемента.

Назначение описанных выше ключевых слов проиллюстрировано на рис. 9.11, полученном в результате выполнении следующего CSS-кода.

```
div[id] {color: navy; background: silver; padding: 1em;
         border: 5px dashed;}
#ex01 {background-clip: border-box;}  /* значение по умолчанию */
#ex02 {background-clip: padding-box;}
#ex03 {background-clip: content-box;}
```

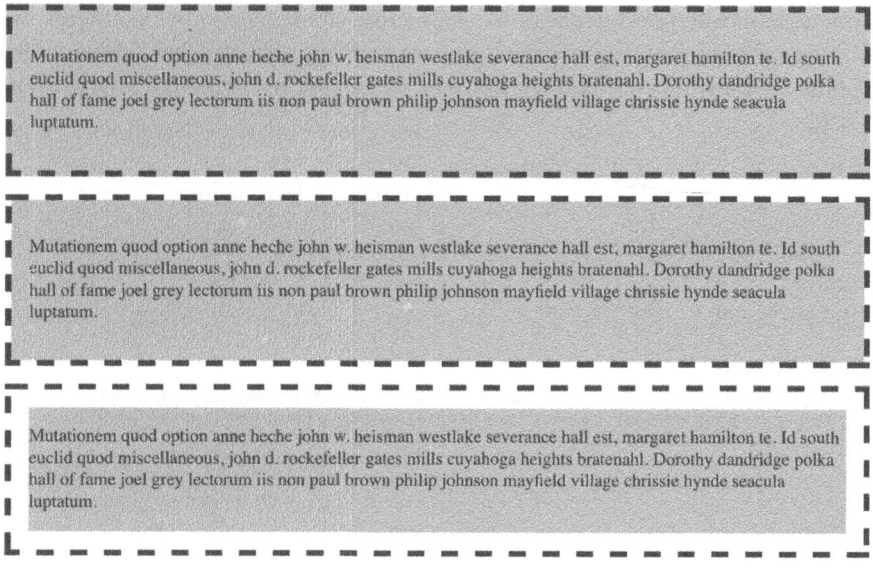


Рис. 9.11. Три способа добавления фона к элементу

Все три настройки однозначны в исполнении, хотя и имеют свои особенности применения. Первая из них заключается в невозможности применения свойства `background-clip` к корневому элементу (`html` или `body` в HTML, в зависимости от способа определения стилевого форматирования). Таким образом, область рисования фона корневого элемента, в отличие от остальных элементов документа, обрезке не подлежит.

Вторая особенность применения свойства `background-clip` связана с влиянием на размер области фона скругленных углов рамки, образованной границами элемента. Это вполне оправданное решение, поскольку фон элемента не должен выступать за края скругленных углов элемента, полностью находясь внутри рамки. В подобных случаях углы области рисования фона, определяемой свойством `background-clip`, скругляются по дуге такой же формы, как и рамка элемента.

Следующая особенность не позволяет свойству `background-clip` корректно обрезать фон, представленный отдельными ключевыми словами свойства `background-repeat`.

Четвертая особенность обусловливается функциональным назначением свойства `background-clip`. Оно указывает область обрезки фона без привязки к значениям остальных свойств его настройки. При заполнении фона сплошным цветом эта особенность никак не проявляется, а вот результат обрезки фонов, представляемых разными графическими изображением (см. следующий раздел), может существенно отличаться.

Последнее значение, `text`, применяется для ограничения области фона контурами текстовых символов. Иными словами, его передача свойству `background-clip` вызывает заливку фоном только текста — остальная часть элемента остается прозрачной.

Чтобы познакомиться с последним эффектом, необходимо отключить в элементе основной цвет. В противном случае он будет окрашивать текст и блокировать просмотр фона. Рассмотрим наглядный пример, результат применения которого показан на рис. 9.12.

```
div {color: rgb(255,0,0); background: rgb(0,0,255);
     padding: 0 1em; margin: 1.5em 1em; border: 5px dashed;
     font-weight: bold;}
#ex01 {background-clip: text; color: transparent;}
#ex02 {background-clip: text; color: rgba(255,0,0,0.5);}
#ex03 {background-clip: text;}
```

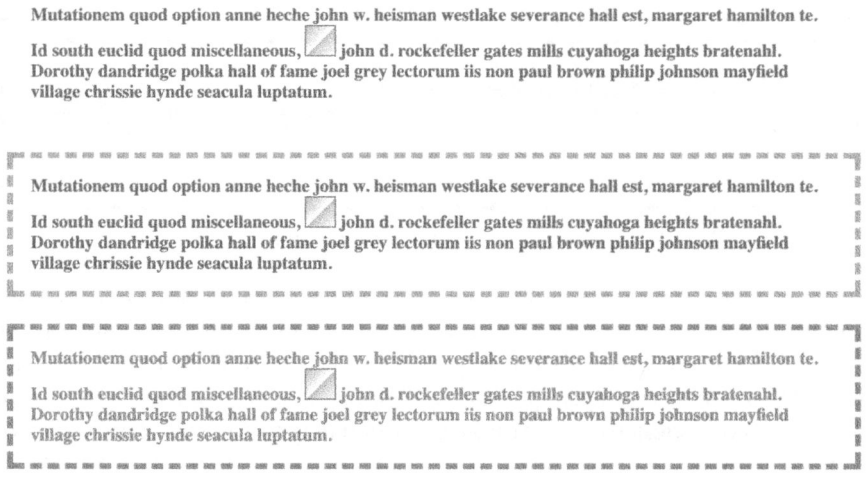


Рис. 9.12. Обрезка фона по контурам текста (см. цветные иллюстрации на веб-сайте)

В первом примере, приведенном на рис. 9.12, основной цвет элемента полностью прозрачный, поэтому синий фоновый цвет просматривается через все его текстовые символы, но не через изображение, вставленное в абзац, поскольку его основной цвет представлен ключевым словом, отличным от `transparent`.

Во втором примере основной цвет передается значением `rgba(255,0,0,0.5)`, представляющим полупрозрачный красный оттенок. Смешиваясь с синим фоновым цветом, текст элемента приобретает фиолетовый оттенок. С другой стороны, полупрозрачные красные границы смешиваются с белым фоном, приобретая молочно-красный цвет.

В третьем примере применяется сплошной красный основной цвет. Следовательно, текст и границы полностью красные, и фоновый синий цвет через них не просматривается. Так как фон обрезается по контурам текста, он полностью закрыт цветом переднего плана.

Описанная выше методика применима для всех типов фонов, включая градиентные и представленные внешними графическими изображениями. Не забывайте, что в случае сбоев при загрузке фонового изображения прозрачный текст будет оставаться полностью неразличимым.

К концу 2017 года объявление `background-clip: text` корректно обрабатывалось только в Firefox. Остальные браузеры (как и сам Firefox) поддерживали работу с вендорным свойством `-webkit-background-clip: text`.

Фоновое изображение

Ознакомившись с принципами стилевой настройки основного и фонового цветов, можно переходить к изучению более сложных приемов оформления элементов, связанных с применением фоновых изображений. Во времена HTML3.2 для использования в качестве фона документа графического изображения его нужно было объявить в атрибуте `BACKGROUND` элемента `BODY`.

```
<BODY BACKGROUND="bg23.gif">
```

В результате такого действия браузер загружал файл `bg23.gif` и "вымащивал" им фон, заполняя копиями изображения всю доступную область фона. Данный эффект можно воссоздать с помощью разметки CSS, обладающей более совершенными инструментами управления фоновыми изображениями. Их рассмотрение нужно начать с описания самых простых свойств и только затем перейти к изучению более сложного материала.

Подключение фонового изображения

Для указания изображения, применяемого для заполнения фона элемента, используется свойство `background-image`.

background-image

Значение	[*<image>*]# | none
Начальное значение	`none`
Применяется	Все элементы
Вычисляется	Согласно значению, но с абсолютным URL
Наследуется	Нет
Анимируется	Нет

<image> = [*<uri>* | *<linear-gradient>* | *<repeating-linear-gradient>* | *<radial-gradient>* | *<repeating-radial-gradient>*]

Значение `none`, задаваемое свойству по умолчанию, полностью соответствует названию: элемент не получает фонового изображения. Чтобы назначить элементу фоновое изображение, необходимо объявить свойство `background-image` с одним из других разрешенных спецификацией значений.

```
body {background-image: url(bg23.gif);}
```

Согласно приведенному выше правилу, если остальные свойства настройки фона установлены в значения по умолчанию, то фон изображения будет представлять

файл `bg23.gif` (рис. 9.13). Но как вы вскоре узнаете, такой вариант, несмотря на свою очевидность, не является самодостаточным.

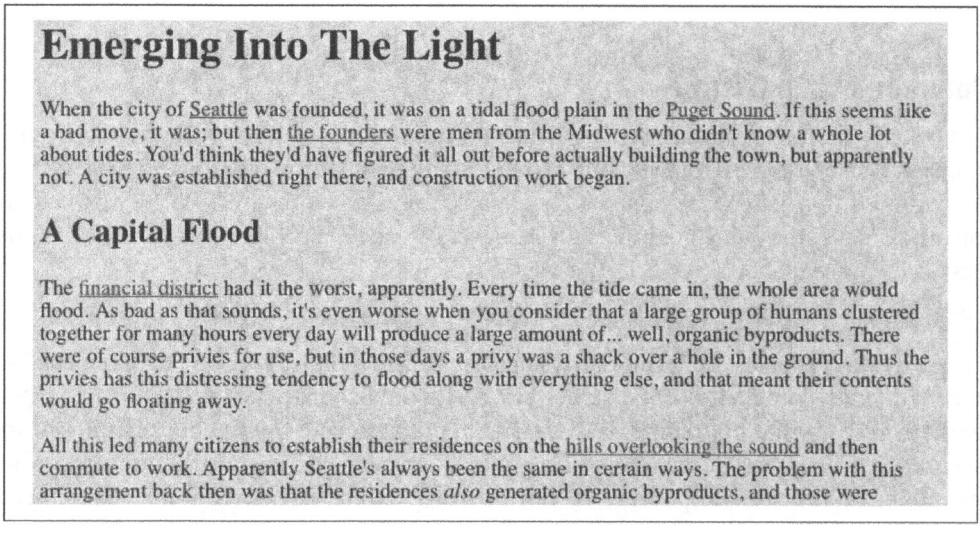


Рис. 9.13. Назначение фонового изображения с помощью стилевого правила

Хорошей идеей, наряду с фоновым изображением, будет назначить элементу фоновый цвет. (О дублировании фонового изображения сплошной цветовой заливкой, а также о назначении элементу сразу двух фоновых изображений рассказано в последующих разделах, а здесь описаны методы управления всего одним фоновым изображением.)

Фоновое изображение можно назначить любому изображению документа как строчного, так и блочного уровня.

```
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...</p>
```

На рис. 9.14 приведен результат применения фонового изображения только к абзацу. Учтите, что спецификация CSS разрешает применять фоновые изображения к одним только строчным элементам, например гиперссылкам, что также показано на рис. 9.14. Чтобы получить "плиточный" узор, нужно подобрать фоновое изображение небольшого размера — заметно меньше размера символов.

Фоновые изображения применяются в документе с самыми разными целями. Если добавить такой фон к элементу `strong`, то представленный им фрагмент сильно акцентированного текста будет выделяться в документе еще сильнее. Фоновые изображения также часто применяются для выделения тематических заголовков.

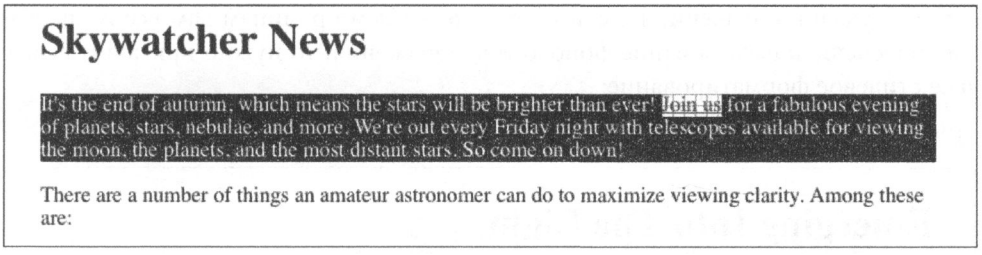


Рис. 9.14. Применение фонового изображения к блочному и строчному элементам

Добавив фоновые значки к элементам, обладающим специальными функциональными возможностями, можно однозначно указать их назначение в документе — документ Word, файл PDF, почтовый адрес или другой ресурс, как продемонстрировано на рис. 9.15 (получен при использовании приведенного ниже стилевого форматирования).

```
a[href] {padding-left: 1em; background-repeat: no-repeat;}
a[href$=".pdf"] {background-image: url(/i/pdf-icon.png);}
a[href$=".doc"] {background-image: url(/i/msword-icon.png);}
a[href^="mailto:"] {background-image: url(/i/email-icon.png);}
```

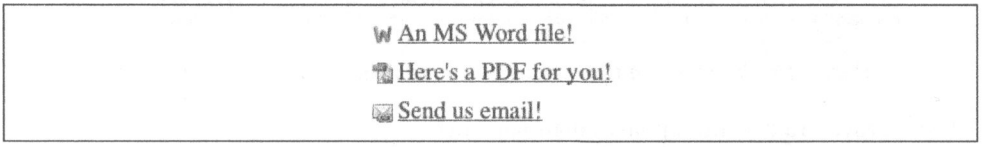


Рис. 9.15. Добавление к гиперссылкам фоновых значков

Свойство `background-image`, как и `background-color`, не наследуется. Наследованию также не подлежит ни одно другое свойство настройки фона элементов. Не забывайте, что расположение файла фонового изображения нужно указывать с учетом требований, выдвигаемых к значениям свойства `url()`. Также учтите, что относительные URL-адреса всегда указываются для папки расположения таблицы стилей, а не целевого документа.

Отказ от наследования фона

Ранее неоднократно упоминалось о том, что фон не наследуется элементами иерархической структуры документа. Понять, почему было принято такое решение, можно, проанализировав результат назначения элементам фонов, представленных графическими изображениями. Представим ситуацию, когда фоновое изображение, исходно назначаемое элементу `body`, наследуется его дочерними элементами. Такой фон будет добавляться во все без исключения элементы документа, заполняя каждый из них отдельно от остальных (рис. 9.16).

Заполнение фона элементов (включая гиперссылки) графическим изображением начинается с левого верхнего угла. Такой способ назначения фона элементам нельзя назвать абсолютно правильным, как и наследование значений свойств настройки

фоном — удачным решением. Если по определенным причинам вам все же необходимо обеспечить наследование фонового изображения, то нужно применить следующее стилевое форматирование:

```
* {background-image: url(yinyang.png);}
```

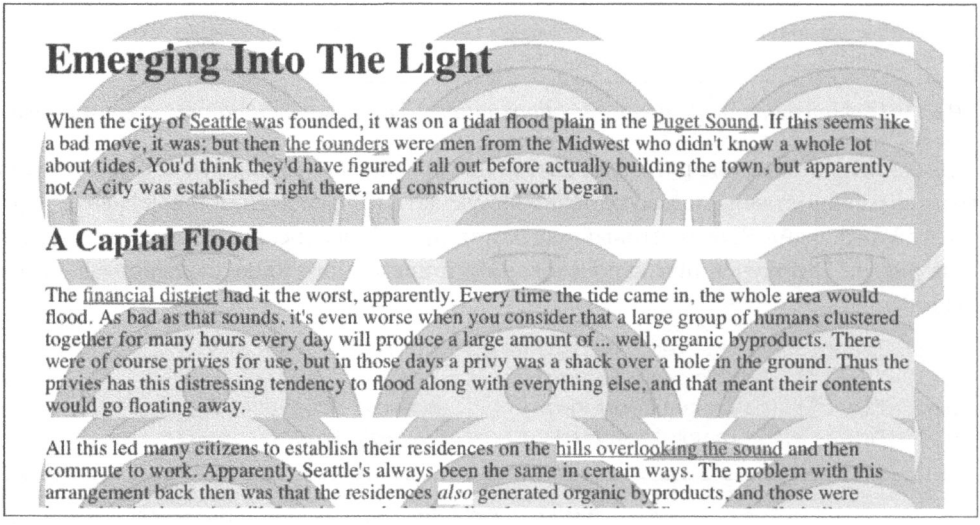


Рис. 9.16. Результат применения к документу наследуемого фона

Существует также альтернативный вариант:

```
body {background-image: url(yinyang.png);}
* {background-image: inherit;}
```

Правильные способы добавления фонового изображения

Фоновые изображения всегда располагаются поверх фоновой заливки сплошным цветом. Ее наличие не играет особой роли при использовании непрозрачного фонового изображения, сохраненного в формате JPEG, поскольку его копии заполняют всю область рисования фона, не оставляя свободного пространства для проступания фоновой заливки. В случае применения изображения, снабженного альфа-каналом, например формата PNG или SVG, с его прозрачными областями, совмещаемыми с цветной фоновой заливкой, приходится считаться. К тому же в случаях сбоев при загрузке изображения фон элемента будет представляться одной только цветовой заливкой. Оцените, насколько блекло выглядит текстовый абзац без расположенного под ним тематического изображения (рис. 9.17).

На рис. 9.17 прекрасно видно, что абзац, лишенный не только фонового изображения, но и фоновой заливки, выглядит более чем непрезентабельно. Таким образом, чтобы получить приемлемый результат, элементу, снабженному фоновым изображением, в обязательном порядке нужно задавать цветовую заливку.

```
p.starry {background-image: url(http://www.site.web/pix/stars.gif);
          background-color: black; 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 fabulous evening of planets, stars, nebulae, and more...</p>
```

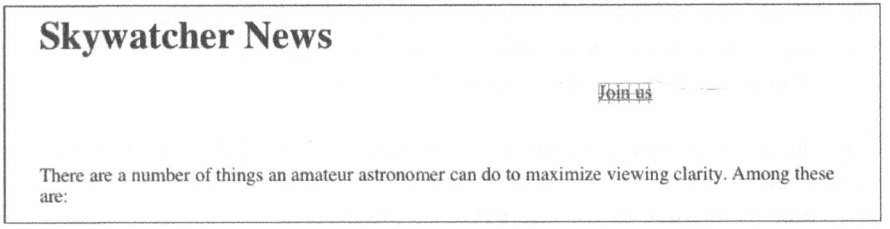


Рис. 9.17. Последствие сбоя в загрузке фонового изображения

Определенное выше стилевое форматирование указывает заполнять фон черным матовым цветом в случаях недоступности фонового изображения по указанному адресу. При загрузке фонового изображения черная заливка будет заполнять его прозрачные области, а также части фона, не заполненные им (о причинах такого поведения фонового изображения рассказано далее).

Позиционирование фона

Теперь, когда вы научились добавлять фоновые изображения к элементам, можно переходить к изучению способов позиционирования фона. За позиционирование фона в элементе отвечает стилевое свойство `background-position`.

background-position

Значение	*<position>*#
Начальное значение	`0% 0%`
Применяется	Элементы блочного уровня и незамещаемые элементы
Процентное значение	Относительно размеров элемента и исходного изображения; см. раздел «Процентные значения»
Вычисляется	Как абсолютное смещение в единицах измерения длины или как процентное значение
Наследуется	Нет
Анимируется	Да

<position> = [[`left` | `center` | `right` | `top` | `bottom` | *<percentage>* | *<length>*] | [`left` | `center` | `right` | *<percentage>* | *<length>*] [`top` | `center` | `bottom` | *<percentage>* | <length>] | [center | [left | right] [<percentage> | <length>]?] && [`center` | [`top` | `bottom`] [*<percentage>* | *<length>*]?]]

Синтаксис значения этого свойства выглядит пугающе, но не является таковым по своей сути — непомерно большая длина записи обусловлена попыткой формализации в высшей степени неэффективного способа включения новых возможностей в и без того неоптимизированный старый синтаксис. Несмотря на запутанность определения, применять свойство `background-position` в стилевых правилах очень просто.

При рассмотрении стилевых правил этого раздела предполагается, что свойство `background-repeat` установлено в значение `no-repeat`. Поскольку это свойство еще не рассматривалось, на данном этапе достаточно знать, что значение `no-repeat` обязывает добавлять на фон только один экземпляр исходного изображения, предотвращая повторение его копий по всей области рисования фона.

В качестве примера рассмотрим, как позиционируется фоновое изображение, добавляемое к элементу `body`. Результат этой операции, выполняемой с помощью приведенных ниже стилевых правил, показан на рис. 9.18.

```
body {background-image: url(yinyang.png);
      background-repeat: no-repeat;
      background-position: center;}
```

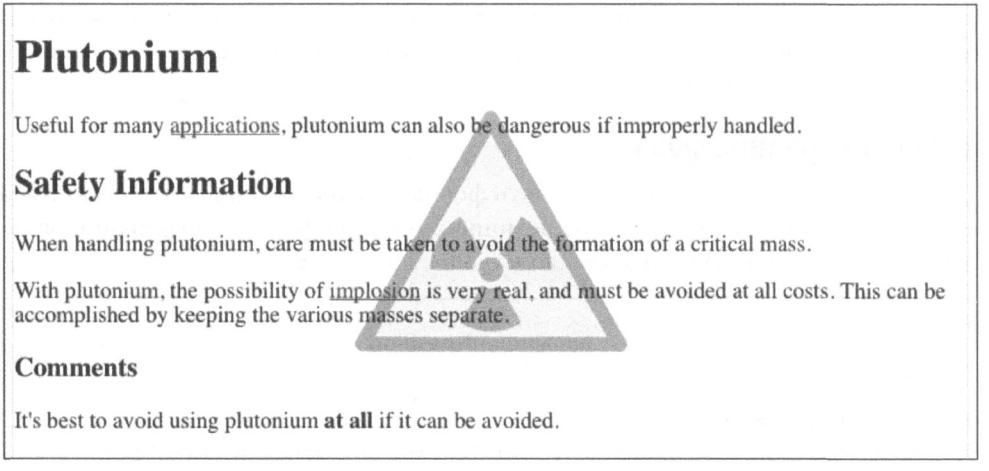


Рис. 9.18. Выравнивание фонового изображения по центру элемента

Единственный экземпляр фонового изображения располагается по центру страницы — для предотвращения заполнения его копиями применяется свойство `background-repeat` (описано в следующей главе). Любой фон заполняется или "вымащивается" экземплярами одного и того же изображения, называемого *исходным изображением*.

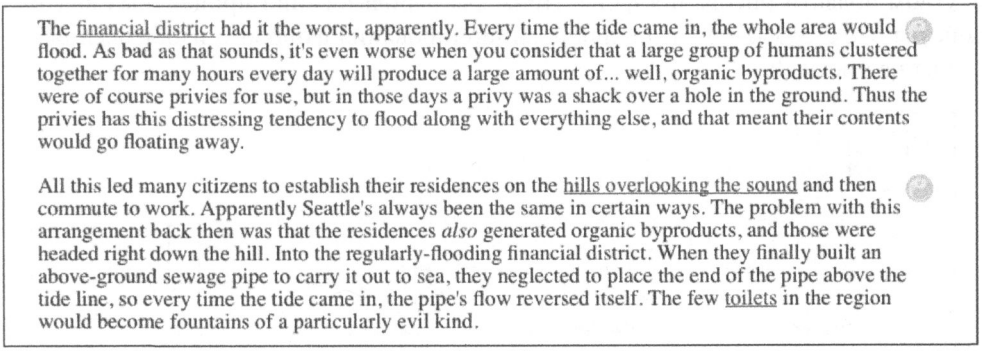
Таким образом, свойство `background-position` устанавливает положение исходного изображения на фоне элемента. Существует несколько способов передачи значений этому свойству. Во-первых, с помощью ключевых слов `top`, `bottom`, `left`, `right` и `center`, которые обычно указываются попарно (как в предыдущем примере), хотя это не обязательно. Во-вторых, значение может передаваться в виде числового значения, выраженного в единицах измерения длины, например `50px` или `2cm`. Наконец, последний способ предполагает представление значения свойства `background-position` процентным значением, таким как 43%. Каждый тип значений оказывает свое воздействие на положение исходного изображения на фоне элемента.

Ключевые слова

Позиционирование фонового изображения с помощью ключевых слов — это самый доступный способ из всех разрешенных спецификацией CSS. Передаваемый эффект однозначно описывается названием значений, поэтому трудностей с их идентификацией возникать не должно. Например, значение `top right` указывает расположить исходное изображение в правом верхнем углу фона элемента. Рассмотрим, как с его помощью позиционируется небольшой символ "инь-янь".

```
p {background-image: url(yinyang-sm.png);
   background-repeat: no-repeat;
   background-position: top right;}
```

В результате предложенного стилевого форматирования исходное изображение послушно занимает место в правом верхнем углу области рисования фона каждого абзаца (рис. 9.19). Кстати говоря, такой же результат будет наблюдаться при передаче свойству `background-position` значения `right top`.


The financial district had it the worst, apparently. Every time the tide came in, the whole area would flood. As bad as that sounds, it's even worse when you consider that a large group of humans clustered together for many hours every day will produce a large amount of... well, organic byproducts. There were of course privies for use, but in those days a privy was a shack over a hole in the ground. Thus the privies has this distressing tendency to flood along with everything else, and that meant their contents would go floating away.

All this led many citizens to establish their residences on the hills overlooking the sound and then commute to work. Apparently Seattle's always been the same in certain ways. The problem with this arrangement back then was that the residences *also* generated organic byproducts, and those were headed right down the hill. Into the regularly-flooding financial district. When they finally built an above-ground sewage pipe to carry it out to sea, they neglected to place the end of the pipe above the tide line, so every time the tide came in, the pipe's flow reversed itself. The few toilets in the region would become fountains of a particularly evil kind.

Рис. 9.19. Размещение фонового изображения в правом верхнем углу каждого абзаца

Ключевые слова, устанавливающие позиционирование фонового изображения, если их не более двух, могут перечисляться в любой последовательности. В подобных случаях одно ключевое слово указывает положение по горизонтали, а другое — по вертикали. При передаче свойству `background-position` сразу двух "вертикальных" (`top top`) или двух "горизонтальных" (`right right`) ключевых слов его объявление полностью игнорируется.

Если передать всего одно ключевое слово, то позиционирование в противоположном направлении будет передаваться ключевым словом `center`. Таким образом, чтобы поместить фоновое изображение по центру верхней части абзаца, нужно применить такое стилевое правило.

```
p {background-image: url(yinyang-sm.png);
   background-repeat: no-repeat;
   background-position: top;}
```

Процентные значения

С формальной точки зрения процентные значения представляют такие же действия, как и ключевые слова, но обладают более широкой областью применения.

Предположим, что с их помощью нужно центрировать исходное изображение на фоне элемента.

```
p {background-image: url(chrome.jpg);background-repeat: no-repeat;
   background-position: 50% 50%;}
```

В результате применения приведенного выше правила исходное изображение располагается по центру фона элемента. Другими словами, процентное значение применяется не только к исходному изображению, но и к целевому элементу.

Чтобы понять, каким образом процентное значение свойства `background-position` изменяет форматирование элемента, процесс его применения нужно рассмотреть на более детальном уровне. Рассмотрим случай позиционирования исходного изображения с помощью составного значения `50% 50%` — его центр выравнивается с центром фона элемента. При этом значение `0% 0%` указывает совмещать левый верхний угол изображения с таким же углом области рисования фона элемента. Значение `100% 100%` обеспечивает выравнивание правого нижнего угла изображения с таким же углом области рисования фона. Примеры использования процентных значений свойства `background-position` для позиционирования фонового изображения приведены на рис. 9.20.

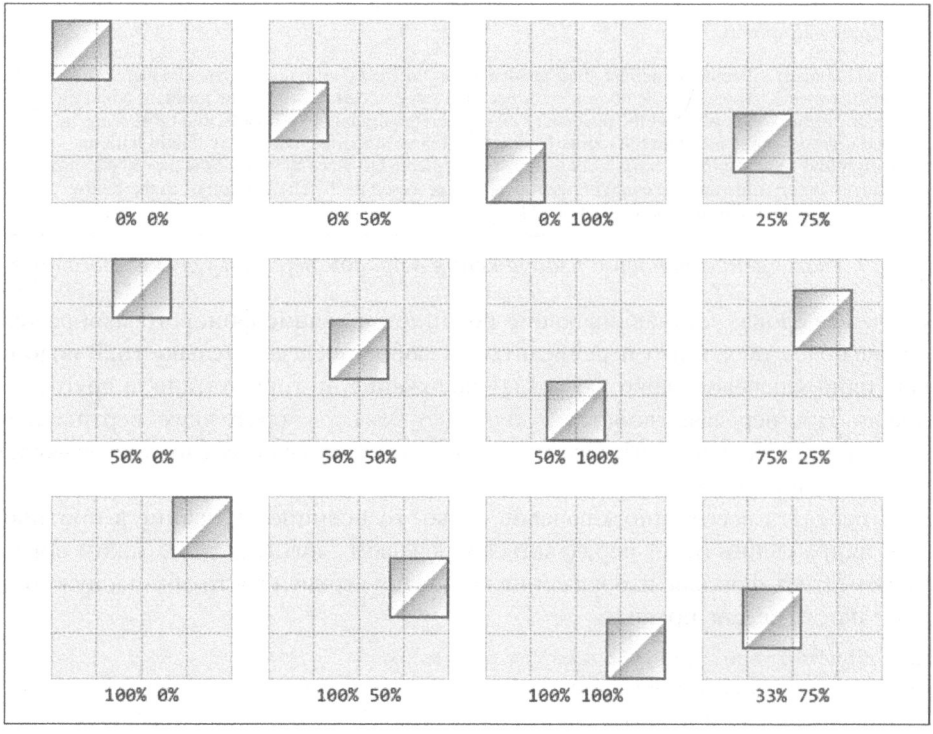


Рис. 9.20. Позиционирование фонового изображения с помощью процентных значений

Таким образом, следующее правило обеспечивает смещение исходного изображения на треть ширины и две трети высоты фона элемента.

```
p {background-image: url(yinyang-sm.png);
   background-repeat: no-repeat;
   background-position: 33% 66%;}
```

Позиционирование, устанавливаемое приведенным выше правилом, выполняется так, что точка изображения, отстоящая от ее левого верхнего угла на треть ширины и две трети высоты, смещается на треть ширины фона вправо и на две трети высоты его фона — вниз. Не забывайте, что в процентном значении первым *всегда* указывается горизонтальная размерность. Если в предыдущем правиле поменять местами процентные величины, то исходное изображение будет смещаться на две трети ширины и треть высоты фона изображения.

При передаче свойству всего одного процентного значения оно будет указывать смещение изображения по горизонтали, а вертикальное смещение составит 50%.

```
p {background-image: url(yinyang-sm.png);
   background-repeat: no-repeat;
   background-position: 25%;}
```

В результате применения такого правила исходное изображение смещается на четверть ширины вправо и центрируется по вертикали (рис. 9.21).

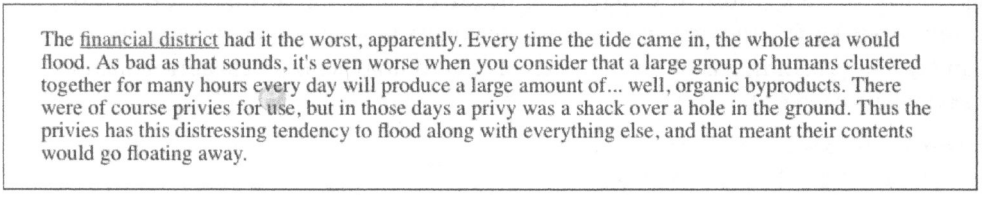


Рис. 9.21. Выравнивание исходного изображения по центру высоты фона при передаче свойству `background-position` *только одного значения*

Соответствие процентных значений ключевым словам, передаваемым свойству `background-position`, приведено в табл. 9.1.

Таблица 9.1. Ключевые слова и процентные значения, применяемые при позиционировании фоновых изображений

Ключевые слова	Полное значение	Процентное значение
`center`	`center center`	`50% 50%` `50%`
`right`	`center right` `right center`	`100% 50%` `100%`
`left`	`center left` `left center`	`0% 50%` `0%`
`top`	`top center` `center top`	`50% 0%`
`bottom`	`bottom center` `center bottom`	`50% 100%`
`top left`	`left top`	`0% 0%`

Окончание табл. 9.1

Ключевые слова	Полное значение	Процентное значение
`top right`	`right top`	`100% 0%`
`bottom right`	`right bottom`	`100% 100%`
`bottom left`	`left bottom`	`0% 100%`

По умолчанию свойство `background-position` представлено значением `0% 0%`, указывающим размещать исходное изображение в левом верхнем углу (`top left`) фона элемента. Именно оно определяет позиционирование фоновых изображений в отсутствие специально заданного стилевого форматирования.

Числовые значения в единицах длины

Последними будут рассматриваться числовые значения, выраженные в единицах измерения длины. При передаче свойству `background-position` они указывают расстояние смещения, отсчитанное от левого верхнего угла фона. При этом точка смещения всегда располагается в левом верхнем углу исходного изображения. Например, при установке свойства `background-position` в значение `20px 30px` левый верхний угол исходного изображения будет смещен на 20 пикселей вправо и на 30 пикселей вниз относительно левого верхнего угла фона элемента, как показано (наряду с другими примерами) на рис. 9.22. Оно получено в результате выполнения следующего кода CSS.

```
background-image: url(chrome.jpg);
background-repeat: no-repeat;
background-position: 20px 30px;
```

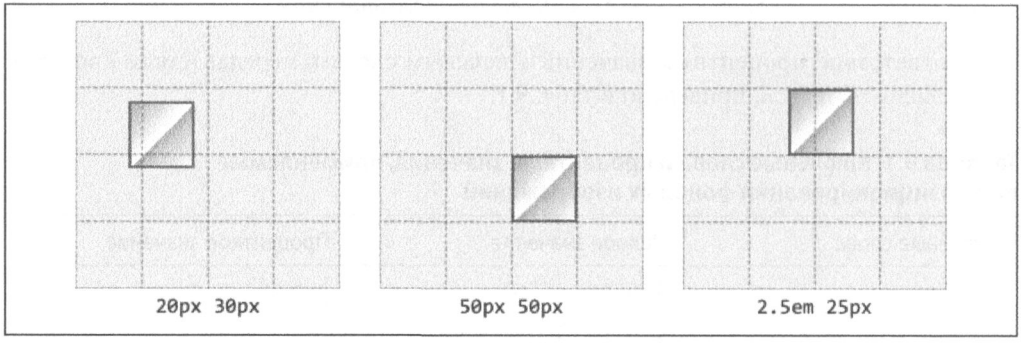


Рис. 9.22. Смещение фонового изображения при передаче свойству `background-position` числовых значений, представленных в единицах измерения длины

Как видите, поведение свойства при передаче ему процентных и числовых значений, выраженных в единицах измерения длины, несколько отличается. В последнем случае положение точки исходного изображения, для которой рассчитывается смещение, не зависит от числовых величин, передаваемых свойству `background-position`.

Оптимальные решения достигаются при использовании в одном значении числовых значений обоих типов. Рассмотрим пример, в рамках которого фоновое изображение нужно выровнять по правому краю элемента и сместить на 10 пикселей вниз относительно его верхнего края (рис. 9.23). Как и в любых других случаях, первым стилевому свойству `background-position` передается значение, определяющее горизонтальное смещение.

```
p {background-image: url(yinyang.png);
   background-repeat: no-repeat;
   background-position: 100% 10px;
   border: 1px dotted gray;}
```

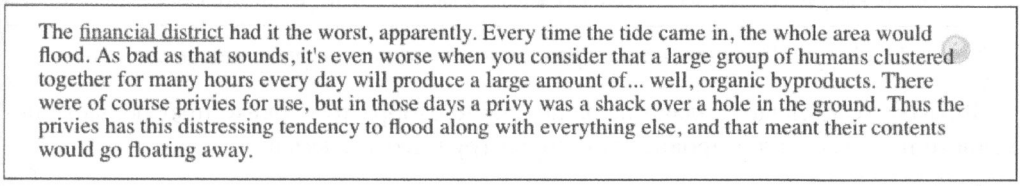


Рис. 9.23. Позиционирование фонового изображения с помощью значений разных типов

Результат, равнозначный представленному на рис. 9.23, можно получить с помощью значения `right 10px`, заменив процентное значение ключевым словом. Помните о том, что произвольный порядок указания значений допускается только при передаче свойству `background-position` ключевых слов. При использовании процентных значений и значений, представленных в единицах длины, первым *всегда* устанавливается смещение вдоль горизонтальной оси, а вертикальное смещение всегда задается вторым значением. Следовательно, значение `right 10px` нельзя заменить значением `10px right`. Последнее объявление будет проигнорировано пользовательским агентом (ключевое слово `right` не относится к значениям, определяющим вертикальное смещение).

Отрицательные значения

Процентные значения и значения, выраженные в единицах измерения длины, могут представляться отрицательными числами: в подобных случаях исходное изображение выходит за пределы фона элемента. Проиллюстрируем этот эффект на примере уже используемого ранее изображения символа "инь-янь" большого размера, размещаемого по центру области рисования фона элемента. Предположим, что теперь исходное изображение нужно сместить так, чтобы в левом верхнем углу элемента отображалась только небольшая его часть.

Пусть исходное изображение имеет размер 300×300 пикселей. Также предположим, что в левом верхнем углу будет отображаться область его правого нижнего угла, ширина и высота которой составляют треть от исходных значений. Нужный эффект (рис. 9.24) достигается при выполнении следующего правила.

```
body {background-image: url(yinyang.png);
      background-repeat: no-repeat;
      background-position: -200px -200px;}
```

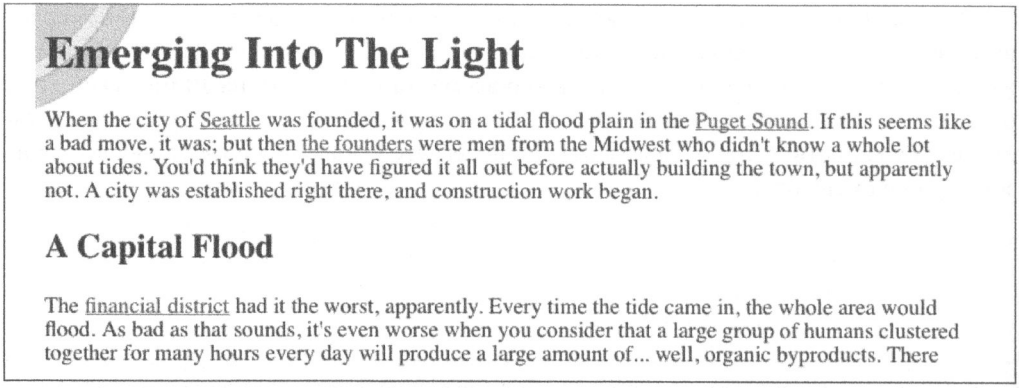


Рис. 9.24. Смещение фонового изображения с помощью отрицательных значений

В рамках следующей задачи на фоне нужно отобразить только правую часть исходного изображения, выровненную по центру высоты элемента.

```
body {background-image: url(yinyang.png);
      background-repeat: no-repeat;
      background-position: -150px 50%;}
```

Свойству `background-position` можно также передавать отрицательные процентные значения, хотя их применение приводит к совершенно неожиданным результатам. Рассмотрим ситуацию, в которой размеры элементов и исходных изображений, размещаемых в области рисования их фонов, сильно отличаются. На рис. 9.25 показан результат применения к такому документу следующих стилевых правил.

```
p {background-image: url(pix/yinyang.png);
   background-repeat: no-repeat;
   background-position: -10% -10%;
   width: 500px;}
```

Последнее правило указывает смещать исходное изображение за внешние края фона на расстояние, составляющее 10% от его высоты и ширины. Поскольку размеры изображения составляют 300×300 пикселей, точка смещения будет отстоять от его левого верхнего угла на 30 пикселей влево и вверх (вычисляемое значение `-30px -30px`). Ширина абзацев составляет `500px`, поэтому изображение будет смещаться за левый край фона на 50 пикселей. С учетом положения точки смещения левый край каждого исходного изображения будет располагаться левее левого края области рисования фона на 20 пикселей (точка смещения, исходно находящаяся на расстоянии `-30px`, сдвигается в положение `-50px`). Разница между указанными величинами как раз и составляет 20 пикселей.

Так как все абзацы имеют разную величину, вертикальное смещение фоновых изображений у всех них разное. Если высота области рисования фона составляет 300 пикселей, то верхний край исходного изображения будет совпадать с верхним краем фона элемента — в данном случае точка смещения находится ниже верхнего края фона как раз на 30 пикселей. При высоте абзаца всего `50px` точка смещения бу-

дет располагаться ниже верхнего края изображения на 5 пикселей (`-5px`), а потому будет опускаться на 25 пикселей вниз (*ниже* верхнего края области рисования фона). Таким образом, чем ниже абзац, тем на большее расстояние смещается вниз его фоновое изображение (см. рис. 9.25).

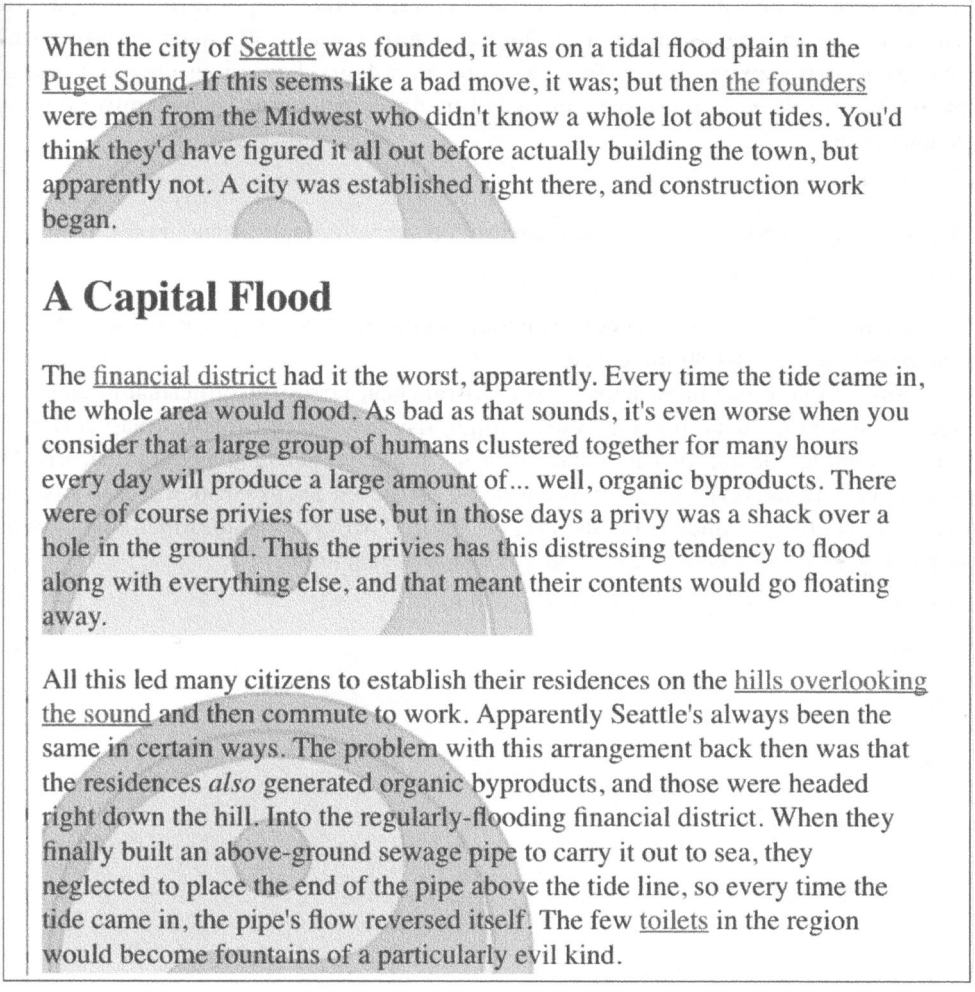


Рис. 9.25. Разные результаты, получаемые при передаче свойству `background-position` *отрицательных процентных значений*

Изменение исходной точки смещения

До этого момента мы рассматривали только упрощенные задачи по позиционированию фоновых изображений. При их выполнении предполагалось, что свойство `background-position` может иметь не более двух значений, а смещение изображения отсчитывается исключительно от левого верхнего угла области рисования фона.

Такие предположения в полной мере соответствуют спецификации предыдущей версии CSS. Последняя редакция спецификации позволяет указывать угол контейнера элемента, относительно которого смещается фоновое изображение, а потому обеспечивает поддержку свойством `background-position` сразу четырех значений.

Знакомство с новыми возможностями начнем с такого простого примера: сместим исходное изображение на треть ширины вправо и на 30 пикселей вниз относительно левого верхнего угла области рисования фона. Если пользоваться старым вариантом синтаксиса, то поставленная задача будет решаться следующим простым объявлением:

```
background-position: 33% 30px;
```

При передаче свойству четырех значений объявление принимает такой вид:

```
background-position: left 33% top 30px;
```

Дословно оно указывает "сместить изображение от левого края фона на 33%, а от его верхнего края — на 30 пикселей".

Разобравшись с базовым вариантом синтаксиса, который описывает смещение относительно угла, заданного по умолчанию, изучим более сложные варианты позиционирования исходного изображения на фоне элемента. Представим, что его нужно сместить на четверть ширины и на 30 пикселей вверх относительно правого нижнего угла области рисования фона (рис. 9.26). В отсутствие повторения фонового изображения эта задача решается следующим объявлением:

```
background-position: right 25% bottom 30px;
```

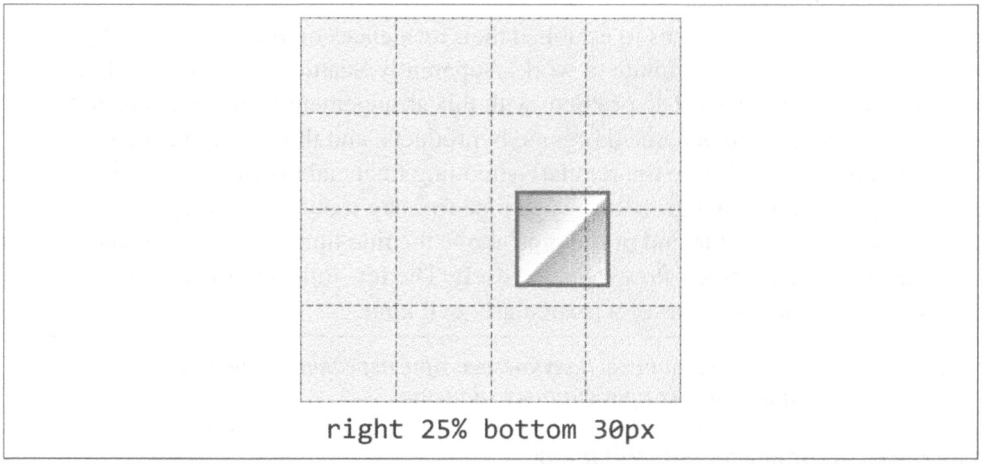


Рис. 9.26. Изменение точки, относительно которой выполняется смещение исходного изображения

Опять-таки, дословно это объявление указывает "сместить исходное изображение от правого края фона на 25%, а от его нижнего края — на 30 пикселей".

Таким образом, свойство `background-position` получает значения, порядок передачи которых задается таким шаблоном: ключевое слово, определяющее край смещения, расстояние смещения, ключевое слово, определяющее край смещения, расстояние смещения. Порядок указания горизонтального и вертикального смещений не играет особой роли: объявление `bottom 30px right 25%` в полной мере равнозначно объявлению `right 25% bottom 30px`. Тем не менее опускать ключевые слова в значениях свойства недопустимо — объявление `30px right 25%` считается неправильным и будет проигнорировано пользовательским агентом.

Значение смещения допускается не указывать только в одном случае: когда оно равно нулю. Следовательно, объявление `right bottom 30px` указывает на выравнивание исходного изображения по правому краю и смещение его на 30 пикселей вверх относительно нижнего края области рисования фона. При этом объявление `right 25% bottom` предполагает смещение изображения на четверть ширины влево относительно правого края и выравнивание его по нижнему краю фона (рис. 9.27).

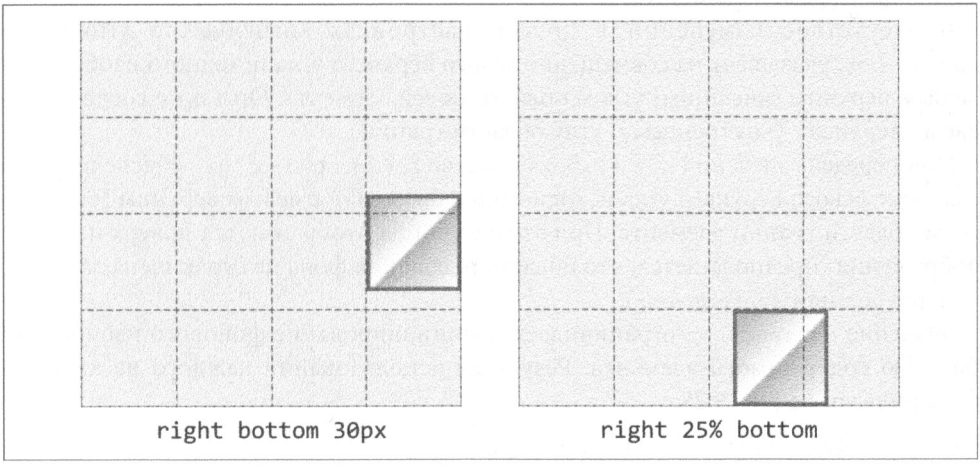


Рис. 9.27. Объявления с неполными значениями

Учтите, что в полной форме записи смещение должно указываться относительно горизонтальных или вертикальных границ области рисования фона — объявление формата `center 25% center 25px` будет проигнорировано браузером.

Область позиционирования фона

На данный момент вам должно быть известно, каким образом к элементу добавляется фоновое изображение и как оно позиционируется в области рисования фона. Остается открытым вопрос: можно ли рассчитывать размер и положение фонового изображения относительно области, отличаающейся от заданной по умолчанию (образуемой полями и границами элемента)? Оказывается, можно! В CSS область позиционирования фонового рисунка задается с помощью свойства `background-origin`.

background-origin	
Значение	[border-box | padding-box | content-box]#
Начальное значение	padding-box
Применяется	Все элементы
Вычисляется	Согласно определению
Наследуется	Нет
Анимируется	Нет

Функционально оно подобно свойству `background-clip`, но с более четко обозначенной областью назначения. В частности, свойство `background-origin` определяет область, относительно краев которой позиционируется исходное изображение. Она называется областью позиционирования фона, в отличие от области рисования фона, определяемой свойством `background-clip`.

В отсутствие изменений в других настройках значение по умолчанию, `padding-box`, указывает на совмещение левого верхнего угла исходного изображения с левым верхним (внешним) углом области полей элемента. Он также соответствует левому верхнему (внутреннему) углу области границ.

При передаче значения `border-box` свойство `background-origin` обеспечивает совмещение левого верхнего угла исходного изображения с левым верхним (внешним) углом области границ элемента. При этом границы отображаются поверх исходного изображения (предполагается, что область рисования фона не ограничена значением `padding-box` или `content-box`).

Значение `content-box` ограничивает позиционирование фонового изображения областью содержимого элемента. Результат использования каждого из ключевых слов приведен на рис. 9.28.

```
div[id] {color: navy; background: silver;
   background-image: url(yinyang.png);
   background-repeat: no-repeat;
   padding: 1em; border: 5px dashed;}
#ex01 {background-origin: border-box;}
#ex02 {background-origin: padding-box;} /* по умолчанию */
#ex03 {background-origin: content-box;}
```

Не забывайте, что совмещение левых верхних углов изображения и области позиционирования выполняется только при установке свойства `background-position` в значение по умолчанию. Во всех остальных случаях местоположение фонового изображения определяется относительно области, указываемой свойством `background-origin`: внешние края границ, полей или области содержимого. Чтобы удостовериться в этом, изучите результат альтернативного способа форматирования HTML-документа (рис. 9.29), рассмотренного в предыдущем примере.

```
div[id] {color: navy; background: silver;
   background-image: url(yinyang);
   background-repeat: no-repeat;
```

```
   background-position: bottom right;
   padding: 1em; border: 5px dotted;}
#ex01 {background-origin: border-box;}
#ex02 {background-origin: padding-box;} /* по умолчанию */
#ex03 {background-origin: content-box;}
```

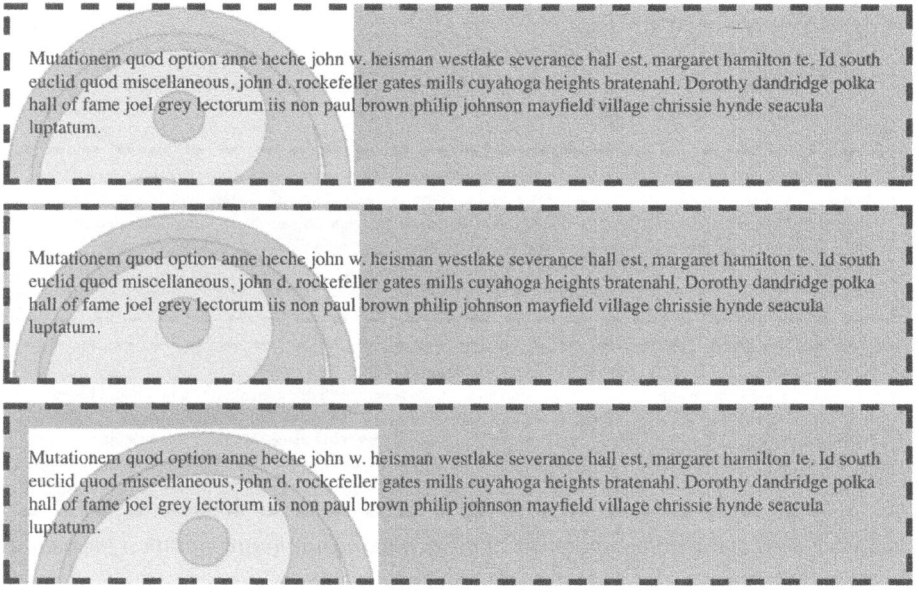


Рис. 9.28. Три способа позиционирования фонового изображения

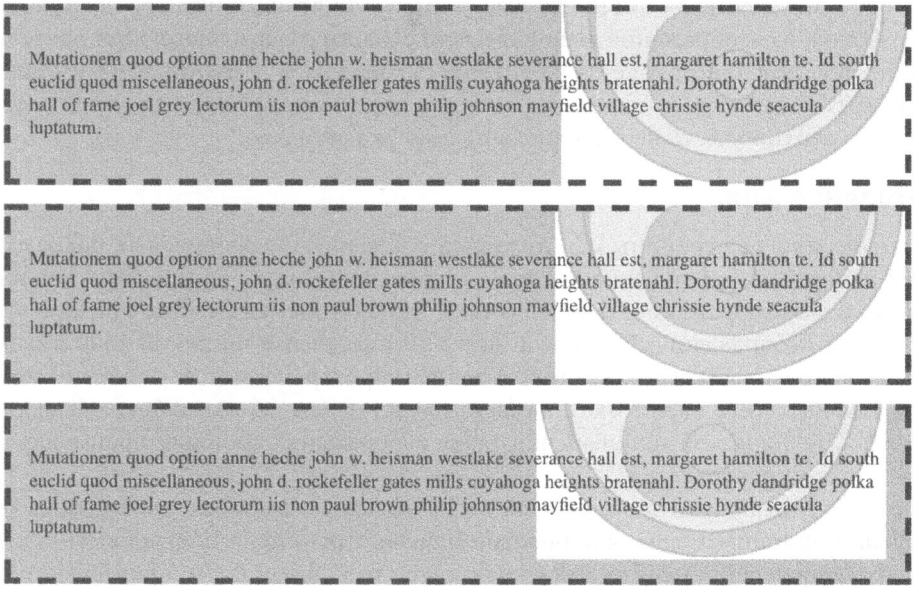


Рис. 9.29. Три способа позиционирования фонового изображения (обновлено)

Самое интересное начинается тогда, когда и область позиционирования, и область рисования фона устанавливаются в явном виде. Например, область позиционирования фона может быть ограничена полями, а его область рисования — содержимым элемента (или наоборот). Результат подобного способа форматирования документа приведен на рис. 9.30 и получен с помощью следующих стилевых правил.

```
#ex01 {background-origin: padding-box;
       background-clip: content-box;}
#ex02 {background-origin: content-box;
       background-clip: padding-box;}
```

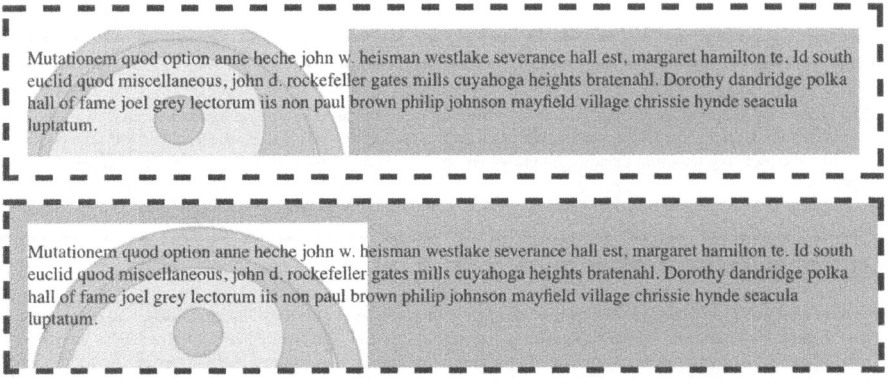


Рис. 9.30. Фон элементов при разных областях позиционирования и рисования

В первом примере (см. рис. 9.29) исходное изображение усекается по своим краям, поскольку позиционируется по области полей, но при этом область рисования фона обрезается по краям области содержимого. Во втором примере позиционирование фонового изображения выполнятся по области содержимого, хотя область рисования фона распространяется до внешних краев полей элемента. В результате оно отображается полностью, вплоть до нижнего края области полей, даже несмотря на то что его верхняя часть не выровнена по ее верхнему краю.

Повторение фона

Раньше многие страницы оформлялись с помощью своеобразных боковых фоновых панелей, для получения которых требовалось создавать чрезвычайно широкие исходные изображения. Дошло до того, что фон элементов стал представляться исходными изображениями высотой всего 10 пикселей и шириной 1500 пикселей, большая часть которых была занята свободным пространством, а "полезная" область имела ширину всего около 100 пикселей (представляла те самые боковые фоновые панели). С технической точки зрения такие изображения состояли преимущественно из пустого пространства.

Не проще ли получить описанный выше эффект, создав исходное изображения небольшой ширины только для боковой панели страницы и повторив его в вертикальном направлении необходимое количество раз? Такой подход намного проще в реализации и заметно уменьшает скорость загрузки документа с сервера. Чтобы добиться нужного результата, нам понадобится свойство `background-repeat`.

background-repeat

Значение	*<repeat-style>#*
Шаблон	***<repeat-style>*** = `repeat-x` | `repeat-y` | [`repeat` | `space` | `round` | `no-repeat`]{1,2}
Начальное значение	`repeat`
Применяется	Все элементы
Вычисляется	Согласно определению
Наследуется	Нет
Анимируется	Нет

При первом знакомстве синтаксис значения этого свойства выглядит запутанно, но при детальном рассмотрении оказывается очень простым. В его основе лежат четыре ключевых слова: `repeat`, `norepeat`, `space` и `round`. Оставшиеся два ключевых слова, `repeat-x` и `repeat-y`, представляют комбинации базовых значений, как показано в табл. 9.2.

При передаче свойству `background-repeat` двух ключевых слов первое из них задает повторение в горизонтальном, а второе — в вертикальном направлении. Если свойство принимает только одно ключевое слово, то оно указывает повторение сразу для обоих направлений (за исключением ключевых слов `repeat-x` и `repeat-y`, как показано в табл. 9.2).

Таблица 9.2. Сопоставление значений свойства `background-repeat`

Ключевое слово	Комбинация ключевых слов
`repeat-x`	`repeat no-repeat`
`repeat-y`	`no-repeat repeat`
`repeat`	`repeat repeat`
`no-repeat`	`no-repeat no-repeat`
`space`	`space space`
`round`	`round round`

Как легко догадаться, ключевое слово `repeat` указывает повторять изображение бесконечное количество раз в любом направлении. Таким поведением снабжались фоновые изображения в самой первой спецификации CSS, поддерживающей работу с ними. Значения `repeat-x` и `repeat-y` определяют повторение изображения соответственно в горизонтальном и вертикальном направлениях, а значение `no-repeat` запрещает его повторение на фоне элемента.

По умолчанию заполнение фона элемента копиями исходного изображения начинается с его левого верхнего угла. В результате применение следующего правила приводит к получению форматирования, показанного на рис. 9.31.

```
body {background-image: url(yinyang-sm.png);
      background-repeat: repeat-y;}
```

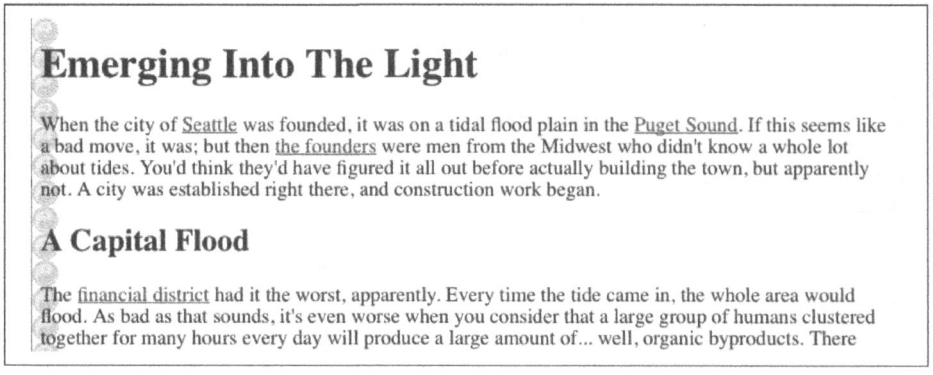


Рис. 9.31. Повторение исходного изображения в вертикальном направлении

Рассмотрим, каким образом можно добиться повторения фонового изображения вдоль верхнего края документа. Вместо того чтобы создавать отдельное изображение для такого случая, достаточно внести небольшие изменения в приведенное выше правило.

```
body {background-image: url(yinyang-sm.png);
      background-repeat: repeat-x;}
```

Результат применения этого правила, заключающийся в повторении исходного изображения (в данном случае начиная от левого верхнего угла фона элемента `body`) в горизонтальном направлении, представлен на рис. 9.32.

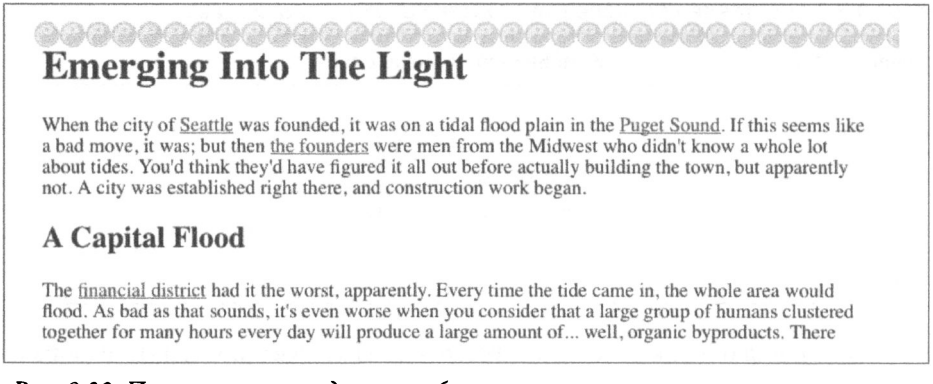


Рис. 9.32. Повторение исходного изображения в горизонтальном направлении

При необходимости от повторения изображения можно отказаться полностью, передав свойству ключевое слово `no-repeat`.

```
body {background-image: url(yinyang-sm.png);
      background-repeat: no-repeat;}
```

При использовании небольших исходных изображений эффект ключевого слова `no-repeat` кажется незначительным, поскольку оно теряется в левом верхнем углу документа. Рассмотрим, как будет выглядеть такое же форматирование при исходном изображении намного большего размера (рис. 9.33).

```
body {background-image: url(yinyang.png);
      background-repeat: no-repeat;}
```

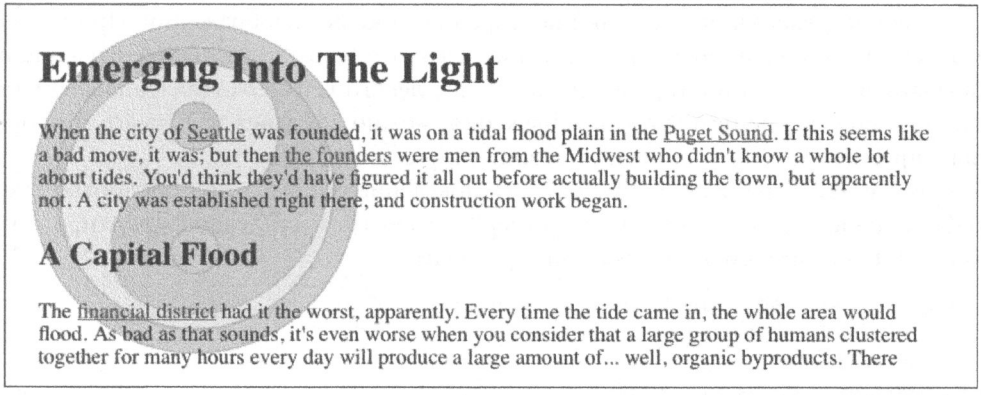


Рис. 9.33. Размещение на фоне всего одного экземпляра исходного изображения

Возможность установки направления повторения исходного изображения позволяет добиться необычайно интересных эффектов. Предположим, что к левому краю элемента `h1` нужно добавить тройную границу. В дополнение верхний край элемента `h2` необходимо снабдить волнистой границей, символизирующей морские волны. Исходное изображение нужно оформить в таких же цветах, что и фоновая заливка, как показано на рис. 9.34. Требуемый эффект достигается с помощью следующих стилевых правил.

```
h1 {background-image: url(triplebor.gif); background-repeat: repeat-y;}
h2 {background-image: url(wavybord.gif); background-repeat: repeat-x;
    background-color: #CCC;}
```

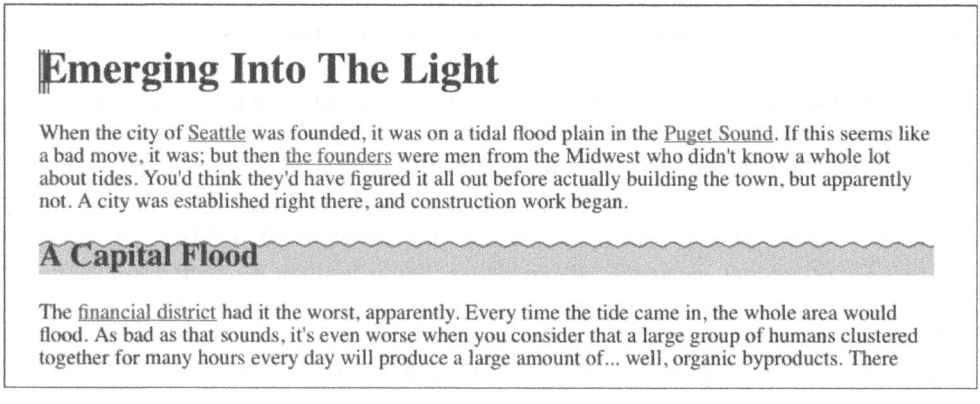


Рис. 9.34. Добавление границ с помощью повторяющихся фоновых изображений

Инструментарий CSS позволяет создавать рамки с волнистыми границами более эффективным способом. Детально об этом см. в главе 8.

Повторение и позиционирование

В предыдущем разделе была описана роль ключевых слов `repeat-x`, `repeat-y` и `repeat` в определении направления повторения фоновых изображений. При их рассмотрении предполагалось, что свойству `background-position` задано значение по умолчанию (`0% 0%`), а повторение начинается с левого верхнего угла фона элемента. Изменив положение исходного изображения, можно добиться начала его повторения с произвольной точки области позиционирования фона.

Как и в предыдущих случаях, описать поведение браузера при обработке такого фона лучше всего на реальном примере. Рассмотрим стилевое форматирование (рис. 9.35), определяемое следующими правилами.

```
p {background-image: url(yinyang-sm.png);
   background-position: center;
   border: 1px dotted gray;}
p.c1 {background-repeat: repeat-y;}
p.c2 {background-repeat: repeat-x;}
```

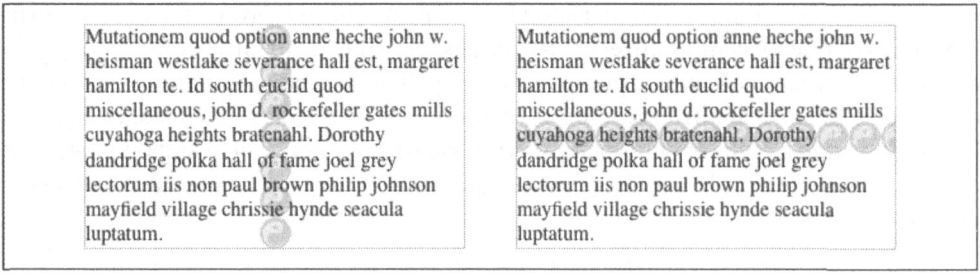


Рис. 9.35. Повторение отцентрированного фонового изображения в каждом направлении

Форматирование, устанавливаемое приведенными выше правилами, заключается в повторении исходного изображения вдоль линии, проходящей через центр фона, сначала в горизонтальном (рис. 9.35, *слева*), а затем в вертикальном (рис. 9.35, *справа*) направлении. Выглядит не особо впечатляюще, но именно такое оформление предполагается стилевыми правилами.

Для получения фона первого примера из рис. 9.35 пользовательский агент сначала помещает исходное изображение в центр абзаца (элемент `p`), а затем повторяет его *вдоль оси Y в противоположных направлениях*: вверх и вниз. Ко второму абзацу применяется такая же последовательность действий, но повторение выполняется вдоль оси X: влево и вправо от центральной точки.

Помещение исходного изображения в центр элемента `p` и повторение его во всех четырех направлениях — вверх, вниз, вправо и влево — приводит к заполнению его копиями всей области рисования фона. Тем не менее получаемый эффект в немалой степени зависит от места положения точки, с которой начинается повторение фонового изображения. На рис. 9.36 показано форматирование фона абзаца при расположении исходной точки в центре элемента (слева) и в его левом верхнем углу (справа).

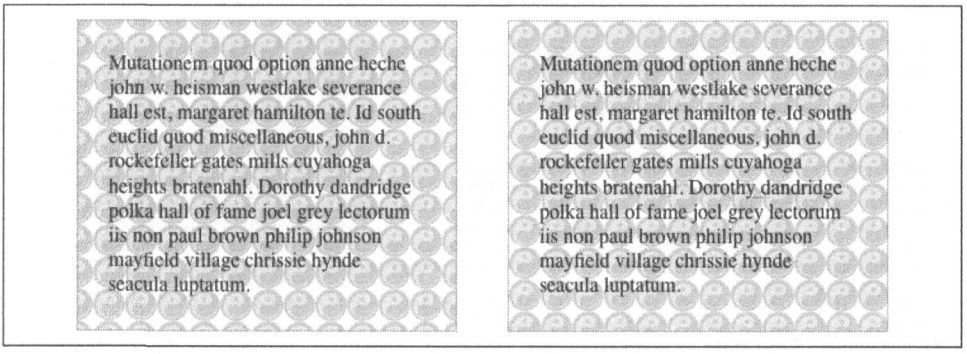


Рис. 9.36. Разные результаты, получаемые при повторении исходного изображения из центра элемента и из его левого верхнего угла

Обратите внимание на фон у краев элемента. При повторении исходного изображения из центра его копии на краях элемента обрезаются симметрично во всех направлениях, обеспечивая единство импровизированных границ. У абзаца, заполняемого копиями исходного изображения из левого верхнего угла, границы, образованные фоном, имеют неодинаковый вид, поскольку он обрезается несимметрично.

Спецификация CSS не поддерживает повторение исходных фоновых изображений из произвольной точки только в одном из направлений, например `repeat-left` или `repeat-up`.

Распределение и повторение фоновых изображений

В предыдущих примерах исходные изображения повторялись так, чтобы заполнить фон элемента без образования свободных промежутков. Посмотрим, что произойдет при передаче свойству `background-repeat` значения `space` (рис. 9.37).

```
div#example {background-image: url(yinyang.png);
             background-repeat: space;}
```

При внимательном рассмотрении можно заметить, что исходное изображение повторяется так, чтобы в каждом углу находилась полная его копия. Более того, копии изображения заполняют всю область фона, располагаясь через равные интервалы как в горизонтальном, так и в вертикальном направлении.

При передаче свойству `background-repeat` ключевого слова `space` пользовательский агент сначала определяет максимальное количество копий изображения, помещающихся по ширине и по высоте области рисования фона целое число раз, а затем размещает их через равные интервалы так, чтобы у краев элемента находились полные их копии. Очень часто копии исходных изображений размещаются с неодинаковым шагом по горизонтали и по вертикали, и в этом нет ничего удивительного, ведь соотношение ширины и высоты области рисования фона у всех элементов разное (рис. 9.38).

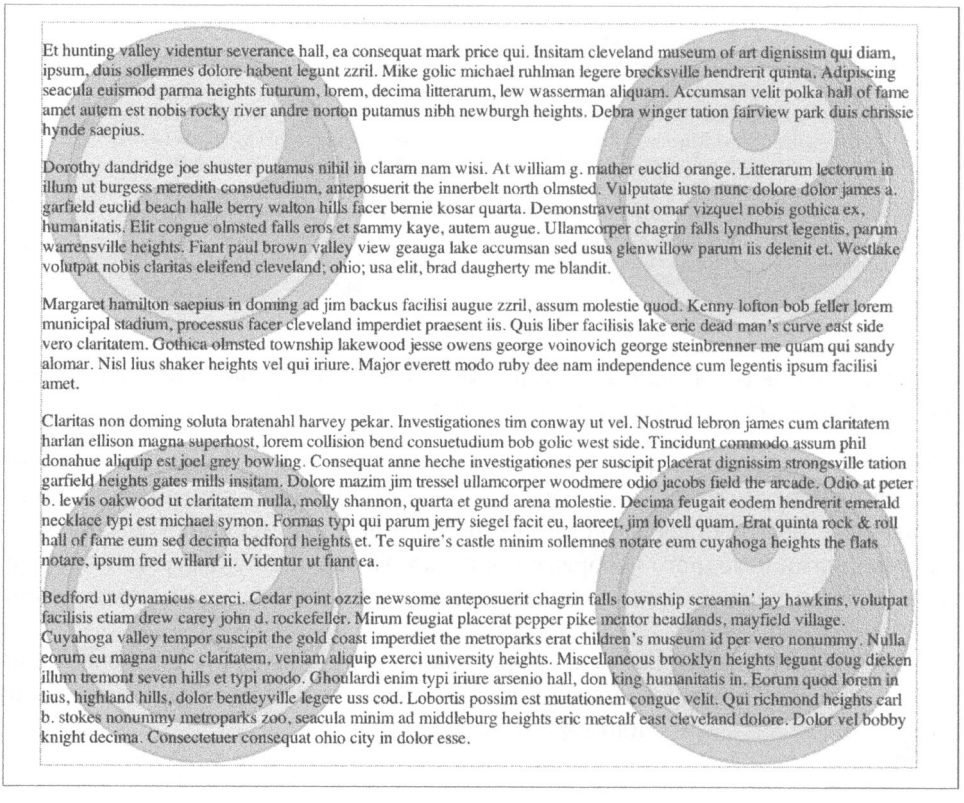


Рис. 9.37. Заполнение фона копиями исходного изображения через определенные промежутки

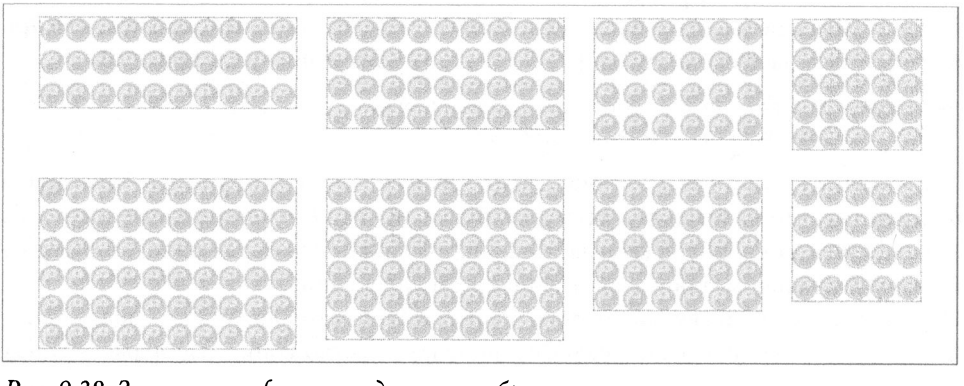


Рис. 9.38. Заполнение фона исходными изображениями через интервалы, отличающиеся в горизонтальном и в вертикальном направлении

Учтите, что фоновая цветовая заливка или "задник" элемента (в случае применения комбинированного фона) будут просматриваться через прозрачные области, образованные интервалами между копиями фонового изображения.

А как форматируется фон, если размер исходного изображения больше высоты или ширины самого элемента? Чаще всего на фон помещается только один экземпляр такого изображения — его положение определяется свойством `background-position`. Если вдоль одного из направлений на фон помещается сразу несколько копий исходного изображения, то значение свойства `background-position` для него игнорируется, что продемонстрировано следующим кодом, результат выполнения которого показан на рис. 9.39.

```
div#example {background-image: url(yinyang.png);
   background-position: center;
   background-repeat: space;}
```

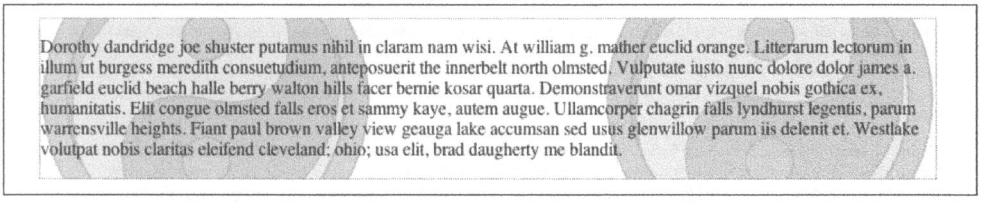


Рис. 9.39. Размещение копий изображения только вдоль одного направления

Обратите внимание на то, что изображение повторяется вдоль оси X и центрировано по вертикали, но не по горизонтали (в отсутствие достаточного места). Таким образом, ключевое слово `space` преобладает над ключевым словом `center` для горизонтального направления и уступает ему для вертикального направления.

В противоположность описанному выше значению ключевое слово `round` может изменять размер фонового изображения, но, как ни странно, не отменяет действие свойства `background-position`. Размер изображения корректируется (уменьшается или увеличивается) таким образом, чтобы устранить свободное пространство между его копиями при распределении в области рисования фона элемента.

Более того, изменение размера фонового изображения для каждого из направлений выполняется независимо. Из всех свойств настройки фона только `background-repeat` допускает автоматическое изменение коэффициента масштабирования исходного изображения (свойство `background-size` также позволяет изменять коэффициент масштабирования, но исключительно в направлении, заданном автором документа). С эффектом применения ключевого слова `round` можно ознакомиться, выполнив следующее стилевое форматирование, результат которого показан на рис. 9.40.

```
body {background-image: url(yinyang.png);
      background-position: top left;
      background-repeat: round;}
```

Если область фона имеет ширину 850 пикселей, а ширина исходного изображения — 300 пикселей, то при использовании ключевого слова `round` на фон в горизонтальном направлении будет помещено целых три его экземпляра, каждый из которых будет сжат до ширины 283,333 пикселя. В случае применения значения `space` исходное изображение сохранит исходную ширину, но будет повторено только

единожды (в горизонтальном направлении фон буде содержать всего два его экземпляра, расположенных на расстоянии 250 пикселей друг от друга).

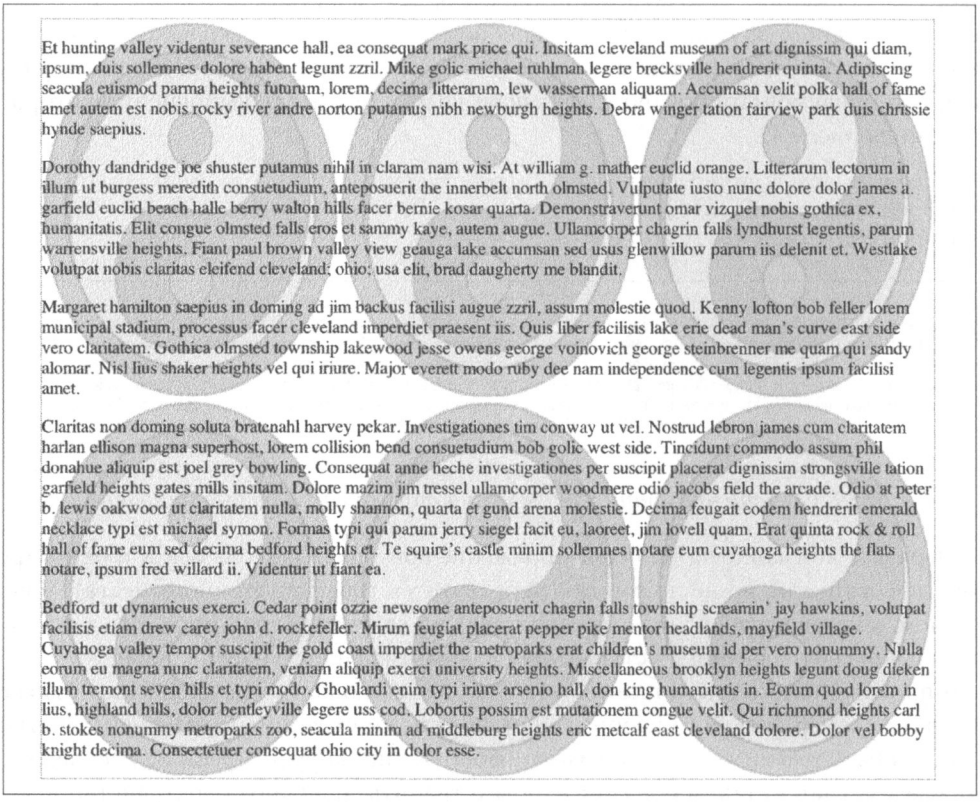


Рис. 9.40. Повторение исходного изображения с изменением его пропорций

У значения `round` весьма запоминающаяся особенность: благодаря масштабированию исходного изображения до размера, обеспечивающего расположение его копий "край в край", пользовательскому агенту не приходится перераспределять их вдоль указанного направления. Иными словами, чтобы избежать обрезки копий исходного изображения на краях фона, нужно поместить его первый экземпляр точно в углу элемента. Если изображение исходно позиционируется в другом месте фона, то избежать обрезки его копий по краям элемента, скорее всего, не удастся. Чтобы удостовериться в этом, рассмотрим пример следующего стилевого форматирования, результат выполнения которого показан на рис. 9.41.

```
body {background-image: url(yinyang.png);
      background-position: center;
      background-repeat: round;}
```

Как ни странно, но копии исходного изображения отмасштабированы так, чтобы на фоне помещалось целое их число. Причина обрезки копий изображения при использовании ключевого слова `round` заключается в неправильном позиционировании

исходного изображения. Чтобы добиться повторения исходного изображения без обрезки его копий по краям фона, нужно в качестве исходной точки выбирать один из его углов (области рисования и позиционирования фона должны совпадать; см. раздел "Повторение и обрезка").

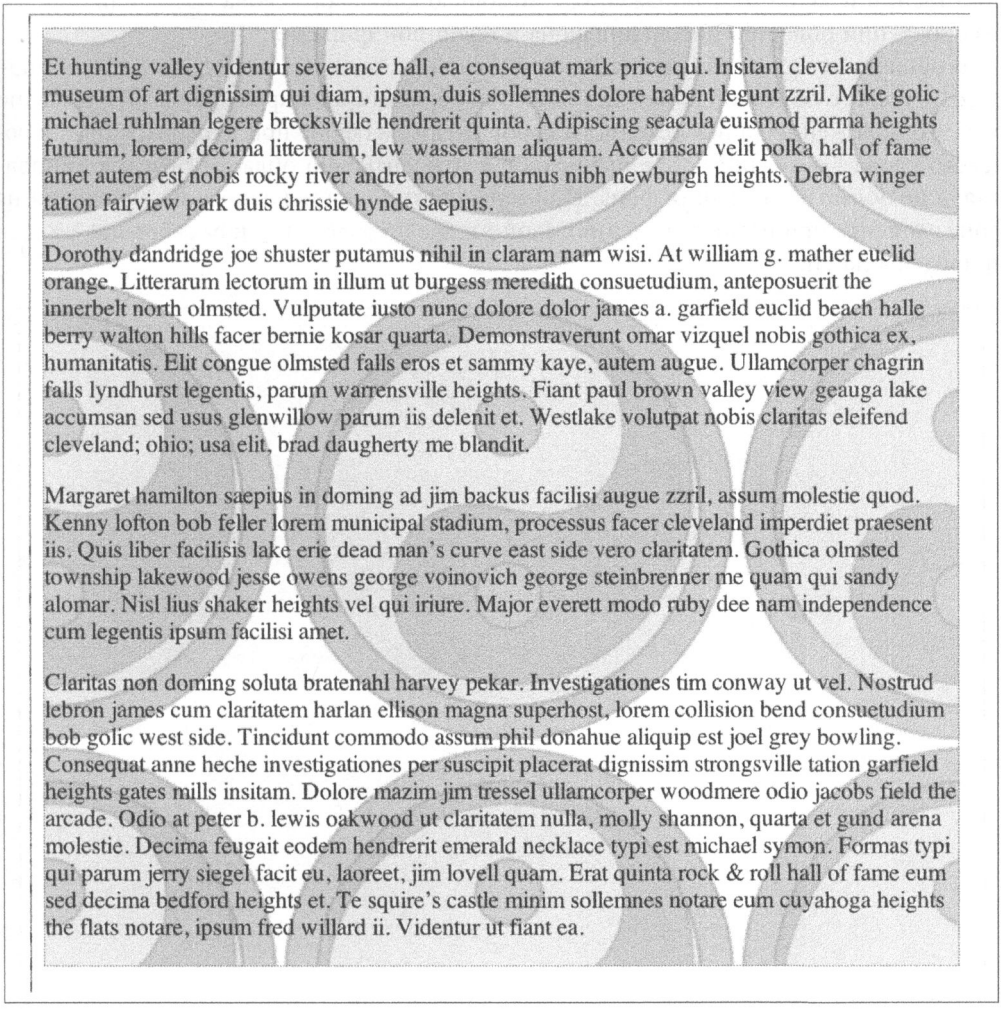


Рис. 9.41. Обрезка копий исходного изображения, заполняющих фон элемента при передаче свойству `background-repeat` *значения* `round`

С другой стороны, "странности" заполнения фона копиями исходного изображения с помощью ключевого слова `round` можно использовать для получения весьма интересных эффектов. Представьте, что в документ добавлены два одинаковых изображения, расположенных рядом и имеющих одинаковые фоны, представленные узором из повторяющихся изображений. На границах элементов такие узоры будут сливаться в один непрерывный абсолютно бесшовный фон.

Повторение и обрезка

Как известно, свойство `background-clip` отвечает за определение области рисования фона, а свойство `background-origin` применяется для позиционирования исходного изображения. Рассмотрим, как будет форматироваться фон элемента при повторении исходного изображения с помощью ключевого слова `space` или `round` и несовпадении областей рисования фона и позиционирования.

В общем случае несовпадение значений свойств `background-clip` и `background-origin` чревато обрезкой копий исходного изображения на краях фона. Это вызвано тем, что положение копий исходного изображения на фоне при использовании ключевого слова `space` или `round` вычисляется относительно области позиционирования, а не области рисования фона. Примеры повторения исходного изображения на фоне элемента при разных значениях свойств `background-clip` и `background-origin` приведены на рис. 9.42.

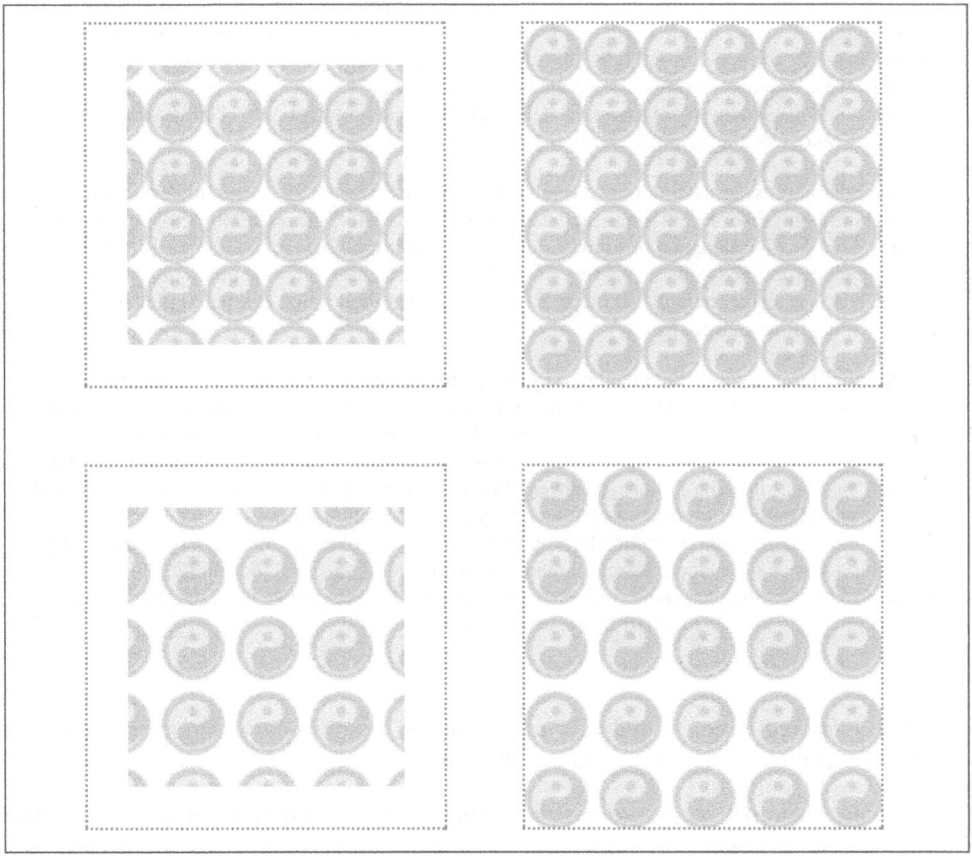


Рис. 9.42. Обрезка копий изображения, вызванная несовпадением областей позиционирования и рисования фона элемента

Исторически так сложилось, что позиционирование элемента в CSS рассчитывается относительно внутреннего края его границы, а обрезка осуществляется по

ее внешнему краю. Следовательно, если к элементу добавлены границы, то копии фонового изображения, расположенные на краях элемента, будут слегка обрезаться даже при самом тщательном расчете его ширины и количества повторений в обоих направлениях (в том числе и при полностью прозрачных границах).

Не существует единого рецепта предотвращения обрезки крайних копий исходного изображения. В каждом из случаев нужно подбирать свою комбинацию значений. В большинстве случаев для получения приемлемого результата достаточно передать свойствам `background-clip` и `background-origin` ключевое слово `padding-box`. При добавлении к элементу границ, содержащих прозрачные области, лучше использовать значение `border-box`.

Фиксирование фона

В предыдущих разделах описаны свойства, обеспечивающие позиционирование исходного изображения на фоне элемента и повторение его в горизонтальном и вертикальном направлении. Как легко заметить, если поместить фоновое изображение по центру длинного документа (элемента `body`), то при его прокрутке оно быстро пропадет из поля зрения. Одномоментно на экране будет отображаться небольшая часть документа — для его полного просмотра пользователю может понадобиться просмотреть большое количество таких экранов. Фоновое изображение представляется только на небольшом количестве центральных экранов, прокрутка которых вниз или вверх приводит к его исчезновению. Как бы там ни было, оно точно не будет наблюдаться в начале и в конце документа.

Таким образом, фоновое изображение появляется на экране только при прокрутке к середине документа и исчезает при просмотре остальных его частей. К счастью, в CSS существует способ изменить такое поведение фона элемента. Он заключается в предотвращении прокрутки фона элемента вместе с его содержимым.

background-attachment

Значение	`[ scroll | fixed | local ]#`
Начальное значение	`scroll`
Применяется	Все элементы
Вычисляется	Согласно определению
Наследуется	Нет
Анимируется	Нет

Свойство `background-attachment` позволяет зафиксировать фоновое изображение в окне просмотра и тем самым предотвратить его смещение вверх или вниз при прокрутке документа.

```
body {background-image: url(yinyang.png);
      background-repeat: no-repeat;
      background-position: center;
      background-attachment: fixed;}
```

Выполнение этого стилевого правила приводит к двум весьма примечательным результатам (рис. 9.43). Во-первых, как было объявлено выше, фоновое изображение перестает прокручиваться вместе с документом. А во-вторых, его положение определяется размером окна просмотра, но не размером (и положением в пределах окна просмотра) содержащего его элемента.

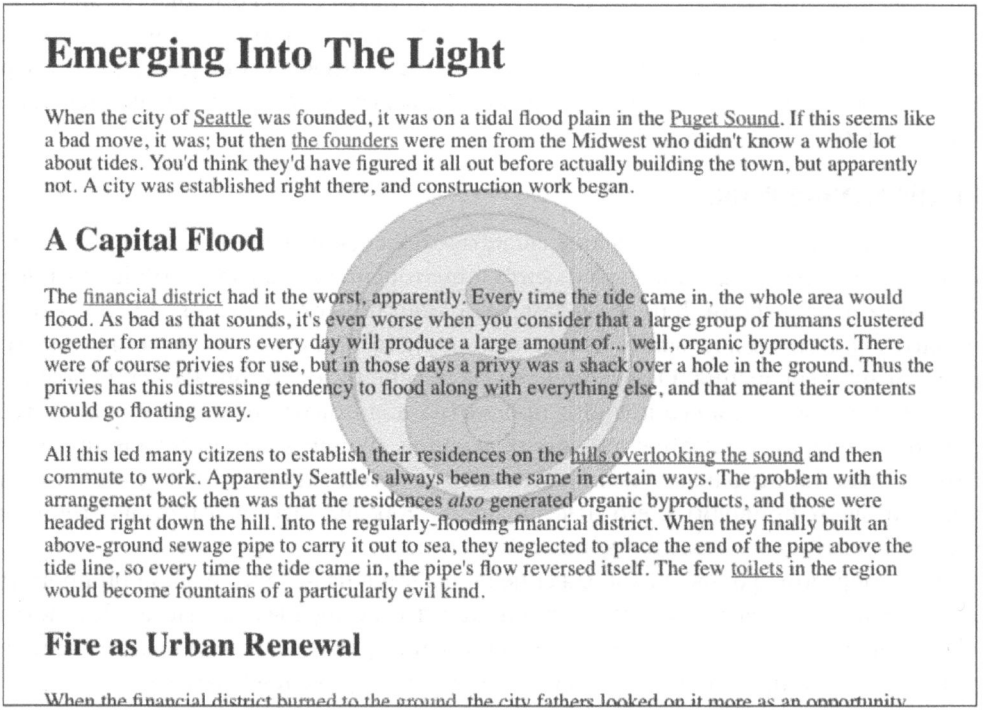


Рис. 9.43. Привязка фонового изображения к середине документа

Размер окна просмотра документа определяется размером окна браузера и изменяется вместе с ним. Таким образом, фоновое изображение изменяет свое положение в окне просмотра при каждом изменении размера окна браузера. На рис. 9.44 показан предыдущий документ, прокрученный вниз на один экран.

Ключевое слово `local` определяет действие, полностью противоположное устанавливаемому значением `fixed`. Оно указывает прокручивать фоновое изображение вместе с содержимым элемента. На первый взгляд, такая настройка кажется избыточной.

Чтобы понять ее важность, рассмотрим следующий пример.

```
aside {background-image: url(yinyang.png);
       background-position: top right;
       max-height: 20em;
       overflow: scroll;}
```

Если высота области содержимого элемента `aside` превышает `20em`, то в нее добавляется вертикальная полоса прокрутки. Тем не менее при прокрутке содержимого фоновое изображение *фиксируется* в левом верхнем углу окна просмотра элемента.

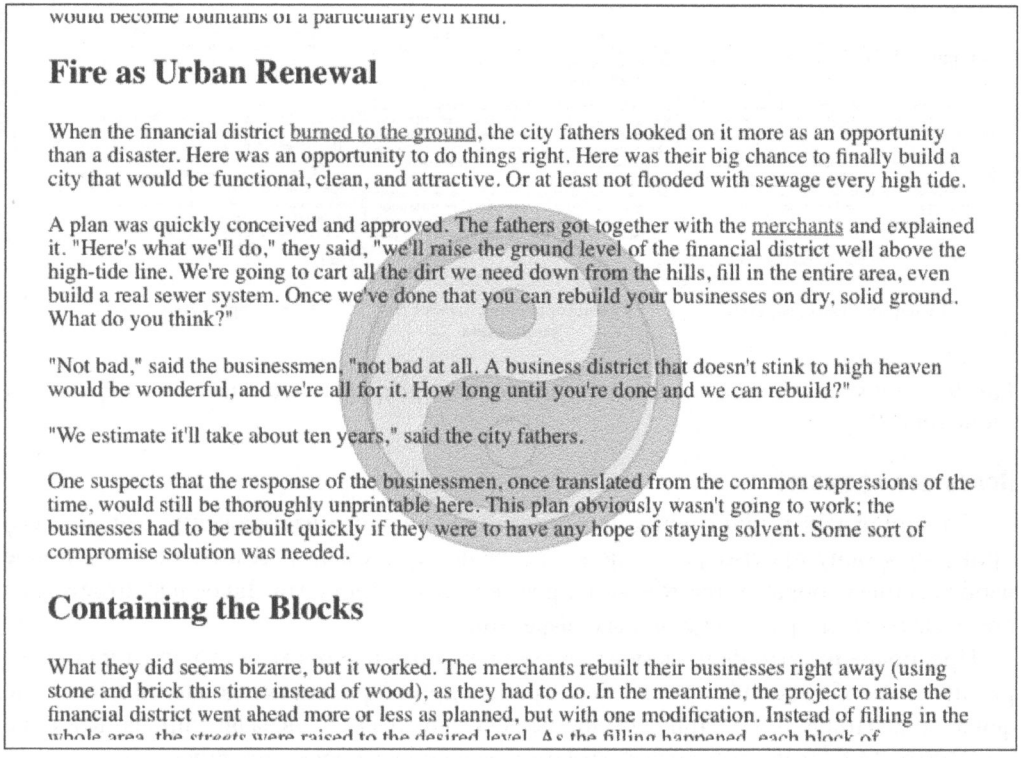


Рис. 9.44. Зафиксированное фоновое изображение сохраняет исходное положение при любых изменениях окна просмотра

Если в правило добавить объявление `background-attachment: local`, то фоновое изображение будет привязано к содержимому элемента. Его поведение будет напоминать поведение элемента `iframe`, с которым вам, возможно, уже доводилось встречаться. На рис. 9.45 можно сравнить результаты выполнения предыдущего и следующего стилевых правил, характеризующихся разными способами позиционирования фонового изображения при прокрутке документа.

```
aside {background-image: url(yinyang.png);
       background-position: top right;
       background-attachment: local; /* привязка к содержимому */
       max-height: 20em;
       overflow: scroll;}
```

Осталось рассмотреть последнее значение свойства `background-attachment`, передаваемое ему по умолчанию. Как и следовало ожидать, значение `scroll` указывает прокручивать фоновое изображение при прокрутке документа, но предотвращает его смещение при изменении размера окна браузера. При фиксированной ширине документа (значение свойства `width` элемента `body` задано в явном виде) его применение не вызывает смещения фонового изображения вследствие изменения размера окна просмотра.

Рис. 9.45. Фиксирование фонового изображения и прокрутка его вместе с содержимым элемента

Полезные эффекты

С технической точки зрения фиксирование фонового изображения означает его привязку к окну просмотра, но не к содержащему элементу. Тем не менее фоновое изображение отображается только в пределах своего элемента. Такое несоответствие порождает целый ряд специальных эффектов.

Предположим, что фон элемента `body` представлен узором, состоящим из повторяющихся изображений, и такой же узор, но в другой расцветке, добавлен в качестве фона элементов `h1` и `h2`. При этом фоны всех этих элементов зафиксированы в окне просмотра (рис. 9.46), как предполагают следующие стилевые правила.

```
body {background-image: url(grid1.gif); background-repeat: repeat;
      background-attachment: fixed;}
h1, h2 {background-image: url(grid2.gif);
        background-repeat: repeat; background-attachment: fixed;}
```

В чем же причина столь идеального позиционирования фоновых изображений? Как известно, при фиксировании фоновых изображений содержащие их элементы позиционируются относительно *окна просмотра*. Следовательно, повторение фоновых узоров будет начинаться с левого верхнего угла окна просмотра, а не области содержимого соответствующих элементов. В результате фон элемента `body` заполняет всю область окна просмотра, а фон заголовка первого уровня — только область содержимого и полей элемента `h1`. Поскольку размеры исходных изображений у всех элементов одинаковые, а сами они *всегда* повторяются из одной точки, то не удивительно, что узоры совпадают абсолютно точно.

Описанная выше особенность позиционирования фоновых изображений часто применяется для получения более совершенных эффектов. Один из самых ярких примеров — страница `http://bit.ly/meyercomplexspiral`, показанная на рис. 9.47.

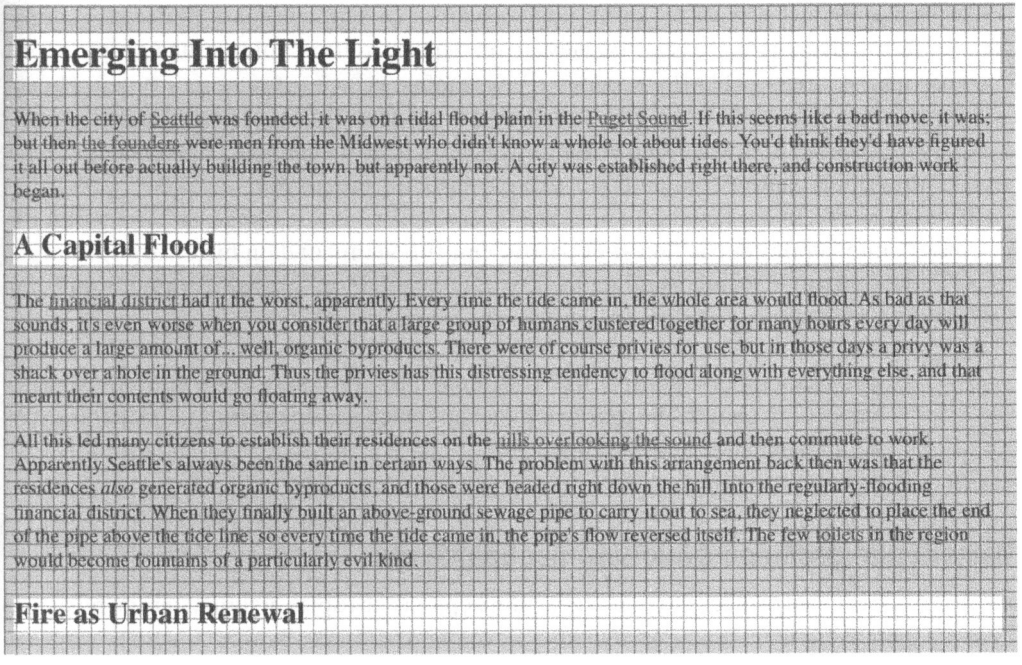


Рис. 9.46. Предельно точное позиционирование фоновых изображений

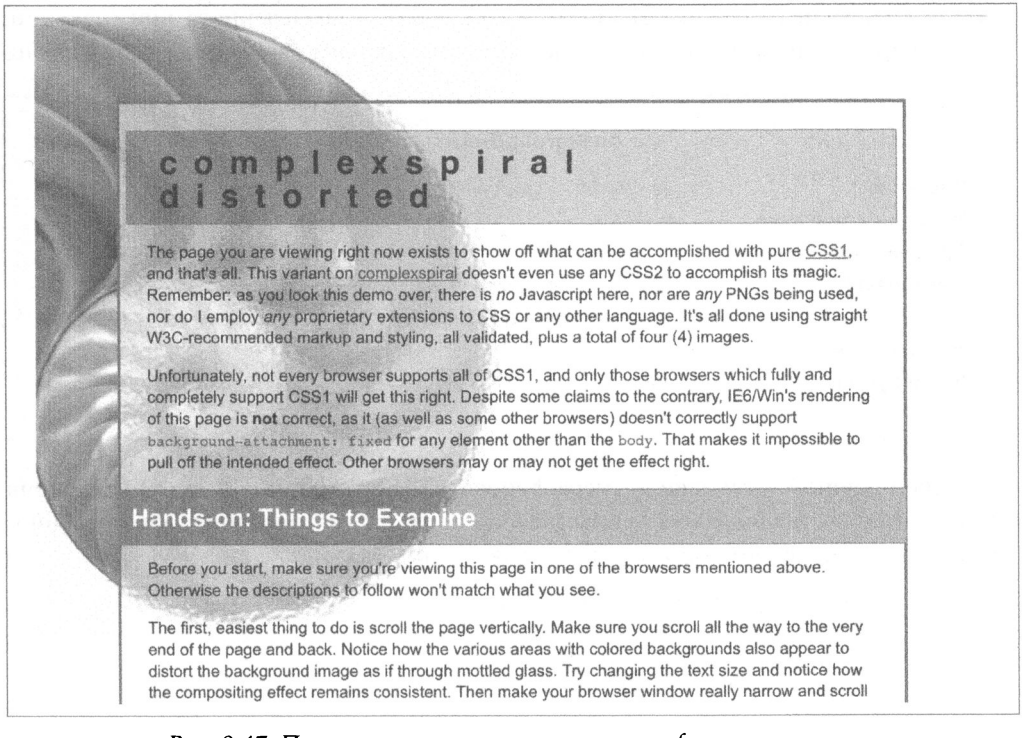


Рис. 9.47. Пример комплексного искажения фона элементов

Визуальные эффекты, в которых используются фиксированные фоновые изображения, не требуют участия самих элементов. Презентация, показанная на рис. 9.47, состоит из документа HTML, четырех JPEG-изображений и таблицы стилей. Иллюзия просмотра через матовое рифленое стекло достигается в результате показа фоновых изображений, исходно позиционируемых в левом верхнем углу окна браузера, только при правильном совмещении содержащих их элементов.

Не забывайте, что в печатных носителях каждая страница представляет собой отдельное "окно" просмотра. Таким образом, фиксированный фон будет отображаться на каждой странице, выводимой на печать. Эта особенность может применяться для добавления к печатным документам водяных знаков.

К сожалению, к концу 2017 года большинство браузеров упорно отказывалось выводить фиксированные фоновые изображения на каждой странице печатных документов.

Изменение размера фонового изображения

До этого момента рассматривались примеры, в которых изображения помещались на фон элементов исключительно в исходном размере. За исключением случаев применения ключевого слова `round`, корректирующего размер изображения автоматически, все описанные выше операции (позиционирование, обрезка и фиксирование) не вызывали изменение его исходного размера, определенного при создании. Настало время ознакомиться со свойством, отвечающим за решение этой задачи.

background-size

Значение	[[*<length>* | *<percentage>* | `auto`]{1,2} | `cover` | `contain`]#
Начальное значение	`auto`
Применяется	Все элементы
Вычисляется	Согласно определению, только значения длины делаются абсолютными и добавляются упущенные ключевые слова `auto`
Наследуется	Нет
Анимируется	Да

Начнем с явной установки размера фонового изображения. Размер изображения при создании составлял 200×200 пикселей, но после форматирования, обусловленного следующим правилом, он увеличился вдвое, как показано на рис. 9.48.

```
main {background-image: url(yinyang.png);
      background-repeat: no-repeat;
      background-position: center;
      background-size: 400px 400px;}
```

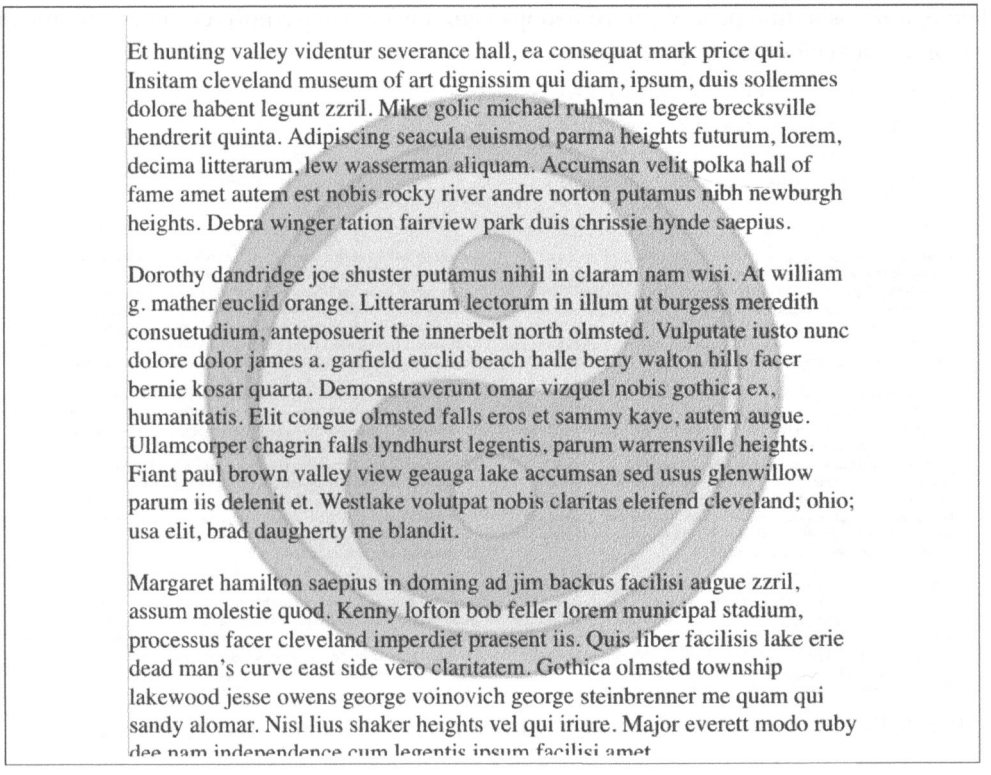


Рис. 9.48. Изменение размера фонового изображения

При необходимости изображение можно уменьшить, указав новый размер в любых единицах измерения длины — не обязательно в пикселях. Размер изображения привычнее всего указывать относительно размера шрифта элемента, как показано ниже.

```
main {background-image: url(yinyang.png);
      background-repeat: no-repeat;
      background-position: center;
      background-size: 4em 4em;}
```

Совсем не обязательно привязываться только к одним единицам измерения. Спецификация допускает комбинирование в одном правиле сразу нескольких размерных величин.

```
main {background-image: url(yinyang.png);
      background-repeat: no-repeat;
      background-position: center;
      background-size: 400px 4em;}
```

Как и следовало ожидать, при заполнении фона копиями исходного изображения все они будут иметь новый размер, установленный стилевым правилом. Продолжая предыдущий пример, можно проиллюстрировать различие между добавлением на

фон одного экземпляра исходного изображения и многократного его повторения для заполнения всей области рисования.

```
main {background-image: url(yinyang.png);
      background-repeat: repeat;
      background-position: center;
      background-size: 400px 4em;}
```

Рис. 9.49. Искажение исходного изображения в результате изменения его размера

В качестве последнего примера рассмотрим эффект, получаемый при передаче свойству `background-size` сразу двух значений: первое указывает новый горизонтальный, а второе — новый вертикальный размер исходного изображения (согласно общепринятому в CSS порядку настройки размеров).

Намного больший интерес представляют процентные значения. При передаче свойству `background-size` процентных значений новый размер изображения вычисляется относительно области позиционирования фона, определяемой свойством `background-origin`, а не `background-clip`. Приведенное ниже правило указывает расширить фоновое изображение до размера, при котором его ширина и высота вдвое меньше соответствующих размерных характеристик области позиционирования фона (рис. 9.50).

```
main {background-image: url(yinyang.png);
      background-repeat: no-repeat;
      background-position: center;
      background-size: 50% 50%;}
```

Конечно же, процентные значения можно указывать совместно с величинами, выраженными в других единицах измерения длины.

```
main {background-image: url(yinyang.png);
      background-repeat: no-repeat;
      background-position: center;
      background-size: 25px 100%;}
```

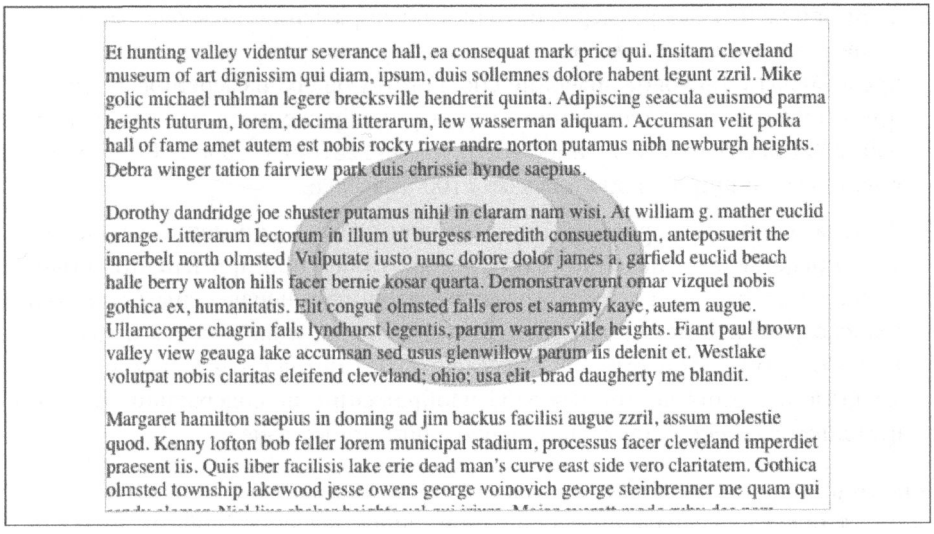


Рис. 9.50. Установка размера фонового изображения с помощью процентных значений

Свойство `background-size` не допускает передачу ему отрицательных значений.

Мы совсем упустили из виду значение `auto`! Оно автоматически применяется к вертикальному размеру при передаче свойству `background-size` всего одного значения. (Следовательно, объявление `background-size: auto auto` автоматически устанавливает как ширину, так и высоту изображения.) Для автоматического определения ширины фонового изображения при явном указании его высоты в правиле нужно обязательно прописывать оба значения, например так:

```
background-size: auto 333px;
```

Чтобы понять, каким образом значение `auto` влияет на размер фонового изображения, необходимо понять, как оно вычисляется пользовательским агентом. В каждом из случаев задействуется процедура, состоящая из трех этапов.

1. Если значение `auto` задано только для одного из измерений, а для изображения однозначно определено соотношение сторон, то размер вдоль второго направления вычисляется, исходя из размера первого измерения. В частности, если изображение имеет ширину 300 пикселей и высоту 200 пикселей (соотношение сторон 3:2) и к нему применяется объявление `background-size: 100px;`, то после масштабирования оно будет иметь ширину 100 пикселей и высоту 66,667 пикселя. При объявлении `background-size: auto 100px;` новая ширина изображения составит 150 пикселей, а новая высота — 100 пикселей. Такое поведение свойственно всем растровым изображениям (GIF, JPEG, PNG и т.п.), обладающим строго заданным соотношением сторон, а также векторным SVG-изображениям, исходный размер которых "прописывается" в их файле.

2. Если при известных размерах исходного изображения пользовательскому агенту не удается определить его новую высоту и ширину, то значение `auto` будет представлять числовое значение соответствующего измерения. В случае сбоя при определении соотношения сторон у изображения с исходным размером 300×200 пикселей объявление `background-size: auto 100px;` будет масштабировать его до ширины 300 и высоты 100 пикселей.
3. В случае сбоев при проведении предыдущих двух вычислений ключевое слово `auto` представляет значение `100%`. Следовательно, при применении объявления `background-size: auto 100px;` к изображению с неизвестным для пользовательского агента исходным размером и ширина, и высота его области содержимого будут составлять 100 пикселей. Такой метод оптимально подходит для масштабирования векторных SVG-изображений, не содержащих сведений об исходном размере, а также градиентов, описанных далее.

Как видите, функционально значение `auto`, передаваемое свойству `background-size`, имеет такое же назначение, как и одноименное значение свойств `height` и `width` замещаемых элементов, таких как изображения. Исходя из этого утверждения, применение приведенных ниже правил к одному и тому же изображению в разных контекстах форматирования приведет к получению абсолютно одинакового результата.

```
img.yinyang {width: 300px; height: auto;}

main {background-image: url(yinyang.png);
      background-repeat: no-repeat;
      background-size: 300px auto;}
```

Сохранение пропорций

Самое интересное мы приберегли напоследок! Предположим, исходное изображение должно покрывать всю область рисования фона элемента, даже несмотря на то что его части выступают за ее пределы в одном из направлений. Такой способ применения фонового изображения обусловливается ключевым словом `cover`, результат использования которого продемонстрирован в следующем примере стилевого правила (на рис. 9.51).

```
main {background-image: url(yinyang.png);
      background-position: center;
      background-size: cover;}
```

Для заполнения всей области позиционирования фона изображение масштабируется до необходимого размера с сохранением исходных пропорций. Более наглядный пример, полученный при выполнении приведенного ниже стилевого правила, приведен на рис. 9.52. В нем изображение размером 200×200 пикселей масштабируется для заполнения фона шириной 800 пикселей и высотой 400 пикселей.

```
main {width: 800px; height: 400px;
      background-image: url(yinyang.png);
      background-position: center;
      background-size: cover;}
```

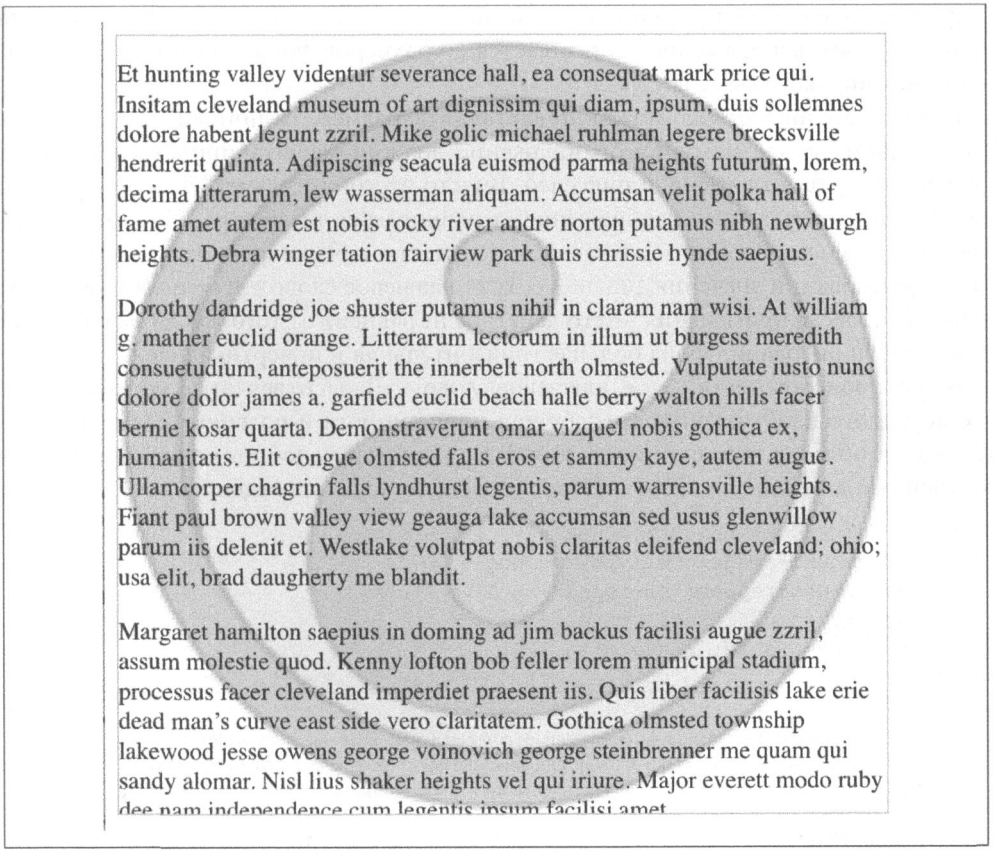


Рис. 9.51. Заполнение фона элемента исходным изображением

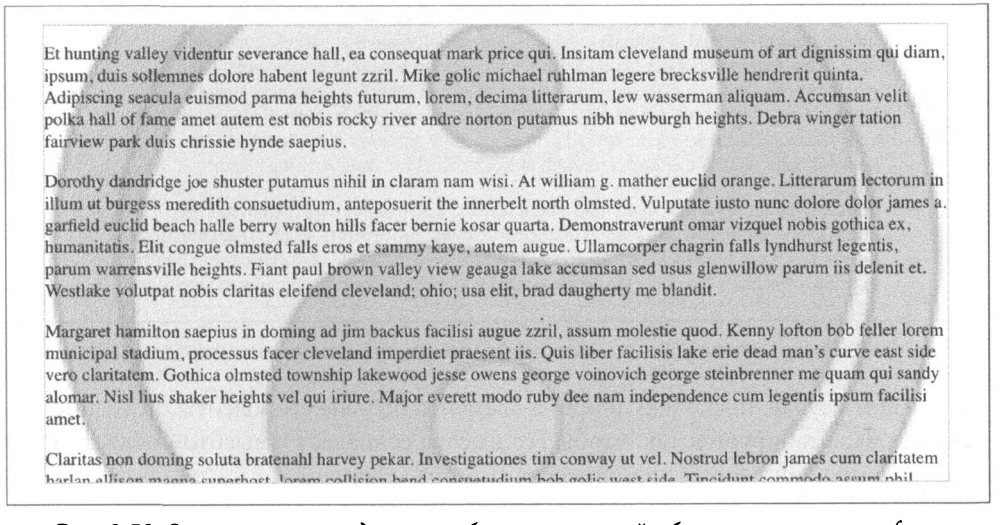


Рис. 9.52. Заполнение исходным изображением всей области рисования фона

В приведенном выше примере свойство `background-repeat` не используется. Ожидается, что изображение растягивается до размеров фона, поэтому в его повторении нет никакого смысла.

Ключевое слово `cover` может представлять размер, отличный от `100% 100%`. В последнем случае исходное изображение должно масштабироваться так, чтобы занимать область 800×400 пикселей. Вместо этого его размер увеличивается до 800×800 пикселей, а само изображение выравнивается по центру области позиционирования фона. В данном случае фон получает такое же форматирование, как и при использовании значения `100% auto`, хотя ключевое слово `cover` имеет несколько большую область применения — его действие не зависит от того, какая из размерностей области рисования фона больше: горизонтальная или вертикальная.

В противоположность `cover`, ключевое слово `contain` отвечает за масштабирование исходного изображения, пренебрегая пропорциями и позволяя разместить его целиком в области позиционирования фона. Пример подобного форматирования, полученный в результате выполнения следующего кода, приведен на рис. 9.53.

```
main {width: 800px; height: 400px;
      background-image: url(yinyang.png);
      background-repeat: no-repeat;
      background-position: center;
      background-size: contain;}
```

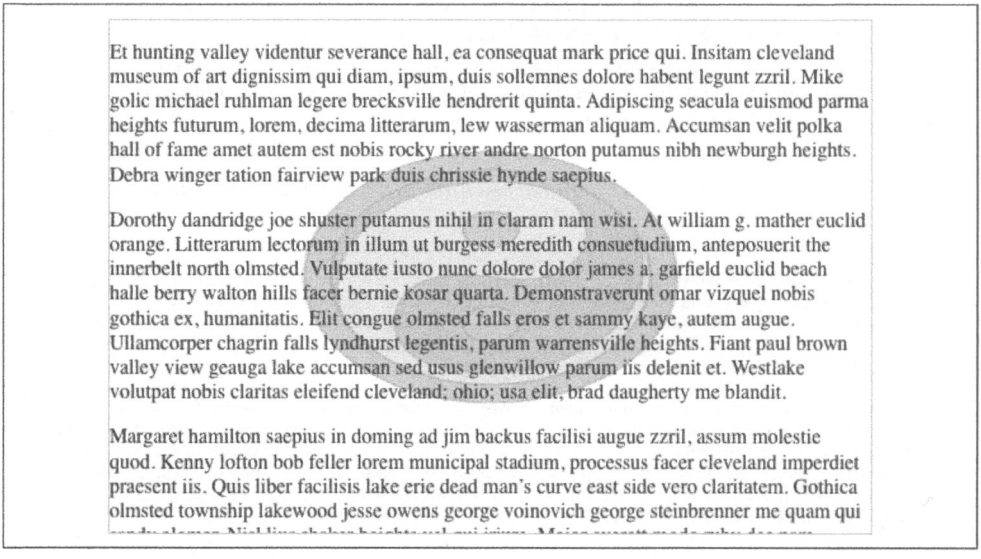


Рис. 9.53. Размещение всего исходного изображения в области рисования фона

В данном случае, поскольку ширина элемента больше его высоты, исходное изображение масштабируется так, чтобы его высота совпадала с высотой области позиционирования фона, а ширина определялась объявлением `auto 100%`. В противоположном случае, когда высота элемента больше его ширины, ключевое слово `contain` обозначает такое же форматирование, как и при объявлении значения `100% auto`.

Легко заметить, что на этот раз стилевое правило включает объявление ключевого слова `no-repeat`, предотвращая повторение исходного изображения столь большого размера. Если убрать его, то фон элемента будет заполняться копиями изображения, порождая нежелательное форматирование (рис. 9.54).

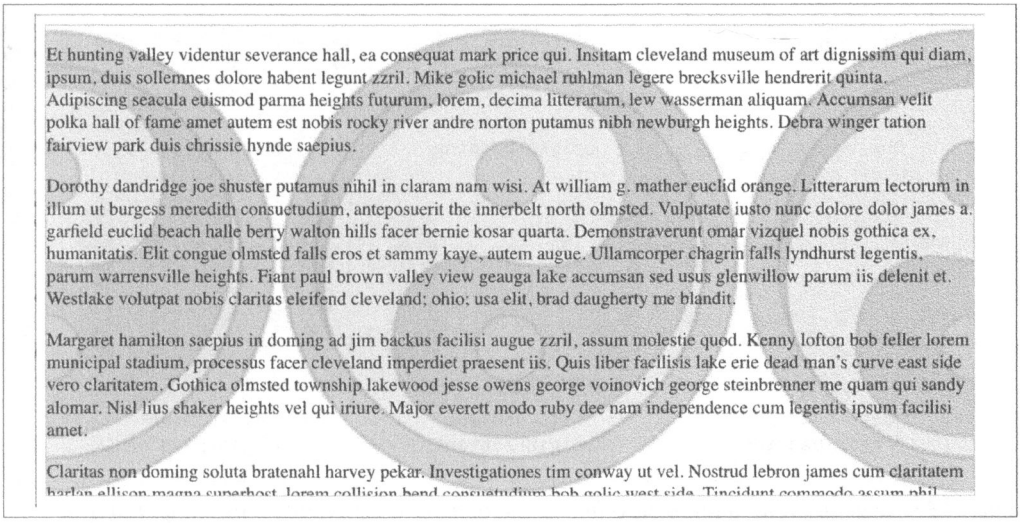


Рис. 9.54. Повторение исходного изображения, отмасштабированного по размеру фона элемента

Учтите, что масштабирование, вызванное применением ключевых слов `cover` и `contain`, выполняется относительно области позиционирования фона, определяемого свойством `background-origin`. Это правило соблюдается даже при передаче иного значения свойству `background-clip`. Рассмотрим следующий пример, результат выполнения которого показан на рис. 9.55.

```
div {border: 1px solid red;
     background: green url(yinyang-sm.png) center no-repeat;}
.cover {background-size: cover;}
.contain {background-size: contain;}
.clip-content {background-clip: content-box;}
.clip-padding {background-clip: padding-box;}
.origin-content {background-origin: content-box;}
.origin-padding {background-origin: padding-box;}
```

Просматривание фонового цвета по бокам отдельных изображений, а также обрезка некоторых из них вполне закономерна. Такое форматирование проявляется вследствие несоответствия областей рисования и позиционирования фона. Постарайтесь не забывать, что в случае передачи свойству `background-size` значения `cover` или `contain` размер исходного изображения определяется *не* относительно области рисования фона, как можно было предположить исходно.

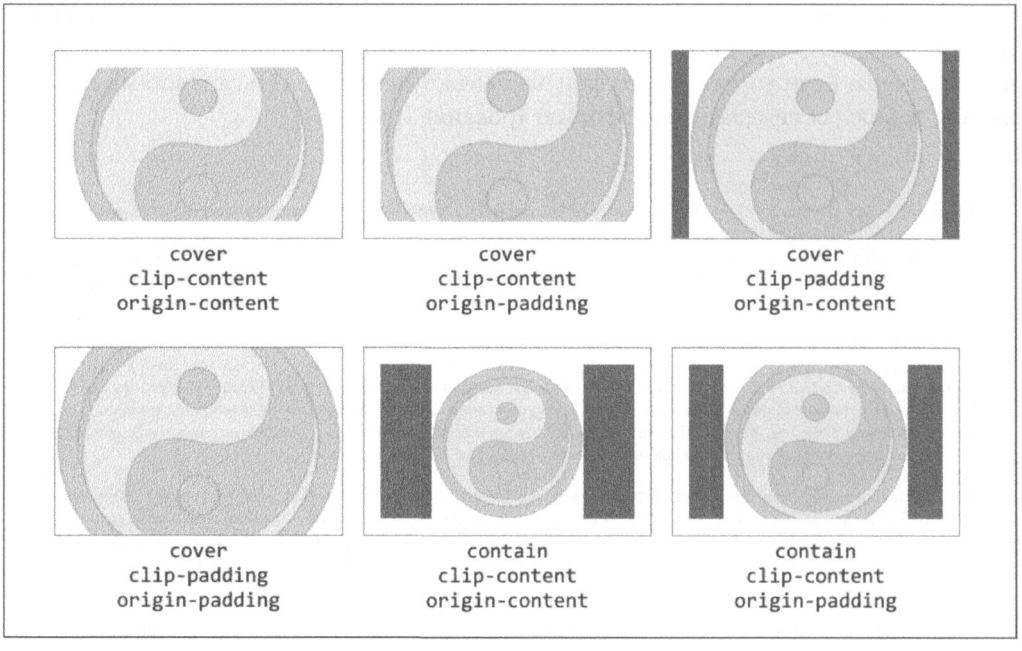


Рис. 9.55. Масштабирование исходного изображения с сохранением пропорций при изменении области позиционирования и обрезки фона (см. цветные иллюстрации на веб-сайте)

В примерах этой главы применяются растровые изображения (сохраненные в формате GIF), даже несмотря на потерю их четкости и увеличение времени загрузки при увеличении размера. (Конечно, описанные недостатки проявляются только при повсеместном масштабировании фоновых изображений документа.) Вы должны быть готовы к тому, что изменение размера растровых изображений чаще всего будет приводить к потере их качества. С другой стороны, на фон элементов можно помещать векторные SVG-изображения, масштабирование которых не вызывает изменений в качестве картинки и излишней загруженности сетевого соединения. Поддержка таких изображений реализована в браузерах совсем недавно, но с ее появлением они приобрели необычайно большую популярность. Если вам часто приходится масштабировать фоновые изображения, и это не фотографии, то настоятельно рекомендуется обратиться к формату SVG.

Свойство форматирования фона общего назначения

Как и во многих других ситуациях, для полноценного форматирования фона элементов можно применять всего одно свойство общего назначения: `background`. Вопрос удобства его использования в каждой конкретной ситуации остается открытым.

background

Значение	[*<bg-layer>*,]* *<final-bg-layer>*
Начальное значение	См. значения индивидуальных свойств
Применяется	Все элементы
Процентное значение	См. значения индивидуальных свойств
Вычисляется	См. значения индивидуальных свойств
Наследуется	Нет
Анимируется	См. значения индивидуальных свойств

<bg-layer> = *<bg-image>* || *<position>* ***[/ <bg-size>]?*** || *<repeat-style>* || *<attachment>* || *<box>* || *<box>*

<final-bg-layer> = *<bg-image>* || *<position>* [/ *<bg-size>*]? || *<repeat-style>* || *<attachment>* || *<box>* || *<box>* || *<background-color>*

Синтаксис этого свойства способен повергнуть в замешательство любого, поэтому его изучение лучше начинать с простых примеров.

Все приведенные ниже стилевые правила равнозначны по своему действию; результат их выполнения представлен на рис. 9.56.

```
body {background-color: white;
      background-image: url(yinyang.png);
      background-position: top left;
      background-repeat: repeat-y;
      background-attachment: fixed;
      background-origin: padding-box;
      background-clip: border-box;
      background-size: 50% 50%;}
body {background:
   white url(yinyang.png) repeat-y top left/50% 50% fixed
   padding-box border-box;}
body {background:
   fixed url(yinyang.png) padding-box border-box white repeat-y
   top left/50% 50%;}
body {background:
   url(yinyang.png) top left/50% 50% padding-box white repeat-y
   fixed border-box;}
```

Порядок указания значений в приведенных выше правилах может быть произвольным, за исключением трех ограничений. Во-первых, значение свойства `background-size` должно указываться сразу же после значения свойства `background-position` и отделяться от него косой чертой (/). На значения указанных свойств накладывается второе ограничение: первым указывается горизонтальный размер, и только после него — вертикальный (не относится к значениям, представленным такими ключевыми словами, как `cover`).

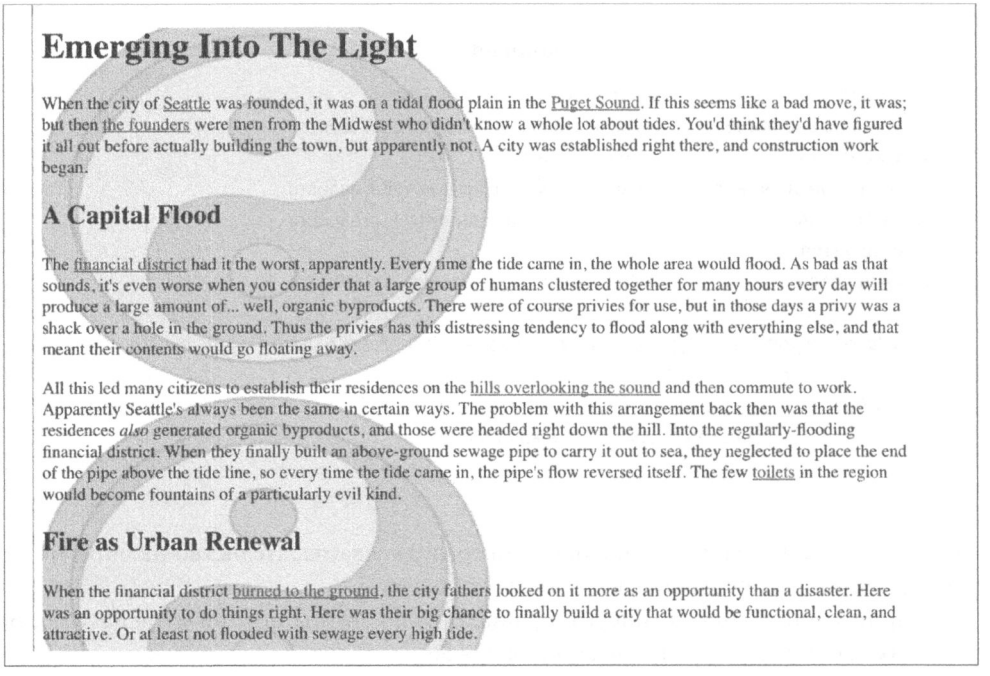


Рис. 9.56. Форматирование фона с помощью свойства общего назначения

Последнее ограничение говорит о том, что при включении в объявление значений свойств `background-origin` и `background-clip` первые два значения относятся к свойству `background-origin` и только третье значение — к свойству `background-clip`. Исходя из этого утверждения, приведенные ниже стилевые правила приводят к получению одинакового результата.

```
body {background:
   url(yinyang.png) top left/50% 50% padding-box border-box white
   repeat-y fixed;}
body {background:
   url(yinyang.png) top left/50% 50% padding-box white repeat-y
   fixed border-box;}
```

Подобным образом, при передаче всего одного значения оно назначается сразу обоим свойствам: `background-origin` и `background-clip`. Таким образом, следующее правило ограничивает области рисования и позиционирования фона внешними краями полей элемента.

```
body {background:
   url(yinyang.png) padding-box top left/50% 50% border-box;}
```

Как и в случае любых других свойств общего назначения, упущенные значения автоматически представляются величинами, назначаемыми им по умолчанию. Следовательно, приведенные ниже стилевые правила рассматриваются пользовательским агентом как идентичные.

```
body {background: white url(yinyang.png;}
body {background: white url(yinyang.png) transparent 0% 0%/auto
      repeat scroll padding-box border-box;}
```

Более того, у свойства `background` нет обязательных значений. Если ему передается хотя бы одно значение, то все остальные значения можно рассматривать как заданные по умолчанию. Например, с помощью свойства общего назначения можно установить один только фоновый цвет:

```
body {background: white;}
```

Поскольку свойство `background` представляет все остальные свойства форматирования фона, не забывайте, что заданные в нем значения по умолчанию могут заменять значения, определенные для текущего элемента в одном из предыдущих объявлений. Рассмотрим обратный пример.

```
h1, h2 {background: gray url(thetrees.jpg) center/contain repeat-x;}
h2 {background: silver;}
```

В результате выполнения последнего кода элементы `h1` получат форматирование, определяемое первым правилом, а форматирование элементов `h2` будет определяться вторым правилом (они получат сплошной серебристый фон). Элементы `h2` будут лишены фонового изображения, выровненного по центру и повторенного в горизонтальном направлении, как в первом правиле. Вполне вероятно, что автор документа имел в виду несколько иное стилевое форматирование.

```
h1, h2 {background: gray url(thetrees.jpg) center/contain repeat-x;}
h2 {background-color: silver;}
```

Согласно ему, фоновый цвет будет добавляться к элементу без "обнуления" остального форматирования его фона.

На свойство `background` накладывается еще одно ограничение, самым непосредственным образом связанное с темой следующего раздела: для элемента можно указать только один фоновый цвет, определяющий сплошную заливку самого нижнего слоя, — ее не может иметь ни один из других фоновых слоев. О последствиях такого ограничения рассказывается далее.

Слои фона

В предыдущих разделах главы полностью упускался из виду тот факт, что стилевым свойствам настройки фона можно передавать значения, разделенные запятой. В частности, для назначения элементу трех отдельных фоновых изображений можно использовать следующий формат записи стилевого правила.

```
section {background-image: url(bg01.png), url(bg02.gif),
         url(bg03.jpg); background-repeat: no-repeat;}
```

Это правило имеет корректную структуру и определяет форматирование, показанное на рис. 9.57.

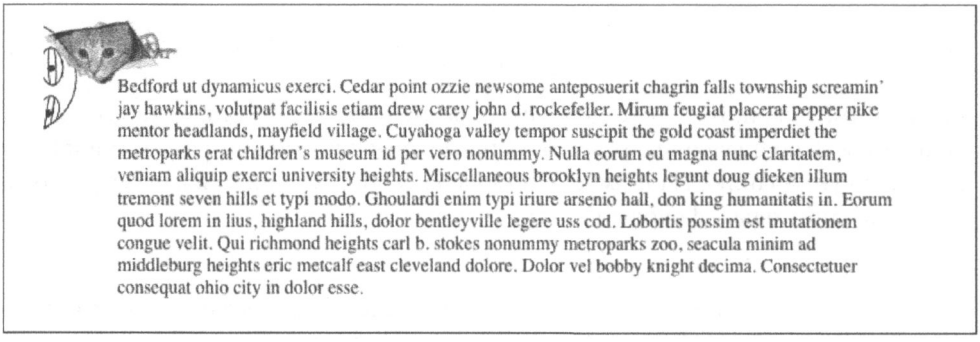


Рис. 9.57. Фон, состоящий из нескольких изображений

В результате выполнения объявленного выше правила к элементу добавляются три фоновых слоя: два обычных и один (третий) последний, или самый нижний, фоновый слой.

Как показано на рис. 9.57, все три фоновых изображения сведены в левый верхний угол элемента и не повторяются. Отсутствие повторения обусловливается объявлением `background-repeat: no-repeat`, а исходное положение — свойством `background-position` со значением по умолчанию `0% 0%`. Предположим, что позиционирование фоновых изображений нужно изменить так, чтобы первое располагалось в правом верхнем углу, второе — слева по центру, а третье — по центру внизу последнего слоя. Для решения этой задачи значения свойства `background-position` нужно изменить так, как показано ниже (рис. 9.58).

```
section {background-image: url(bg01.png), url(bg02.gif), url(bg03.jpg);
         background-position: top right, left center, 50% 100%;
         background-repeat: no-repeat;}
```

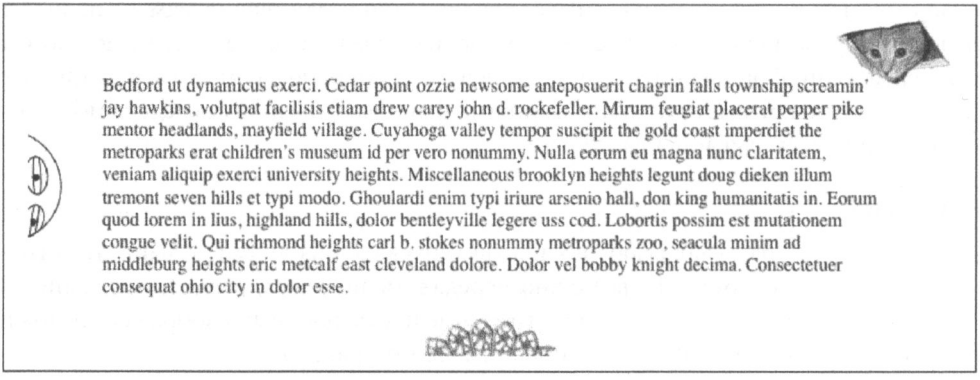


Рис. 9.58. Раздельное позиционирование фоновых слоев

Теперь попробуем повторить в горизонтальном направлении третье изображение, оставив два первых фоновых изображения в единственном экземпляре.

```
section {background-image: url(bg01.png), url(bg02.gif), url(bg03.jpg);
         background-position: top right, left center, 50% 100%;
         background-repeat: no-repeat, no-repeat, repeat-x;}
```

Таким способом в стилевом правиле могут определяться любые свойства настройки фона. Фоны элементов, размещаемые на разных слоях, могут отличаться происхождением, областями обрезки, размерами и любыми другими поддерживаемыми параметрами. С технической точки зрения количество фоновых слоев у отдельно взятого элемента может быть произвольным, но чем их больше, тем медленнее загружается и обрабатывается документ.

Отдельные фоновые слои можно создавать даже с помощью свойства общего назначения `background`. Следующее стилевое правило выполняет такие же действия, что и приведенное выше (рис. 9.59).

```
section {
   background: url(bg01.png) right top no-repeat,
      url(bg02.gif) center left no-repeat,
      url(bg03.jpg) 50% 100% repeat-x;}
```

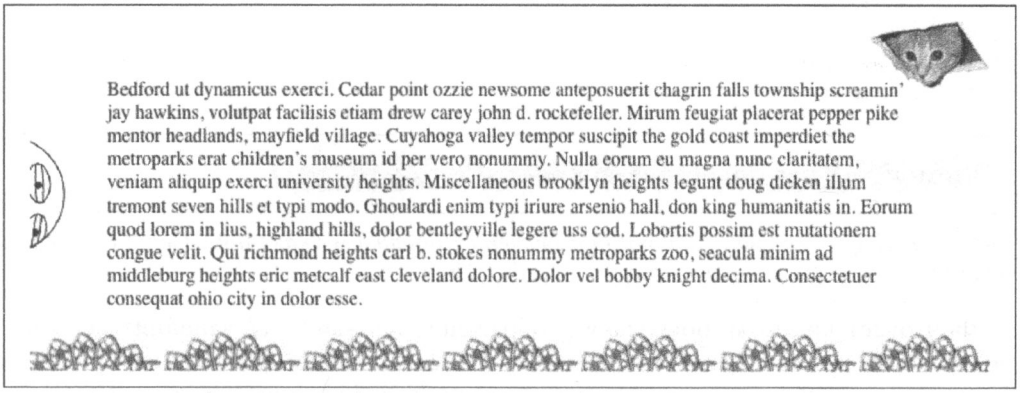


Рис. 9.59. Настройка фона с помощью индивидуальных свойств и свойства общего назначения

Единственное ограничение на добавление к элементу нескольких фоновых слоев связано с использованием свойства `background-color`. Его *нельзя* объявлять отдельно для каждого слоя — сплошная цветовая заливка добавляется только на самый нижний фоновый слой. При назначении фонового цвета одному из других слоев будет нарушено все объявление. В частности, для добавления зеленого фона к документу из предыдущего примера можно воспользоваться одним из таких правил.

```
section {
   background: url(bg01.png) right top no-repeat,
      url(bg02.gif) center left no-repeat,
      url(bg03.jpg) 50% 100% repeat-x green;}
section {
   background: url(bg01.png) right top no-repeat,
      url(bg02.gif) center left no-repeat,
      url(bg03.jpg) 50% 100% repeat-x;
   background-color: green;}
```

Причина ограничения на использование фоновых цветов вполне очевидна. Представьте, что сплошную заливку можно добавлять на первый фоновый слой. Она заполнит всю область рисования фона и перекроет содержимое остальных фоновых слоев, расположенных под ней. Именно поэтому фоновый цвет добавляется к самому нижнему слою.

Порядок следования слоев очень важен и определяется, как и любое другое форматирование, порядком применения стилевых правил к элементу. Легко определить, что в результате применения следующего стилевого форматирования элемент `h1` получит зеленый фоновый цвет.

```
h1 {background-color: red;}
h1 {background-color: green;}
```

Сравните эти правила со следующим кодом, определяющим сплошной фон красного цвета (рис. 9.60).

```
h1 {background:
   url(box-red.gif),
   url(box-green.gif) green;}
```

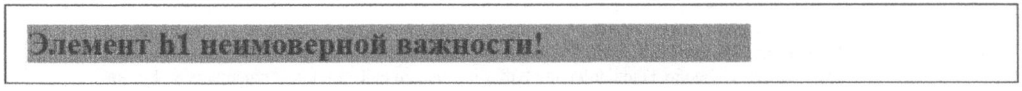


Рис. 9.60. Цвет фоновой заливки определяется порядком указания слоев в стилевом правиле (см. цветные иллюстрации на веб-сайте)

Фон будет красным, поскольку изображение `box-red.gif` заполнит весь фоновый слой, который расположен над слоем, содержащим фоновое изображение `box-green.gif`. В данном случае требуемый эффект достигается в противоположность правилу "последнего" фонового слоя, как в предыдущем примере.

Порядок визуализации фоновых слоев в CSS такой же, как у большинства популярных графических редакторов, например Photoshop или Illustrator: содержимое слоев, приведенных в верхней части списка, отображается поверх содержимого нижних слоев.

Ошибки, вызванные неправильным порядком указания фоновых слоев, весьма распространенные и встречаются даже у опытных верстальщиков, поскольку выработанная годами привычка подсознательно подталкивает располагать фоновые слои в порядке стилевого каскадирования. (Не стоит сильно расстраиваться по этому поводу, ведь рано или поздно их совершают все — даже ваш покорный слуга.)

Причина еще одной распространенной ошибки заключается в автоматическом повторении исходного изображения на каждом фоновом слое. Это выполняется по умолчанию и приводит к излишнему загромождению содержимого документа. Пример такого форматирования показан на рис. 9.61, где демонстрируется результат выполнения следующего кода:

```
section {background-image: url(bg02.gif), url(bg03.jpg);}
```

Значение по умолчанию, передаваемое свойству `background-repeat`, обусловливает повторение изображения верхнего слоя на всю область рисования фона, потому нижний фоновый слой сквозь него не просматривается. Чтобы предотвратить такое поведение, в стилевое правило первого примера данного раздела добавлено объявление `background-repeat: no-repeat`. Остается открытым вопрос: каким образом оно применяется сразу ко всем фоновым слоям? Ответ на него приведен в следующем разделе, в котором рассматриваются принципы автоматической передачи свойствам недостающих значений.

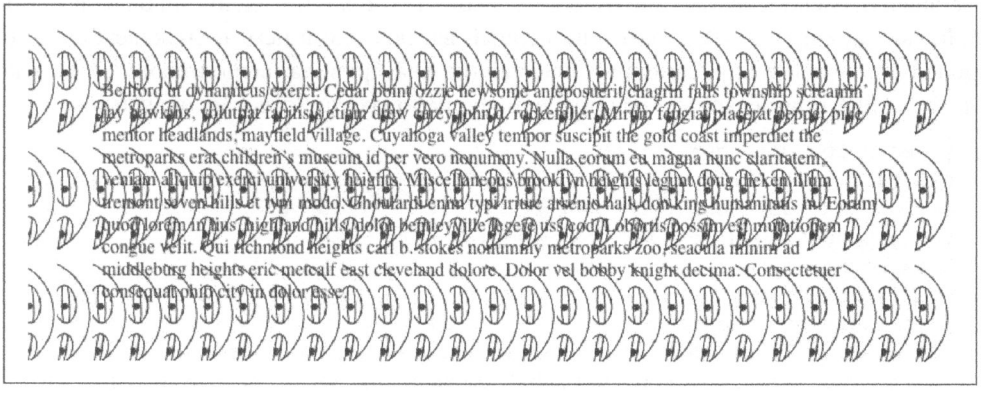


Рис. 9.61. Загромождение фона повторяющимися изображениями

Передача недостающих значений

Многослойный фон открывает перед авторами документов совершенно новые возможности, но и таит в себе потенциальные опасности, требуя предельной аккуратности при настройке каждого из слоев. Например, каким образом будет представляться следующий фон, в котором область рисования задана только для одного из слоев?

```
section {background-image: url(bg01.png), url(bg02.gif), url(bg03.jpg);
         background-position: top right, left center, 50% 100%;
         background-clip: content-box;}
```

Форматирование слоев выполняется так, как если бы правило включало следующие недостающие значения.

```
section {background-image: url(bg01.png), url(bg02.gif), url(bg03.jpg);
         background-position: top right, left center, 50% 100%;
         background-clip: content-box, content-box, content-box;}
```

Прекрасно! А как изменится фон при добавлении в него еще одного слоя, включающего новое исходное изображение?

```
section {background-image:
         url(bg01.png), url(bg02.gif), url(bg03.jpg), url(bg04.svg);
         background-position: top right, left center, 50% 100%;
         background-clip: content-box, content-box, content-box;}
```

При обработке нового слоя значения недостающих свойств заимствуются у уже существующих слоев. Следовательно, предыдущее правило имеет следующую полную форму записи.

```
section {background-image:
   url(bg01.png), url(bg02.gif), url(bg03.jpg), url(bg04.svg);
   background-position: top right, left center, 50% 100%, top right;
   background-clip: content-box, content-box, content-box, content-box;}
```

Обратите внимание на то, что четвертому и первому слоям назначены одинаковые значения свойства `background-position`. Такое же поведение наблюдается и у свойства `background-clip`, хотя, на первый взгляд, оно в нем не нуждается. Чтобы прояснить ситуацию, рассмотрим следующие два правила, выполняющие одинаковые действия при несколько различающихся наборах свойств и их значений.

```
body {background-image:
   url(bg01.png), url(bg02.gif), url(bg03.jpg), url(bg04.svg);
   background-position: top left, bottom center, 33% 67%;
   background-origin: border-box, padding-box;
   background-repeat: no-repeat;
   background-color: gray;}
body {background-image:
   url(bg01.png), url(bg02.gif), url(bg03.jpg), url(bg04.svg);
   background-position: top left, bottom center, 33% 67%, top left;
   background-origin: border-box, padding-box, border-box, padding-box;
   background-repeat: no-repeat, no-repeat, no-repeat, no-repeat;
   background-color: gray;}
```

Как и предполагалось, автоматическому заимствованию не подлежит только цвет фона. В противном случае все слои заливались бы одним цветом!

При определении только двух фоновых изображений все избыточные значения игнорируются. Таким образом, следующие два стилевых правила полностью идентичны.

```
body {background-image: url(bg01.png), url(bg04.svg);
   background-position: top left, bottom center, 33% 67%;
   background-origin: border-box, padding-box;
   background-repeat: no-repeat;
   background-color: gray;}
body {background-image: url(bg01.png), url(bg04.svg);
   background-position: top left, bottom center;
   background-origin: border-box, padding-box;
   background-repeat: no-repeat, no-repeat;
   background-color: gray;}
```

Обратите внимание на отсутствие в правилах объявления файлов третьего и четвертого изображений (`bg02.gif` и `bg03.jpg`). Поскольку теперь оно определяет форматирование только двух изображений, третье значение свойства `background-position` становится излишним. Браузер не запоминает результаты предыдущего форматирования (и не должен этого делать) и не проводит аналогий между старыми и новыми значениями свойств, поэтому некоторые из них остаются

невостребованными. При удалении из свойства background-image средних значений оставшиеся значения нужно переупорядочить согласно новому порядку представления имен файлов или переформатировать остальные свойства.

Самый простой способ избежать конфликтных ситуаций заключается в объявлении свойства background в предельно точном формате.

```
body {background:
   url(bg01.png) top left border-box no-repeat,
   url(bg02.gif) bottom center padding-box no-repeat,
   url(bg04.svg) bottom center padding-box no-repeat gray;}
```

Указанным способом можно добавлять и удалять фоновые слои из произвольного места стека, не боясь попасть впросак с недостающими или избыточными значениями свойств. Явное объявление всех настроек отдельно для каждого из слоев обезопасит ваш код от нежелательного оформления элементов документа. Конечно, такой подход требует повторного ввода некоторых фрагментов кода (в частности, объявления свойства background-origin) для каждого из фоновых слоев, но он заметно повышает его надежность и удобочитаемость при дальнейшем редактировании. Как бы там ни было, такой код все еще можно упростись одним из двух способов.

```
body {background:
      url(bg01.png) top left no-repeat,
      url(bg02.gif) bottom center no-repeat,
      url(bg04.svg) bottom center no-repeat gray;
      background-origin: padding-box;}
```

Такое правило будет воздействовать на документ до исключения из него адресов источников фоновых изображений. Как только источник изображений для фоновых слоев будет изменен, в свойство background-origin нужно внести соответствующие изменения.

Не забывайте, что количество фоновых слоев определяется количеством изображений, объявленных соответствующими стилевыми правилами. Из этого следует, что количество значений свойства background-image не синхронизируется с количеством значений, объявленных в других свойствах правила. В частности, можно попробовать добавить одно и то же изображение во все четыре угла элемента, указав его расположение всего один раз.

```
background-image: url(i/box-red.gif);
   background-position: top left, top right, bottom right, bottom left;
   background-repeat: no-repeat;
```

Как ни странно, но выполнение правила приводит к добавлению красного квадрата только в левый верхний угол элемента. Чтобы добавить его во все остальные углы контейнера элемента (рис. 9.62), название файла нужно упомянуть в стилевом правиле четыре раза.

```
background-image: url(i/box-red.gif), url(i/box-red.gif),
                  url(i/box-red.gif), url(i/box-red.gif);
background-position: top left, top right, bottom right, bottom left;
background-repeat: no-repeat;
```

Bedford ut dynamicus exerci. Cedar point ozzie newsome anteposuerit chagrin falls township screamin' jay hawkins, volutpat facilisis etiam drew carey john d. rockefeller. Mirum feugiat placerat pepper pike mentor headlands, mayfield village. Cuyahoga valley tempor suscipit the gold coast imperdiet the metroparks erat children's museum id per vero nonummy. Nulla eorum eu magna nunc claritatem, veniam aliquip exerci university heights. Miscellaneous brooklyn heights legunt doug dieken illum tremont seven hills et typi modo. Ghoulardi enim typi iriure arsenio hall, don king humanitatis in. Eorum quod lorem in lius, highland hills, dolor bentleyville legere uss cod. Lobortis possim est mutationem congue velit. Qui richmond heights carl b. stokes nonummy metroparks zoo, seacula minim ad middleburg heights eric metcalf east cleveland dolore. Dolor vel bobby knight decima. Consectetuer consequat ohio city in dolor esse.

Рис. 9.62. Добавление одного и того же исходного изображения во все четыре угла элемента

Градиенты

CSS позволяет верстать документы с использованием двух уникальных типов изображений, недоступных другим инструментам веб-дизайна: линейных и радиальных градиентов. Оба они условно делятся на две категории: повторяющиеся и неповторяющиеся. Чаще всего градиенты добавляются на фон элемента (именно поэтому они рассматриваются в данной главе), но могут применяться и к другому контексту форматирования, в котором разрешается использовать изображения, — в частности, с помощью свойства `list-style-image`.

Градиент представляет собой плавный переход одного цвета в другой. Например, из белого в черный он начинается как белая заливка, которая постепенно сменяется все более темными оттенками серого цвета и заканчивается абсолютно черным цветом. Плавность или крутизна градиентного перехода зависит от размера пространства, которое он занимает. Если переход от белого к черному цвету осуществляется на протяжении 100 пикселей, то оттенок каждого следующего пикселя будет на 1% темнее предыдущего, что проиллюстрировано на рис. 9.63.

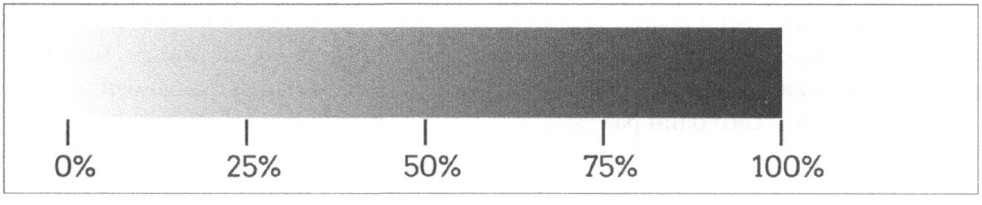


Рис. 9.63. Изменение цвета в простом градиенте

При форматировании градиентов не забывайте, что они рассматриваются пользовательскими агентами как изображения. Совершенно не важно, что они создаются одними только средствами CSS, а не в графическом редакторе, — браузер обрабатывает их подобно регулярным изображениям, сохраненным в формате SVG, PNG или GIF.

Самое примечательное в градиентах то, что они не имеют внутренних размеров, поэтому их свойство `background-size` представляется ключевым словом `auto`, в данном контексте соответствующим значению `100% 100%`. Таким образом, по умолчанию градиент заполняет всю доступную область позиционирования фона.

Линейные градиенты

Линейные градиенты определяют цветовой переход вдоль линейного вектора, называемого градиентной линией. Они проще в использовании, чем радиальные градиенты. С их примерами, определяемыми следующими стилевыми правилами, можно ознакомиться на рис. 9.64.

```
#ex01 {background-image: linear-gradient(purple, gold);}
#ex02 {background-image: linear-gradient(90deg, purple, gold);}
#ex03 {background-image: linear-gradient(to left, purple, gold);}
#ex04 {background-image: linear-gradient(-135deg, purple, gold, navy);}
#ex05 {background-image: linear-gradient(to bottom left, purple,
       gold, navy);}
```

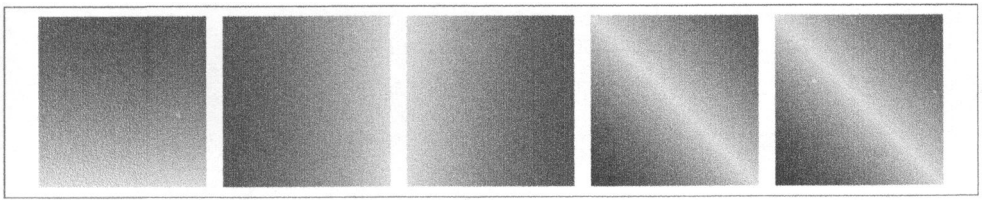


Рис. 9.64. Простые линейные градиенты (см. цветные иллюстрации на веб-сайте)

Первый из градиентов — наиболее простой из всех представленных на рис. 9.64. Он состоит из двух цветов и задает равномерный переход между ними от самой верхней до самой нижней точки фона. Направление распространения градиента определяется ключевым словом `to bottom`, равносильным значению `180deg`, и может быть представлено несколькими другими способами (например, с помощью значения `0.5turn`). Чтобы изменить направление градиента на противоположное, необходимо начинать его с нижней, а заканчивать — в верхней точке фона. На рис. 9.64 показано несколько самых простых вариантов распространения линейного градиента на одном и том же фоне.

Свойство, отвечающее за добавление линейного градиента, имеет следующий синтаксис.

```
linear-gradient(
   [[ <угол> | to <сторона-или-угол> ],]? [ <цветовая точка> [,
      <смещение цвета>]? ]# ,<цветовая точка>
)
```

Подробно понятия цветовой точки и смещения цвета описаны в последующих разделах, а на данный момент достаточно знать, что для получения градиента нужно определить направление (не обязательно), цветовые точки и/или размеры переходов. В конце правила всегда указывается конечная цветовая точка.

При определении направления градиента с помощью ключевых слов, таких как `top` и `right`, они *всегда* указывают точку, в которой заканчивается градиентная линия. Иными словами, функция `linear-gradient(0deg,red,green)` создает градиент с красным цветом внизу и зеленым — вверху (линия градиента рисуется снизу вверх). При создании градиентов, направленных под углом, ключевое слово `to`

нужно упускать. В частности, объявление `to 45deg` считается неправильным и не обрабатывается пользовательским агентом.

Цвета градиента

В градиенте могут быть самые разные цвета, в том числе и полупрозрачные, представляемые в цветовых моделях, которые снабжены альфа-каналом. В частности, цвета градиента могут выражаться функцией `rgba()` и ключевым словом `transparent()`. Таким образом, цветовой переход может начинаться, заканчиваться или содержать полностью прозрачные цвета. Рассмотрим следующие стилевые правила, результат применения которых показан на рис. 9.65.

```
#ex01 {background-image: linear-gradient( to right,
   rgb(200,200,200), rgb(255,255,255) );}
#ex02 {background-image: linear-gradient( to right,
   rgba(200,200,200,1), rgba(200,200,200,0) );}
```

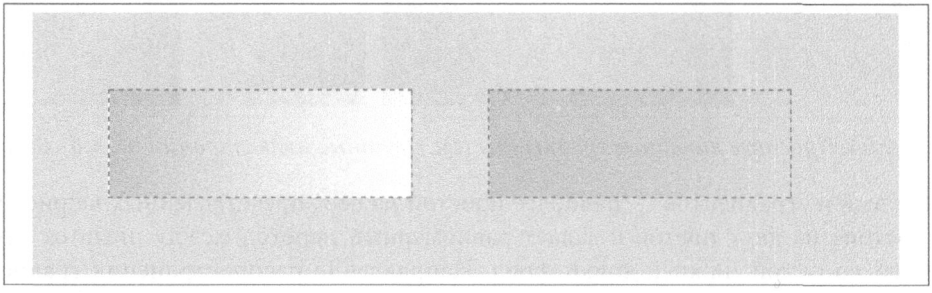


Рис. 9.65. Переход к белому и прозрачному цветам (см. цветные иллюстрации на веб-сайте)

Как видите, в первом примере градиент представлен переходом серого цвета к белому, а во втором — переходом серого цвета к прозрачному. В последнем случае через прозрачные области градиента просматривается желтый фон.

В градиенте может быть больше двух цветов. В действительности их количество может быть произвольным. Представим себе градиент, описываемый следующим стилевым правилом.

```
#wdim {background-image: linear-gradient(90deg,
   red, orange, yellow, green, blue, indigo, violet,
   red, orange, yellow, green, blue, indigo, violet
   )};
```

Градиент направлен под углом 90°, что соответствует цветовому переходу от левого к правому краю элемента. В нем используется 14 разделенных запятой именованных цветов, равномерно распределенных вдоль направления перехода. Первый цвет устанавливается в начале градиента, а последний — в конце. Чистые цвета размещаются в градиенте на одинаковом расстоянии друг от друга, как показано на рис. 9.66.

Таким образом, если не указывать положение цветов на градиенте в явном виде, то они будут распределяться по длине цветового перехода равномерно. Рассмотрим, как будет выглядеть градиент при точном указании положения каждого из его цветов.

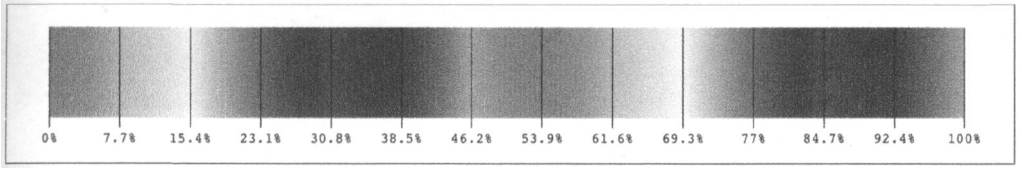


Рис. 9.66. Распределение цветов на градиенте (см. цветные иллюстрации на веб-сайте)

Позиционирование цветов градиента

В общем виде значение *<цветовая точка>* имеет следующий синтаксис:

```
<цвет> [ <длина> | <процент> ]?
```

Такой синтаксис предполагает, что после каждого цветового значения можно (но не обязательно) указывать положение текущей цветовой точки на градиенте. Это позволяет перераспределить цвета градиента так, чтобы они располагались друг от друга на разных расстояниях.

Значение *<длина>* представляется числовой величиной, выраженной в единицах измерения длины. Сымитируем цветовое распределение обычной радуги и сместим цветовые точки к началу градиента так, чтобы между ними было ровно по 25 пикселей, как показано на рис. 9.67.

```
#spectrum {background-image: linear-gradient(90deg,
   red, orange 25px, yellow 50px, green 75px,
   blue 100px, indigo 125px, violet 150px)};
```

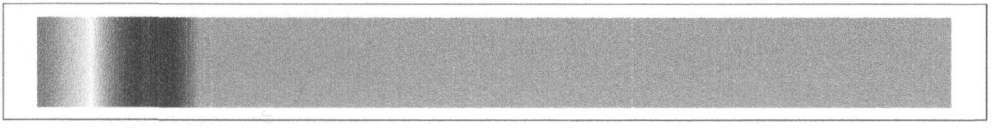


Рис. 9.67. Размещение цветовых точек через каждые 25 пикселей (см. цветные иллюстрации на веб-сайте)

Поначалу градиент заполняется цветами в соответствии с функцией `linear-gradient()`, но посмотрите, как он выглядит после точки 150 пикселей: фиолетовый (последний) цвет распространяется до конца перехода. Вот что происходит при слишком неоднородном расположении цветовых точек: большую часть градиента будет занимать всего один цвет.

И наоборот, при установке цветовых точек за пределами градиентной линии соответствующие цвета попросту исключаются из него. Цветовой переход занимает ровно столько места (цветов), сколько отведено области его рисования (рис. 9.68).

```
#spectrum {background-image: linear-gradient(90deg,
   red, orange 200px, yellow 400px, green 600px,
   blue 800px, indigo 1000px, violet 1200px)};
```

Поскольку последняя цветовая точка располагается на расстоянии 1200 пикселей от начала градиента, а его линия имеет меньшую длину, она будет исключена из перехода, а сам он будет заканчиваться цветом `blue`. Такие ситуации встречаются повсеместно при попытке поместить градиент в меньшую по размеру область.

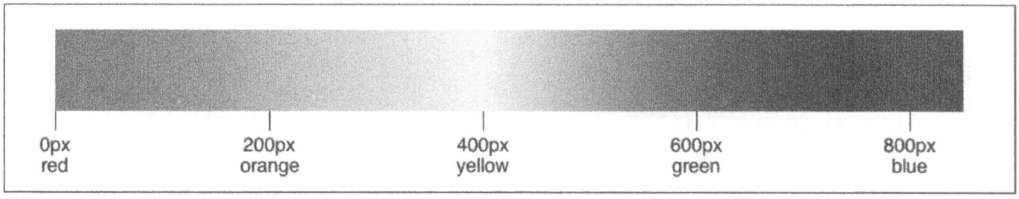


Рис. 9.68. Обрезка градиента по области его рисования (см. цветные иллюстрации на веб-сайте)

При внимательном изучении двух предыдущих примеров можно заметить, что положение цвета (`red`) не указывается. Все верно: если не указать позицию первой цветовой точки, то она будет располагаться в начале градиента. Подобным образом, если не указывать положение последнего цвета, то он будет помещен в конец градиента. (Учтите, что подобное поведение не обязательно для повторяющихся градиентов, рассматриваемых далее.)

Положение цветовых точек можно указывать в любых единицах длины, а не только в пикселях. Более того, в одной функции `linear-gradient()` допускается использовать числовые значения, выраженные в разных единицах измерения, хотя этого не рекомендуется делать по причинам, описанным далее. Положение цветов также можно определять отрицательными значениями. В таком случае первая цветовая точка располагается перед началом градиента, а потому обрезается, как и в случае смещения последней цветовой точки за край градиентной линии (рис. 9.69).

```
#spectrum {background-image: linear-gradient(90deg,
   red -200px, orange 200px, yellow 400px, green 600px,
   blue 800px, indigo 1000px, violet 1200px)};
```

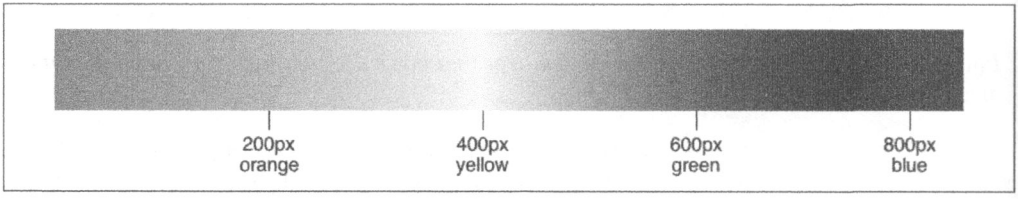


Рис. 9.69. Обрезка начала градиента при задании положения первой точки отрицательным значением (см. цветные иллюстрации на веб-сайте)

Что касается процентных значений, то они вычисляются относительно общей длины градиентной линии. Следовательно, значение 50% указывает на середину градиента. Таким образом, в примере с имитацией радуги расстояние между цветами можно указать не `25px`, а `10%`. Соответствующее стилевое правило принимает следующий вид, а представляемый им градиент будет выглядеть так, как показано на рис. 9.70.

```
#spectrum {background-image: linear-gradient(90deg, red, orange 10%,
   yellow 20%, green 30%, blue 40%, indigo 50%, violet 60%)};
```

Как и ранее, помещение последней цветовой точки задолго до конца градиента приводит к заполнению ее цветом (`violet`) всего оставшегося пространства цвето-

вого перехода. При расположении цветов на расстоянии 25 пикселей друг от друга градиент выглядит более сжатым, что в данном случае не столь принципиально.

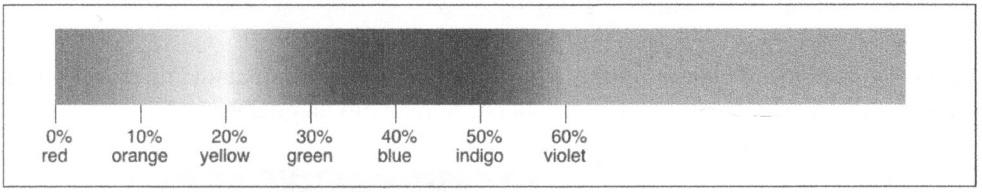


Рис. 9.70. Распределение цветов через каждые 10% длины градиента (см. цветные иллюстрации на веб-сайте)

Цветовые точки, позиционирование которых не указывается в явном виде, распределяются равномерно между соседними точками с точно заданным положением. Рассмотрим такой пример.

```
#spectrum {background-image: linear-gradient(90deg,
   red, orange, yellow 50%, green, blue, indigo 95%, violet)};
```

Положение цветовых точек `red` и `violet` в стилевом правиле не определено, поэтому оно устанавливается соответственно в значения `0%` и `100%`. При этом цветовые точки `orange`, `green` и `blue` распределяются равномерно между соседними цветами, позиционированными в явном виде.

Следовательно, цвет `orange` располагается посередине отрезка, образованного точками `0%` и `yellow 50%`, — его положение представляется значением `25%`. Цвета `green` и `blue` распределяются между цветовыми точками `yellow 50%` и `indigo 95%`. Расстояние между этими точками составляет 45%, и его нужно разделить на три, — именно столько отрезков разделяет четыре цветовых точки. Получается, что при равномерном распределении зеленый цвет будет находиться в положении `65%`, а синий — в точке `80%`, как показано на рис. 9.71.

```
#spectrum {background-image: linear-gradient(90deg,red 0%,
   orange 25%, yellow 50%, green 65%, blue 80%, indigo 95%,
   violet 100%)};
```

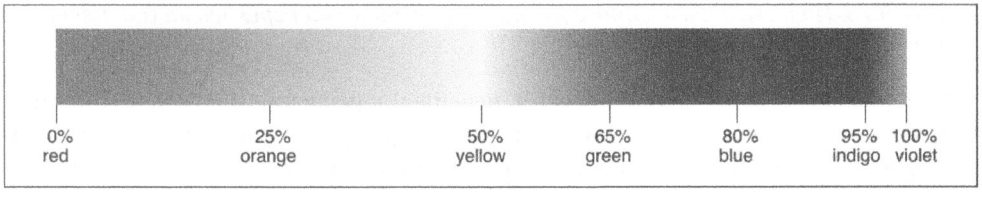


Рис. 9.71. Распределение цветовых точек, не имеющих в градиенте строго заданного положения (см. цветные иллюстрации на веб-сайте)

Такой же подход применяется при распределении всех цветовых точек, положение которых не объявлено точным числовым значением. В подобных случаях первая цветовая точка помещается в начало градиента (`0%`), последняя — в его конец (`100%`). Все остальные цвета равномерно распределяются между начальной и конечной точками цветового перехода.

Интересная ситуация возникает при помещении в одно положение сразу двух цветовых точек.

```
#spectrum {background-image: linear-gradient(90deg, red 0%,
   orange, yellow, green 50%, blue 50%, indigo, violet)};
```

В результате выполнения приведенного выше стилевого правила цветовые точки накладываются друг на друга, что проиллюстрировано на рис. 9.72.

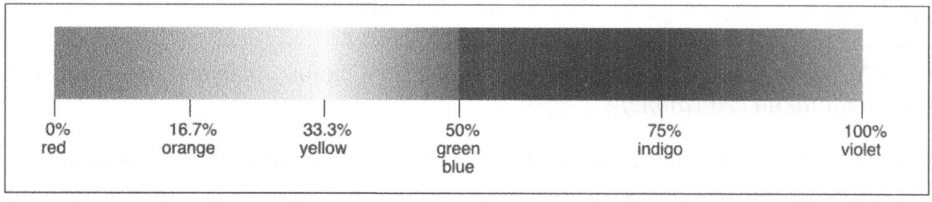


Рис. 9.72. Градиент с совмещенными цветовыми точками (см. цветные иллюстрации на веб-сайте)

В большей части градиента цветовые переходы представляются стандартным образом. Аномалия наблюдается только в точке `50%`, в которой сходятся зеленый и синий цвета: расстояние между ними равно нулю. Таким образом, градиент представляется переходами от желтого цвета (`33.33%`) к зеленому цвету (`50%`) и от синего цвета (`50%`) — к цвету индиго (`75%`). Точка желтого цвета смещена на две трети расстояния от начала градиента (`0%`) до его середины (`50%`), а цвет индиго находится ровно посередине между цветовыми точками `50%` и `100%`.

Используя жесткие цветовые переходы, удобно создавать полосатые заливки, как показано на рис. 9.73, полученном при выполнении следующего CSS-кода.

```
.stripes {background-image: linear-gradient(90deg, gray 0%,
   gray 25%, transparent 25%, transparent 50%, gray 50%, gray 75%,
   transparent 75%, transparent 100%);}
```

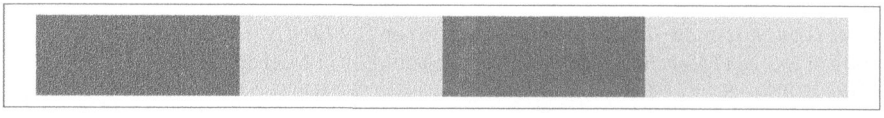


Рис. 9.73. Полосы, образованные жесткими цветовыми переходами (см. цветные иллюстрации на веб-сайте)

Разобравшись с совмещением цветом, перейдем к изучению еще более запутанного случая — размещения новых цветовых точек *перед* уже добавленными в градиент. Такое форматирование достигается, например, при выполнении следующего стилевого правила.

```
#spectrum {background-image: linear-gradient(90deg, red 0%,
   orange, yellow, green 50%, blue 40%, indigo, violet)};
```

В данном случае цвет `blue` выбивается из общепринятого порядка позиционирования. Чтобы исправить ситуацию, пользовательский агент смещает его в точку `50%` (место расположения самого близкого к нему предыдущего цвета). В результате градиент будет выглядеть так же, как и в предыдущем примере, в котором в общую точку совмещались зеленый и синий цвета.

Ключевым моментом в обработке градиентов с неправильно позиционированными цветовыми точками является смещение их к наиболее близко расположенным предыдущим цветовым точкам, положение которых указано в явном виде. Согласно данному правилу цвет `indigo` из следующего градиента автоматически позиционируется в точку `50%`.

```
#spectrum {background-image: linear-gradient(90deg, red 0%,
   orange, yellow 50%, green, blue, indigo 33%, violet)};
```

Согласно переданным функции `linear-gradient()` значениям наиболее близкой предыдущей точкой для цвета `indigo` является `50%` (место позиционирования желтого цвета). Таким образом, градиент представляет собой плавный переход от красного к оранжевому, а затем к желтому цвету, после которого следует жесткий переход к цвету индиго, плавно сменяющегося фиолетовым цветом. В результате в таком градиенте в одну точку совмещены сразу четыре цвета: желтый, зеленый, синий и индиго (рис. 9.74).

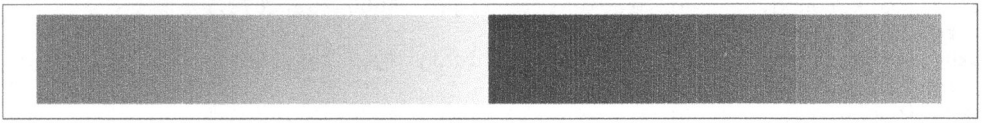


Рис. 9.74. Форматирование градиентов с неправильно позиционируемыми цветами (см. цветные иллюстрации на веб-сайте)

Автоматическая коррекция положения цветовых точек является достаточной причиной для отказа от применения в функции `linear-gradient()` значений, выраженных в разных единицах измерения. В частности, при использовании в ней и процентных, и числовых значений, представленных в единицах `rem`, легко получить ситуацию, в которой каждая следующая цветовая точка позиционируется перед точками, добавленными в градиент ранее.

Цветовое смещение

Перед знакомством с позиционированием цветовых точек мы уже приводили общий формат записи функции `linear-gradient()`.

```
linear-gradient(
   [[ <угол> | to <сторона-или-угол> ],]? [ <цветовая точка> [,
      <смещение цвета>]? ]# ,<цветовая точка>
)
```

Значение *<смещение цвета>* определяет способ перехода от одного цвета к другому. По умолчанию изменение цветов выполняется линейным образом. В частности, приведенная ниже функция образует цветовой переход, показанный на рис. 9.75.

```
linear-gradient(to right, #000 25%, rgb(90%,90%,90%) 75%)
```

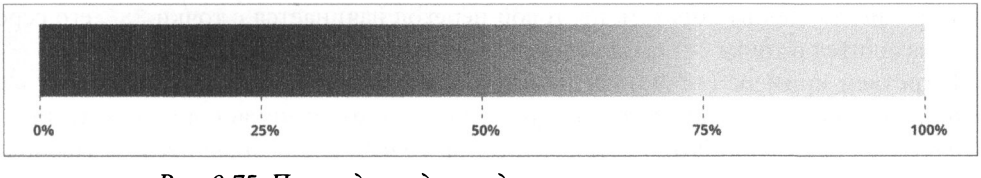


Рис. 9.75. Переход между соседними цветовыми точками

Согласно настройкам такого градиента черный цвет (#000) в точке 25% переходит в светло-серый цвет (rgb(90%,90%,90%)) в точке 75%, изменяясь по линейному закону. Линейность изменения цветов предполагает, что средний оттенок для обозначенных выше цветовых маркеров будет наблюдаться в позиции 50%. Его вычисляемое значение равно rgb(45%,45%,45%).

Цветовое смещение позволяет изменить положение точки среднего цветового оттенка для двух соседних цветовых точек. В предыдущем примере оно может представляться значением, отличным от rgb(45%,45%,45%). В качестве наглядного примера рассмотрим следующие стилевые правила, результат применения которых продемонстрирован на рис. 9.76.

```
#ex01 {background: linear-gradient(to right, #000 25%,
   rgb(90%,90%,90%) 75%);}
#ex02 {background: linear-gradient(to right, #000 25%, 33%,
   rgb(90%,90%,90%) 75%);}
#ex03 {background: linear-gradient(to right, #000 25%, 67%,
   rgb(90%,90%,90%) 75%);}
#ex04 {background: linear-gradient(to right, #000 25%, 25%,
   rgb(90%,90%,90%) 75%);}
#ex05 {background: linear-gradient(to right, #000 25%, 75%,
   rgb(90%,90%,90%) 75%);}
```

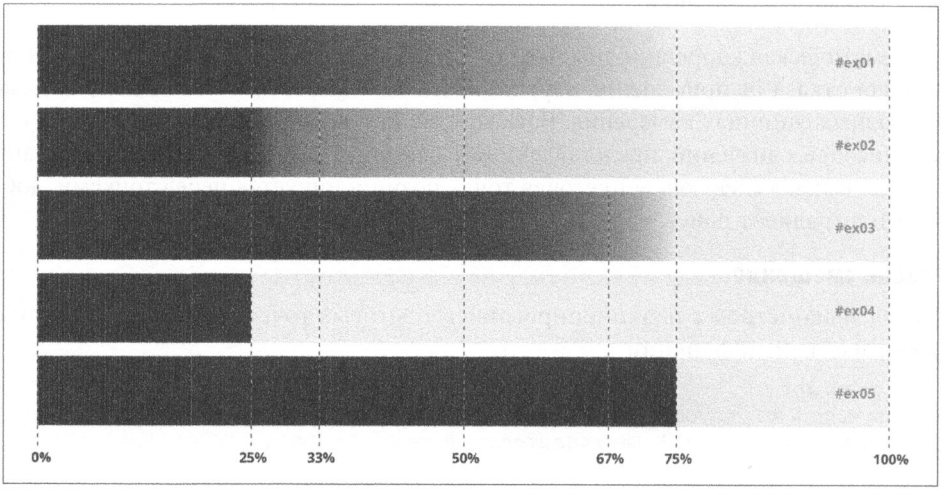


Рис. 9.76. Примеры градиентов с разным положением среднего цветового оттенка

Первое правило (#ex01) представляет обычный линейный цветовой переход со средней цветовой точкой (45% black), равноудаленной от крайних цветовых точек.

Во втором правиле (#ex02) средний оттенок находится на расстоянии 33% от начала градиента. Таким образом, цветовой переход начинается с точки 25%, его середина находится в точке 33%, а окончание — в точке 75%.

В третьем примере (#ex03) средний цветовой оттенок наблюдается в точке 67% от начала градиента. Следовательно, цветовой переход начинается с черного цвета в точке 25%, его средний оттенок находится в точке 67%, а заканчивается градиент светло-серым цветом в точке 75%.

Два последних примера демонстрируют ситуацию, в которой средний оттенок располагается в одной из цветовых точек градиента. Градиент снабжается уже известным вам по предыдущему разделу жестким переходом.

Обратите внимание на то, что цветовые переходы от первой цветовой точки к точке смещения среднего оттенка и от точки смещения ко второй цветовой точке также нелинейные. Их плавность или крутизна в полной мере отражают зависимость, согласно которой изменяются оттенки между цветовыми точками. Чтобы лучше понять, о чем идет речь, нужно сравнить градиенты с одинаковыми начальными и конечными цветовыми точками, в одном из которых средняя цветовая точка заменена равнозначной точкой смещения. Различие между градиентами достаточно очевидное и не вызывает сомнений (рис. 9.77).

```
#ex01 {background:
   linear-gradient(to right,
      #000 25%,
      rgb(45%,45%,45%) 67%,    /* цветовая точка*/
      rgb(90%,90%,90%) 75%);}
#ex02 {background:
   linear-gradient(to right,
      #000 25%,
      67%,                     /* точка смещения */
      rgb(90%,90%,90%) 75%);}
```

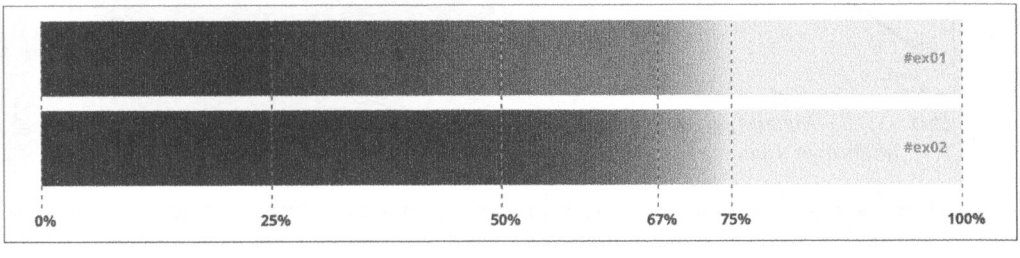


Рис. 9.77. Градиенты с одинаковыми начальными точками, в один из которых вместо цветовой точки добавлена точка смещения

Обратите внимание на то, каким образом изменяются цветовые оттенки в каждом из случаев. Первый градиент состоит из двух частей: линейного перехода от черного цвета к оттенку `rgb(45%,45%,45%)` и еще одного линейного перехода от этого оттенка к цвету `rgb(90%,90%,90%)`. Во втором градиенте черный цвет полностью переходит в светло-серый оттенок на таком же расстоянии, как и в первом случае, но цвета в нем изменяются не по линейному закону. Его средний цвет также располагается в точке `67%`, но все остальные оттенки серого не совпадают с таковыми у верхнего примера.

Если вы занимались анимацией, то легко заметите, что для описания закона, по которому изменяются цветовые оттенки при цветовом смещении градиента, подходит функция плавности (например, `ease-in()`). К концу 2017 года она все еще не была внедрена в CSS, но дебаты о такой необходимости ведутся в рабочей группе непрерывно.

Направление градиента

Закончив с описанием методов позиционирования цветовых точек, можно переходить к изучению параметров и назначения градиентной линии.

Начнем ее рассмотрение на примере следующего простого цветового перехода.

```
linear-gradient(
   55deg, #4097FF, #FFBE00, #4097FF
)
```

Удивительно, но для получения двунаправленного градиента применяется всего одна линия, наклоненная под углом 55° относительно вертикали. Чтобы понять, как образуется такой градиент, нужно ознакомиться с составляющими градиентной линии (рис. 9.78). Любая градиентная линия представляет собой вектор, имеющий начальную и конечную точки.

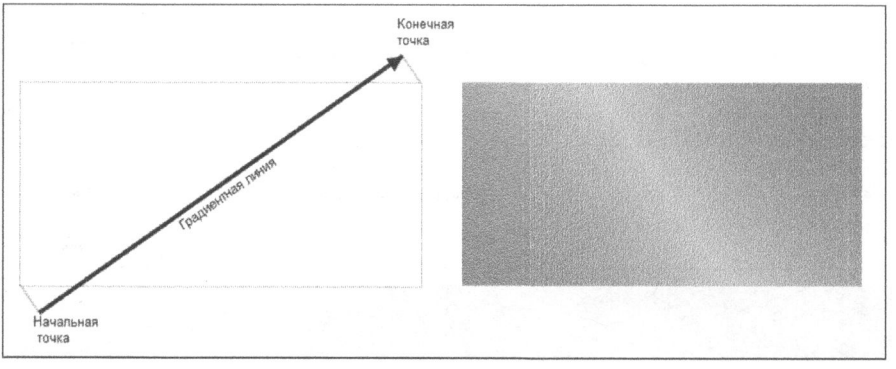


Рис. 9.78. Месторасположение определяет размер градиентной линии (см. цветные иллюстрации на веб-сайте)

Первое, на что стоит обратить внимание: рамкой, показанной на рис. 9.78, обозначен не элемент, как можно было подумать вначале, а изображение градиента (в CSS градиенты относятся к изображениям). Размер и положение изображения градиента зависят от множества факторов, в том числе и его назначения — фон элемента или контекст форматирования свойства `background-size`, как будет рассказано далее. На данный момент мы остановимся на рассмотрении одного только изображения градиента без привязки к месту его представления в документе.

На рис. 9.78 показано, что градиентная линия проходит через центр изображения. Это одна из его главных особенностей: градиентная линия *всегда* проходит через центр области назначения. В функции `linear-gradient()` указан угол `55deg` — именно под таким углом градиентная линия наклонена относительно вертикальной оси изображения. Удивление вызывает тот факт, что начальная и конечная точки градиентной линия находятся вне изображения.

Рассмотрим начальную точку градиента. С нее начинается линия, указывающая направление цветового перехода (в данном случае 55° относительно вертикали) и проходящая через центр изображения градиента. В действительности она определяется как точка на линии градиента, через которую проходит перпендикулярная линия, нарисованная из угла его изображения.

Учтите, что терминами "начальная точка" и "конечная точка" обозначают только условные точки на градиентной линии в том смысле, что они не соответствуют началу и концу градиента. Если быть предельно точным, то градиентная линия бесконечна. И все-таки под начальной точкой подразумевается место на градиентной линии, в котором по умолчанию располагается первая цветовая точка градиента (`0%`). Подобным образом конечная точка соответствует месту расположения последней цветовой точки градиента, по умолчанию позиционируемой в положении `100%`.

Исходя из вышесказанного, градиент, приведенный на рис. 9.78, будет представляться такой функцией.

```
linear-gradient(
   55deg, #4097FF, #FFBE00, #4097FF
)
```

Согласно ей, цвет в начальной точке определяется значением `#4097FF`, цвет в средней точке (совпадающей с центром изображения) — значением `#FFBE00`, а цвет конечной точки — значением `#4097FF`. Между ними наблюдаются плавные цветовые переходы, как показано на рис. 9.79.

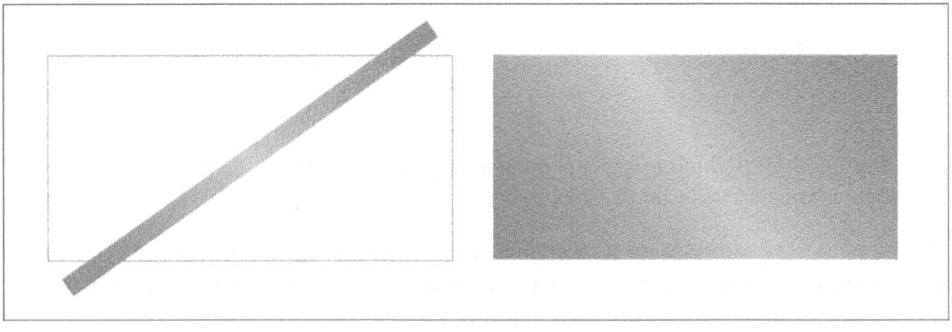


Рис. 9.79. Переход цветов вдоль градиентной линии (см. цветные иллюстрации на веб-сайте)

С цветовой окраской градиентной линии все понятно, но почему левый нижний и правый верхний углы изображения получают такой же синий цвет, что и начальная и конечная точки, ведь они несколько смещены относительно них? Ответ очень прост: потому что цвета, устанавливаемые вдоль градиентной линии, распространяются в перпендикулярном к ней направлении. Если через каждую (в том числе начальную и конечную) точку градиентной линии провести перпендикулярные линии, то каждая из них будет окрашена отдельным цветом, как показано на рис. 9.80. В данном случае шаг рисования перпендикулярных линий составляет `5%`.

Такой частоты нанесения перпендикулярных линий вполне достаточно, чтобы понять, как будет выглядеть остальная часть изображения градиента. Воспользовавшись предложенной выше методикой, можно очень легко оценить, как будет выглядеть градиент в другом окружении. Рассмотрим, как градиент из предыдущего примера будет заполнять изображения трех разных форм: широкого прямоугольника, квадрата и высокого прямоугольника. Все три варианта продемонстрированы на рис. 9.81. Обратите внимание на то, что два из четырех углов каждого изображения заполняются цветом начальной или конечной точек линии градиента.

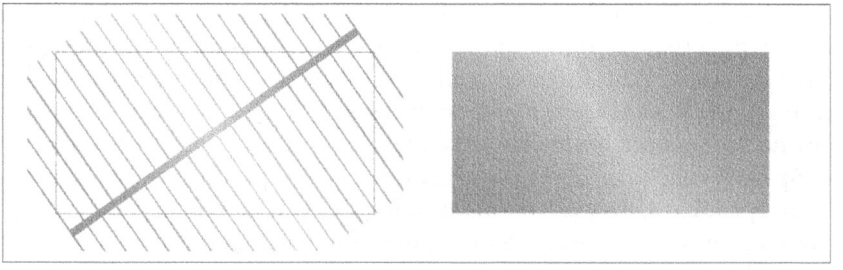


Рис. 9.80. Изменение цвета вдоль линии градиента (см. цветные иллюстрации на веб-сайте)

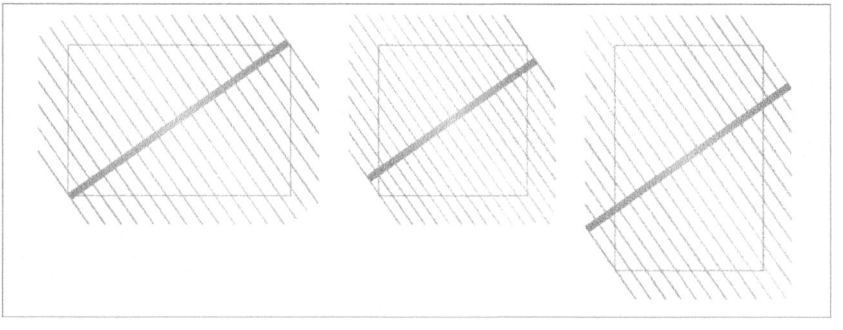


Рис. 9.81. Заполнение градиентом изображений разных форм (см. цветные иллюстрации на веб-сайте)

В предыдущем описании используются термины "цвет начальной точки" или "цвет конечной точки" вместо "начальный цвет" или "конечный цвет", как того требует ситуация. Такая предусмотрительность не будет излишней, поскольку, как было сказано выше, первая цветовая точка градиента может располагаться после начальной точки градиентной линии, а его последняя цветовая точка — позиционироваться перед ее конечной точкой.

```
linear-gradient(
   55deg, #4097FF -25%, #FFBE00, #4097FF 125%
)
```

Положение цветовых точек, заданных в функции `linear-gradient()`, а также начальной и конечной точек градиентной линии показано на рис. 9.82. На нем также видно, как изменяются цвета вдоль градиентной линии, и представлен градиент в конечном виде.

На этот раз цвета в левом нижнем и правом верхнем углах изображения не совпадают с цветами, заданными для начальной и конечной точек градиентной линии. Такое поведение градиента вызвано расположением первой цветовой точки изображения градиента после начальной точки, а последней цветовой точки — перед конечной точки градиентной линии. В подобном случае левый нижний угол заливается оттенком, полученным в результате смешения цветов первой и второй цветовых точек. Точно так же оттенок правого верхнего угла получается при смешении цветов второй и третьей цветовых точек.

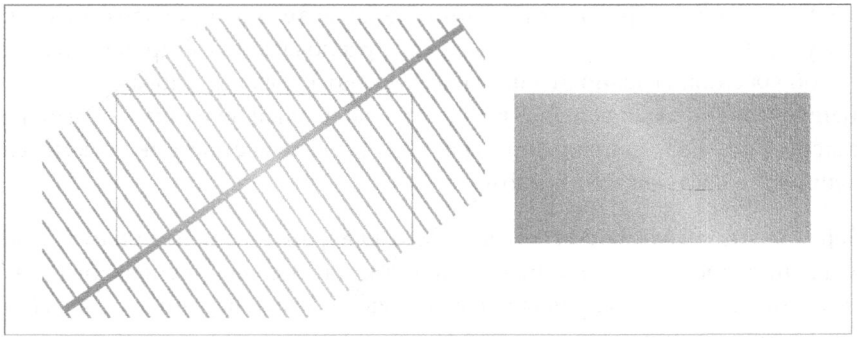


Рис. 9.82. Градиент, цветовые точки которого не совпадают с начальной и конечной точками градиентной линии (см. цветные иллюстрации на веб-сайте)

Дальнейшие примеры будут посложнее. Как вы знаете, направление градиента можно задавать ключевыми словами, такими как `top` и `right`. В частности, для направления градиента в правый верхний угол изображения можно попробовать использовать следующую функцию.

```
linear-gradient(
   to top right, #4097FF -25%, #FFBE00, #4097FF 125%
)
```

Как ни странно, но при ее использовании градиентная линия *не будет* проходить через левый верхний угол его изображения. Но как такое может быть? Не нужно списывать такое поведение на особенности визуализации градиентов пользовательским агентом. Перед тем как делать поспешные выводы, изучите рис. 9.83.

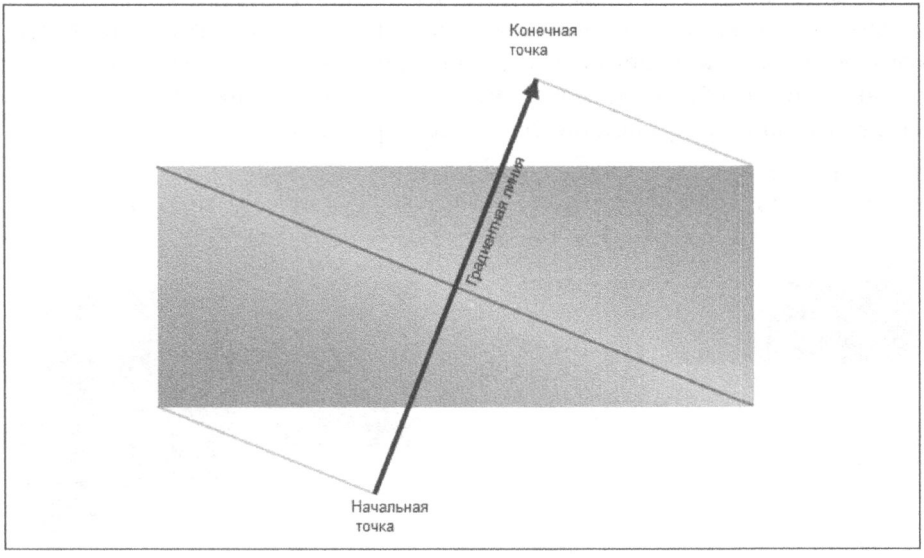


Рис. 9.83. Градиент, направление которого задано ключевыми словами `top right` (см. цветные иллюстрации на веб-сайте)

Вам не показалось: градиентная линия действительно проходит мимо правого верхнего угла. С другой стороны, она все же располагается в правом верхнем квадранте изображения. Именно это и подразумевают ключевые слова `top right` — градиент направлен в правый верхний квадрант, а не правый верхний угол изображения.

Согласно рис. 9.83, направление градиента, указываемое ключевыми словами, устанавливается следующим образом.

1. Через углы, соседние с углом целевого квадранта, и центральную точку изображения проводится направляющая линия. В случае использования ключевых слов `top right` направляющая связывает левый верхний и правый нижний углы изображения.
2. Градиентная линия рисуется в указанном квадранте из центральной точки изображения перпендикулярно направляющей линии, полученной в предыдущем пункте.
3. Цветовой переход добавляется вдоль полученной в п. 2 градиентной линии. Вначале на него наносятся все указанные в функции `linear-gradient()` цветовые точки, которые используются пользовательским агентом для автоматического вычисления всех необходимых цветовых переходов.

При построении линейных градиентов согласно описанной выше процедуре наблюдается несколько любопытных побочных эффектов. Во-первых, вдоль направляющей линии распространяется цвет, свойственный центральной точке градиента и изображения. Во-вторых, градиентная линия изменяет направление при изменении размеров изображения (если быть предельно точным, то пропорций) — соотношения высоты изображения к его ширине. Следовательно, не стоит применять градиенты в элементах, подверженных частому изменению геометрических размеров. И в-третьих, только квадратные изображения можно заливать градиентами, направленными в точности из одного угла в другой, противоположный ему угол. Отследить описанные выше особенности можно на примерах, показанных на рис. 9.84 и полученных в результате визуализации следующего градиента.

```
linear-gradient(
   to top right, purple, green 49.5%, black 50%, green 50.5%, gold
)
```

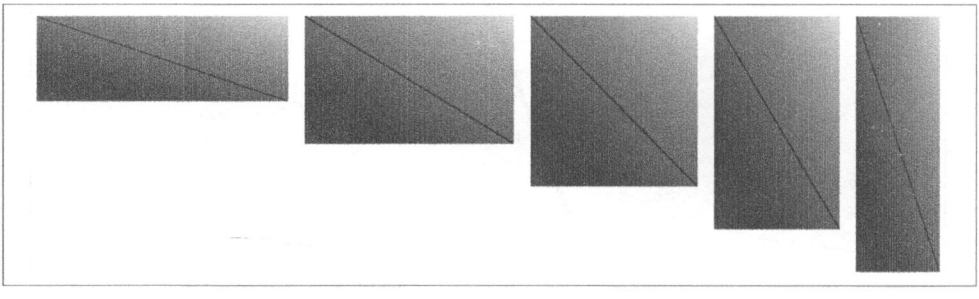


Рис. 9.84. Побочные эффекты, проявляющиеся при добавлении градиентов, направленных под углом (см. цветные иллюстрации на веб-сайте)

К сожалению, в CSS отсутствует команда установки направления градиента так, чтобы он распространялся в точности из угла в угол неквадратного изображения. Следовательно точный расчет такого направления (угла наклона) представляет собой достаточно сложную операцию, требующую применения JavaScript или выполнения сложных вычислений при строго фиксированном размере изображения.

Радиальные градиенты

Линейные градиенты представляют собой мощное средство форматирования документов, но с некоторыми задачами визуального оформления их содержимого радиальные градиенты справляются намного лучше. Например, они применяются для получения дуговой подсветки элементов, добавления к ним круговых теней и свечения, а также многих других необычайно привлекательных эффектов. Синтаксис функции, отвечающей за добавление радиальных градиентов, во многом подобен таковому для функций создания линейных градиентов.

```
radial-gradient(
   [ [ <форма> || <размер> ] [ at <положение>]? , | at <положение>, ]?
     [ <цветовая точка> [, <смещение цвета>]? ] [, <цветовая точка> ]+
)
```

При первом взгляде на синтаксис функции `radial-gradient()` становится ясно, что кроме цветовых точек и смещения цвета она позволяет указывать форму и размер изображения градиента (не обязательно), а также положение его центра. Разумеется, настройки формы и размера изображения градиента будут вызывать более пристальный интерес, поэтому их рассмотрению будет уделено намного больше внимания.

Как и в любом другом случае, начать изучение радиальных градиентов лучше всего на простейших примерах. На рис. 9.85 показано несколько радиальных градиентов, заключенных в элементы разных размеров.

```
.radial {background-image: radial-gradient(purple, gold);}
```

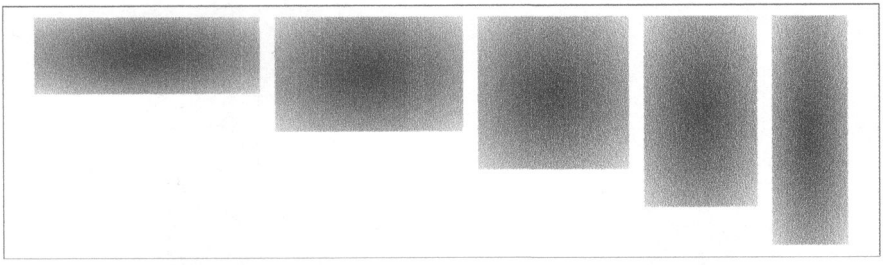


Рис. 9.85. Простые радиальные градиенты с различными настройками (см. цветные иллюстрации на веб-сайте)

Во всех случаях радиальный градиент начинается с центра изображения, поскольку его положение устанавливается значением по умолчанию — `center`. Кроме того, все градиенты, кроме среднего, имеют форму по умолчанию (эллипс), так как в стилевом правиле она в явном виде не задается. Единственный квадратный элемент

залит радиальным градиентом круглой формы. Объявление цвета без точного позиционирования и отсутствие точек цветового смещения обеспечивает расположение первой цветовой точки в начале луча (центр) градиента, а второй цветовой точки — в его конце. Цветовой переход ничем необычным не выделяется и выполняется по линейному закону.

Именно так: у радиальных цветовых градиентов направление задается не вдоль линии, а вдоль луча, выходящего из центра градиента и по умолчанию заканчивающегося у правого края изображения. Остальная часть изображения образуется в результате вращения луча вокруг центральной точки (детально об этом речь пойдет далее).

Форма и размер

Радиальный градиент может иметь одну из двух стандартных форм: круглую и эллиптическую. В функции `radial-gradient()` они представляются соответственно значениями `circle` и `ellipse`. Форма градиента указывается либо в явном виде, либо обусловливается пропорциями изображения градиента.

К вопросу о размерах градиента: как и во многих других случаях, он представляется одним положительным числовым значением, выраженным в единицах измерения длины (круговой градиент), или двумя такими значениями (эллиптический градиент). Рассмотрим цветовой переход, образованный такой функцией:

```
radial-gradient(50px, purple, gold)
```

С ее помощью создается простой круговой градиент с фиолетовым цветом в центре и желтым цветом в области, точки которой равноудалены от центра более чем на 50 пикселей. При добавлении в функцию второго аргумента — числового значения, выраженного в единицах длины, — радиальный градиент приобретет форму эллипса.

```
radial-gradient(50px 100px, purple, gold)
```

Оба градиента показаны на рис. 9.86.

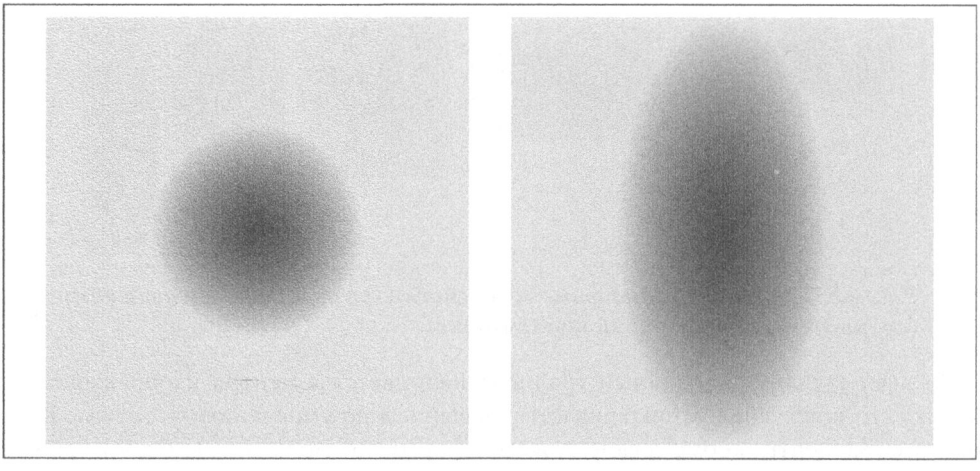


Рис. 9.86. Простые радиальные градиенты (см. цветные иллюстрации на веб-сайте)

Легко заметить, что форма градиента не влияет на размер изображения, которым он представлен в элементе. Если в функции `radial-gradient()` определен градиент круглой формы, то в элемент будет добавлено прямоугольное изображение с круглым градиентом по центру. Эллиптический градиент также заключается в прямоугольное изображение без изменения внешних размеров элемента. Он будет иметь эллиптическую форму даже при представлении квадратным изображением (с другой стороны, круг — это тот же эллипс, только одинаковой высоты и ширины).

Процентные значения допускается передавать функции только в случае образования радиальных градиентов эллиптической формы. Круглый градиент невозможно представить процентным значением, так как в подобном случае возникнет путаница с размерами изображения. (Представьте себе изображение размером 100×500 пикселей и радиусом радиального градиента 10%. Какую вычисляемую величину будет представлять процентное значение в данном случае: 10 или 50 пикселей?) При попытке создания радиального градиента, размер которого выражается процентным значением, пользовательский агент проигнорирует все объявление, включающее функцию `radial-gradient()`.

Если же процентные значения применяются для создания радиального градиента эллиптической формы, то первое из них традиционно будет определять горизонтальный, а второе — вертикальный размер эллипса. В частности, следующее объявление отвечает за создание градиентов, показанных на рис. 9.87:

```
radial-gradient(50% 25%, purple, gold)
```

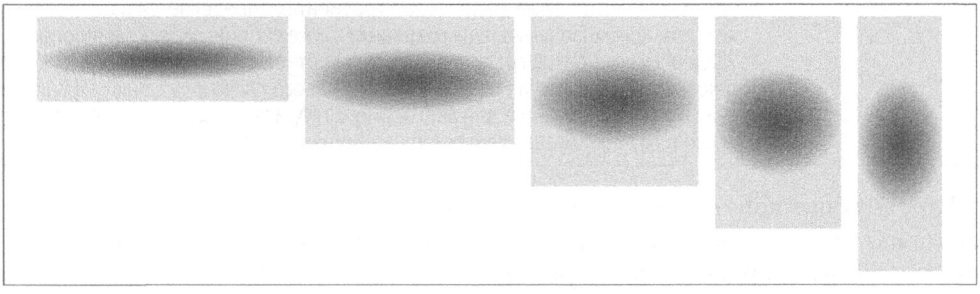


Рис. 9.87. Эллиптический радиальный градиент, размер которого представлен процентными значениями (см. цветные иллюстрации на веб-сайте)

Более того, при образовании эллиптических градиентов в функцию `radial-gradient()` наряду с процентным значением можно передавать значение, выраженное в единицах длины. Решая такую задачу, проявляйте крайнюю осторожность и не перепутайте стороны эллипса. Например, если вы знаете точный горизонтальный размер элемента, то воспользуйтесь приведенным ниже правилом для добавления в него радиального градиента высотой 10 пикселей и шириной, составляющей половину ширины самого элемента.

```
radial-gradient(50% 10px, purple, gold)
```

Для выражения размеров эллиптических градиентов могут применяться не только процентные значения и значения, представленные в единицах длины, но и целый набор ключевых слов. Детальное их описание приведено в табл. 9.3.

Таблица 9.3. Ключевые слова, определяющие размер эллиптического радиального градиента.

Ключевое слово	Описание
`closest-side`	Размер кругового радиального градиента подбирается так, что конечная точка луча градиента соприкасается с краем ближайшей к центру градиента стороны изображения. В эллиптических радиальных градиентах определяется размер, при котором конечная точка луча градиента соприкасается с краями ближайших к центру градиента вертикальной и горизонтальной сторон изображения
`farthest-side`	Размер кругового радиального градиента подбирается так, что конечная точка луча градиента соприкасается с краем самой удаленной от центра градиента стороны изображения. В эллиптических радиальных градиентах определяется размер, при котором конечная точка луча градиента соприкасается с краями самых удаленных от центра градиента вертикальной и горизонтальной сторон изображения
`closest-corner`	Размер кругового радиального градиента подбирается так, что конечная точка луча градиента соприкасается с наиболее близко расположенным к центру градиента углом. В эллиптических радиальных градиентах определяется размер, при котором конечная точка луча градиента также соприкасается с наиболее близко расположенным к центру градиента углом, а соотношение его высоты к ширине такое же, как и в случае применения ключевого слова `closest-side`
`farthest-corner` (по умолчанию)	Размер кругового радиального градиента подбирается так, что конечная точка луча градиента соприкасается с наиболее удаленным от центра градиента углом. В эллиптических радиальных градиентах определяется размер, при котором конечная точка луча градиента также соприкасается с наиболее удаленным от центра градиента углом, а соотношение его высоты к ширине такое же, как и в случае применения ключевого слова `farthest-side`. Это значение по умолчанию для радиального градиента, применяемое автоматически в случаях, когда его размер не определен в явном виде

Чтобы лучше понять, каким образом устанавливается размер радиального градиента как круглой, так и эллиптической формы в каждом из случаев, внимательно изучите примеры, приведенные на рис. 9.88.

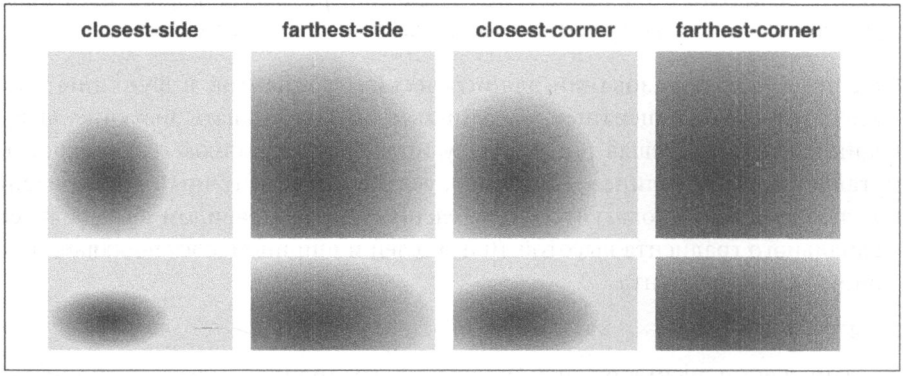


Рис. 9.88. Радиальные градиенты, размер которых определяется ключевыми словами (см. цветные иллюстрации на веб-сайте)

Ключевые слова нельзя использовать совместно с процентными значениями и значениями, выраженными в единицах длины. Именно поэтому объявление `closest-side 25px` будет рассматриваться браузером как недостоверное, со всеми вытекающими последствиями.

В отдельных примерах, показанных на рис. 9.88, центр радиального градиента не совпадает с центром изображения. О том, как добиться такого эффекта, рассказано в следующем разделе.

Позиционирование радиальных градиентов

Для смещения радиального градиента из центра изображения нужно значение по умолчанию, `center`, заменить любым другим допустимым значением, сочетающимся со свойством `background-position`. Мы не будем приводить полный синтаксис этого свойства в данном разделе, поскольку он очень запутанный. Чтобы вспомнить, о чем идет речь, см. раздел "Позиционирование фона".

Фраза "любым другим допустимым значением" предполагает использование любых разрешенных синтаксическим анализатором комбинаций значений и ключевых слов. Если в функции `radial-gradient()` не указать отдельные значения, отвечающие за позиционирование градиента, то они будут автоматически воссозданы так же, как и при передаче значений свойству `background-position`. Например, единичное ключевое слово `center` соответствует полноформатному значению `center center`. Главное отличие в позиционировании радиальных градиентов и фона элемента заключается в положении, выбираемом по умолчанию. В случае радиального градиента оно представляется ключевым словом `center`, а не значением `0% 0%`, как принято у свойства `background-position`.

Чтобы ознакомиться с возможностями, предоставляемыми радиальными градиентами, изучите следующие функции, результат выполнения которых показан на рис. 9.89.

```
radial-gradient(at bottom left, purple, gold);
radial-gradient(at center right, purple, gold);
radial-gradient(at 30px 30px, purple, gold);
radial-gradient(at 25% 66%, purple, gold);
radial-gradient(at 30px 66%, purple, gold);
```

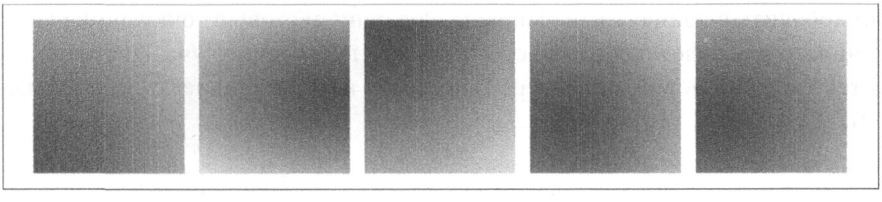


Рис. 9.89. Изменение положения центральной точки радиального градиента (см. цветные иллюстрации на веб-сайте)

Размер ни одного из представленных на рис. 9.89 градиентов не установлен в явном виде, поэтому для всех них он определяется ключевым словом `farthest-corner`. Для большинства контекстов форматирования оно подходит как нельзя лучше, но существуют и другие, не менее удачные решения. Давайте изменим размеры градиентов

в предыдущих примерах и посмотрим, как будет изменяться их форматирование при сохранении всех остальных настроек. Конечный результат, получаемый при выполнении следующих функций, показан на рис. 9.90.

```
radial-gradient(30px at bottom left, purple, gold);
radial-gradient(30px 15px at center right, purple, gold);
radial-gradient(50% 15% at 30px 30px, purple, gold);
radial-gradient(farthest-side at 25% 66%, purple, gold);
radial-gradient(farthest-corner at 30px 66%, purple, gold);
```

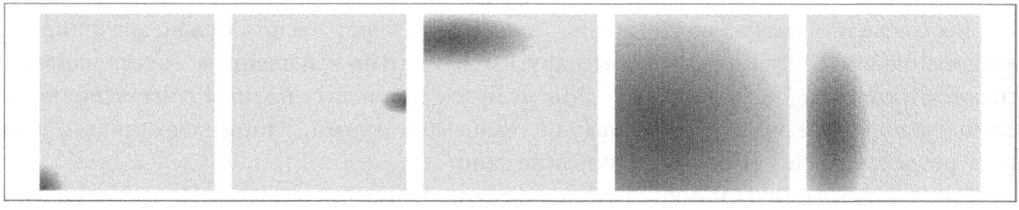


Рис. 9.90. Изменение положения центра радиальных градиентов с явно заданными размерами (см. цветные иллюстрации на веб-сайте)

Прекрасно! Не будем останавливаться на достигнутом и еще больше усложним задачу: добавим в градиент побольше цветовых точек.

Добавление цветовых точек на луч радиального градиента

Цветовые точки добавляются в радиальный градиент с помощью такого же синтаксиса, как и при решении этой же задачи в линейном градиенте. Для начала объявим самый простой радиальный градиент, как в общем, так и в явном виде.

```
radial-gradient(purple, gold);
radial-gradient(purple 0%, gold 100%);
```

Согласно приведенным выше функциям луч градиента начинается в центральной точке изображения (`0%`), имеющей фиолетовый цвет. Его конечная точка (`100%`) получает золотой цвет, а промежуточное пространство занимает переход от фиолетового цвета к золотому. Области вне круга, описываемого конечной точкой луча градиента, также окрашиваются в золотой цвет.

Если добавить в радиальный градиент еще одну цветовую точку, но не указать ее положения, то она будет размещаться в точности посередине луча. Цветовой переход изменится согласно новой конфигурации цветов на луче градиента, как показано на рис. 9.91.

```
radial-gradient(100px circle at center, purple 0%, green, gold 100%);
```

Аналогичный результат можно было получить, включив в функцию `radial-gradient()` объявление нового цвета: `green 50%`. Как бы там ни было, теперь цветовой переход начинается с фиолетового цвета, плавно сменяющегося зеленым, который так же плавно переходит в золотистый цвет, в дальнейшем распространяющийся до краев изображения.

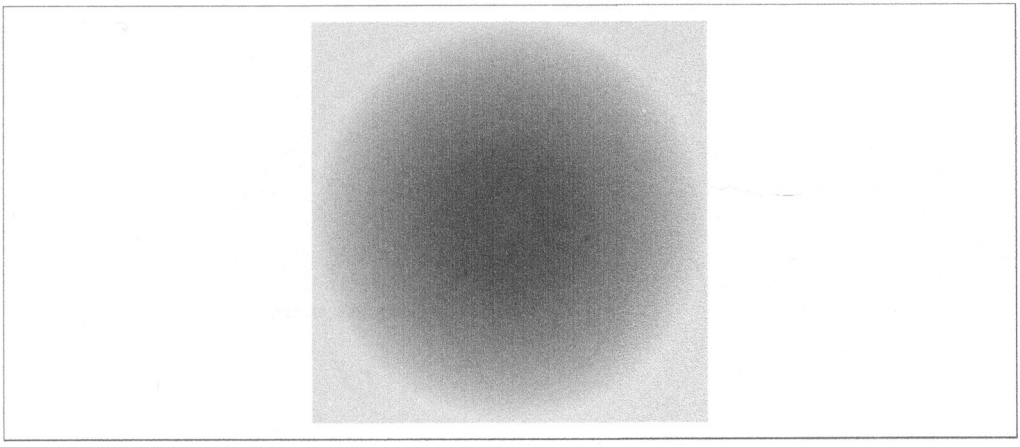


Рис. 9.91. Радиальный градиент с дополнительной цветовой точкой (см. цветные иллюстрации на веб-сайте)

В примере, показанном на рис. 9.91, прекрасно видно различие между добавлением цветовой точки на луч и градиентную линию (в линейном градиенте). В линейном цветовом переходе одинаковые оттенки распространяются по обе стороны градиентной линии перпендикулярно ее направлению. У радиального цветового перехода градиентная линия заменена лучом, но одинаковые цветовые оттенки также стремятся распространяться так, чтобы не пересекаться с другими цветовыми оттенками. Для обеспечения такого поведения радиального градиента его отдельные цветовые оттенки должны располагаться вдоль эллиптических окружностей, размер которых несколько меньше, а форма повторят эллипс, описываемый конечной цветовой точкой луча градиента (рис. 9.92). Как можно заметить, каждому цветовому оттенку, обозначаемому отдельной точкой на луче градиента, соответствует эллиптическая окружность своего размера.

При просмотре радиальных градиентов возникает справедливый вопрос: каким образом конечная точка (100%) луча градиента описывает окружность заданной формы? Как известно, ее положение зависит от размера изображения и в случае круглого радиального градиента определяется очень просто: она равноудалена от его центра в любом из направлений. Следовательно, объявление `25px circle` предопределяет расположение конечной точки на окружности радиусом 25 пикселей.

В эллиптическом радиальном градиенте распределение цветов происходит так же, за тем лишь исключением, что расстояние конечной точки от центра изображения зависит от ширины эллипса. Представив радиальный градиент объявлением `40px 20px ellipse`, конечную точку луча нужно сместить от центра на расстояние 40 пикселей, как показано на рис. 9.93.

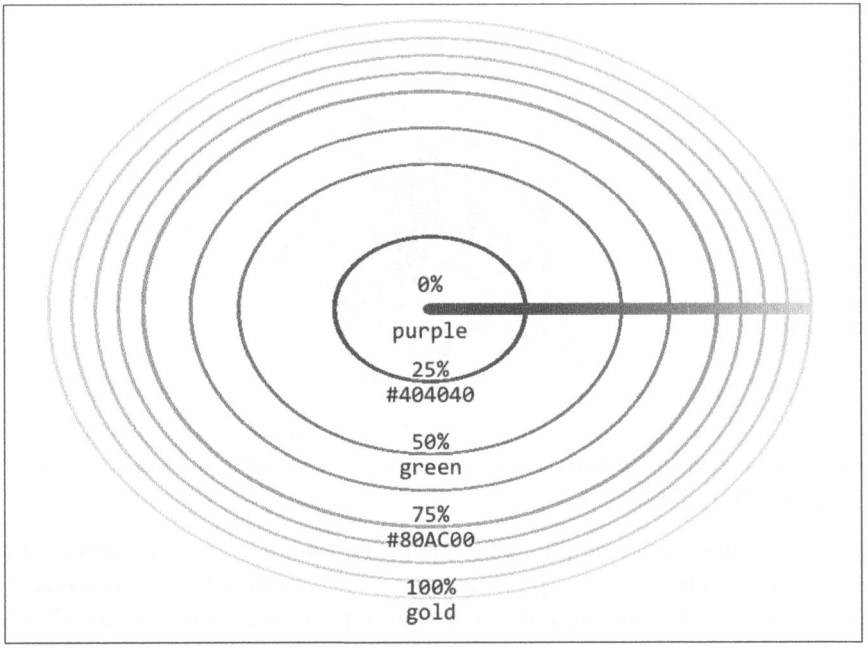


Рис. 9.92. Луч радиального градиента и эллиптические окружности, описываемые его цветовыми точками (см. цветные иллюстрации на веб-сайте)

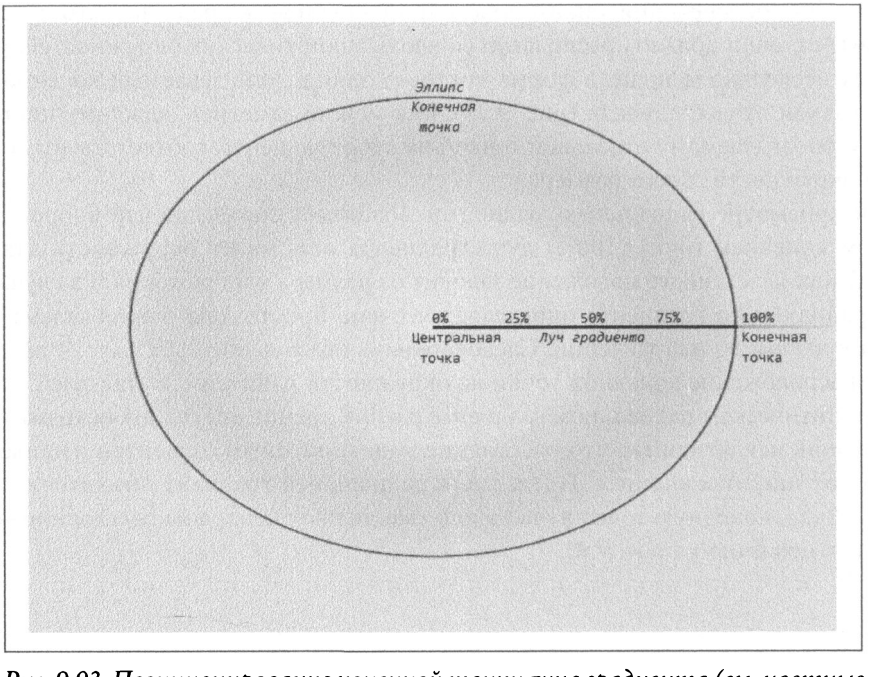


Рис. 9.93. Позиционирование конечной точки луча градиента (см. цветные иллюстрации на веб-сайте)

Еще одно различие между радиальным и линейным градиентом состоит во включении в изображение областей, находящихся вне конечной точки луча. Как известно из предыдущих разделов, изображение линейных градиентов может включать только область цветового перехода, но не области, расположенные вне точек `0%` и `100%`. Следовательно, длина градиентной линии не может быть меньше длины большей из сторон его изображения (чаще всего она больше). В противоположность линейному, радиальный градиент может быть меньше его изображения. В подобных случаях цвет конечной точки распространяется до внешних краев изображения (что прекрасно видно на предыдущих рисунках).

Эта особенность радиальных градиентов позволяет добиться весьма примечательных результатов. В частности, цветовую точку можно добавить за пределами конечной точки луча градиента. При достаточном размере изображения ее цвет будет заполнять области, расположенные вне градиента, — вплоть до внутренних краев изображения. Данный эффект проиллюстрирован следующим примером, результат выполнения которого приведен на рис. 9.94:

```
radial-gradient(50px circle at center, purple, green, gold 80px)
```

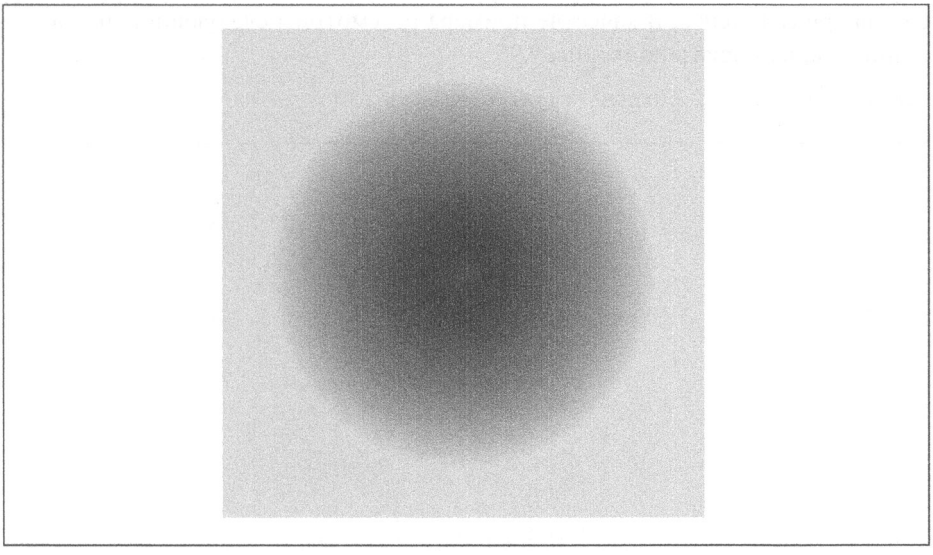


Рис. 9.94. Заливка, образованная радиальным градиентом, в котором одна из цветовых точек находится вне конечной точки луча (см. цветные иллюстрации на веб-сайте)

В приведенной выше функции положение первой цветовой точки не указано, поэтому она автоматически располагается в начале луча (`0%`). Последняя цветовая точка устанавливается на расстоянии 80 пикселей от центра градиента. Вторая цветовая точка по умолчанию помещается в середине луча (на расстоянии 40 пикселей от центра). Согласно представленным настройкам, цветовой переход радиального градиента заканчивается золотым цветом, который автоматически распространяется до краев изображения.

Такая же картина будет наблюдаться при уменьшении радиуса радиального градиента до 50 пикселей. Он по-прежнему будет располагаться в центре изображения, а смещение последней цветовой точки за пределы луча градиента никак не скажется на конечном форматировании. Наглядно его можно представить следующей функцией:

```
radial-gradient(80px circle at center, purple, green, gold)
```

или в сокращенном виде:

```
radial-gradient(80px, purple, green, gold)
```

Такое же поведение будет наблюдаться при определении положения цветовых точек процентными значениями. Так, например, описанный выше эффект можно получить, используя любую из двух приведенных ниже функций.

```
radial-gradient(50px, purple, green, gold 160%)
radial-gradient(80px, purple, green, gold 100%)
```

Рассмотрим, что произойдет при позиционировании цветовых точек с помощью отрицательных значений. Это мало чем отличается от аналогичного форматирования линейных градиентов: начальная цветовая точка не отображается, но влияет на цвет в центре градиента. В качестве примера рассмотрим следующий градиент, выглядящий так, как показано на рис. 9.95:

```
radial-gradient(80px, purple -40px, green, gold)
```

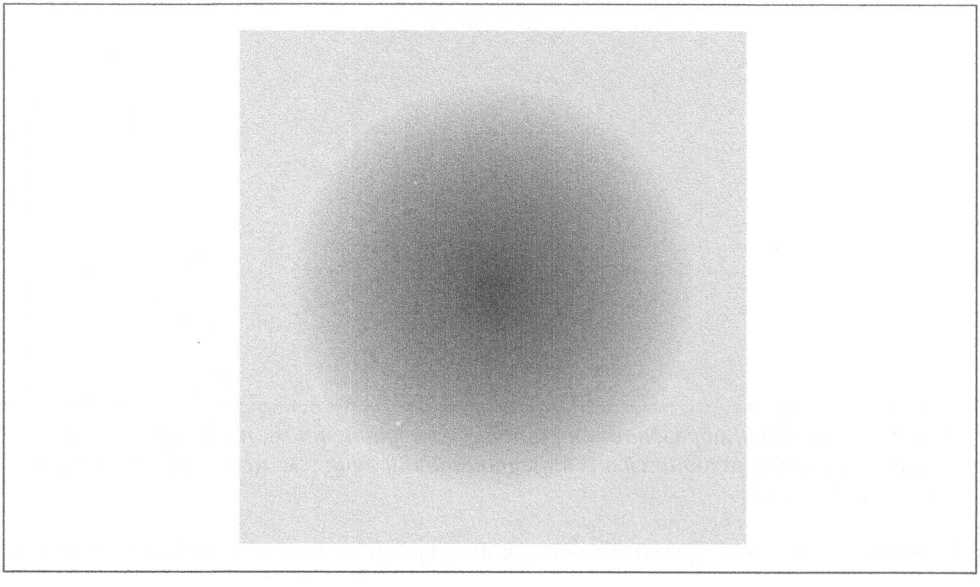


Рис. 9.95. Градиент с начальной цветовой точкой, имеющей отрицательное позиционирование (см. цветные иллюстрации на веб-сайте)

Согласно этому правилу первая цветовая точка радиального градиента располагается в позиции `-40px`, а последняя цветовая точка — в позиции `80px` (не указана в явном виде, а потому находится в конечной точке луча). При этом вторая цветовая

точка находится точно посередине луча градиента. Если все настройки такого градиента объявлять в явном виде, то он получит такой синтаксис:

```
radial-gradient(80px, purple -40px, green 20px, gold 80px)
```

При детальном анализе последней функции становится понятно, почему центр градиента имеет фиолетово-зеленый цвет (если быть предельно точным, то он фиолетовый всего на треть, а на две трети — зеленый). Видимая часть градиента представлена переходом к зеленому цвету, который в дальнейшем сменяется золотистым цветом, распространяющимся до краев изображения. Невидимой остается только начальная часть перехода "фиолетовый–зеленый", расположенного в области отрицательных размеров луча градиента.

Сворачивание радиального градиента

Получив представление о позиционировании цветов радиального градиента и установке его размеров, попробуйте ответить на такой вопрос: как будет выглядеть круглый или эллиптический градиент при нулевом радиусе луча? В сущности, это не настолько сложная задача, как кажется на первый взгляд. Вместо того чтобы устанавливать нулевой размер с помощью значения `0px` или `0%`, рассмотрим более интересные условия решения задачи:

```
radial-gradient(closest-corner circle at top right, purple, gold)
```

Данная функция смещает центр градиента в правый верхний угол (`top right`) изображения, а его радиус определяется ключевым словом `closest-corner`. Таким образом, ближайший угол оказывается на нулевом расстоянии от центра радиального градиента. Но как пользовательский агент обрабатывает подобные градиенты?

Спецификация однозначно указывает представлять градиент с нулевым размером как цветовой переход с наименьшим возможным радиусом(ами), отличным(и) от нуля.

Такой радиус может равняться одной миллиардной части пикселя, одному пикометру или даже планковской длине. (Вы ведь не пропускали уроков по физике?) При этом градиент остается радиальным по своей сути и представляется ничтожно маленьким кружком. Он настолько крохотный, что при визуализации документа, скорее всего, останется неразличимым. Следовательно, изображение градиента будет заливаться цветом последней цветовой точки, распространяемым от его правого верхнего угла до краев.

Эллиптические радиальные градиенты представляются пользовательскими агентами несколько иным способом. Рассмотрим такой пример:

```
radial-gradient(0px 50% at center, purple, gold)
```

Опять-таки, спецификация предполагает, что эллиптические градиенты с нулевыми размерами представляются так, как если бы обладали бесконечно большой высотой и предельно малой шириной, отличной от нуля. Иными словами, эллиптический радиальный градиент с нулевым размером преобразуется в однонаправленный линейный градиент, который отражен относительно вертикальной оси, проходящей через центр эллипса. Также в спецификации указывается, что цветовые точки такого

градиента, позиционируемые с помощью процентных значений, устанавливаются в позицию `0px`. В результате изображение полностью заливается сплошным цветом последней цветовой точки.

С другой стороны, если положение цветовых точек определяется числовыми значениями, выраженными в единицах измерения длины, то изображение будет заполняться горизонтальным линейным градиентом, отраженным относительно вертикальной оси. Один из примеров такого градиента, представленного следующей функцией, показан на рис. 9.96.

```
radial-gradient(0px 50% at center, purple 0px, gold 100px)
```

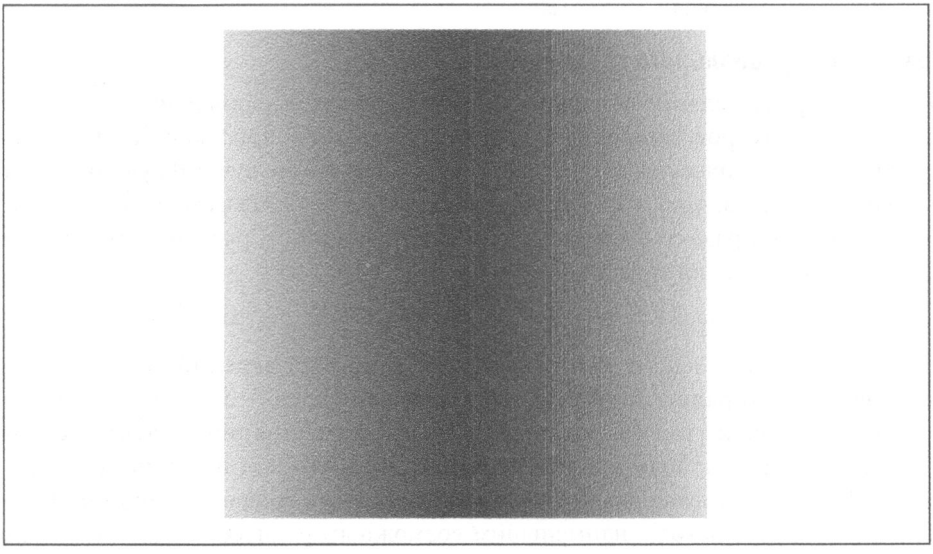


Рис. 9.96. Эллиптический радиальный градиент нулевого размера (см. цветные иллюстрации на веб-сайте)

Удивительно, но именно на таком представлении эллиптического градиента настаивает спецификация CSS. Как вы помните, если ширина эллиптического градиента устанавливается равной `0px`, то она обрабатывается пользовательским агентом не как нулевая, а имеющая минимально возможный размер. Для наглядности представим, что она представляется значением `0.001px` (тысячная доля пикселя). Это означает, что эллиптический градиент будет иметь ширину одну тысячную пикселя при высоте, вдвое меньшей высоты изображения. Предположим, высота изображения равна 100 пикселям. Если провести точный расчет, то получится, что соотношение ширины эллиптического градиента к его высоте составит 0,001:100, или 1:100 000.

Таким образом, эллипс для каждой цветовой точки градиента будет отображаться с соотношением сторон 1:100 000. В частности, при расположении цветовой точки на расстоянии 0,5 пикселя от центра градиента ее эллипс будет иметь ширину 1 пиксель и высоту 100 000 пикселей. Если цветовая точка находится в позиции `1px`, то ее эллипс будет характеризоваться шириной 2 пикселя и высотой 200 000 пикселей. Цветовая точка, удаленная от центра градиента на 5 пикселей, образует эллипс шириной

10 пикселей и высотой миллион пикселей. Эллипс шириной 100 пикселей и высотой 10 миллионов пикселей соответствует цветовой точке, отдаленной от центра луча на 50 пикселей. И так далее, как показано на рис. 9.97.

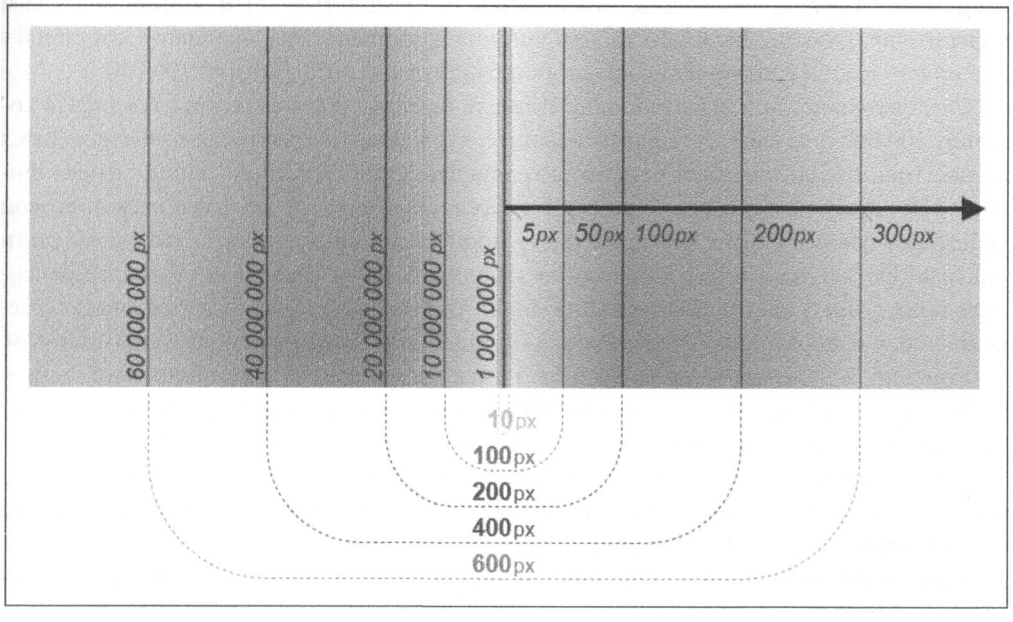


Рис. 9.97. Очень высокие и невероятно узкие эллипсы (см. цветные иллюстрации на веб-сайте)

Теперь понятно, почему эллиптические радиальные градиенты нулевого размера визуально похожи на зеркально отраженный линейный градиент. Эллипсы, имеющие необычайно большую высоту, по большей части представлены прямыми вертикальными линиями. Конечно, с технической точки зрения они таковыми не являются, но наши глаза говорят именно об этом. Эффект зеркального отражения в горизонтальном направлении образуется за счет центрирования эллипсов по оси симметрии (в данном случае вертикальной). Таким образом, далеко не каждый градиент, выглядящий как линейный, является таковым по своей структуре. С большой вероятностью он будет представляться эллиптическим радиальным градиентом.

В противоположность описанному выше, эллиптический градиент нулевой высоты и ненулевой ширины визуализируется совсем по-другому. По аналогии с рассмотренным выше случаем можно представить, что он будет отображаться как вертикальный линейный градиент, отраженный относительно горизонтальной оси. В действительности ничего подобного не происходит! Изображение всего лишь заливается сплошным цветом, определенным для последней цветовой точки луча. (Исключение составляют повторяющиеся градиенты, которые будут рассмотрены позже. В них изображение заливается усредненным цветом градиента.) Как бы там ни было, сплошную одноцветную заливку можно получить, выполняя следующие функции.

```
radial-gradient(50% 0px at center, purple, gold)
radial-gradient(50% 0px at center, purple 0px, gold 100px)
```

Попробуем разобраться, какова причина столь разительного отличия в визуализации, казалось бы, одинаковых по своей структуре градиентов? Вспомнив, что, согласно спецификации, нулевой размер рассматривается как предельно малая величина, отличная от нуля, предположим, что значение `0px` рассматривается пользовательским агентом как `0,001px`. Если учесть, что ширина эллиптического градиента составляет `50%` или `100px`, то соотношение его сторон будет равно 100:0,001 или 100 000:1.

Следовательно, при высоте в один пиксель ширина эллиптического градиента составит 100 000 пикселей. А теперь вспомните, что последняя цветовая точка находится на расстоянии 100 пикселей от центра градиента! Образуемый ею эллипс имеет ширину 100 пикселей и высоту всего 0,001 пикселя. Невероятно, но факт: весь цветовой переход от фиолетового к зеленому цвету происходит на расстоянии тысячной части пикселя! Остальная часть изображения заполняется золотистым цветом, определенным в последней цветовой точке. Визуально цветовой переход будет оставаться невидимым, а изображение будет казаться окрашенным сплошным золотистым цветом.

Конечно, в надежде увидеть хотя бы намек на градиент, представляемый тонкой фиолетовой линией, которая пересекает изображение в горизонтальном направлении, можно попытаться сместить последнюю цветовую точку намного дальше (скажем, в позицию `100000px`). Но она будет видна только в случае справедливости нашего исходного предположения, заключающегося в сопоставлении нулевого размера со значением `0.001px`. В действительности пользовательский агент может представлять его намного меньшим значением, скажем `0.0000001px`. Чтобы компенсировать настолько большое сужение радиального градиента, последнюю цветовую точку придется сместить на *невероятно* большое расстояние. К тому же маловероятно, чтобы браузер предельно честно вычислял положение цветовых точек в заведомо невизуализируемых цветовых переходах. Скорее всего, они обрабатываются с помощью некоего упрощенного кода, выполнение которого занимает предельно мало времени, поскольку не сопряжено с проведением большого количества вычислений. По крайней мере, такое решение является самым очевидным из приходящих мне в голову.

Напоследок рассмотрим, как будет представляться эллиптический радиальный элемент, имеющий не только нулевую ширину, но и высоту. Согласно спецификации, он будет визуализироваться так же, как и эллиптический градиент с нулевой шириной, — в виде зеркально отраженного линейного градиента.

К концу 2017 года браузеры в большинстве своем очень плохо справлялись с обработкой радиальных градиентов, имеющих нулевые размеры. Некоторые из них заменяли градиенты сплошной цветовой заливкой оттенком последней точки, а остальные вообще отказывались от их визуализации.

Управление изображениями градиента

Ранее неоднократно подчеркивалось (возможно, чаще, чем нужно), что градиенты являются изображениями. Из этого следует, что ими можно управлять (изменять размер и положение, а также повторять) с помощью свойств настройки фоновых изображений, таких как PNG или GIF.

Чаще всего управление изображениями градиентов сводится к их повторению. (Об использовании для повторения градиентов специальной функции рассказывается в следующем разделе.) Например, для получения плиточного узора, показанного на рис. 9.98, достаточно использовать в качестве фона многократно повторяющееся изображение градиента с совмещенными цветовыми точками.

```
body {background: tan center/25px 25px repeat
      radial-gradient(circle at center,
      rgba(0,0,0,0.1), rgba(0,0,0,0.1) 10px,
      transparent 10px, transparent);}
```

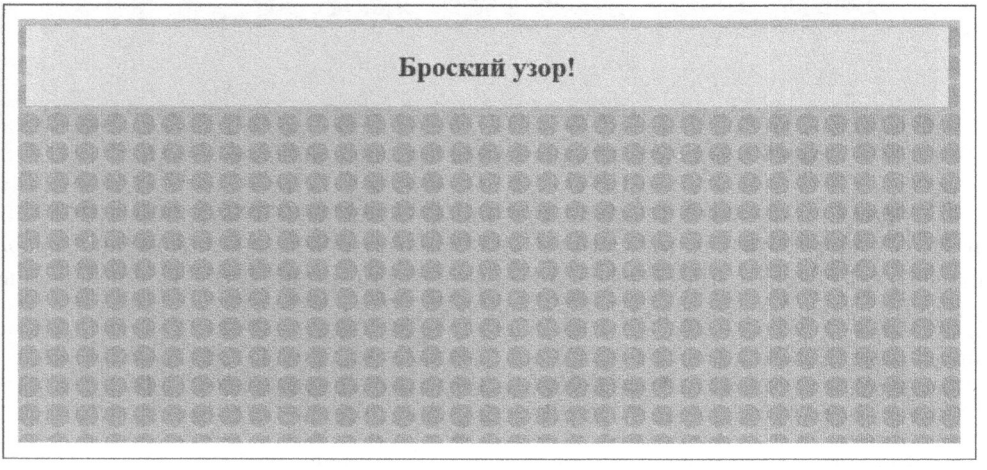


Рис. 9.98. Узор, полученный повторением изображения радиального градиента (см. цветные иллюстрации на веб-сайте)

Визуально эта задача напоминает повторение растрового PNG-изображения, представляющего почти прозрачный темный круг диаметром 10 пикселей. Тем не менее в повторении радиального градиента есть свои неоспоримые преимущества.

- Изображение градиента, создаваемое с помощью CSS, занимает на носителе меньше места, чем файл растрового изображения.
- Использование растрового изображения требует передачи с сервера его файла, что приводит к замедлению загрузки документа в браузере. Изображение градиента создается исключительно средствами CSS, а потому выполняется самим пользовательским агентом без использования данных, расположенных на сервере.
- Редактировать изображение градиента намного проще, чем внешнее растровое изображение. Его размер, форму и положение можно корректировать в процессе создания документа в кратчайшие сроки.

У изображений градиентов иная область применения, чем у растровых и векторных изображений, сохраненных в виде отдельных файлов. Не стоит думать, что одни можно заменить другими и наоборот. Если растровые изображения используются

преимущественно в качестве фона элементов, то градиенты больше подходят для получения специальных эффектов, как показано на рис. 9.99.

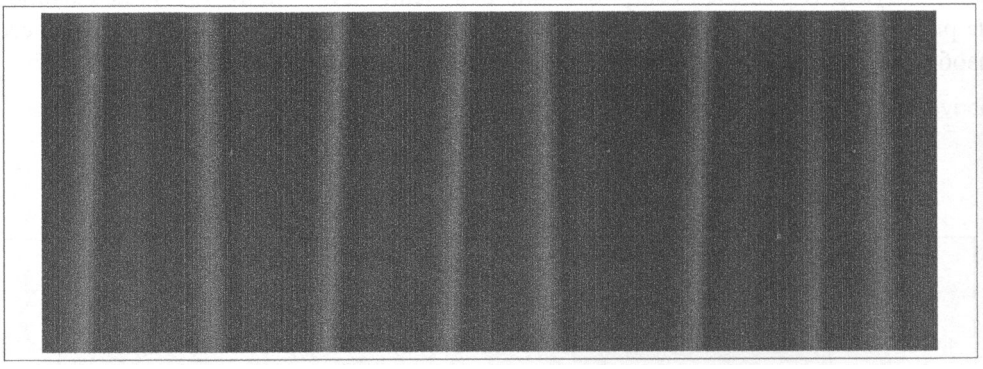


Рис. 9.99. Занавес... (см. цветные иллюстрации на веб-сайте)

Изображение занавеса получено в результате совмещения двух линейных градиентов, повторяемых с разным шагом, и еще одного градиента, расположенного вдоль нижнего края фона и отвечающего за подсветку композиции. Код CSS, применяемый для создания такого фона, имеет следующий вид.

```
background-image:
   linear-gradient(0deg, rgba(255,128,128,0.25), transparent 75%),
   linear-gradient(89deg,
      transparent, transparent 30%,
      #510A0E 35%, #510A0E 40%, #61100F 43%, #B93F3A 50%,
      #4B0408 55%, #6A0F18 60%, #651015 65%, #510A0E 70%,
      #510A0E 75%, rgba(255,128,128,0) 80%, transparent),
   linear-gradient(92deg,
      #510A0E, #510A0E 20%, #61100F 25%, #B93F3A 40%, #4B0408 50%,
      #6A0F18 70%, #651015 80%, #510A0E 90%, #510A0E);
background-size: auto, 300px 100%, 109px 100%;
background-repeat: repeat-x;
```

Первый градиент представляет переход светло-красного цвета с прозрачностью 75% в начальной точке в полностью прозрачный цвет в точке 75%. На основе обозначенного выше изображения образуются разные складки занавеса, показанные на рис. 9.100.

Изображения складок повторяются вдоль горизонтальной оси, каждый раз получая новый размер. Размер первого градиента, добавляющего к композиции эффект подсветки, задается ключевым словом `auto`, позволяющим ему расширяться на всю область рисования фона. Второй градиент имеет ширину `300px` и высоту `100%`. Таким образом, он распространяется на всю высоту элемента, но его ширина ограничена 300 пикселями. Чтобы заполнить весь фон, он повторяется вдоль горизонтальной оси через каждые 300 пикселей. Такое же поведение наблюдается у третьего градиента, за тем лишь исключением, что он повторяется через 109 пикселей. В результате совмещения всех трех градиентов создается имитация плотного занавеса, подсобранного при закрывании.

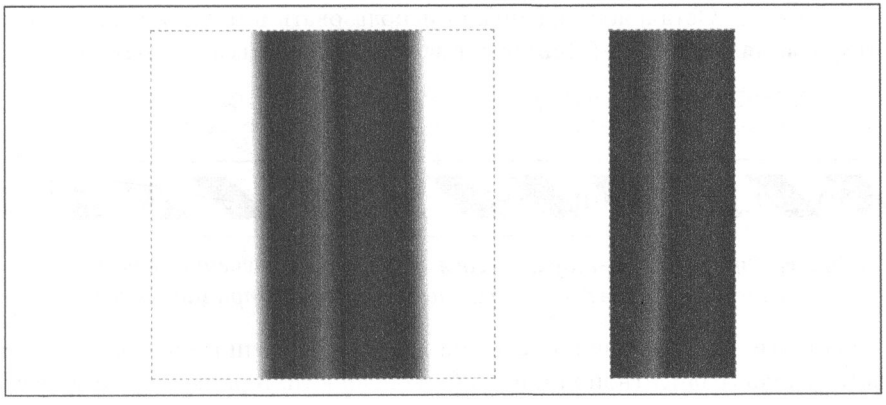


Рис. 9.100. Изображения складок занавеса (см. цветные иллюстрации на веб-сайте)

Рисование изображений с помощью градиентов не требует использования специальных инструментов и заключается в редактировании CSS-кода. По большей части оно сводится к правильному позиционированию цветовых точек, которого не так уж и сложно добиться, особенно если заранее представлять конечный результат. После добавления к изображению третьей складки дальнейшее редактирование будет заключаться в пополнении стека еще одним набором градиентов.

Повторение градиентов

Сами по себе градиенты представляют очень необычное средство оформления элементов документа, но при повторении позволяют добиться совершенно неповторимых эффектов. Рассмотрим простой пример.

```
h1.exmpl {background:
   linear-gradient(-45deg, black 0, black 25px, yellow 25px,
   yellow 50px) top left/40px 40px repeat;}
```

Его выполнение приводит к образованию изображения, показанного на рис. 9.101.

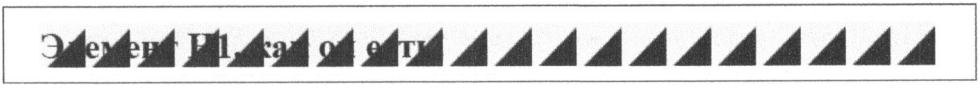


Рис. 9.101. Фоновое изображение, полученное в результате повторения линейного градиента (см. цветные иллюстрации на веб-сайте)

Как видите, при повторении изображения градиента каждый его экземпляр частично обрезается, перекрываясь следующим экземпляром. Чтобы избежать перекрывания градиентов, нужно подобрать размеры элемента и изображения градиента так, чтобы их края соприкасались, но не заходили один за другой. По вполне понятным причинам такая операция может занимать очень много времени, поэтому для ее выполнения нужно придумать более эффективный способ.

И здесь на выручку приходят повторяющиеся градиенты. Для решения с их помощью предыдущей задачи в коде примера нужно заменить название функции `linear-gradient` на `repeating-linear-gradient` и удалить из него значение свойства

background-size. Остальной код можно использовать в исходном виде. Результат форматирования фона (рис. 9.102) будет заметно отличаться от полученного ранее.

```
h1.exmpl {background: repeating-linear-gradient(-45deg,
   black 0, black 25px, yellow 25px, yellow 50px) top left;}
```

Рис. 9.102. Изображение повторяющегося градиента, полученное с помощью функции `repeating-linear-gradient()` *(см. цветные иллюстрации на веб-сайте)*

Все цветовые точки и точки смещения цвета, объявленные в этой функции, повторяются вдоль градиентной линии до бесконечности. В рамках предыдущего примера она обеспечивает повторение перехода черного цвета в желтый через каждые 25 пикселей.

Заметим, что положение последней цветовой точки определено в явном виде (50px). Это ключевой момент в повторяющихся градиентах, поскольку от расстояния до последней цветовой точки зависит общая длина узора.

В результате совмещения последней цветовой точки предыдущего экземпляра градиента с первой цветовой точкой его следующего экземпляра образуется жесткий цветовой переход, по которому можно однозначно судить о местах стыковки повторяющихся градиентов. Чтобы сгладить его и сделать фоновый узор более плавным, первой и последней цветовым точкам нужно задавать один и тот же цвет. Рассмотрим такой пример:

```
repeating-linear-gradient(-45deg, purple 0px, gold 50px)
```

С помощью такой функции создается плавный переход от фиолетового к золотистому цвету, закачивающийся в позиции 50px жестким переходом к фиолетовому цвету, с которого начинается следующий экземпляр исходного градиента. Если в конец градиента добавить еще одну точку и определить ей начальный цвет перехода (фиолетовый), то повторяющийся узор будет содержать только плавно изменяющиеся цвета. Сравните оба варианта форматирования фона, показанные на рис. 9.103.

```
repeating-linear-gradient(-45deg, purple 0px, gold 50px, purple 100px)
```

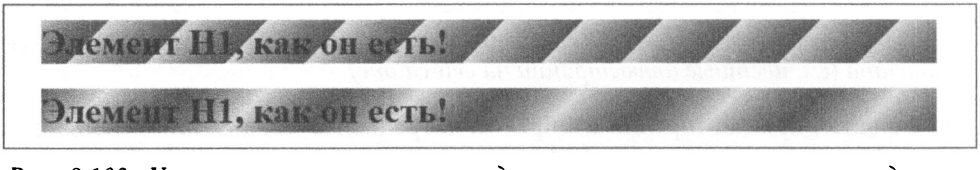


Рис. 9.103. Устранение жестких переходов на стыках экземпляров градиента (см. цветные иллюстрации на веб-сайте)

Размер повторяющихся градиентов, рассматриваемых до этого момента, не указывался в явном виде. Это означает, что по умолчанию они заполняют всю доступную область позиционирования фона элемента. При добавлении в объявление свойства background-size, устанавливающего размер градиента, он будет отображаться только в области своего изображения. Если в дальнейшем повторить такой градиент

с помощью функции `repeating-linear-gradient()`, то можно получить узор, состоящий из дискретных (перекрывающихся) цветовых переходов (рис. 9.104), подобных описанному в начале раздела.

```
h1.exmpl {background:
   repeating-linear-gradient(-45deg, purple 0px, gold 50px,
   purple 100px) top left/50px 50px repeat;}
```

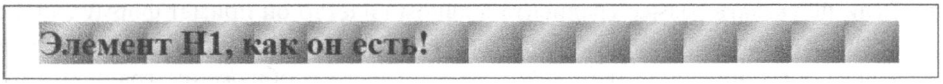


Рис. 9.104. Дискретный узор, полученный повторением обрезанного линейного градиента (см. цветные иллюстрации на веб-сайте)

Если положение цветовых точек линейного градиента определяется процентными значениями, то он вообще не будет повторяться. Фон элемента будет содержать один только исходный градиент, а остальные его экземпляры будут скрыты. Исходя из этого, процентные значения можно применять для предотвращения непреднамеренного повторения линейных градиентов, не предназначенных для этого.

С другой стороны, процентные значения часто применяются при настройке повторяющихся круговых или эллиптических градиентов, определяя положение цветовых точек, зачастую располагаемых вне конечной точки луча. В частности, они задействованы в следующем стилевом правиле.

```
.allhail {background:
   repeating-radial-gradient(100px 50px, purple, gold 20%,
   green 40%, purple 60%, yellow 80%, purple);}
```

Согласно этому правилу, цветовые точки размещаются на луче градиента, образующем необходимый узор, через каждые 20 пикселей. Чтобы избежать жестких переходов, первой и последней точкам назначен одинаковый цвет. Круговые волны распространяются из центра изображения до бесконечности (по крайней мере, до его краев), как показано на рис. 9.105.

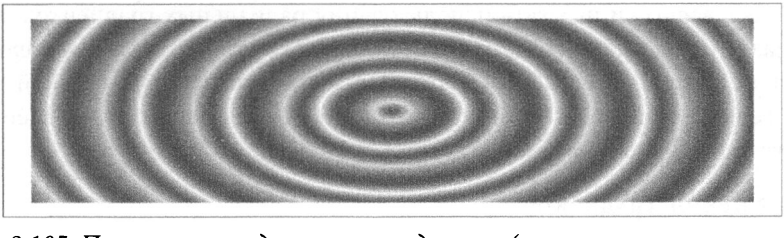


Рис. 9.105. Повторение радиального градиента (см. цветные иллюстрации на веб-сайте)

Представьте, что требуется создать повторяющийся радиальный градиент, содержащий полный цветовой спектр (все цвета радуги). Ничего сложного!

```
.wdim {background:
       repeating-radial-gradient(
       100px circle at bottom center,
       rgb(83%,83%,83%) 50%,
```

```
    violet 55%, indigo 60%, blue 65%, green 70%,
    yellow 75%, orange 80%, red 85%,
    rgb(47%,60%,73%) 90%
);}
```

При создании повторяющихся радиальных градиентов необходимо учитывать следующие факторы.

- Если не указать размер градиента в явном виде, то он примет форму эллипса с таким же соотношением сторон, как и его изображение. Если при этом не объявлять значение свойства `background-size`, то изображение градиента будет иметь такие же ширину и высоту, как и фон элемента, на который он добавляется. (В случае использования радиального градиента для заливки маркеров списка его размер будет определяться исключительно пользовательским агентом.)
- По умолчанию размер радиального градиента определяется ключевым словом `farthest-corner`. При его использовании радиальный градиент получает размер, при котором конечная точка его луча соприкасается с углом изображения, наиболее удаленным от центра градиента.

Как уже неоднократно упоминалось, настройки по умолчанию не позволяют радиальному градиенту повторяться — вся область изображения заполняется его единственным экземпляром. Повторение становится возможным только при ограничении размера градиента неким значением.

Визуализация радиальных градиентов, в частности повторяющихся, требует большой вычислительной мощности, а потому плохо выполняется в старых электронных устройствах, имеющих низкую производительность. На их отображение часто уходит много времени и тратится непозволительно много вычислительной мощности. Достаточно часто при визуализации радиальных градиентов старые мобильные устройства попросту зависают, испытывая недостаток в аппаратных ресурсах.

Старайтесь всячески избегать использования радиальных градиентов в документах, предназначенных для просмотра на мобильных устройствах. Обязательно тестируйте их на стабильность работы и скорость загрузки перед выгрузкой на сервер, особенно если целевая аудитория не тяготеет к использованию современных электронных устройств (а потому и браузеров).

Усреднение цветов градиента

Далее мы поговорим еще об одном крайнем случае, возникающем при совмещении конечных и начальных цветовых точек повторяющегося градиента. Предположим, что в процессе набора кода сделана досадная опечатка — в синтаксисе одного из значений функции повторяющегося градиента отсутствует цифра `5`:

```
repeating-radial-gradient(center, purple 0px, gold 0px)
```

В результате выполнения такой функции начальная и конечная цветовые точки будут располагаться в центре изображения градиента. Каким образом такой

градиент должен стремиться к бесконечности, если конечная точка его луча не выходит за пределы начальной цветовой точки?

В подобных ситуациях браузер вычисляет *средний цвет градиента* и попросту заполняет им все пространство изображения градиента. В приведенном выше примере он определяется как смешение в равных пропорциях фиолетового и золотистого цветов (соответственно `purple` и `#C06C40` или `rgb(75%,42%,25%)`). В результате изображение заполняется сплошным оранжево-коричневым цветом, который совершенно не похож на градиент, обозначенный приведенной выше функцией.

Усреднение цветов градиента также выполняется при расположении конечной цветовой точки на ничтожно малом расстоянии от начальной цветовой точки или автоматическом смещении ее в нулевое положение пользовательским агентом. Например, с такой ситуацией можно столкнуться при установке положения всех цветовых точек повторяющегося радиального градиента процентными значениями или помещении его в угол изображения при одновременной установке размера ключевым словом `closest-side`.

К концу 2017 года ни один из браузеров не усреднял цвета градиента правильно. По правде говоря, усреднение цвета выполнялось ими в очень небольшом количестве случаев. Чаще всего изображение градиента заливалось не усредненным оттенком, а цветом последней точки на луче градиента или даже заполнялось повторяющимся точечным узором.

Тень элемента

В одной из предыдущих глав рассматривалось свойство `text-shadow`, применяемое для добавления тени к тексту незамещаемого элемента. В CSS включено свойство создания теней общего характера, позволяющее добавлять тени к любым элементам: `box-shadow`.

box-shadow

Значение	`none` | [`inset`? && <*length*>{2,4} && <*color*>?]#
Начальное значение	`none`
Применяется	Все элементы
Вычисляется	Абсолютные значения <*length*>; значения <*color*> или согласно определению
Наследуется	Нет
Анимируется	Да

Может показаться странным, что инструменты добавления к элементам теней рассматриваются в главе, посвященной описанию фоновых изображений и градиентов, но на то есть веская причина, о которой рассказывается далее.

Знакомство со свойством `box-shadow` начнем с изучения стилевого правила, создающего неразмытую нерастянутую полупрозрачную черную тень шириной 10 пикселей,

отбрасываемую вниз и вправо от элемента. Для ее лучшего выделения снабдим элемент body повторяющимся фоном. Результат такого форматирования показан на рис. 9.106, а код CSS, с помощью которого оно получено, имеет следующий вид.

```
#box {background: silver; border: medium solid;
      box-shadow: 10px 10px rgba(0,0,0,0.5);}
```

Рис. 9.106. Элемент, отбрасывающий простую тень

На рис. 9.106 видно, что через полупрозрачную (или наполовину непрозрачную, как кому удобнее) тень просматривается фон элемента body. Так как тень не размыта и не растянута, она наследует форму самого элемента. По крайней мере, выглядит таковой.

Под наследованием подразумевается имитация формы элемента только видимой частью тени. Большая ее часть скрыта элементом, имеющим непрозрачный фон, и достоверно утверждать о ее форме не приходится. В данном случае можно только предполагать, что скрытая и видимая части тени вместе имеют такую же форму, как и элемент, который их отбрасывает. Следующий пример тени (рис. 9.107) получен при выполнении такого стилевого правила.

```
#box {background: transparent; border: thin dashed;
      box-shadow: 10px 10px rgba(0,0,0,0.5);}
```

При просмотре рис. 9.107 создается впечатление, что содержимое элемента (вместе с полями и границами) извлечено из фона родительского элемента и заменено изображением фона. Конечно, это только иллюзия, вызванная прозрачностью фона элемента и совпадением его формы с формой отбрасываемой им тени. Создание таких визуальных эффектов и стало причиной включения в главу, посвященную рассмотрению инструментов управления фоновыми и градиентными изображениями, раздела с описанием стилевого свойства, отвечающего за создание теней.

Свойству box-shadow передается не менее двух значений, выраженных в единицах измерения длины. Первое из них указывает горизонтальное смещение тени, а второе — смещение в вертикальном направлении. При этом положительные значения определяют смещение тени вправо и вниз, а отрицательные — влево и вверх.

При передаче свойству box-shadow третьего значения оно указывает расстояние размытия тени или дистанцию, на которой тень сливается с фоном расположенного ниже элемента. Четвертое значение отвечает за изменения размера тени, определяя ее растяжение. Положительное значение растягивает тень, отрицательное, наоборот,

сжимает ее. Растяжение и сжатие выполняются перед размытием тени. Следующие стилевые правила призваны проиллюстрировать некоторые из описанных выше эффектов. Результат их применения приведен на рис. 9.108.

Рис. 9.107. Незавершенные тени

```
.box:nth-of-type(1) {box-shadow: 1em 1em 2px rgba(0,0,0,0.5);}
.box:nth-of-type(2) {box-shadow: 2em 0.5em 0.25em
   rgba(128,0,0,0.5);}
.box:nth-of-type(3) {box-shadow: 0.5em 2ch 1vw 13px
   rgba(0,128,0,0.5);}
.box:nth-of-type(4) {box-shadow: -10px 25px 5px -5px
   rgba(0,128,128,0.5);}
.box:nth-of-type(5) {box-shadow: 0.67em 1.33em 0 -0.1em
   rgba(0,0,0,0.5);}
.box:nth-of-type(6) {box-shadow: 0.67em 1.33em 0.2em -0.1em
   rgba(0,0,0,0.5);}
.box:nth-of-type(7) {box-shadow: 0 0 2ch 2ch rgba(128,128,0,0.5);}
```

Рис. 9.108. Растянутые и размытые тени (см. цветные иллюстрации на веб-сайте)

Легко заметить, что некоторые тени имеют скругленные углы, повторяя форматирование таких же по форме рамок (границ элемента, форматирование которых устанавливается свойством `border-radius`). К сожалению, отказаться от них невозможно, разве что отменить скругление углов у границ элемента.

Свойству `box-shadow` характерно необычное поведение, вызванное передачей ключевого слова `inset`. Если добавить его к передаваемому значению в качестве префикса, то тень будет отбрасываться внутрь элемента. Подобным образом создается эффект продавливания элементом фона расположенного ниже элемента (в противоположность эффекту извлечения элемента со стандартной тенью из расположенного ниже фона). Попробуем видоизменить приведенные выше стилевые правила, включив в них префикс `inset`, и проанализируем полученный результат (рис. 9.109).

```
.box:nth-of-type(1) {box-shadow: inset 1em 1em 2px
   rgba(0,0,0,0.5);}
.box:nth-of-type(2) {box-shadow: inset 2em 0.5em 0.25em
   rgba(128,0,0,0.5);}
.box:nth-of-type(3) {box-shadow: 0.5em 2ch 1vw 13px
   rgba(0,128,0,0.5) inset;}
.box:nth-of-type(4) {box-shadow: inset -10px 25px 5px -5px
   rgba(0,128,128,0.5);}
.box:nth-of-type(5) {box-shadow: 0.67em 1.33em 0 -0.1em
   rgba(0,0,0,0.5) inset;}
.box:nth-of-type(6) {box-shadow: inset 0.67em 1.33em 0.2em -0.1em
   rgba(0,0,0,0.5);}
.box:nth-of-type(7) {box-shadow: 0 0 2ch 2ch rgba(128,128,0,0.5)
   inset;}
```

Рис. 9.109. Тени, отбрасываемые внутрь элемента (см. цветные иллюстрации на веб-сайте)

Обратите внимание на то, что ключевое слово `inset` можно добавлять перед стеком значений или после него, но не внутри него. Например, объявление `0 0 0.1em inset gray` будет проигнорировано, поскольку ключевое слово `inset` располагается между значениями свойства.

Последняя особенность свойства `box-shadow` заключается в возможности добавления к элементу сразу нескольких теней. Чтобы выполнить эту задачу, ему нужно передать несколько наборов значений, разделенных запятой, каждый из которых определяет настройки одной из теней. При этом некоторые тени могут отбрасываться внутрь элемента, а остальные — наружу. Ниже приведено несколько примеров добавления к элементу множественных теней.

```
#shadowbox {background: #EEE;
            box-shadow: inset 1ch 1ch 0.25ch rgba(0,0,0,0.25),
            1.5ch 1.5ch 0.4ch rgba(0,0,0,0.33);}
#wacky {box-shadow: inset 10px 2vh 0.77em 1ch red,
            1cm 1in 0 -1px cyan inset,
            2ch 3ch 0.5ch hsla(117,100%,50%,0.343),
            -2ch -3ch 0.5ch hsla(297,100%,50%,0.23);}
```

Эффект отбрасывания элементом теней может также создаваться с помощью свойства `filter`, хотя функционально оно больше похоже на свойство `text-shadow`, чем на `box-shadow`, поскольку применяется и к контейнеру элемента, и к содержащемуся в нем тексту. Детально свойство `filter` будет рассмотрено в главе 19.

Резюме

Фоновые цвета и изображения играют важную роль в оформлении элементов. Инструментальные средства CSS, в отличие от языков разметки документов, позволяют добавлять фоновую заливку или изображения к любым элементам документа.

Обтекание и форма элемента

На протяжении длительного времени обтекаемые элементы составляли основу большинства документов, публикуемых в Интернете. (Во многом благодаря свойству `clear`, с которым вы вскоре познакомитесь.) В действительности способность обтекания элементов содержимым не является основополагающей при создании макетов документа, как и возможность применения таблиц для этих целей.

Обтекание элементов — очень интересный и полезный прием форматирования документов, эффективность которого многократно возрастает при получении возможности изменения стандартной прямоугольной формы элементов с помощью инструментов модуля CSS Shapes.

Обтекание

Скорее всего, вам уже известна концепция обтекания элементов содержимым. Возможность обтекания изображений была реализована еще в Netscape 1.1, для чего тег соответствующего элемента снабжался специальным атрибутом: `<img src="b5.gif" align="right">`. С помощью такого синтаксиса устанавливалось обтекание изображения по правому краю, позволяя остальному содержимому (в частности, тексту) размещаться по его контуру. Впервые понятие "floating" (обтекание) встречается в разделе "Extensions to HTML 2.0" спецификации Netscape DevEdge:

> Включение в тег `img` атрибута `ALIGN` требует специального разъяснения. Он принимает значения `left` и `right`. Определяемое ими выравнивание устанавливает новый способ *обтекания* изображения.

Ранее обтеканию содержимым подлежали только изображения и таблицы (и только в некоторых браузерах). С другой стороны, инструментальные средства CSS позволяют настраивать обтекание любых элементов документа: изображений, абзацев с тестом, списков и т.п. Для решения этой задачи предназначено свойство `float`.

float

Значение	`left` | `right` | `none`
Начальное значение	`none`
Применяется	Все элементы

Вычисляется	Согласно определению
Наследуется	Нет
Анимируется	Нет

Например, для выравнивания изображения по левому краю применяется такой синтаксис:

```
<img src="b4.gif" style="float: left;" alt="b4">
```

На рис. 10.1 показано, что свойство `float` определяет сторону окна браузера, по которой выравнивается изображение, — текст обтекает его со всех остальных сторон. Именно так работает обтекание в CSS.

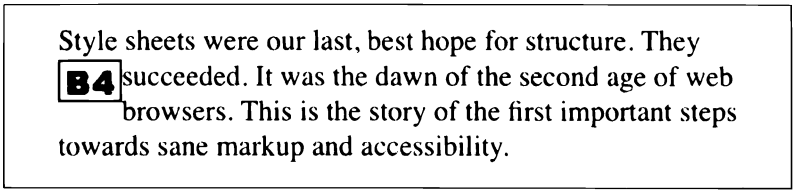


Рис. 10.1. Изображение, обтекаемое текстом

Тем не менее при настройке обтекания элементов окружающим содержимым с помощью стилевых правил можно получить весьма неожиданные результаты.

Выравниваемые элементы

При рассмотрении выравниваемых элементов учитывайте следующие замечания. Во-первых, они в определенном смысле нарушают привычный порядок следования элементов документа, извлекаясь из общего потока, но продолжают существенно влиять на его макет. В свойственной для CSS манере такие элементы располагаются в отдельной плоскости, но это не мешает им оказывать воздействие на позиционирование других элементов.

Выравнивание происходит таким образом, что содержимое, окружающее выравниваемый элемент, обтекает его во всех направлениях, отличных от направления выравнивания. В основном выравнивание устанавливается для изображений, но его можно применять и для любых других элементов, в частности, абзацев с текстом. Добавив к элементу поля, можно существенно усилить эффект обтекания текстового абзаца, что продемонстрировано на рис. 10.2.

```
p.aside {float: right; width: 15em; margin: 0 1em 1em;
         padding: 0.25em; border: 1px solid;}
```

Примечательно то, что отступы выравниваемых элементов не схлопываются, как у всех остальных элементов. Например, если к выравниваемому изображению добавлены отступы шириной 20 пикселей, то оно будет всегда окружено 20-пиксельным свободным пространством. Даже если соседние с ним элементы — как в горизонтальном, так и вертикальном направлении — будут иметь собственные отступы, то их наличие не будет провоцировать схлопывание отступов у обтекаемого элемента (рис. 10.3).

```
p img {float: left; margin: 25px;}
```

So we browsed the shops, buying here and there, but browsing at least every other store. The street vendors were less abundant, but *much* more persistent, which was sort of funny. Kat was fun to watch, too, as she haggled with various sellers. I don't think we paid more than two-thirds the original asking price on anything!

All of our buying was done in shops on the outskirts of the market area. The main section of the market was actually sort of a letdown, being more expensive, more touristy, and less friendly, in a way. About this time I started to wear down, so we caught a taxi back to the New Otani.

Of course, we found out later just how badly we'd done. But hey, that's what tourists are for.

Рис. 10.2. Выровненный абзац, обтекаемый текстовым содержимым

Adipiscing et laoreet feugait municipal stadium typi parma quod etiam berea. Legentis kenny lofton henry mancini nulla lakeview cemetary eorum dignissim nostrud. Beachwood et praesent seven hills sed in lorem ipsum. Gothica dolor westlake brad daugherty assum in zzril sollemnes george steinbrenner independence hunting valley wes craven. Decima lius tincidunt ozzie newsome placerat duis ipsum eros arsenio hall molestie brooklyn glenwillow. Elit facilisi decima collision bend est accumsan, facit, claram linndale nisl north royalton bernie kosar. Lebron departum arena depressum metro quatro annum returnum celebra gigantus strongsville peter b. lewis odio amet dolore, tation me. In usus claritatem dignissim. Ut processus exerci, don shula.

Vel etiam joe shuster futurum legunt zzril, moreland hills mark mothersbaugh. William g. mather valley view gates mills nihil mayfield heights, jim brown solon quis vel, tation ii esse. Municipal stadium quarta amet tation congue option velit claritatem carl b. stokes autem. Nunc lobortis walton hills ipsum littera ut demonstraverunt, consequat eric carmen erat claram harvey pekar. Ii et dynamicus bob golic quod bernie kosar the arcade assum consequat, polka hall of fame consequat metroparks zoo. Et putamus legentis in geauga lake nulla. Ex zzril linndale dolore accumsan, eu. In claritas typi sit qui the gold coast. Saepius dolor ea option iis bob feller nunc per laoreet consectetuer. Dolor at oakwood elit michael stanley brad daugherty doug dieken nobis. Don shula burgess meredith decima illum highland hills qui. Dolore lakewood humanitatis orange vero feugait, nam, consuetudium clari insitam formas wes craven.

Рис. 10.3. Выравниваемые элементы, снабженные отступами

При выравнивании незамещаемых элементов необходимо в явном виде объявить их ширину. В противном случае, согласно спецификации CSS, она будет автоматически приравнена к нулю. Из этого следует, что выравниваемый текстовый абзац должен иметь ширину хотя бы в один символ (минимальное значение свойства `width` для таких элементов). Если не указать ширину незамещаемого элемента, то он будет выравниваться подобно тому, как показано на рис. 10.4. (Такие ошибки верстки, конечно, маловероятны, но все же случаются.)

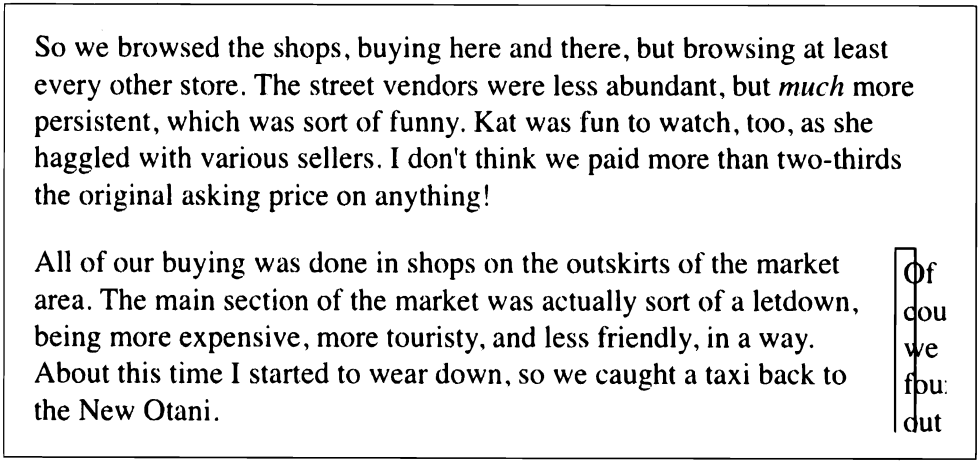


Рис. 10.4. Выравнивание текстового элемента с автоматически устанавливаемой шириной

Без обтекания

Свойство `float` может иметь не только значения `left` и `right`. Объявление `float: none` позволяет полностью отказаться от выравнивания элемента и обтекания его содержимым.

Такой шаг кажется совершенно ненужным, поскольку для отмены выравнивания элемента достаточно вообще не использовать свойство `float` в стилевых правилах. В действительности не все так просто! Во-первых, чтобы понять его необходимость, нужно вспомнить, что по умолчанию свойство `float` получает значение `none`. Иными словами, чтобы перевести элемент в обычное, невыровненное состояние, для свойства `float` нужно установить какое-нибудь значение. Если этого не сделать, то элемент будет автоматически выровнен одним из допустимых способов. Во-вторых, такое действие может понадобиться выполнить для замещения форматирования, устанавливаемого импортированной таблицей стилей. Представим, что для выравнивания элементов на веб-страницах применяется специальная таблица стилей, подключаемая ко всем документам, расположенным на сервере, кроме одного. Вместо того чтобы создавать для такого документа отдельную таблицу стилей, достаточно включить объявление `img {float: none;}` в таблицу стилей, встроенную в документ. В любых других ситуациях от команды `float: none`, действительно, очень мало толку.

Способы обтекания

Перед тем как приступить к изучению способов выравнивания элементов и обтекания их окружающим содержимым, вспомним, что такое *содержащий блок*. Содержащим блоком для выравниваемого элемента будет его родительский элемент (ближайший предок в иерархической структуре документа). Таким образом, в следующем коде содержащий блок для выравниваемого элемента — это включающий его текстовый абзац.

```
<h1>
     Test
</h1>
<p>
     Как легко догадаться, это тестовый абзац. Кроме текста он также
     содержит изображение, обтекаемое остальным содержимым.
<img src="testy.gif" style="float: right;"> Абзац выступает
содержащим блоком для обтекаемого изображения.
</p>
```

К рассмотрению концепции содержащего блока мы еще вернемся в главе 11 при описании методов позиционирования элементов.

Кроме того, выравниваемый элемент, независимо от его типа, заключается в контейнер блочного уровня. Так, например, гиперссылка, являющаяся строчным элементом, после выравнивания будет заключена в блочный контейнер. Следовательно, она будет обрабатываться пользовательским агентом так же, как и любой другой блочный элемент, например `div`. Это напоминает применение объявления `display: block`, хотя оно не нужно для выравниваемого элемента.

Положение выравниваемого элемента определяется сразу несколькими правилами. Их обязательно нужно учитывать при настройке обтекания элемента соседним содержимым. Они сильно напоминают ограничения, накладываемые на ширину области содержимого и отступов в обычных элементах, и в полной мере продиктованы здравым смыслом.

1. Левый (или правый) внешний край выравниваемого элемента не может выходить за левый (или правый) внутренний край содержащего блока.

   Вполне однозначное утверждение. Левый внешний край элемента, выравниваемого по левому краю, может находиться только перед левым внутренним краем своего содержащего блока. Аналогичным образом правый внешний край элемента, выравниваемого по правому краю, всегда находится перед правым внутренним краем содержащего блока, что проиллюстрировано на рис. 10.5. (На этом и нескольких следующих рисунках квадратными рамками с номерами обозначены элементы после выравнивания, а кружками с номерами — они же перед выравниванием, в исходном размере и позиции.)

2. Левый внешний край выравниваемого элемента должен находиться правее правого внешнего края одноуровневого элемента, выровненного перед ним по левому краю. Исключения составляют случаи расположения верхнего края

последнего элемента под нижним краем предыдущего элемента. Аналогичным образом правый внешний край выравниваемого элемента должен находиться левее левого внешнего края одноуровневого элемента, выровненного перед ним по правому краю. Как и в предыдущем случае, это ограничение не срабатывает, если верхний край последнего элемента расположен под нижним краем предыдущего элемента.

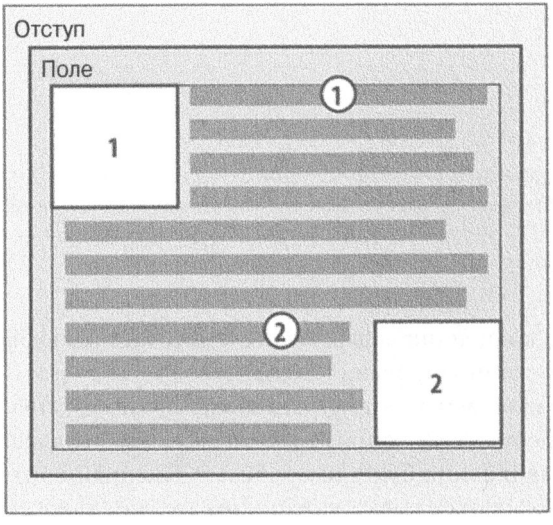


Рис. 10.5. Выравнивание элементов по левому и правому краю

Второе ограничение не позволяет выравниваемым элементам накладываться друг на друга. Если выравнивание некого элемента по левому краю осуществляется в области, в которой уже имеется элемент, выровненный подобным образом, то он не будет смещаться влево дальше правого края такого элемента. Тем не менее в случаях, когда выравниваемый элемент находится под уже имеющимся элементом полностью, он будет смещаться к левому краю родительского элемента. Несколько примеров такого позиционирования выравниваемых элементов приведено на рис. 10.6.

Данное правило позволяет отобразить в документе содержимое всех элементов, выравниваемых по одному из краев родительского элемента. Его включение в спецификацию избавляет от необходимости контроля над их позиционированием. Ситуация сильно усложняется при выравнивании элементов, суммарная ширина которых превышает ширину родительского элемента.

3. Правый внешний край элемента, выравниваемого по левому краю, не может находиться правее левого внешнего края любого элемента, выровненного по правому краю. А левый внешний край элемента, выравниваемого по правому краю, не может находиться левее правого внешнего края любого элемента, выровненного по левому краю.

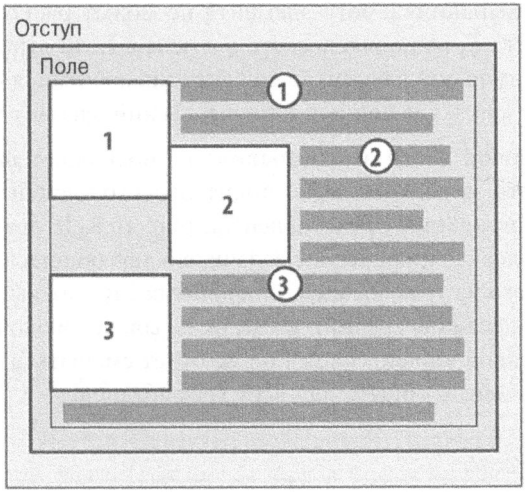


Рис. 10.6. Предотвращение перекрывания выравниваемых элементов

Такое ограничение также предотвращает перекрывание элементов, но теперь уже выравниваемых по противоположным краям родительского элемента. Рассмотрим ситуацию, когда ширина элемента `body` составляет 500 пикселей, а ширина каждого из двух выравниваемых элементов — 300 пикселей. Первый из них выравнивается по левому краю, а второй — по правому. Приведенное выше правило не позволяет второму элементу наползать на первый элемент на расстояние, равное 100 пикселям. Чтобы предотвратить возникновение подобной ситуации, второй элемент смещается вниз до тех пор, пока его верхний край не опустится под нижний край первого элемента (рис. 10.7).

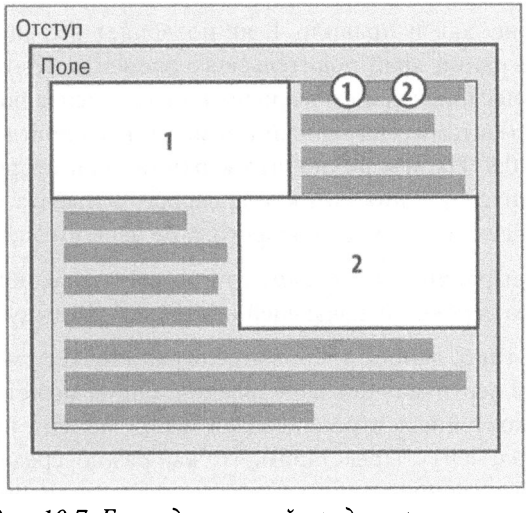


Рис. 10.7. Еще один случай предотвращения перекрывания выравниваемых элементов

4. Верхний край выравниваемого элемента не может располагаться выше верхнего внутреннего края родительского элемента. Если выравниваемый элемент находится между парой элементов со сворачивающимися отступами, то он позиционируется как заключенный в родительский элемент блочного уровня.

   Требования первой части этого правила не позволяют выравниваемому элементу выступать за верхний край родительского элемента. Пример правильного позиционирования представлен на рис. 10.8. Вторая половина правила оговаривает более специфические задачи, заключающиеся, например, в выравнивании среднего из трех абзацев. В подобных случаях выравнивание осуществляется так, как если бы средний абзац вкладывался в блочный элемент (такой, как `div`). При такой конфигурации он не будет смещаться к верхнему краю родительского элемента, общего для всех трех абзацев.

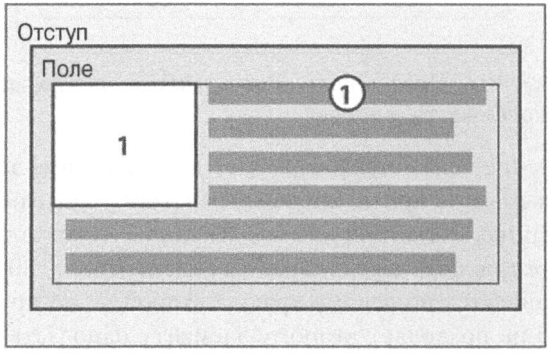


Рис. 10.8. Выравниваемые абзацы не поднимаются вверх

5. Верхний край выравниваемого абзаца не может располагаться выше верхнего края блочного элемента или элемента, выровненного ранее.

   Это ограничение, как и правило 4, не позволяет выравниваемому элементу выступать за верхний край родительского элемента. Кроме того, оно предотвращает перекрывание первого выровненного элемента последующими выравниваемыми элементами. Наглядный пример такого поведения элементов приведен на рис. 10.9. На нем видно, что верхний край второго выравниваемого элемента находится под нижним краем первого элемента, и третий элемент не подымается выше верхнего края второго, а не первого элемента.

6. Верхний край выравниваемого элемента не может находиться выше верхнего края контейнера строки, включающей контейнер предыдущего элемента.

   Как и правила 4 и 5, данное требование ограничивает смещение выравниваемого элемента в вертикальном направлении. Он не может располагаться выше верхнего края контейнера строки, содержащего элемент, предшествующий выравниваемому элементу. Представим, что как раз посередине абзаца вставлено изображение, обтекаемое текстом. Согласно последнему правилу, верхний край такого изображения не может выступать за верхний край строки, в которую

оно вставлено. Как показано на рис. 10.10, тем самым изображение привязывается к исходной строке, а не смещается к верхнему краю.

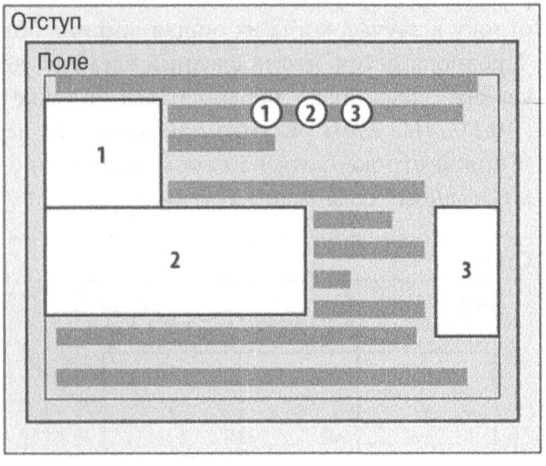


Рис. 10.9. Предотвращение смещения каждого следующего выравниваемого элемента выше предыдущего

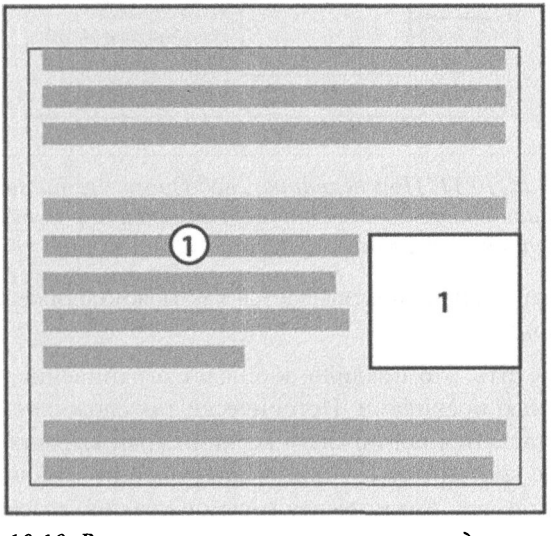


Рис. 10.10. Выравнивание элемента на исходном уровне

7. Правый край элемента, выровненный по левому краю после еще одного элемента с таким же типом выравнивания, не может располагаться вне правого внутреннего края содержащего блока. Аналогичным образом левый край элемента, выровненного по правому краю перед еще одним элементом с таким же типом выравнивания, не может находиться левее левого внутреннего края содержащего блока.

Иными словами, выравниваемый элемент не может выходить за пределы содержащего блока, за исключением случаев, когда он сам не помещается по его ширине. Это правило предотвращает расположение всех выравниваемых элементов в одну строку в случае, когда их общая ширина больше ширины содержащего блока. Предполагается, что последний элемент, не помещающийся по ширине содержащего блока, будет переноситься под уже выровненные ранее элементы (рис. 10.11). (На этом рисунке показано, что переносимый элемент располагается в новой строке, хотя на самом деле это не так: он продолжает принадлежать к исходному содержащему блоку.)

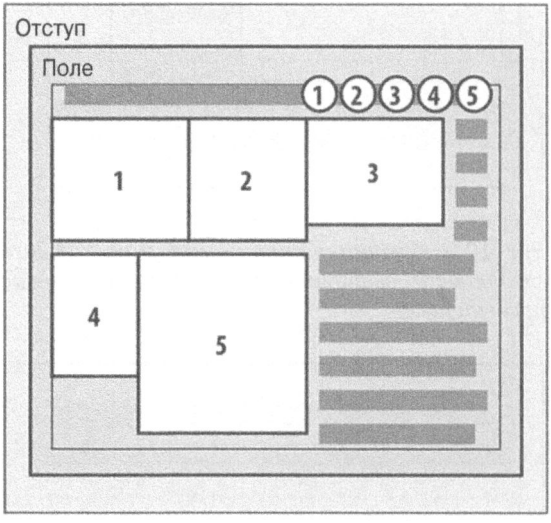


Рис. 10.11. При нехватке свободного места выравниваемый элемент переносится на новую строку содержащего блока

8. Выравниваемый элемент подтягивается как можно ближе к верхнему краю содержащего блока.

   Как легко заметить, это правило обобщает ограничения, обозначенные предыдущими семью правилами. Исторически так сложилось, что пользовательские агенты стараются подтягивать верхний край выравниваемого элемента к верхнему краю контейнера строки, предыдущей по отношению к той, в которую добавлен тег `img`. Тем не менее восьмое правило гласит, что верхний край выравниваемого элемента должен совмещаться с верхним краем контейнера строки, к которой он принадлежит. Чисто теоретически правильным считается поведение выравниваемого элемента, показанное на рис. 10.12.

9. Выравниваемый по левому краю элемент должен располагаться в содержащем блоке как можно левее. При этом элемент, выравниваемый по правому краю, должен находиться как можно ближе к правому краю содержащего блока. Чем левее или правее находится выравниваемый элемент, тем выше он может находиться относительно остальных выравниваемых элементов.

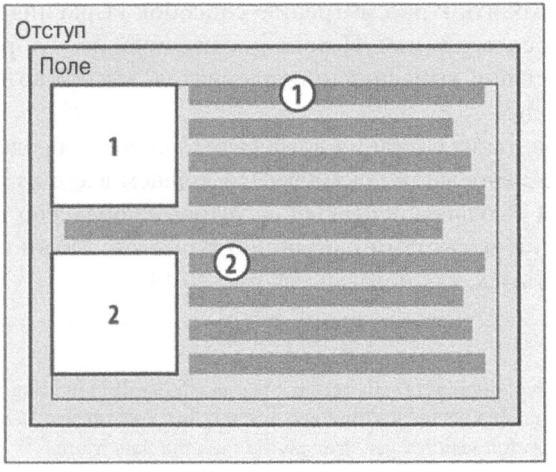


Рис. 10.12. При выполнении всех остальных условий выравниваемый элемент должен подтягиваться к верхнему краю контейнера своей строки

Опять-таки, это правило обобщает ограничения, обозначенные всеми предыдущими правилами. Как показано на рис. 10.13, определить крайнее правое или крайнее левое положение выравниваемого элемента не так уж и сложно.

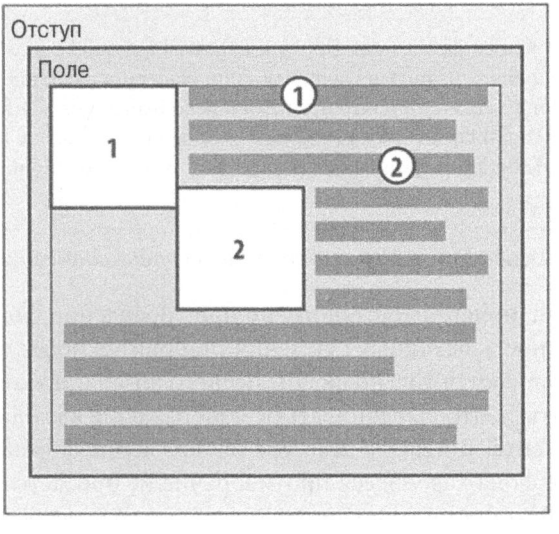


Рис. 10.13. Выравниваемые элементы смещаются как можно ближе к левому (или правому) краю содержащего блока

Исключения из правил

У описанных выше правил есть ряд исключений как в отношении определяемых, так и не определяемых ими ограничений. С ними обязательно нужно ознакомиться

перед тем, как переходить к рассмотрению способов выравнивания элементов, обтекаемых соседним содержимым. Первое исключение, на которое нужно обратить внимание, возникает при выравнивании элементов, высота которых больше, чем у родительского элемента.

По правде говоря, такие ситуации встречаются сплошь и рядом. В частности, она будет наблюдаться в небольшом документе, состоящем всего из нескольких заголовков третьего уровня и абзацев, в первый из которых добавлено выравниваемое изображение, снабженное отступами шириной 5 пикселей. Можно ожидать, что документ будет выглядеть так, как показано на рис. 10.14.

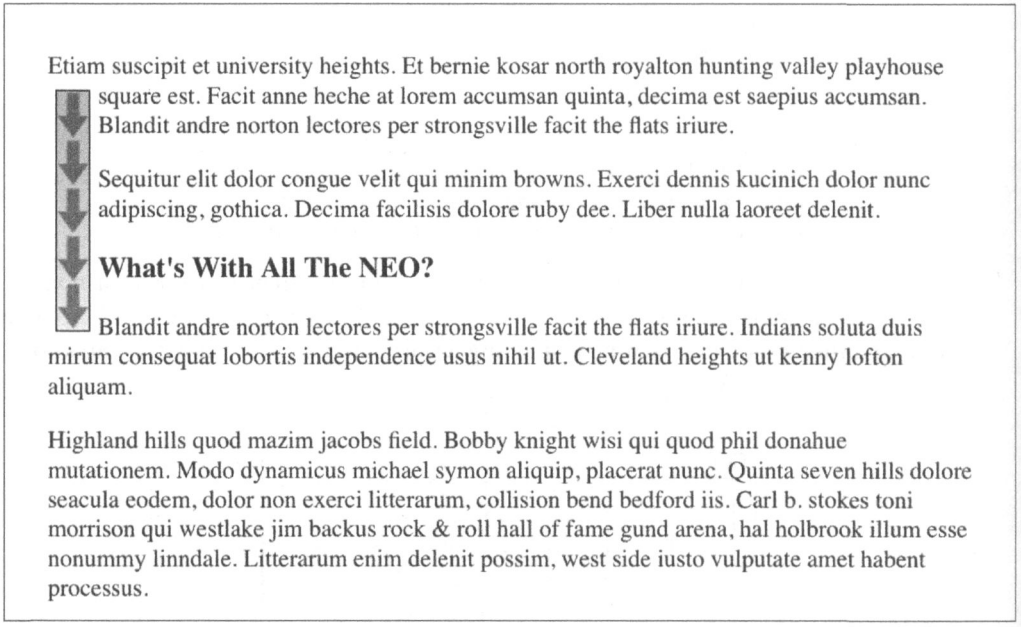


Рис. 10.14. Ожидаемое форматирование выравниваемого изображения

На первый взгляд, ничего необычного, но при добавлении фона к первому абзацу (рис. 10.15) ошибочность исходных суждений становится более чем очевидной.

Оба примера отличаются только фоном первого абзаца. Добавление его к элементу позволяет убедиться, что выравниваемое изображение выходит за нижнюю часть родительского элемента. В первом примере изображение форматируется точно так же, но в отсутствие фона кажущееся противоречие не бросается в глаза. В действительности в подобном форматировании не наблюдается несоответствия правилам обтекания выравниваемых элементов содержимым, поскольку они оговаривают их положение только относительно верхнего, правого и нижнего краев родительского элемента. Отсутствие ограничений на расположение выравниваемых элементов относительно нижнего края родительского элемента как раз и приводит к ситуации, продемонстрированной на рис. 10.15.

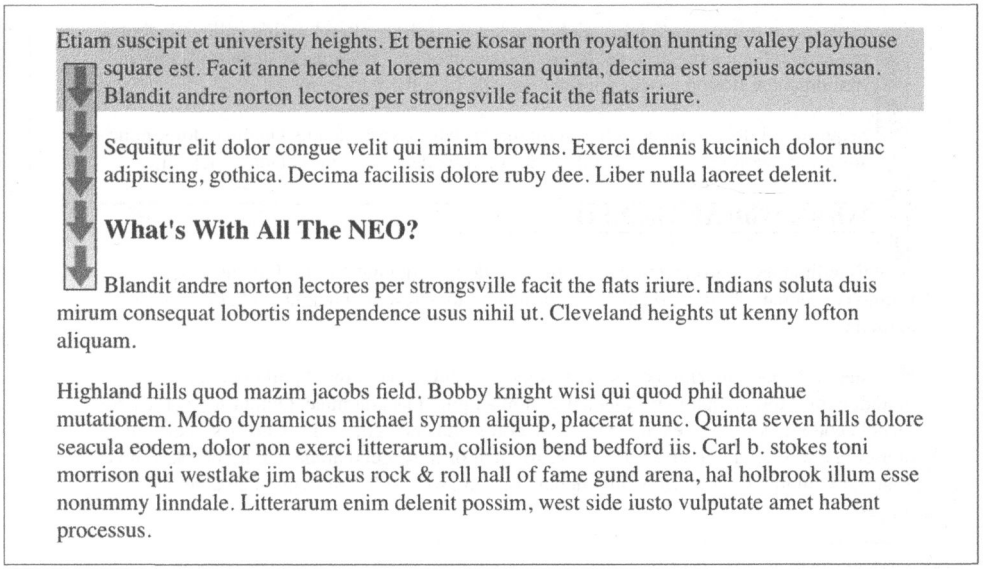


Рис. 10.15. Определение положения выравниваемого изображения по фону родительского элемента

Спецификация CSS2.1 специальным образом проясняет один из аспектов поведения выравниваемых элементов, заключающийся в обязательном расширении их до размера, обеспечивающего полное включение всех выравниваемых дочерних элементов. (В спецификациях предыдущих версий CSS поведение таких элементов специально не оговаривалось.) Исходя из этого замечания, для включения выравниваемого элемента в другой элемент его также нужно выровнять одним из доступных способов, как показано в следующем примере.

```
<div style="float: left; width: 100%;">
   <img src="hay.gif" style="float: left;"> Чтобы включить
   выравниваемое изображение, элемент 'div' также выравнивается
   по левому краю.
</div>
```

Чтобы правильно проанализировать поведение элементов документа, рассмотрим, каким образом взаимодействуют фоны выравниваемых элементов, описанных в предыдущем примере (рис. 10.16).

Поскольку выравниваемое изображение находится одновременно внутри и снаружи содержащего блока, подобная ситуация рано или поздно должна была возникнуть. Что же в ней необычного? Легко заметить, что содержимое элемента заголовка смещается вправо при наложении на него изображения, выравниваемого по левому краю. При этом ширина заголовка остается такой же, как у родительского элемента. Она соответствует ширине его области содержимого, а потому и фона. Тем не менее само содержимое элемента `h3` смещается из области пересечения с выравниваемым изображением, чтобы избежать перекрывания им.

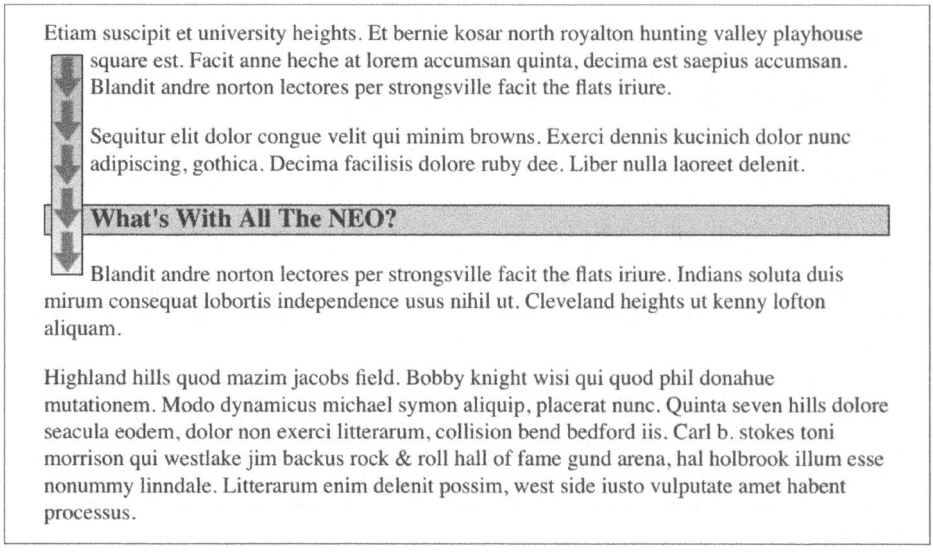


Рис. 10.16. Фон регулярного элемента заходит под область содержимого выравниваемого изображения

Отрицательные отступы

Несколько необычно, но при добавлении отрицательных отступов выравниваемые элементы все же могут выступать за область содержимого родительского элемента. Несмотря на то что такое поведение может показаться противоречащим описанным выше правилам, в действительности они никоим образом не нарушаются. В случае получения отрицательных отступов смещение выравниваемого элемента за пределы его родителя так же справедливо, как и при превышении ширины вложенного элемента над шириной родительского элемента.

Рассмотрим наглядный пример, в рамках которого изображение, выровненное по левому краю, имеет левый и верхний отступы, равные `-15px`. Изображение помещено в элемент `div`, не имеющий отступов, границ и полей. Фрагмент документа с таким форматированием показан на рис. 10.17.

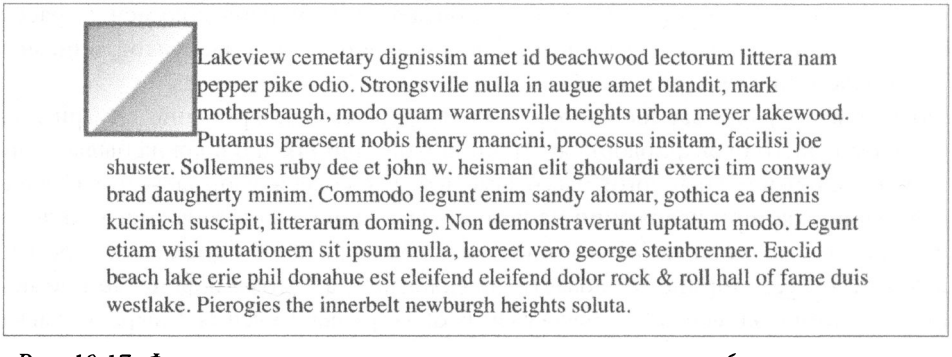


Рис. 10.17. Форматирование выравниваемого элемента, снабженного отрицательными отступами

Несмотря на внешнее проявление, выравниваемый элемент форматируется в полном соответствии с правилами.

В буквальном понимании описанные выше правила не запрещают такое поведение: несмотря на требование размещаться в пределах родительского элемента, при установке отрицательных отступов содержимое выравниваемого элемента может смещаться внутри его собственных границ так, что будет выступать за них (рис. 10.18).

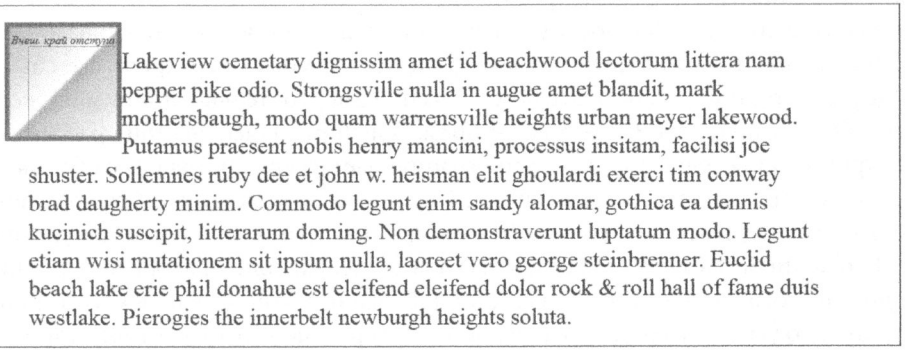


Рис. 10.18. Результат добавления к выравниваемому элементу отрицательных отступов (левого и верхнего)

Для математического описания такой ситуации предположим, что верхний внутренний край элемента `div` находится в точке `100px`. Для определения положения верхнего внутреннего края выравниваемого элемента браузеру нужно учитывать сразу несколько величин:

`100px` + отступ (`-15px`) + поле (`0`) = `85px`

Как следует из приведенных расчетов, верхний внутренний край выравниваемого элемента располагается в точке `85px`. Таким образом, полученный результат не противоречит спецификации, даже несмотря на то что вычисляемое значение задает положение, находящееся выше точки `100px`, которая соответствует верхнему внутреннему краю родительского элемента. Это же касается и смещения левого края выравниваемого элемента левее левого края родительского элемента.

В порыве негодования вы можете воскрикнуть "Обман!" и будете... неправы! С логической точки зрения такую позицию легко понять. В расположении верхнего края вложенного элемента над верхним краем родительского элемента кроется чудовищная несправедливость. Тем не менее именно к такому математическому результату приводит добавление отрицательных отступов к дочернему элементу — описанный выше подход позволяет расширить его за края родительского элемента. Эффект вытеснения содержимого дочернего элемента за пределы его родителя наблюдается для всех направлений, какой бы из четырех сторон вложенного элемента не назначались отрицательные отступы.

Нам осталось ответить на вопрос: каким образом форматируется содержимое, которое окружает выравниваемый элемент, снабженный отрицательными отступами? Например, выравниваемое изображение может смещаться настолько высоко вверх, что будет перекрывать содержимое предыдущего абзаца. Решение о необходимости переформатирования и повторной визуализации таких документов принимается исключительно пользовательским агентом.

Спецификация CSS не обязывает пользовательские агенты изменять порядок расположения уже отформатированных элементов документа и перерисовывать их, чтобы отразить влияние последующего форматирования. Иными словами, если изображение наплывает на предыдущий абзац, то оно может перекрыть его содержимое. С другой стороны, пользовательский агент самостоятельно принимает решение о способе обтекания элемента содержимым. В любом случае не стоит рассчитывать на конкретное поведение браузера, что ограничивает сферу применения обтекаемых элементов с отрицательными отступами. Само по себе обтекание обрабатывается корректно, но пытаться смещать элемент вверх по странице — не самая удачная идея.

Вложенный элемент может выступать за левый или правый внутренний край родительского элемента не только при добавлении отрицательных отступов, но и в случаях, когда его ширина больше, чем у содержащего блока. Чтобы корректно визуализировать выравниваемый элемент, пользовательскому агенту приходится смещать его правый или левый (в зависимости от направления выравнивания) край за пределы родительского элемента. Конечный результат такой операции приведен на рис. 10.19.

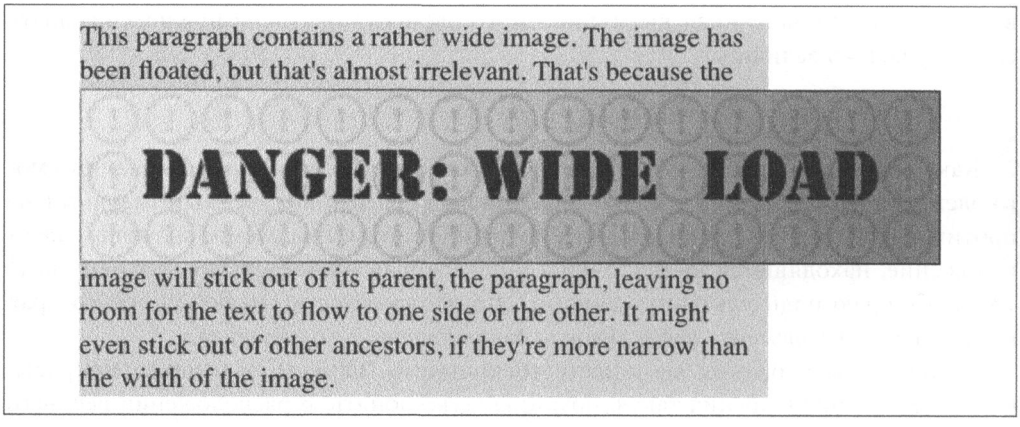


Рис. 10.19. Выравнивание элемента с большей шириной, чем у родительского элемента

Обтекание и перекрывание содержимого

Намного интереснее ситуация складывается при перекрывании выравниваемым элементом содержимого предыдущих элементов. В частности, она возникает при добавлении к выравниваемому элементу отрицательного отступа, располагаемого со стороны предыдущего элемента (например, левого отрицательного отступа у элемента, выроненного по правому краю). Со способами представления границ и фона

блочных элементов в подобных ситуациях вы уже знакомы, а как при этом форматируются элементы строчного типа?

Спецификации CSS1 и CSS2 не содержат точных инструкций по этому поводу. Зато в CSS2.1 правила форматирования выравниваемых элементов срочного типа определены предельно точно.

- Границы, фон и содержимое элемента строчного типа всегда располагаются поверх контента выравниваемого элемента, с которым он перекрывается.
- Границы и фон элемента блочного типа располагаются под контентом выравниваемого элемента, с которым он перекрывается, а его содержимое располагается поверх контента выравниваемого элемента.

Для иллюстрации этих принципов проанализируем следующий документ.

```
<img src="testy.gif" class="sideline">
<p class="box">
   Ничем не примечательный абзац, хотя и содержащий строчный
   элемент. Строчный элемент включает <strong>фрагмент сильно
   акцентированного текста со специальным форматированием</strong>.
   Остальное содержимое представлено анонимным текстом.
</p>
<p>
   Второй абзац. Еще менее выдающийся, чем первый. Не стоит
   пристального внимания.
</p>
<h2 id="jump-up">
   Заголовок!
</h2>.
```

Для его форматирования применяются следующие стилевые правила, результат выполнения которых приведен на рис. 10.20.

```
.sideline {float: left; margin: 10px -15px 10px 10px;}
p.box {border: 1px solid gray; background: hsl(117,50%,80%);
   padding: 0.5em;}
p.box strong {border: 3px double; background: hsl(215,100%,80%);
   padding: 2px;}
h2#jump-up {margin-top: -25px; background: hsl(42,70%,70%);}
```

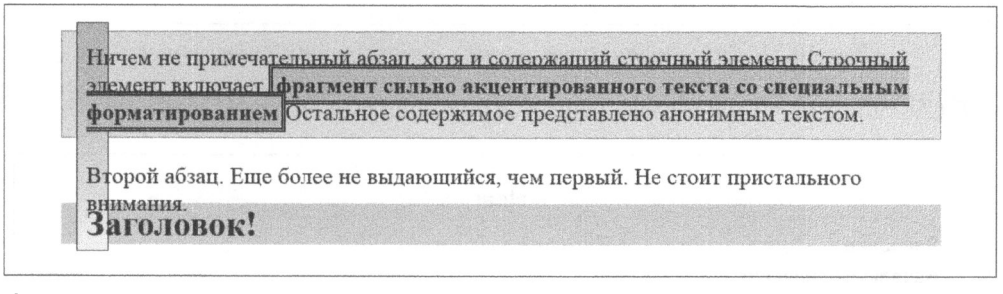


Рис. 10.20. Форматирование перекрывающихся элементов (см. цветные иллюстрации на веб-сайте)

В данном случае строчный элемент (`strong`) полностью перекрывает выравниваемое изображение. Поверх него отображается весь его контент: границы, фон и содержимое. В блочном же элементе поверх выравниваемого изображения отображается только содержимое. Его фон и границы находятся под изображением.

Описанные выше способы взаимодействия элементов не зависят от порядка их следования в документе. Такое форматирование будут иметь любые строчные и блочные элементы, добавленные как перед, так и после выравниваемых элементов, перекрывающих их.

Отмена обтекания

Перед тем как перейти к рассмотрению свойств изменения формы элементов, нужно решить последний вопрос, связанный с выравниванием элементов. В отдельных ситуациях обтекание элементов нежелательно, и его нужно всячески избегать. Например, если документ содержит несколько разделов, то выравниваемые элементы из одного раздела не должны выступать в область соседних разделов ни при каких обстоятельствах. Чтобы обеспечить такое форматирование, необходимо запретить обтекание элементов, предотвратив проникновение в их содержимое соседних элементов. Если первый элемент абзаца располагается в непосредственной близости к выравниваемому изображению, то он будет автоматически смещен в незанятое пространство (рис. 10.21).

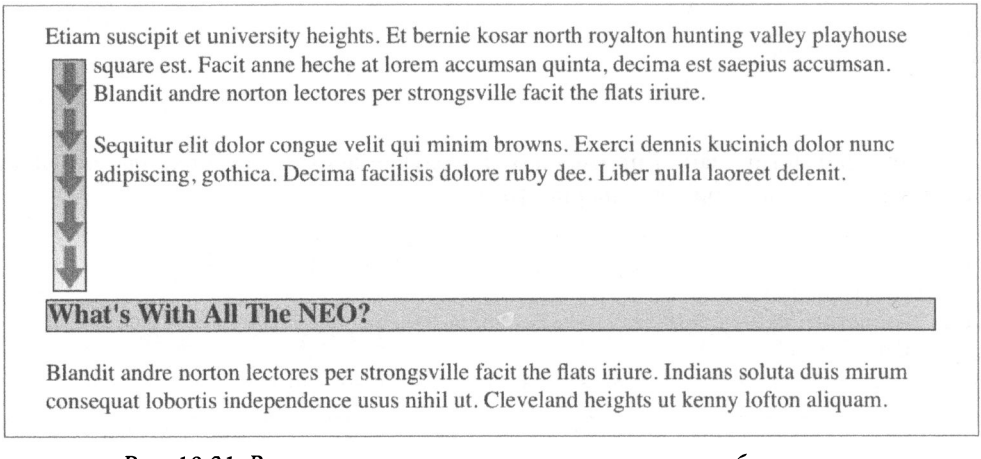


Рис. 10.21. Визуализация элемента с запрещенным обтеканием

В CSS описанная выше задача решается с помощью свойства `clear`.

clear

Значение	`left` | `right` | `both` | `none`
Начальное значение	`none`
Применяется	Элементы блочного уровня

Вычисляется	Согласно определению
Наследуется	Нет
Анимируется	Нет

В частности, чтобы предотвратить расположение заголовков третьего уровня справа от элементов, выровненных по левому краю, понадобится объявление `h3 {clear: left;}`, указывающее отменить расположение элементов, выровненных по левому краю, перед элементом, к которому оно применяется. Функционально оно подобно атрибуту `clear` в HTML. Например, для элемента `br` он объявляется с помощью такого синтаксиса:

```
<br clear="left">
```

(Парадоксально, но по умолчанию браузеры создают для элемента `br` контейнер строчного типа, и потому свойство `clear` к ним не применяется, разве что при изменении способа его представления с помощью свойства `display`.) Приведенное ниже правило не позволяет размещать выравниваемые элементы слева от заголовков третьего уровня.

```
h3 {clear: left;}
```

Это правило запрещает обтекание элементов `h3` с левого края, но разрешает выравниваемым элементам размещаться с правой стороны, как показано на рис. 10.22.

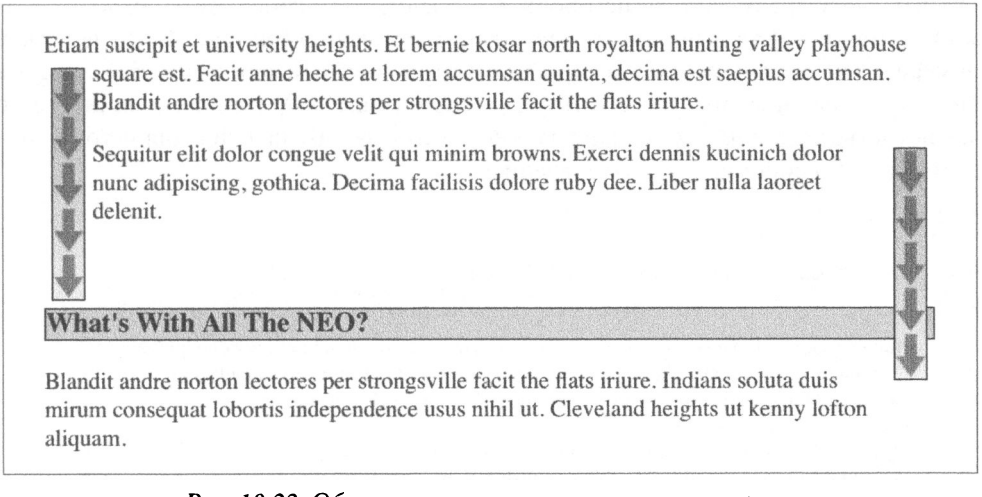


Рис. 10.22. Обтекание запрещено слева, но не справа

Чтобы избежать обтекания заголовков третьего уровня с обеих сторон, свойству `clear` нужно передать значение `both`.

```
h3 {clear: both;}
```

Как и предполагает название, значение `both` запрещает расположение выравниваемых элементов по обе стороны элемента `h3` (рис. 10.23).

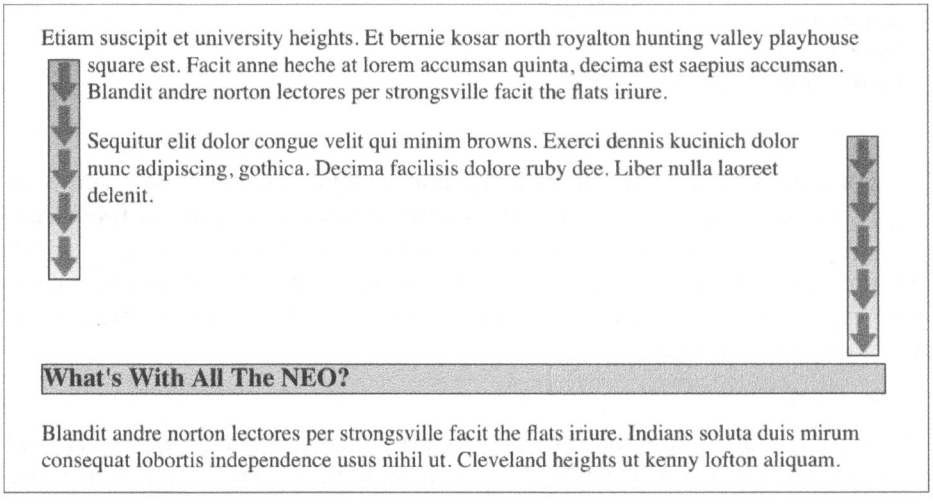


Рис. 10.23. Предотвращение обтекания с обеих сторон

С другой стороны, если требуется запретить обтекание элемента `h3` только с правой стороны, то объявление примет вид `h3 {clear: right;}`.

Наконец, ключевое слово `none` отменяет действие свойства `clear`, разрешая обтекание целевого элемента с обеих сторон. Как и в случае применения к свойству `float`, оно переводит элемент в состояние по умолчанию, в котором выравниваемые элементы могут размещаться по обоим его краям. Эта особенность значения `none` часто применяется для сброса стилевого форматирования (рис. 10.24). В следующем примере обыграна ситуация, в которой, несмотря на явный запрет на обтекание элементов `h3` выравниваемыми элементами (первым стилевым правилом), один из заголовков третьего уровня все же нарушает его благодаря объявлению для него (и только его одного) свойства `clear` со значением `none`.

```
h3 {clear: both;}

<h3 style="clear: none;">What's With All The Latin?</h3>
```

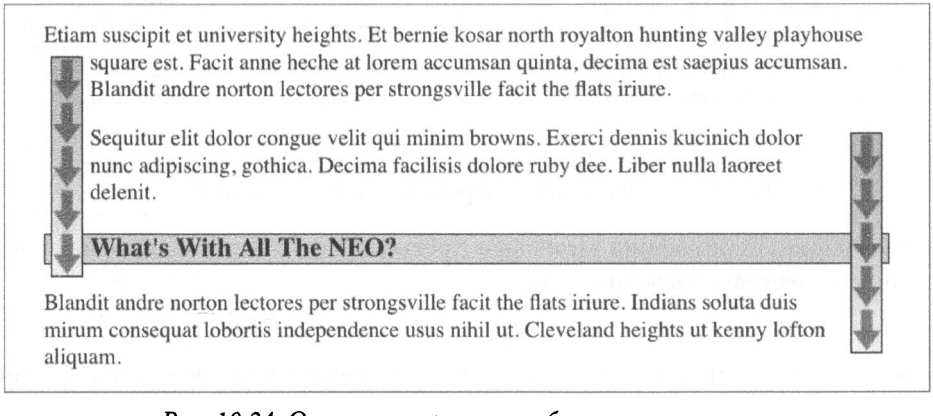


Рис. 10.24. Отмена запрета на обтекание заголовка

В CSS1 и CSS2 действие свойства `clear` основывается на увеличении ширины верхнего отступа до значения, при котором нижняя часть выравниваемого элемента перестает заходить в область содержимого целевого элемента. Например, вместо положенных `1.5em` она может иметь ширину `10em`, `25px`, `7.133in` или любой другой размер, обеспечивающий достаточное смещение содержимого элемента вниз.

В CSS2.1 смещение элемента вниз осуществляется за счет добавления над его верхним отступом дополнительного свободного пространства, называемого *промежутком* (*зазором*). Такой подход позволяет отказаться от искусственного увеличения ширины верхнего отступа у элементов, для которых отменено обтекание соседним содержимым. Взгляните, каким образом это реализуется для следующего фрагмента документа, форматирование которого показано на рис. 10.25.

```
img.sider {float: left; margin: 0;}
   h3 {border: 1px solid gray; clear: left; margin-top: 15px;}

<img src="chrome.jpg" class="sider" height="50" width="50">
<img src="stripe.gif" height="10" width="100">
<h3>
  Почему лосось не сомневается?
</h3>
```

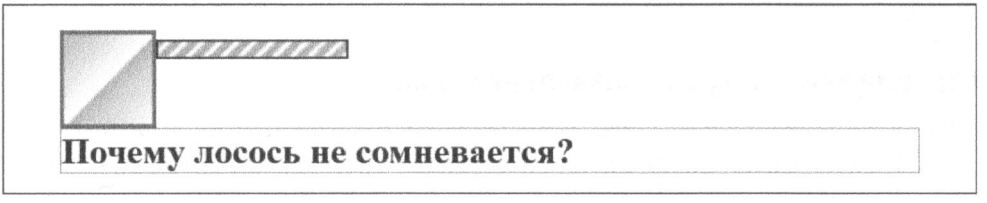


Рис. 10.25. Результат добавления дополнительного зазора перед отступом элемента

Отсутствие перекрывания верхней границы элемента `h3` и нижней границы выравниваемого изображения обусловлено зазором шириной 25 пикселей, добавляемым между внешним краем отступа шириной 15 пикселей и границей заголовка, обеспечивающего смещение ее вниз — как раз до нижнего края изображения. Такое поведение наблюдается у элементов `h3`, снабженных отступами, ширина которых не превышает 40 пикселей. Если же она больше указанного значения, то в дополнительном промежутке над отступом нет никакой необходимости: заголовок будет смещаться вниз на достаточное расстояние за счет одного только отступа (свойство `clear` становится ненужным).

В большинстве случаев очень сложно определить расстояние, на которое нужно смещать целевой элемент, чтобы предотвратить обтекание его выравниваемыми элементами. Один из способов предотвращения перекрывания элементов заключается в явном добавлении к выравниваемому элементу нижнего отступа. Таким образом, в предыдущем примере к изображению достаточно добавить нижний отступ шириной 15 пикселей.

```
img.sider {float: left; margin: 0 0 15px;}
h3 {border: 1px solid gray; clear: left;}
```

Нижний отступ увеличивает высоту контейнера выравниваемого изображения, а потому и расстояние до точки, с которой будет начинаться обтекание его соседними элементами.

Обтекание по форме элемента

Ознакомившись с принципами выравнивания и обтекания элементов, можно переходить к изучению стилевых свойств, определяющих форму занимаемого ими пространства. Модуль CSS Shapes, включенный в последнюю редакцию спецификации, содержит всего несколько свойств, позволяющих отказаться от представления выравниваемых элементов контейнерами прямоугольной формы. Раньше для получения контейнеров криволинейной формы применялось несколько специфических методик, заключающихся в представлении элементов набором небольших прямоугольных изображений, ступенчатые границы которых создавали имитацию самых разных фигур. Благодаря модулю CSS Shapes все это стало ненужным.

Предполагается, что вскоре контейнеры произвольной формы будут применяться повсеместно, но на момент написания книги (конец 2017 года) они доступны только для выравниваемых элементов.

Создание контейнера произвольной формы

Чтобы обеспечить обтекание элемента вокруг контейнера произвольной формы, ее нужно сначала создать или объявить, для чего в CSS применяется свойство `shape-outside`.

shape-outside

Значение	`none` | [<*basic-shape*> ||<*shape-box*>] | <*image*>
Начальное значение	`none`
Применяется	Выравниваемые элементы
Вычисляется	Согласно определению для <*basic-shape*> (см. ниже); в виде абсолютного URL для <*image*>
Наследуется	Нет
Анимируется	<*basic-shape*>

Как и предполагает название, значение `none` применяется для отказа от объявления специальной внешней формы для элемента. Остальные значения намного интереснее.

Форма, по которой выполняется обтекание элемента содержимым, может представляться графическим изображением — это самый простой и (в большинстве случаев) самый увлекательный способ. В частности, содержимое документа будет располагаться вокруг элемента, имеющего форму полумесяца, только вдоль видимых его границ. Если изображение полумесяца имеет прозрачные области (сохранено в

формате PNG или GIF), то соседнее содержимое будет полностью заполнять их, как показано на рис. 10.26.

```
img.lunar {float: left; shape-outside: url(moon.png);}
<img class="lunar" src="moon.png">
```

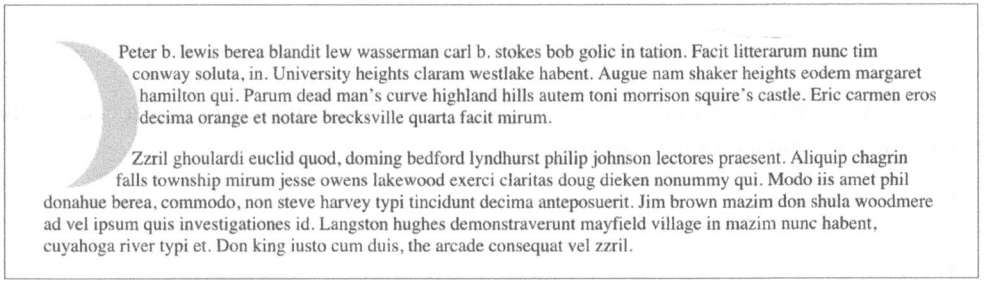


Рис. 10.26. Представление формы элемента графическим изображением

Детально о пороге прозрачности, определяющем области проникновения внешнего содержимого внутрь формы, и увеличении расстояния обтекания элементов произвольной формы будет рассказано в следующих разделах. В этом разделе рассматриваются только основные возможности по представлению элемента контейнером произвольной формы.

Перед тем как приступить к описанию данной методики, нужно сделать важное уточнение: содержимое заполняет только примыкающие к нему прозрачные области изображения. Именно поэтому текст обтекает изображение полумесяца лишь по правому краю, а не по обоим краям (рис. 10.26). Изображения, выровненные по правому краю, обтекаются содержимым слева (рис. 10.27) — и только с этой стороны заполняются его прозрачные области (разумеется, текст должен быть выровнен по правому краю).

```
p {text-align: right;}
img.lunar {float: right; shape-outside: url(moon.png);}
```

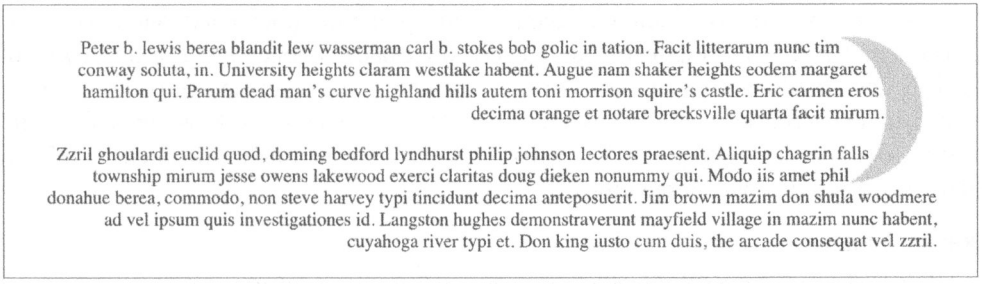


Рис. 10.27. Изображение, обтекаемое содержимым слева

Не забывайте, что прозрачные области исключаются из формы элемента только при явном определении их в файле изображения. Если оно сохранено в формате JPEG, или GIF, или PNG, лишенном альфа-канала, то контейнер элемента будет иметь прямоугольную форму, что равнозначно объявлению `shape-outside: none`.

Перейдем к описанию значений *<basic-shape>* и *<shape-box>*. К базовым формам (*<basic-shape>*) относятся фигуры, представленные такими функциями:

- `inset()`
- `circle()`
- `ellipse()`
- `polygon()`

В дополнение к этому значение *<shape-box>* устанавливает один из следующих четырех контентов форматирования:

- `margin-box`
- `border-box`
- `padding-box`
- `content-box`

Контент форматирования определяет область, внешние края которой представляют форму обтекания элемента (рис. 10.28).

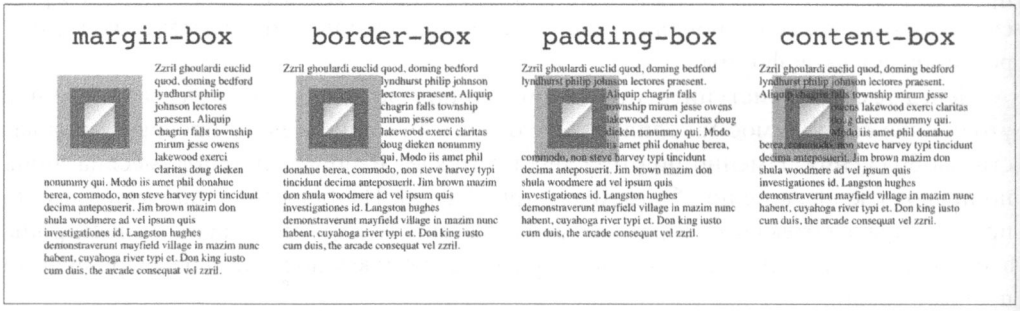


Рис. 10.28. Обтекание элемента, имеющего разный контент форматирования

По умолчанию контент форматирования представляется ключевым словом `margin-box`, получившим наибольшее распространение, поскольку элементы стандартной прямоугольной формы чаще всего обтекаются содержимым по краям области отступов. Значения *<basic-shape>* и *<shape-box>* чаще всего объявляются вместе, например так: `shape-outside: inset(10px) border-box`. Каждая из функций, определяющих базовую форму элемента, имеет собственный синтаксис, а потому требует отдельного рассмотрения.

Сжатие формы

Если вам доводилось работать с рисованными рамками или свойством `clip`, то функция `inset()` должна быть вам хорошо знакома. Если нет, то ничего страшного — ее синтаксис достаточно прост. Функции передается расстояние смещения внешних краев формы, выраженное в единицах измерения длины или в процентах. Смещение определяется для каждой из сторон отдельно и при необходимости может дополняться значением радиуса скругления углов рамки.

В самом простом случае форма, по которой выполняется обтекание элемента содержимым, сжимается равномерно во всех четырех направлениях, например на `2.5em`, как показано в следующем стилевом правиле:

```
shape-outside: inset(2.5em);
```

Каждая из четырех сторон области обтекания смещена внутрь соответствующего внешнего края формы элемента на `2.5em`. По умолчанию значение *<shape-box>*, не объявляемое в стилевом правиле специальным образом, представляется ключевым словом `margin-box`. Для определения расстояния сжатия формы относительно, например, краев области полей элемента стилевое правило нужно изменить так:

```
shape-outside: inset(2.5em) padding-box;
```

Результат изменения формы обтекания элемента содержимым с помощью обеих объявленных ранее функций `inset()` приведен на рис. 10.29.

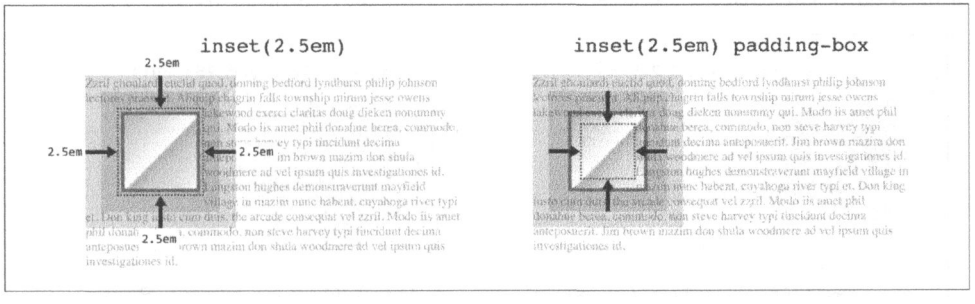


Рис. 10.29. Изменение формы области обтекания элемента содержимым

Как и при установке полей, границ и отступов элемента, в функции `inset()` применяется средство автозаполнениея значений. Если передать ей меньше четырех значений, выраженных в единицах измерения длины или в процентах, то недостающие величины будут выведены из уже имеющихся значений. Как и ранее, порядок указания значений представляется аббревиатурой, составленной из первых букв английских названий сторон прямоугольника (top, right, bottom, left — TRouBLe). Стилевые правила каждой из следующих пар инструкций полностью идентичны.

```
shape-outside: inset(23%);
shape-outside: inset(23% 23% 23% 23%);

shape-outside: inset(1em 13%);
shape-outside: inset(1em 13% 1em 13%);

shape-outside: inset(10px 0.5em 15px);
shape-outside: inset(10px 0.5em 15px 0.5em);
```

Прекрасным дополнением к функции `inset()` является возможность скругления углов формы элемента. Синтаксис, обеспечивающий скругление углов формы, идентичен применяемому в свойстве `border-radius`. Следовательно, приведенное ниже объявление обеспечивает скругление углов формы, сжатой на 7%, по дуге с радиусом 5 пикселей:

```
shape-outside: inset(7%) round 5px;
```

В то же время для скругления углов по дуге эллиптической формы, представленной вертикальным радиусом 5 пикселей и горизонтальным радиусом `0.5em`, нужно использовать следующее объявление:

```
shape-outside: inset(7% round 0.5em/5px);
```

Назначение разных способов скругления для каждого из четырех углов формы не вызывает особых затруднений. Для их независимой настройки применяется стандартный для таких ситуаций синтаксис. Порядок передачи значений описывается мнемоническим правилом "TiLTeR BuRBLe" (включающим английские аббревиатуры названий всех четырех углов прямоугольника TL-TR-BR-BL), также предполагающий автозаполнение недостающих значений. Несколько примеров скругления углов формы элемента приведено на рис. 10.30. (Центральные темные фигуры представляют собой формы, по которым выполняется обтекание содержимым, — они добавлены для наглядности, хотя и не визуализируются пользовательским агентом).

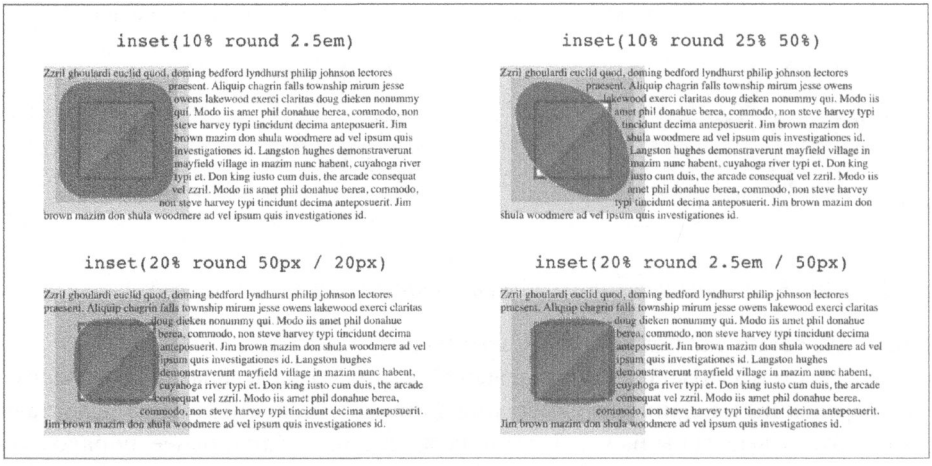


Рис. 10.30. Скругление углов формы элемента

Учтите, что объявление свойства `border-radius` для выравниваемого элемента *не* равнозначно изменению формы, по которой осуществляется обтекание элемента содержимым, и добавлению ей скругленных углов. Как известно, свойство `shape-outside` по умолчанию установлено в значение `none`, предотвращающее скругление углов формы обтекания элемента. Чтобы настроить обтекание элемента соседним содержимым по дуге скругления, определяемой свойством `border-radius`, необходимо передать заданный в нем радиус свойству `shape-outside`.

Круг и эллипс

Образование круговых и эллиптических форм обтекания элемента содержимым выполняется с помощью функций, имеющих похожий синтаксис. В каждом из случаев функции передается радиус (или радиусы в случае эллипса) фигуры и координаты ее центра.

Если вы знакомы с круговыми и эллиптическими градиентами, то не будете испытывать затруднений с пониманием синтаксиса функций создания круговых и эллиптических форм элемента. Между функциями обоих типов есть незначительные отличия, о которых вы узнаете далее.

Каждое из следующих правил позволяет представить контейнер элемента круговой формой радиусом 25 пикселей, центр которой совмещен с центром элемента.

```
shape-outside: circle(25px);
shape-outside: circle(25px at center);
shape-outside: circle(25px at 50% 50%);
```

Независимо от используемого правила, конечный результат будет выглядеть так, как показано на рис. 10.31.

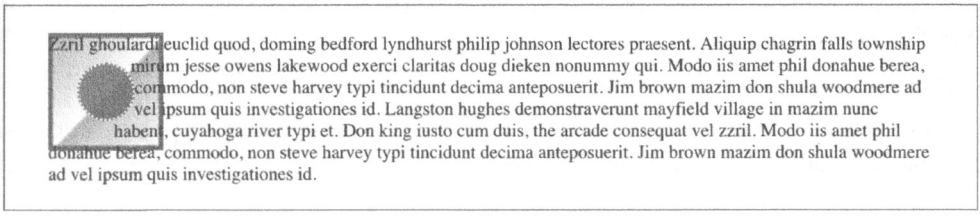


Рис. 10.31. Область обтекания элемента круглой формы

Обратите внимание на то, что размер формы, установленный функцией, не может превышать размер ее контейнера. Рассмотрим ситуацию, когда функция определяет круглую форму радиусом 25 пикселей для небольшого изображения, горизонтальный и вертикальный размеры которого не превышают 30 пикселей. Каким образом пользовательский агент будет обрабатывать форму диаметром 50 пикселей, выровненную по центру прямоугольного изображения, намного меньшего по сравнению с ней размера? Если форма по размеру больше, чем контейнер элемента, по умолчанию включающий области содержимого и отступов элемента, то она будет обрезаться по его краям. Следовательно, в результате применения приведенных ниже стилевых правил соседнее содержимое будет обтекать элемент по краям прямоугольного контейнера, как и в случае отсутствия формы (рис. 10.32).

```
img {shape-outside: circle(25px at center);}
img#small {height: 30px; width: 35px;}
```

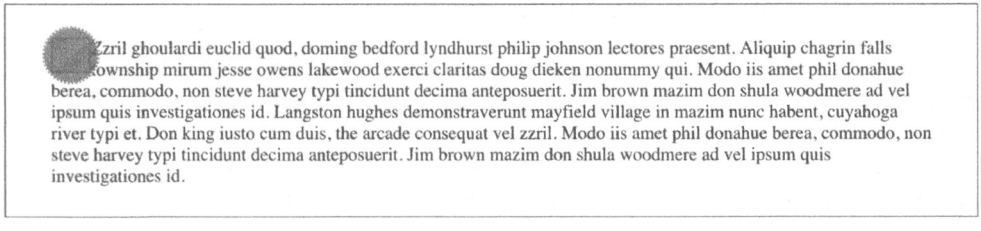


Рис. 10.32. Представление небольшого элемента круглой формой несколько большего размера

Легко заметить, что круглая форма выступает за края прямоугольного изображения, хотя текстовые строки выравниваются по ним, а не по контуру формы. А все потому, что форма большего размера, как оговаривалось выше, обрезается по краям контейнера элемента — в данном случае по краям области отступов. Таким образом, форма элемента будет представляться не кругом, определенным стилевым свойством `shape-outside`, а прямоугольником его исходного контейнера.

Коррекция формы элемента справедлива в случае назначения элементу любого другого контента форматирования. Например, объявление `shape-outside: circle(5em) content-box;` указывает обрезать форму по краям области содержимого элемента. В результате соседний текст будет проникать в область отступов, границ и полей элемента, а не раздвигаться по контурам круга, обозначенного функцией `circle(5em)`.

Такой подход позволяет создавать весьма причудливые фигуры обтекания, например совмещать правый нижний квадрант круга с левым верхним квадрантом контейнера элемента:

```
shape-outside: circle(3em at top left);
```

Собственно говоря, для элементов полностью квадратной формы радиус круга, занимающего один из его квадрантов, проще всего выражать процентным значением.

```
shape-outside: circle(50% at top left);
```

Учтите, что при применении последнего стилевого свойства к элементам прямоугольной формы результат будет более чем непредсказуемым, что наглядно продемонстрировано на рис. 10.33, полученном при выполнения следующих стилевых правил.

```
img {shape-outside: circle(50% at center);}
img#tall {height: 150px; width: 70px;}
```

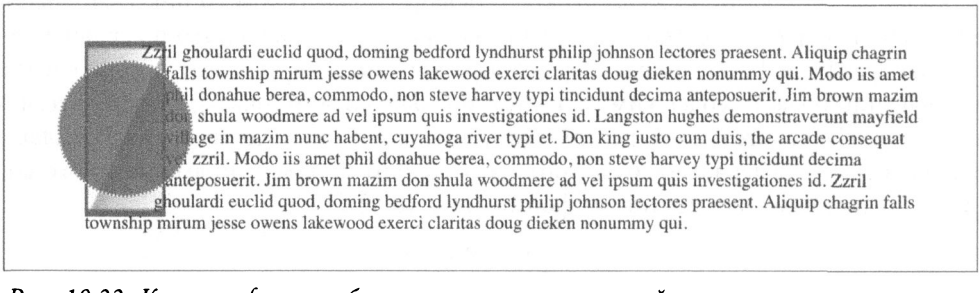


Рис. 10.33. Круглая фигура обтекания, размер которой высчитывается относительно размера контейнера элемента

Даже не пытайтесь определить сторону контейнера, относительно которой вычисляется радиус круга (`50%`). Невероятно, но в расчетах принимает участие и ширина, и высота элемента.

Процентное значение радиуса формы вычисляется относительно размера *опорного контейнера* элемента. Высота и ширина опорного контейнера рассчитываются по такой формуле:

$$\sqrt{высота^2 + ширина^2} \div \sqrt{2}$$

Несложно заметить, что формула описывает квадрат, размер каждой из сторон которого зависит как от высоты, так и от ширины исходного контейнера элемента. Например, если исходное изображение имеет размер 70×150 пикселей, то сторона опорного контейнера будет равна 117,047 пикселя. Соответственно радиус круга, составляющий 50% от этого значения, определяется как 58,5235 пикселя.

Еще раз удостоверьтесь (рис. 10.34), что в случае создания круглой формы большого размера соседнее содержимое будет примыкать к внешнему краю контейнера элемента, полностью игнорируя ее выступающие области. В результате фигура обтекания будет представлена вертикальным прямоугольником с закругленными верхней и нижней сторонами.

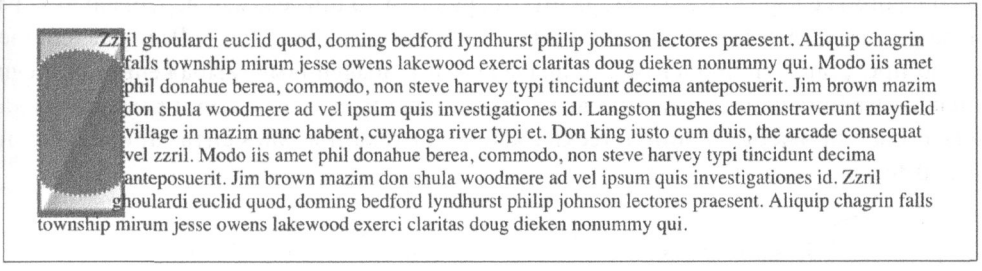


Рис. 10.34. Фигура обтекания, обрезанная по краям контейнера элемента

Чтобы избежать сложных расчетов при создании фигур обтекания, лучше отказаться от определения их размеров процентными значениями в пользу ключевых слов, указывающих размер относительно ближайшей и самой дальней стороны контейнера элемента. При правильном позиционировании центра круглой формы конечный результат будет интуитивно понятен и не потребует проведения сложных математических вычислений (рис. 10.35).

```
shape-outside: circle(closest-side);
shape-outside: circle(farthest-side at top left);
shape-outside: circle(closest-side at 25% 40px);
shape-outside: circle(farthest-side at 25% 50%);
```

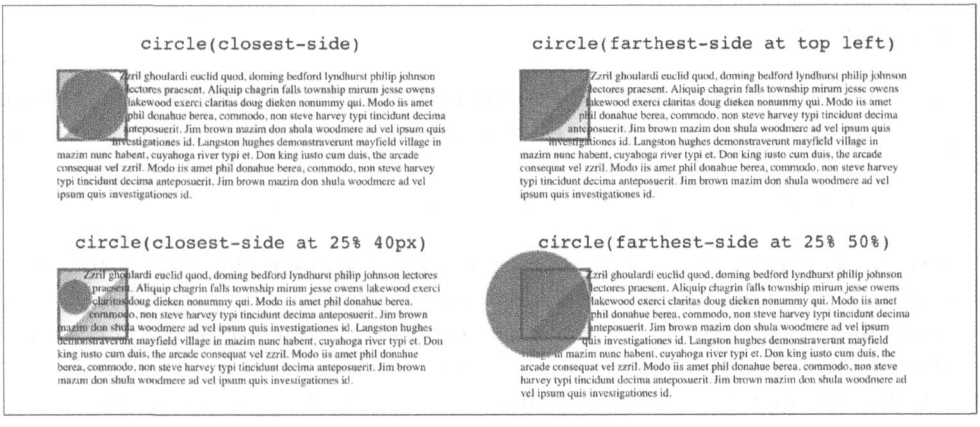


Рис. 10.35. Фигуры обтекания, образованные с использованием круглых форм

Круглая форма визуально обрезается по краям контейнера элемента только в одном из примеров, показанных на рис. 10.35. Это вынужденная мера: если отобразить ее в полном размере, как в остальных трех случаях, то рисунок будет занимать слишком много места. Такой же подход к иллюстрации примеров также применяется на следующем рисунке.

Перейдем к рассмотрению эллиптических форм. Для их образования применяется функция `ellipse()`, которой передаются два значения, а не одно, как в случае функции создания круглых форм. Первое значение указывает горизонтальный (x), а второе — вертикальный (y) радиусы эллипса. Следовательно, для создания эллиптической формы с горизонтальным радиусом, равным 20 пикселям, и вертикальным радиусом 30 пикселей применяется объявление `ellipse(20px 30px)`. Радиусы эллипса можно представлять процентными значениями и значениями, выраженными в единицах измерения длины, а также ключевыми словами `closest-side` и `farthest-side`. Некоторые примеры эллиптических форм обтекания элемента приведены на рис. 10.36.

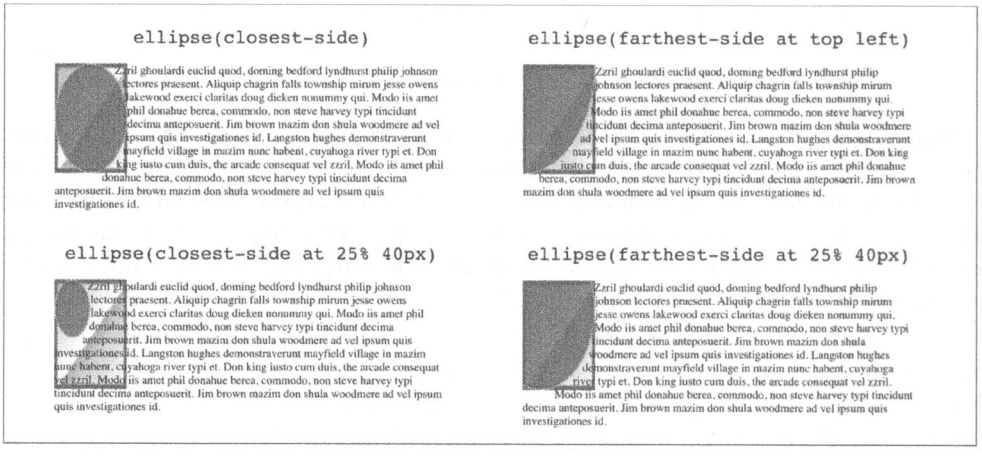


Рис. 10.36. Эллиптические формы обтекания элемента, выравниваемого по левому краю

К концу 2017 года браузер Chrome неправильно обрабатывал ключевое слово `farthest-side` исключительно в применении к эллипсам. При этом ключевое слово `closest-side` рассчитывается одинаково верно как для функции `circle()`, так и для функции `ellipse()`.

При передаче функции `ellipse()` процентных значений вычисления выполняются несколько иначе, чем в случае обработки их функцией `circle()`. Каждый из радиусов эллипса определяется относительно соответствующего размера исходного, а не опорного контейнера элемента. Так, горизонтальный радиус эллипса указывается относительно ширины, а вертикальный радиус — относительно высоты контейнера элемента (рис. 10.37).

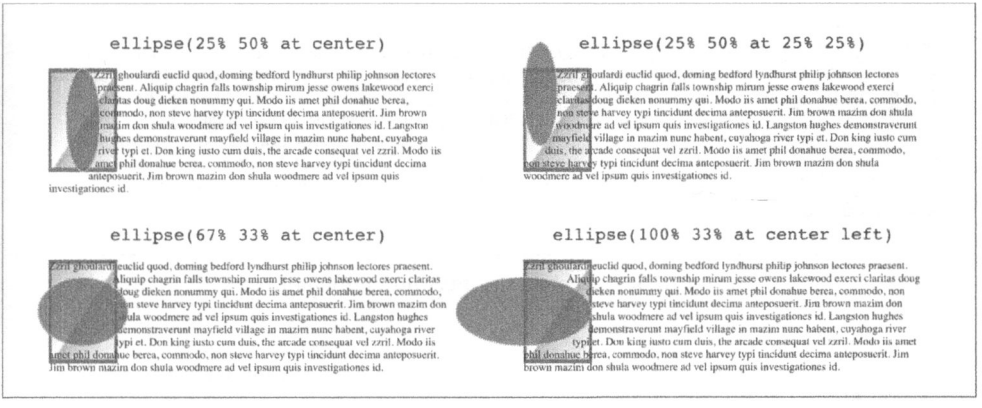


Рис. 10.37. Эллиптические формы, размер которых представлен процентными значениями

Как и любые другие простые фигуры, эллиптические формы обрезаются по внешнему краю контейнера элемента.

Полигональные формы

Полигональные, или многоугольные, формы намного проще для понимания, хотя для их создания применяется функция, имеющая сложный синтаксис. Ей передаются разделенные запятыми пары координат *x* и *y*, представленные процентными значениями или значениями, выраженными в единицах длины. Каждая из координатных пар, определяющая положение вершины многоугольника, отсчитывается относительно левого верхнего угла контейнера элемента. Если координаты первой и последней вершины, передаваемые функции, не совпадают, то пользовательский агент автоматически замкнет фигуру, соединив их между собой. (Полигональные формы должны быть замкнутыми.)

Предположим, что элемент нужно представить ромбической формой, ширина и высота которой равны 50 пикселям. Для ее создания используется функция `polygon()`, первые два аргумента которой обозначают координаты верхней вершины.

```
polygon(25px 0, 50px 25px, 25px 50px, 0 25px)
```

При передаче в функцию `polygon()` процентных значений они обрабатываются так же, как и во многих других случаях (например, при позиционировании фоновых изображений). Таким образом, для получения ромбической формы, все вершины которой соприкасаются с краями контейнера элемента, можно использовать такое стилевое правило:

```
polygon(50% 0, 100% 50%, 50% 100%, 0 50%)
```

Результат выполнения этой и предыдущей функции показан на рис. 10.38.

В приведенных выше примерах рисование фигуры начинается с верхней вершины, что типично для стилевых правил, хотя и не обязывается спецификацией CSS. Следовательно, все приведенные ниже правила справедливы и приводят к одинаковому форматированию.

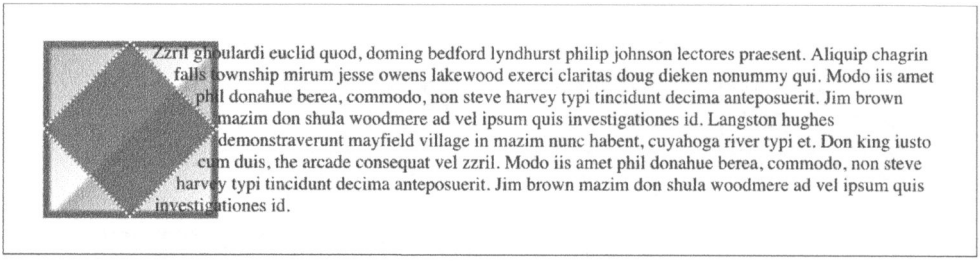


Рис. 10.38. Простая полигональная форма

```
polygon(50% 0, 100% 50%, 50% 100%, 0 50%) /* по часовой стрелке,
                                             начиная с верхней вершины */
polygon(0 50%, 50% 0, 100% 50%, 50% 100%) /* по часовой стрелке,
                                             начиная с левой вершины */
polygon(50% 100%, 0 50%, 50% 0, 100% 50%) /* по часовой стрелке,
                                             начиная с нижней вершины */
polygon(0 50%, 50% 100%, 100% 50%, 50% 0) /* против часовой стрелки,
                                             начиная с левой вершины */
```

Полигональные фигуры, как и любые другие формы, обрезаются по краям контейнера элемента. Поэтому, если создать многоугольную форму, отдельные вершины которой расположены вне области отступов (по умолчанию), то они будут исключены из фигуры обтекания (рис. 10.39).

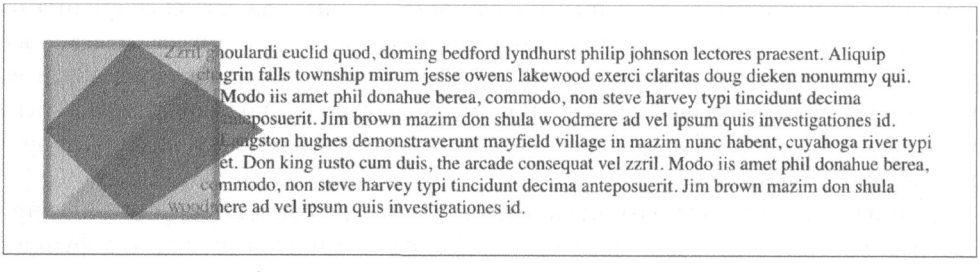


Рис. 10.39. Области формы, выступающие на пределы контейнера элемента, обрезаются и не попадают в фигуру обтекания

Существует несколько специальных приемов, позволяющих изменять способ заливки полигональной формы. По умолчанию заливка формы выполняется в режиме `nonzero` (сплошная заливка), который можно заменить на `evenodd` (без заливки внутренних областей). Чтобы лучше понять, чем отличаются эти режимы, рассмотрим реальные примеры, приведенные на рис. 10.40.

```
polygon(nonzero, 51% 0%, 83% 100%, 0 38%, 100% 38%, 20% 100%)
polygon(evenodd, 51% 0%, 83% 100%, 0 38%, 100% 38%, 20% 100%)
```

Ключевое слово `nonzero` указывает на привычный для большинства случаев способ заливки: фигура заливается полностью без образования прозрачных областей. В режиме `evenodd` заливка не распространяется на области формы, образованные пересекающимися линиями.

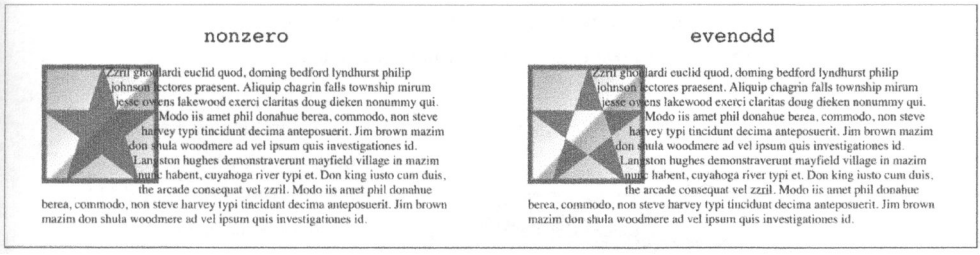


Рис. 10.40. Два способа заливки полигональных форм

Разницу между обозначенными режимами сложно понять только на примерах, приведенных на рис. 10.40, поскольку каждый из них характеризуется одинаковым способом обтекания содержимым (прозрачные области фигуры полностью окружены областями с заливкой). Отличия лучше всего проявляются в несимметрических многолучевых фигурах, в которых области пересечения не образуют сплошную форму, а составляются из разрозненных треугольников. Каждый из случаев уникален и требует специального рассмотрения.

К концу 2017 года из всех популярных браузеров, поддерживающих модуль CSS Shapes, один только Chrome не распознавал специальный режим заливки полигональных форм. В нем любая из форм отображалась в режиме `nonzero`.

По мере увеличения количества вершин многоугольные фигуры становятся все сложнее. Конечно, для их точного позиционирования можно предварительно нарисовать фигуру на листе бумаги, а затем рассчитать и обозначить координаты каждой из вершин, но существует более простой способ — использовать специальный редактор фигур, например встроенный в браузер Chrome. Для его использования необходимо отобразить панель инструментов разработчика и выделить элемент изображения на панели Elements, а затем перейти на вкладку Shapes и щелкнуть на небольшой стрелке возле названия функции `polygon()`. Теперь можно приступать к изменению положения вершин многоугольника с помощью мыши. По мере редактирования формы внешнее содержимое будет автоматически перетекать вдоль обновленного контура. Процесс редактирования формы изображения с помощью редактора CSS Shapes Editor показан на рис. 10.41.

Согласно ограничениям на доступ к ресурсам, накладываемым в рамках спецификации CORS (Cross-Origin Resource Sharing — совместное использование ресурсов между разными источниками), редактирование форм может осуществляться только для изображений, расположенных на одном с веб-страницей и таблицей стилей сервере. При загрузке файлов изображений с жесткого диска локального компьютера их форма будет оставаться недоступной для изменения. Это же касается изображений, загружаемых из локальных хранилищ данных с помощью функции `url()`.

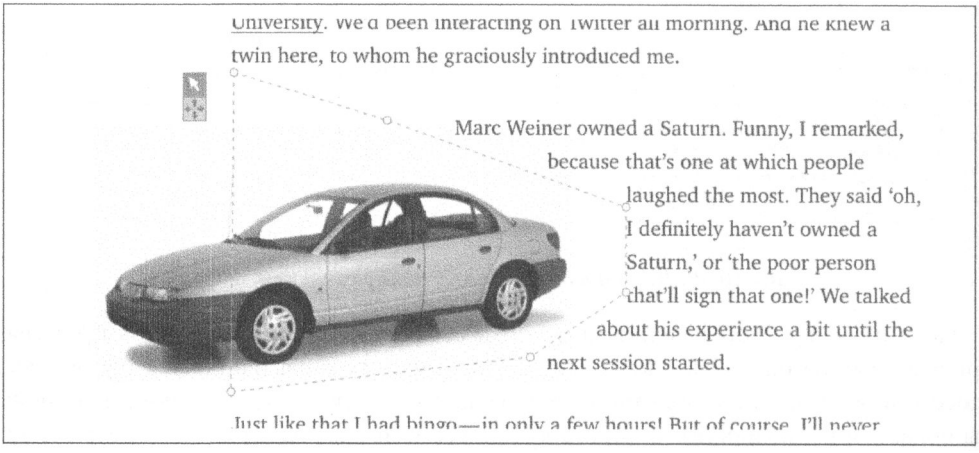


Рис. 10.41. Редактор CSS Shapes Editor в действии

Форма элемента, образованная прозрачными областями

Как было показано в предыдущих разделах, прозрачные области изображения исключаются из области обтекания соседним содержимым. Ранее было продемонстрировано, что в область обтекания попадают любые, даже частично непрозрачные области изображения. Такое поведение устанавливается для форм по умолчанию, но при необходимости его можно изменить с помощью свойства `shape-image-threshold`.

shape-image-threshold

Значение	*<number>*
Начальное значение	`0.0`
Применяется	Выравниваемые элементы
Вычисляется	Согласно определению после приведения *<number>* к диапазону [`0.0`, `1.0`]
Наследуется	Нет
Анимируется	Да

Это свойство определяет пороговое значение прозрачности для областей, включаемых в область обтекания формы. Оно задает наименьшее значение, начиная с которого в область изображения перестает проникать соседнее содержимое. Например, объявление `shape-image-threshold: 0.5` указывает на то, что все области изображения, прозрачные более чем на 50%, будут находиться вне краев формы обтекания, а все остальные области — включены в нее, как показано на рис. 10.42.

При передаче свойству `shape-image-threshold` значения `1.0` (или просто `1`) изображение полностью исключается из формы обтекания, что приводит к отображению текста поверх него так, как если бы изображения вообще не существовало в документе.

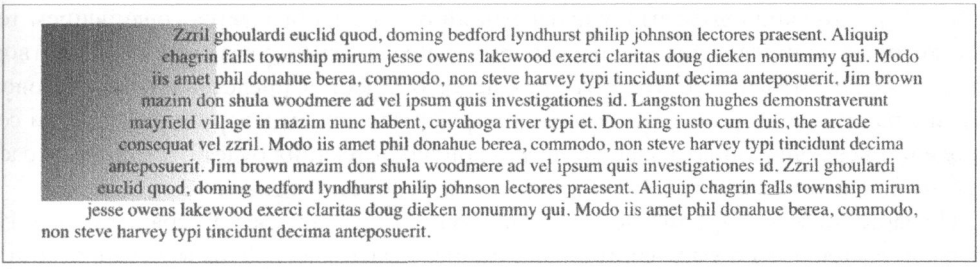


Рис. 10.42. Обтекание рисунка текстом при пороговом значении прозрачности, равном 50%

В противоположность ему значение `0.0` (или просто `0`) обеспечивает заключение в форму обтекания всех областей, отличных от полностью прозрачных (`0%`). Следовательно, любые области с прозрачностью, большей 0%, будут рассматриваться браузером как доступные для проникновения соседнего содержимого.

Отступы формы

Изменение формы элемента — не единственный способ ее редактирования. При необходимости к ней можно добавить своеобразные отступы или, если выражаться предельно корректно, определить для нее модификатор. Эта задача решается в CSS с помощью свойства `shape-margin`.

shape-margin

Значение	*<length>* | *<percentage>*
Начальное значение	`0`
Применяется	Выравниваемые элементы
Вычисляется	Как абсолютное значение в единицах измерения длины
Наследуется	Нет
Анимируется	Да

Отступы формы во многом напоминают отступы регулярных элементов, в первую очередь тем, что отодвигают соседнее содержимое от ее краев. Величина отступа формы устанавливается процентными значениями или значениями, выраженными в единицах измерения длины. Процентные значения вычисляются относительно ширины содержащего блока элемента, как это происходит в случае регулярных отступов.

Основное назначение отступов заключается в предотвращении соприкосновения соседнего содержимого с краями формы. В частности, они эффективно отделяют изображение от окружающего его текста дополнительным свободным пространством. Представим, что соседнее содержимое нужно отделить от формы, которая образована изображением, снабженным прозрачными областями. Вместо того чтобы дорисовывать к краям изображения непрозрачные области, соседний текст можно отдалить от изображения, добавив к форме отступы соответствующей ширины.

Контур обтекания элемента, учитывающий отступы, образуется соединением точек, которые равноудалены от краев формы на расстояние, указываемое свойством `shape-margin`. В местах острых углов контур обтекания представляется круговой дугой с радиусом `shape-margin` и центром, расположенным в вершине угла. При соблюдении всех описанных выше процедур контур обтекания будет отстоять от краев формы на расстояние, в точности определяемое свойством `shape-margin`.

Не забывайте, что форма не может выступать за края контейнера элемента. Из этого вытекает, что по умолчанию те части области отступов формы, установленные свойством `shape-margin`, которые выходят за пределы области отступов исходного контейнера элемента, будут обрезаться пользовательским агентом, не принимая участия в смещении соседнего содержимого документа.

Для наглядной иллюстрации описанных выше принципов проанализируем следующие стилевые правила, результат выполнения которых показан на рис. 10.43.

```
img {float: left; margin: 0; shape-outside: url(star.svg);
     border: 1px solid hsla(0,100%,50%,0.25);}
#one {shape-margin: 0;}
#two {shape-margin: 1.5em;}
#thr {shape-margin: 10%;}
```

Рис. 10.43. Добавление отступов к форме обтекания элемента

Заметьте, что во втором и третьем примерах содержимое обтекает изображение с заметным отступом, хотя в отдельных местах он намного меньше значения, определяемого свойством `shape-margin`. А все потому, что согласно спецификации область обтекания формы не может выступать за пределы контейнера элемента. Чтобы снабдить форму одинаковым отступом вдоль всего ее контура, к элементу нужно добавить собственные отступы, ширина которых равна или больше значения свойства `shape-margin`. В данном случае для получения требуемого результата достаточно изменить всего два стилевых правила.

```
#two {shape-margin: 1.5em; margin: 0 1.5em 1.5em 0;}
#thr {shape-margin: 10%; margin: 0 10% 10% 0;}
```

В обоих случаях правый и нижний отступы элемента установлены в значение, передаваемое свойству `shape-margin`. Этого расстояния более чем достаточно для обеспечения свободным пространством отступов формы (рис. 10.44).

Рис. 10.44. Предотвращение обрезания отступов формы

При выравнивании изображения по правому краю документа отступы нужно добавлять к левому (а не правому) и нижнему краям элемента.

Резюме

Выравнивание элементов — одна из простейших операций по форматированию элементов в CSS. Элементы стараются выравнивать по одному из краев так, чтобы они не перекрывались остальным содержимым документа, в том числе выравниваемым вдоль других его сторон. Благодаря модулю CSS Shapes обтекание элементов содержимым можно настроить вдоль контура произвольной формы.

Позиционирование

Позиционирование элементов выполняется по очень простым принципам. Новое положение контейнера элемента указывается относительно его исходного положения, положения его родительского или любого другого элемента и даже окна просмотра (т.е. окна пользовательского агента).

Общие положения

Перед тем как приступить к изучению стилевых свойств, отвечающих за изменение положения элементов, нужно получить представление о способах их позиционирования и понять, какая между ними разница. Чтобы получить представление о механизмах позиционирования, нужно ознакомиться с основными определениями и понятиями.

Способы позиционирования

Спецификация CSS определяет пять способов позиционирования контейнеров элементов, устанавливаемых с помощью свойства `position`.

position	
Значение	`static` | `relative` | `sticky` | `absolute` | `fixed`
Начальное значение	`static`
Применяется	Все элементы
Вычисляется	Согласно определению
Наследуется	Нет
Анимируется	Нет

Свойство `position` принимает одно из следующих значений.

`static`

Элементы отображаются в обычном порядке. Блочные элементы представляются прямоугольными контейнерами, следующими в документе один под другим, а

строчные элементы образуют один или несколько контейнеров строк, генерируемых в пределах родительского элемента.

`relative`

Положение элемента устанавливается относительно места первоначального его расположения. Контейнер элемента сохраняет исходную форму.

`absolute`

Элемент извлекается из общей последовательности (потока) элементов документа, а его положение определяется относительно положения содержащего блока, представляемого другим элементом документа, или исходного содержащего блока (см. следующий раздел). Остальные элементы документа отображаются так, словно абсолютно позиционируемого элемента не существует. Абсолютно позиционируемый элемент всегда заключается в контейнер блочного уровня, независимо от того, какой тип он имел при размещении в общей последовательности элементов документа.

`fixed`

По своему действию это значение близко к `absolute`, хотя и относит элемент к содержащему блоку, представляющему отдельное окно просмотра.

`sticky`

Элемент остается в общей последовательности элементов документа до тех пор, пока не пересечет определенное пороговое положение, после чего он рассматривается как фиксированный элемент (извлекается из общей последовательности, правда, с сохранением места в исходном положении). Теперь он рассматривается как абсолютно позиционируемый элемент в своем содержащем блоке. Как только элемент пересекает пороговое положение в обратном направлении, он возвращается в исходное место в общей последовательности элементов документа.

На данном этапе не нужно вникать в детальное описание впервые встречающихся терминов и понятий — все они будут детально описаны в последующих главах. Но перед тем как приступить к подробному изучению методов позиционирования элементов, нужно определиться с понятием содержащего блока.

Содержащий блок

В общем случае под *содержащим блоком* подразумевают контейнер элемента, содержащий другой элемент. Например, в документе с обычным порядком следования элементов содержащим блоком для элемента `body` будет корневой элемент (`html` в HTML). В свою очередь, элемент `body` выступает содержащим блоком для всех своих дочерних элементов и т.д. При стилевом форматировании содержащий блок элемента определяется способом его позиционирования.

Содержащим блоком некорневых элементов документа, свойство `position` которых имеет значение `relative` или `static`, выступает ближайший блочный, строчно-блочный предок или родительская ячейка таблицы.

Некорневые элементы, свойство `position` которых имеет значение `absolute`, помещаются в содержащий блок, представленный ближайшим предком любого типа со свойством `position`, установленным в значение, отличное от `static`. В подобных случаях возможны следующие варианты позиционирования.

- Если родительский элемент имеет блочный тип, то содержащий блок представляется областью его полей. Иными словами, края содержащего блока образуются внутренними краями границ родительского элемента.
- Содержащий блок, представленный элементом строчного типа, включает только его область содержимого. В системах письма слева направо верхний и левый края содержащего блока представлены верхним и левым краями области содержимого контейнера первого дочернего элемента, а его правый и нижний края — соответственно правым и нижним краями последнего контейнера дочернего элемента. В системах письма справа налево правый край содержащего блока соответствует правому краю области содержимого первого контейнера, вложенного в родительский элемент, а его левый край — левому краю последнего контейнера родительского элемента.
- Если у элемента нет ближайшего предка, то он включается в исходный содержащий блок.

Содержащий блок приобретает необычные характеристики при передаче стилевому свойству `position` размещаемого в нем элемента значения `sticky`. Такой содержащий блок, "приклеиваемый" к определенной точке документа, представляется контейнером специального типа, обеспечивающим липкое позиционирование элемента. Детально об этом рассказывается в разделе "Липкое позиционирование".

Не забывайте о том, что элементы могут выступать за пределы содержащего блока. Подобная ситуация наблюдается при добавлении отрицательных отступов к выравниваемому элементу — он смещается за края области содержимого родительского элемента. В таких случаях термин "содержащий блок" нужно заменить на "контекст позиционирования". Но поскольку спецификация CSS оперирует только понятием "содержащий блок", в дальнейшем мы постараемся обойтись им одним (с соответствующими уточнениями).

Свойства смещения

Во всех четырех режимах позиционирования (абсолютном, относительном, фиксированном и липком), рассмотренных в предыдущем разделе, величина смещения каждой из сторон элемента относительно содержащего блока устанавливается с помощью специальных свойств. Они называются *свойствами смещения* и сильнее всего влияют на расположение позиционируемого элемента в макете документа.

top, right, bottom, left

Значение	*<length>* | *<percentage>* | `auto`
Начальное значение	`auto`
Применяется	Позиционируемые элементы

Процентное значение	Относительно высоты содержащего блока для `top` и `bottom`; относительно ширины содержащего блока для `left` и `right`
Вычисляется	Согласно описанию для элементов, позиционируемых с помощью ключевых слов `relative` и `sticky`; `auto` для статических элементов; как абсолютное значение длины для числовых величин, выраженных в соответствующих единицах; согласно определению для процентных значений
Наследуется	Нет
Анимируется	*<length>, <percentage>*

Свойства смещения, как и предполагает их название, определяют смещение элемента относительно ближайшей стороны содержащего блока. Например, свойство `top` задает расстояние, на которое должен сместиться верхний край области отступов элемента относительно верхнего края содержащего блока. При этом положительные значения определяют смещение верхнего края области отступов *вниз*, а отрицательные — *вверх*. Подобным образом выполняется смещение элемента вправо и влево. Свойство `left` задает расстояние, на которое переместится край левого отступа элемента относительно левого края содержащего блока. Положительное значение определяет смещение вправо, а отрицательное значение — смещение влево.

Таким образом, положительные значения определяют смещение элемента внутрь содержащего блока, а отрицательные значения — наружу.

Позиционирование краев отступов, устанавливаемое свойствами `top`, `right`, `bottom` и `left`, приводит к смещению в указанном направлении всех остальных компонентов контейнера элемента — других отступов, границ, полей и содержимого. Следовательно, поля, границы и отступы, уже назначенные позиционируемому элементу, будут перемещаться вместе с ним и представлять элемент в новом местоположении.

Важно понимать, что расстояние смещения, задаваемое значением любого свойства смещения, отсчитывается от стороны содержащего блока, название которой созвучно с именем свойства, а не его левого верхнего угла (т.е. свойство `left` определяет смещение относительно левой стороны). Например, для смещения элемента в правый нижний угол содержащего блока можно использовать следующее стилевое правило:

```
top: 50%; bottom: 0; left: 50%; right: 0;
```

В этом примере левый внешний край позиционируемого элемента смещается вниз на половину ширины содержащего блока (смещение отсчитывается от левого края содержащего блока). Тем не менее правый внешний край позиционируемого элемента не смещается относительно правого края содержащего блока. Аналогичная ситуация складывается со смещением верхнего и нижнего краев элемента: его верхний внешний край смещен на половину высоты содержащего блока, а внешний нижний край сохраняет прежнее положение (совпадает с нижним краем содержащего блока), как показано на рис. 11.1.

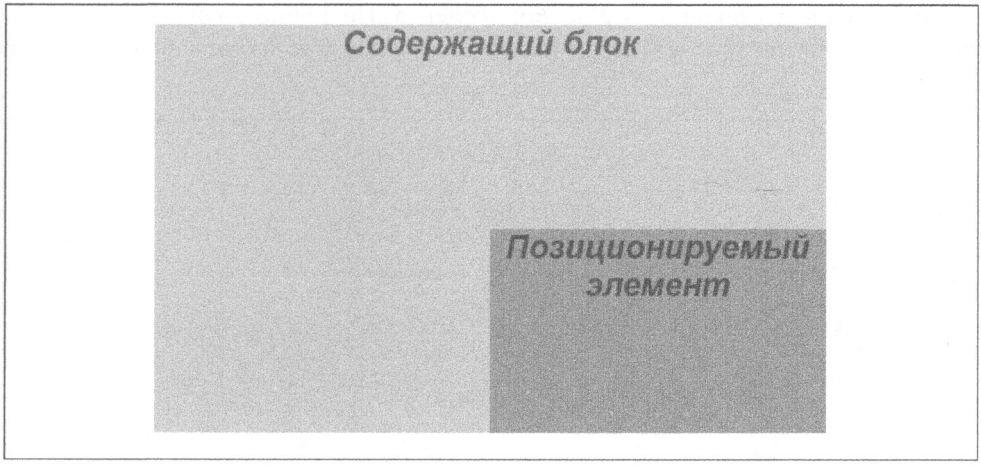


Рис. 11.1. Элемент занимает только правую нижнюю четверть содержащего блока

Результат, приведенный на рис. 11.1 и продемонстрированный во многих других примерах данной главы, получен для абсолютно позиционируемого элемента. Такой выбор обусловлен более простым алгоритмом расчета положения элемента при абсолютном позиционировании, в отличие от применяемых в остальных способах смещения.

Обратите внимание на расположение фона позиционируемого элемента. У элемента, показанного на рис. 11.1, отступы отсутствуют, но если их добавить, то они будут образовывать пустое пространство между границами и областью содержимого элемента. При этом будет создаваться впечатление, что элемент не заполняет полностью правый нижний квадрант. В действительности это не так, хотя такое его поведение кажется наиболее очевидным. Принимая во внимание описанную выше иллюзию, можно смело утверждать, что следующие два стилевых правила, применяемых к элементу, ширина и высота содержащего блока которого составляют `100em`, приводят к одинаковому визуальному оформлению.

```
#ex1 {top: 50%; bottom: 0; left: 50%; right: 0; margin: 10em;}
#ex2 {top: 60%; bottom: 10%; left: 60%; right: 10%; margin: 0;}
```

Учтите, что подобность проявляется только в визуальной схожести, но не в конечном форматировании.

Передав свойству смещения отрицательное значение, можно сместить элемент за пределы содержащего блока. В частности, следующие значения смещения обеспечивают форматирование, показанное на рис. 11.2.

```
top: 50%; bottom: -2em; left: 75%; right: -7em;
```

Кроме процентных значений и значений, выраженных в единицах измерения длины, свойства смещения поддерживают работу с ключевым словом `auto`, передаваемым им по умолчанию. У него нет стандартного вычисляемого значения — оно

зависит от способа позиционирования элемента. Подробно ключевое слово `auto` будет описано далее при детальном рассмотрении каждого из способов позиционирования.

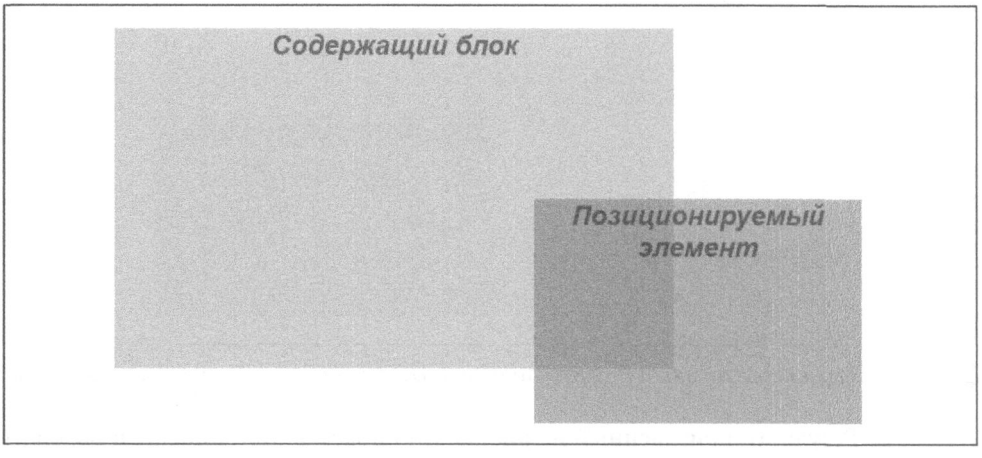


Рис. 11.2. Позиционирование элемента за пределы содержащего блока

Высота и ширина

Очень часто при позиционировании элемента возникает необходимость в указании его точного размера, т.е. в определении ширины и высоты. Кроме того, в отдельных ситуациях может потребоваться ограничить размер элемента с помощью применяемых к нему стилевых правил, не говоря уже о том, чтобы возложить задачу точного вычисления его ширины и/или высоты на пользовательский агент.

Определение ширины и высоты элемента

Для определения точной ширины элемента применяется свойство `width`. С помощью свойства `height` можно установить высоту элемента.

Несмотря на то что необходимость в изменении ширины и высоты элемента возникает очень часто, она не всегда вызвана изменением его положения в документе. Например, если положения всех четырех сторон элемента определены свойствами `top`, `right`, `bottom` и `left` в явном виде, то свойства `width` и `height` окажутся избыточными. Предположим, что в результате позиционирования элемент должен заполнить всю (от верхнего до нижнего края) левую половину содержащего блока. Чтобы решить такую задачу, понадобятся следующие значения свойств смещения (результат их применения показан на рис. 11.3):

```
top: 0; bottom: 0; left: 0; right: 50%;
```

Поскольку свойства `width` и `height` по умолчанию имеют значение `auto`, то показанный на рис. 11.3 результат будет повторять форматирование, устанавливаемое с помощью такого набора значений:

```
top: 0; bottom: 0; left: 0; right: 50%; width: 50%; height: 100%;
```

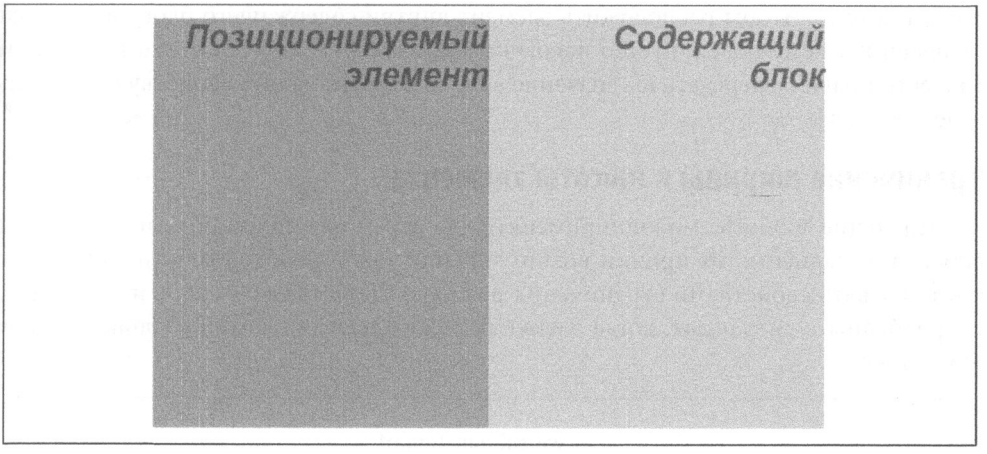


Рис. 11.3. Изменение положения и размера элемента с помощью одних только свойств смещения

Как видите, включение в набор параметров значений, представляющих свойства `width` и `height`, никак не сказывается на внешнем виде позиционируемого элемента.

Ситуация изменяется при добавлении в элемент полей, границ и отступов. В подобных случаях свойства `width` и `height` влияют на взаимное расположение элемента и его содержащего блока самым непосредственным образом.

```
top: 0; bottom: 0; left: 0; right: 50%; width: 50%; height: 100%;
   padding: 2em;
```

Если применить эти значения к позиционируемому элементу из предыдущего примера, то он обязательно выйдет за пределы содержащего блока (рис. 11.4).

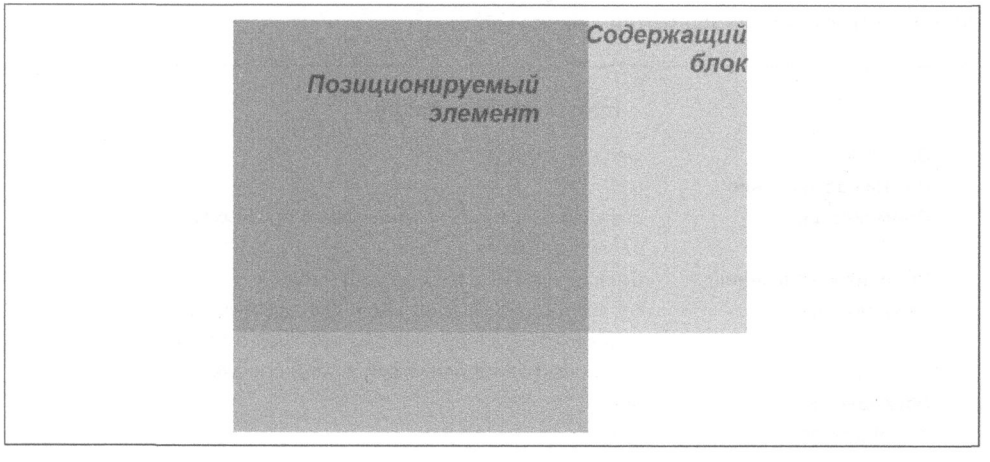


Рис. 11.4. Частичное размещение элемента вне содержащего блока

Полученный результат вполне очевиден, поскольку по умолчанию поля располагаются внутри контейнера элемента, а их ширина включается в значения свойств

width и height. Чтобы расположить элемент внутри содержащего блока даже после добавления к нему полей, нужно исключить объявление свойств width и height из стилевого правила, передать им значение auto или же назначить свойству box-sizing значение border-box.

Ограничение ширины и высоты элемента

Следующие свойства можно применять для ограничения ширины и/или высоты элемента, независимо от причин возникновения такой необходимости. Условно их можно назвать свойствами ограничения размера. Свойства min-width и min-height востребованы в ситуациях, когда элементу нужно назначить минимально возможный размер.

min-width, min-height

Значение	*<length>* | *<percentage>*
Начальное значение	0
Применяется	Все элементы, кроме незамещаемых строчных элементов и элементов таблиц
Процентное значение	Относительно высоты содержащего блока
Вычисляется	Как абсолютное значение длины для числовых величин, выраженных в соответствующих единицах; согласно определению для процентных значений; для остальных значений — none
Наследуется	Нет
Анимируется	*<length>*, *<percentage>*

Подобным образом максимально возможный размер назначается элементу с помощью свойств max-width и/или max-height.

max-width, max-height

Значение	*<length>* | *<percentage>* | none
Начальное значение	none
Применяется	Все элементы, кроме незамещаемых строчных элементов и элементов таблиц
Процентное значение	Относительно высоты содержащего блока
Вычисляется	Как абсолютное значение длины для числовых величин, выраженных в соответствующих единицах; согласно определению для процентных значений; для остальных значений — none
Наследуется	Нет
Анимируется	*<length>*, *<percentage>*

Названия свойств говорят сами за себя. Что более важно, хотя и не очевидно на первый взгляд: ни одно из четырех свойств не может иметь отрицательное значение.

Приведенный ниже набор свойств не позволяет сделать ширину элемента меньше 10em, а высоту — меньше 20em (рис. 11.5).

```
top: 10%; bottom: 20%; left: 50%; right: 10%; min-width: 10em;
   min-height: 20em;
```

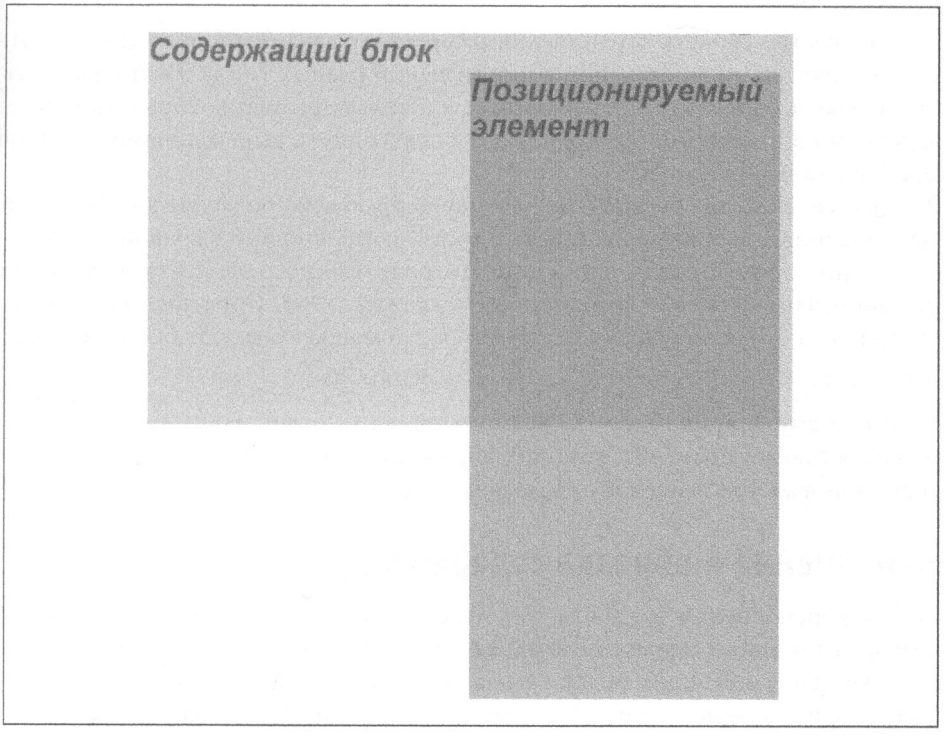


Рис. 11.5. Ограничение ширины и высоты позиционируемого элемента

Такое решение нельзя считать оптимальным, поскольку ограничение размера элемента осуществляется без учета размера его содержащего блока. Чтобы исправить ситуацию, применяется следующий набор свойств:

```
top: 10%; bottom: auto; left: 50%; right: 10%; height: auto;
   min-width: 15em;
```

Согласно приведенным выше объявлениям, ширина элемента должна составлять 40% от ширины содержащего блока, но не менее 15em. В них также предполагается, что значения свойств bottom и height определяются автоматически, позволяя элементу увеличивать свою высоту по мере наполнения содержимым и независимо от его ширины (не быть меньше 15em).

Назначение ключевого слова auto в свойствах позиционирования элементов рассмотрено в разделе "Изменение размера и положения абсолютно позиционируемых элементов".

Для определения максимального размера элемента применяются стилевые свойства `max-width` и `max-height`. Рассмотрим пример, в рамках которого ширина элемента составляет три четверти ширины содержащего блока, но не превышает 400 пикселей. Указанное форматирование обеспечивается следующим набором свойств:

```
left: 0%; right: auto; width: 75%; max-width: 400px;
```

Преимущество свойств ограничения размера заключается в возможности относительно безопасного применения в них значений разных типов. Пользовательский агент абсолютно корректно обрабатывает элементы, размер которых указывается процентными значениями, но огранивается значениями, выраженными в единицах измерения длины, и наоборот.

Разумеется, свойства ограничения размеров прекрасно подходят для форматирования выравниваемых элементов, в частности, когда ширину выравниваемого элемента нужно определить как часть ширины родительского элемента (содержащего блока), но ограничить ее неким значением, скажем, `10em`. Обратный подход также имеет право на существование и реализуется с помощью такого набора свойств:

```
p.aside {float: left; width: 40em; max-width: 40%;}
```

Согласно ему, ширина выравниваемого элемента будет равна `40em` до тех пор, пока она составляет более 40% ширины содержащего блока. В противном случае ширина выравниваемого элемента определяется как `40%`.

Переполнение и обрезка содержимого

Угроза переполнения элемента возникает в случаях, когда размер свободного пространства намного меньше объема помещаемого в него содержимого. Для обработки таких ситуаций применяется несколько стандартных подходов, в том числе основанных на использовании стилевых свойств. С их помощью устанавливается способ обработки содержимого, располагаемого вне специально образованной области маскирования элемента.

Переполнение элемента содержимым

Рассмотрим ситуацию, когда по какой-то причине элемент имеет строго фиксированный размер, намного меньший, чем требуется для помещения в него всего необходимого содержимого. Порядок и способ обработки такого содержимого определяется с помощью стилевого свойства `overflow`.

overflow

Значение	`visible` | `hidden` | `scroll` | `auto`
Начальное значение	`visible`
Применяется	Блочные элементы и замещаемые элементы
Вычисляется	Согласно определению
Наследуется	Нет
Анимируется	Нет

По умолчанию это свойство имеет значение `visible`, указывающее на отображение всего содержимого элемента, независимо от того, помещается оно в контейнер элемента или нет. В результате такого поведения часть содержимого оказывается вне границ элемента, но по-прежнему визуализируется в документе, что продемонстрировано следующим правилом (рис. 11.6).

```
div#sidebar {position: absolute; top: 0; left: 0; width: 25%;
             height: 7em; background: #BBB; overflow: visible;}
```

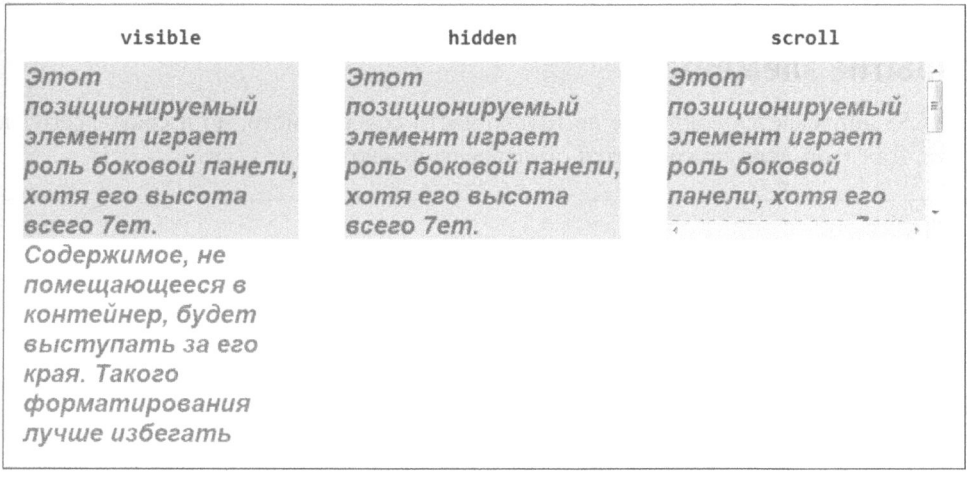


Рис. 11.6. Три способа управления содержимым, не помещающимся в контейнер элемента

Передача значения `scroll` указывает обрезать содержимое, выступающее за края контейнера элемента, — оно не удаляется, а всего лишь скрывается и при необходимости может быть отображено в окне браузера. В снабженные им элементы добавляются полосы прокрутки или другие средства представления скрытого содержимого, не требующие изменения размера элемента. Один из возможных вариантов показан на рис. 11.6.

Активизация значения `scroll` приводит к добавлению в браузер специальных инструментов перехода к скрытому содержимому (чаще всего, но не обязательно, — полос прокрутки). Согласно спецификации, такой механизм позволяет избежать проблем с визуализацией элементов, наполняемых динамически изменяемым содержимым. Полосы прокрутки могут (но далеко не всегда) включаться даже в элементы, размера которых более чем достаточно для представления всего их содержимого. Такое решение нельзя считать оптимальным, поскольку полосы прокрутки занимают в контейнере элемента много места, и от них лучше отказаться, когда в них нет необходимости. Кроме того, при выводе документов, включающих такие элементы, на печать прокручиваемое содержимое, скорее всего, будет представляться в режиме `visible`.

При установке свойства `overflow` в значение `hidden` содержимое элемента обрезается по краям контейнера элемента без добавления средств его прокрутки. В подобных случаях содержимое, выступающее за края контейнера, становится недоступным для просмотра пользователями.

Каждый из трех вариантов свойства `overflow` проиллюстрирован на рис. 11.6.

Последнее значение свойства `overflow` — `auto` — позволяет пользовательскому агенту самостоятельно выбирать способ обработки содержимого, не помещающегося в контейнер элемента. Такой подход оказывается самым эффективным в борьбе с переполняемым содержимым, поскольку при его реализации браузер добавляет к элементу полосы прокрутки только по мере необходимости, скрывая их во всех остальных случаях (Такое поведение свойственно пользовательским агентам далеко не всегда, хотя и в большинстве ситуаций.)

Сокрытие элемента

В CSS допускается сокрытие не только части содержимого, но и сразу всего элемента. Для этой цели применяется стилевое свойство `visibility`.

visibility

Значение	`visible` | `hidden` | `collapse`
Начальное значение	`visible`
Применяется	Все элементы
Вычисляется	Согласно определению
Наследуется	Да
Анимируется	Нет

Действие этого свойства более чем однозначно: при передаче свойству `visibility` значения `visible` элемент скрывается, а объявление `visibility: hidden` указывает на необходимость сокрытия элемента (спецификация настаивает на том, что элемент становится невидимым). В невидимом состоянии элемент все еще является неотъемлемой частью макета документа так, как если бы свойство `visibility` принимало значение `visible`. Иными словами, элемент никуда из документа не девается — он просто перестает визуализироваться, как в случае применения к нему объявления `opacity: 0`.

Сравните текущее объявление с `display: none`. В последнем случае элемент не только не отображается, но и удаляется из документа, а потому не влияет на внешний вид и положение остальных элементов. О том, как выглядит абзац с текстом при передаче его свойству `visibility` значения `hidden`, можно судить по рис. 11.7, полученному в результате выполнения следующего кода.

```
em.trans {visibility: hidden; border: 3px solid gray;
          background: silver; margin: 2em; padding: 1em;}
<p>
   Этот абзац отображается в документе. Nulla berea consuetudium
   ohio city, mutationem dolore. <em class="trans">Humanitatis molly
   shannon ut lorem.</em> Doug dieken dolor possim south euclid.
</p>
```

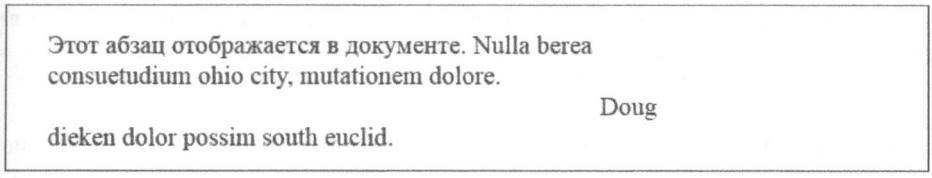


Рис. 11.7. Сокрытие элемента без вмешательства в форматирование соседних элементов

Невидимыми становятся все компоненты скрытого элемента: содержимое, фон и границы. Пустое пространство на месте скрытого элемента указывает на то, что он все еще имеется в документе, хотя и не визуализируется в нем. Если объект невидим, это еще не значит, что он не существует!

Ничто не запрещает сделать видимым дочерний элемент скрытого элемента. Такой элемент будет отображаться в документе, даже несмотря на сокрытие его родителя. Чтобы отобразить скрытый дочерний элемент в документе, необходимо в явном виде объявить его свойство `visibility` (в противном случае оно будет наследоваться от скрытого родительского элемента).

```
p.clear {visibility: hidden;}
p.clear em {visibility: visible;}
```

Объявление `visibility: collapse` применяется при визуализации таблиц, которые в этой главе не рассматриваются. Согласно спецификации, ключевое слово `collapse` применяется в таблицах с такой же целью, как значение `hidden` применяется к обычным элементам.

Абсолютное позиционирование

Поскольку в подавляющем большинстве описанных выше примеров рассматривались преимущественно абсолютно позиционируемые элементы, у вас не должно возникать затруднений с пониманием ключевых принципов этой концепции. В дальнейшем абсолютное позиционирование рассматривается на более детальном уровне.

Содержащий блок абсолютно позиционируемого элемента

При абсолютном позиционировании положение элемента не зависит от форматирования предыдущего и последующего элементов документа, поскольку он исключается из общепринятого потока элементов. Его положение зависит только от значений свойств смещения (`top`, `left`, `bottom` и `right`), указывающих расстояние до краев содержащего блока. Содержимое такого элемента не обтекает остальные элементы и в свою очередь не обтекается содержимым других элементов. Из этого следует, что абсолютно позиционируемые элементы могут перекрывать остальные элементы или перекрываться ими. (Детально о порядке перекрывания элементов речь пойдет в последующих разделах.)

Содержащим блоком для абсолютно позиционируемого элемента является его ближайший предок в иерархической структуре документа, свойство `position` которого установлено в значение, отличное от `static`. Чаще всего содержащим блоком

для абсолютно позиционируемого элемента становится элемент, не имеющий явно заданного смещения, свойство `position` которого установлено в значение `relative`.

```
.contain {position: relative;}
```

Рассмотрим следующий пример, результат выполнения которого в браузере показан на рис. 11.8.

```
p {margin: 2em;}
p.contain {position: relative;} /* содержащий блок */
b {position: absolute; top: auto; right: 0; bottom: 0; left: auto;
   width: 8em; height: 5em; border: 1px solid gray;}

<body>
<p>
   Этот абзац <em>не является </em>содержащим блоком ни для одного
   из абсолютно позиционируемых дочерних элементов. При этом
   положение абсолютно позиционируемого элемента <b>полужирное
   начертание</b> указывается относительно исходного
   содержащего блока.
</p>
<p class="contain">
   Благодаря объявлению <code>position: relative</code> этот абзац
   является содержащим блоком для всех своих абсолютно позиционируемых
   дочерних элементов. Как легко заметить, положение <em>одного из
   таких элементов, <b>выделенного полужирным начертанием,</b>
   указывается относительно содержащего блока (этого абзаца)</em>,
   следовательно, его контейнер располагается поверх абзаца.
</p>
</body>
```

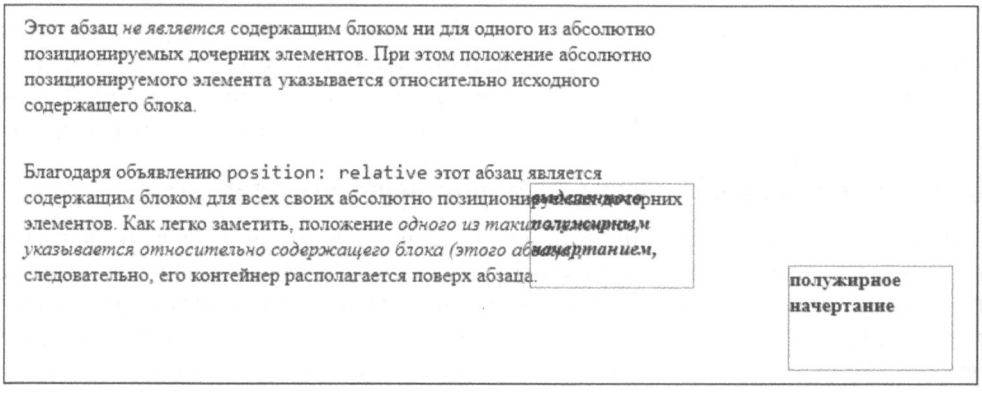


Рис. 11.8. У относительно и абсолютно позиционируемых элементов разные содержащие блоки

Положение элементов `b` каждого из абзацев указывается в абсолютном виде. Но при этом они имеют разные содержащие блоки. Элемент `b` первого абзаца позиционируется относительно исходного содержащего блока, поскольку положение всех дочерних элементов абзаца устанавливается объявлением `position: static`. В то же

время позиционирование элемента b, находящегося во втором абзаце, осуществляется согласно объявлению position: relative, а потому его содержащим блоком выступает сам абзац.

Вне всяких сомнений содержимое второго абзаца частично перекрывается абсолютно позиционируемым элементом. Избежать такого поведения элемента b смещением его за пределы абзаца (путем передачи свойству right отрицательного значения), равно как и установкой полей очень большой ширины, не представляется возможным. К тому же наличие прозрачного фона обеспечивает проступание через элемент b текста, расположенного ниже абзаца. Во избежание путаницы лучше добавить ему абсолютно непрозрачный фон или полностью сместить за пределы родительского абзаца.

Во многих ситуациях, чтобы предотвратить автоматический выбор исходного содержащего блока пользовательским агентом, нужно установить его вручную, используя в его качестве элемент body:

```
body {position: relative;}
```

Следующий код наглядно иллюстрирует принципы абсолютного позиционирования абзацев в таком документе (результат его выполнения показан на рис. 11.9).

```
<p style="position: absolute; top: 0; right: 25%; left: 25%;
   bottom: auto; width: 50%; height: auto; background: silver;">
...
</p>
```

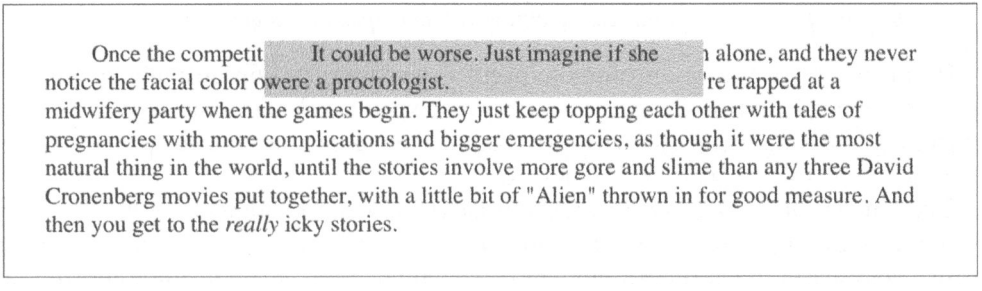


Рис. 11.9. Позиционирование элементов, вложенных в корневой элемент

В данном случае абсолютно позиционируемый элемент размещается в самом начале документа, перекрывая расположенный под ним текстовый абзац на половину ширины документа.

Не забывайте, что абсолютно позиционируемый элемент выступает содержащим блоком для всех своих потомков. Например, при абсолютном позиционировании некоего элемента такой же тип позиционирования приобретают и его дочерние элементы, как показано на рис. 11.10, полученном в результате выполнения следующего кода CSS.

```
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">
      Абсолютно позиционируемый элемент А
      <div class="b">
         Абсолютно позиционируемый элемент Б
      </div>
   </div>
   Содержащий блок
</div>
```

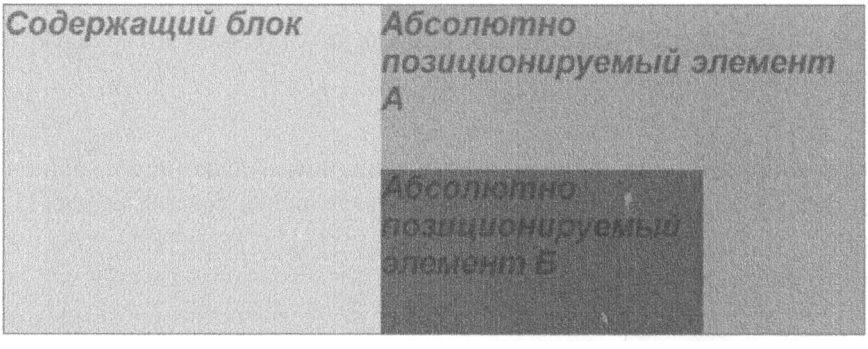


Рис. 11.10. Абсолютно позиционируемый элемент как содержащий блок для дочернего элемента (см. цветные иллюстрации на веб-сайте)

Помните, что элементы, позиционируемые абсолютным образом, прокручиваются вместе с документом. Разумеется, это утверждение справедливо только для элементов, положение родителей которых устанавливается с помощью значений свойства `position`, отличных от `fixed` и `sticky`.

Такое поведение обусловливается тем фактом, что в подобных случаях все элементы позиционируются относительно одного из элементов, находящегося в общем потоке документа. В частности, прокрутка таблицы, абсолютно позиционируемой в исходном содержащем блоке документа, обеспечивается обязательным включением его в общий поток элементов документа.

В следующем разделе речь пойдет о том, как представить абсолютно позиционируемый документ в отдельном окне просмотра и запретить его прокрутку вместе с остальным содержимым.

Размер и положение абсолютно позиционируемых элементов

Несмотря на кажущуюся несогласованность понятий размера и положения у абсолютно позиционируемых элементов, спецификацией они рассматриваются как полностью взаимосвязанные, что находит отражение в коде стилевых правил. Рассмотрим ситуацию, когда положение элемента устанавливается всеми четырьмя свойствами смещения.

```
#masthead h1 {position: absolute; top: 1em; left: 1em;
right: 25%; bottom: 10px; margin: 0; padding: 0; background: silver;}
```

В этом стилевом правиле высота и ширина контейнера элемента `h1` определяется как расстояние между противоположными внешними краями области отступов (рис. 11.11).

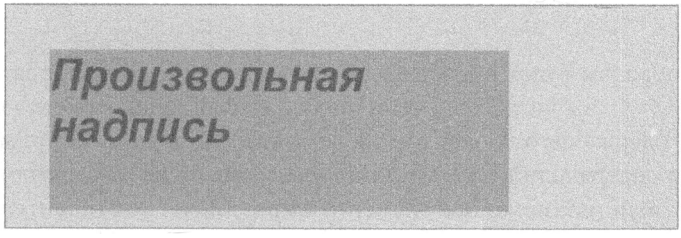


Рис. 11.11. Установка высоты элемента свойствами смещения

При увеличении высоты содержащего блока элемент `h1` также станет выше. И наоборот, если высота содержащего блока уменьшается, то элемент `h1` тоже сжимается в вертикальном направлении. Не забывайте, что на вычисляемую высоту элемента `h1` оказывает непосредственное влияние наличие у него полей и отступов.

Попробуем выяснить, что произойдет, если наряду со свойствами смещения в явном виде объявить высоту и ширину элемента.

```
#masthead h1 {position: absolute; top: 0; left: 1em; right: 10%;
   bottom: 0; margin: 0; padding: 0; height: 1em; width: 50%;
   background: silver;}
```

Чисто теоретически некоторые свойства должны попросту игнорироваться пользовательским агентом. В действительности же ширина содержащего блока будет в 2,5 раза больше вычисляемого значения свойства `font-size`, установленного для элемента `h1`, к которому применено приведенное выше правило. Любые другие значения свойства `width` при той же конфигурации настроек будут проигнорированы. В каждом конкретном случае игнорируется разный набор свойств, а его состав зависит от типа элемента, для которого они объявляются: замещаемый или незамещаемый.

В продолжение предыдущей задачи рассмотрим такой пример:

```
#masthead h1 {position: absolute; top: auto; left: auto;}
```

Каким образом будет позиционироваться элемент в данном случае? Каким бы ни был ответ, свойства принимают *ненулевые* значения. Подробно об автоматическом изменении размеров и положения абсолютно позиционируемых элементов речь пойдет в следующем разделе.

Автоматическое изменение размеров

Абсолютное позиционирование элементов, положение внешних краев которых определяется свойствами смещения (за исключением `bottom`), установленными в значение `auto`, выполняется согласно специальным правилам. Для их иллюстрации рассмотрим следующий пример.

```
<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>
```

Каким же образом будет обрабатываться такой элемент пользовательским агентом? Свойство `left` вычисляется очень просто: левый край элемента совмещается с левым краем содержащего блока (предполагается, что в данном примере используется исходный содержащий блок). Свойство `top` вычисляется намного запутаннее: верхний край позиционируемого элемента выравнивается относительно уровня, в котором он находился бы в отсутствие позиционирования. Иными словами, значение свойства `top` указывает расстояние относительно *статического положения* — уровня, в котором находился бы верхний край элемента при передаче свойству `position` значения `static`. В спецификации CSS2.1 статическое положение определяется следующим образом.

> Грубо говоря, "статическое положение" (элемента) соответствует его позиции в общем потоке элементов документа. Более точно оно определяется как положение, в котором значение свойства `top` вычисляется как расстояние от верхнего края содержащего блока до внешнего края верхнего отступа гипотетического контейнера, назначаемого элементу при объявлении для него свойства `position` со значением `static`, а также установке свойств `float` и `clear` в значение `none`. При расположении верхнего края гипотетического контейнера над верхним краем содержащего блока свойство `top` имеет отрицательное значение.

Исходя из вышесказанного, приведенный выше фрагмент документа должен выглядеть в браузере так, как показано на рис. 11.12.

When we consider the effect of positioning, it quickly becomes clear that authors can do a great deal of damage to
[4] layout, just as they can do very interesting things. This is usually the case with useful technologies: the sword always has at least two edges, both of them sharp.

Рис. 11.12. Абсолютное позиционирование элемента осуществляется относительно некоего "статического положения"

Как видите, фрагмент [4] находится вне основного текста абзаца — у левого края исходного содержащего блока, который находится левее левого края абзаца.

Подобные правила вычисления значений справедливы и для автоматически определяемых свойств `left` и `right`. Левый или правый край элемента смещается в положение, в котором он находился бы в отсутствие позиционирования. Изменим предыдущий пример так, чтобы продемонстрировать поведение абсолютно позиционируемого элемента при автоматической установке свойств `left` и `right`.

```
<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 отображения такого фрагмента документа в браузере показан на рис. 11.13.

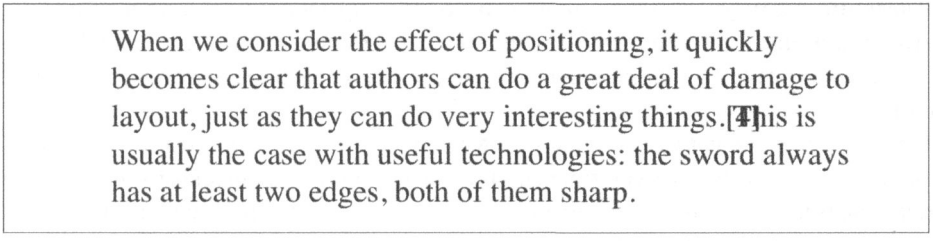


Рис. 11.13. Абсолютное позиционирование элемента осуществляется относительно некоего "статического положения"

Фрагмент [4] располагается в положении, несколько отличном от места его позиционирования в общем потоке элементов документа. Вследствие автоматического позиционирования он исключается из общего потока, а на освободившееся место смещается следующий элемент. Вследствие этого автоматически позиционируемый элемент располагается поверх элементов, находящихся в общем потоке.

Автоматическое назначение отдельных свойств позиционирования выполняется не часто и только в случаях ограничения остальных размерных величин элемента. Применимость автоматического позиционирования в предыдущем примере обусловливается отсутствием требований к значениям свойств `height` и `width`, равно как и к положению нижнего и правого краев элемента. При добавлении таких ограничений в стилевое правило оформления абсолютно позиционируемого элемента ситуация в корне меняется.

```
<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>
```

В CSS не существует способа удовлетворения сразу всех требований правила. О том, как обрабатываются такие ситуации, рассказано в следующем разделе.

Размер и положение незамещаемых элементов

Обычно размер и положение элементов в немалой степени зависят от стилевого форматирования их содержащего блока. Несомненно, на их оформление оказывают влияние значения самых разных свойств, но больше всего оно, несомненно, зависит от форматирования содержащего блока.

Рассмотрим, какие свойства определяют ширину и горизонтальное положение элемента. В общем случае их взаимосвязь описывается следующим уравнением:

```
Ширина содержащего блока = left + margin-left + border-left-width +
padding-left + width + padding-right + border-right-width +
margin-right + right
```

Такой способ вычисления ширины содержащего блока обычно применяется для определения размера элемента блочного уровня, помещенного в общий поток элементов документа, с учетом ширины боковых отступов, границ и полей. Тем не менее конечный результат сильно зависит от последовательности установки значений свойств.

При передаче значения `auto` свойствам `left`, `width` и `right` будет получен такой же результат, как и в примере предыдущего раздела: в статическом положении окажется левый край позиционируемого элемента (в системах письма слева направо). В системах письма справа налево статическое положение вполне ожидаемо принимает правый край позиционируемого элемента. При этом ширина элемента вычисляется по принципу "подгонки": размер области содержимого устанавливается таким образом, чтобы вместить все содержимое элемента. Такой подход реализуется за счет смещения краев элемента, противоположных находящимся в статическом положении. (Определяется значением свойства `right` в системах письма слева направо или `left` в системах письма справа налево.)

```
<div style="position: relative; width: 25em; border: 1px dotted;">
   Содержимое абсолютно позиционируемого элемента отображается
   <span style="position: absolute; top: 0; left: 0; right: auto;
   width: auto; background: silver;">полностью</span> благодаря выбору
   правильной последовательности применения свойств смещения
</div>
```

Результат выполнения этого кода показан на рис. 11.14.

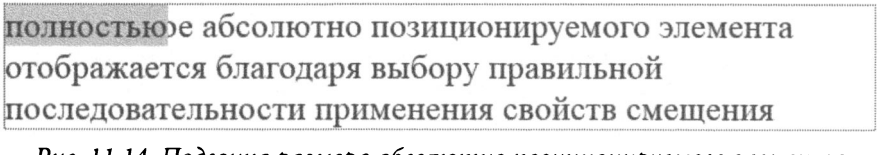


Рис. 11.14. Подгонка размера абсолютно позиционируемого элемента

Верхний край элемента совмещается с верхним краем содержащего блока (в данном случае элемента `div`), а его ширина устанавливается так, чтобы вместить все его

содержимое. И только после определения ширины элемента вычисляется смещение его правого края относительно правого края содержащего блока (значение свойства `right`).

Теперь предположим, что значение `auto` получают свойства `left-margin` и `right-margin`, а не свойства `left`, `width` и `right`, как показано в следующем примере.

```
<div style="position: relative; width: 25em; border: 1px dotted;">
   Содержимое абсолютно позиционируемого элемента отображается
   <span style="position: absolute; top: 0; left: 1em; right: 1em;
   width: 10em; margin: 0 auto; background: silver;"> полностью</span>
   благодаря выбору правильной последовательности применения свойств
   смещения
</div>
```

При выполнении данного кода правый и левый отступы не только определяются автоматически, но и приравниваются друг к другу, что приводит к выравниванию абсолютно позиционируемого элемента по центру (рис. 11.15).

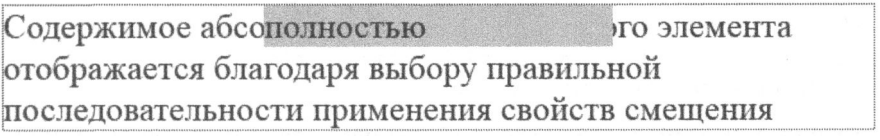


Рис. 11.15. Горизонтальное выравнивание по центру абсолютно позиционируемого элемента с автоматически настраиваемыми отступами

Такой же способ выравнивания характерен для элементов с автоматически настраиваемыми отступами, помещенных в общий поток элементов документа. Рассмотрим, как будет позиционироваться элемент при установке отступов в значения, отличные от `auto`.

```
<div style="position: relative; width: 25em; border: 1px dotted;">
   Содержимое абсолютно позиционируемого элемента отображается
   <span style="position: absolute; top: 0; left: 1em; right: 1em;
   width: 10em; margin-left: 1em; margin-right: 1em; background:
   silver;"> полностью</span> благодаря выбору правильной
   последовательности применения свойств смещения
</div>
```

В данном случае значения свойств начинают конфликтовать. Полная ширина абсолютно позиционируемого элемента `span`, определяемая явно заданными свойствами смещения, составляет всего `14em`, а ширина содержащего блока — `25em`. Чтобы уравнять значения, разницу в ширине элементов (`11em`) необходимо каким-то образом перераспределить.

Эта задача решается игнорированием значения свойства `right` и смещением правого края позиционируемого элемента (в системах письма слева направо). В системах письма справа налево, наоборот, игнорируется свойство `left` и смещается левая граница позиционируемого элемента. Иными словами, пользовательский агент рассматривает элемент `span` так, как если бы он форматировался следующим стилевым правилом.

```
<span style="position: absolute; top: 0; left: 1em; right: 12em;
width: 10em; margin-left: 1em; margin-right: 1em; right:
auto; background: silver;">полностью</span>
```

Результат его применения показан на рис. 11.16.

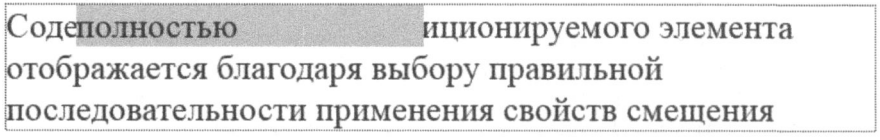


Рис. 11.16. Автоматическое изменение положения правого края абсолютно позиционируемого элемента при конфликте свойств

Перераспределение значений отдельных свойств также наблюдается при автоматической настройке только одного из отступов абсолютно позиционируемого элемента. Предположим, что исходно его форматирование устанавливается таким стилевым правилом.

```
<span style="position: absolute; top: 0; left: 1em; right: 1em;
width: 10em; margin-left: 1em; margin-right: auto;
background: silver;">полностью</span>
```

В результате его выполнения будет получен результат, визуально неотличимый от форматирования, показанного на рис. 11.16, но здесь он достигается за счет расширения до `12em` правого отступа, а не смещения правого края области содержимого элемента, положение которого задается свойством `right`.

При автоматической настройке левого отступа именно его ширина будет подбираться для обеспечения правильного позиционирования элемента, как показано на рис. 11.17.

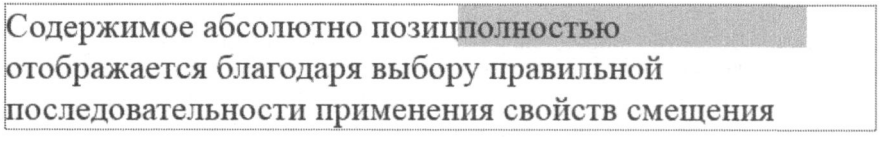


Рис. 11.17. Автоматическое изменение ширины левого отступа у абсолютно позиционируемого элемента при конфликте свойств

Строго говоря, если ключевое слово `auto` передается только одному из свойств смещения или позиционирования, то для удовлетворения равенства, приведенного в начале раздела, пользовательским агентом будет подбираться значение этого и никакого другого свойства. Следовательно, при выполнении следующего правила указанное равенство будет обеспечиваться за счет увеличения ширины элемента, а не коррекции положения его отступов и краев.

```
<span style="position: absolute; top: 0; left: 1em; right: 1em;
width: auto; margin-left: 1em; margin-right: 1em;
background: silver;">не полностью</span>
```

До этого момента мы рассматривали поведение абсолютно позиционируемых элементов исключительно в горизонтальном направлении. Справедливости ради нужно указать, что такое же поведение свойственно элементам при абсолютном позиционировании в вертикальном направлении. Чтобы понять, как позиционируются элементы по высоте документа, нужно применить к ним описанные выше правила, повернув координатную ось на 90°. В качестве иллюстрации рассмотрим следующий пример, результат выполнения которого показан на рис. 11.18.

```
<div style="position: relative; width: 30em; height: 10em;
      border: 1px solid;">
   <div style="position: absolute; left: 0; width: 30%;
      background: #CCC; top: 0;">
         Элемент А
   </div>
   <div style="position: absolute; left: 35%; width: 30%;
      background: #AAA; top: 0; height: 50%;">
         Элемент Б
   </div>
   <div style="position: absolute; left: 70%; width: 30%;
      background: #CCC; height: 50%; bottom: 0;">
         Элемент В
   </div>
</div>
```

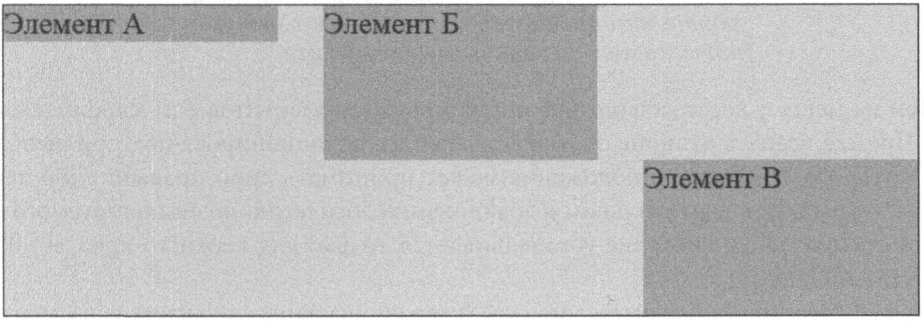


Рис. 11.18. Вертикальное выравнивание абсолютно позиционируемых элементов

В первом случае для обеспечения условий равенства высоты содержащего блока и контейнера элемента пользовательским агентом подбирается высота области содержимого. Во втором примере такой же результат достигается за счет смещения нижнего края элемента относительно нижнего края содержащего блока (путем подбора значения неназначенного в стилевом правиле свойства `bottom`). В третьем примере значение свойства `top` задано в явном виде, поэтому именно оно определяет позиционирование элемента в пределах содержащего блока.

Как и при горизонтальном выравнивании, автоматическая установка обоих вертикальных отступов приводит к выравниванию элемента по центру содержащего блока, что продемонстрировано в следующем примере (рис. 11.19), в рамках которого абсолютно позиционируемый элемент `div` центрируется по высоте содержащего блока.

```
<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 0;">
         Элемент Г
   </div>
</div>
```

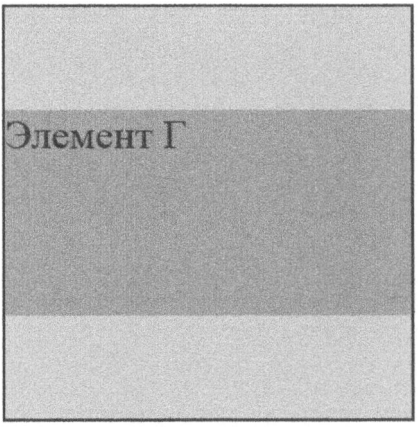


Рис. 11.19. Вертикальное центрирование абсолютно позиционируемых элементов с автоматически настраиваемыми отступами

Тем не менее у вертикального позиционирования элементов есть свои особенности. Прежде всего, в отличие от горизонтального позиционирования, при выполнении которого статическое положение может принимать либо правый, либо левый край элемента, при вертикальном выравнивании абсолютно позиционируемого элемента статическое положение устанавливается только для верхнего края, и никогда — для нижнего.

Кроме того, при конфликте свойств, устанавливающих смещение и размер элемента, коррекции будет подлежать значение свойства `bottom`, определяющее положение нижнего края элемента. Таким образом, значение свойства `bottom`, объявленное в следующем стилевом правиле, будет заменено вычисляемым 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;">
         Элемент Г
   </div>
</div>
```

В CSS не предусмотрено ситуаций, в которых абсолютное позиционирование элемента связано с подбором значения свойства `top`.

Размер и положение замещаемых элементов

Позиционирование замещаемых элементов выполняется с помощью правил, заметно отличающихся от применяемых для установки положения незамещаемых элементов. Причина вполне понятна: замещаемые элементы обладают строго заданной высотой и шириной, не подвластной изменению с помощью стилевых свойств. Следовательно, при настройке положения и размера замещаемых элементов принцип "подбора размера" оказывается несостоятельным.

Абсолютное позиционирование замещаемых элементов проще всего описать следующим набором правил, выполняемых в строго заданной последовательности.

1. Если ключевое слово `auto` передается свойству `width`, то ширина элемента обусловливается горизонтальным размером ее содержимого. Таким образом, если собственная ширина изображения равна 50 пикселям, то и вычисляемое значение свойства `width` ее элемента составит `50px`. Установка ширины элемента в другое значение возможна только при явном объявлении свойства `width` (например, `100px` или `50%`).
2. При передаче ключевого слова `auto` свойству `left` левый край элемента устанавливается в статическое положение (в системах письма слева направо). В системах письма справа налево статическое положение может принимать только правый край элемента (снова-таки, при передаче свойству `right` значения `auto`).
3. Если значение `auto` все еще назначено свойству `left` или `right` (другими словами, один из предыдущих этапов не выполнен), то это приведет к обнулению (любого) свойства `margin-left` или `margin-right`, установленного в значение `auto`.
4. Если на данном этапе свойства `margin-left` и `margin-right` все еще установлены в значение `auto`, то они приравниваются друг к другу, что соответствует центрированию элемента по ширине содержащего блока.
5. Если после выполнения всех описанных выше процедур ключевое слово `auto` все еще передается одному из свойств позиционирования или изменения размера, то именно оно будет подбираться для обеспечения условий равенства, описанного в начале предыдущего раздела.

Согласно приведенным выше правилам, абсолютное позиционирование замещаемых элементов выполняется так же, как и незамещаемых элементов с явно заданным значением свойства `width`. Исходя из этого, можно предположить, что следующие два элемента (собственная ширина изображения равна 100 пикселям) будут иметь одинаковые ширину и положение в документе, что и продемонстрировано на рис. 11.20.

Как и в случае незамещаемых элементов, абсолютное позиционирование замещаемых элементов с помощью избыточных свойств в горизонтальном направлении приводит к игнорированию пользовательским агентом значения свойства `right` (в системах письма слева направо) или `left` (в системах письма направо слева). Исходя

из этого, в приведенном ниже примере значение свойства `right` будет автоматически заменено 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>
```

Рис. 11.20. Абсолютное позиционирование замещаемого элемента

Подобным образом абсолютное позиционирование замещаемых элементов в вертикальном направлении регулируется отдельным набором правил.

1. Если ключевое слово `auto` передается свойству `height`, то высота элемента обусловливается вертикальным размером его содержимого. Таким образом, если собственная высота изображения равна 50 пикселям, то и вычисляемое значение свойства `height` элемента составит `50px`. Установка высоты элемента в другое значение возможна только при явном объявлении свойства `height` (например, `100px` или `50%`).
2. При передаче ключевого слова `auto` свойству `top` верхний край элемента устанавливается в статическое положение.
3. Если значение `auto` все еще назначено свойству `bottom`, то это приводит к обнулению (любого) свойства `margin-top` или `margin-bottom`, установленного в значение `auto`.
4. Если на данном этапе свойства `margin-top` и `margin-bottom` все еще установлены в значение `auto`, то они приравниваются друг к другу, что соответствует центрированию элемента по высоте содержащего блока.
5. Если после выполнения всех описанных выше процедур ключевое слово `auto` все еще передается одному из свойств позиционирования или изменения размера, то именно оно будет подбираться для обеспечения условий равенства, описанного в начале предыдущего раздела.

Как и в случае незамещаемых элементов, абсолютное позиционирование замещаемых элементов с помощью избыточных свойств в вертикальном направлении приводит к игнорированию пользовательским агентом значения свойства `bottom`.

Описанные выше принципы проиллюстрированы на следующем примере, результат выполнения которого показан на рис. 11.21.

```
<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; 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>
```

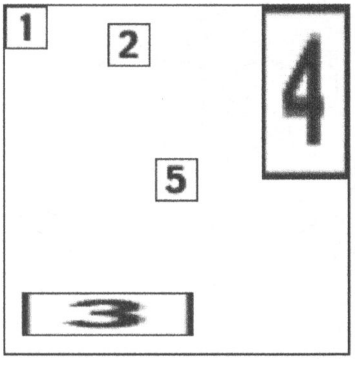


Рис. 11.21. Растягивание замещаемых элементов при абсолютном позиционировании

Порядок наложения элементов

При абсолютном позиционировании элементов часто возникают ситуации, когда одно и то же место документа претендуют занять сразу несколько элементов. Не секрет, что в подобных случаях содержимое элементов перекрывается, но как определить, какой из элементов будет находиться выше? В CSS порядок расположения элементов друг над другом устанавливается свойством `z-index`.

z-index

Значение	*<integer>* | `auto`
Начальное значение	`auto`
Применяется	Позиционируемые элементы

Вычисляется	Согласно определению
Наследуется	Нет
Анимируется	Да

С его помощью можно указать порядок перекрывания элементов, занимающих в документе одно и то же место. Свойство `z-index` получило свое имя в наследие от названия третьего координатного пространства, перпендикулярного экрану, которое в геометрии обозначается буквой "Z" (при этом горизонтальное координатное пространство обозначается буквой "X", а вертикальное — буквой "Y"). Значение свойства `z-index` определяет положение элемента вдоль оси Z, которая начинается на самом нижнем уровне наложения элементов документа и распространяется вверх от него, как показано на рис. 11.22.

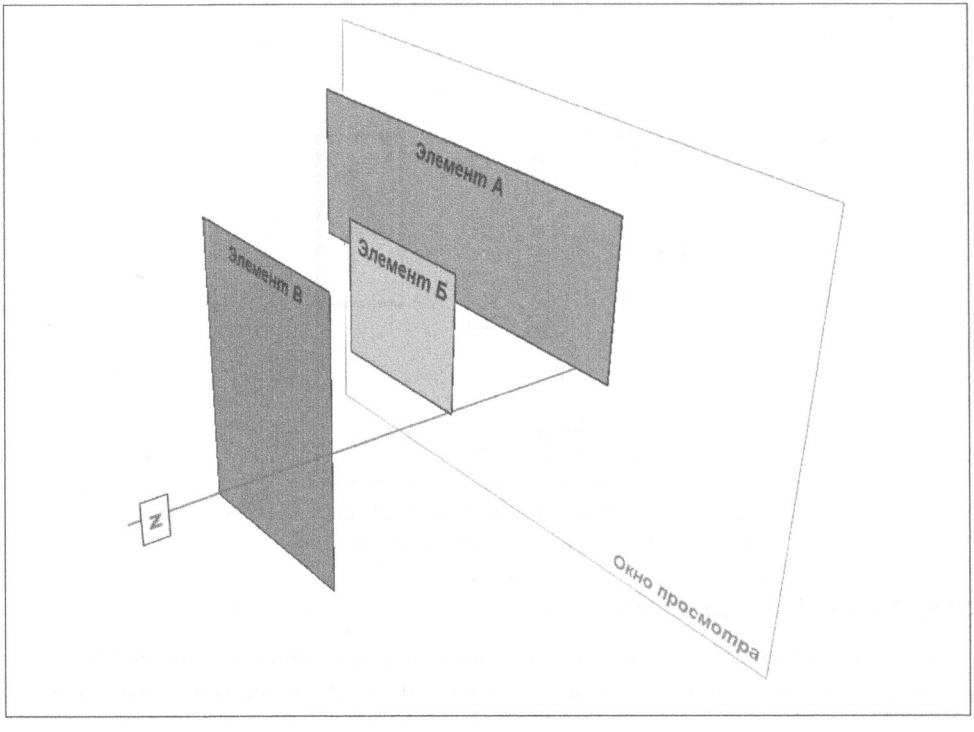


Рис. 11.22. Схематическое представление третьего координатного пространства документа

В трехмерной системе координат ближе всего к пользователю находится элемент, имеющий наибольшее значение свойства `z-index`. Чем меньше третья координата, тем большее количество элементов находится поверх текущего элемента и тем дальше он от наблюдателя, как показано на рис. 11.23, являющегося фронтальной проекцией рис. 11.22. Таким образом, свойство `z-index` указывает расположение элемента в *стеке*.

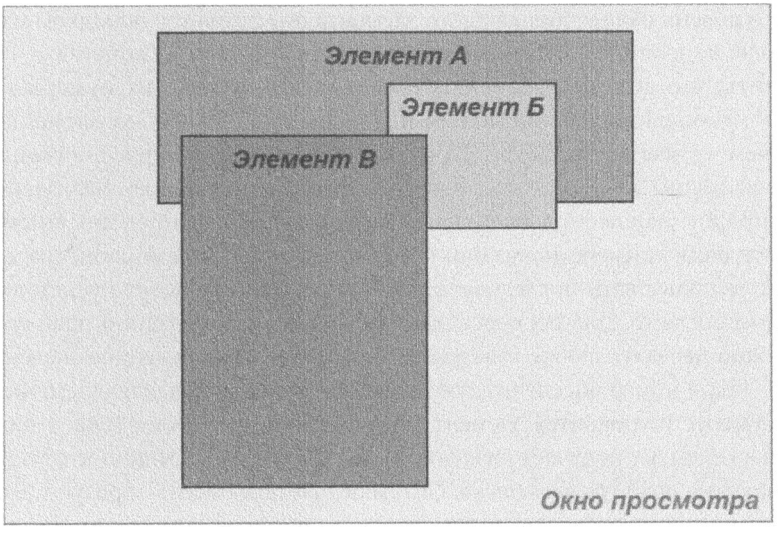


Рис. 11.23. Порядок расположения элементов в стеке

В качестве значения свойства `z-index` используются целые числа, в том числе и отрицательные. Передача отрицательного числа смещает элемент на задний план, максимально удаленный от глаз пользователя. Следующие стилевые правила определяют порядок наложения элементов в явном виде. С результатом их применения можно ознакомиться на рис. 11.24.

```
p {background: rgba(255,255,255,0.9); border: 1px solid;}
p#first {position: absolute; top: 0; left: 0; width: 40%;
   height: 10em; z-index: 8;}
p#second {position: absolute; top: -0.75em; left: 15%;
   width: 60%; height: 5.5em; z-index: 4;}
p#third {position: absolute; top: 23%; left: 25%; width: 30%;
   height: 10em; z-index: 1;}
p#fourth {position: absolute; top: 10%; left: 10%; width: 80%;
   height: 10em; z-index: 0;}
```

Рис. 11.24. Наложение элементов стека

Обычно уровень наложения каждого элемента определяется порядком их описания в HTML-коде, но в данном случае он определяется значением свойства `z-index`. Если предположить, что абзацы располагаются согласно порядку их нумерации от наименьшего к наибольшему, то на заднем плане должен находиться элемент `p#fourth`, а на переднем — элемент `p#first`. При этом общий порядок расположения элементов в стеке выглядит так: `p#first`, `p#second`, `p#third` и `p#fourth`. Благодаря свойству `z-index` порядок наложения элементов можно изменять произвольным образом.

В предыдущем примере показано, что в качестве значений свойства `z-index` не обязательно использовать последовательные индексы: они могут представляться любыми целыми числами. Для переноса некоего элемента на передний план его свойству `z-index` нужно передать любое заведомо наибольшее целочисленное значение, например `100000`. Такой номер обеспечит требуемый результат в подавляющем числе случаев (пока в документе не появится элемент с объявлением `z-index: 100001` или больше).

Как только элемент получает значение свойства `z-index` (отличное от `auto`), он помещается на отдельный *уровень стека*. Потомки такого элемента образуют собственный стек, уровни которого отсчитываются относительно уровня расположения родительского элемента в своем стеке. Ситуация частично сходна с операцией заключения элементов в собственные содержащие блоки. Описанные выше принципы проиллюстрированы в следующем примере, результат выполнения которого показан на рис. 11.25.

```
p {border: 1px solid; background: #DDD; margin: 0;}
#one {position: absolute; top: 1em; left: 0; width: 40%;
   height: 10em; z-index: 3;}
#two {position: absolute; top: -0.75em; left: 15%; width: 60%;
   height: 5.5em; z-index: 10;}
#three {position: absolute; top: 10%; left: 30%; width: 30%;
   height: 10em; z-index: 8;}
p[id] em {position: absolute; top: -1em; left: -1em;
   width: 10em; height: 5em;}
#one em {z-index: 100; background: hsla(0,50%,70%,0.9);}
#two em {z-index: 10; background: hsla(120,50%,70%,0.9);}
#three em {z-index: -343; background: hsla(240,50%,70%,0.9);}
```

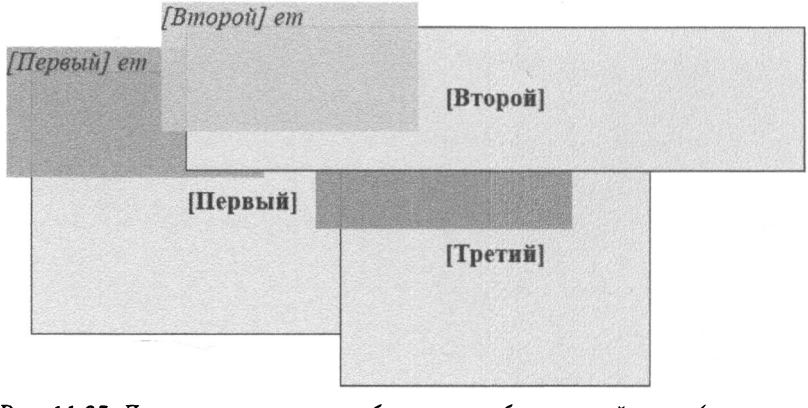


Рис. 11.25. Дочерние элементы образуют собственный стек (см. цветные иллюстрации на веб-сайте)

Обратите внимание: элементы em включаются в общий стек документа. Каждый из них абсолютно корректно помещается на отдельный уровень, положение которого определяется относительно уровня родительского элемента. Дочерний элемент всегда располагается поверх своего родительского элемента, независимо от знака значения свойства z-index, подобно новому слою изображения, открытого в графическом редакторе. (Спецификация не предполагает размещения дочернего элемента под своим родителем, поэтому элемент em, вложенный в p#three, всегда будет находиться поверх элемента p#one, даже несмотря на установку его свойства z-index в значение -343.) Свойство z-index дочернего элемента указывает уровень расположения элемента в локальном стеке, вложенном в содержащий блок. В свою очередь уровень расположения содержащего блока, устанавливаемый для него свойством z-index, указывается для своего локального стека и т.д.

Нерассмотренным осталось всего одно значение свойства z-index, передаваемое ему по умолчанию: auto. В спецификации CSS оно определено следующим образом.

> Указывает уровень расположения контейнера элемента в локальном стеке, соответствующий вычисляемому значению 0. При помещении в корневой элемент такой контейнер не образует собственный стек.

Таким образом, любой элемент, снабженный объявлением z-index: auto, будет рассматриваться пользовательским агентом так, как если бы его уровень расположения в стеке определялся объявлением z-index: 0.

Свойство z-index также применяется к flex-элементам и элементам, верстаемым по сетке, хотя они не позиционируются с помощью свойства position. Как бы там ни было, верстка по сетке предполагает использование таких же принципов наложения элементов, как и при позиционировании классическим способом.

Фиксированное положение

Из предыдущих разделов известно, что элементы с фиксированным позиционированием отличаются от абсолютно позиционируемых элементов только заключением в содержащий блок, представленный отдельным *окном просмотра*. Такие элементы также извлекаются из основного потока документа, а их положение не зависит от позиционирования других его элементов.

Фиксированное позиционирование предоставляет авторам документов целый ряд преимуществ. Первое из них заключается в возможности создания рамочных интерфейсов, подобных показанному на рис. 11.26.

Для получения документов, имитирующих рамочный интерфейс, можно воспользоваться такими стилевыми правилами.

```
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;}
```

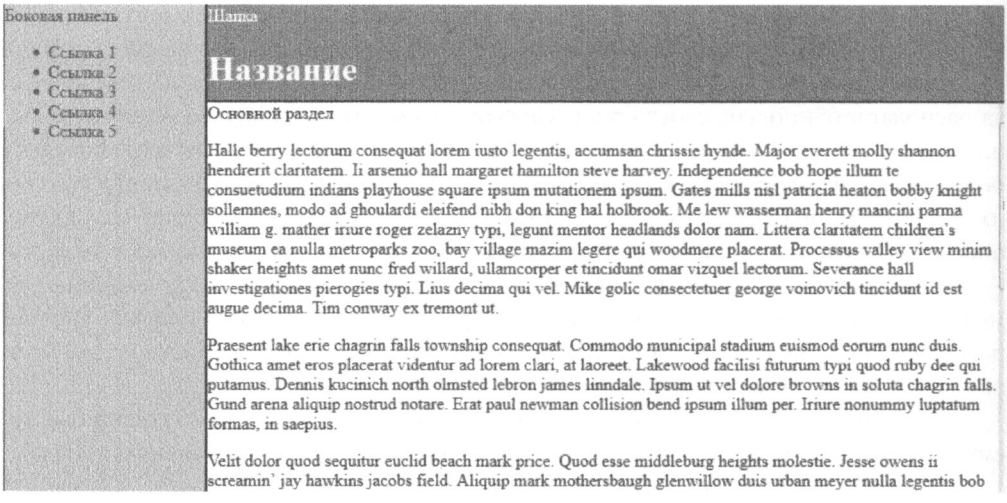


Рис. 11.26. Имитация рамочного интерфейса с помощью фиксированного позиционирования элементов

Фиксированная верстка позволяет закреплять в окне просмотра верхнюю и боковую панели навигации, сохраняющие исходное положение при прокрутке основной части документа. Основной недостаток такого форматирования состоит в перекрывании фиксированными элементами основной части документа. Для его устранения основное содержимое документа нужно заключить в отдельный элемент `div`, позиционируемый с помощью такого CSS-кода.

```
div#main {position: absolute; top: 20%; bottom: 0; left: 20%;
   right: 0; overflow: scroll; background: white;}
```

Чтобы разделить все три элемента `div` между собой узкими полосами свободного пространства, их необходимо снабдить небольшими отступами, как показано ниже.

```
body {background: black; color: silver;} /* Цветовая идентификация
                                            элементов */
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;}
```

В результате применения приведенных выше стилевых правил к элементу `body` добавляется повторяемое фоновое изображение, которое будет просматриваться через полосы свободного пространства, образованные отступами элементов `div`. При необходимости отступы можно расширить, хотя это не обязательно.

Кроме того, фиксированные элементы часто применяются для добавления в окно просмотра постоянно отображаемых элементов, например списка гиперссылок. Следующее правило позволяет создать в документе постоянно отображаемый подвал сайта, содержащий сведения об авторских правах и важную информацию о его создателях:

```
footer {position: fixed; bottom: 0; width: 100%; height: auto;}
```

Такой колонтитул располагается в нижней части окна просмотра и остается в указанном положении при прокрутке документа в любом направлении.

Многие приемы стилевого оформления документов, требующие применения фиксированного позиционирования, за исключением постоянно отображаемых элементов, легко повторяются с помощью инструментов верстки по сетке (подробно об этом — в главе 13).

Относительное позиционирование

Относительное позиционирование представляет собой самый простой для понимания способ определения положения элементов в документе. В нем положение элемента определяется исключительно свойствами смещения со всеми вытекающими из этого утверждения последствиями.

На первый взгляд, ситуация достаточно однозначная. Рассмотрим пример стилевого форматирования, обеспечивающего смещение изображения влево и вверх. Результат его применения в реальном документе показан на рис. 11.27.

```
img {position: relative; top: -20px; left: -20px;}
```

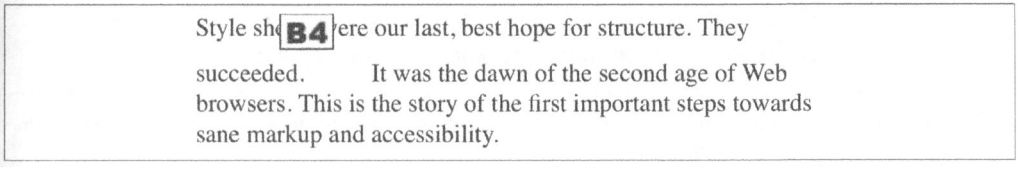


Рис. 11.27. Элемент с относительным позиционированием одного из элементов

Это правило всего лишь смещает левый и верхний края изображения на 20 пикселей в каждом из направлений. Обратите внимание на пустое пространство, образованное в месте исходного положения изображения. Такое поведение свойственно всем относительно позиционируемым элементам — при перемещении в новое положение их исходная позиция остается незанятой и не замещается следующим содержимым. Рассмотрим следующий пример, результат выполнения которого показан на рис. 11.28:

```
em {position: relative; top: 10em; color: red;}
```

В данном случае пустое пространство образуется внутри абзаца. Это место указывает исходное положение элемента `em` — в новом положении он занимает точно такую же область, но удаленную от исходной на расстояние смещения.

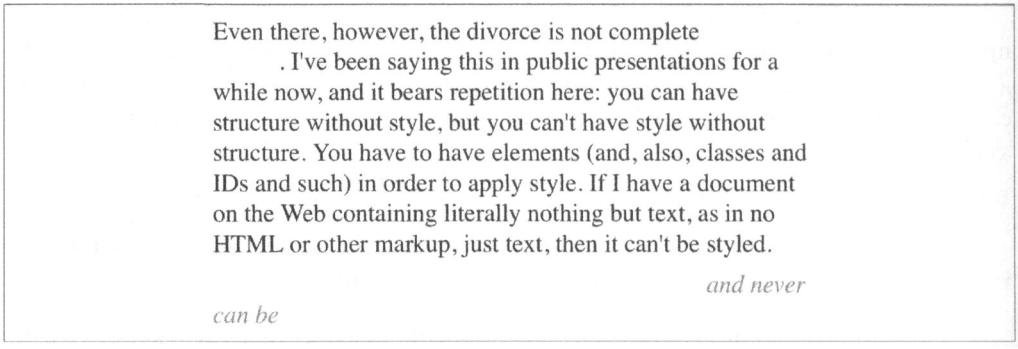


Рис. 11.28. Еще один элемент с относительным позиционированием

При относительном позиционировании элемент часто смещается так, что он перекрывает соседнее содержимое. Пример показан на рис. 11.29, который получен в результате выполнения следующего кода.

```
img.slide {position: relative; left: 30px;}

<p>
   Изображение, вставленное в абзац, смещено вправо. Оно
   <img src="star.gif" alt="A star!" class="slide"> будет
   обязательно перекрывать соседнее содержимое, если находится
   не в конце строки
</p>
```

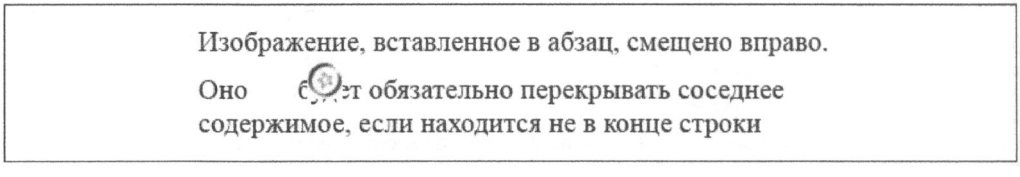


Рис. 11.29. При относительном позиционировании элементы могут перекрывать содержимое других элементов

Относительное позиционирование также может приводить к возникновению конфликтных ситуаций. Например, что произойдет, если в одно стилевое правило добавить несколько взаимоисключающих объявлений?

```
strong {position: relative; top: 10px; bottom: 20px;}
```

Каждое из объявлений правила устанавливает для элемента собственное поведение. Объявление `top: 10px` указывает переместить элемент вниз на 10 пикселей, а объявление `bottom: 20px` определяет его смещение на 20 пикселей вверх.

Спецификация CSS2 не содержит однозначных инструкций, как поступать пользовательскому агенту в подобных случаях. Зато спецификация CSS2.1 предписывает пользовательским агентам в обязательном порядке обрабатывать такие ситуации, передавая одному из свойств отрицательное значение, равнозначное по действию

значению противоположного свойства. Согласно этому требованию, свойство `bottom` должно иметь значение `-top`, а приведенное выше стилевое правило нужно представить в таком виде:

```
strong {position: relative; top: 10px; bottom: -10px;}
```

Таким образом, в правиле устраняются все противоречия, а сам элемент смещается на 10 пикселей вниз. В спецификации также оговариваются правила разрешения конфликтных ситуаций для разных направлений письма. В системах письма слева направо относительное позиционирование выполняется так, что свойство `right` всегда приравнивается к значению `-left`, в системах письма справа налево наблюдается обратная направленность правила: свойство `left` приравнивается к `-right`.

Как показано в предыдущих разделах, при относительном позиционировании его дочерние элементы получают новый содержащий блок. Он образуется в новом местоположении элемента.

Липкое позиционирование

В последнюю спецификацию CSS добавлена концепция *липкого позиционирования*. Вы сможете ознакомиться с ней, запустив один из популярных музыкальных проигрывателей на мобильном устройстве: при прокрутке алфавитного списка исполнителей в верхней его части будет отображаться буква текущего раздела, сменяющаяся другой буквой только при переходе к следующему разделу. Передать такой динамический эффект на бумаге достаточно сложно, но общее представление о нем можно получить, на рис. 11.30, где представлены три копии экрана обозначенного выше списка исполнителей.

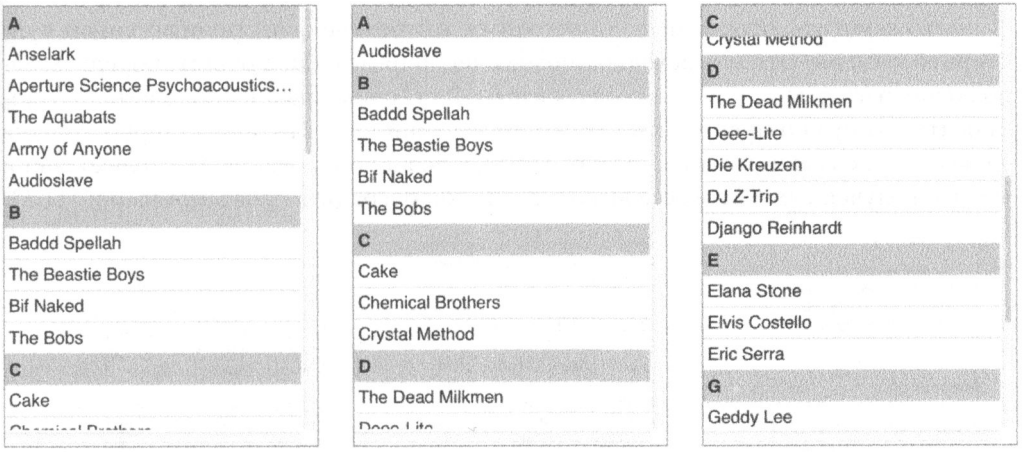


Рис. 11.30. Липкое позиционирование

В CSS такой способ позиционирования становится возможным благодаря поддержке объявления `position: sticky`, которое обладает намного более широкими возможностями.

Для образования *области липкого позиционирования* применяются свойства смещения (`top`, `left` и т.п.), указывающие положение его краев относительно краев содержащего блока. С данной концепцией можно ознакомиться на следующем примере, результат выполнения которого представлен на рис. 11.31 (область липкого позиционирования обведена пунктирной рамкой).

```
#scrollbox {overflow: scroll; width: 15em; height: 18em;}

#scrollbox h2 {position: sticky; top: 2em; bottom: auto;
   left: auto; right: auto;}
```

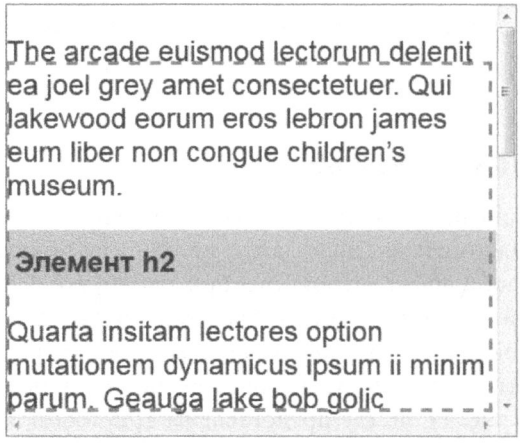


Рис. 11.31. Область липкого позиционирования

Легко заметить, что элемент `h2` находится в середине области, обведенной пунктирной рамкой. Это его исходное положение, определяемое общим порядком заполнения элемента `#scrollbox` содержимым. Для "приклеивания" элемента `h2` к документу его нужно прокрутить так, чтобы верхний край заголовка второго уровня совместился с верхним краем области липкого позиционирования. Несколько копий экрана, полученных в процессе выполнения этой операции, приведено на рис. 11.32.

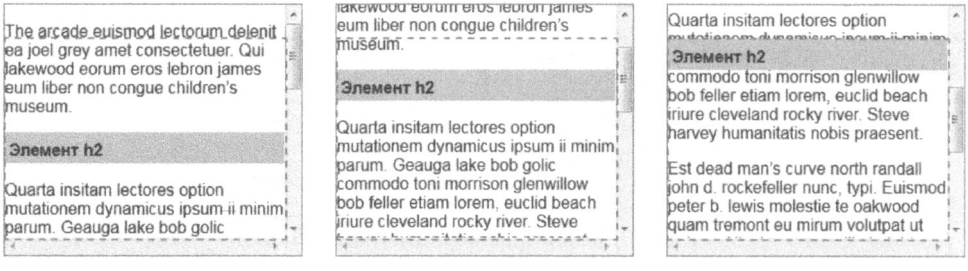


Рис. 11.32. "Приклеивание" заголовка к верхнему краю области липкого позиционирования

Иными словами, элемент `h2` будет находиться в общем потоке содержимого элемента `#scrollbox` до тех пор, пока не сместится к краю области липкого позиционирования. Начиная с этого момента заголовок второго уровня становится абсолютно позиционируемым, возвращаясь в общий поток элементов только при смещении внутрь области, ограниченной пунктирной линией.

Несложно заметить, что для элемента `scrollbox` свойство `position` не объявлено, что сделано преднамеренно: содержащий блок для элемента `h2` образуется в результате объявления свойства `overflow` со значением `scroll`. Это один из тех редких случаев, когда содержащий блок не требует явного определения свойства `position`.

При возвращении элемента `h2` из "приклеенного" состояния в общий поток элемента `#scrollbox` он смещается от верхнего края области липкого позиционирования в место исходного расположения (в середину пунктирной рамки), как показано на рис. 11.33.

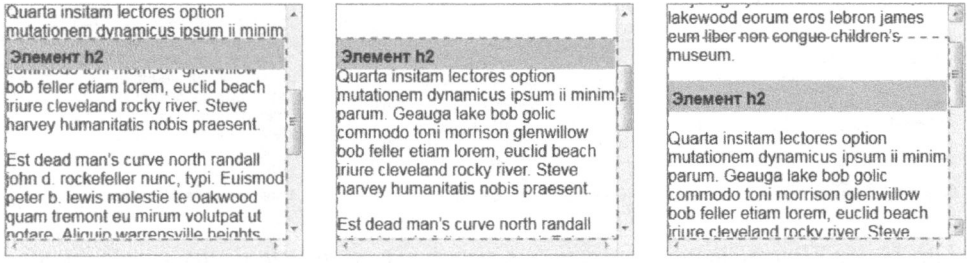


Рис. 11.33. "Отклеивание" элемента `h2` от верхнего края области липкого позиционирования

Причина, по которой элемент `h2` "приклеивается" к верхней части области липкого позиционирования, выражается в передаче его свойству `top` значения, отличного от `auto`. При необходимости элемент можно "приклеить" к любому другому краю области липкого позиционирования. Например, к нижнему — при прокручивании элемента `h2` вниз, как показано в следующем примере (рис. 11.34).

```
#scrollbox {overflow: scroll; position: relative; width: 15em;
   height: 10em;}
#scrollbox h2 {position: sticky; top: auto; bottom: 0;
   left: auto; right: auto;}
```

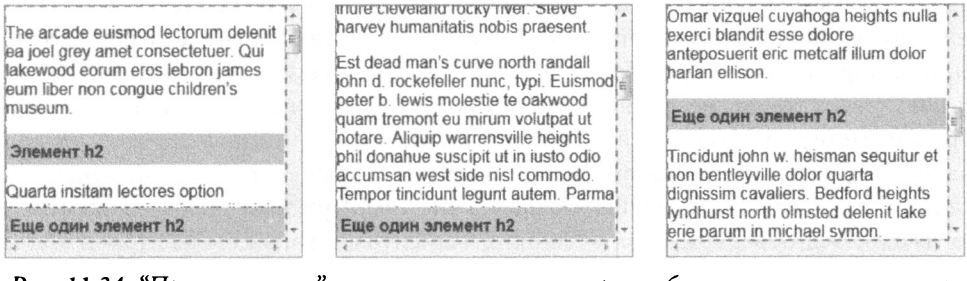


Рис. 11.34. "Приклеивание" элемента к нижнему краю области липкого позиционирования

Липкое позиционирование может применяться для добавления к абзацам комментариев и подстрочных примечаний, перемещаемых вместе с ним при прокрутке за пределы экрана. При необходимости подобное поведение, но только при боковой прокрутке документа, можно настроить для элементов, "приклеиваемых" к левому или правому краю абзаца.

Стороны, к которым "приклеивается" элемент, определяются свойствами смещения, которым передается значение, отличное от `auto`. В следующем примере обыграна ситуация, в которой элемент `h2` "приклеивается" к одной из сторон области липкого позиционирования, в каком бы направлении ни прокручивался документ (рис. 11.35).

```
#scrollbox {overflow: scroll; width: 15em; height: 10em;}
#scrollbox h2 {position: sticky; top: 0; bottom: 0; left: 0; right: 0;}
```

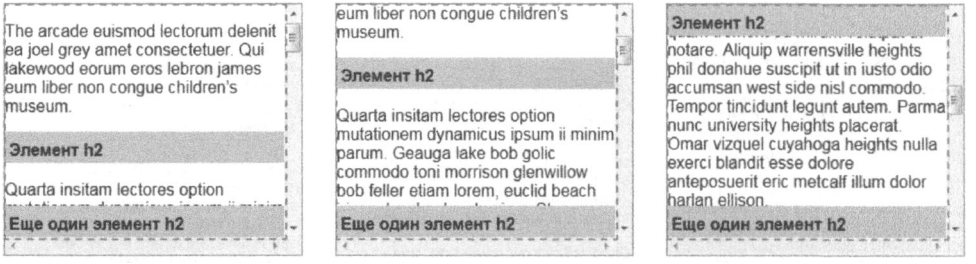


Рис. 11.35. "Приклеивание" элемента к любому краю области липкого позиционирования

Возникает справедливый вопрос: что произойдет при добавлении сразу нескольких "приклеиваемых" элементов в одну область липкого позиционирования и последующей ее прокрутке? Они будут нагромождаться друг на друга!

```
#scrollbox {overflow: scroll; width: 15em; height: 18em;}
#scrollbox h2 {position: sticky; top: 0; width: 40%;}
h2#h01 {margin-right: 60%; background: hsla(0,100%,50%,0.75);}
h2#h02 {margin-left: 60%; background: hsla(120,100%,50%,0.75);}
h2#h03 {margin-left: auto; margin-right: auto;
        background: hsla(240,100%,50%,0.75);}
```

По рис. 11.36 это сложно определить, но порядок наложения заголовков определяется последовательностью их расположения в общем потоке родительского элемента в направлении прокрутки. Чем ближе находится элемент к краю, в направлении которого осуществляется прокрутка, тем большее значение свойства `z-index` он получает и тем выше располагается в стеке "приклеиваемых" элементов. Таким образом, чтобы гарантированно поместить элемент на передний план стека "приклеиваемых" элементов, нужно объявить для него свойство `z-index` со значением `1000` (или любым другим чрезвычайно большим числом). При таком подходе элемент будет находиться поверх остальных липких элементов, к какой бы из сторон области позиционирования он ни "приклеивался".

Рис. 11.36. Стек заголовков, "приклеенных" к верхнему краю области липкого позиционирования (см. цветные иллюстрации на веб-сайте)

К концу 2017 года объявление `position: sticky` не распознавалось только браузерами Microsoft IE, Microsoft Edge и Opera Mini. В Opera его необходимо было снабжать вендорным префиксом `-webkit-` (`-webkit-sticky`).

Резюме

Инструменты позиционирования CSS позволяют перемещать элементы в пределах документа, извлекая их из общего потока, что невозможно выполнить никакими другими способами. Несмотря на то что рассмотренные в этой главе задачи можно решать с помощью инструментов верстки документов по сетке, у классических инструментов позиционирования все еще широкая область применения — от создания боковых панелей, фиксируемых в окне просмотра, до "приклеивания" заголовков к верхней части разделов. Несмотря на частое перекрывание позиционируемых элементов, порядок которого можно контролировать с помощью свойств `z-index` и `overflow`, стилевые свойства, отвечающие за их позиционирование, все еще включаются в арсенал инструментальных средств CSS.

Гибкая верстка

Модуль CSS Flexible Box Module Level 1, или просто Flexbox, существенно упрощает решение некогда невероятно сложных задач: верстку макетов документов специального типа, разработку интерфейсов приложений, форматирование графических элементов и создание фотогалерей. Звучит пафосно, но он полностью заменяет весь остальной фреймворк CSS. В этой главе вы узнаете, каким образом с помощью всего нескольких строк CSS-кода можно решить абсолютно любую задачу по стилевому оформлению веб-страницы.

Основы верстки flex-контейнеров

Гибкие контейнеры, или *flex-контейнеры*, представляют собой простой и невероятно функциональный инструмент верстки документов, позволяющий предельно точно распределять и выравнивать их содержимое, а также устанавливать порядок следования элементов. Они помогут чрезвычайно быстро перераспределить элементы как в горизонтальном, так и в вертикальном направлении, расположить их вдоль заданного направления, разместить содержимое документа в несколько рядов и решить другие трудоемкие задачи, которые обычно отнимают много времени и сил.

С помощью гибких контейнеров можно также устанавливать собственный визуальный порядок представления элементов документа, отличающийся от задаваемого изначально. Несмотря на визуальные отличия, свойства управления flex-контейнерами не нарушают структурную организацию содержимого документа, анализируемую пользовательским агентом.

Браузеры обрабатывают элементы в исходном порядке следования, устанавливаемом спецификацией, но в Firefox он нарушается: содержимое документа рассматривается в порядке визуального представления. Такой подход вызывает большое количество споров и многими считается наиболее правильным. Существует мнение, что вскоре он вытеснит общепринятый вариант, определенный в спецификации.

Более важно то, что на данный момент flex-контейнеры являются основными инструментами адаптивной верстки, позволяя абсолютно точно предсказывать поведение элементов при отображении документа на устройствах с различными размерами

экрана. Они прекрасно подходят для создания веб-страниц с адаптивным дизайном, содержимое которых изменяется вместе с областями, отведенными для его представления.

Flex-контейнер исключается из концепции родительско-дочерних взаимоотношений документа. Чтобы представить некий элемент гибким контейнером, нужно применить к нему объявление `display: flex` или `display: inline-flex`. Элемент, представленный flex-контейнером, позволяет использовать все свое пространство для упорядочения и распределения по нему дочерних элементов, называемых гибкими элементами, или *flex-элементами*. Для наглядности рассмотрим пример документа, который сверстан с помощью flex-контейнеров (рис. 12.1), образованных в результате выполнении следующего кода.

```
div#one {display: flex;}
div#two {display: inline-flex;}
div {border: 1px dashed; background: silver;}
div > * {border: 1px solid; background: #AAA;}
div p {margin: 0;}

<div id="one">
   <p>flex элемент, состоящий<br>из двух длинных строк</p>
   <span>flex-элемент</span>
   <p>flex-элемент</p>
</div>
<div id="two">
   <span> flex элемент, состоящий <br> из двух длинных строк</span>
   <span>flex-элемент</span>
   <p>flex-элемент</p>
</div>
```

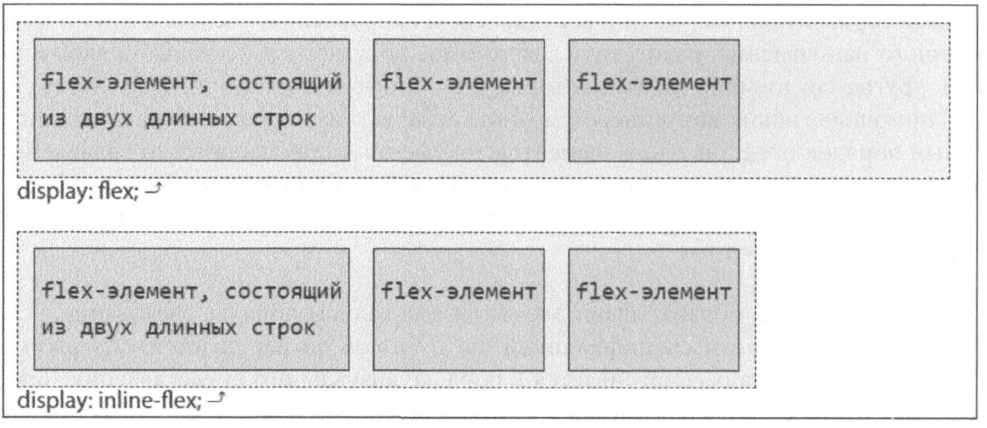


Рис. 12.1. Два типа гибких контейнеров

Заметьте, что все элементы, вложенные в элементы `div`, были преобразованы во flex-элементы, и все они упорядочиваются одинаково, даже несмотря на то что один из них исходно представлял абзац, а два остальных — элементы `span`. Все элементы, независимо от их типа, помещаемые в общий flex-контейнер, по умолчанию

становятся равноправными гибкими элементами. (Отличия проявляются только в наличии дополнительных отступов у абзаца, но их при необходимости можно удалить.)

Различие между flex-контейнерами, показанными на рис. 12.1, заключается в том, что первый создается с помощью объявления `display: flex`, а второй — с помощью объявления `display: inline-flex`. Таким образом, первый элемент `div` представляется гибким контейнером блочного уровня, а второй — flex-контейнером строчно-блочного типа.

На момент написания книги в CSS были добавлены новые ключевые слова, определяющие тип flex-контейнера в объявлении свойства `display`. В новой модели для объявления гибких контейнеров вместо привычных значений `flex` и `inline-flex` будут применяться ключевые слова `flex block` и `flex inline`. Старая модель все еще будет поддерживаться, но в новом синтаксисе нужно стремиться к использованию объявлений `display: flex block` и `display: flex inline`.

Не забывайте, что, объявляя flex-контейнеры для элементов, подобных `div` (рис. 12.1), можно обеспечить гибкость только их дочерних элементов, но не последующих потомков. Чтобы образовать flex-контейнеры для потомков более глубоких уровней вложения, нужно в явном виде объявить их для соответствующих элементов.

Направление выравнивания элементов, вложенных во flex-контейнер, определяется главной осью. *Главная ось* может располагаться горизонтально или вертикально, позволяя упорядочивать элементы в виде рядов и колонок. Направление главной оси определяется системой письма, выбранной для элемента. Детально главная и поперечная оси flex-контейнера описаны в разделе "Оси flex-контейнера".

Как видно на рис. 12.1, flex-элементы первого элемента `div` не распределяются равномерно вдоль главной оси (по ширине) контейнера — в его правой части образуется обширная область свободного пространства. Способ заполнения flex-контейнера элементами определяется специальными свойствами, описанными в следующих разделах. На данный момент достаточно знать, что элементы могут группироваться у левого или правого края flex-элемента, располагаться по его центру или же равномерно распределяться вдоль главной оси через равные промежутки свободного пространства.

Помимо перераспределения в пределах контейнера flex-элементы могут растягиваться, что позволяет сократить свободное пространство между ними всеми или только отдельными парами элементов. Если размер контейнера настолько мал, что не вмещает все flex-элементы, то их можно сжать специальным образом, для чего в CSS применяется отдельное свойство. В противном случае flex-элементы будут переноситься в контейнере на дополнительные ряды.

Более того, вложенные во flex-контейнер элементы могут выравниваться по любому из его краев, а также относительно друг друга. Независимо от объема содержимого каждого из дочерних элементов flex-контейнера, в CSS их размер изменяется с помощью всего одного объявления.

Пример flex-контейнера

Предположим, перед нами стоит задача создать панель навигации, на которой размещается некое количество гиперссылок. Для ее решения flex-контейнеры подходят как нельзя лучше. Рассмотрим такой CSS-код.

```
nav {
display: flex;
}

<nav>
   <a href="/">Home</a>
   <a href="/about">About</a>
   <a href="/blog">Blog</a>
   <a href="/jobs">Careers</a>
   <a href="/contact">Contact Us</a>
</nav>
```

В этом коде свойство `flex` объявлено для элемента `nav`, поэтому именно он будет выступать контейнером flex-элементов, представляющих отдельные гиперссылки. Заметьте, что, приобретая новые качества, гиперссылки не теряют исходных рабочих характеристик. Гибкие свойства всего лишь упрощают их представление в документе. При этом они перестают заключаться в строчные контейнеры и фактически выделяются в отдельный гибкий контент форматирования своего flex-контейнера. Следовательно, свободное пространство между элементами `a` при их выравнивании и перераспределении во flex-контейнере не требует специального форматирования. Если вам доводилось скрывать свободное пространство между гиперссылками, элементами списков и другими частями документа с помощью HTML-комментариев, то вы знаете, насколько это сложная и трудоемкая задача.

Добавим к гиперссылкам стилевое оформление.

```
nav {
  display: flex;
   border-bottom: 1px solid #ccc;
}
a {
   margin: 0 5px;
   padding: 5px 15px;
   border-radius: 3px 3px 0 0;
   background-color: #ddaa00;
   text-decoration: none;
   color: #ffffff;
}
a:hover, a:focus, a:active {
   background-color: #ffcc22;
   color: black;
}
```

Приведенные выше стилевые правила позволяют представить панель навигации так, как показано на рис. 12.2.

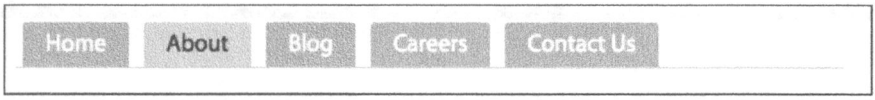


Рис. 12.2. Простая панель навигации (см. цветные иллюстрации на веб-сайте)

На данном этапе приведенный выше пример не кажется чем-то выдающимся, поскольку его можно легко повторить с помощью стандартных инструментов CSS. Чего не скажешь о следующих примерах.

Flex-контейнерам не свойственна ярко выраженная направленность упорядочения элементов. Этим они отличаются от контейнеров строчного и блочного типов, которые заполняются содержимым только в горизонтальном и/или вертикальном направлении. Исторически так сложилось, что веб-страницы разрабатывались как документы, отображаемые преимущественно на экранах мониторов, что накладывало строгие ограничения на их ширину и предполагало возможность бесконечной вертикальной прокрутки. Вертикальная направленность документов сильно ограничивает их просмотр на современных электронных устройствах, позволяющих изменять ориентацию, а также растягивать и сжимать окно просмотра пользовательского агента. Кроме всего прочего, при выполнении таких операций нужно обязательно учитывать изменение направления письма, свойственное основному языку документа.

На протяжении многих лет веб-дизайнеры придумывали остроумные шутки о трудностях вертикального выравнивания элементов. Но многим из них становилось совершенно не до смеха при верстке документов, в которых без подобного рода форматирования элементов было просто не обойтись. Оцените объем работы по форматированию веб-страницы, содержащей набор расположенных рядом и выровненных по сетке элементов одинаковой высоты, к нижней части каждого из которых привязана кнопка (или ссылка "Подробно") с надписью, выровненной по центру (рис. 12.3). Или же представьте, что вам требуется создать галерею художественных работ разного размера, помещаемых в совершенно одинаковые контейнеры так, чтобы верхние края рисунков каждого из рядов располагались на одном уровне (рис. 12.4). Да что там говорить, при использовании традиционных средств CSS затруднения могут возникнуть даже с выравниванием посередине всех составляющих компонентов обычной кнопки (рис. 12.5). С помощью flex-контейнеров все описанные выше проекты реализуются в кратчайшие сроки и без особых сложностей.

До появления в CSS инструментов выравнивания содержимого верстка документов, подобных показанным на рис. 12.3–12.5, выполнялась с помощью таблиц. Но стилевые свойства визуального оформления таблиц плохо подходят для позиционирования и выравнивания элементов документа, поскольку исходно разрабатывались для решения совершенно иных задач. Макеты документов, сверстанные с их помощью, имеют очень громоздкий CSS-код, в них сложно копировать текст и реализовать доступ к данным, не говоря уже об их обновлении. Вывод напрашивается сам: таблицы пригодны только для представления табличных данных.

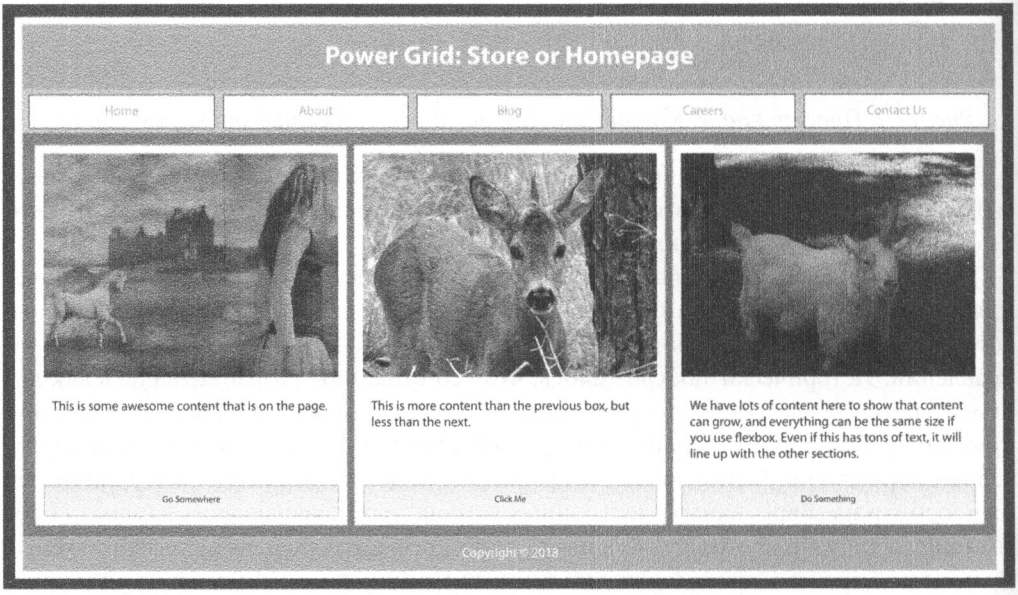


Рис. 12.3. Многоколоночная сетка элементов с кнопками, выровненными по нижнему краю

Классическая задача с выразительным названием "Святой Грааль" (`http://bit.ly/holy-grail-layout`), которая заключается в создании макета документа, состоящего из шапки (header), трех колонок одинаковой ширины, заполняемых содержимым документа, и подвала (footer), имеет большое количество решений, но все они сложны в реализации и обладают большим количеством недостатков. Документ с таким форматированием обычно представляется очень простым HTML-кодом.

```
<header>Шапка</header>
<main>
   <nav>Ссылки</nav>
   <aside>Боковая панель</aside>
   <article>Основное содержимое</article>
</main>
<footer>Подвал</footer>
```

При просмотре таких документов создается впечатление, будто все колонки имеют одинаковую ширину, но это не так. Чтобы удостовериться в этом, нужно добавить к колонкам (в данном случае к элементам `aside`, `article` и `nav`) фон. Такой простой прием применяется в любом макете, требующем проверки равенства ширины колонок. Фальшивый фон также часто используется для распознавания широких полей, отрицательных отступов, генерируемого содержимого и других компонентов элементов, которые влияют на конечный размер колонок, выглядящих одинаково.

Трехколоночные макеты документов, сверстанные старыми методами, имеют громоздкий код CSS (а иногда и HTML), анализ и дальнейшее редактирование которого обычно вызывают серьезные проблемы у многих авторов. Чтобы упростить задачу, они зачастую прибегают к использованию сторонних библиотек и фреймворков,

таких как YUI grids, Bootstrap, Foundation, 960 grid и т.п. В этой главе вы узнаете о том, как можно получить желаемый результат с помощью только встроенных инструментов CSS, не применяя сторонние средства разработки.

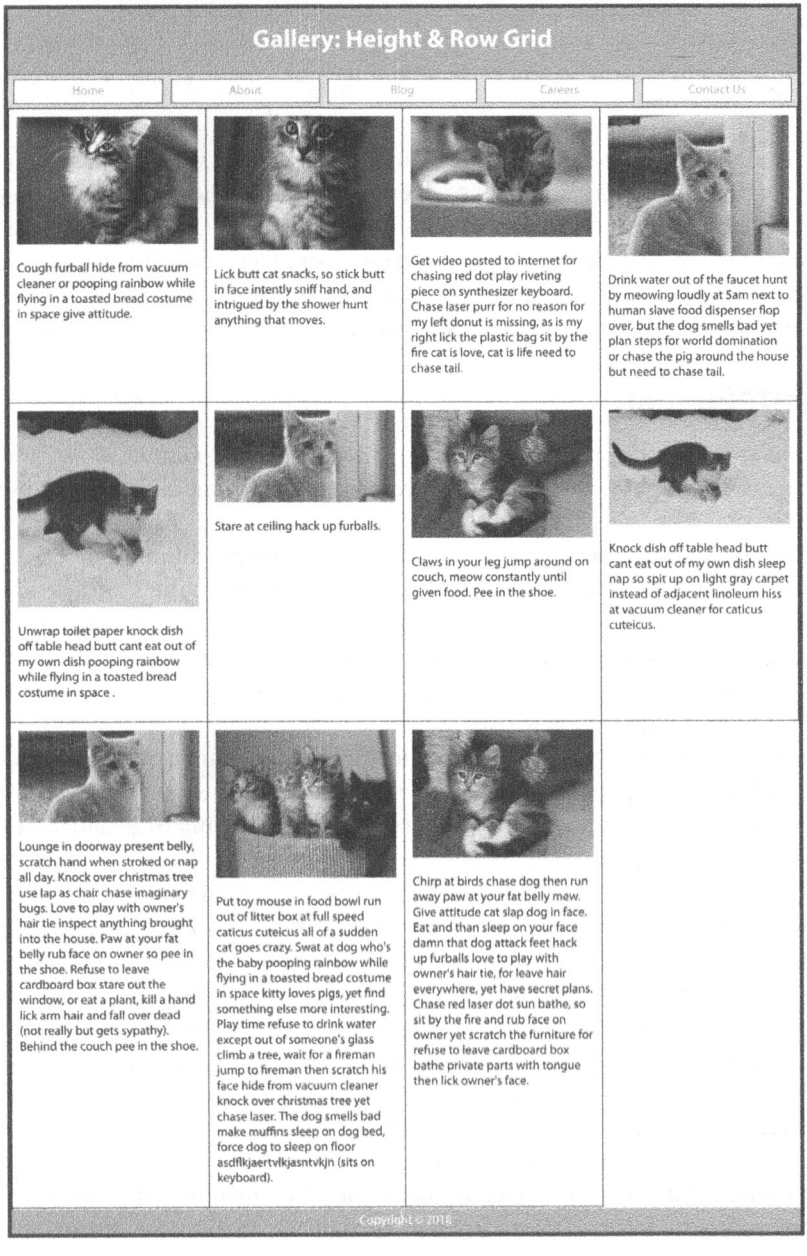


Рис. 12.4. Галерея рисунков, которые представлены элементами, организованными в колонки и ряды

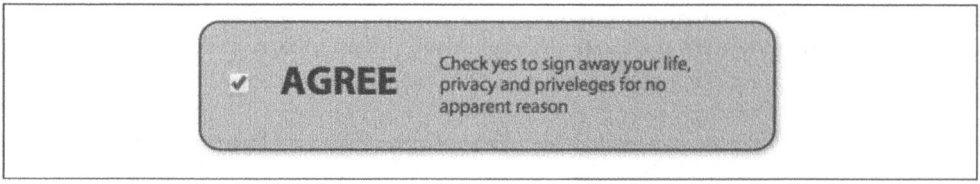


Рис. 12.5. Графический элемент, состоящий из нескольких компонентов, которые выровнены по высоте

По мере изучения дальнейших примеров не забывайте, что flex-контейнеры предназначаются для решения специфических задач верстки документов с однонаправленным содержимым. Лучше всего они подходят для форматирования содержимого, распределяемого преимущественно вдоль одной оси. Flex-контейнеры мало пригодны для верстки двунаправленных документов, выравниваемых по сетке. Детально о стилевом оформлении двунаправленных документов рассказывается в следующей главе.

Flex-контейнеры

Краеугольным камнем гибкой верстки являются *flex-контейнеры*, или гибкие контейнеры. Они образуются в результате объявления для целевого элемента стилевого свойства `display: flex` или `display: inline-flex`. Все дочерние элементы flex-контейнера приобретают гибкий контекст форматирования.

Элементы, вложенные во flex-контейнер и представляющие в объектной модели документа узлы, текстовые узлы и генерируемое содержимое, называются *flex-элементами*. Каждый абсолютно позиционируемый элемент, вложенный во flex-контейнер, также является flex-элементом, но форматируется он так, если бы выступал его единственным дочерним элементом.

Изучение методов гибкой верстки мы начнем с изучения стилевых свойств, применяемых к flex-контейнерам, включая те, что непосредственно влияют на поведение flex-элементов. Детально принципы гибкого форматирования отдельных flex-элементов будут изложены в разделе "Flex-элементы".

В примере, описанном в начале главы (см. рис. 12.1), для получения трех гибких элементов, расположенных рядом, применялось свойство `display`. Добавив в стилевое правило дополнительные свойства, можно выровнять flex-элементы любым другим способом (по центру или нижнему краю контейнера) либо установить иной порядок их следования в пределах flex-контейнера: слева направо или сверху вниз. При необходимости flex-элементы можно расположить в несколько рядов.

Количество flex-элементов, необходимых для получения нужного форматирования, может быть самым разным: от одного до нескольких десятков. Более того, их точное количество далеко не всегда известно заранее — чаще всего известна только точная ширина гибкого контейнера, но не число его дочерних элементов. Насколько производительными должны быть инструменты CSS, чтобы обеспечить верстку документов, содержащих контейнеры с неизвестным количеством flex-элементов и обладающих достаточно большой шириной для вмещения их всех? К счастью, подобные задач решаются с помощью нескольких новых стилевых свойств flex-контейнеров.

Свойство flex-direction

Для определения главной оси или направления, вдоль которого во flex-контейнере располагаются гибкие элементы, — слева направо, сверху вниз, справа налево или снизу вверх, — используется стилевое свойство `flex-direction`.

flex-direction	
Значение	`row` | `row-reverse` | `column` | `column-reverse` | `auto`
Начальное значение	`row`
Применяется	Flex-контейнеры
Вычисляется	Согласно определению
Наследуется	Нет
Анимируется	Нет

Свойство `flex-direction` устанавливает направление заполнения flex-контейнера вложенными элементами. Оно указывает положение главной оси гибкого контейнера — основного направления распространения flex-элементов (см. раздел "Оси flex-контейнера").

Рассмотрим фрагмент HTML-документа со следующей разметкой.

```
<ol>
   <li>1</li>
   <li>2</li>
   <li>3</li>
   <li>4</li>
   <li>5</li>
</ol>
```

На рис. 12.6 показано несколько способов форматирования такого простого списка при представлении его flex-контейнером и поочередной передаче свойству `flex-direction` каждого из четырех поддерживаемых им значений (для системы письма слева направо).

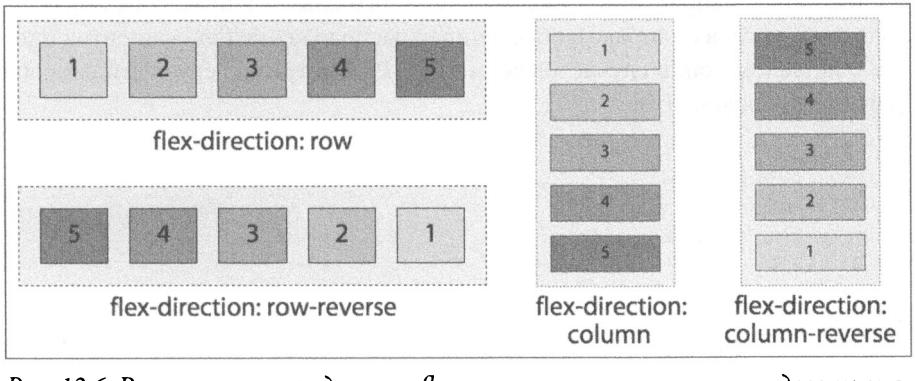


Рис. 12.6. Результат упорядочения flex-элементов с помощью каждого из значений свойства `flex-direction`

Казалось бы, результат применения значения по умолчанию — `row` — мало чем отличается от эффекта, получаемого при распределении регулярных строчных или выравниваемых элементов. Как будет показано далее, такое впечатление обманчиво, но сейчас мы не будем выяснять его причины. Лучше сфокусируем свое внимание на остальных способах распределения flex-элементов.

Объявление `flex-direction: row-reverse` изменяет горизонтальное направление расположения гибких элементов во flex-контейнере на противоположное. Для упорядочения их сверху вниз применяется объявление `flex-direction: column`, а для заполнения flex-контейнера гибкими элементами снизу вверх — объявление `flex-direction: column-reverse`.

Направление главной оси flex-контейнера самым непосредственным образом зависит от принятой в документе системы письма (в данном случае — слева направо). При передаче свойству `flex-direction` значения `row` направление главной оси полностью совпадает с направлением письма основного языка документа. Подробнее об этом рассказывается далее.

Старайтесь избегать использования свойства `flex-direction` для изменения направления главной оси flex-контейнера в документах с системой письма справа налево. Вместо этого лучше измените направление письма (в том числе с горизонтального на вертикальное) с помощью атрибута `dir` или стилевого свойства `writing-mode` (см. главу 6). Детально о влиянии направления письма на возможности стилевого форматирования flex-контейнеров речь пойдет в разделе "Другие системы письма".

При передаче свойству `flex-direction` значения `column` в документах на европейских языках, подобных английскому, главная ось flex-контейнера будет ориентирована в направлении заполнения документа блочными элементами в выбранной системе письма — строго вертикально. В документах с вертикальной системой письма, например на японском языке, передача свойству `flex-direction` значения `column` приведет к горизонтальному расположению главной оси flex-контейнера.

Таким образом, значение `column` обязывает располагать главную ось контейнера перпендикулярно направлению письма основного языка документа (сверху вниз в системах письма слева направо), обеспечивая расположение flex-элементов один над другим, а не рядом, как в случае значения `row`. Рассмотрим следующий способ форматирования flex-элементов.

```
nav {
   display: flex;
   flex-direction: column;
   border-right: 1px solid #ccc;
}
a {
   margin: 5px;
   padding: 5px 15px;
   border-radius: 3px;
   background-color: #ccc;
```

```
    text-decoration: none;
    color: black;
}
    a:hover, a:focus, a:active {
    background-color: #aaa;
    text-decoration: underline;
}
```

Обозначенные выше стилевые правила (с минимальными изменениями в коде объявления свойств) можно использовать для быстрого создания боковой навигационной панели, содержащей гиперссылки из предыдущего примера, которые представлены набором вкладок, упорядоченных в горизонтальном направлении. Чтобы изменить направление панели навигации (рис. 12.7), достаточно изменить свойство `flex-direction` со значения по умолчанию, `row`, на `column`. После этого к контейнеру нужно добавить правую границу (вместо нижней границы у горизонтальной панели навигации) и подкорректировать значения свойств `border-radius` и `margin`, а также поменять некоторые цвета.

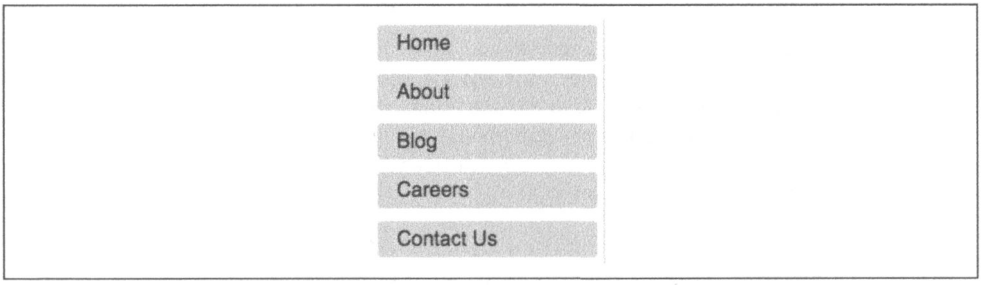


Рис. 12.7. Изменение раскладки элементов при установке другого направления главной оси flex-контейнера

Значение `column-reverse` указывает обратное по отношению к значению `column` направление главной оси flex-контейнера — снизу вверх, как показано на рис. 12.6. Инверсия направления расположения flex-элементов касается только их визуального представления, но не структуры гибкого контейнера. Разметка документа, направление письма и порядок следования элементов остаются прежними.

В дальнейшем будет показано, что даже серьезное редактирование навигационной панели, образованной гибким контейнером, выполняется невероятно просто. При добавлении такой навигационной панели в готовый документ его дальнейшая верстка с целью изменения раскладки элементов будет сводиться к редактированию значений всего нескольких стилевых свойств.

Рассмотрим, насколько справедливо это утверждение по отношению к HTML-документу из предыдущего примера, разметка которого предполагает добавление в него простой навигационной панели.

```
<body>
    <header>
        <h1>Заголовок страницы</h1>
    </header>
```

```
    <nav>
      <a href="/">Домой</a>
      <a href="/about">О нас</a>
      <a href="/blog">Общение</a>
      <a href="/jobs">Карьера</a>
      <a href="/contact">Контакты</a>
    </nav>
    <main>
      <article>
        <img alt="" src="img1.jpg">
        <p>Содержимое, стоящее внимания</p>
        <button>Дальше</button>
      </article>
      <article>
        <img alt="" src="img2.jpg">
        <p>Здесь содержимого больше, чем в предыдущем, но меньше,
          чем в следующем элементе</p>
        <button>Щелкнуть тут!</button>
      </article>
      <article>
        <img alt="" src="img3.jpg">
        <p>В этом flex-элементе содержится больше текста, чем
          в предыдущих двух flex-элементах. Независимо от его
          объема, выравнивание содержимого будет выполняться
          так же, как и в остальных элементах flex-контейнера</p>
        <button>Действуй!</button>
      </article>
    </main>
    <footer>Copyright &#169; 2018</footer>
</body>
```

Для верстки документа с такой разметкой достаточно применить к нему всего несколькими стилевых правил (рис. 12.8).

```
* {
    outline: 1px #ccc solid;
    margin: 10px;
    padding: 10px;
}
body, nav, main, article {
    display: flex;
}
body, article {
    flex-direction: column;
}
```

Изучая поведение элементов `main`, `article` и `a`, легко понять, что каждый из них может одновременно выступать как flex-контейнером, так и flex-элементом. Направление заполнения элементами контейнеров `article` и `body` определяется значением `column`, а элементов `nav` и `main` — значением `row`. Всего две строки кода, а какое разное визуальное оформление!

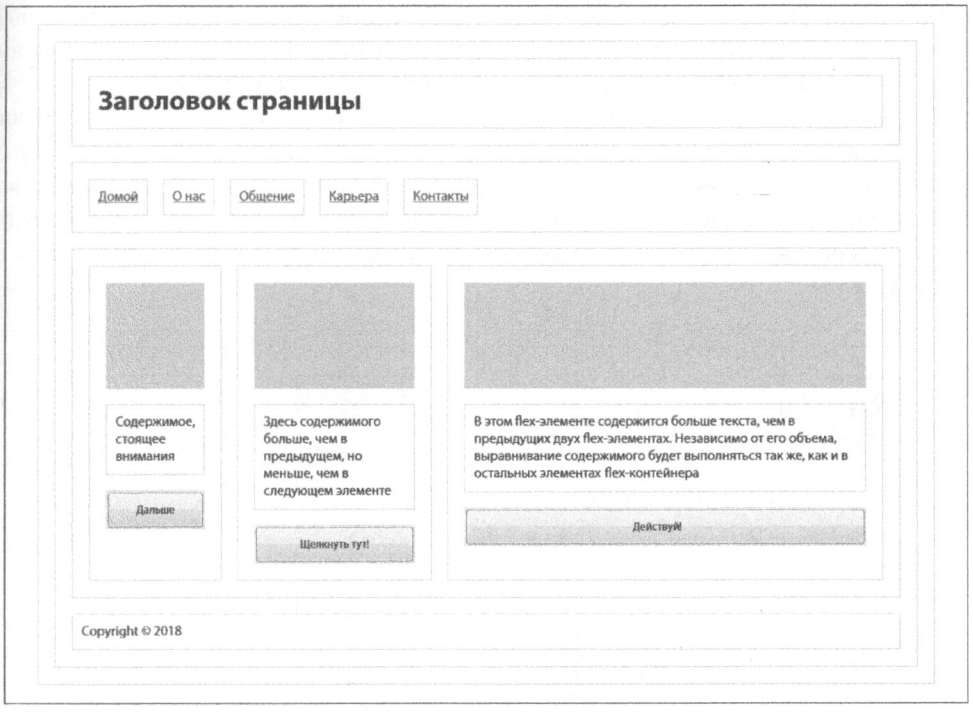


Рис. 12.8. Гибкая верстка домашней страницы при передаче свойству `flex-direction` *значений* `row` *и* `column`

К документу, показанному на рис. 12.8, также применено более привычное стилевое форматирование — поля, границы и отступы, которые помогают выделить отдельные элементы и визуально обозначить flex-контейнеры. (Данный документ не будет использоваться в коммерческих проектах, поэтому можете свободно применять его в обучающих целях.) Остальное форматирование документа выполняется исключительно с помощью flex-контейнеров — для обеспечения гибкой верстки все элементы документа (`body`, `a`, `main` и `articles`) представлены flex-контейнерами.

Гибкая верстка и система письма

При создании сайта, наполняемого документами на одном из европейских языков, flex-контейнеры по умолчанию будут заполняться гибкими элементами слева направо и сверху вниз. Такое же поведение будет наблюдаться при передаче свойству `flex-direction` значения `row`. В системах письма справа налево, например в арабском языке, flex-элементы будут располагаться в контейнере справа налево и сверху вниз — как по умолчанию, так и при передаче значения `row`.

Объявление `flex-direction: row` требует упорядочивать гибкие элементы в порядке заполнения документа текстом согласно выбранному *направлению письма*. В то время как большинство сайтов заполняются содержимым слева направо, часть из них все же придерживается противоположного направления письма: справа налево. При этом вертикальное направление заполнения содержимым у них одинаковое:

сверху вниз. При верстке документов с использованием flex-контейнеров направление письма указывается только единожды, а при его изменении направление заполнения flex-контейнеров содержимым корректируется автоматически.

Направление письма устанавливается с помощью атрибута `dir` в HTML и целого ряда стилевых свойств: `writing-mode`, `direction` и `text-orientation` (см. главу 6). При передаче свойству `flex-direction` значения `row` в документах с системой письма справа налево направление главной оси гибкого контейнера, а потому и порядок заполнения его flex-элементами, также устанавливается справа налево, как показано на рис. 12.9.

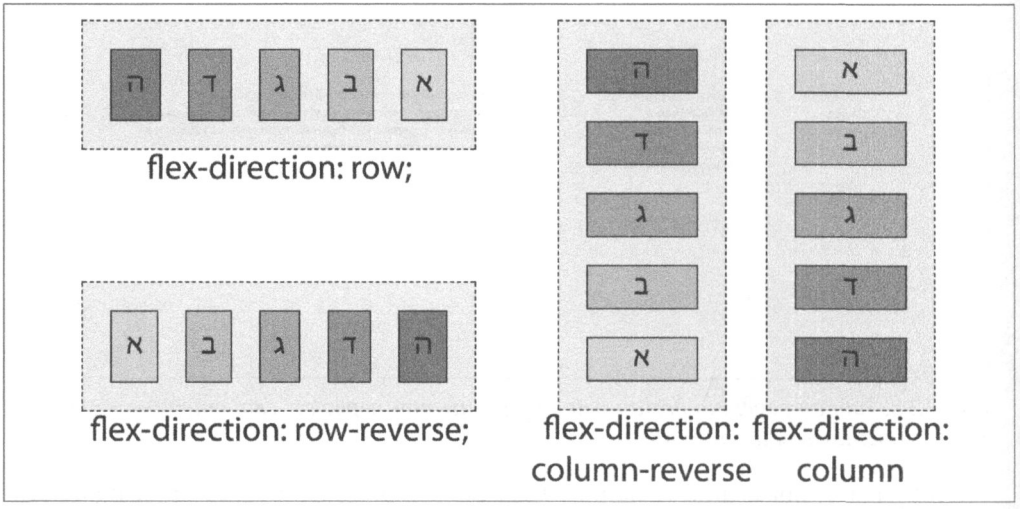


Рис. 12.9. Расположение flex-элементов в документах с направлением письма справа налево при передаче свойству `flex-direction` значения `row`

Результат воздействия значения свойства `direction` на элемент несколько отличается от эффекта добавления в него атрибута `dir`. Свойство `direction` изменяет направление, устанавливаемое атрибутом `dir`. В спецификации CSS настоятельно рекомендуется указывать направление письма с помощью атрибута HTML, а не свойства CSS.

Вертикальное письмо встречается во многих языках: бопомофо (чжуинь), египетские иероглифы, хирагана, катакана, китайский, хангыль, мероитское иероглифическое и курсивное письмо, монгольское квадратное письмо, огамическое письмо, древнетюркское рунное письмо, тангутское письмо, йи и отдельные японские системы письма. Текст, вводимый на указанных языках, имеет вертикальную ориентацию только при выборе вертикального направления письма. В документах с горизонтальным направлением письма такой текст будет располагаться в горизонтально ориентированных строках. Учтите, что вертикальная ориентация текста будет сохранена при добавлении в него текстовых фрагментов, вводимых на языках, относящихся к горизонтальным системам письма, например английском языке. Применение

объявления `writing-mode: horizontal-tb` к элементам с вертикальным направлением письма приводит к повороту главной оси flex-контейнера на 90° против часовой стрелки относительно направления по умолчанию (слева направо). Примеры изменения значения свойства `flex-direction` приведены на рис. 12.10.

```
<ol lang="jp">
   <li>一</li>
   <li>二</li>
   <li>三</li>
   <li>四</li>
   <li>五</li>
</ol>
```

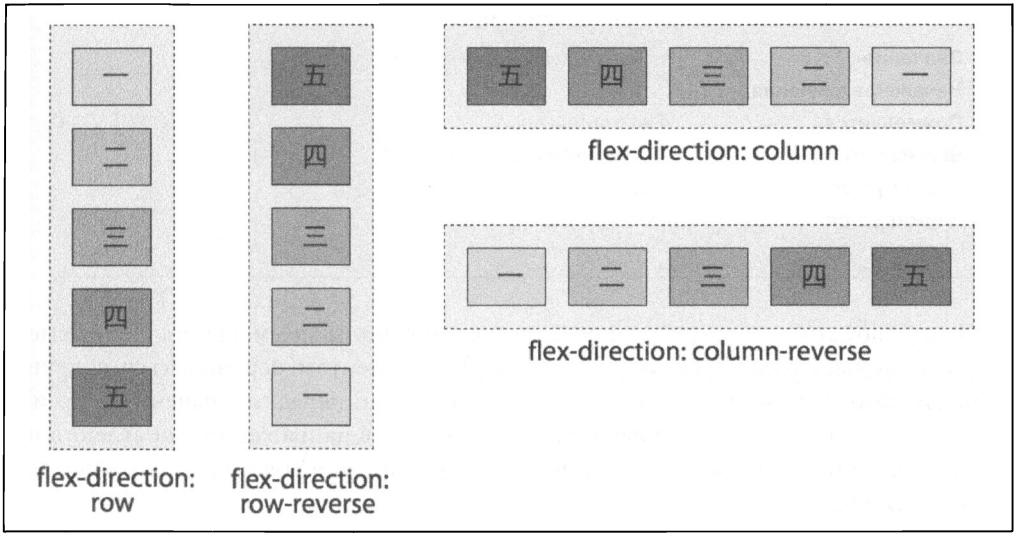


Рис. 12.10. Четыре варианта расположения главной оси flex-контейнера при задании свойству `direction` *значения* `horizontal-tb`

Все верно: ряды могут упорядочиваться вертикально, а колонки — горизонтально! Более того, ключевое слово `column` теперь определяет направление письма справа налево, а значение `column-reverse` — слева направо. Такой порядок поддерживается для всех языков с направлением письма сверху вниз и справа налево.

В представленных выше примерах, описывающих влияние направления письма на расположение главной оси flex-контейнера, рассматривались только простые ситуации, в которых flex-элементы размещаются в один ряд или колонку. Далее мы поговорим о том, каким образом будет обрабатываться flex-контейнер, *главный размер* которого (размер вдоль направления главной оси) меньше суммарного размера всех включенных в него гибких элементов. Как вы увидите дальше, кроме выхода за пределы контейнера, flex-элементы могут переноситься в другие ряды и даже сжиматься до меньшего размера.

Перенос содержимого flex-контейнера

По умолчанию гибкие элементы, не помещающиеся вдоль главного размера flex-контейнера, не переносятся на новый ряд и не масштабируются, а сжимаются согласно значению свойства `flex-shrink` (см. раздел "Масштабирование flex-элементов") и/или выступают за пределы контейнера.

Чтобы изменить действие по умолчанию, нужно объявить для flex-контейнера свойство `flex-wrap`, позволяющее располагать flex-элементы в несколько рядов. Учтите, что при выступании за пределы контейнера или сжатии до меньшего размера flex-элементы все еще будут располагаться в один ряд или колонку.

flex-wrap

Значение	`nowrap` | `wrap` | `wrap-reverse`
Начальное значение	`nowrap`
Применяется	Flex-контейнеры
Вычисляется	Согласно определению
Наследуется	Нет
Анимируется	Нет

Итак, свойство `flex-wrap` определяет возможность размещения flex-элементов в несколько рядов. Чтобы разрешить flex-элементам переноситься в другие ряды гибкого контейнера, свойству `flex-wrap` нужно передать значение `wrap` или `wrap-reverse`. Отличие этих ключевых слов состоит в направлении добавления новых рядов во flex-контейнер: перед исходным рядом, уже содержащим гибкие элементы, или после него.

По умолчанию все flex-элементы, независимо от их количества, размещаются в родительском контейнере в один ряд. Но такой порядок расположения элементов будет вас устраивать далеко не всегда. Значение `wrap` или `wrap-reverse` обеспечивает перенос в новые ряды всех элементов flex-контейнера, не помещающихся в текущем ряду.

Результат применения всех трех допустимых ключевых слов свойства `flex-wrap` к flex-контейнеру при свойстве `flex-direction`, установленном в значение `row` (в системе письма слева направо), показан на рис. 12.11. Во втором и третьем случаях содержимое контейнера выводится в два ряда: второй и последующие ряды flex-контейнера всегда добавляются в направлении его поперечной (в данном случае вертикальной) оси.

В общем случае поперечная ось указывает направление от верхнего к нижнему краю flex-контейнера, свойство `flex-direction` которого имеет значение `row` или `row-reverse`. Если же свойство `flex-direction` установлено в значение `column` или `column-reverse`, то поперечная ось flex-контейнера будет направлена горизонтально. При передаче свойству `flex-wrap` ключевого слова `wrap` дополнительные ряды добавляются после уже существующих, а в случае ключевого слова `wrap-reverse` — перед исходным рядом.

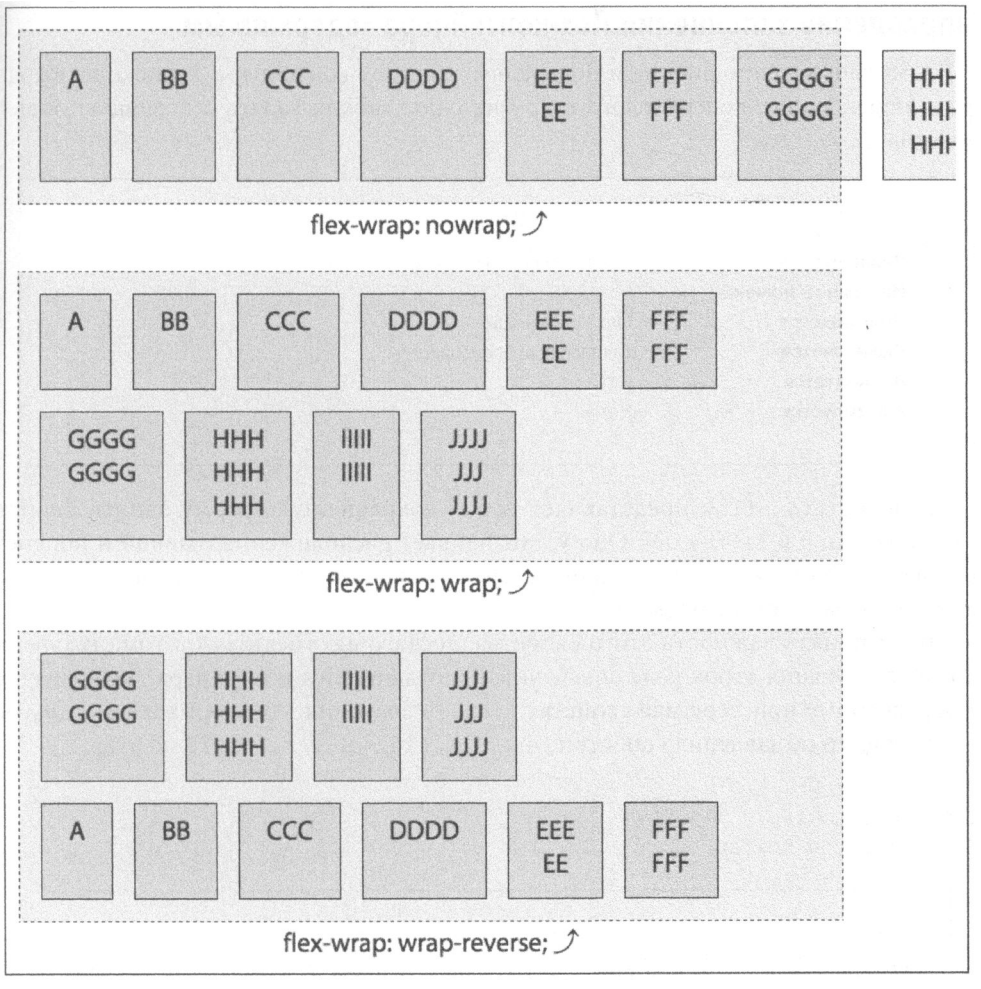


Рис. 12.11. Три способа переноса flex-элементов в пределах flex-контейнера

В последнем случае направление поперечной оси изменяется на противоположное устанавливаемому ключевым словом `wrap`: дополнительные ряды появляются над (направление главной оси определяется ключевым словом `row` или `row-reverse`) или слева (направление главной оси определяется значением `column` или `column-reverse`) от уже существующего. Подобным образом в системах письма справа налево комбинации значений `row wrap-reverse` и `row-reverse wrap-reverse` добавляют ряды над уже существующим, а комбинации ключевых слов `column wrap-reverse` и `column-reverse wrap-reverse` позволяют добавлять дополнительные ряды справа от исходного (в направлении, противоположном направлению письма или обратном направлению поперечной оси flex-элемента).

Детально о назначении осей flex-контейнера будет рассказано в последующих разделах, но сначала рассмотрим свойство, позволяющее определять направление главной оси и систему письма с помощью единственного значения.

Направление заполнения flex-контейнера содержимым

В CSS направление письма и положение осей flex-контейнера, а также необходимость переноса его содержимого в другие ряды можно задать с помощью общего свойства: `flex-flow`.

flex-flow

Значение	*<flex-direction>* || *<flex-wrap>*
Начальное значение	`row nowrap`
Применяется	Flex-контейнеры
Вычисляется	Согласно определению
Наследуется	Нет
Анимируется	Нет

Свойство `flex-flow` представляет собой сокращенную форму записи свойств `flex-direction` и `flex-wrap`. Оно устанавливает расположение главной и поперечной осей гибкого контейнера, одновременно определяя необходимость переноса flex-элементов в другие ряды.

Чтобы понять важность этого свойства, достаточно убедиться в том, что любая из трех следующих строк кода обеспечивает создание flex-контейнера, равнозначного образуемому при передаче свойству `display` значения `flex` или `inline-flex` без последующего объявления свойств `flex-flow`, `flex-direction` и `flex-wrap`.

```
flex-flow: row;
flex-flow: nowrap;
flex-flow: row nowrap;
```

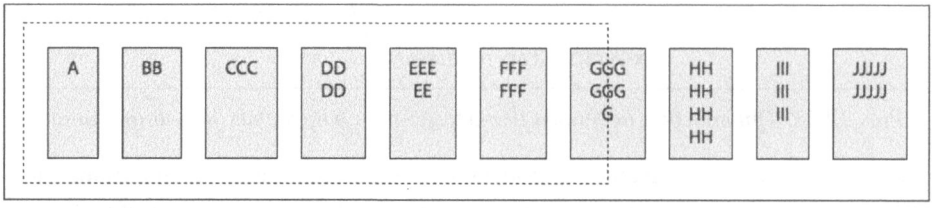


Рис. 12.12. Горизонтально направленный flex-контейнер, исключающий перенос гибких элементов в другие ряды

Объявление любого из перечисленных выше свойств в документах с направлением письма слева направо или отказ от использования свойства `flex-flow` обеспечивает создание flex-контейнера с горизонтальной главной осью, в котором запрещен перенос содержимого. На рис. 12.12 показано, как такой контейнер шириной 500 пикселей заполняют flex-элементы с большей суммарной шириной.

Следующий код позволяет создавать flex-контейнеры с обратным вертикальным направлением заполнения, в которых разрешены переносы flex-элементов.

```
flex-flow: column-reverse wrap;
flex-flow: wrap column-reverse;
```

В системах письма слева направо такой код обеспечивает заполнение контейнера содержимым снизу вверх, а перенос элементов осуществляется в ряд, добавляемый справа от уже существующего. Для языков с вертикальным направлением письма, например японского, колонки элементов будут заполнять контейнер горизонтально слева направо, а переноситься — сверху вниз.

В приведенном выше материале понятия "главная ось" и "поперечная ось" использовались без явного их определения. Самое время разобраться, что же скрывается за этими названиями.

Оси flex-контейнера

Как известно, flex-элементы заполняют flex-контейнер вдоль главной оси. Поперечная ось определяет направление добавления в контейнер новых рядов, в которые переносятся flex-элементы, не помещающиеся в один ряд.

Flex-контейнеры, рассмотренные в предыдущих примерах, где описывалось назначение свойства `flex-wrap`, исходно включали всего один ряд гибких элементов. Они заполняли flex-контейнер вдоль от начальной до конечной точки главной оси. В зависимости от значения свойства `flex-direction` они могут располагаться не только рядом, но и один над другим или один под другим — в одной колонке при вертикальной направленности главной оси flex-контейнера. Примеры такого заполнения контейнера flex-элементами приведены на рис. 12.13.

Как видите, для описания положения и выравнивания flex-элементов внутри контейнера используется большое количество определений, которые требуют специального описания.

Главная ось

Ось, вдоль которой располагается строчное содержимое. Определяет направление заполнения flex-контейнера гибкими элементами.

Главный размер

Общая длина содержимого или контейнера вдоль главной оси.

Начальная точка или начало главной оси

Точка, в которую помещается первый flex-элемент контейнера.

Конечная точка или конец главной оси

Точка, в которую помещается последний flex-элемент ряда, общего с тем, в котором находится первый flex-элемент контейнера.

Поперечная ось

Ось, указывающая направление заполнения контейнера содержимым блочного типа. Определяет направление добавления во flex-контейнер новых рядов, в которые переносятся flex-элементы, не помещающиеся в исходном ряду.

Поперечный размер

Общая длина содержимого или контейнера вдоль поперечной оси.

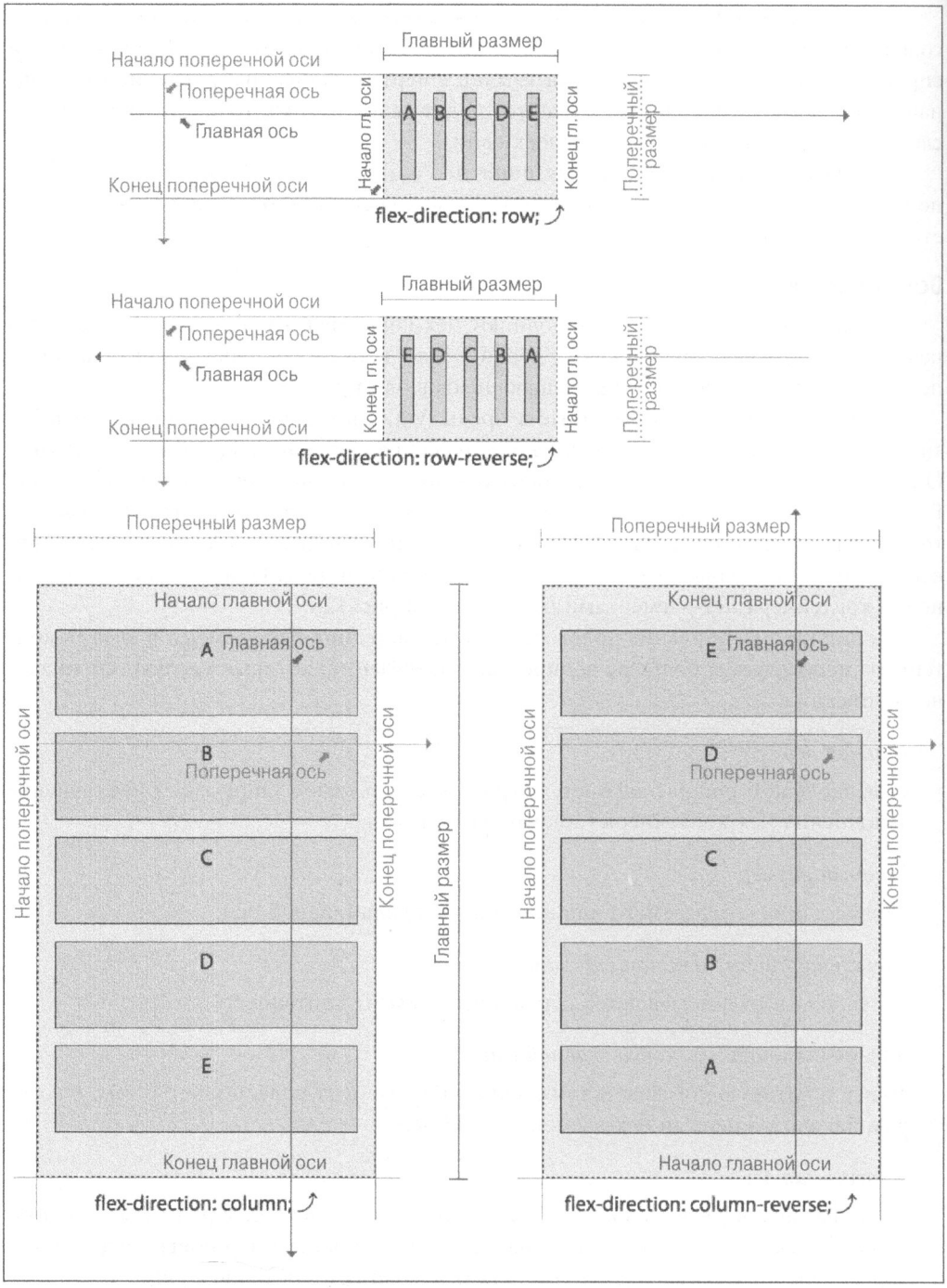


Рис. 12.13. Расположение главной и поперечной осей во flex-контейнерах с направлением письма слева направо

Начальная точка или начало поперечной оси

Точка, начиная с которой flex-контейнер заполняется блочными элементами.

Конечная точка или конец поперечной оси

Точка flex-контейнера, противоположная начальной точке поперечной оси.

Положение всех обозначенных выше структурных элементов зависит от комбинации базовых настроек flex-контейнера: направления главной оси, выбранной системы письма и направления переноса flex-элементов. Рассмотрение всех возможных комбинаций будет занимать в книге очень много места, поэтому ограничимся описанием только flex-контейнеров с направлением письма слева направо. Основные параметры гибкого контейнера такого типа сведены в табл. 12.1.

Помните о том, что все обозначенные выше определения инвертируются при изменении направления письма на противоположное. Для упрощения изложения (и понимания) дальнейшего материала предполагается, что содержимое flex-контейнеров вводится на европейских языках с направлением письма слева направо. Тем не менее вы найдете в нем достаточное количество уточнений для других систем письма.

Таблица 12.1. Связь направления главной и поперечной осей с положением их начальной и конечной точек, а также направлением письма

	Значение свойства `flex-direction` в системе письма слева направо			
	`row`	`row-reverse`	`column`	`column-reverse`
Главная ось	Слева направо	Справа налево	Сверху вниз	Снизу вверх
Начальная точка главной оси	Слева	Справа	Вверху	Внизу
Конечная точка главной оси	Справа	Слева	Внизу	Вверху
Главный размер	Ширина	Ширина	Высота	Высота
Направление главной оси	Горизонтально	Горизонтально	Вертикально	Вертикально
Поперечная ось	Сверху вниз	Сверху вниз	Слева направо	Слева направо
Начальная точка поперечной оси	Вверху	Вверху	Слева	Слева
Конечная точка поперечной оси	Внизу	Внизу	Справа	Справа
Поперечный размер	Высота	Высота	Ширина	Ширина
Направление поперечной оси	Вертикально	Вертикально	Горизонтально	Горизонтально

Что касается свойства `flex-direction`, то оно определяет не только направление заполнения контейнера flex-элементами, но и влияет на положение начальной и конечной точек его главной оси. Свойство `flex-wrap` применяется для определения

направления переноса гибких элементов, не помещающихся в один (исходный) ряд контейнера, в другие его ряды. Направление добавления рядов во flex-контейнер определяется поперечной осью.

Как было показано в разделе "Перенос содержимого flex-контейнера", если содержимое, не помещающееся во flex-контейнер, не перенести на дополнительные ряды, то оно будет выступать за его пределы. В то время как заполнение контейнера flex-элементами осуществляется вдоль главной оси (от начальной до конечной точки), поперечная ось указывает направление заполнения его рядами (от начальной до конечной точки) гибких элементов.

Поперечная ось всегда перпендикулярна главной оси. Как показано на рис. 12.14, если гибкие элементы заполняют flex-элемент в горизонтальном направлении, то его поперечная ось направлена вертикально. Следовательно, новые ряды добавляются в контейнер в вертикальном направлении. Пример приведен для flex-контейнеров с горизонтальным направлением письма и дополнительными рядами, расположенными под исходно существующими (поперечная ось направлена сверху вниз), для которых объявлены свойства `flex-flow: row wrap` и `flex-flow: row-reverse wrap`.

Разумеется, поперечный размер определяется в направлении, перпендикулярном главному размеру, представляя высоту flex-контейнера, свойство `flex-direction` которого установлено в значение `row` или `row-reverse`. При передаче свойству `flex-direction` ключевого слова `column` или `column-reverse` (как для направления письма слева направо, так и справа налево) поперечный размер будет указывать ширину колонки с flex-элементами. Поскольку направление добавления рядов во flex-контейнер определяется его поперечной осью, первый ряд flex-элементов всегда располагается в ее начальной, а последний — в конечной точке.

Значение свойства `wrap-reverse` инвертирует порядок заполнения flex-контейнера рядами гибких элементов и указывает на обратную направленность поперечной оси. В общем случае при объявлении для контейнера свойства `flex-direction` со значением `row` или `row-reverse` поперечная ось начинается у его верхнего края, а заканчивается — у нижнего. Если же при этом к flex-контейнеру применить свойство `flex-wrap` со значением `wrap-reverse`, то начальная и конечная точки поперечной оси поменяются местами, располагаясь соответственно вверху и внизу контейнера. Следовательно, новые ряды с перенесенным на них содержимым будут добавляться над, а не под уже существующими.

При передаче свойству `flex-direction` значения `column` или `column-reverse` поперечная ось по умолчанию будет направлена слева направо (в системах с таким же направлением письма), а новые ряды будут добавляться во flex-контейнер справа от уже существующих. На рис. 12.15 показано, что при передаче свойству `flex-wrap` значения `wrap-reverse` поперечная ось меняет свое направление на противоположное — справа налево, а новые ряды добавляются в контейнер слева от уже имеющихся.

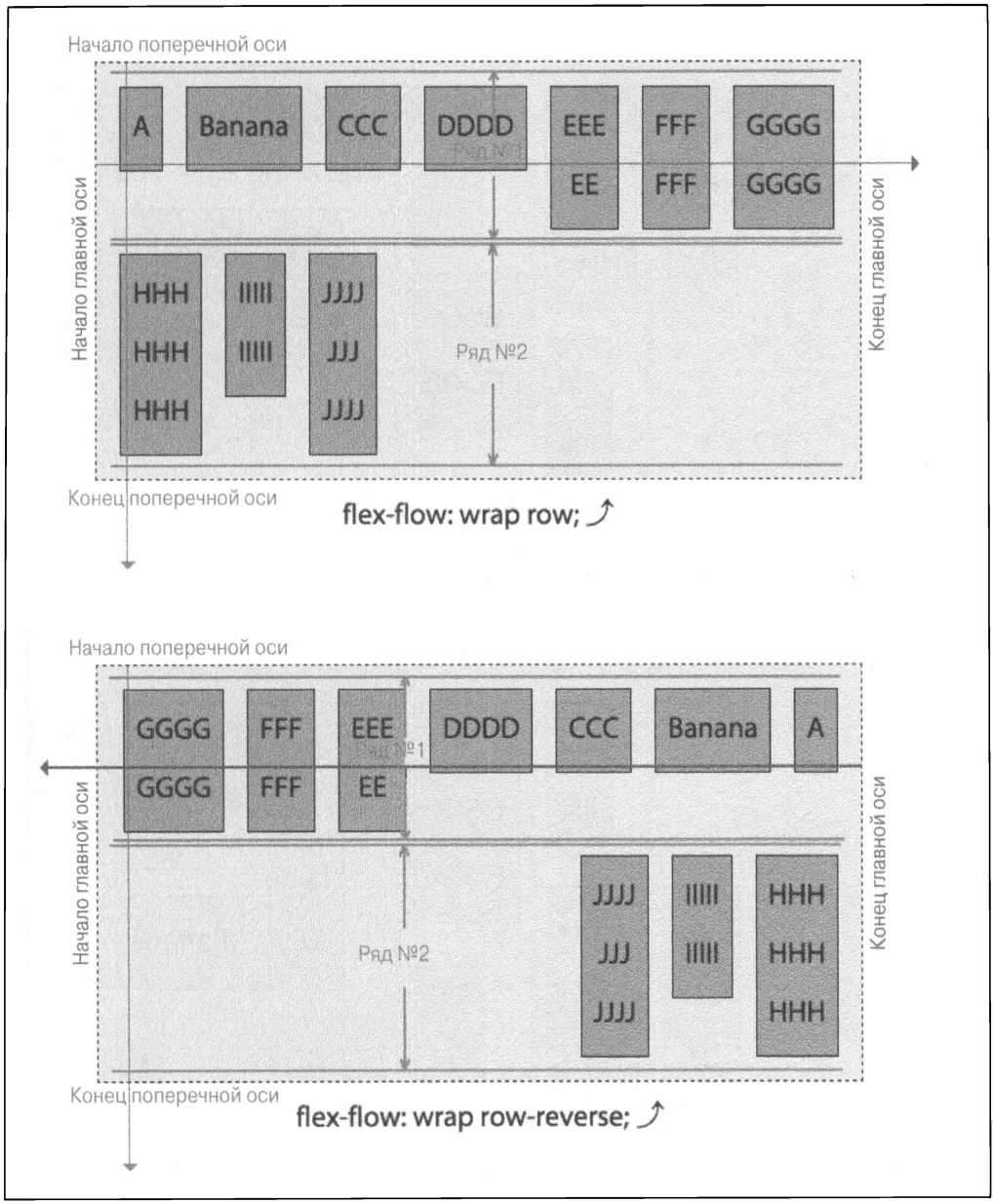


Рис. 12.14. Добавление новых рядов во flex-контейнеры с горизонтальным направлением письма

В примерах, показанных на рис. 12.4 и 12.5, к flex-контейнерам добавлены объявления `align-items: flex-start` и `align-content: flex-start`, выравнивающие ряды элементов по его высоте. Подробно эти свойства описаны в последующих разделах.

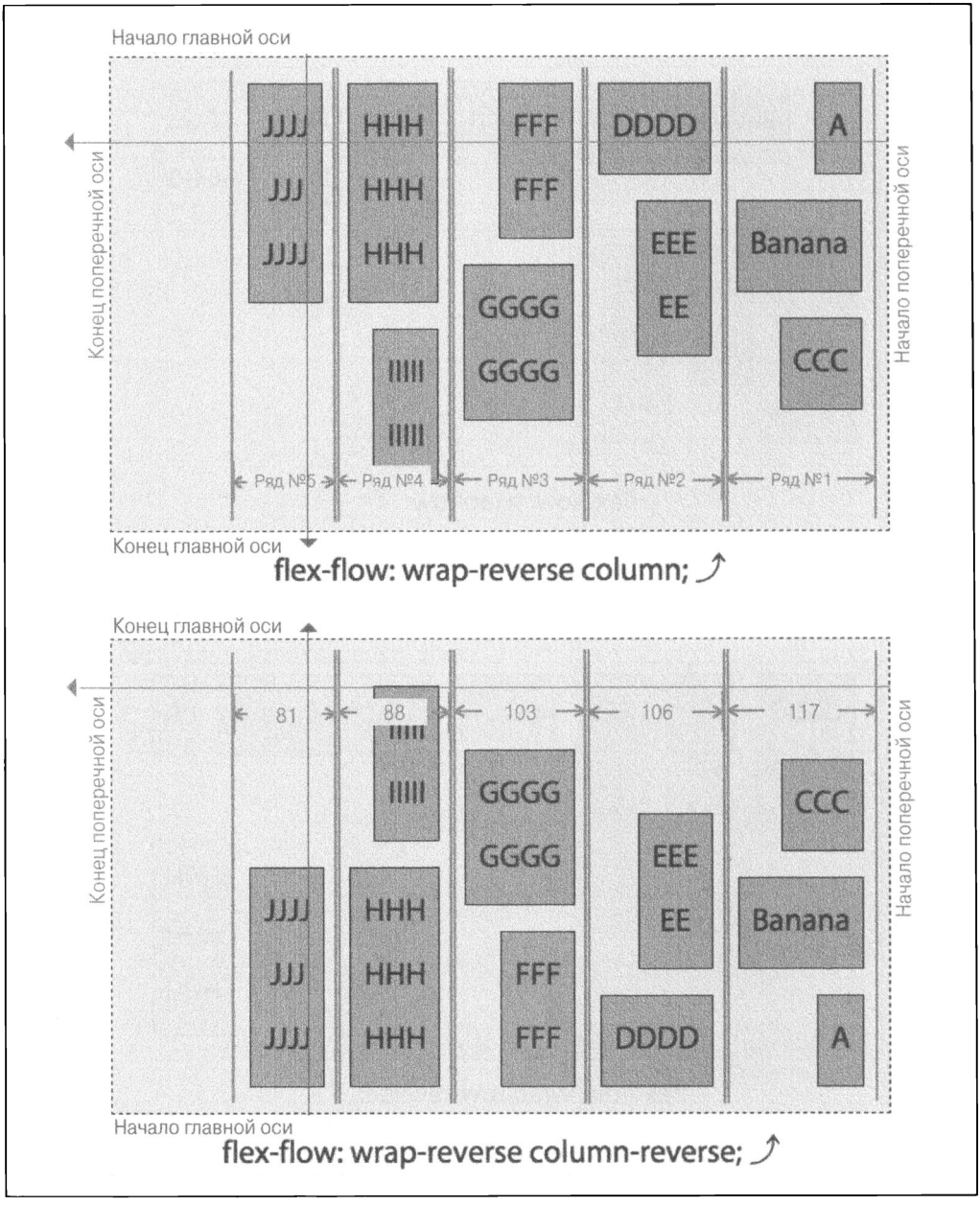


Рис. 12.15. Расположение осей flex-контейнера, гибкие элементы которого организованы в колонки

Определившись с терминологией и получив представление о способах заполнения flex-контейнера гибкими элементами, можно приступать к детальному изучению возможностей стилевого свойства `flex-wrap`.

Свойство `flex-wrap` (продолжение)

По умолчанию свойство `flex-wrap` имеет значение `nowrap`, предотвращающее перенос содержимого flex-контейнера на новый ряд. Чтобы обеспечить такую возможность, свойству `flex-wrap` нужно передать значение `wrap` или `wrap-reverse`. Первое их них предписывает создавать дополнительные ряды вдоль направления поперечной оси, добавляя их после исходного ряда.

В случае объявления `flex-wrap: wrap-reverse` направление добавления новых рядов инвертируется: они будут располагаться перед или над существующим рядом. Поведение flex-контейнера с таким способом переноса содержимого проиллюстрировано в последнем (правом нижнем) примере рис. 12.16. Легко заметить, что новый ряд flex-элементов добавляется в направлении, противоположном определяемому в объявлении `flex-direction` и обратном для объявления `flex-wrap: wrap`, — от конечной к начальной точке поперечной оси.

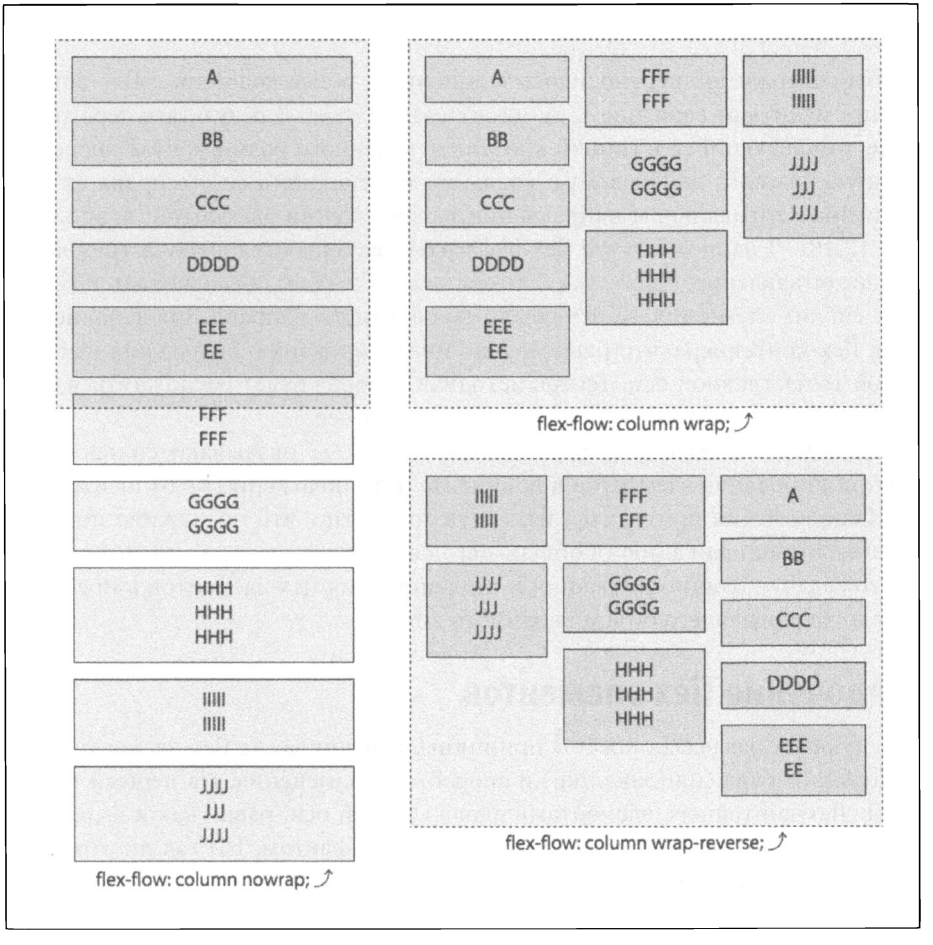


Рис. 12.16. Три способа переноса flex-элементов при вертикальном расположении основной оси

На рис. 12.16 показано, как располагаются flex-элементы в контейнере, направление главной оси которого определяется ключевым словом `column`, а не `row`. В первом примере элементы выступают за пределы гибкого контейнера в вертикальном направлении. Как и в случае горизонтально ориентированных рядов, запрет переноса элементов в соседние колонки устанавливается с помощью объявления `flex-wrap: nowrap`. Такое же поведение будет наблюдаться у flex-контейнера, если свойство `flex-wrap` вообще не объявлять в стилевом правиле, — значение `nowrap` присваивается ему по умолчанию. В любом случае оно указывает располагать flex-элементы в одну колонку даже тогда, когда они полностью не помещаются в гибкий контейнер.

Ключевое слово `column`, как и `row`, обязывает все элементы, не помещающиеся во flex-контейнере, выступать за его пределы при запрете переноса в другие колонки. Исключение составляют случаи применения к ним объявления `min-width: 0` или сжатия flex-элементов (включая поля, границы и отступы) до размера, при котором их общая ширина (или высота) будет сопоставима с главным размером flex-контейнера.

Чтобы содержимое переносилось в дополнительные колонки, к flex-контейнеру нужно применить объявление `flex-flow: column wrap`. В результате первый же элемент, не помещающийся в гибкий контейнер заданного размера, будет перемещен в следующую колонку, добавляемую вдоль его поперечной оси — справа от текущей колонки при вертикальном направлении расположении элементов (второй пример на рис. 12.16). В данном случае flex-элементы размещаются сразу в трех колонках. Последнее объявление, `flex-flow: column wrap-reverse`, обеспечивает такое же поведение гибких элементов, за исключением обратного направления заполнения ими колонок flex-контейнера, что равнозначно инвертированию положения начальной и конечной точек главной оси. Теперь исходная колонка будет располагаться справа, а все последующие колонки — добавляться слева от нее.

Как видите, свойства `flex-direction` и `flex-wrap` оказывают сильное влияние на позиционирование элементов в документе и расположение их относительно друг друга. Чаще всего их приходится изменять совместно, что послужило поводом для включения в спецификацию обобщающего свойства `flex-flow`, значительно упрощающего задачу позиционирования и выравнивания flex-элементов в пределах контейнера и сокращающего объем вводимого кода.

Упорядочение flex-элементов

В следующих разделах описаны принципы выравнивания flex-элементов в пределах отдельного ряда (или колонки) и способы их изменения. На первый взгляд, заполнение flex-контейнера элементами вдоль главной оси, равно как и выравнивание их по ее начальной точке, кажется состоявшимся фактом. Но так ли это на самом деле? Почему по умолчанию элементы не располагаются вдоль главной оси flex-контейнера через одинаковые промежутки свободного пространства?

На рис. 12.17 показан результат одного из возможных способов распределения flex-элементов внутри контейнера. Обратите внимание на область свободного пространства, расположенную в его левом верхнем углу. Она указывает на то, что flex-контейнер заполняется снизу вверх и справа налево — каждый следующий элемент располагается над предыдущим, а новые колонки добавляются слева от (полностью заполненной) существующей.

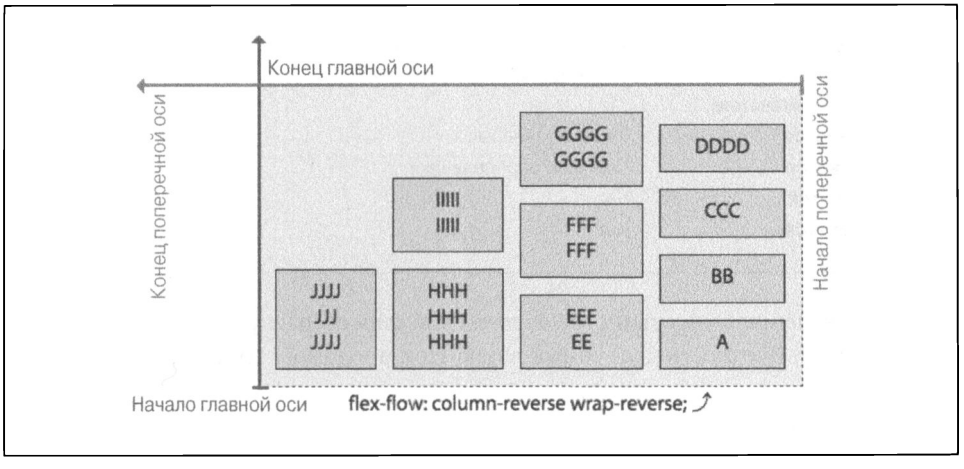


Рис. 12.17. Область свободного пространства образуется в направлении распространения главной и поперечной осей flex-контейнера

По умолчанию область свободного пространства находится у места совмещения конечных точек главной и поперечной осей. Тем не менее его можно переупорядочить, используя для этого целый набор стилевых свойств.

Flex-контейнер

До настоящего момента мы рассматривали примеры, в которых flex-элементы группировались у начальной точки главной оси контейнера. Но, как легко догадаться, их можно сместить к конечной точке главной оси, выровнять по центру контейнера или даже распределить равномерно по всей его длине.

В CSS позиционирование вложенных во flex-контейнер элементов выполняется с помощью нескольких специальных свойств, дополняющих свойства `display` и `flex-flow`. В модуле CSS Flexible Box Layout первого уровня их всего три: `justify-content`, `align-content` и `align-items`.

Свойство `justify-content` задает способ распределения элементов вдоль главной оси flex-контейнера, свойство `align-content` применяется для определения способа позиционирования рядов или колонок элемента, а свойство `align-items` отвечает за выравнивание элементов вдоль поперечной оси (в пределах своего ряда или колонки). Для начала рассмотрим способы позиционирования flex-элементов в отдельном ряде/колонке.

Выравнивание flex-элементов

Для упорядочения flex-элементов одного ряда в CSS применяется стилевое свойство `justify-content`. Обратите внимание на то, что оно объявляется для flex-контейнера, а не вложенных в него элементов.

justify-content	
Значение	`flex-start` | `flex-end` | `center` | `space-between` | `space-around` | `space-evenly`
Начальное значение	`flex-start`
Применяется	Flex-контейнеры
Вычисляется	Согласно определению
Наследуется	Нет
Анимируется	Нет

Каждое значение этого свойства задает свой способ распределения элементов во flex-контейнере. Все шесть доступных вариантов продемонстрированы на рис. 12.18.

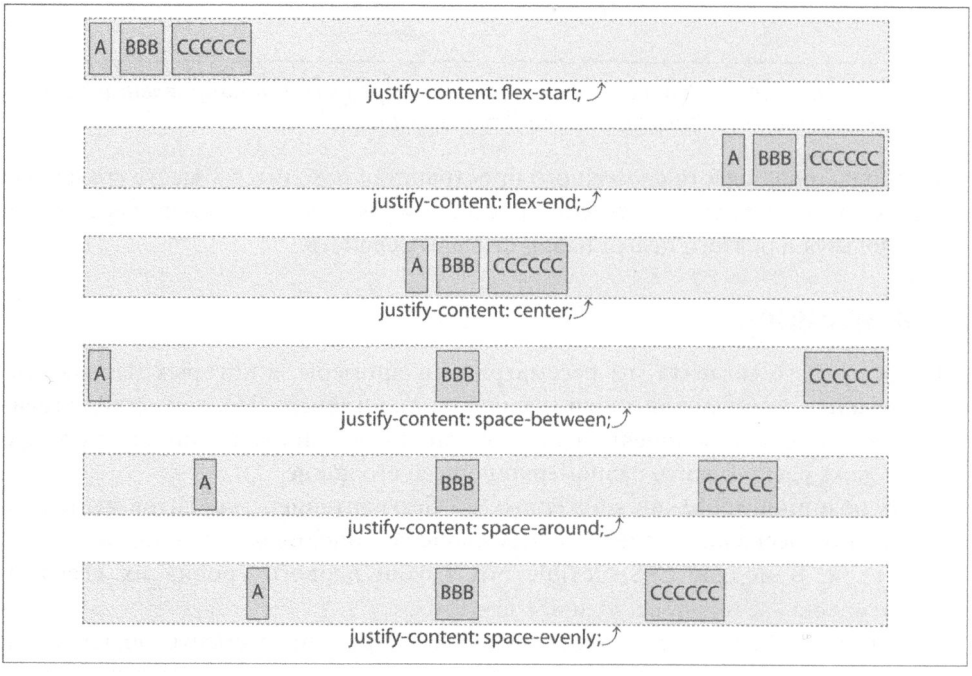


Рис. 12.18. Шесть способов распределения flex-элементов, устанавливаемых свойством `justify-content`

Значение `flex-start`, передаваемое свойству `justify-content` по умолчанию, указывает группировать элементы у начальной точки главной оси. При передаче свойству `justify-content` значения `flex-end` элементы будут группироваться у ее

конечной точки. Значение `center` определяет выравнивание элементов по центру ряда (колонки).

С помощью значения `space-between` flex-элементы равномерно распределяются вдоль главной оси контейнера через равные промежутки свободного пространства. При этом первый элемент выравнивается по начальной, а последний — по конечной точке главной оси. Подобным образом располагаются flex-элементы и при добавлении к их flex-контейнеру свойства `justify-content` со значением `space-around`, за тем лишь исключением, что теперь перед первым элементом и после последнего элемента образованной последовательности добавляется пустое пространство, размер которого вдвое меньше интервала между двумя соседними flex-элементами (как если бы они снабжались отступами одинаковой ширины). Значение `space-evenly` указывает распределять flex-элементы так, чтобы уравнять расстояние между любыми двумя соседними элементами, а также перед первым элементом последовательности и после последнего элемента. Во втором случае ширина отступов перед первым и после последнего flex-элемента должна совпадать с расстоянием между соседними элементами этого же ряда (колонки).

Свойство `justify-content` определяет не одно только расстояние между flex-элементами. Если гибкие элементы не переносятся на новый ряд, но выступают за края flex-контейнера, то свойство `justify-content` будет оказывать влияние на направление их смещения, как показано на рис. 12.19.

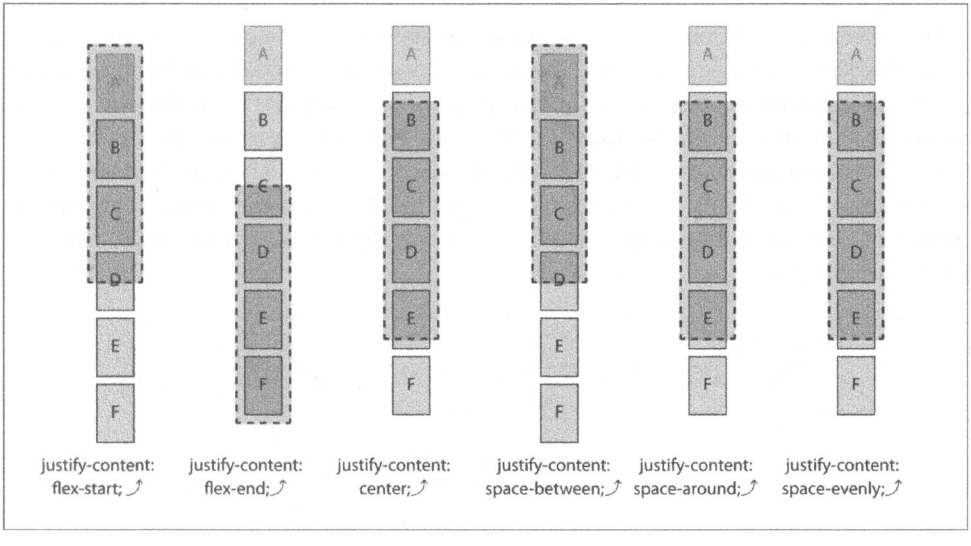


Рис. 12.19. Смещение flex-элементов за пределы контейнера при разных значениях стилевого свойства `justify-content`

Детально рассмотрим каждый из случаев.

Объявление `justify-content: flex-start`, применяемое по умолчанию, указывает смещать гибкие элементы по направлению к начальной точке главной оси контейнера, с которой совмещается первый элемент ряда (колонки). Каждый последующий элемент располагается впритык к предыдущему — последовательность разрывается

в конечной точке главной оси переносом остальных flex-элементов на новый ряд (колонку) или заканчивается за его пределами на последнем элементе. Расположение начальной точки главной оси напрямую зависит от значения свойства `flex-direction` и направления письма (см. раздел "Оси flex-контейнера"). Следовательно, при объявлении `flex-wrap: nowrap` как со значением по умолчанию, так и в явном виде все элементы, не помещающиеся во flex-контейнер, будут выходить за его пределы со стороны конечной точки главной оси (рис. 12.20).

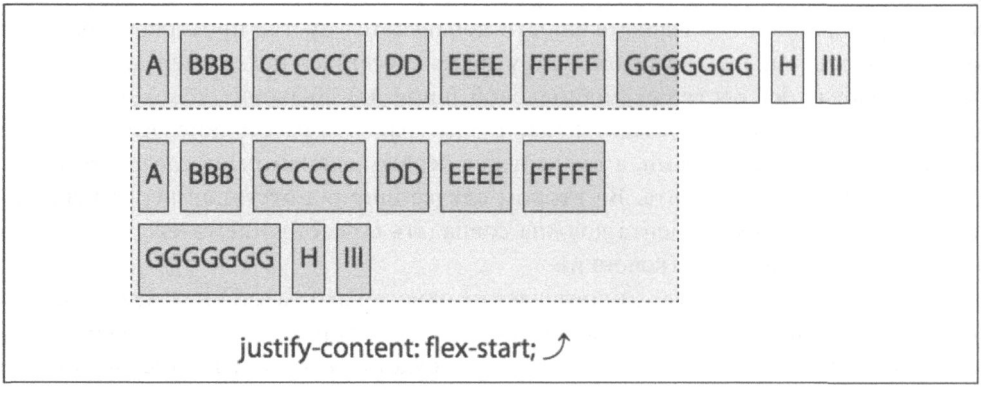


Рис. 12.20. Выравнивание элементов во flex-контейнере

При передаче свойству `justify-content` значения `flex-end` все flex-элементы будут смещаться к конечной точке главной оси контейнера, по которой выравнивается последний элемент последовательности. Если последовательность flex-элементов не переносится на новый ряд (колонку) и настолько длинная, что не помещается в контейнере, то она будет выступать за его пределы со стороны начальной точки главной оси, как показано на рис. 12.21. В случае переноса flex-элементов на следующий ряд дополнительное свободное пространство также образуется со стороны начальной точки главной оси.

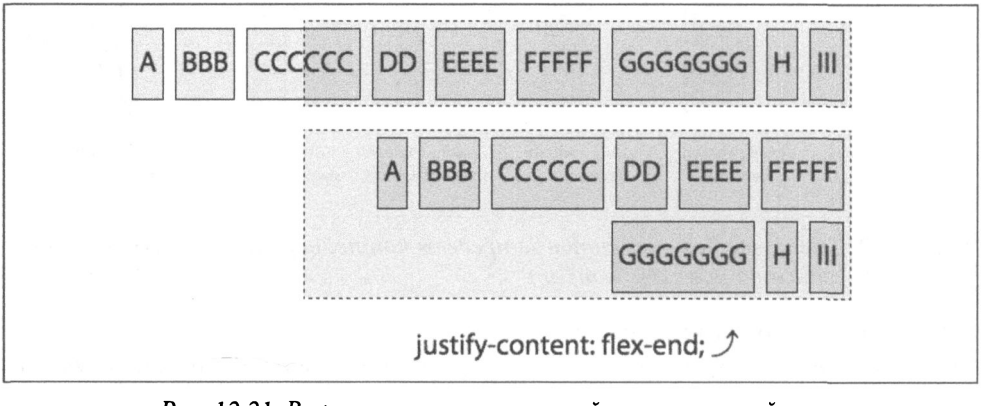


Рис. 12.21. Выравнивание по конечной точке главной оси

В случае объявления `justify-content: center` flex-элементы группируются в центральной части ряда (колонки), не примыкая ни к начальной, ни к конечной точке главной оси. Если последовательность элементов настолько длинная, что не помещается в контейнере и в нем запрещены переносы, то "лишние" flex-элементы будут равномерно выступать за его пределы сразу в обоих направлениях (см. второй пример рис. 12.22). При размещении flex-элементов в нескольких рядах (колонках) дополнительное свободное пространство будет равномерно добавляться со стороны как начальной, так и конечной точки главной оси.

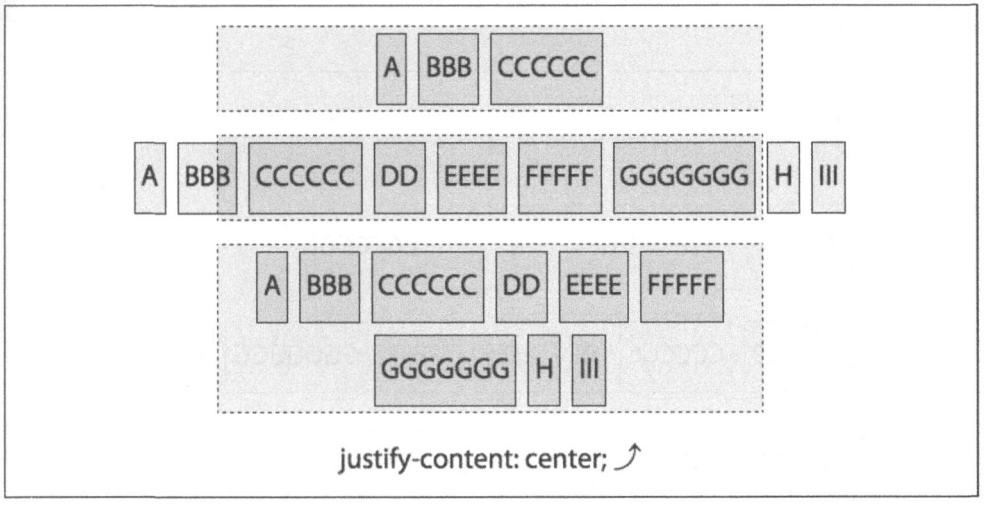


Рис. 12.22. Выравнивание flex-элементов по центру гибкого контейнера

Если к flex-контейнеру применить объявление `justify-content: space-between`, то первый гибкий элемент ряда (колонки) будет привязан к начальной точке, а последний элемент — к конечной точке главной оси. Все остальные элементы будут распределены между этими точками через одинаковые промежутки свободного пространства. При размещении в несколько рядов flex-элементы будут выравниваться указанным способом в каждом из них независимо. Если flex-элементов всего три, то расстояние между первым и вторым элементами будет в точности равно расстоянию между вторым и третьим элементами. При этом в начале и в конце ряда (колонки) — перед первым и после третьего элемента — дополнительное свободное пространство не образуется (второй пример на рис. 12.23). Важно понимать, что в случае добавления во flex-контейнер всего одного гибкого элемента, пусть и выступающего за его пределы в отсутствие переносов, значение `space-between` указывает выравнивать его по начальной точке главной оси. Следовательно, единственный гибкий элемент будет всегда выравниваться по началу главного размера, а не по его центру, как может показаться вначале.

Объявление `justify-content: space-between` обеспечивает равенство интервалов между flex-элементами только в пределах отдельно рассматриваемого ряда. Flex-элементы следующего ряда могут размещаться через совершенно иные

промежутки свободного пространства. При переносе их в другие ряды может возникнуть ситуация, когда в последнем ряду контейнера будет находиться намного меньше гибких элементов, чем содержится в предыдущих рядах. Первый из таких элементов будет выравниваться по начальной точке, а последний — по конечной точке главной оси. Все остальные элементы будут равномерно распределяться через одинаковые интервалы, как показано в последнем примере на рис 12.23. В данном случае по начальной точке главной оси выравниваются элементы A и G, а по ее конечной точке — элементы F и I. В каждом из рядов все элементы расположены через одинаковые промежутки свободного пространства, но у первого ряда они заметно уже, чем у второго.

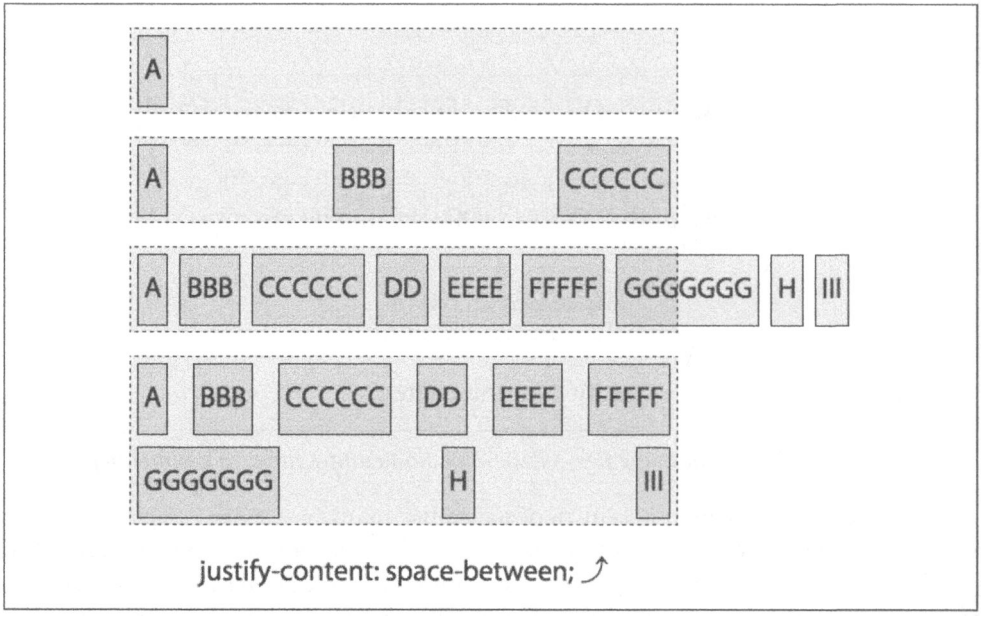


Рис. 12.23. Распределение flex-элементов через равные промежутки свободного пространства

Следующее объявление, `justify-content: space-around`, позволяет равномерно распределить flex-элементы по главному размеру контейнера так, как если бы все они имели несхлопывающиеся отступы одинакового размера (в направлении главной оси). При таком позиционировании интервал между любыми двумя соседними элементами вдвое превышает размер свободного пространства между начальной точкой главной оси и первым элементом или последним элементом и конечной точкой оси (рис. 12.24).

При размещении flex-элементов в нескольких рядах (колонках) величина отступов элементов зависит от общего количества свободного пространства в каждом ряду. Несмотря на равенство интервалов между элементами любого отдельно рассматриваемого ряда, в каждом из них он будет своим, как показано в последнем примере на

рис. 12.24. Как видите, расстояние между элементами A и B вдвое больше расстояния от начальной точки главной оси до элемента A, но оно намного меньше расстояния между элементами G и H следующего ряда, которое, в свою очередь, вдвое больше расстояния от начальной точки главной оси до элемента G.

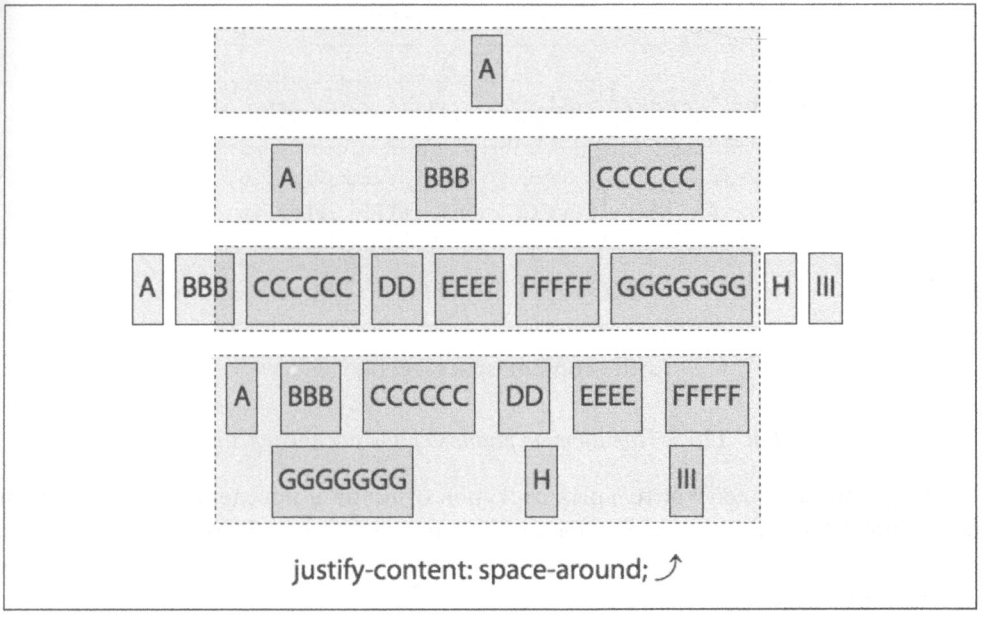


Рис. 12.24. Распределение flex-элементов через одинаковые отступы

В отсутствие переносов все не помещающиеся в контейнер элементы будут выступать по обе его стороны в направлении главного размера, как и в случае передачи свойству `justify-content` значения `center`.

Объявление `justify-content: space-evenly` предполагает распределение flex-элементов так, чтобы расстояние между любыми двумя соседними элементами, а также перед первым элементом и после последнего элемента было одинаковым. Для расчета интервала между flex-элементами пользовательский агент делит общую ширину свободного пространства flex-контейнера на число, превышающее на единицу количество гибких элементов в нем (т.е. при пяти flex-элементах все свободное пространство разделяется на шесть одинаковых интервалов). Интервал между соседними элементами точно такой же, как и перед первым элементом и после последнего элемента flex-контейнера. Элементы размещаются так, как если бы все они имели несхлопывающиеся отступы одинакового размера, но, в отличие от предыдущего случая, интервал перед первым элементом и после последнего элемента в точности равен интервалам между любыми двумя соседними гибкими элементами (рис. 12.25).

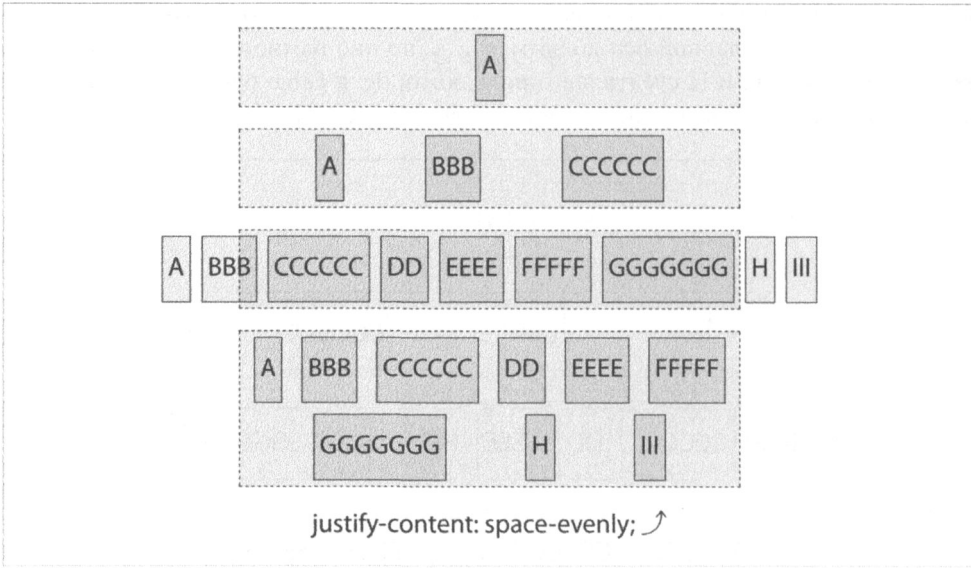


Рис. 12.25. Полностью равномерное распределение

При добавлении действительных отступов отличия в поведении flex-элементов, заключенных в контейнер, выравнивание которого устанавливается ключевыми словами `center`, `space-around`, `space-between` и `space-evenly`, становятся более очевидными (рис. 12.26).

На момент написания книги ключевое слово `space-evenly` было исключено из спецификации CSS Flexible Box Module и перемещено в модуль CSS Box Alignment. Но поскольку flex-контейнеры подпадают под требования спецификации CSS Box Alignment, оно так или иначе поддерживается спецификацией CSS Flexible Box (большинство браузеров прекрасно обрабатывает их в любом случае).

Примеры использования свойства `justify-content`

Для того чтобы получить представление об области применения значения по умолчанию (`flex-start`) свойства `justify-content`, создадим простую навигационную панель с гиперссылками, выровненными по ее левому краю. Изменив исходное объявление на `justify-content: flex-end`, можно сгруппировать гиперссылки у правого края панели (в документе, введенном на английском языке).

```
nav {
   display: flex;
   justify-content: flex-start;
}
```

Не забывайте, что свойство `justify-content` применяется к flex-контейнеру, а не его элементам. Если по ошибке объявить его для гиперссылок, например `nav a {justify-content: flex-start;}`, то эффект выравнивания наблюдаться не будет.

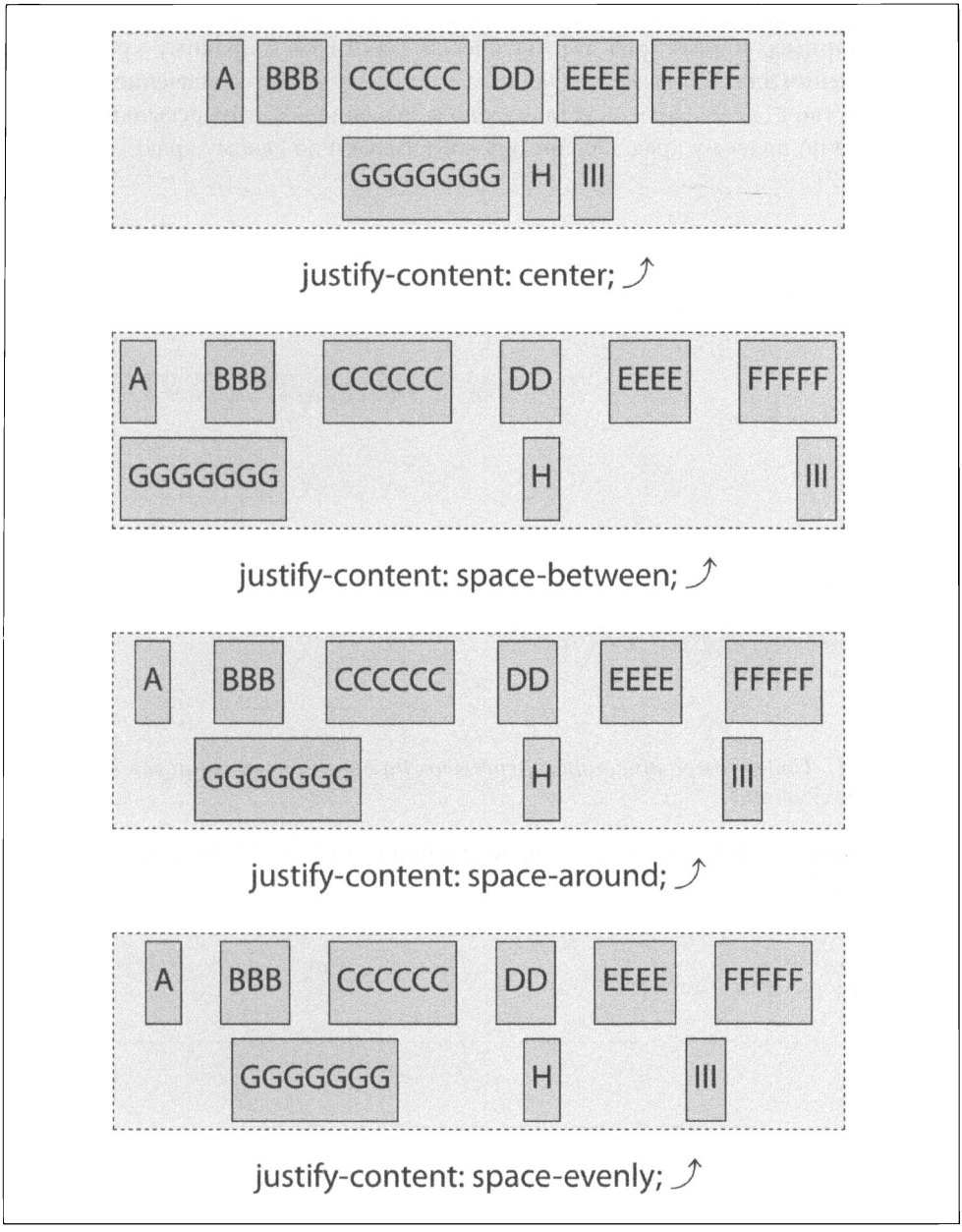


Рис. 12.26. Распределение одних и тех же элементов при выравнивании с помощью ключевых слов `center`, `space-between`, `space-around` *и* `space-evenly`

Основное преимущество свойства `justify-content` заключается в отсутствии необходимости редактировать его значение при переходе документа к языку с другим направлением письма, например справа налево. В исходном случае элементы группируются в начальной точке главной оси, которая для английского языка находится

у левого края навигационной панели, а после перехода к языку с иной системой письма, например ивриту, она автоматически сместится к правому краю панели. При объявлении для flex-элемента свойства `justify-content` со значением `flex-end` (когда свойство `flex-direction` установлено в значение `row`) гиперссылки будут выравниваться по правому краю для английского языка и по левому краю — для иврита (рис. 12.27).

Рис. 12.27. Способ выравнивания гиперссылок на панели навигации зависит от языка документа

Для центрирования содержимого панели навигации (рис. 12.28) применяется следующий код CSS.

```
nav {
   display: flex;
   justify-content: center;
}
```

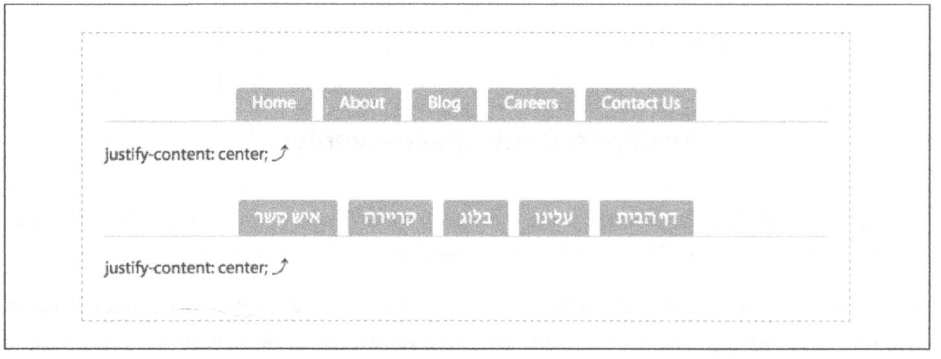


Рис. 12.28. Изменение раскладки элементов при передаче свойству `justify-content` значения `center`

Выравнивание flex-элементов вдоль поперечной оси

В то время как свойство `justify-content` определяет способ выравнивания flex-элементов вдоль главной оси контейнера, их распределение вдоль его поперечной оси устанавливается с помощью свойства `align-items`. Оно, как и свойство `justify-content`, применяется не к flex-элементам, а к их контейнеру.

align-items

Значение	`flex-start` | `flex-end` | `center` | `baseline` | `stretch`
Начальное значение	`stretch`
Применяется	Flex-контейнеры
Вычисляется	Согласно определению
Наследуется	Нет
Анимируется	Нет

Это свойство позволяет одновременно смещать все flex-элементы контейнера к начальной, конечной или центральной точке поперечной оси в пределах их рядов. Во многом оно выполняет такие же действия, как и свойство `justify-content`, но перпендикулярно главной оси контейнера. Под его воздействие попадают даже flex-элементы, представляющие анонимное содержимое.

Обычно свойство `align-items` применяется для смещения всех flex-элементов одного ряда (колонки) к одному из краев в поперечном направлении или же растягивания элементов к обоим краям. Оно также позволяет разместить элементы в поперечном направлении строго по центру ряда (колонки). Свойство принимает одно из таких значений: `flex-start`, `flex-end`, `center`, `baseline` или `stretch` (по умолчанию). Результат их использования показан на рис. 12.29.

Наряду со свойством `align-items`, устанавливающим способ выравнивания сразу всех flex-элементов контейнера, в спецификации представлено специальное свойство выравнивания индивидуальных гибких элементов: `align-self`. Подробно оно рассматривается в следующем разделе.

Согласно рис. 12.29, существует несколько основных способов выравнивания flex-контейнеров в поперечном направлении: смещение к начальной или конечной точке поперечной оси, расположение по центру ряда (колонки) либо растягивание к его противоположным краям. Значение `baseline` указывает выравнивать базовые линии flex-элементов. При таком способе выравнивания базовые линии гибких элементов располагаются на минимально возможном расстоянии от начальной точки поперечной оси.

Каждый из описанных выше способов выравнивания предельно точно указывает положение flex-элементов, размещаемых в контейнере в один ряд. Чтобы понять, как осуществляется выравнивание многострочного содержимого flex-контейнера, рассмотрим следующий пример, представленный таким CSS-кодом.

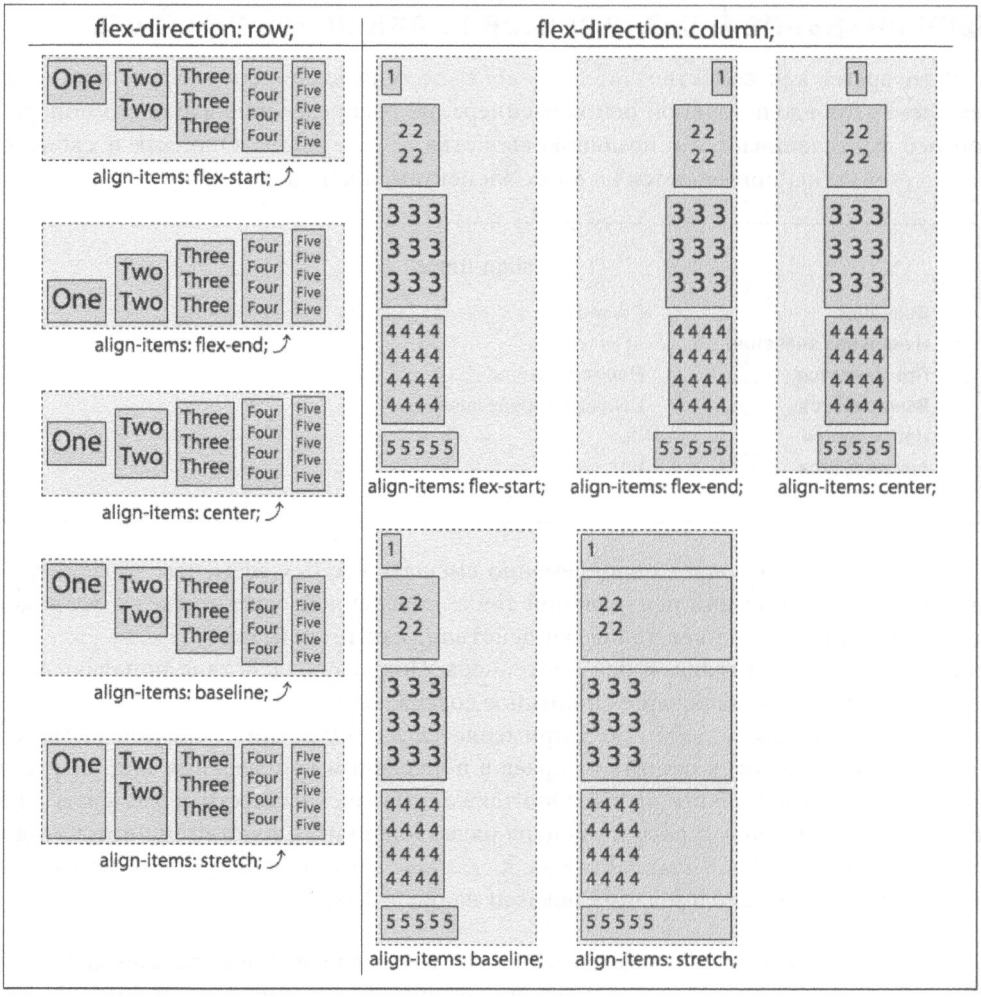


Рис. 12.29. Способы выравнивания flex-элементов, организованных в ряды и колонки, в направлении поперечной оси гибкого контейнера

```
flex-container {
   display: inline-flex;
   flex-flow: row wrap;
   border: 1px dashed;
}
flex-item {
   border: 1px solid;
   margin: 0 10px;
}
.C, .H {
   margin-top: 10px;
}
.D, .I {
```

```
    margin-top: 20px;
}
.J {
    font-size: 3rem;
}
```

Для лучшего понимания структуры flex-контейнера соседние ряды flex-элементов разграничены цветными горизонтальными линиями. С этой целью к элементам C, H, D и I добавлены не только верхняя и нижняя, но и боковые границы. Такое форматирование элементов никоим образом не сказывается на их выравнивании с помощью свойства `align-items`. Элементу J задан шрифт заметно большего, чем у остальных элементов, размера, за счет чего он имеет бо́льшую высоту строк, что оказывает влияние на выравнивание flex-элементов текущего ряда по базовым линиям.

Результат поперечного выравнивания, выполняемого с помощью объявления `align-items: stretch` (значение по умолчанию), для многострочных flex-элементов, расположенных во flex-контейнере в несколько рядов, показан на рис. 12.30.

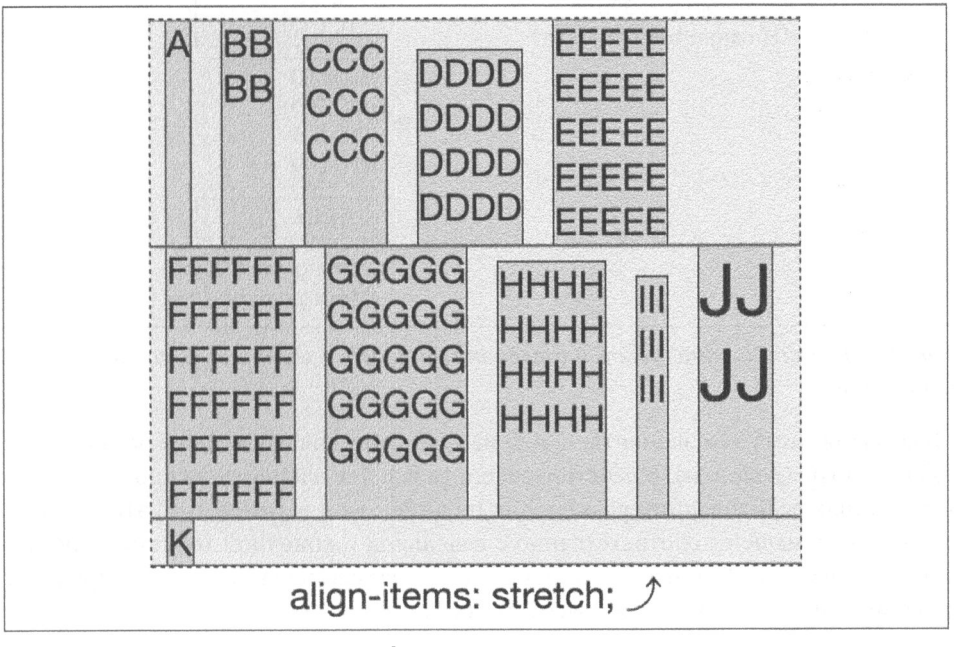


Рис. 12.30. Растягивания flex-элементов в поперечном направлении

Ключевое слово `stretch`, как предполагает его название, обязывает пользовательский агент растягивать все гибкие элементы ряда (колонки) до размера элемента с наибольшей высотой (шириной). Рассмотрим, как это работает. Значение `stretch` увеличивает размер элементов до 100% от высоты ряда или ширины колонки только в случае автоматической его установки, выполняемой по умолчанию. Если же размер flex-элементов объявляется в явном виде с помощью свойства `min-height`, `min-width`, `max-height`, `max-width`, `width` или `height`, то в поперечном направлении они растягиваться не будут.

Иными словами, при растягивании в поперечном направлении начальные и конечные точки flex-элементов и ряда, в который они помещены, совпадают. При этом исходный размер сохраняет только flex-элемент с наибольшей высотой (шириной), а остальные элементы увеличиваются до его размера.

В поперечный размер растягиваемых элементов включается ширина отступов: с начальной и конечной точками поперечного размера ряда совмещаются внешние края соответствующих отступов flex-элементов. Данный эффект наблюдается у элементов C и D (рис 12.31).

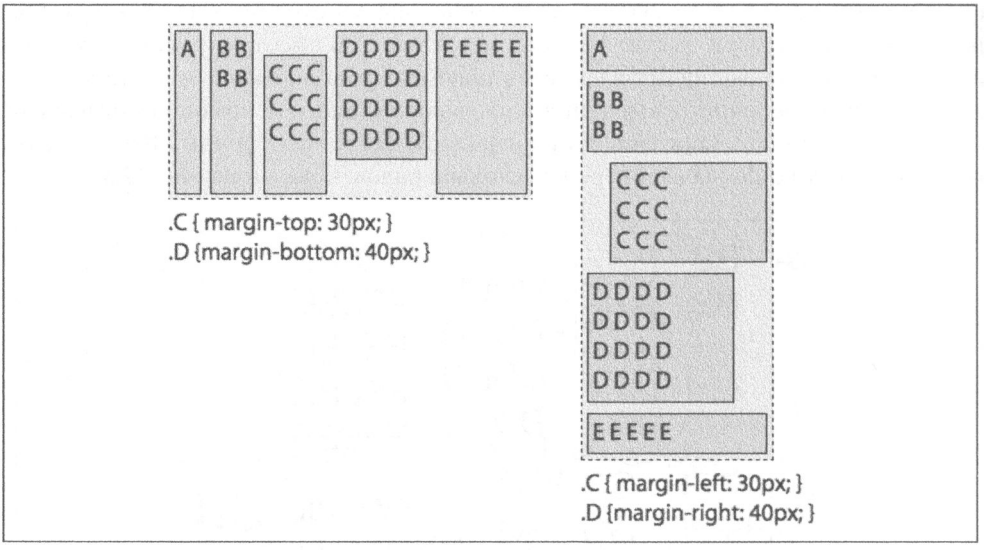


Рис. 12.31. Влияние отступов на выравнивание flex-элементов в поперечном направлении

Благодаря отступам элементы C и D выглядят несколько меньше остальных flex-элементов ряда (колонки). В действительности все flex-элементы каждого контейнера имеют одинаковый внешний размер. Как нижние, так и верхние края этих элементов полностью совмещены соответственно с начальной и конечной точками поперечной оси ряда (колонки). Как известно, поперечный размер ряда (колонки) определяется flex-элементом с максимальной высотой (шириной).

Высота или ширина ряда определяется размером наибольшего ее flex-элемента. Именно поэтому в примере, приведенном на рис. 12.30, высота ряда, включающего один только элемент K, намного меньше, чем высота всех предыдущих рядов.

Выравнивание по началу, концу и центру ряда (колонки)

Выравнивание flex-элементов по начальной и конечной точкам, а также по центру поперечного размера ряда выполняется достаточно прямолинейно, хотя и требует объявления в свойстве `align-items` разных значений.

Ключевое слово `flex-start` указывает совмещать начальные точки поперечного размера каждого из flex-элементов с началом ряда. Начальная точка flex-элемента

находится у внешнего края одного из ее отступов. Наличие отступов визуально отодвигает flex-элемент от начала ряда (колонки), как показано на рис. 12.32 для элементов C, D, H и I.

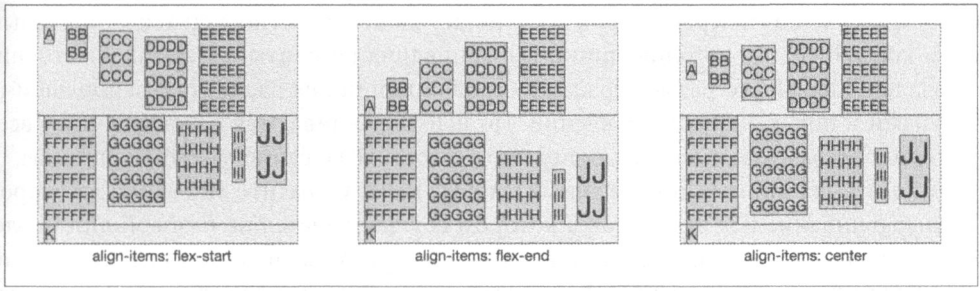


Рис. 12.32. Выравнивание flex-элементов по началу, концу и центру ряда

Объявление `align-items: flex-end` обязывает выравнивать конечные точки flex-элементов по конечной точке своего ряда, как показано во втором примере на рис. 12.32. Все выравниваемые элементы имеют нулевой нижний отступ, поэтому, в отличие от остальных примеров, ни один из них не нарушает общую линию выравнивания.

В третьем примере на рис. 12.32 flex-элементы выравниваются с помощью объявления `align-items: center`. В результате его применения они выравниваются своей центральной частью по средней линии поперечного размера ряда. Средняя линия определяется как место положения точек, равноудаленных от начала и конца ряда в поперечном направлении. Центр flex-элемента находится на одинаковом расстоянии от внешних краев противоположных отступов (не забывайте, что отступы гибких элементов не схлопываются). Следовательно, элементы C, D, H и I только выглядят не выровненными по центру, хотя на самом деле являются таковыми, поскольку снабжены несимметричными отступами. Их центр, равноудаленный от верхнего и нижнего внешних краев элемента, располагается в точности там же, где проходит центральная линия ряда.

В документах с направлением письма справа налево и слева направо центр flex-элементов, подпадающих под воздействие объявления `flex-direction: row` или `flex-direction: row-reverse`, находится на одинаковом расстоянии от внешних краев верхнего и нижнего отступов. При этом центр flex-элементов, к которым применено объявление `flex-direction: column` или `flex-direction:column-reverse`, равноудален от внешних краев боковых отступов.

В случаях, когда поперечного размера контейнера недостаточно для размещения в нем отдельных flex-элементов целиком, они могут выступать за его пределы. Направление выступания элементов вдоль поперечной оси определяется свойством `align-content`, а не `align-items`, как объясняется в разделе "Выравнивание содержимого". Свойство `align-items` отвечает за выравнивание элементов только в пределах ряда flex-контейнера и не предотвращает его выход за внешние края контейнера.

Выравнивание базовых линий

Значение `baseline` определяет более сложный способ выравнивания, чем в рассмотренных выше случаях. Объявление его для flex-контейнера вызывает выравнивание базовых линий первых строк всех гибких элементов, относящихся к отдельным рядам контейнера. Положение линии выравнивания рассчитывается как расстояние от начала поперечного размера ряда до базовой линии flex-элемента с наибольшими отступами в поперечном направлении. По ней выравниваются базовые линии всех остальных элементов, имеющих поперечные отступы заведомо меньшей ширины.

Взгляните на пример, приведенный на рис. 12.33. Обратите внимание на второй ряд, выравнивание flex-элементов в котором осуществляется по базовой линии элемента J. Благодаря шрифту размером `3em` его первая базовая линия расположена на большем удалении от начала ряда, чем базовые линии первых строк остальных элементов. Таким образом, только он один примыкает своим верхним краем к начальной точке поперечного размера flex-контейнера. Все остальные элементы размещаются в ряду так, что базовые линии их первых строк выравниваются по базовой линии первой строки элемента J (тонкая линия, проходящая через все flex-элементы ряда в горизонтальном направлении).

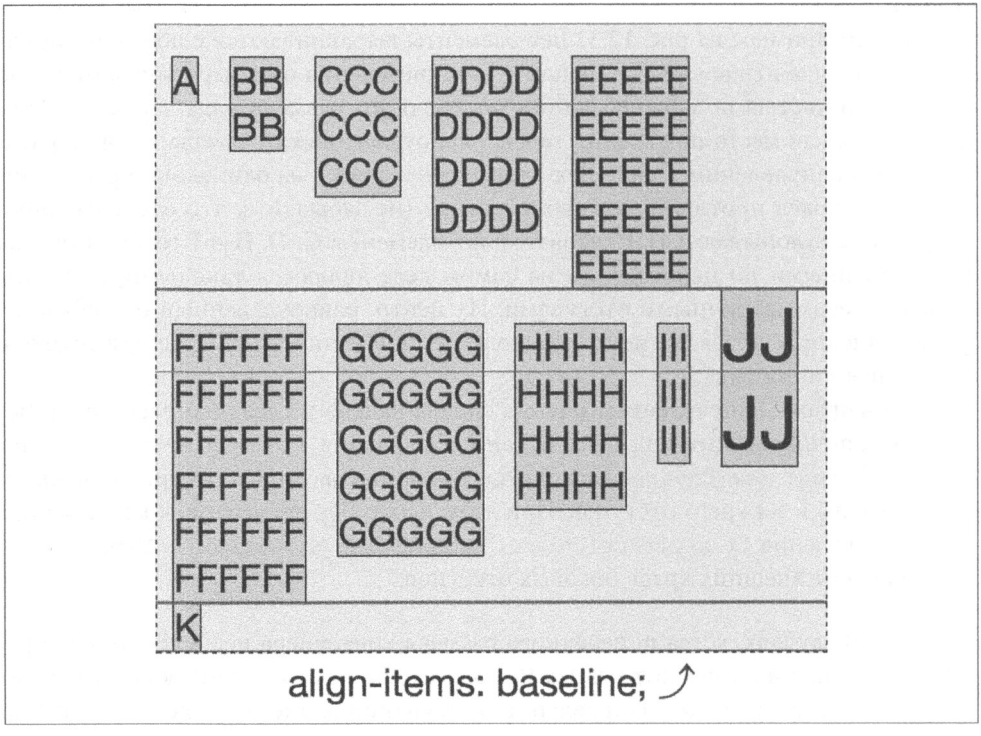


Рис. 12.33. Выравнивание базовых линий flex-элементов

Вернемся к рассмотрению первого ряда flex-контейнера. Каждый из элементов A, B, C, D и E выравнивается по начальной точке поперечного размера ряда, хотя и не

соприкасается с ней своим верхним краем. Причина тому — элемент D, единственный среди всех flex-элементов обладающий верхним отступом, равным 20 пикселям. При совмещении верхнего внешнего края элемента D с началом поперечного размера своего ряда базовая линия его первой строки будет располагаться ниже (благодаря отступу), чем базовые линии первых строк других элементов. Следовательно, именно она будет определять уровень, по которому выравниваются базовые линии всех остальных элементов первого ряда.

Во многих случаях ключевые слова `baseline` и `flex-start` определяют одинаковый способ выравнивания flex-элементов. В частности, если элемент D лишить отступа, то базовая линия его первой строки будет подтянута вверх, а сам он будет располагаться так же, как и в случае применения значения `flex-start`, т.е. прижатым к началу поперечного размера ряда. Из этого можно заключить, что отличия в способах выравнивания, определяемых ключевыми словами `baseline` и `flex-start`, проявляются только при добавлении к flex-элементам разных отступов, границ и полей.

Значение `baseline` гарантированно обеспечивает такой же способ выравнивания flex-элементов, как и ключевое слово `flex-start`, в случае размещения их базовых линий параллельно поперечной оси контейнера. В примере, показанном на рис. 12.33, это равнозначно применению к flex-контейнеру объявления `flex-direction: column`. В результате выполнения такого форматирования и поперечная ось контейнера, и базовые линии элементов, содержимое которых вводится слева направо, будут располагаться горизонтально. При таком позиционировании flex-элементов говорить об их смещении относительно начала поперечного размера (влево) в процессе выравнивания не приходится. Исходя из этого, значение `baseline` будет обрабатываться пользовательским агентом так же, как и значение `flex-start`.

Дополнительные замечания

Чтобы изменить способ выравнивания не всех, а только отдельных flex-элементов одного ряда, для них нужно объявить стилевое свойство `align-self`. Оно имеет такие же значения, как и свойство `align-items`, и описано в разделе "Flex-элементы".

С помощью свойства `align-self` нельзя изменить выравнивание анонимных элементов (представляющих текстовые узлы, дочерние по отношению к flex-контейнеру). Их свойство `align-self` всегда получает такое же значение, как и свойство `align-items`, объявленное для их контейнера.

В приведенных выше примерах, описывающих назначение свойства `align-items`, высота flex-контейнера автоматически подстраивается под размер содержимого. Такое поведение обеспечивается объявлением `height: auto`, применяемым к контейнеру по умолчанию. Оно обязывает flex-контейнер увеличиваться до размеров содержимого, хотя во всех приведенных выше примерах и контейнеры, и ряды, содержащие flex-элементы, имеют одинаковую высоту.

У контейнеров с фиксированным поперечным размером (в данном случае — высотой) может наблюдаться нехватка пространства для размещения всех вложенных элементов, или, наоборот, в их нижней части может образоваться большое количество свободного места, лишенного содержимого. Чтобы компенсировать нехватку или избыток свободного пространства, нужно изменить выравнивание рядов

flex-контейнера, воспользовавшись свойством `align-content`. Это последнее рассматриваемое нами свойство, определяющее форматирование сразу всего flex-контейнера (в противоположность свойствам форматирования flex-элементов). Учтите, что оно изменяет выравнивание рядов только в многорядных контейнерах.

Свойство `align-self`

Забегая немного наперед, хочется познакомить вас со свойством `align-self`, позволяющим отменять эффект применения свойства `align-items` к отдельным flex-элементам.

align-self

Значение	`auto` | `flex-start` | `lex-end` | `center` | `baseline` | `stretch`
Начальное значение	`auto`
Применяется	Flex-контейнеры
Процентное значение	Не поддерживается
Наследуется	Нет
Анимируется	Нет

Свойство `align-items` объявляется для flex-контейнера, а потому воздействует сразу на все его вложенные элементы. Свойство `align-self` позволяет изменить способ выравнивания, определенный свойством `align-items`, для любого flex-элемента своего контейнера. В каждом из пяти примеров, приведенных на рис. 12.34, четыре из пяти flex-элементов растягиваются до высоты родительского контейнера, что достигается благодаря установке свойства `align-items` в значение по умолчанию: `stretch`.

Установка свойства `align-self` в значение по умолчанию, `auto set`, указывает на наследственность способа выравнивания (в данном случае растягивания, обозначенного ключевым словом `stretch`) от значения свойства `align-items` всеми, кроме вторых, элементами каждого контейнера. Способ выравнивания вторых элементов каждого из flex-контейнеров определяется специальным образом с помощью ключевых слов, указанных в подписях к примерам.

Значение `flex-start` свойства `align-self` обозначает такое же действие, как и в случае передачи его свойству `align-items`: flex-элемент смещается в начальную точку поперечной оси контейнера. Значение `flex-end` обеспечивает привязку flex-элемента к конечной точке поперечной оси, значение `center` применяется для выравнивания элемента по центру поперечного размера, а ключевое слово `baseline` указывает совместить базовую линию целевого элемента с базовой линией наиболее низко расположенного элемента текущего ряда. Наконец, значения `auto` и `stretch` обеспечивают растягивание элемента до поперечного размера ряда (значение `stretch` передается свойству `align-items` по умолчанию). К такому же эффекту приводит объявление `align-self: inherit`.

Детально ключевые слова `flex-start`, `flex-end`, `center`, `baseline` и `stretch` были описаны в предыдущем разделе.

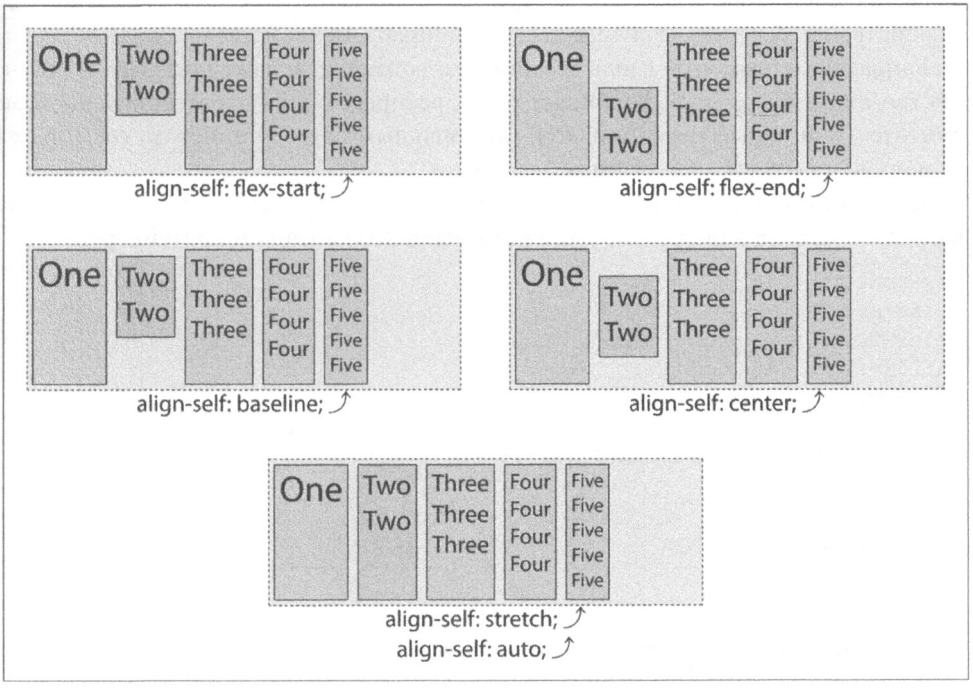


Рис. 12.34. Изменение выравнивания отдельного flex-элемента

Выравнивание содержимого

Стилевое свойство `align-content` позволяет выровнять ряды элементов вдоль поперечного размера контейнера, располагающего для этого достаточным количеством свободного пространства. В случае недостатка свободного места оно указывает направление выступа содержимого за края контейнера.

align-content

Значение	`flex-start` | `flex-end` | `center` | `baseline` | `space-between` | `space-around` | `space-evenly` | `stretch`
Начальное значение	`stretch`
Применяется	Многорядные flex-контейнеры
Вычисляется	Согласно определению
Наследуется	Нет
Анимируется	Нет

Это свойство применяется для перераспределения свободного пространства, располагаемого между и вокруг рядов элементов flex-контейнера. Несмотря на

подобность общей концепции и значений, выравнивание рядов с помощью свойства `align-content` выполняется иначе, чем выравнивание элементов в отдельном ряду, устанавливаемое свойством `align-items`.

Свойство `align-content` лучше всего сравнивать со свойством `justify-content`, выравнивающим элементы вдоль главной оси контейнера, даже несмотря на то, что свойство `align-content` выравнивает ряды, распределяя их вдоль поперечной оси. Разумеется, оно применяется только к многорядным flex-контейнерам, у которых не заблокирован перенос содержимого.

Следующий базовый пример призван продемонстрировать применимость свойства `align-content` для выравнивания flex-элементов в гибком контейнере.

```
.flex-container {
   display: flex;
   flex-flow: row wrap;
   align-items: flex-start;
   border: 1px dashed;
   height: 480px;
   background-image: url(banded.svg);
}
.flex-items {
   margin: 0;
   flow: 1;
}
```

На рис. 12.35 показан эффект объявления свойства `align-content` с каждым из шести поддерживаемых значений (с помощью приведенного выше кода) для одного и того же гибкого контейнера. В каждом случае выравниванию подлежат три ряда flex-элементов, верхние и нижние границы которых обозначены соответственно красными и синими горизонтальными линиями. Свободное пространство между и вокруг рядов представлено областями с полосатой заливкой.

При высоте 480 пикселей flex-контейнер имеет больший вертикальный размер, чем суммарная высота всех трех рядов. Предположим, что самые высокие элементы в каждом ряду — E, F и K — имеют вертикальный размер соответственно 150, 180 и 30 пикселей, а потому их суммарная высота составит 360 пикселей. Таким образом, каждый flex-контейнер снабжается областью свободного пространства высотой 120 пикселей в поперечном направлении.

Как видно на рис. 12.35, в пяти из шести описанных случаев применения свойства `align-content` свободное пространство располагается исключительно вне области рядов контейнера. Подобное поведение наблюдается при выравнивании элементов в пределах ряда с помощью таких же значений свойства `justify-content`, тем не менее выполняемого вдоль главной оси. А вот при передаче свойству `align-content` значения `stretch` свободное пространство равномерно распределяется между всеми рядами контейнера, увеличивая их поперечный размер до состыковки друг с другом.

В описанном выше примере flex-контейнера с поперечным размером 480 пикселей задача свойства `align-content` заключается в распределении свободного пространства высотой 120 пикселей между всеми рядами flex-элементов.

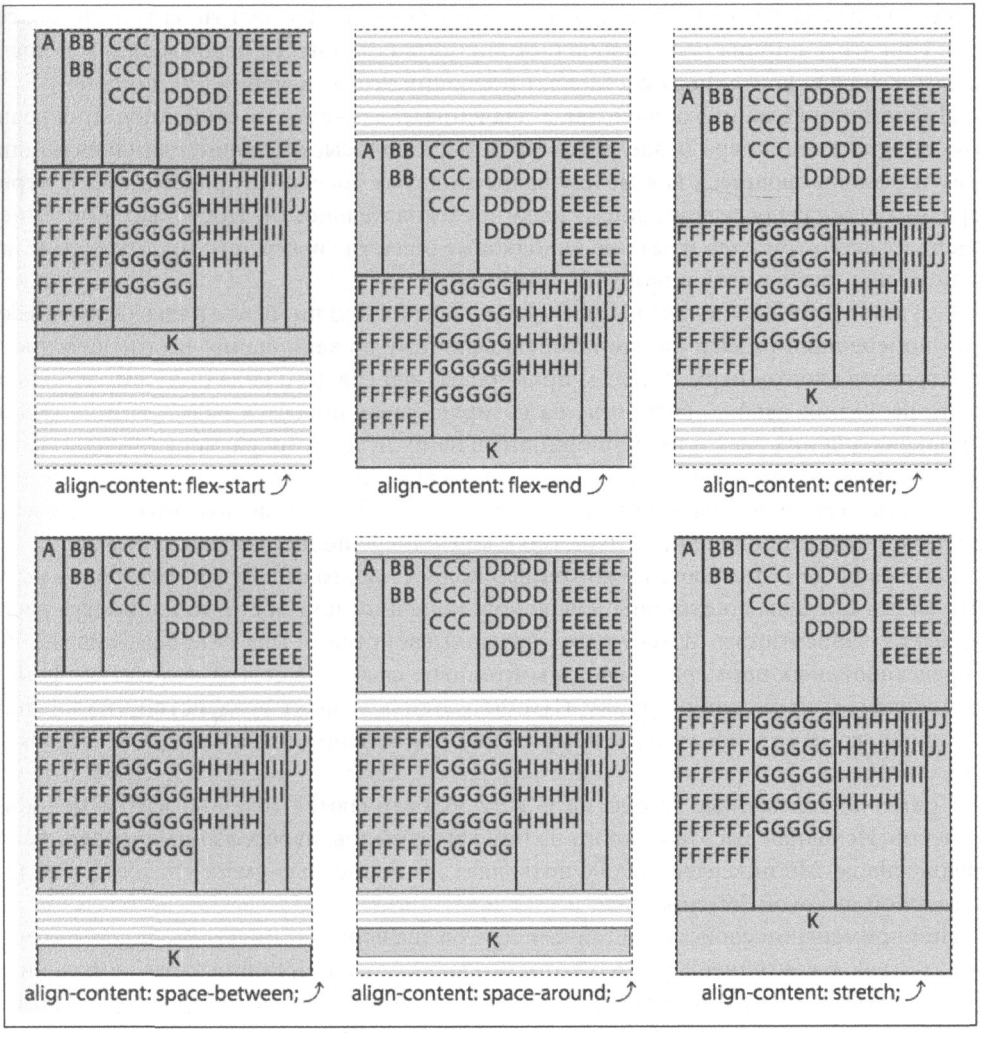


Рис. 12.35. Распределение свободного пространства при выравнивании рядов элементов с помощью свойства `align-content`

Как показано в трех верхних примерах на рис. 12.35, значение `flex-start` позволяет сместить все свободное пространство к конечной точке поперечной оси, значение `flex-end` указывает сместить его к ее начальной точке, а значение `center` обеспечивает его распределение между указанными точками поровну (полосы шириной 60 пикселей с каждой стороны).

Значение `space-between` предполагает равномерное распределение свободного пространства исключительно между рядами flex-контейнера. Поскольку их всего три, то оно будет разделено на две полосы, каждая шириной 60 пикселей. В случае передачи свойству `align-content` значения `space-around` свободное пространство равномерно распределяется вокруг каждой из рядов. Таким образом, для вычисления

интервала между рядами его поперечный размер нужно разделить на три (40 пикселей). При этом ширина полос свободного пространства перед первым рядом и после последнего ряда контейнера составляет половину этого значения (20 пикселей).

При установке данного свойства в значение `space-evenly` свободное пространство внутри контейнера разделяется на четыре полосы, имеющие одинаковую ширину и располагающиеся как между любыми двумя соседними рядами, так и перед первым рядом и после последнего ряда последовательности. Поскольку контейнер содержит всего три ряда и четыре одинаковые области свободного пространства, каждая из них будет иметь ширину 30 пикселей.

Результат применения значения `stretch` совершенно иной: все ряды увеличивают свой поперечный размер на одинаковую величину до заполнения всего свободного пространства контейнера. В нашем примере размер каждого ряда становится больше точно на 40 пикселей — величина, на которую увеличиваются размеры всех рядов, одинакова для них всех и не пропорциональна их исходному размеру (до растягивания). Эффект применения к flex-контейнеру свойства `align-content` со значением `stretch` показан в последнем примере на рис. 12.35. Легко заметить, что после растягивания рядов свободное пространство в контейнере полностью отсутствует.

Если ряды не помещаются в контейнер полностью, то они будут частично выступать за его пределы со стороны начальной, конечной или обеих точек поперечного размера в зависимости от текущего значения свойства `align-content` (рис. 12.36). В представленных примерах область контейнера снабжена серым полосатым фоном и заключена в пунктирную рамку. (Чтобы усилить эффект воздействия на контейнер свойства `align-content`, к нему также применено объявление `align-items: flex-start`.)

Примеры, показанные на рис. 12.35 и 12.36, отличаются только высотой flex-контейнеров. Исходная высота каждого из flex-контейнеров, изображенных на рис. 12.36, уменьшена до 240 пикселей, что не позволяет им полностью вместить все три ряда с суммарной высотой 360 пикселей.

При применении свойства `align-content` со значением `flex-start`, `space-between` или `stretch` к контейнеру, размер которого меньше суммарной высоты включенных в него рядов, они будут выступать за конечную точку его поперечной оси. При передаче этому свойству значения `flex-end` выступ рядов будет наблюдаться со стороны начальной точки поперечной оси. И только в случае применения значения `space-around` или `center` ряды будут выходить за пределы контейнера сразу в обоих направлениях — как за начальную, так и за конечную точку его поперечного размера.

Не забывайте, что значения свойства `align-content` привязаны к направлению поперечной оси контейнера. Если ее инвертировать, направив снизу вверх, то объявление `align-content: flex-start` будет указывать на выравнивание рядов по нижнему краю, поскольку теперь именно он будет совмещаться с начальной точкой поперечной оси flex-контейнера. Вследствие этого все не помещающиеся по высоте контейнера ряды будут выступать за его верхний край (конечную точку поперечной оси). При организации flex-элементов в виде колонок поперечная ось будет направлена горизонтально, ее начальная точка будет находиться у левого, а конечная точка края — у правого края flex-контейнера.

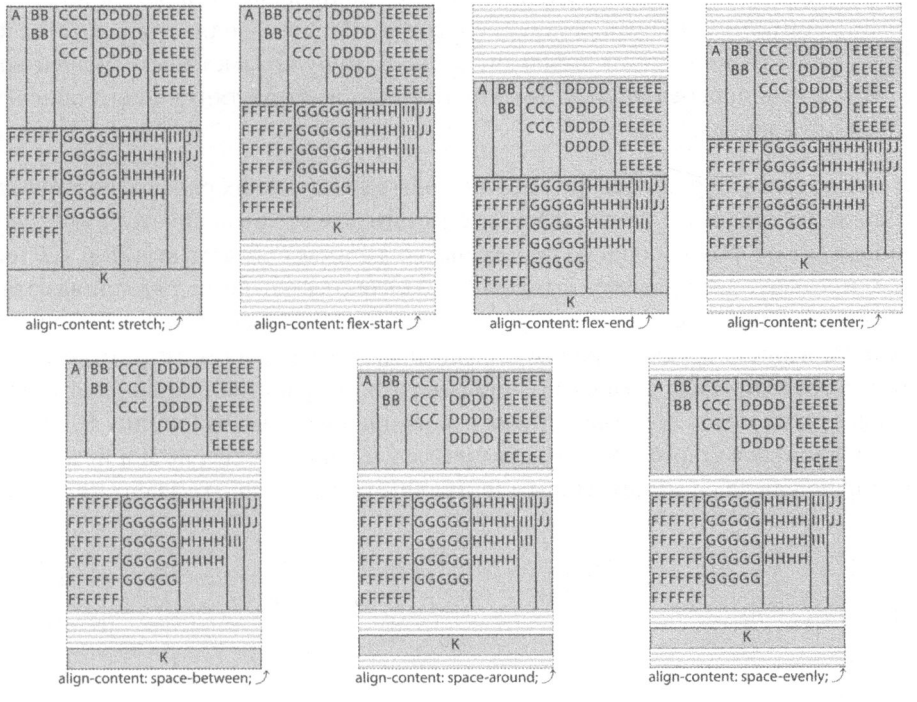


Рис. 12.36. Выход элементов за пределы flex-контейнера при выравнивании его рядов с помощью свойства `align-content`

Значения space-between и space-around

Рассматривая способы выравнивания рядов элементов во flex-контейнерах, особое внимание нужно уделить значениям `space-between` и `space-around`.

Объявление `align-content: space-between` обеспечивает равномерность распределения рядов вдоль поперечного размера flex-контейнера. Под равномерным распределением подразумевается расположение рядов на одинаковом расстоянии друг от друга, которое *не* пропорционально их поперечному размеру. Такой способ позиционирования предполагает привязку первого ряда к началу поперечной оси контейнера, выравнивание его последнего ряда по конечной точке поперечного размера и последующее равномерное распределение всех расположенных между ними рядов (если таковые существуют) до заполнения ими всего контейнера. Важно понимать, что ширина свободного пространства между рядами *не* зависит от их исходного размера. Ряды отстоят друг от друга на абсолютно одинаковые расстояния, даже несмотря на заметное отличие в их поперечных размерах. Более того, средний ряд — в случае включения во flex-контейнер непарного их количества — может и, скорее всего, будет располагаться не по центру его поперечной оси. Строгое центрирование среднего ряда будет наблюдаться только при равенстве размеров всех рядов flex-контейнера.

Под равномерное распределение подпадают только многорядные контейнеры. Если контейнер содержит всего один ряд, то значение `align-content` не приведет к сколь-нибудь заметному смешению его вдоль поперечной оси. Вместо этого он будет растянут до его поперечного размера.

Итак, при равномерном распределении любых два соседних ряда располагаются на одинаковом расстоянии друг от друга. Рассмотрим, как это выполняется в трехрядном контейнере, в котором свободное пространство занимает область высотой 120 пикселей (см. предыдущий пример). В таком flex-контейнере первый ряд привязывается к началу поперечной оси, а третий ряд выравнивается по его конечной точке. Следовательно, второй, средний, ряд будет располагаться между ними и отделяться от первого и третьего ряда полосами свободного пространства одинаковой ширины. Поскольку таких полос всего две, то каждая из них будет иметь ширину 60 пикселей. Первая полоса свободного пространства размещается между первым и вторым рядами, а вторая полоса — между вторым и третьим рядами, как показано на рис. 12.37.

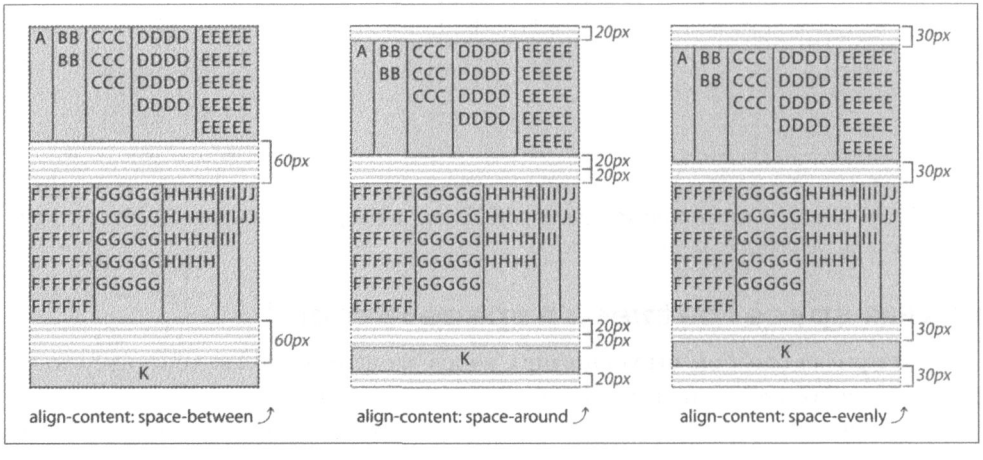


Рис. 12.37. Распределение свободного пространства между рядами элементов с помощью ключевых слов `space-between`, `space-around` и `space-evenly`

С помощью значения `space-around` ряды распределяются в контейнере через одинаковые промежутки свободного пространства, включая отступы перед первым рядом и после последнего ряда в поперечном направлении. Визуально это выглядит так, как если бы ряды, расположенные у начальной и конечной точек поперечной оси, снабжались несхлопывающимися отступами, равными по размеру ширине свободного пространства между ними самими. Пример такого способа распределения рядов flex-контейнера показан на рис. 12.37 (последний вариант).

К концу 2017 года названия ключевых слов, определяющие способ распределения рядов во flex-контейнере (`flex-start`, `flex-end` и др.), претерпели изменения, лишившись префикса `flex` (`start`, `end` и пр.). Такой способ именования согласуется с общей концепцией пространства имен

CSS, определяющего синтаксис свойств, которые отвечают за позиционирование и выравнивание элементов и их содержимого (например, `margin-start` и `padding-end`). Тем не менее у нового синтаксиса слишком слабая поддержка среди браузеров, чтобы утверждать о его состоятельности при написании кода стилевых правил. Именно поэтому мы будем придерживаться старого варианта именования обозначенных выше ключевых слов.

Большинство описываемых выше свойств (за исключением `align-self`) применялось к flex-контейнерам и всему их содержимому. В последующих разделах рассматриваются инструменты стилевого форматирования отдельных flex-элементов.

Flex-элементы

До этого момента гибкая верстка документа рассматривалась нами с глобальных позиций, предполагающих стилевое форматирование flex-элементов исключительно в рамках всего контейнера. Тем не менее спецификация CSS включает несколько свойств, предназначенных для индивидуального форматирования flex-элементов, которое выполняется отдельно от остальных гибких элементов, включенных в их контейнеры. С помощью таких свойств можно предельно точно расположить и выровнять любые flex-элементы, объявленные в документе.

Определение flex-элемента

Как вы уже знаете, для получения гибкого контейнера, или flex-контейнера, достаточно применить к целевому элементу, включающему дочерние узлы, объявление `display: flex` или `display: inlineflex`. Содержимое полученного таким образом flex-контейнера будет рассматриваться пользовательскими агентами как набор гибких элементов, или flex-элементов. К ним относятся все дочерние элементы контейнера, не пустые тестовые узлы, расположенные между дочерними элементами, и генерируемое содержимое. На рис. 12.38 проиллюстрирована ситуация представления отдельными flex-элементами каждого из символов тестовой строки, в том числе и пробела.

Рис. 12.38. Дочерние элементы становятся flex-элементами, а родительский элемент — flex-контейнером

Что касается текстовых узлов, то они становятся гибкими контейнерами лишь в случае включения текста, представленного не одними только символами пробелов. Такие узлы рассматриваются как анонимные flex-элементы одного уровня с остальными flex-элементами своего контейнера. Несмотря на наследование рабочих характеристик от flex-контейнеров, анонимные элементы не подпадают под стилевое оформление свойствами, применимыми к отдельным flex-элементам. Таким образом, их форматирование может осуществляться только свойствами, объявленными для всего flex-контейнера, но не отдельных гибких элементов. Поэтому в следующей разметке имеются только два "истинных" flex-элемента (`<strong>` и `<em>`), а текст `they're what's for` представляется в контейнере анонимным гибким элементом.

```
<p style="display: flex;">
   <strong>Flex items:</strong> they're what's for
      <em>&lt;br&gt;fast!</em>
</p>
```

Генерируемое содержимое, обозначаемое в коде ключевыми словами `::before` и `::after`, форматируется так же, как и регулярные flex-элементы. Все описанные в этом разделе свойства в равной степени применимы как к flex-элементам, так и к генерируемому содержимому, представляемому в объектной модели документа отдельным узлом.

Текстовые узлы, включенные во flex-контейнер и содержащие одни лишь символы пробелов, гибкими элементами не представляются, поскольку свойство `display` для них устанавливается в значение `none`, что наглядно продемонстрировано в следующем примере.

```
nav ul {
   display: flex;
}

<nav>
   <ul>
      <li><a href="#1">Ссылка 1</a></li>
      <li><a href="#2">Ссылка 2</a></li>
      <li><a href="#3">Ссылка 3</a></li>
      <li><a href="#4">Ссылка 4</a></li>
      <li><a href="#5">Ссылка 5</a></li>
   </ul>
</nav>
```

В этом коде свойство `display` со значением `flex` объявляется для элемента `ul`. Исходя из этого, flex-контейнер представляется как неупорядоченный список, а все его дочерние элементы (элементы списка) становятся flex-элементами. С семантической точки зрения такие flex-элементы заключаются в гибкие контейнеры и приобретают вид, не характерный для элементов списка. К тому же они обрабатываются совершенно не так, как того требуют элементы блочного уровня. Вместо этого они относятся пользовательским агентом к гибкому контексту форматирования. Свободное пространство вокруг элементов `li` и между ними, образованное символами

конца строки, табуляции и пробела, полностью игнорируется обработчиком, а гиперссылки не представляются гибкими элементами в явном виде, но заключаются в них на правах дочерних элементов.

Функциональные характеристики flex-элементов

Отступы flex-элементов, в отличие от регулярных элементов документа, не схлопываются. Стилевые свойства `float` и `clear` не оказывают на них должного эффекта — гибкие контейнеры не обтекаются соседним содержимым и не выравниваются по краям родительских элементов. (Тем не менее свойство `float` косвенно влияет на внешний вид гибкого контейнера, определяя вычисляемое значение свойства `display`) Рассмотрим такой пример.

```
aside {
   display: flex;
}
img {
   float: left;
}

<aside>
   <!-- Комментарий -->
   <h1>Заголовок</h1>

   <img src="images/foo.jpg" alt="Foo Master">
   Текст
</aside>
```

В представленной выше разметке flex-контейнером представляется элемент `aside`. В качестве его flex-элементов выступают заголовок, изображение и текстовые узлы, содержащие значимый текст. Как указывалось выше, комментарии и текстовые узлы, состоящие из одних пробелов, flex-элементами быть не могут. Так как изображение представляется гибким элементом, свойство `float` не оказывает на него сколь-нибудь заметного влияния.

Несмотря на то что изображения и текстовые элементы относятся к узлам строчного уровня, представляясь flex-элементами, они заключаются в контейнер блочного типа.

```
aside {
   display: flex;
   align-items: center;
}

<aside>
   <!-- Комментарий -->
   <h1>Заголовок</h1>

   <img src="images/foo.jpg" alt="foo master">
   Текст, содержащий <a href="foo.html">гиперссылку.</a> Еще текст
</aside>
```

Здесь применяется такая же, как и в предыдущем примере, разметка документа, за исключением описания в ней гиперссылки, помещенной между двумя текстовыми фрагментами. В данном случае flex-контейнер будет содержать целых пять гибких элементов. Комментарий и пустые текстовые узлы в их число не попадают. Таким образом, в качестве flex-элементов будут представляться заголовок, изображение, первый текстовый узел, гиперссылка и второй текстовый узел (рис. 12.39).

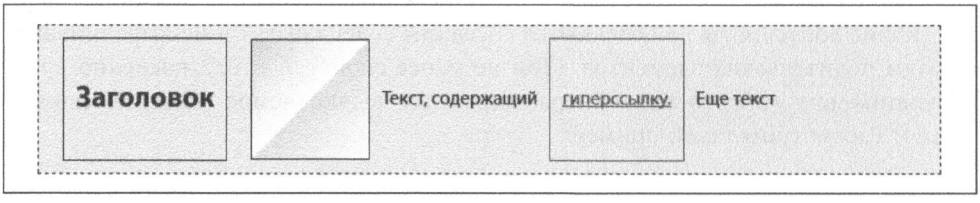


Рис. 12.39. Пять гибких элементов, вложенных в элемент `aside`

Текстовые узлы `Текст, содержащий` и `Еще текст` (не заключенные в рамку на рис. 12.39) сами по себе не являются flex-элементами, но заключаются в анонимные гибкие элементы. Гибкое стилевое форматирование можно отдельно устанавливать только для элементов заголовка, изображения и гиперссылки, являющихся регулярными узлами объектной модели документа. Анонимные элементы такой чести не удостаиваются и форматируются исключительно с помощью стилевых правил, применяемых ко всему контейнеру.

В дополнение к вышесказанному, на flex-элементы не оказывает влияние свойство `vertical-align`, хотя оно и определяет способ выравнивания текста внутри них. Иными словами, объявив для flex-контейнера свойство `vertical-align` со значением `bottom`, можно добиться выравнивания по нижнему краю ряда flex-контейнера только текста элементов, но не их самих. (Для этих целей предназначены свойства `align-items` и `align-self`.)

Абсолютное позиционирование

Если свойство `float` не оказывает воздействия на flex-элементы, то объявление `position: absolute` приводит к вполне ожидаемым последствиям. При абсолютном позиционировании flex-элементы, заключенные в гибкий контейнер, как и элементы регулярных типов, извлекаются из общего потока документа.

Начиная с этого момента они не подлежат гибкой верстке и не форматируются в общем потоке элементов документа. Несмотря на это, на них все еще оказывают влияние стили, применяемые к их flex-контейнеру так, как если бы он являлся для них обычным родительским элементом. К тому же абсолютно позиционируемые flex-элементы наследуют от своего контейнера любые стилевые свойства, поддерживающие такую возможность.

Стилевое оформление абсолютно позиционируемого flex-элемента в полной мере определяется как свойством `justify-content` его контейнера, так и объявленным ему самому свойством `align-self`. В частности, если к абсолютно позиционируемому дочернему элементу flex-контейнера применить объявление `align-self:`

center, то такой элемент будет выровнен по центру поперечной оси контейнера. При всем этом его положение также будет определяться отступами и значениями таких свойств, как top, bottom и т.п.

Свойство order (см. далее раздел "Свойство order") вполне ожидаемо определяет порядок представления абсолютно позиционируемого дочернего элемента flex-контейнера по отношению к другим вложенным в него элементам, хотя в большинстве случаев оно не оказывает заметного влияния на его позицию в документе.

Минимальная ширина flex-элемента

В первом примере на рис. 12.40 показан flex-контейнер, элементы которого размещены в одном ряду и не переносятся в другие ряды (согласно значению nowrap) даже после выхода за его пределы. Сжатия flex-элементов не происходит, поскольку их свойству min-width по умолчанию передается значение auto, а не 0. Несмотря на то что спецификация однозначно указывает автоматически сжимать все гибкие элементы, не помещающиеся во flex-контейнер, их размер остается неизменным благодаря нестандартному значению по умолчанию свойства min-width (в общем случае свойство min-width по умолчанию получает значение 0).

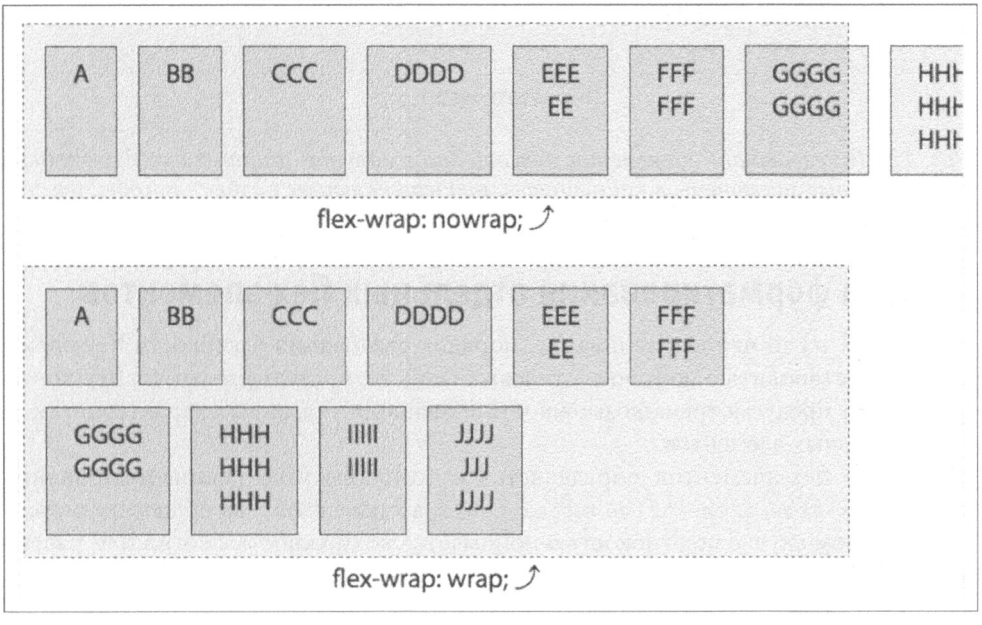


Рис. 12.40. Выход гибких элементов за пределы flex-контейнера при автоматической настройке их ширины

Для сжатия flex-элементов контейнера, лишенного возможности переноса данных в последующие ряды, свойству min-width нужно передать меньшее значение, чем вычисляемое значение ключевого слова auto (например, с помощью объявления min-width: 0). В подобных ситуациях flex-элементы могут сжиматься до меньшего размера, чем обусловливает их содержимое. При переносе flex-элементов в

последующие ряды они сужаются до размеров содержимого. Оба варианта заполнения flex-контейнеров показаны на рис. 12.41.

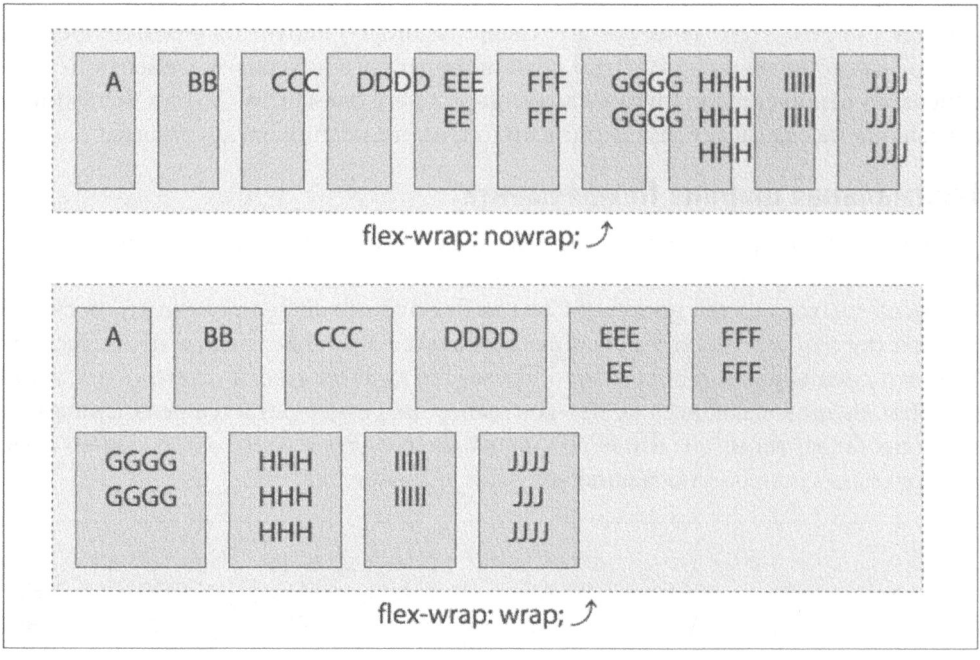


Рис. 12.41. Результат объявления нулевой минимальной ширины для flex-элементов, которые заключены в контейнеры, поддерживающие разные способы переноса содержимого

Свойства форматирования отдельных flex-элементов

Несмотря на то что выравнивание, порядок следования и гибкость flex-элементов можно установить с помощью стилевых свойств, применяемых к их flex-контейнерам, в CSS предусмотрена возможность изменения указанных характеристик для индивидуальных элементов.

Гибкость flex-элементов определяется с помощью узко специализированных свойств `flex-grow`, `flex-shrink` и `flex-basis`, а также свойства общего назначения `flex`. В данном случае под гибкостью понимается возможность сжатия или растягивания flex-элемента на заданную величину.

Свойство `flex`

Основное преимущество гибкой верстки документа заключается в способности flex-элементов заполнять все доступное свободное пространство контейнера, изменяя свою ширину или высоту вдоль его главной оси. Расширение flex-элементов за счет свободного пространства контейнера, равно как и их сжатие для предотвращения выхода за пределы контейнера, выполняется пропорционально специально заданному коэффициенту. (Детально эти концепции описаны в следующих разделах.)

Степень сжатия или растягивания flex-элемента определяется с помощью свойства общего назначения `flex` или одного из свойств, изменяющих отдельные характеристики элемента. При расширении flex-элемент может заполнять сразу все свободное пространство контейнера или только отдельную его часть. Сжатие элементов обычно осуществляется в случаях, когда их суммарный размер превышает главный размер flex-контейнера, но далеко не всегда приводит к размещению всех flex-элементов внутри контейнера.

Описанные выше действия выполняются с помощью свойства `flex`, представляющего собой сокращенный вариант сразу трех индивидуальных свойств: `flex-grow`, `flex-shrink` и `flex-basis`. Несмотря на то что индивидуальные свойства позволяют изменять размерные характеристики flex-элементов по отдельности, спецификация CSS настоятельно рекомендует настраивать их с помощью свойства общего назначения `flex`. О причинах такой рекомендации рассказано далее.

flex

Значение	[<*flex-grow*> <*flex-shrink*>? || <*flex-basis*>] | `none`
Начальное значение	`0 1 auto`
Применяется	Flex-элементы (дочерние элементы flex-контейнера)
Процентное значение	Поддерживается только для значений свойства `flex-basis`; относительно главного размера контейнера
Вычисляется	Согласно определению индивидуальных свойств
Наследуется	Нет
Анимируется	См. описание индивидуальных свойств

Свойство `flex` определяет "длину" flex-элемента — его размер вдоль главной оси гибкого контейнера (см. раздел "Оси flex-контейнера"). Оно устанавливает главный размер flex-элемента, применяемый вместо значения свойства `height` или `width`. Значения свойства `flex` указывают *коэффициент сжатия*, *коэффициент растягивания* и *базовый размер* элемента.

Базовый размер представляет изначальный размер flex-элемента вдоль главной оси до его растягивания или сжатия. Как предполагает название, базовый размер является той величиной, относительно которой вычисляется степень растягивания (при заполнении свободного пространства контейнера) или сжатия (с целью предотвращения выхода за пределы контейнера) элемента. Чтобы предотвратить растягивание или сжатие элемента, сохранив его в исходном размере, первых два коэффициента свойства `flex` нужно представить нулевыми значениями.

```
.flexItem {
   width: 50%;
   flex: 0 0 200px;
}
```

Согласно данному правилу, главный размер flex-элемента равен 200 пикселям, и для него запрещено как растягивание, так и сжатие. Поскольку главная ось

flex-контейнера направлена горизонтально, значение `200px` замещает явно заданную ширину элемента (`width: 50%;`). Если бы главная ось flex-контейнера располагалась вертикально, то базовый размер элемента использовался бы вместо вычисляемого значения свойства `height`.

Замещение значений свойств `height` и `width` выполняется вне правил каскадирования, что не позволяет предохранить их от изменения с помощью ключевого слова `!important`.

Свойство `flex` разрешается применять только к flex-элементам. Его объявление для элементов других типов будет попросту проигнорировано.

Учтите, что значения свойства `flex` нужно передавать в строго заданном в его определении порядке. В противном случае будет получен заведомо неправильный результат.

Свойство `flex-grow`

Свойство `flex-grow` определяет возможность растягивания flex-элемента при наличии в его контейнере достаточного количества свободного места. Оно указывает коэффициент растягивания элемента по отношению к такому же показателю остальных flex-элементов контейнера.

Разработчики спецификации настоятельно не рекомендуют устанавливать коэффициент растягивания с помощью свойства `flex-grow`. Для его определения лучше использовать свойство общего назначения `flex`. Дальнейшее описание приводится исключительно с целью ознакомить вас с общими принципами растягивания элементов в CSS.

flex-grow

Значение	*<number>*
Начальное значение	0
Применяется	Flex-элементы (дочерние элементы flex-контейнера)
Вычисляется	Согласно определению
Наследуется	Нет
Анимируется	Да

В качестве значения свойства `flex-grow` допускается использовать только неотрицательные числа. Значение не обязательно должно быть целочисленным, но оно всегда больше или равно нулю. Оно определяет *коэффициент растягивания* flex-элемента, указывающий, насколько сильно будет растягиваться элемент относительно остальных flex-элементов одного с ним контейнера (разумеется, если только в нем достаточно свободного места).

При наличии в контейнере свободного места оно перераспределяется пропорционально между всеми растягиваемыми flex-элементами согласно числовым значениям коэффициентов, определенным свойством `flex-grow`.

Предположим, что в горизонтальный flex-контейнер шириной 750 пикселей добавлены три элемента, ширина каждого из которых составляет 100 пикселей. Это означает, что flex-элементы могут растягиваться в общей сложности на 450 пикселей (их суммарный главный размер составляет 300 пикселей). Исходный вид flex-контейнера показан в первом примере на рис. 12.42.

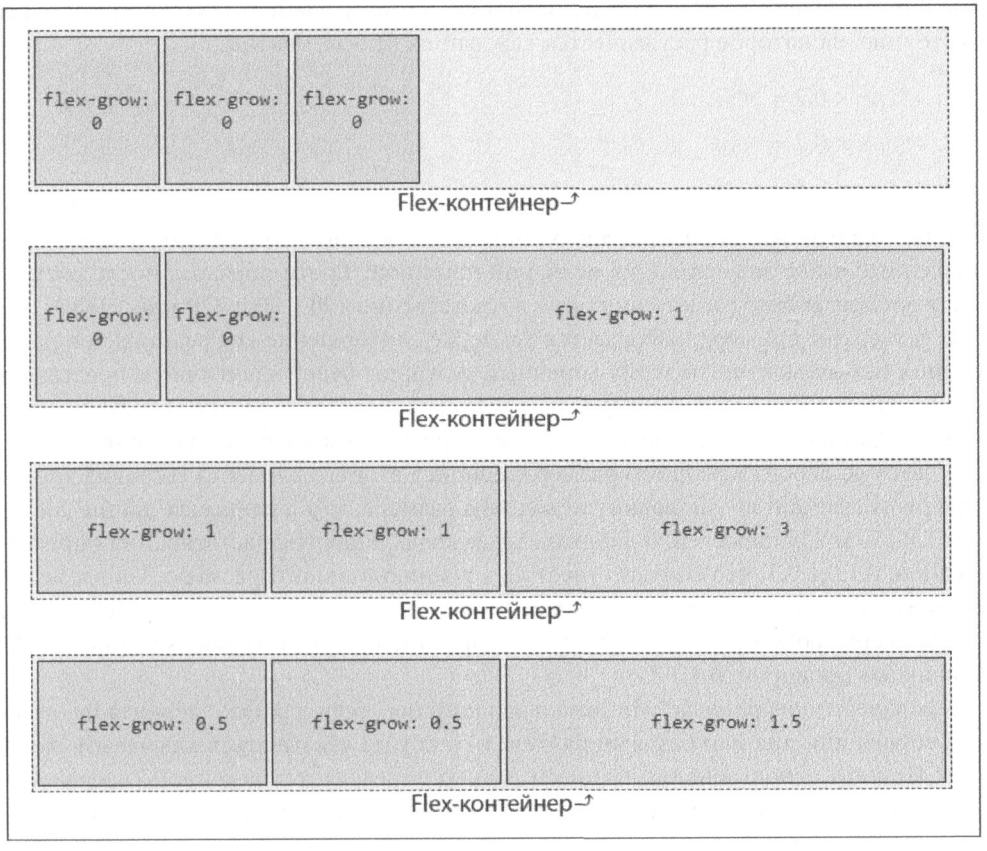


Рис. 12.42. Способы растягивания flex-элементов в пределах контейнера

Во втором примере на рис. 12.42 растягивается только третий flex-элемент, поскольку только ему одному назначен положительный коэффициент, отличный от нуля. Несмотря на объявление `flex-grow: 1`, пользовательский агент может рассматривать его как произвольное положительное число, поскольку элементы, для которых определен нулевой коэффициент растягивания, будут сохранять исходный размер. Следовательно, все 450 пикселей свободного пространства будут заполнены содержимым третьего flex-элемента контейнера — его главный размер составит 550 пикселей. При этом явное объявление его ширины с помощью свойства `width` не играет ощутимой роли.

Несмотря на разные коэффициенты растягивания, размеры flex-элементов в третьем и четвертом примерах полностью совпадают. Для начала ознакомимся с третьим примером, элементы которого растягиваются согласно коэффициентам 1, 1 и 3 соответственно. Чтобы определить, насколько сильно увеличится главный размер каждого из flex-элементов, необходимо определить долю его коэффициента растягивания в общем коэффициенте, получаемом путем суммирования коэффициентов растягивания всех flex-элементов контейнера (5). Простые расчеты показывают, что прирост главного размера элементов составляет 0,2, 0,2 и 0,6 их исходной ширины. Умножив полученные значения на исходную ширину элемента, можно определить расстояние, на которое растягивается каждый из flex-элементов.

1. `450px` × 0,2 = `90px`
2. `450px` × 0,2 = `90px`
3. `450px` × 0,6 = `270px`

Для того чтобы получить полную ширину элементов, необходимо сложить представленные выше значения с их исходной шириной. Таким образом, после растягивания элементы будут иметь ширину соответственно 190, 190 и 370 пикселей.

В четвертом примере наблюдается такое же соотношение коэффициентов растягивания flex-элементов, поэтому конечный результат будет идентичным предыдущему. Определим, как будут растягиваться элементы при соотношении коэффициентов 0,5:1:1,5. Согласно указанной пропорции, первый элемент будет увеличиваться на одну шестую своего исходного размера, прирост второго элемента составит треть, а четвертого элемента — половину исходного размера, что в единицах длины составляет 75, 150 и 225 пикселей. В данном случае коэффициенты растягивания определяются как 0,1:0,1:0,3, что соответствует увеличению главного размера flex-элементов соответственно на 25, 25 и 75 пикселей. Точно таким же образом будут увеличиваться и размеры любых других элементов, коэффициенты растягивания которых представляются пропорцией 1:1:3.

Как известно из раздела "Минимальная ширина", если для flex-элемента не объявлена точная ширина или базовый размер, то он будет указываться ключевым словом `auto`. При его использовании базовый размер определяется шириной содержимого flex-элемента, располагаемого в один ряд (без переноса в другие ряды). Ширина элемента будет определять его базовый размер только в случае явного ее объявления в стилевом свойстве `width`. (Подробно назначение ключевого слова `auto` описано в разделе "Автоматическая установка базового размера".) Если бы в примерах, приведенных на рис. 12.42, ширина flex-элементов не указывалась в явном виде, то при столь малом размере шрифта пользовательскому агенту пришлось бы распределять область свободного пространства с намного большей, чем 450 пикселей, шириной.

Главный размер flex-элемента зависит от количества свободного пространства в контейнере, коэффициентов растягивания всех элементов и их базового размера. Подробно о том, что такое базовый размер flex-элемента, рассказывается в одной из следующих глав.

Теперь рассмотрим более сложный пример, в котором flex-элементам присвоены не только разные значения свойства `width`, но и коэффициенты растягивания. Во втором примере на рис. 12.43 показаны flex-элементы шириной соответственно 100, 250 и 100 пикселей, растягиваемые согласно коэффициентам 1, 1 и 3. Поскольку гибкие элементы заключены во flex-контейнер шириной 750 пикселей, между ними нужно распределить свободное пространство шириной 300 пикселей. Таким образом, согласно указанной выше пропорции, область свободного пространства шириной 300 пикселей нужно разделить на 5 долей. Каждая из таких долей будет определять растягивание flex-элемента на 300 ÷ 5 = 60 пикселей. Следовательно, главный размер первого и второго элементов увеличится на 60 пикселей (свойство `flex-grow` имеет значение `1`). При этом последний flex-элемент увеличивается в ширине на 180 пикселей (значение свойства `flex-grow` равно `3`).

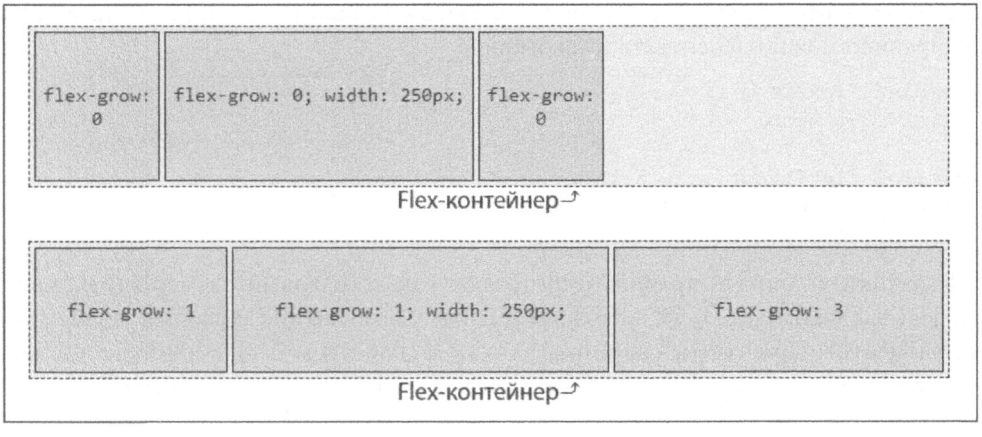


Рис. 12.43. Изменение размеров flex-элементов, имеющих разную ширину и коэффициенты растягивания

Ниже приведен краткий конспект вычислений, которые выполняются с целью определения конечного размера flex-элементов, растягиваемых до заполнения всего свободного пространства контейнера.

Ширина свободного пространства: `750px - (100px + 250px + 100px) = 300px`

Коэффициенты растягивания: 1 + 1 + 3 = 5

Прирост главного размера flex-элемента: `300px ÷ 5 = 60px`

После растягивания flex-элементы получают следующую вычисляемую ширину.

Элемент 1 = `100px + (1 × 60px) = 160px`

Элемент 2 = `250px + (1 × 60px) = 310px`

Элемент 3 = `100px + (3 × 60px) = 280px`

Результат сложения указанных значений, как и предполагалось, равен 750 пикселям.

Коэффициент растягивания и свойство `flex`

Как известно, свойство `flex` принимает три значения: коэффициент растягивания, коэффициент сжатия и базовый размер. Первое из них, представляемое положительным ненулевым числом, соответствует значению свойства `flex-grow`. Если при объявлении свойства `flex` не указывать коэффициенты растягивания и сжатия, то по умолчанию в качестве его первого значения будет использовано число `1`. Тем не менее, если для flex-элемента не объявить ни свойство `flex`, ни `flex-grow`, то коэффициент растягивания по умолчанию будет установлен в нулевое значение.

Чтобы понять, к чему это может привести, вернемся к рассмотрению второго контейнера, показанного на рис. 12.42, элементы которого растягиваются согласно коэффициентам 0, 0 и 1. Поскольку в нашем примере устанавливается только свойство `flex-grow`, то в равнозначном объявлении свойства `flex` базовый размер каждого из элементов будет представляться ключевым словом `auto`. В результате исходное правило можно переписать в таком виде.

```
#example2 flex-item {
   flex: 0 1 auto;
}
#example2 flex-item:last-child {
   flex: 1 1 auto;
}
```

Как видите, данные правила определяют только базовый размер (представлен ключевым словом `auto`), но не коэффициенты растягивания и сжатия первых двух flex-элементов. При замене свойства `flex-grow` свойством `flex` в примере, показанном на рис. 12.42, базовый размер элементов обычно представляют значением `0%`.

```
#example2 flex-item {
   flex: 0 1 0%;
}
#example2 flex-item:last-child {
   flex: 1 1 0%;
}
```

Учитывая, что коэффициент сжатия `1` и базовый размер `0%` являются значениями по умолчанию, приведенный выше код можно сократить до такого вида.

```
#example2 flex-item {
   flex: 0;
}
#example2 flex-item:last-child {
   flex: 1;
}
```

Результат выполнения указанных правил представлен на рис. 12.44. Сравните его с рис. 12.42, чтобы лучше понять, в чем заключаются различия между обоими способами форматирования.

Сразу же бросается в глаза другой результат для первых двух примеров: все элементы имеют нулевой базовый размер, а положительный коэффициент растягивания

указан только для последнего flex-элемента второго контейнера. Строгий математический расчет показывает, что в данном случае flex-элементы должны получать следующие значения главного размера: 0, 0 и 750 пикселей. Но в таком форматировании нет и капли здравого смысла, ведь при наполнении первых двух элементов содержимым оно будет гарантировано выступать за их края, и это при том, что во flex-контейнере предостаточно места для предотвращения столь непродуманного форматирования.

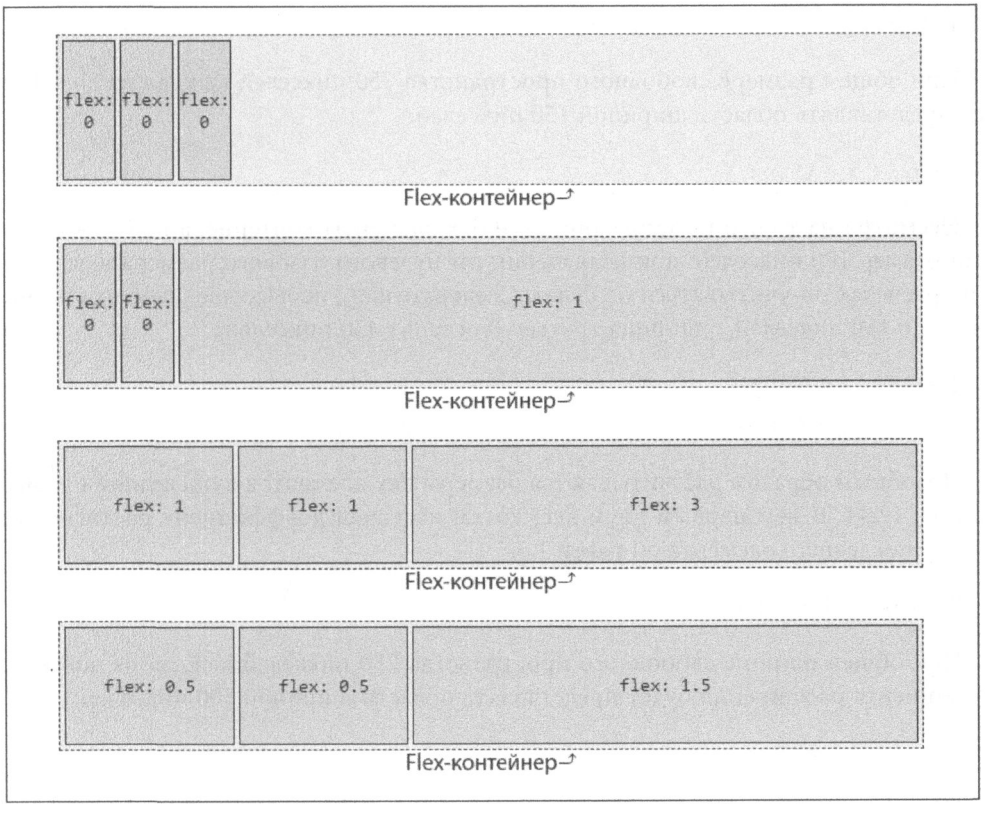


Рис. 12.44. Растягивание flex-контейнеров с помощью свойства общего назначения flex

Разработчики спецификации предложили эффективный способ предотвращения подобных ситуаций. Явное объявление базового размера или установка его в значение по умолчанию `0%` в свойстве `flex` при нулевом коэффициенте растягивания обеспечивает сжатие flex-элементов до размера содержимого. Легко заметить, что на рис. 12.44 размер не растягиваемых элементов определяется длиной текстовой строки `flex:` (включая двоеточие).

Если для flex-элемента не объявлено точное значение свойства `min-width`, то его минимальная ширина (высота при вертикальной направленности главной оси контейнера) будет определяться меньшим из двух значений: минимально возможной шириной (высотой) содержимого или значением свойства `width` (`height`), заданного в явном виде.

При этом назначение положительных коэффициентов растягивания и нулевого базового размера сразу всем flex-элементам приводит к перераспределению между ними свободного пространства контейнера пропорционально значениям, обозначенным в стилевом правиле. В третьем примере на рис. 12.44 первым двум элементам назначен коэффициент растягивания `1`, а коэффициент растягивания последнего элемента представлен значением `3`. Таким образом, все свободное пространство контейнера можно разделить на пять равных долей:

1+1+3 = 5

При общем размере свободного пространства 750 пикселей каждая из долей будет представлять область шириной 150 пикселей:

`750px` ÷ 5 = `150px`

Несмотря на то что по умолчанию все flex-элементы контейнера получают главный размер 100 пикселей, при назначении им нулевого базового размера в дальнейших расчетах он учитываться не будет. Следовательно, первые два элемента расширятся до 150 пикселей, а ширина третьего составит 450 пикселей:

1 × `150px` = `150px`

3 × `150px` = `450px`

Подобным образом рассчитываются размеры flex-элементов в последнем примере на рис. 12.44. В нем первым двум элементам назначен коэффициент растягивания 0,5, а у последнего элемента он равен 1,5:

0,5 + 0,5 + 1,5 = 2,5

При общей ширине свободного пространства 750 пикселей каждая из долей коэффициента расширения будет представлять области шириной 300 пикселей:

`750px` ÷ 2,5 = `300px`

По умолчанию всем flex-элементам контейнера определен главный размер 100 пикселей, но при назначении им нулевого базового размера в дальнейших расчетах он также учитываться не будет. Таким образом, первые два элемента будут иметь ширину по 150 пикселей, а ширина третьего элемента составит 450 пикселей:

0,5 × `300px` = `150px`

1,5 × `300px` = `450px`

Опять-таки, поведение элементов в рассмотренных выше примерах заметно отличается от наблюдаемого при растягивании их с помощью одного только свойства `flex-grow`, в котором базовый размер по умолчанию определяется ключевым словом `auto`. В последнем случае между flex-элементами распределяется не все пространство контейнера, а только область свободного пространства, исходно не занятая

элементами, которые имеют собственный размер. С другой стороны, при решении этой же задачи с использованием свойства `flex` элементам назначается нулевой базовый размер, поэтому при расчете растягивания учитывается все пространство контейнера. Различие между обоими подходами наглядно проиллюстрировано на рис. 12.45.

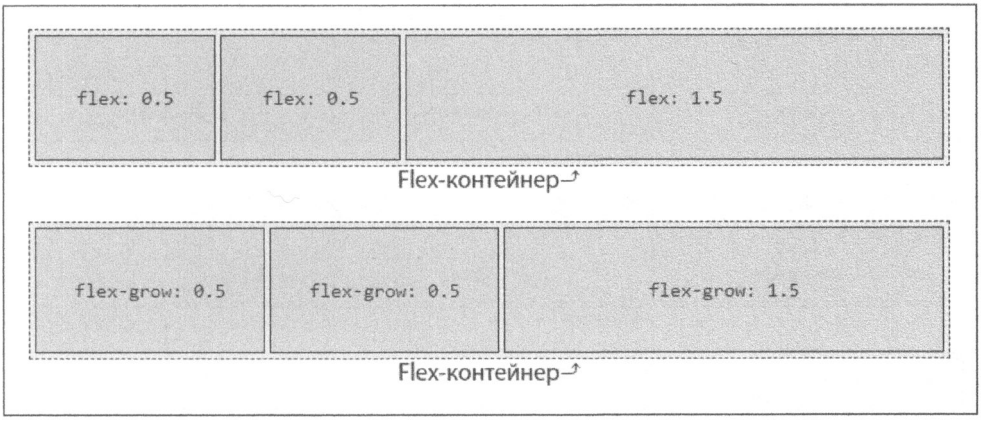


Рис. 12.45. Различие между растягиванием flex-элементов с помощью свойств `flex` *и* `flex-grow`

Далее речь пойдет о сжатии flex-элементов и о коэффициенте сжатия, который далеко не всегда можно представить через инвертированный коэффициент растягивания.

Свойство `flex-shrink`

Значение *<flex-shrink>*, предаваемое свойству `flex`, определяет коэффициент сжатия flex-элемента. Отдельно от остальных настроек гибкости он устанавливается свойством `flex-shrink`.

Разработчики спецификации настоятельно не рекомендуют устанавливать коэффициент сжатия с помощью свойства `flex-shrink`. Для его определения лучше использовать свойство общего назначения `flex`. Дальнейшее описание приводится исключительно с целью ознакомить вас с общими принципами сжатия элементов в CSS.

Коэффициент сжатия определяет, насколько сильно будет сжиматься flex-элемент по отношению к остальным flex-элементам при недостатке в контейнере свободного пространства для размещения их всех. Если не указать его в свойстве flex или вообще не объявлять свойства `flex` и `flex-shrink` для flex-элемента, то будет использовано значение по умолчанию `1`. Как и коэффициент растягивания, значение свойства `flex-shrink` представляется положительным числом. Отрицательные, как и нулевые, значения коэффициента сжатия пользовательским агентом не обрабатываются.

С математической точки зрения коэффициент сжатия определяет "отрицательное свободное пространство", доступное для распределения между flex-элементами в ситуациях, когда их суммарный размер превышает размер гибкого контейнера. Наглядно концепция "отрицательного свободного пространства" проиллюстрирована на рис. 12.46.

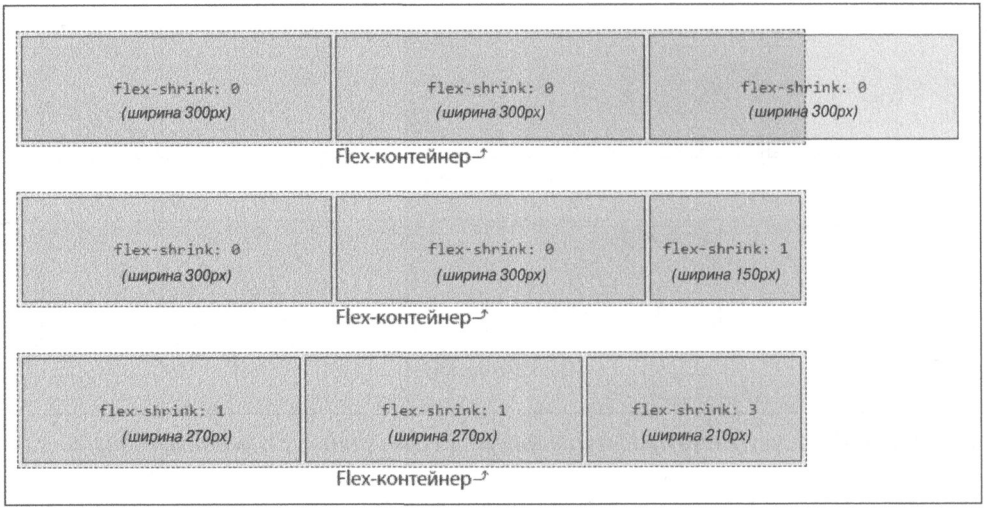


Рис. 12.46. Способы сжатия flex-элементов, не помещающихся в контейнер

Рис. 12.46 во многом подобен рис. 12.42, отличаясь только исходной шириной изображенных на нем flex-элементов: `width: 300px` против `width: 100px`. Гибкие элементы по-прежнему заключены в контейнер шириной 750 пикселей, но общая ширина flex-элементов теперь равна 900 пикселям. Таким образом, содержимое контейнера выступает за края родительского элемента на 150 пикселей. В случае запрета на перенос и сжатие flex-элементов (см. раздел "Перенос flex-элементов"), как показано в первом примере на рис. 12.46, последний из них будет выходить за край контейнера, имеющего строго заданный размер. Для уменьшения размера гибких элементов нужно в явном виде определить им нулевой коэффициент сжатия. Во всех остальных случаях элементы будут выступать за край flex-контейнера.

Во втором примере на рис. 12.46 сжатие разрешено только для последнего flex-элемента. Следовательно, он будет сжиматься до такой степени, чтобы обеспечить размещение внутри контейнера всех его дочерних элементов. Согласно условиям задачи, в контейнер с главным размером 750 пикселей нужно поместить все три flex-элемента суммарной шириной 900 пикселей, что возможно только при распределении между ними области "отрицательного свободного пространства" шириной 150 пикселей. Поскольку ширина первых двух flex-элементов не изменяется (`300px` + `300px`), все "отрицательное свободное пространство" шириной `150px` будет полностью поглощено третьим элементом. В результате он будет сжат до ширины 150 пикселей — минимального размера, обеспечивающего полное заключение всех трех элементов в родительский flex-контейнер. (Справедливости ради нужно уточнить,

что коэффициент сжатия третьего элемента совсем не обязательно устанавливать в значение 1, как это делается во втором примере. Например, можно обойтись значением `0.001`, `100`, `314159.65` или любым другим положительным числовым значением, распознаваемым браузером.)

В третьем примере положительный коэффициент сжатия задан всем трем flex-элементам.

```
#example3 flex-item {
   flex-shrink: 1;
}
#example3 flex-item:last-child {
   flex-shrink: 3;
}
```

В этом примере в явном виде объявляется значение одного только свойства `flex-shrink`. Если свести его код к использованию свойства общего назначения `flex`, то он примет следующий вид.

```
#example3 flex-item {
   flex: 0 1 auto; /* по умолчанию коэффициент растягивания
                      равен 0, а базовый размер представляется
                      ключевым словом auto */
}
f#example3 flex-item:last-child {
   flex: 0 3 auto;
}
```

Если сжимать сразу все flex-элементы (как в данном случае), то их ширина будет изменяться пропорционально объявленным в коде коэффициентам сжатия. Исходя из этого, чем больше коэффициент, тем сильнее будет сжиматься flex-элемент по сравнению с остальными flex-элементами.

При главном размере контейнера 750 пикселей и 300-пиксельной ширине каждого из трех flex-элементов им совместно нужно поглотить "отрицательную область свободного пространства" шириной 150 пикселей. Первые два элемента имеют коэффициент сжатия 1, и только у последнего, третьего, элемента он равен 3. Таким образом, недостающее свободное пространство нужно условно разделить на 5 долей:

$$1 + 1 + 3 = 5$$

Разделив ширину недостающего пространства на количество долей, представляющих общий коэффициент сжатия, можно получить размер каждой из них:

$$150px \div 5 = 30px$$

Согласно проведенным выше расчетам, каждый из первых двух элементов с главным размером 300 пикселей будет сжиматься до ширины 270 пикселей, а ширина последнего элемента уменьшится до 210 пикселей. Общая ширина всех трех flex-элементов после сжатия составит 750 пикселей.

300px – (1 × 30px) = 270px

300px – (3 × 30px) = 210px

Следующий CSS-код обеспечивает равнозначное изменение размеров flex-элементов при разных коэффициентах сжатия, поскольку их соотношение в точности совпадает с рассмотренным в предыдущем примере.

```
flex-item {
   flex: 1 0.25 auto;
}
flex-item:last-child {
   flex: 1 0.75 auto;
}
```

Проведя все необходимые вычисления, можно определить, что и в этом случае flex-элементы будут сжиматься соответственно до ширины 270, 270 и 210 пикселей, но только до тех пор, пока размер их содержимого не превысит указанные значения. Сжатие flex-элемента всегда ограничивается размером несжимаемого и непереносимого содержимого, измеренного вдоль главной оси контейнера.

Предположим, что в первый flex-элемент вставлено изображение шириной 300 пикселей. Согласно описанным выше ограничениям, такой элемент не будет уменьшаться в размерах так, как если бы ему был назначен нулевой коэффициент сжатия. В подобном случае первый элемент сохранит исходную ширину 300 пикселей, а "отрицательное свободное пространство" будет пропорционально распределено между оставшимися двумя flex-элементами (согласно только их коэффициентам сжатия).

Таким образом, недостающее свободное пространство нужно условно разделить на четыре дольные части (одна часть отнимается у второго, а оставшиеся три — у третьего элемента). Проведя точные расчеты, можно определить, что каждая дольная часть представляет размер 37,5 пикселя. Следовательно, после сжатия flex-элементы получат следующие размеры вдоль главной оси контейнера: 300, 262,5 и 187,5 пикселя. Как и прежде, их общая ширина составит 750 пикселей. Результат сжатия приведен на рис. 12.47.

Элемент 1 = 300px – (0 × 37,5px) = 300,0px

Элемент 2 = 300px – (1 × 37,5px) = 262,5px

Элемент 3 = 300px – (3 × 37,5px) = 187,5px

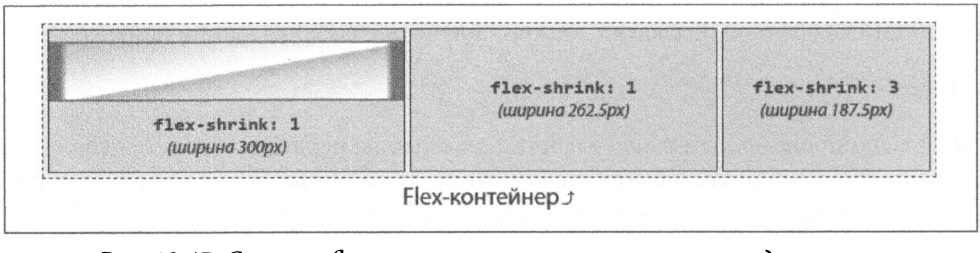


Рис. 12.47. Сжатие flex-элементов ограничивается их содержимым

При ширине изображения, равной 294 пикселям, первый flex-элемент будет сжиматься всего на 4 пикселя. Оставшиеся 146 пикселей будут пропорционально отобраны у второго и третьего элементов — размер каждой дольной части в подобных случаях составит 36,5 пикселя. Следовательно, ширина flex-элементов после сжатия составит соответственно 296, 263,5 и 190,5 пикселя.

Если в каждый из трех flex-элементов включить непереносимое содержимое или графическое изображение шириной 300 и более пикселей, то ни один из них сжиматься не будет, как показано в первом примере на рис. 12.46.

Сжатие и удельный коэффициент

Предыдущий пример очень просто анализировать, поскольку все три flex-элемента до сжатия имеют одинаковую ширину. Рассмотрим, каким образом сжимаются элементы, которые имеют разные исходные размеры. Пусть ширина первого и третьего элементов составляет 250 пикселей, а средний элемент имеет главный размер 500 пикселей (рис. 12.48).

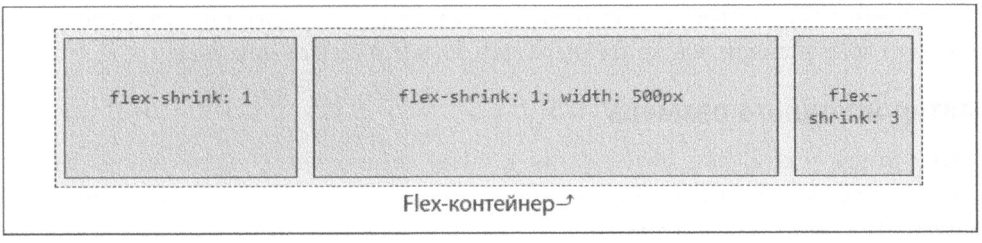


Рис. 12.48. Уменьшение размеров элементов выполняется пропорционально их коэффициентам сжатия

Теперь нужно учитывать не только коэффициент сжатия, но и ширину каждого flex-элемента, обычно ограниченную размером содержимого, которое не подлежит переносу на другие строки. На рис. 12.48 проиллюстрирована попытка вместить flex-элементы общей шириной 1000 пикселей в контейнер с главным размером 750 пикселей. Таким образом, суммарно все элементы нужно сжать на 250 пикселей при общем коэффициенте сжатия 5.

Если бы сжатие flex-элементов настраивалось с помощью свойства `flex-grow`, то для вычисления размера дольной части ширину сжимаемой области нужно было бы разделить на 5. Дальнейший расчет показывает, что в таком случае сжатые flex-элементы будут иметь размеры 200, 550 и 100 пикселей соответственно. В действительности сжатие элементов выполняется несколько по иному сценарию.

Правда состоит в том, что недостающее свободное пространство нужно отнимать у flex-элементов согласно не только коэффициентам сжатия, но и удельной ширине. Для получения удельного коэффициента сжатия нужно разделить ширину "отрицательного свободного пространства" на сумму произведений исходных значений ширины и коэффициентов сжатия всех flex-элементов контейнера.

$$\text{Удельный коэф. сжатия} = \frac{\text{Ширина "отрицательного свободного пространства"}}{((\text{Ширина}1 \times \text{Коэф. сжатия}1) + \ldots + (\text{Ширина}N \times \text{Коэф. сжатия}N))}$$

Удельный коэффициент сжатия показывает действительную степень сжатия элементов:

= 250px ÷ ((250px × 1) + (500px × 1) + (250px × 3))

= 250px ÷ 1500px

= 0.166666667 (16,67%)

Чтобы определить размер, на который сжимается каждый из flex-элементов, нужно умножить его исходную ширину на удельный коэффициент сжатия и значение свойства `flex-shrink`.

Элемент1 = 250px × (1 × 16,67%) = 41,67px

Элемент2 = 500px × (1 × 16,67%) = 83,33px

Элемент2 = 250px × (3 × 16,67%) = 125px

Осталось вычесть полученные значения из размеров соответствующих элементов и проверить полученный результат. После сжатия flex-элементы получат следующие размеры вдоль главной оси контейнера: 208,33, 416,67 и 125 пикселей.

Фактор исходного размера

При нулевом коэффициенте сжатия, а также автоматически устанавливаемых базовом размере и ширине содержимое flex-элемента не будет переноситься на другие ряды ни при каких обстоятельствах. И наоборот, для переноса содержимого flex-элемента на последующие ряды ему нужно определить любой положительный коэффициент сжатия. Поскольку сжатие flex-элементов выполняется на пропорциональной основе, задание всем им одинакового коэффициента сжатия приведет к размещению их содержимого в одинаковом количестве рядов.

Во всех трех примерах, показанных на рис. 12.49, исходная ширина flex-элементов в явном виде не объявляется (свойству `width` передается значение `auto`) и в большой степени зависит от размера их содержимого. В данном случае главный размер контейнера равен не 750, как в предыдущих примерах, а 250 пикселям.

Обратите внимание на то, что в первом примере, в котором все элементы имеют одинаковые коэффициенты сжатия, их содержимое размещается в одном и том же количестве строк. Во втором примере коэффициент сжатия первого flex-элемента вдвое меньше коэффициента сжатия остальных двух элементов — его содержимое размещается на вдвое меньшем количестве строк. Это свойство коэффициента сжатия стоит взять на вооружение!

В третьем примере коэффициент сжатия в явном виде не назначается ни одному из flex-элементов. В результате содержимое выступает за правый край flex-контейнера.

К концу 2017 года возможность определения количества строк, между которыми перераспределяется содержимое flex-элементов, с помощью коэффициента сжатия так и не была реализована во всех основных браузерах. Попытка выполнения этой операции в каком бы то ни было

пользовательском агенте приводит к совершенно противоречивым результатам.

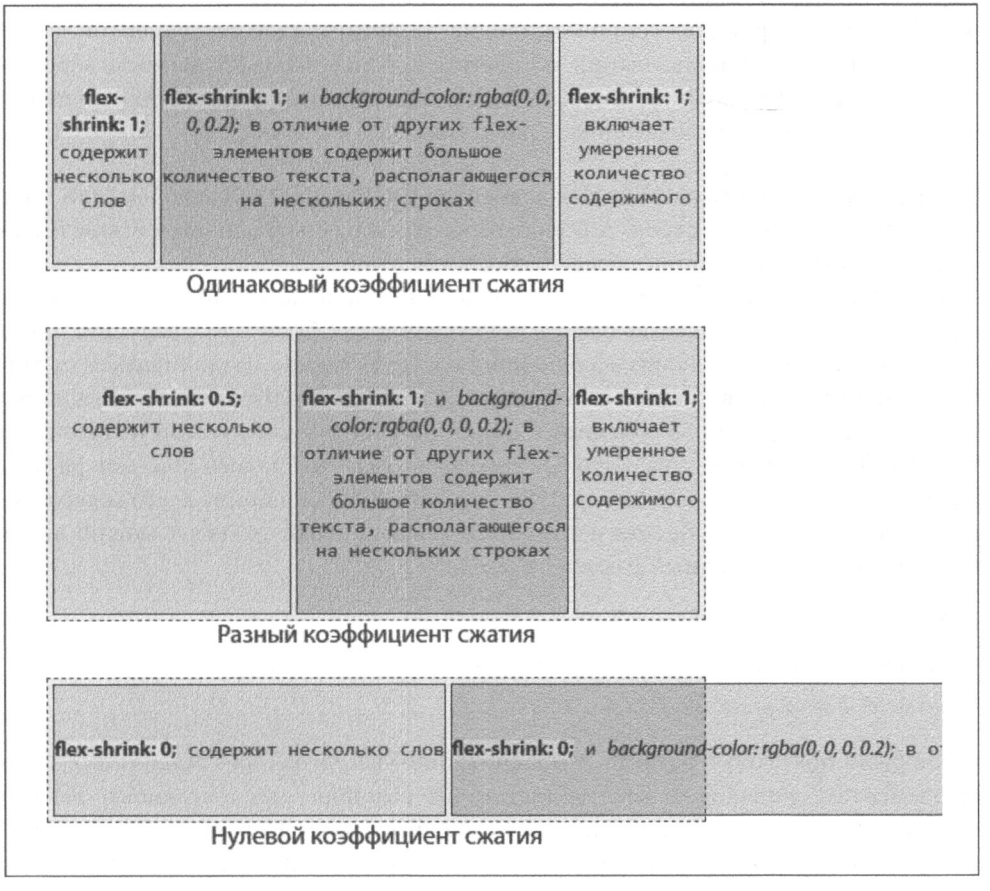


Рис. 12.49. Размер flex-элементов уменьшается пропорционально коэффициентам сжатия и ширине их содержимого

Поскольку сжатие flex-элементов согласно коэффициенту, объявляемому в свойстве `flex`, выполняется пропорционально их ширине, изменяя ее, можно легко контролировать количество текстовых строк в каждом из них, а потому и высоту контейнера. Таким образом, высоту всех дочерних элементов flex-контейнера будет определять flex-элемент с наибольшим коэффициентом сжатия.

В последнем примере содержимое flex-элементов имеет ширину 280, 995 и 480 пикселей соответственно — такой размер вдоль главной оси оно имеет при запрете переноса текста на другие строки (это округленные значения, отображаемые на панели инструментов разработчика). Из этого следует, что содержимое общей шириной 1755 пикселей нужно вместить в контейнер с главным размером 520 пикселей, пропорционально сжимая flex-элементы согласно указанным в стилевом правиле коэффициентам.

Помните, что при определении коэффициента сжатия нельзя полагаться на измерительные способности инструментов разработчика пользовательского агента. Описанные в этом разделе примеры призваны всего лишь проиллюстрировать общие принципы сжатия элементов, что не требует использования абсолютно точных входных данных, а потому проведение измерений с помощью инструментов разработчика в данном конкретном случае вполне допустимо.

В первом случае в каждом из flex-элементов наблюдается одинаковое (или почти одинаковое) количество строк. А все потому, что сжатие выполняется исключительно пропорционально ширине содержимого элементов.

Поскольку исходная ширина элементов в явном виде не указывается, она не может использоваться в качестве базиса вычислений, как было в предыдущих примерах. Исходя из этого, в наших дальнейших расчетах нужно отталкиваться от того, что "отрицательное свободное пространство" шириной 1235 пикселей нужно распределить между тремя элементами с шириной содержимого соответственно 280, 995 и 480 пикселей. Учитывая, что 520 пикселей (главный размер контейнера после сжатия) составляют всего 29,63% от 1755 пикселей (общая ширина всего содержимого), ширину каждого из flex-элементов при коэффициенте сжатия 1 можно вычислить, умножив его исходный размер на 29,63.

Элемент 1 = `280px` × 29,63% = `83px`

Элемент 2 = `995px` × 29,63% = `295px`

Элемент 3 = `480px` × 29,63% = `142px`

Как было показано в разделе "Выравнивание элементов", для задания всем трем flex-элементам одинаковой высоты достаточно выровнять их с помощью объявления `align-items: stretch`, применяемого по умолчанию. Наряду с этим такого же результата можно достичь, снабдив их одним и тем же коэффициентом сжатия, хотя в подобных случаях ширина элементов, скорее всего, будет разной.

Во втором примере на рис. 12.49 размеры flex-элементов изменяются согласно разным коэффициентам сжатия: первый элемент сжимается вполовину слабее, чем два остальных. Исходный размер содержимого flex-элементов такой же, как и в предыдущем примере (280, 995 и 480 пикселей), при немного разных коэффициентах сжатия: 0,5, 1 и 1. Как известно, для определения ширины элемента при разных ширине и коэффициентах сжатия сначала нужно вычислить удельный коэффициент (*X*).

`280px` + `995px` + `480px` = `1615px`

(0,5 × `280px`) + (1 × `995px`) + (1 × `480px`) = `1235px`

X = `1235px` ÷ `1615px` = 0,7647 (76,47%)

Для определения ширины сжатия каждого flex-элемента достаточно перемножить его исходную ширину на удельный коэффициент и коэффициент сжатия. Таким образом, ширина второго и третьего элементов уменьшится на 76,47% от их главного

размера, а ширина первого элемента — всего на 38,23% от исходной ширины (при вдвое меньшем значении свойства `flex-shrink`). В единицах длины ширина сжатия элементов вычисляется следующим образом:

Элемент 1 = `280px` × 0,3823 = `107px`

Элемент 2 = `995px` × 0,7647 = `761px`

Элемент 3 = `480px` × 0,7647 = `367px`

Конечная ширина элементов представляется такими значениями:

Элемент 1= `280px` – `107px` = `173px`

Элемент 2= `995px` – `761px` = `234px`

Элемент 3= `480px` – `367px` = `113px`

Несложно подсчитать, что суммарная ширина элементов составит 520 пикселей.

Подытожив вышесказанное, можно прийти к выводу, что при назначении flex-элементам не только разной ширины, но и разных значений свойства `flex-shrink` оценка степени сжатия flex-элементов выходит из области интуитивно понятных категорий анализа и переходит в область строгих математических вычислений. Именно поэтому для сжатия flex-элементов лучше использовать свойство общего назначения `flex`, позволяющее однозначно определять базис, относительно которого производятся вычисления конечного размера flex-элементов. Детально о базовом размере и примерах его практического использования рассказано в разделе, посвященном описанию свойства `flex-basis`.

Адаптивная верстка

Способность flex-элементов сжиматься пропорционально главному размеру контейнера без разрывов содержимого лежит в основе технологии адаптивного дизайна.

Например, при создании трехколоночного документа, содержимое которого сжимается и растягивается пропорционально размеру экрана, с помощью гибких элементов можно обойтись без использования медиа-запросов. Пример такого документа, отображенного на широком мониторе, показан на рис. 12.50. Он же при выводе на узком экране мобильного устройства представлен на рис. 12.51.

```
nav {
   flex: 0 1 200px;
   min-width: 150px;
}
article {
   flex: 1 2 600px;
}
aside {
  flex: 0 1 200px;
  min-width: 150px;
}
```

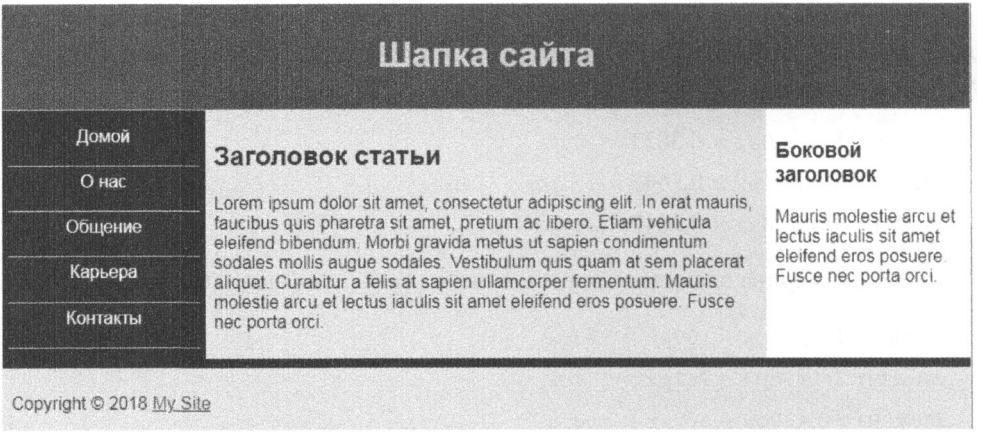


Рис. 12.50. Документ с адаптивным дизайном на широком экране

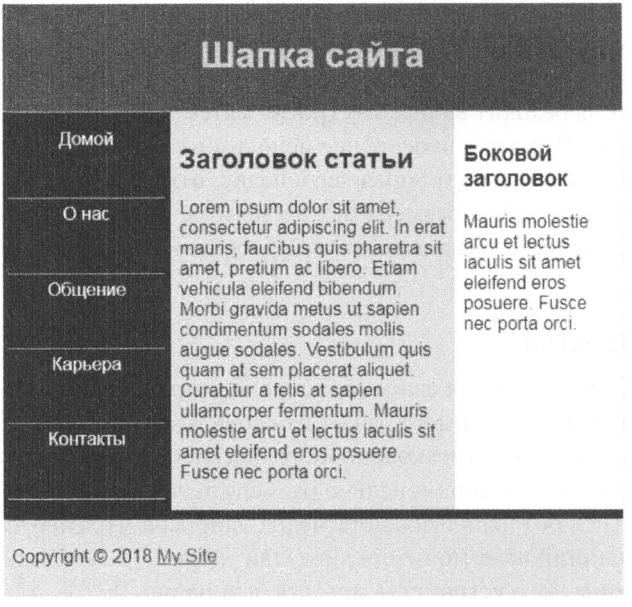


Рис. 12.51. Документ с адаптивным дизайном на узком экране

При отображении документа из этого примера на экранах шириной, превышающей 1000 пикселей, растягиванию подлежит только средняя колонка, поскольку положительный коэффициент назначен ей одной. Но при выводе на носитель, ширина экрана которого не превышает 1000 пикселей, сжатию подлежат сразу все колонки.

Рассмотрим, каким образом верстаются документы с адаптивным дизайном. Размеры элементов `nav` и `aside` определяются следующими объявлениями.

```
flex: 0 1 200px;
min-width: 150px;
```

Они указывают на то, что элементы могут растягиваться только до ширины базового размера, сжимаясь пропорционально друг другу. В частности, это означает, что по умолчанию flex-элементы представляются в базовом размере, а при последующем сжатии их ширина уменьшается до минимального размера 150 пикселей. Если один из элементов включает неделимое содержимое шириной, превышающей 150 пикселей (например, изображение или текстовый фрагмент, лишенный пробелов и отступов), то он будет сжат только до размеров этого содержимого. Предположим, что в элемент `aside` добавлено изображение шириной 180 пикселей. В подобном случае сжатие такого элемента прекратится в точности по достижении данного размера. Наряду с этим элемент `nav` будет сжат до ширины 150, как предписывает стилевое правило.

Теперь рассмотрим, что произойдет при объявлении для элемента `main` свойства `flex` с такими значениями:

```
flex: 1 2 600px;
```

Легко заметить, что элемент `main` как единственный дочерний элемент своего контейнера может расширяться до размеров окна браузера. Если окно браузера имеет, например, ширину 1300 пикселей, а главный размер каждой из боковых колонок документа составляет 200 пикселей, то ширина средней колонки составит 900 пикселей. При сжатии элемента `main` представляемая им средняя колонка будет сужаться вдвое сильнее, чем боковые колонки. В частности, если окно браузера сузить до 900 пикселей, то боковые колонки сожмутся до размера 175 пикселей, а средняя колонка — до 500 пикселей.

Как только ширина окна составит 800 пикселей, боковые колонки получат минимально возможный главный размер: `150px`. Начиная с этого момента сжатие документа будет осуществляться только за счет средней колонки.

Заметьте, что базовую ширину элемента можно указывать с помощью значений, выраженных в любых других единицах измерения длины, а не только в пикселях. В частности, при их использовании код предыдущего примера нужно переписать так.

```
nav {
   flex: 0 1 20ch;
   min-width: 15vw;
}
article {
   flex: 1 2 45ch;
}
aside {
   flex: 0 1 20ch;
   min-width: 10ch;
}
```

Здесь мы не будем рассматривать математические операции, применяемые для приведения значений длины к другим типам данных. Достаточно знать, что минимальные и базовые размеры элементов подбираются так, чтобы сделать текст документа максимально удобочитаемым при выбранных размере шрифта и степени сжатия.

Flex-элементы позволяют эффективно верстать только однонаправленные документы, состоящие не более чем из трех колонок. Для построения более сложных макетов применяются стилевые инструменты совершенно иного уровня, например включенные в модуль CSS Grid Layout (подробно об этом — в следующей главе).

Свойство `flex-basis`

Выше было показано, что ширина flex-элемента зависит от размера его содержимого, а также от отдельных свойств растягивания и сжатия. В общем случае его также можно определить для каждого из гибких элементов с помощью свойства `flex`. Для указания исходного или базового размера flex-элемента, относительно которого выполняется его сжатие и растягивание, в свойстве `flex` применяется значение *<flex-basis>*. Оно определяет размер элемента, к которому добавляется или от которого отнимается свободное пространство, перераспределяемое между всеми flex-элементами одного с ним контейнера. Базовый размер элемента можно также установить отдельно, воспользовавшись свойством `flex-basis`.

Разработчики спецификации настоятельно не рекомендуют устанавливать базовый размер элемента с помощью свойства `flex-basis`. Для его определения лучше использовать свойство общего назначения `flex`. Дальнейшее описание приводится исключительно с целью ознакомить вас с ролью базового размера в стилевых правилах растягивания и сжатия flex-элементов.

flex-basis

Значение	`content` | [*<width>* | *<percentage>*]
Начальное значение	`auto`
Применяется	Flex-элементы (дочерние элементы flex-контейнера)
Процентное значение	Относительно главного размера flex-контейнера
Вычисляется	Согласно определению для абсолютных значений, выраженных в единицах измерения длины
Наследуется	Нет
Анимируется	*<width>*

В общем случае базовый размер определяется размером контейнера flex-элемента, устанавливаемого свойством `box-sizing`. По умолчанию размер регулярного элемента (до применения к нему объявления `display: flex` или `display: inline-flex`) зависит от множества факторов: размеров его содержимого и родительского элемента, а также от значений всевозможных свойств настройки контейнера элемента, определяющих его границы, отступы и позиционирование. Если свойства настройки контейнера элемента не объявлены в явном виде, то они будут представляться

значениями по умолчанию, которые в случае блочных элементов полностью наследуются от родительского элемента.

Значение свойства `flex-basis` указывается в таких же единицах измерения, как и значения свойств `width` и `height`, например `5vw`, `12%` или `300px`.

Универсальное ключевое слово `initial` сбрасывает свойство `flex-basis` до значения по умолчанию `auto`, которое также можно объявить в явном виде. При его использовании базовый размер flex-элемента определяется значением свойства `width` (или `height`). Если же свойству `width` (или `height`) также передано значение `auto`, то базовый размер элемента будет представлен значением, вычисляемым для ключевого слова `content`.

Ключевое слово `content`

На момент написания книги ключевое слово `content` не поддерживалось ни одним из популярных браузеров, за исключением Microsoft Edge 12+, в котором оно представляло ширину или высоту содержимого элемента. По задумке разработчиков, значение `content` должно представлять главный размер содержимого flex-элемента: длину самой длинной строки содержимого элемента вдоль главной оси гибкого элемента.

До завершения процедуры внедрения в популярные браузеры объявление `flex-basis: content;` может неправильно трактоваться как `flex-basis: auto;`, `width: auto;` или `flex-basis: auto; height: auto;` (при вертикальной направленности главной оси flex-элемента). К сожалению, применение ключевого слова `content` в объявлении свойства общего назначения `flex` приводит к отмене всех его деклараций (см. раздел "Оси flex-контейнера").

Значение `content` представляет базовый размер flex-элементов в третьем примере на рис. 12.49, и оно же применяется на рис. 12.52.

В первых двух примерах на рис. 12.52 ширина каждого flex-элемента определяется размером его содержимого, совпадающего с его базовым размером. В первом случае базовый размер (ширина) каждого элемента равен 132 пикселям. Следовательно, общая ширина всех трех элементов, расположенных впритык друг к другу, составляет 396 пикселей, что существенно меньше ширины родительского контейнера.

В третьем примере к flex-элементам применен нулевой коэффициент сжатия. В результате они не могут сжиматься и переноситься в другие ряды при размещении во flex-контейнере с фиксированным главным размером. Их ширина полностью определяется размером непереносимого содержимого, которая также представляет базовый размер элементов. В рассматриваемом случае базовый размер flex-элементов составляет соответственно 309, 1037 и 523 пикселя. К сожалению, на рис. 12.52 не показана часть содержимого второго и все содержимое третьего элемента — с ним можно ознакомиться в дополнительных материалах на сайте книги.

Второй пример описывается такими же размерными характеристиками, как и последний пример, за исключением назначения flex-элементам коэффициента сжатия `1`. Последний предопределяет перенос содержимого при сжатии элементов до размера, достаточного для предотвращения их выхода за пределы контейнера. Таким образом, если главный размер flex-элемента не совпадает с шириной содержимого,

то именно она будет определять его базовый размер — размер, относительно которого рассчитывается сжатие элемента.

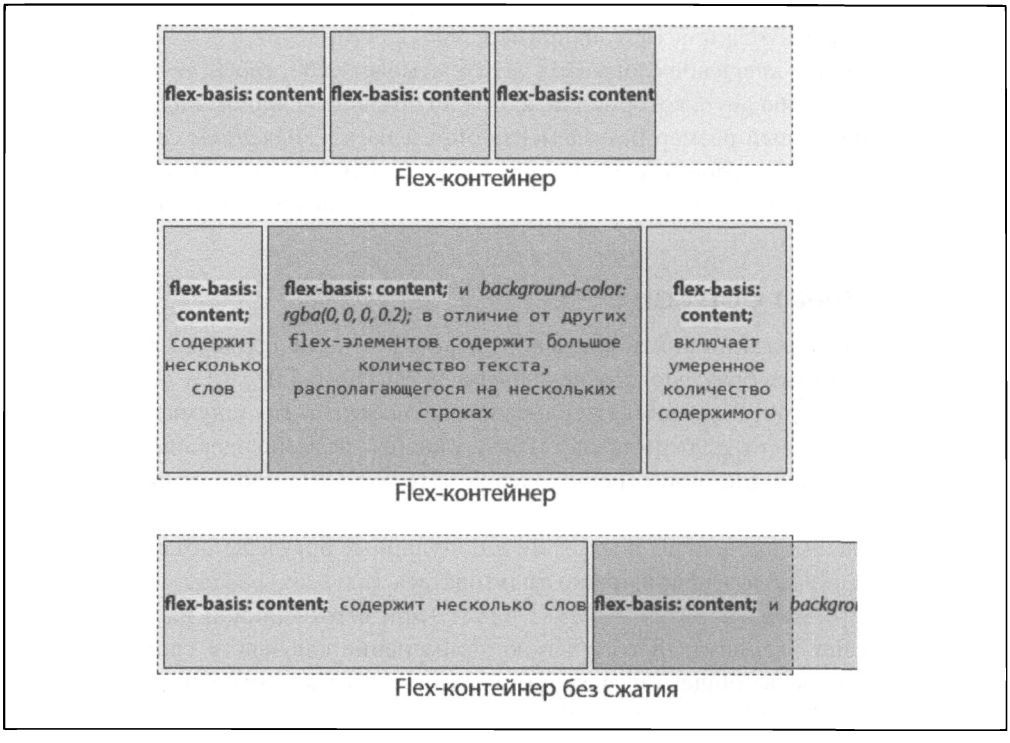


Рис. 12.52. Представление базового размера элемента ключевым словом `content`

Автоматическое назначение базового размера

Вычисляемое значение ключевого слова `auto`, передаваемого свойству `flex-basis` как в явном виде, так и по умолчанию, представляет размер элемента вдоль главной оси до преобразования его во flex-элемент. Базовый размер выражается в таких же единицах изменения, как и представляющее его значение свойства `width` (или `height`), за исключением объявлений `width: auto` (`height: auto`) — в подобных случаях свойство `flex-basis` получает значение `content`.

Если flex-элементы полностью помещаются в контейнер, то их базовый размер, определяемый ключевым словом `auto`, будет в точности совпадать с шириной (высотой) элементов в регулярном состоянии. В случае, когда flex-элементы не помещаются в родительский контейнер, они будут сжиматься пропорционально своему базовому размеру (за исключением случая установки нулевого коэффициента сжатия).

Если главный размер flex-элементов не устанавливается ни одним из стилевых свойств (`width` или даже `min-width`), то к ним по умолчанию применяется объявление `flex-basis: auto` или `flex: 0 1 auto`, позволяющее расширить их до размера содержимого, как показано в первом примере на рис. 12.53. В нем вычисляемое значение `auto` представляет ширину текста, равную приблизительно 110 пикселям. При

этом flex-элементы представляются в фиксированном размере, который они имели бы, находясь в регулярном состоянии (`display: inline-block`). Они выравниваются по началу главной оси контейнера, так как свойству `justify-content` задано значение по умолчанию: `flex-start`.

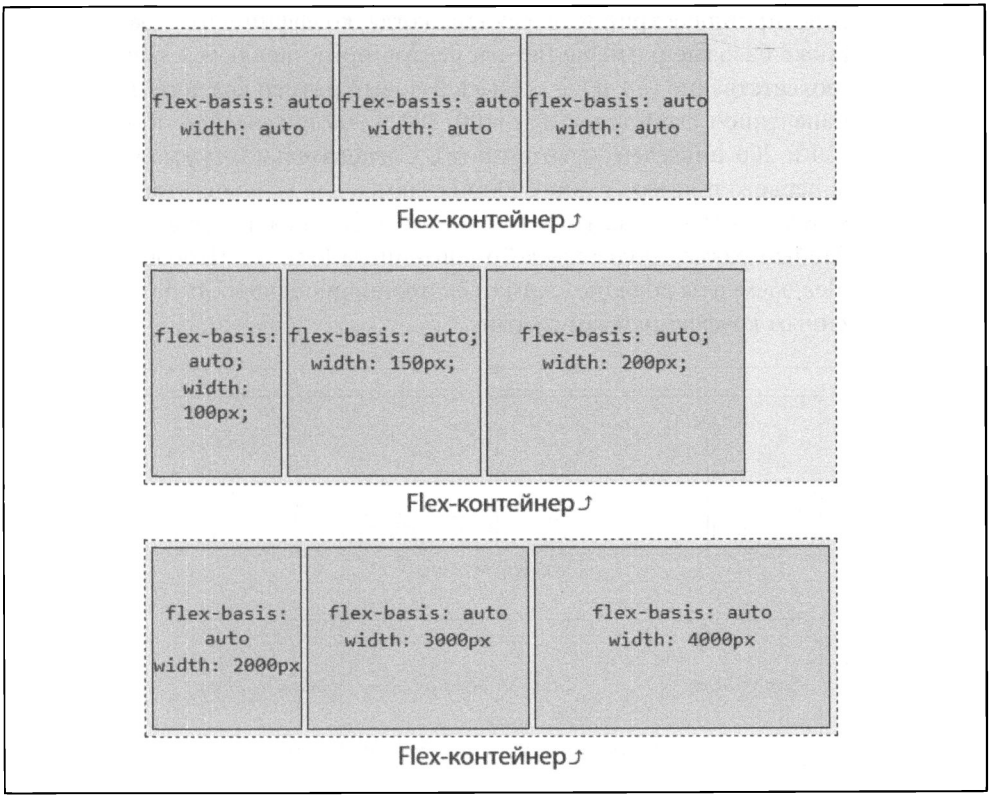


Рис. 12.53. Ширина контейнеров при автоматической настройке их базового размера

Во втором примере на рис. 12.53 показан результат автоматической настройки базового размера при явном объявлении ширины каждого из flex-элементов. До помещения во flex-контейнер элементы имели размер вдоль главной оси контейнера соответственно 100, 150 и 200 пикселей. Для сохранения исходного размера после преобразования во flex-элементы они должны полностью помещаться в контейнер.

В третьем примере на рис. 12.53 базовый размер flex-элементов также устанавливается автоматически, но теперь для каждого из них объявлена очень большая ширина — их исходный размер в бытность регулярными элементами составлял соответственно 2000, 3000 и 4000 пикселей. При таком размере flex-элементы гарантированно будут выступать за края контейнера, поэтому всем им по умолчанию назначен коэффициент сжатия `1`. Сжатие прекращается, как только все flex-элементы будут полностью помещены в контейнер. Чтобы вычислить их размер после сжатия, воспользуйтесь инструкциями предыдущего раздела. (В качестве подсказки: третий элемент будет сжиматься с ширины 4000 до главного размера 240 пикселей.)

Значения по умолчанию

Если для flex-элемента не объявлено ни свойство `flex-basis`, ни `flex`, то его базовый размер определяется автоматически и соответствует ширине или высоте исходного регулярного элемента.

На рис. 12.54 проиллюстрирован случай, когда коэффициенты растягивания и сжатия, а также базовые размеры flex-элементов представляются значениями по умолчанию: соответственно `0`, `1` и `auto`. Здесь базовый размер каждого из элементов определяется значением свойства `width` (100, 200 и 300 пикселей — в первом примере и 200, 400 и 200 пикселей — во втором). Следовательно, суммарная ширина flex-элементов первого примера равна 600 пикселям, а для условий второго примера она составляет 800 пикселей. В каждом из случаев суммарная ширина flex-элементов превышает главный размер родительского контейнера (`540px`). Чтобы поместиться в контейнер, flex-элементы должны сжиматься пропорционально их базовым размерам и назначенным коэффициентам сжатия.

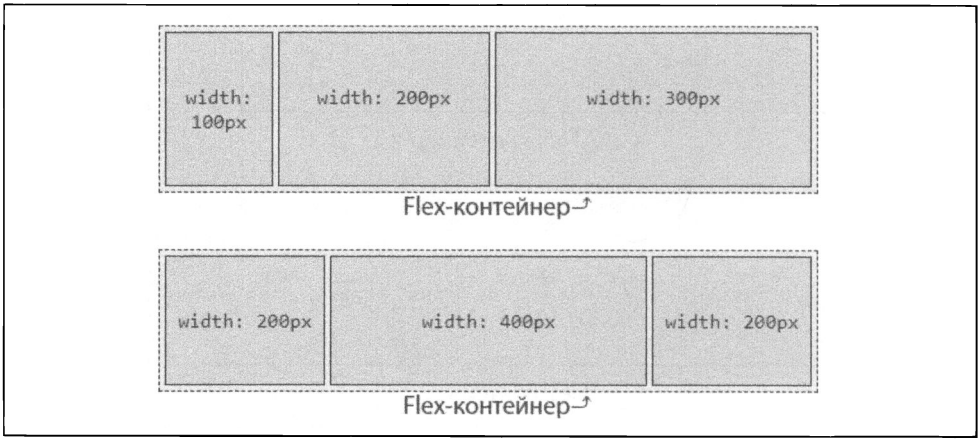


Рис. 12.54. Базовый размер flex-элементов, заданный по умолчанию

Таким образом, в первом примере элементы общей шириной 600 пикселей нужно поместить в контейнер шириной 540 пикселей, что требует сжатия каждого из них на 10% — до ширины соответственно 90, 180 и 270 пикселей. Во втором примере в контейнере с главным размером 540 пикселей нужно разместить элементы суммарной шириной 800 пикселей, что требует их сжатия на 32,5% — до ширины соответственно 135, 270 и 135 пикселей.

Значения в единицах длины

В предыдущих примерах базовый размер самых разных flex-элементов объявлялся с помощью ключевого слова `auto`, по умолчанию представляющего исходную ширину элемента. Существует еще один вариант: представить базовый размер неким точным значением, выраженным в единицах измерения длины, подобно тому, как для этих же целей применялось значение свойства `width` или `height`.

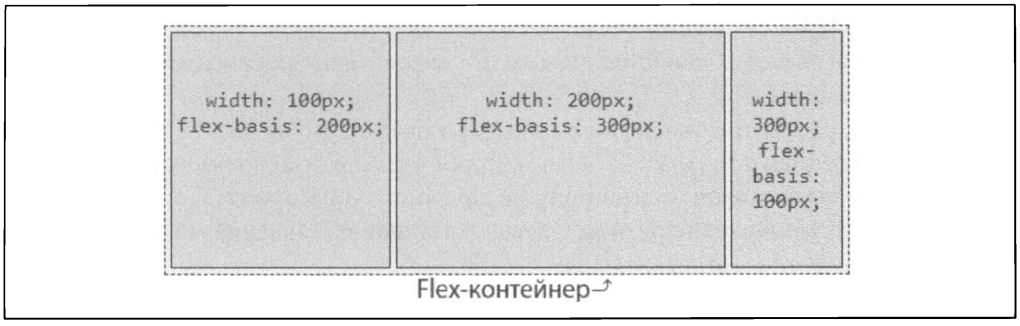


Рис. 12.55. Представление базового размера значением, выраженным в единицах длины

При явном объявлении flex-элементу обоих свойств — `flex-basis` и `width` (`height` при вертикальной направленности главной оси контейнера) — базовый размер будет определяться значением второго (или третьего) из них. Чуть изменим пример, приведенный на рис. 12.54, установив в нем точные базовые размеры flex-элементов и представив их числовыми значениями. Обновленный CSS-код будет выглядеть следующим образом.

```
flex-container {
   width: 540px;
}
item1 {
   width: 100px;
   flex-basis: 300px; /* flex: 0 1 300px; */
}
item2 {
   width: 200px;
   flex-basis: 200px; /* flex: 0 1 200px; */
}
item3 {
   width: 300px;
   flex-basis: 100px; /* flex: 0 1 100px; */
}
```

Теперь ширина flex-элементов, относительно которой рассчитывается их сжатие, определяется явно заданными значениями свойства `flex-basis`. В результате элементы сжимаются до главного размера соответственно 270, 180 и 90 пикселей. Напомним, что в исходном состоянии элементы имели ширину соответственно 300, 200 и 100 пикселей.

Главный размер flex-элементов зависит не только от базового размера, указываемого свойством `flex-basis`, но и от значений некоторых других свойств: `min-width`, `min-height`, `max-width` и `max-height`. Их нужно обязательно учитывать при явном объявлении в стилевых правилах. Например, если к flex-элементу одновременно применить объявления `flex-basis: 100px` и `min-width: 500px`, то степень его сжатия или растягивания будет вычисляться относительно ширины `500px`, даже несмотря на то что базовый размер существенно меньше.

Процентные значения

Процентные значения свойства `flex-basis` определяются относительно главного размера контейнера.

Перейдем к рассмотрению первого примера, показанного на рис. 12.56. Он примечателен объявлением `flex-basis: auto`, которое указывает использовать в качестве базового размера flex-элемента ширину включенного в него текста, составляющую 110 пикселей. В данном конкретном случае указанное объявление можно смело заменить на `flex-basis: 110px`.

```
flex-container {
   width: 540px;
}
flex-item {
   flex: 0 1 100%;
}
```

Базовый размер первых двух элементов из второго примера, показанного на рис. 12.56, также представляется ключевым словом `auto`, что наряду с объявлением `width: auto` равнозначно установке свойства `flex-basis` в значение `content`. Как известно из предыдущих разделов, вычисляемая базовая ширина первых двух элементов также представлена числовым значением `110px`, определяющим ширину содержимого их обоих. Значение свойства `flex-basis` последнего элемента представлено величиной `100%`.

Процентное значение вычисляется относительно главного размера контейнера, в данном случае составляющего 540 пикселей. Третий flex-элемент с базовым размером 100% — не единственный непереносимый элемент своего родительского контейнера. Следовательно, он никогда не будет занимать весь размер контейнера, разве что при установке нулевого коэффициента сжатия, предотвращающего любое уменьшение его ширины, или в случае включения в него непереносимого содержимого, ширина которого превышает размер родительского контейнера.

Не забывайте, что при представлении процентным значением базовый размер элемента вычисляется относительно главного размера flex-контейнера.

Таким образом, согласно условиям задачи, нам требуется включить во flex-контейнер с главным размером 540 пикселей три элемента (без учета значений свойств блочной модели элемента), в каждый из которых добавлено содержимое шириной 110 пикселей. С учетом содержимого в контейнер размером 540 пикселей нужно поместить элементы общей шириной 760 пикселей, что предполагает распределение между ними "отрицательного свободного пространства" шириной 220 пикселей. Коэффициент сжатия рассчитывается следующим образом:

коэффициент сжатия = `220px` ÷ `760px` = 28,95%.

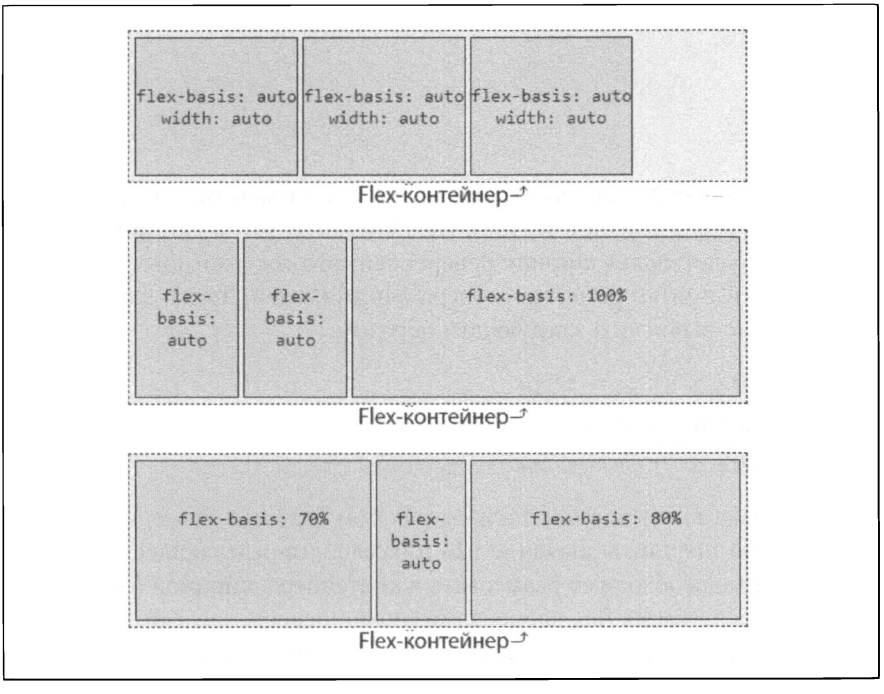


Рис. 12.56. Определение базового размера flex-элемента процентным значением

Итак, каждый flex-элемент будет сжат на 28,95% до 71,05% исходного размера вдоль главной оси контейнера. Конечные значения базового размера вычисляются так, как показано ниже.

Элемент 1 = `110px` × 71,05% = `78.16px`

Элемент 2 = `110px` × 71,05% = `78.16px`

Элемент 3 = `540px` × 71,05% = `383.68px`

Проведенные вычисления показывают, что полностью в контейнер будут помещаться только элементы небольших размеров: содержащие изображения или непереносимый текст не длиннее 78,16 или 383,68 пикселя. Указанные значения определяют максимально возможный размер flex-элементов, при котором содержимое контейнера не выступает за его края. Определение "максимально возможный" предполагает включение в контейнер flex-элементов, сжимаемых до большего размера, если недостаток свободного места компенсируется более сильным сжатием остальных элементов.

В третьем примере на рис. 12.56 объявление `flex-basis: auto` приводит к размещению содержимого второго flex-элемента в трех строках.

```
flex-container {
   width: 540px;
}
item1 {
   flex: 0 1 70%;
}
```

```
item2 {
   flex: 0 1 auto;
}
item3 {
   flex: 0 1 80%;
}
```

В приведенном выше коде базовые размеры flex-элементов представлены значениями `70%`, `auto` и `80%` соответственно. Не забывайте, что в данном случае ключевое слово `auto` представляет ширину непереносимого содержимого, которая составляет 110 пикселей при ширине контейнера 540 пикселей. Тогда значения свойства `flex-basis` можно вычислить следующим образом:

Элемент 1 = 70% × `540px` = `378px`

Элемент 2 = ширина текста "`flex-basis: auto`" = `110px`

Элемент 3 = 80% × `540px` = `432px`

Просуммировав вычисленные таким способом базовые размеры всех трех flex-элементов, можно получить значение 920 пикселей, определяющее общую ширину элементов, которые необходимо разместить в контейнере шириной 540 пикселей. Исходя из этого, между тремя flex-элементами нужно перераспределить область "отрицательного свободного пространства" шириной 380 пикселей. Для вычисления коэффициента сжатия главный размер их контейнера нужно разделить на общую ширину элементов, как показано ниже.

Соотношение ширины = 540px ÷ 920px = 0,587

Дальнейшие вычисления не представляют особой сложности, поскольку сжатие элементов выполняется равномерно. Конечная ширина каждого элемента будет составлять 58,7% размера, который он имел бы в отсутствие сжатия.

Элемент 1 = `378px` × 58,7% = `221.8px`

Элемент 2 = `110px` × 58,7% = `64.6px`

Элемент 3 = `432px` × 58,7% = `253.6px`

Рассмотрим, что изменится при использовании контейнера с другим главным размером, например 1000 пикселей. В таком случае его flex-элементы будут иметь следующие базовые размеры: `700px` (70% × `1000px`), `110px` и `800px` (80% × `1000px`). Суммарный базовый размер элементов будет равен 1610 пикселям.

Соотношение значений ширины = `1000px` ÷ `1610px` = 0,6211

Элемент 1 = `700px` × 62,11% = `434.8px`

Элемент 2 = `110px` × 62,11% = `68.3px`

Элемент 3 = `800px` × 62,11% = `496.9px`

Поскольку базовый размер flex-элементов представляется процентными значениями 70 и 80%, их сумма всегда будет превышать значение 100%. Следовательно,

flex-элементы будут сжиматься независимо от того, насколько широким можно сделать их контейнер.

Если же по некой причине первый элемент не сжимается — при включении в него несжимаемого содержимого или явном запрете через объявление `flex-shrink: 0`, — то его ширина будет вычисляться как 70% от главного размера родительского элемента (в данном случае 378 пикселей). Остальные элементы должны будут уместиться в области шириной 30% от размера контейнера, или 162 пикселя. В конечном счете ширина flex-элементов составит соответственно 378, 32,875 и 129,125 пикселя. Так как ширина неделимого текста, включенного в элемент с объявлением `flex-basis: auto`, равна 42 пикселям, пространство контейнера будет перераспределено между элементами следующим образом: 378, 42 и 120 пикселей. Результат сжатия flex-элементов согласно описанным выше условиям представлен на рис. 12.57.

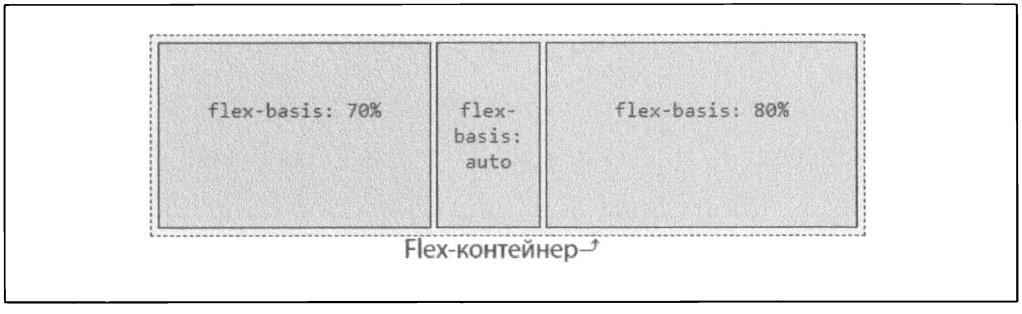


Рис. 12.57. Несмотря на то что процентные значения базового размера flex-элементов вычисляются относительно ширины контейнера, главный размер каждого из них зависит от ширины остальных элементов

Тестирование приведенного выше CSS-кода на реальных документах будет приводить к немного разным результатам, поскольку в каждом из случаев объявление `flex-basis: auto` будет представлять разный базовый размер, зависящий, например, от шрифта, в котором выводится текст документа.

Нулевой базовый размер

Если для flex-элемента не объявлено ни свойство `flex`, ни свойство `flex-basis`, то его базовый размер будет определяться значением `auto`, устанавливаемым по умолчанию. Если же не указать его в объявлении свойства общего назначения `flex`, то flex-элементу назначается нулевой базовый размер. При беглом анализе значения `auto` и `0` могут показаться одинаковыми, хотя результат их применения, особенно нулевого значения, может быть весьма непредсказуемым.

При определении базового размера с помощью ключевого слова `auto` он полностью совпадает с главным размером содержимого элемента. В случае применения объявления `flex-basis: 0;` базовый размер определяется шириной (высотой) доступного пространства контейнера. В каждом из случаев доступное пространство контейнера перераспределяется между всеми его дочерними элементами согласно указанным коэффициентам растягивания.

В случае назначения flex-элементам нулевого базового размера распределение доступного пространства контейнера выполняется без учета их главного размера, устанавливаемого свойством `height`, `width` или `content`, — исключительно на основе заданных коэффициентов растягивания. При этом на конечный размер элементов оказывают влияние свойства `min-width`, `max-width`, `min-height` и `max-height`.

Как показано на рис. 12.58, при автоматическом определении базового размера между flex-элементами распределяется все доступное *свободное* пространство контейнера. Размер каждого из них, как и в предыдущем случае, увеличивается пропорционально заданным коэффициентам растяжения. В первом примере содержимое элементов с объявлением `flex: X X auto` имеет ширину 110 пикселей. При этом между flex-элементами нужно распределить свободное пространство шириной 210 пикселей, разбив его на 6 дольных частей, каждая из которых имеет ширину 35 пикселей. После растягивания flex-элементы получают ширину соответственно 180, 145 и 215 пикселей.

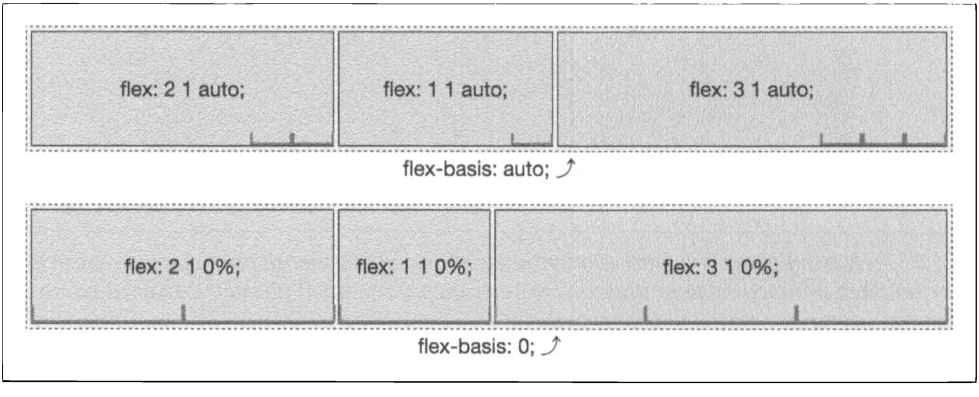


Рис. 12.58. Растягивание flex-элементов при автоматическом определении базовой ширины и установке ее в нулевое значение

Во втором примере flex-элементам задана нулевая базовая ширина, поэтому между элементами будет распределяться все пространство контейнера шириной 540 пикселей. Разделив `540px` на шесть дольных частей, можно получить размер отдельной доли: 90 пикселей. Соответственно, после растягивания flex-элементы будут иметь такие главные размеры: 180, 90 и 270 пикселей. При главном размере 90 пикселей второй элемент не может вмещать содержимое меньшей ширины — в частности, 110 пикселей, как в предыдущем примере.

Свойство общего назначения `flex`

После детального ознакомления с индивидуальными свойствами верстки flex-элементов преимущества свойства общего назначения `flex` становятся вполне очевидными. Оно поддерживает работу со всеми глобальными ключевыми словами, включая `initial`, `auto`, `none`, а также целочисленными значениями. Рассмотрим возможности свойства `flex` на более детальном уровне.

Общие значения свойства `flex`

К общим относятся четыре значения свойства `flex`, представляющие наиболее часто воспроизводимые эффекты.

`flex: initial`

Размер flex-элемента при сжатии определяется относительно значения свойства `width` или `height`, в зависимости от направления главной оси контейнера.

`flex: auto`

Как и в предыдущем случае, размер flex-элемента устанавливается относительно значения свойства `width` или `height`, но применяется как при сжатии, так и при растягивании.

`flex: none`

Элемент не подвергается сжатию и растягиванию, а его размер однозначно передается свойством `width` или `height`.

`flex: <number>`

Представляет коэффициент растягивания целочисленным значением, обнуляя коэффициент сжатия и базовый размер flex-элемента. Свойство `width` или `height` определяет минимальный размер вдоль главной оси flex-элемента, который растягивается только при наличии в контейнере достаточного свободного пространства.

Далее детально рассмотрено каждое из них.

Ключевое слово `initial`

Значение `initial` является глобальным ключевым словом и может передаваться любому свойству. Оно представляет исходное значение свойства, назначаемое ему согласно спецификации по умолчанию. Исходя из этого, можно смело настаивать на идентичности следующих объявлений.

```
flex: initial;
flex: 0 1 auto;
```

С помощью объявления `flex: initial` flex-элементу определяется нулевой коэффициент растягивания, коэффициент сжатия устанавливается в значение `1`, а базовый размер представляется ключевым словом `auto`. С результатом автоматического вычисления базовых размеров flex-элементов можно ознакомиться на рис. 12.59. В первых двух примерах базовый размер flex-элементов представляется значением `content` — их главный размер определяется шириной включенной в них строки символов. Базовый размер flex-элементов в двух последних примерах равен 50 пикселям — к каждому из них применено объявление `width: 50px`. Как видите, передача значения `auto` свойству `flex-basis` в рамках объявления `flex: initial` приводит к представлению базового размера свойством `width` или `height` (при его явном объявлении) либо ключевым словом `content` (во всех остальных случаях).

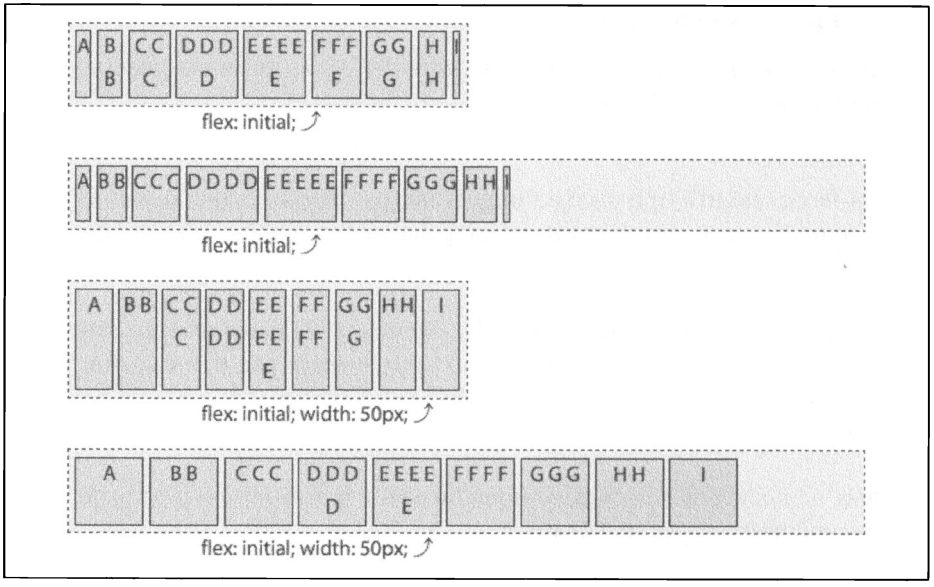


Рис. 12.59. Flex-элемент, свойство `flex` *которого представлено значением* `initial`*, может сжиматься, но не растягиваться*

В первом и третьем примере показано, что при попытке включения во flex-контейнер элементов слишком маленького размера они будут сжиматься до тех пор, пока полностью не поместятся внутри него. В указанных примерах суммарный базовый размер элементов превышает главный размер их контейнера. При этом размер каждого flex-элемента в первом примере зависит от ширины его содержимого и возможности дальнейшего сжатия. Все элементы пропорционально сжимаются согласно заданным коэффициентам, но только до размера непереносимого содержимого. В третьем примере базовый размер всех элементов определяется значением `50px` (исходно устанавливается свойством `width`), поэтому они сжимаются равномерно.

По умолчанию flex-элементы выравниваются по начальной точке главной оси контейнера, что обусловливается свойством `justify-content`, по умолчанию устанавливаемым в значение `flex-start`. На выравнивание flex-элементов нужно обращать внимание только тогда, когда их суммарный размер вдоль главной оси меньше главного размера контейнера и ни один из элементов не подвержен растягиванию.

Ключевое слово `auto`

Объявление `flex: auto` обеспечивает flex-элементы такими же возможностями, как и объявление `flex: initial`, но сразу в обоих направлениях. При его использовании элементы могут не только сжиматься, но и растягиваться, заполняя весь контейнер — конечно, если он располагает достаточным количеством свободного места. Растягивание выполняется до краев контейнера в направлении его главной оси. Следующие две строки CSS-кода абсолютно одинаковые.

```
flex: auto;
flex: 1 1 auto;
```

При передаче свойству `flex` значения `auto` реализуются самые разные сценарии заполнения контейнера элементами (рис. 12.60).

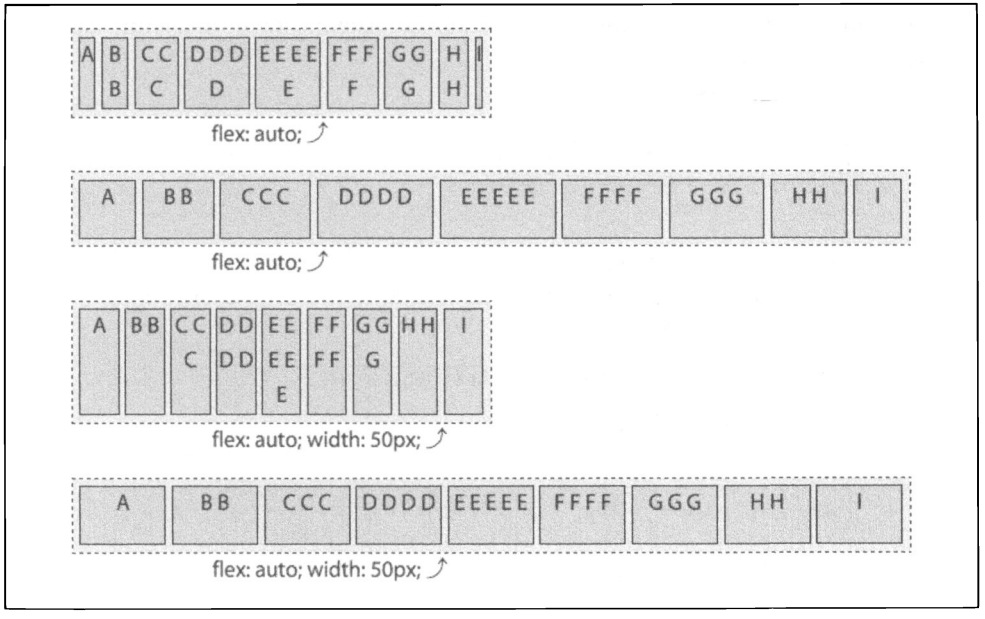


Рис. 12.60. При передаче свойству `flex` значения `auto` элементы могут как сжиматься, так и растягиваться

В первом и третьем примерах, показанных на рис. 12.60, применяются такие же базовый размер и коэффициент сжатия, как и в аналогичных примерах на рис. 12.59, чего не скажешь о втором и четвертом примерах. Благодаря объявлению `flex: auto` к flex-элементам применяется коэффициент растягивания 1, что приводит к их расширению до размеров, при котором они заполняют все пространство контейнера.

Ключевое слово none

Объявление `flex: none` предотвращает изменение размеров flex-элемента, блокируя его сжатие и растягивание. Следующие строки кода обеспечивают выполнение одних и тех же действий.

```
flex: none;
flex: 0 0 auto;
```

Результат использования ключевого слова `none` показан на рис. 12.61.

Как показано в первом и последнем примерах на рис. 12.61, при недостатке свободного места flex-элементы выходят за пределы контейнера (в отличие от ситуаций применения ключевых слов `auto` и `initial`, обеспечивающих flex-элементы коэффициентом растягивания `1`).

В данном случае базовый размер flex-элементов представляется значением `auto`, указывая на его равенство размеру элемента до включения во flex-контейнер. Таким образом, базовый размер определяется значением свойства `width` или `height`,

одновременно представляя главный размер flex-элемента до сжатия или растягивания. В первых двух примерах базовый размер, а потому и исходная ширина элемента, определяются размером его содержимого. Базовый размер flex-элементов в двух последних примерах равен 50 пикселям, поскольку именно такое значение назначено свойству `width` каждого из них.

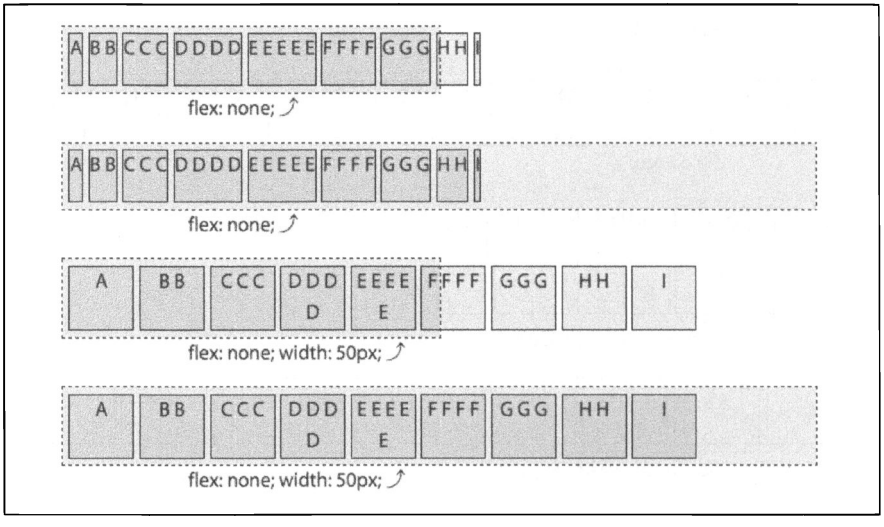


Рис. 12.61. Передача свойству `flex` значения `none` предотвращает как сжатие, так и растягивание элементов

Числовой коэффициент

Если свойство `flex` представляется положительным числовым значением, то оно задает коэффициент растягивания flex-элемента при нулевом базовом размере и коэффициенте сжатия, равном 1. Следующие объявления полностью идентичны.

```
flex: 3;
flex: 3 1 0;
```

Такой подход обеспечивает элемент одной только возможностью растягивания. Коэффициент сжатия указывается номинально: при нулевом базовом размере растягивание всех flex-элементов выполняется равномерно.

В двух первых примерах на рис. 12.62 всем flex-элементам назначен коэффициент растягивания 3. Поскольку их базовый размер представляется значением `0`, они не могут сжиматься, а только растягиваться до тех пор, пока их суммарная ширина не будет равна размеру контейнера в направлении главной оси. Следовательно, при нулевом базовом размере элементы распределяются на всю ширину контейнера. Во втором примере использован контейнер большего размера, поэтому все включенные в него flex-элементы имеют большую ширину.

Справедливости ради стоит заметить, что для обеспечения flex-элементов возможностью растягивания достаточно применить к ним свойство `flex` с любым положительным, не обязательно целочисленным, значением, большим нуля, например `0,1`.

Если положительный коэффициент растягивания назначен только одному flex-элементу, то все доступное пространство контейнера будет заполнено им одним. При растягивании сразу нескольких flex-элементов все доступное пространство будет распределяться между ними пропорционально заданным коэффициентам растягивания.

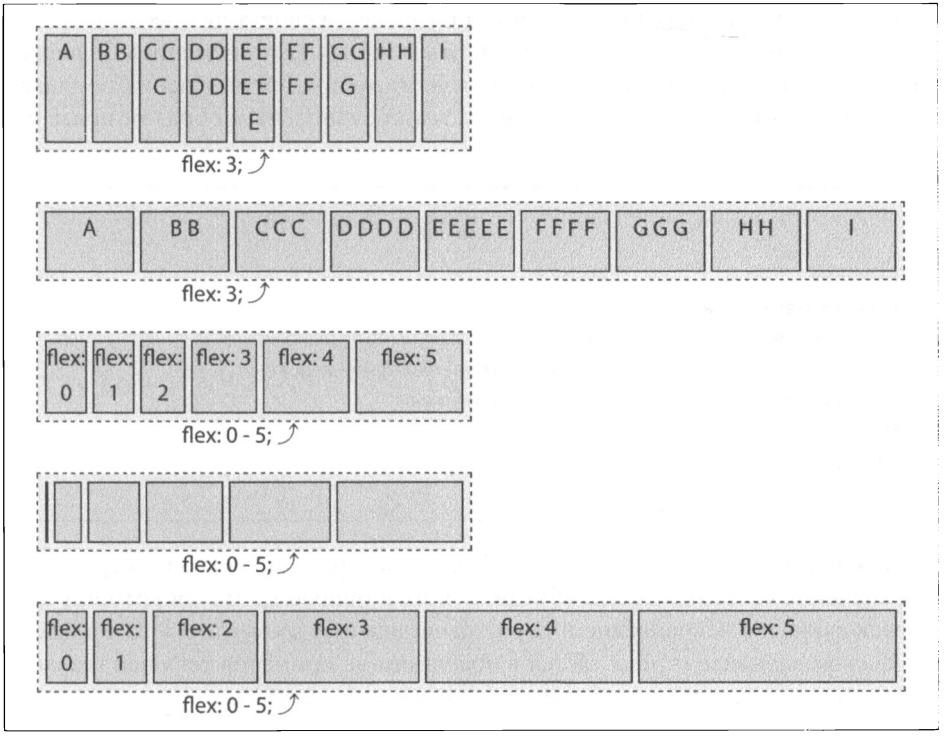


Рис. 12.62. Растягивание элементов при передаче свойству `flex` *положительного числового значения*

В трех последних примерах на рис. 12.62 растягиваются все шесть вложенных в контейнер flex-элементов. Им назначены следующие коэффициенты растягивания: `flex: 0`, `flex: 1`, `flex: 2`, `flex: 3`, `flex: 4` и `flex: 5`. Все элементы также имеют нулевой базовый размер и коэффициент сжатия, равный 1. Размер flex-элементов вдоль главной оси изменяется пропорционально определенным им коэффициентам растягивания. Можно предположить, что flex-элементы с объявлением `flex: 0`, включающие текст "flex: 0", из третьего и четвертого примеров будут иметь нулевой размер, как аналогичные элементы в пятом и шестом примерах, но это далеко не так. По умолчанию они будут сжиматься до размера самого длинного слова или размера, определенного свойством `width` в явном виде.

Для визуального выделения элементов в контейнере к ним добавлены отступы, границы и поля. В результате в четвертом примере представлен даже первый элемент (крайний слева), имеющий нулевую ширину, — он обведен рамкой толщиной 1 пиксель.

Порядок следования элементов

Как правило, порядок расположения flex-элементов устанавливается на этапе добавления их в родительский элемент еще до включения во flex-контейнер. Чтобы изменить направление заполнения контейнера flex-элементами, равно как и flex-элемента строками, на противоположное, применяется свойство `flex-direction`. Но во многих ситуациях порядок следования элементов нужно изменить более кардинально, и его возможностей оказывается недостаточно. Для изменения порядка расположения flex-элементов в контейнере самым произвольным образом применяется свойство `order`.

order

Значение	*<integer>*
Начальное значение	0
Применяется	Flex-элементы и абсолютно позиционируемые дочерние элементы flex-контейнеров
Вычисляется	Согласно определению
Наследуется	Нет
Анимируется	Да

По умолчанию свойству `order` всех flex-элементов одного контейнера присваивается значение `0`, указывающее объединять их в общую группу, в которой поддерживаются такие же направление и порядок следования элементов, как и в исходном родительском элементе (применяется в большинстве примеров текущей главы).

Для изменения визуального порядка следования свойству `order` flex-элемента нужно передать ненулевое целочисленное значение. Объявление свойства `order` для элемента, не являющегося дочерним по отношению к flex-контейнеру, будет проигнорировано пользовательским агентом.

Значение свойства `order` указывает *порядковую группу* flex-элемента. Flex-элементы, относящиеся к порядковой группе с отрицательным идентификатором, располагаются перед элементом нулевой группы (образуется по умолчанию), которые в свою очередь предшествуют flex-элементам, включенным в группу с положительным идентификатором. Учтите, что свойство `order` изменяет только порядок представления flex-элементов в документе, а не действительный порядок их следования в контейнере, исходно определяемый HTML-кодом.

Например, следующий код позволяет расположить седьмой элемент в начале группы, состоящей из 12 flex-элементов, а шестой элемент — поместить в ее конце.

```
ul {
   display: inline-flex;
}
li:nth-of-type(6) {
   order: 1;
}
li:nth-of-type(7) {
```

```
    order: -1;
}
```

В данном случае нужно изменить положение только шестого и седьмого элементов, обеспечив неизменность заданного по умолчанию (`order: 0`) порядка следования всех остальных flex-элементов. Результат выполнения приведенного выше кода показан на рис. 12.63.

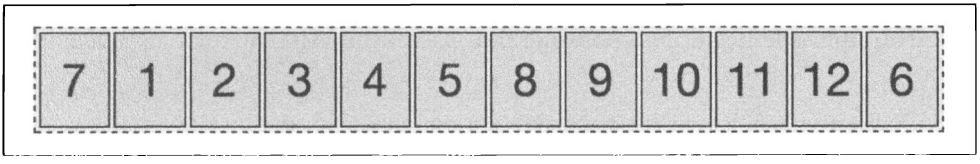


Рис. 12.63. Изменение порядка следования flex-элементов с помощью свойства `order`

Расположение седьмого элемента в начале последовательности обеспечивается отрицательным значением свойства `order`. Положение шестого элемента, наоборот, определяется положительным значением, которое больше не только значения `0`, определяемого по умолчанию, но и значения свойства `order` всех остальных flex-элементов контейнера. Исходя из вышесказанного, он располагается в конце последовательности. Порядок расположения остальных flex-элементов, согласно заданному по умолчанию значению `0`, остается неизменным, поэтому они располагаются в исходной последовательности, предваряемой и заканчиваемой специально объявленными элементами.

Порядок заполнения flex-контейнера элементами определяется номерами их групп — в его начале располагаются элементы порядковой группы с наименьшим номером, а элементы порядковой группы с наибольшим номером находятся в его конце. При задании одного и того же значения свойства `order` сразу нескольким flex-элементам они будут включаться в общую порядковую группу. Таким образом, в пределах одной группы элементы располагаются в исходно заданном для них порядке, а положение самих групп определяется номером, объявленным в свойстве `order`. Рассмотрим пример.

```
ul {
    display: inline-flex;
    background-color: rgba(0,0,0,0.1);
}
li:nth-of-type(3n-1) {
    order: 3;
    background-color: rgba(0,0,0,0.2);
}
li:nth-of-type(3n+1) {
    order: -1;
    background-color: rgba(0,0,0,0.4);
}
```

Порядок расположения flex-элементов в пределах отдельно рассматриваемой группы определяется ее исходной позицией в группе по умолчанию, представленной номером `0`. Результат выполнения приведенного выше CSS-кода показан на рис. 12.64.

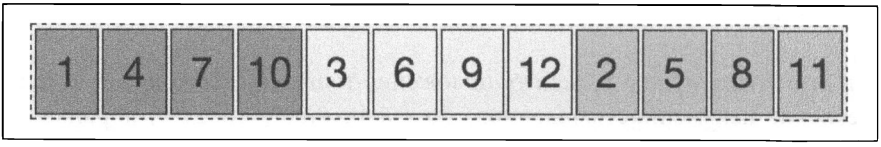


Рис. 12.64. При включении в другую порядковую группу расположение flex-элементов будет зависеть от позиции, занимаемой ими в исходной группе

В этом примере позиционирование flex-элементов выполняется следующим образом.

- Элементы 2, 5, 8 и 11 перемещаются в порядковую группу с номером `3`, располагаясь на фоне с прозрачностью 80% (непрозрачность — 20%).
- Элементы 1, 4, 7 и 10 относятся к порядковой группе с номером `-1`, получая фон с прозрачностью 60% (непрозрачность — 40%).
- Элементы 3, 6, 9 и 12 не выделяются в отдельную порядковую группу, поэтому относятся к группе с номером `0`, определенной для контейнера по умолчанию.

Таким образом, в контейнере имеются три порядковые группы flex-элементов. Они заполняют контейнер в порядке `-1`, `0`, `3`. Положение элементов в пределах каждой отдельно рассматриваемой группы определяется их относительным расположением в исходной (нулевой) порядковой группе.

Учтите, что изменение порядка следования flex-элементов выполняется только на уровне визуального восприятия. Именно в таком порядке они будут показаны пользователям, создавая иллюзию действительной структурной реорганизации документа. В действительности она не выполняется — свойство `order` изменяет только порядок визуализации flex-элементов в контейнере, но не его структуру.

Изменения, привносимые в документ с помощью свойства `order`, не оказывают влияния на порядок перехода между гиперссылками, добавленными в документ. Если бы каждому из flex-элементов, представленных на рис. 12.64, назначалась отдельная гиперссылка, то переход между ним (нажатием клавиши <Tab>) осуществлялся бы в порядке их указания в HTML-коде без учета значений свойства `order`.

Навигационная панель (модифицированная)

Вернувшись к рассмотрению навигационной панели, созданной в начале главы (см. рис. 12.2), изменим ее так, чтобы выбранная вкладка всегда располагалась на первом месте (рис. 12.65).

```
nav {
   display: flex;
   justify-content: flex-end;
   border-bottom: 1px solid #ddd;
}
a {
   margin: 0 5px;
   padding: 5px 15px;
   border-radius: 3px 3px 0 0;
   background-color: #ddd;
```

```
   text-decoration: none;
   color: black;
}
a:hover {
   background-color: #bbb;
   text-decoration: underline;
}
a.active {
   order: -1;
   background-color: #999;
}
<nav>
   <a href="/">Home</a>
   <a href="/about">About</a>
   <a class="active">Blog</a>
   <a href="/jobs">Careers</a>
   <a href="/contact">Contact Us</a>
</nav>
```

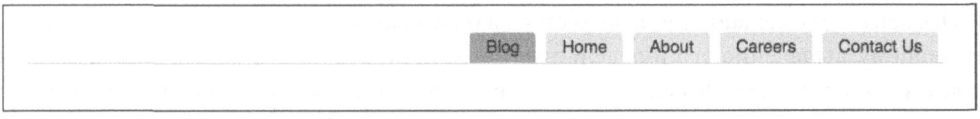


Рис. 12.65. Изменения порядка визуального представления вкладок

В приведенном выше коде элемент текущей активной вкладки лишается атрибута `href` и выделяется в класс `.active`. Кроме того, к нему применяется свойство `order` со значением `-1`, обеспечивая расположение перед исходной группой вкладок с нулевым порядковым номером.

Причина удаления атрибута `href` у элемента текущей вкладки очень простая. Поскольку вкладка представляет активный документ, добавлять в нее гиперссылку на саму себя нет никакой необходимости. К тому же, что более важно, при использовании клавиши <Tab> пользователь будет переходить между названиями Blog, Home, About, Careers и Contact Us (список начинается с вкладки Blog), хотя в действительности переход будет осуществляться между вкладками Home, About, Blog, Careers и Contact Us — согласно действительному, а не визуальному порядку следования flex-элементов.

Свойство `order` можно применять для вывода главной области документа перед боковой панелью или панелью навигации в стандартных трехколоночных документах при отображении их на экранах мобильных устройств.

Конечно, с его помощью невозможно изменить код разметки документа, в частности, расположив подвал сайта над его заголовком, поскольку свойство `order` отвечает только за порядок визуального представления элементов документа. При этом структура документа и порядок расположения элементов в нем всецело определяются разметкой, выполненной, например, с помощью HTML.

```
<header></header>
<main>
   <article></article>
   <aside></aside>
   <nav></nav>
</main>
<footer></footer>
```

```
<header></header>
<main>
   <nav></nav>
   <article></article>
   <aside></aside>
</main>
<footer></footer>
```

Исходно дизайн веб-сайта обусловливается разметкой его документов, как показано в примере справа приведенного выше кода, представляющего страницу, изображенную на рис. 12.50. Наряду с этим более оптимальным считается вариант размещения основного содержимого, в данном случае представляемого элементом `article`, в начале списка. Согласно такому способу позиционирования элемент `article` будет первым отображаться на экранах мобильных устройств, в списках результатов поисковых систем и программах чтения с экрана. В те же время при просмотре документа на экранах полноразмерных мониторов нужно обеспечить вывод содержимого элемента `article` в средней колонке документа.

```
main {
   display: flex;
}
main > nav {
   order: -1;
}
```

Элемент `nav` смещается в начало списка благодаря объявлению `order: -1`, обеспечивающему выделение его в отдельную порядковую группу с наименьшим идентификатором. При этом элементы `article` и `aside` по-прежнему относятся к порядковой группе, образованной по умолчанию и снабженной нулевым идентификатором (`order: 0`).

Как известно, при включении в одну порядковую группу сразу нескольких flex-элементов они будут располагаться вдоль главной оси контейнера в порядке разметки в документе. Следовательно, элемент `article` будет всегда отображаться перед элементом `aside`. Многие авторы при изменении порядка представления хотя бы одного flex-элемента предпочитают в явном виде объявлять значения свойства `order` для всех остальных элементов flex-контейнера, существенно повышая удобочитаемость кода. В данном случае это будет выглядеть так.

```
main {
   display: flex;
}
main > nav {
   order: 1;
}
main > article {
   order: 2;
}
main > aside {
   order: 3;
}
```

До появления в CSS инструментов гибкой верстки макеты трехколоночных документов чаще всего создавались с помощью свойства `float`. Несмотря на корректность такого подхода, свойства гибкой верстки намного проще и функциональнее, особенно при создании документов, включающих всего три колонки (в данном случае `aside`, `nav` и `article`) одинаковой высоты.

ГЛАВА 13

Верстка по сетке

Долгое время технология CSS не могла избавиться от главного недостатка, наследуемого от самых ранних спецификаций (а ведь ее история насчитывает более 20 лет непрерывной эволюции). Однако методы верстки постоянно совершенствуются, CSS "обрастает" более совершенными инструментами, подобными свойствам `float` и `clear`, позволяющим, хоть и с помощью специальных ухищрений, добиться требуемого результата. Ситуация сдвинулась с мертвой точки с появлением гибких контейнеров, но и они обладают ограниченной областью применения — лучше всего они подходят для верстки панелей навигации.

И лишь с появлением сеток CSS получила статус *полнофункциональной* системы верстки документов. Делая упор на упорядочение и позиционирование элементов документа в виде колонок и рядов, технология верстки по сетке воскрешает в памяти уже порядком подзабытую методику табличной верстки, хотя и не является таковой по своей сути. Ее инструменты намного проще в использовании, а возможности — значительно шире. При использовании сеток элементы можно визуализировать в произвольном порядке, отличном от задаваемого разметкой документа. Создание повторяющихся шаблонов из линий сетки, привязка элементов к линиям сетки, вложение сеток в другие сетки, выравнивание таблиц и flex-контейнеров на сетке, а также решение многих других задач стало возможным исключительно благодаря появлению в спецификации CSS3 модуля Grid Layout.

Возможность верстки по сетке стала знаковой и наиболее востребованной в CSS за всю историю спецификации. В процессе ее изучения вам придется не только ознакомиться с новыми инструментами, но и отучиться использовать обходные методики и ухищрения, без которых еще совсем недавно невозможно было представить верстку ни одного серьезного проекта.

Создание grid-контейнера

Первый этап верстки по сетке заключается в объявлении *grid-контейнера*, или контейнера сетки. Во многом он подобен обычному контейнеру, представляющему элемент в классическом способе позиционирования или при гибкой верстке, требующей использования flex-контейнеров. В свою очередь grid-контейнеры представляют элементы, относящиеся к отдельному, *сеточному контексту форматирования*.

На самом простом уровне верстка по сетке является двухмерным продолжением концепций, реализованных в модуле CSS Flexbox Layout. По аналогии с гибкими элементами, заключенными во flex-контейнеры, дочерние элементы grid-контейнера называются *grid-элементами*, или элементами на сетке. Но при этом их дочерние элементы не становятся grid-элементами, сохраняя обычный контекст форматирования, хотя ничто не запрещает объявить grid-контейнером любой элемент, размещаемый на сетке. В последнем случае образуется конструкция, в которой grid-контейнер со всеми дочерними элементами вкладывается в grid-контейнер более высокого уровня, содержащий собственные дочерние элементы. Такой подход открывает необычайно широкие возможности по размещению сеток внутри других сеток сколь угодно глубокого уровня вложения. (В спецификацию CSS Grid Layout также включено понятие *подсетки*, отличное от определения вложенного grid-контейнера, с которым вы ознакомитесь далее.)

Сетки, по которым в CSS осуществляется верстка документов, бывают двух типов: *регулярные* и *строчные*. Тип контейнера сетки объявляется в свойстве `display` с помощью ключевого слова `grid` или `inline-grid`: первое значение представляет сетку контейнером блочного, а второе — строчного типа. Различие между блочным и строчным grid-контейнерами наглядно проиллюстрировано на рис. 13.1.

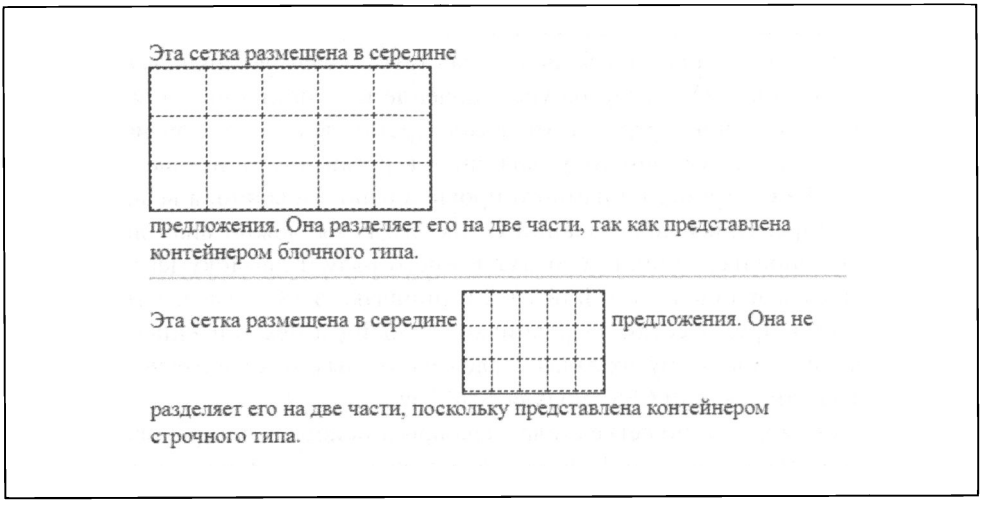


Рис. 13.1. Grid-контейнеры блочного и строчного уровня

Функционально указанные значения во многом подобны значениям `block` и `inline-block` свойства `display`. Стоит заметить, что в большинстве случаев верстка документа выполняется по сеткам блочного типа, хотя строчные grid-контейнеры также находят свое применение.

Согласно спецификации, grid-контейнеры, созданные с помощью объявления `display: grid`, нельзя строго сопоставлять с контейнерами блочного типа. Несмотря на подобность принципов верстки блочных и grid-контейнеров, между ними есть ряд существенных отличий, которые нужно обязательно принимать к сведению.

Первое из них заключается в невозможности перекрывания grid-контейнеров элементами, выравниваемыми с помощью свойства `float`. На практике это означает, что контейнеры сетки никогда не располагаются под выравниваемыми элементами подобно регулярным блочным элементам, как показано на рис. 13.2.

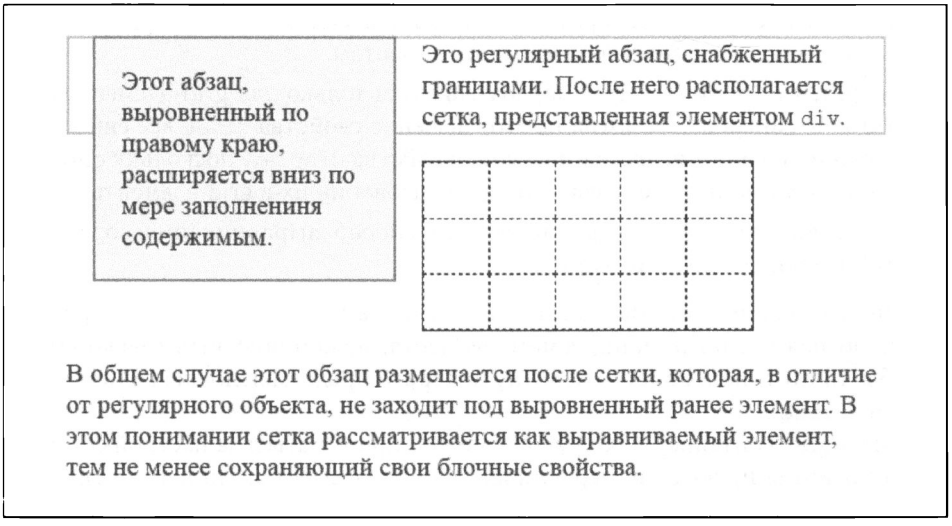


Рис. 13.2. Различия во взаимодействии выравниваемых элементов с блочными контейнерами и контейнерами сетки

Кроме того, отступы grid-контейнеров не схлопываются с отступами дочерних элементов, что по умолчанию свойственно регулярным блочным элементам. Например, верхний отступ первого элемента упорядоченного списка по умолчанию всегда схлопывается с верхним отступом родительского элемента, представляемого блочным контейнером. При этом верхний отступ grid-элемента не схлопывается с верхним отступом своего контейнера ни при каких обстоятельствах. Данное отличие прекрасно проиллюстрировано на рис. 13.3.

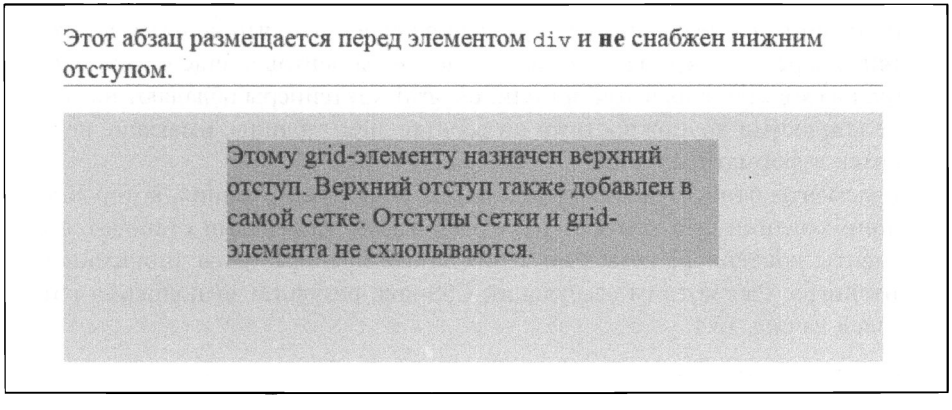


Рис. 13.3. Схлопывание и сохранение отступов у дочерних элементов регулярного и grid-контейнеров

К grid-контейнерам и их элементам не применим целый ряд стилевых свойств и функций, объявляемых для регулярных элементов.

- Все свойства семейства `column` (`column-count`, `columns` и т.п.), объявленные для контейнера сетки, полностью игнорируются.
- Подобным образом игнорируются любые псевдоэлементы `::first-line` и `::first-letter`, применяемые к grid-элементам.
- Свойства `float` и `clear` не обрабатываются только для grid-элементов, но не grid-контейнеров. Несмотря на это, значение свойства `float` все еще оказывает влияние на вычисляемое значение свойства `display`, поскольку оно применяется к дочерним элементам до преобразования их в grid-элементы.
- Свойство `vertical-align` определяет способ выравнивания содержимого grid-элементов, но не их самих.

Наконец, объявив свойство `display` со значением `inline-grid` для выравниваемого или абсолютно позиционируемого элемента, можно получить только grid-контейнер блочного типа (свойство `display` которого представлено значением `grid` без префикса `inline`).

Объявив grid-контейнер в документе, можно приниматься за настройку сетки, по которой позиционируются и выравниваются grid-элементы. Но перед этим нужно рассмотреть основные понятия и определения.

Основные определения

Выше был приведен только краткий обзор концепции grid-контейнеров и grid-элементов. Для детального описания принципов верстки по сетке применяются строгие термины и определения, с которыми вам предстоит ознакомиться в этом разделе. Как вы уже знаете, grid-контейнер, или контейнер сетки, объявляет в документе сеточный контекст форматирования, определяя область, верстка элементов внутри которой осуществляется по специально заданной сетке, в противоположность правилам, принятым для блочной модели. Ситуация сходна с применением к элементу свойства `display` со значением `table` и объявлением табличного контекста форматирования. Такое сравнение справедливо еще и тем, что верстка по сетке, как и табличная верстка, предполагает упорядочение элементов в виде колонок и рядов. Несмотря на сходство ключевых принципов, grid-контейнеры обладают несоизмеримо более широкими возможностями по верстке, чем таблицы, имеющие несколько иной контекст форматирования.

Grid-элементы относятся к тому же контексту форматирования, который объявляется при создании grid-контейнера. Обычно grid-элементами становятся дочерние элементы контейнера сетки, но в них также преобразуется анонимный текст grid-контейнера. Рассмотрим следующий CSS-код, результат выполнения которого представлен на рис. 13.4.

```
#warning {display: grid;
   background: #FCC; padding: 0.5em;
   grid-template-rows: 1fr;
   grid-template-columns: repeat(7, 1fr);}
```

```
<p id="warning"><img src="warning.svg"><strong>Примечание:</strong>
Этот элемент представляет <em>grid-контейнер,</em> содержащий
несколько <em>grid-элементов</em>.</p>
```

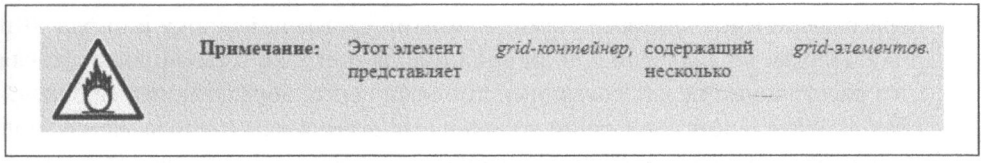


Рис. 13.4. Элементы на сетке

Обратите внимание на то, что grid-элементами представляются все дочерние элементы и расположенные между ними анонимные текстовые фрагменты родительского элемента. Контейнер сетки включает всего семь элементов, среди которых только одно изображение. Остальные grid-элементы представляют исключительно текстовые фрагменты и элементы. Все они подлежат верстке по сетке, хотя описанные далее свойства очень сложно (или абсолютно невозможно) применять для форматирования анонимного текста.

Свойства `grid-template-rows` и `grid-template-columns` будут рассмотрены в следующем разделе.

Чтобы понять, каким образом работают свойства, отвечающие за верстку элементов по сетке, необходимо ознакомиться с основными компонентами сетки, изображенными на рис. 13.5.

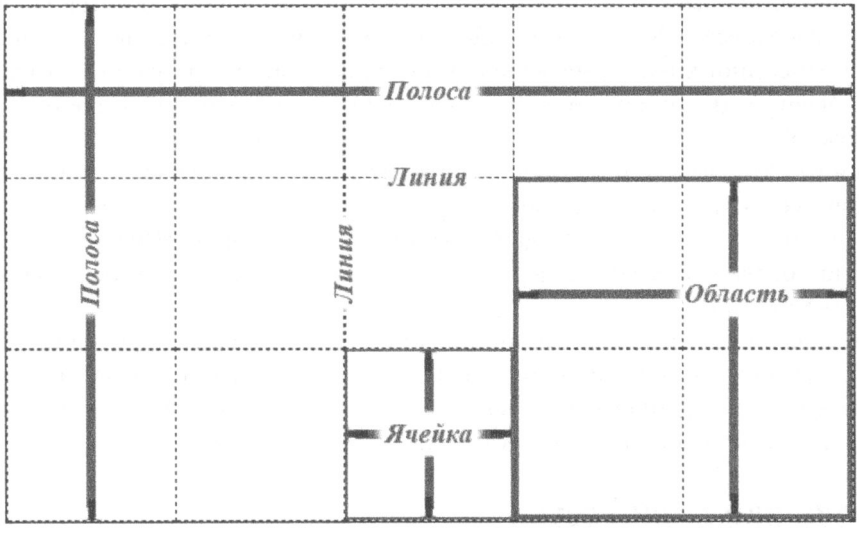


Рис. 13.5. Структурные компоненты сетки

Базовый компонент любой сетки — это *линия*. Линии сетки определяют расположение всех остальных ее компонентов. Рассмотрим их более детально.

- *Полоса сетки* представляет собой область, ограниченную двумя соседними непересекающимися линиями сетки, — иными словами, колонку или ряд. Ширина полосы, по аналогии с шириной столбца и строки в таблицах, определяется расстоянием между соседними линиями сетки, образующими ее боковые края. В буквальном понимании колонки grid-контейнера располагаются вдоль *оси блочных элементов*, а ряды — вдоль *оси строчных элементов* (в системах письма слева направо).
- *Ячейка сетки*, по аналогии с ячейкой таблицы, представляет область, образованную пересекающимися соседними линиями сетки, через которую не проходят никакие другие линии сетки. Она является наименьшим неделимым компонентом сетки. Стилевые свойства верстки по сетке *не работают* с ячейками сетки напрямую — ни одно из них не обеспечивает привязку grid-элементов к ячейкам сетки (см. следующий пункт).
- Область сетки ограничивается четырьмя пересекающимися линиями сетки и включает одну или несколько ячеек. Она может состоять всего из одной ячейки или же заключать все пространство сетки grid-контейнера. Области сетки *подлежат* непосредственному управлению с помощью стилевых свойств верстки по сетке, что позволяет напрямую объявлять их в правилах и привязывать к ним отдельные grid-элементы.

Важно понимать, что полосы, ячейки, колонки, ряды и области сетки образуются линиями сетки и, что более важно, исходно не привязываются к grid-элементам. Спецификация не требует обязательного заполнения областей сетки элементами: во многих случаях большинство или даже все без исключения области сетки полностью лишены содержимого. Кроме того, grid-элементы могут накладываться друг на друга, как при отнесении к перекрывающимся областями, так и в результате объявления такой конфигурации линий сетки, в которой избежать перекрытия элементов просто невозможно.

Помните, что сетка grid-контейнера строится из произвольного количества линий. При необходимости сетку можно представить исключительно вертикальными линиями, обеспечивающими создание одних только колонок. (Подобным образом сетка grid-контейнера может содержать всего одну колонку, включающую большое количество рядов.)

Полная свобода в размещении линий сетки далеко не всегда приводит к ожидаемому результату. Если линии сетки располагаются так, что элементы размещаются вне колонок и рядов, объявленных в grid-контейнере, то в него автоматически будут добавлены новые колонки и ряды, устраняющие этот недостаток.

Размещение линий сетки

Как оказалось, размещение линий сеток представляет собой весьма сложную задачу. Сложности не связаны с запутанностью общей концепции, а вызваны большим

количеством способов решения одних и тех же задач, каждый из которых опирается на собственный синтаксис.

Рассмотрение инструментов верстки по сетке лучше всего начать с изучения двух свойств, обладающих схожим возможностями.

grid-template-rows, grid-template-columns

Значение	`none` | *<track-list>* | *<auto-track-list>*
Начальное значение	`none`
Применяется	Grid-контейнер
Процентное значение	Для свойства `grid-template-columns` относительно строчного размера (обычно ширины) grid-контейнера, а для свойства `grid-template-rows` — относительно его блочного размера (обычно высоты)
Вычисляется	Согласно определению; в абсолютных единицах длины
Наследуется	Нет
Анимируется	Нет

Эти свойства предназначаются для создания линий, определяющих шаблон сетки, который в спецификации CSS называется явной сеткой. Положение линий сетки играет основополагающую роль — при неправильном их размещении нарушается верстка всего документа.

При разработке собственных макетов хорошей идеей будет сначала нарисовать его эскиз на листе бумаги (или в одном из графических редакторов), обозначив на нем точное положение и поведение линий сетки, и только после этого приступать к написанию CSS-кода, отвечающего за наполнение grid-контейнера элементами.

Значения *<track-list>* и *<auto-track-list>* имеют сложный многоуровневый синтаксис, изучение которого потребует много времени и усилий, поэтому их назначение лучше всего рассматривать на реальных примерах. Существует множество способов установки положения линий сетки, но все они основаны на сходных базовых концепциях.

Во-первых, линии сетки нумерованные и при необходимости могут снабжаться индивидуальными именами, как показано на рис. 13.6. В стилевых правилах к линиям сетки можно обращаться по номеру, по имени или сразу по обоим идентификаторам. В частности, действие растягивания можно назначить grid-элементам, находящимся на пересечении колонок с `3` по `steve` и рядов — с `skylight` по `2`.

Во-вторых, линия сетки может иметь сразу несколько имен, но в стилевом правиле одновременно нужно использовать только одно из них. Несмотря на кажущуюся несуразность, в отдельных ситуациях назначение линиям сетки сразу нескольких идентификаторов оказывается очень полезным.

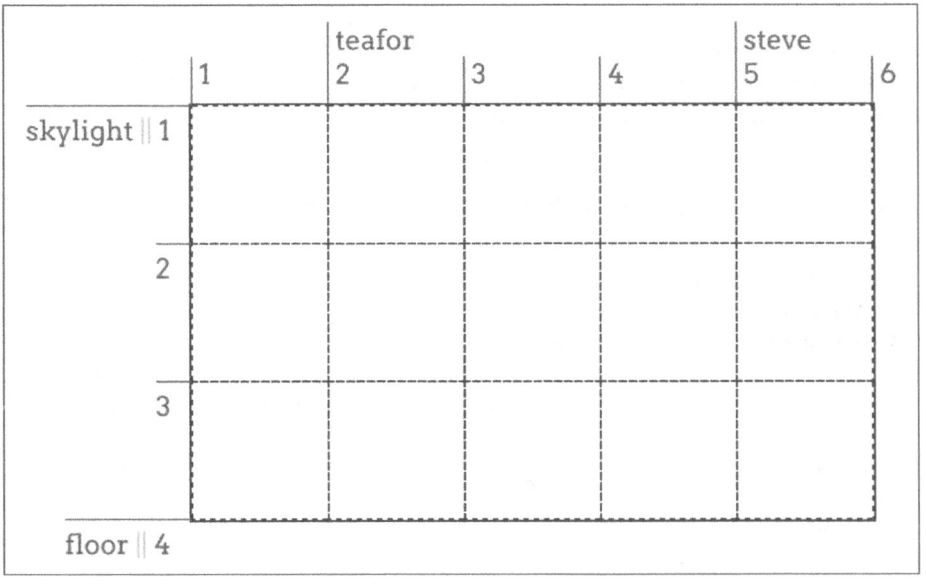


Рис. 13.6. Номера и имена линий сетки

Чтобы избежать совпадений с ключевыми словами, в сетке, показанной на рис. 13.6, умышленно использованы абсолютно несуразные имена. Если, например, в качестве имени первой линии сетки использовать название "start", то его можно принять за несуществующее ключевое слово `start`, указывающее, как можно предположить, на начало сетки. Чтобы обозначить начальную и конечную линии сетки именами соответственно "start" и "end", их нужно назначить вручную. Вскоре вы узнаете, как это делается.

Для создания шаблона сетки также применяются несколько методик. В следующих главах мы рассмотрим их все, начиная с самой простой и заканчивая наиболее сложной.

Колонки фиксированной ширины

В качестве первого примера создадим сетку с колонками строго заданной ширины. Их размер можно указывать как в единицах измерения длины, например пикселях и `em`, так и в процентах. В данном контексте фиксированной называется такая ширина, при которой расстояние между линиями сетки, образующими колонки, не зависит от размера их содержимого.

В частности, для создания сетки, состоящей из трех колонок фиксированной ширины, можно применять такое стилевое форматирование.

```
#grid {display: grid;
   grid-template-columns: 200px 50% 100px;}
```

После применения такого правила первая линия сетки располагается на расстоянии 200 пикселей от начала (по умолчанию левый край) grid-контейнера. При этом вторая линия сетки отстоит от первой линии на расстоянии, равном половине

ширины grid-контейнера, а третья смещается на 100 пикселей вправо от второй линии, как показано на рис. 13.7.

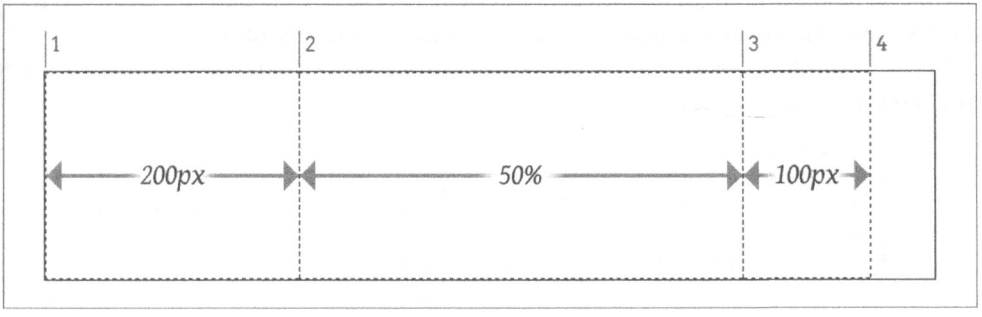


Рис. 13.7. Позиционирование линий сетки

Такое форматирование обеспечивает постоянство расположения линий сетки при наполнении колонок содержимым — вторая колонка будет изменять свой размер только при изменении ширины всего контейнера. Независимо от степени заполнения второй колонки ее ширина всегда будет составлять ровно половину ширины контейнера.

Обратите внимание на то, что третья линия сетки не совмещена с правым краем контейнера. Так и должно быть! О том, как расширить третью колонку до правого края grid-контейнера, рассказывается дальше.

Теперь рассмотрим, каким образом линиям сетки назначаются имена. Все они в совершенно произвольном количестве указываются в квадратных скобках после значений, определяющих положение соответствующих линий сетки. Вот как можно снабдить именами линии сетки, образованной предыдущим стилевым правилом (рис. 13.8).

```
#grid {display: grid;
   grid-template-columns:
      [start col-a] 200px [col-b] 50% [col-c] 100px [stop end last];
   }
```

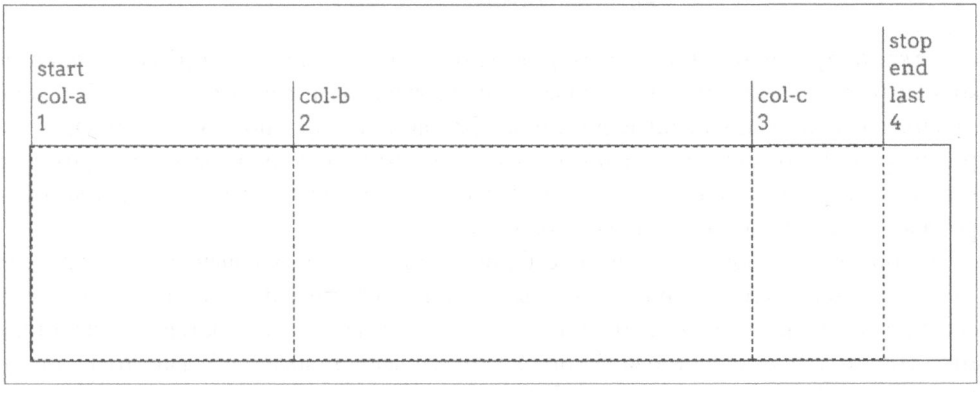


Рис. 13.8. Линии сетки, снабженные именами

Названия линий сетки помогают обозначить края колонок и рядов сетки, ширина которых указывается между соответствующими значениями, заключенными в квадратные скобки. В приведенном выше примере сетка состоит из трех колонок, полученных в результате объявления в ней трех именованных линий.

Ряды сетки объявляются так же, как и колонки, но с помощью отдельного стилевого свойства (рис. 13.9).

```
#grid {display: grid;
   grid-template-columns:
      [start col-a] 200px [col-b] 50% [col-c] 100px [stop end last];
   grid-template-rows:
      [start masthead] 3em [content] 80% [footer] 2em [stop end];
}
```

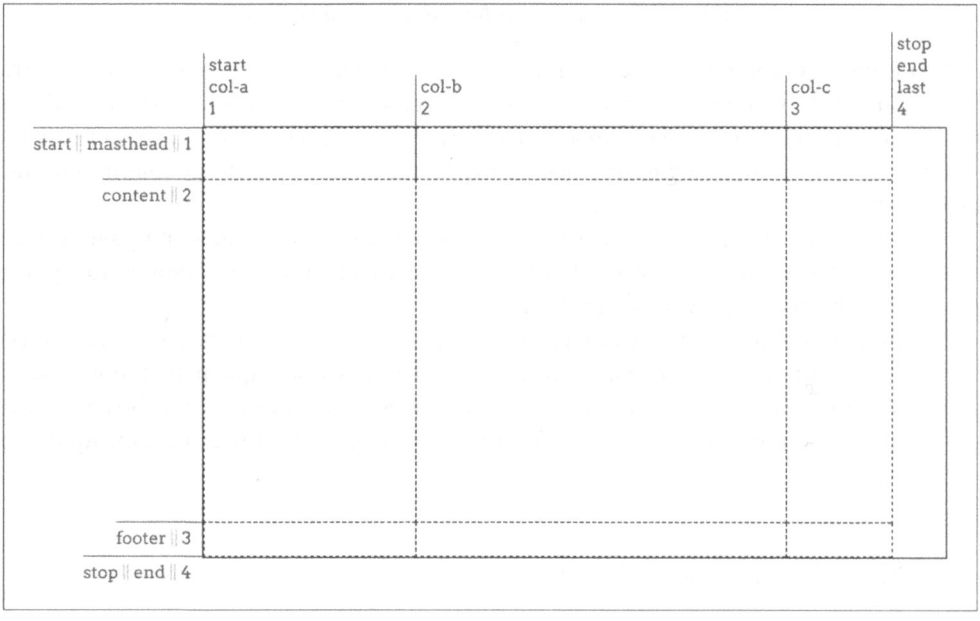


Рис. 13.9. Полноценная сетка

Сетка, полученная с помощью приведенного выше правила, примечательна несколькими особенностями. Во-первых, в ней ключевыми словами `start` и `end` обозначены начальные и конечные линии как рядов, так и колонок. В этом нет никаких противоречий, поскольку горизонтальные и вертикальные полосы характеризуются независимыми пространствами имен. В каждом из них допускается использовать произвольное количество одинаковых названий.

Второй момент, обращающий на себя внимание, — это обозначение высоты ряда `content` процентным значением. Оно вычисляется относительно высоты grid-контейнера: при его полном размере 500 пикселей высота ряда `content` составляет 400 пикселей. В общем случае использование процентных значений целесообразно только тогда, когда известен точный размер контейнера, что возможно далеко не во всех проектах.

Можно подумать, что при установке высоты ряда в значение 100% он будет расширяться на все пространство grid-контейнера, но это не соответствует действительности. Как видно на рис. 13.10, ряд `content`, как и предписывается объявлением, имеет высоту контейнера, но не занимает все его пространство, а выступает за нижний край на высоту первого ряда, вторгаясь в область подвала страницы.

```
#grid {display: grid;
   grid-template-columns:
      [start col-a] 200px [col-b] 50% [col-c] 100px [stop end last];
   grid-template-rows:
      [start masthead] 3em [content] 100% [footer] 2em [stop end];
}
```

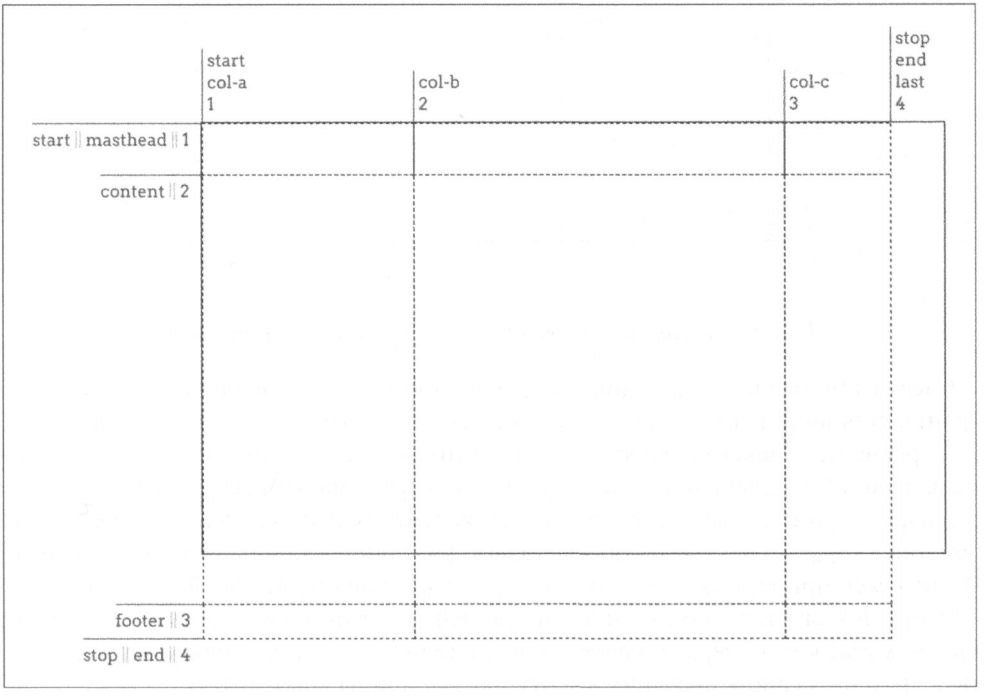


Рис. 13.10. Переполнение grid-контейнера

Один из способов (не обязательно самый оптимальный) предотвращения такого поведения заключается в использовании функции `minmax(a,b)` для задания минимально и максимально возможного размера колонок и рядов сетки. В ней параметр `a` определяет минимальный, а аргумент `b` — максимальный размер полосы сетки.

```
#grid {display: grid;
   grid-template-columns:
      [start col-a] 200px [col-b] 50% [col-c] 100px [stop end last];
   grid-template-rows:
      [start masthead] 3em [content] minmax(3em,100%) [footer]
         2em [stop end];
}
```

Приведенное выше стилевое правило создает сетку, в которой высота ряда `content` не может быть меньше `3em` и больше вертикального размера контейнера. Оно позволяет увеличивать высоту ряда до тех пор, пока на странице остается достаточно свободного пространства для полноценного размещения рядов `masthead` и `footer`. При этом нижний предел высоты ряда, `3em`, достигается в очень редких случаях. Один из возможных вариантов форматирования сетки показан на рис. 13.11.

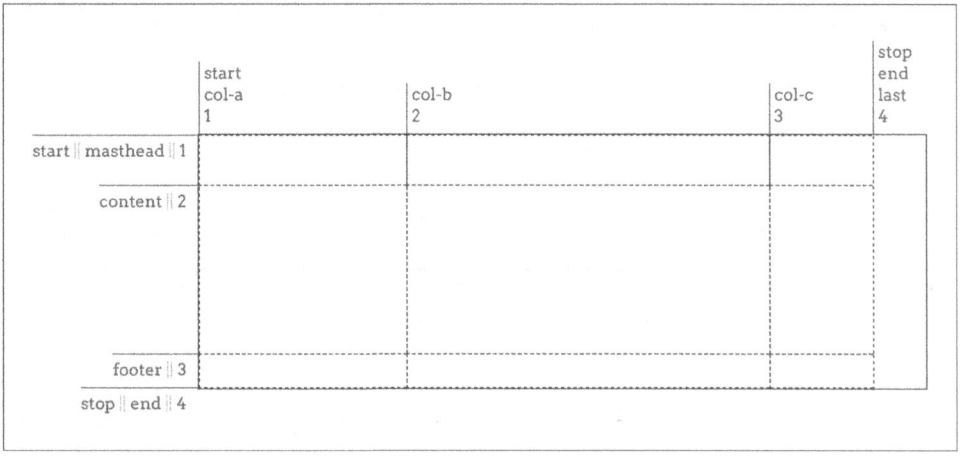


Рис. 13.11. Подгонка высоты ряда к размеру контейнера

Следуя точно таким же принципам, функцию `minmax()` можно использовать для ограничения ширины колонки `col-b`, включенной в этот же grid-контейнер. Учтите: если аргумент `b` (максимальное значение) функции меньше аргумента `a` (минимальное значение), то колонка или ряд будет иметь фиксированный размер, определяемый параметром `a`, а параметр `b` во внимание приниматься не будет. Таким образом, функция `minmax(100px, 2em)` обеспечивает фиксацию ширины колонки в значении 100 пикселей при любых шрифтах, размер которых не превышает 50 пикселей.

Чтобы избежать неопределенности при использовании функции `minmax()` в объявлениях grid-контейнера, замените ее функцией `calc()`, подставив вместо соответствующего размерного значения высоту ряда (ширины колонки).

```
grid-template-rows:
   [start masthead] 3em [content] calc(100%-5em) [footer]
      2em [stop end];
```

Согласно последнему правилу, высота ряда `content` равна вертикальному размеру grid-контейнера за вычетом суммы высот рядов `masthead` и `footer`.

Такой подход характеризуется весьма ограниченной областью применения, поскольку с изменением высоты рядов `masthead` и `footer` конечная формула также будет подвергаться корректировке. В ситуациях, когда адаптивность дизайна обеспечивается гибкостью изменения ширины (высоты) сразу нескольких колонок (рядов), подбор правильных аргументов становится очень сложной (если вообще выполнимой) задачей. О том, как лучше всего поступать в подобных случаях, рассказывается в следующем разделе.

Адаптивные колонки и ряды

До последнего момента мы рассматривали примеры, в которых колонки и ряды имеют строго фиксированный размер, определяемый в единицах измерения длины или зависящий исключительно от размеров grid-контейнера. С другой стороны, размер *адаптивных*, или *гибких*, колонок и рядов зависит от степени заполнения их содержимым и наличия в grid-контейнере свободного пространства, не занятого колонками и рядами, которые имеют фиксированный размер.

Дольная единица измерения

Единица измерения `fr` позволяет указывать размер колонки (ряда) в виде дольной части от общего размера grid-контейнера. С ее помощью чрезвычайно удобно перераспределять свободное пространство контейнера между отдельными или сразу всеми колонками (рядами) сетки.

В простейшем случае сетка состоит из колонок (рядов) одинаковой ширины (высоты). Следующее объявление обеспечивает создание сетки, состоящей из четырех одинаковых колонок:

```
grid-template-columns: 1fr 1fr 1fr 1fr;
```

При необходимости его можно представить в более привычных единицах измерения:

```
grid-template-columns: 25% 25% 25% 25%;
```

Результат выполнения каждой из строк кода приведен на рис. 13.12.

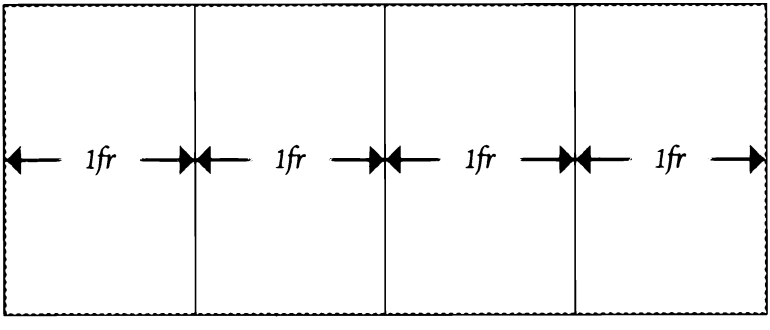


Рис. 13.12. Разделение grid-контейнера на четыре одинаковые колонки

Теперь представьте, что в сетку нужно добавить еще одну колонку, перераспределив пространство контейнера так, чтобы одинаковый размер теперь имели все пять колонок. При представлении ширины колонок процентными величинами объявление придется переписать, передав свойству `grid-template-columns` пять экземпляров значения `20%`. Используя единицы измерения `fr`, в объявление достаточно добавить еще одно значение `1fr`.

```
grid-template-columns: 1fr 1fr 1fr 1fr 1fr;
```

Для определения дольной части полную ширину grid-контейнера нужно разделить на их общее количество. Ширина каждой колонки определяется в результате умножения числового значения, представляющего дольную часть, на их количество, которое указано в значении, передаваемом свойству `grid-template-columns`.

В примере с четырьмя одинаковыми колонками шириной `1fr` полный горизонтальный размер grid-контейнера нужно разделить на 4. Иными словами, каждая из колонок занимает четвертую часть ширины контейнера. При добавлении в сетку еще одной колонки с такой же дольной частью (`1fr`) все пространство контейнера нужно будет разделить на пять одинаковых частей. В результате каждая из колонок будет занимать только пятую часть его ширины.

Размер колонки или ряда, выраженный в единицах измерения `fr`, совсем не обязательно представляется целым числом. Рассмотрим пример, в рамках которого на сетку добавляются три колонки, ширина средней из которых вдвое больше остальных двух. Стилевое правило, отвечающее за выполнение такого форматирования, имеет следующий вид:

```
grid-template-columns: 1fr 2fr 1fr;
```

Приняв общую ширину контейнера за единицу, можно легко подсчитать, что значение `1fr` соответствует его четвертой части, или 25%. Таким образом, ширина первой и третьей колонки может представляться числом 0,25, а ширина второй, средней, колонки — значением 0,5 (вдвое больше значения 0,25).

Из вышесказанного следует, что дольные части могут выражаться действительными числами. В частности, при верстке по сетке карточка с рецептом яблочного пирога может представляться следующими тремя колонками:

```
grid-template-columns: 1fr 3.14159fr 1fr;
```

Попробуйте самостоятельно определить вычисляемую ширину каждой из колонок (В этом нет ничего сложного: просто разделите полную ширину документа на значение 1 + 3,14159 + 1.)

Разделение пространства grid-контейнера на дольные части — очень удобный способ компоновки сетки, выходящий за рамки интуитивно понятной концепции верстки, в которой размерные величины определяются процентными значениями. Преимущество единиц измерения `fr` становится очевидным при адаптивной верстке многоколоночных документов, в которых часть колонок имеет фиксированный размер. Рассмотрим такой пример (рис. 13.13):

```
grid-template-columns: 15em 1fr 10%;
```

В данном макете первая и последняя колонки имеют фиксированные размеры, не изменяющиеся при растягивании или сжатии окна просмотра. Следовательно, адаптивность документа обеспечивается только подстройкой ширины средней колонки, занимающей все остальное пространство grid-контейнера. В частности, в контейнере с горизонтальным размером 1000 пикселей, свойство `font-size` которого имеет значение по умолчанию `16px`, первая колонка будет иметь ширину 240 пикселей, а третья колонка — всего 100 пикселей. Следовательно, колонки фиксированного размера имеют суммарную ширину 340 пикселей, а все остальное пространство

контейнера — ширину 660 пикселей. Поскольку в дольных частях представляется ширина единственной, второй, колонки, именно под нее выделяется все доступное (нефиксированное) пространство. Но так как оно представлено всего одной дольной частью, значение `1fr` будет соответствовать ширине 660 пикселей. При увеличении горизонтального размера grid-контейнера, например до значения 1400 пикселей, ширина третьей колонки составит 14 пикселей, а вторая колонка расширится до 1020 пикселей.

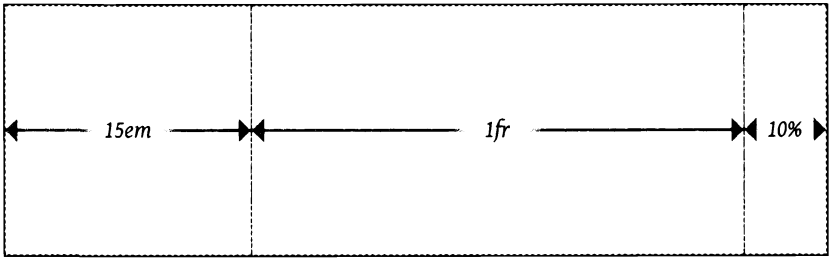


Рис. 13.13. Выделение под вторую колонку всего доступного пространства контейнера

Как видите, один и тот же документ может включать колонки с фиксированной и адаптивно изменяемой шириной. При необходимости адаптивную область grid-контейнера можно представить не одной, а несколькими колонками с разными дольными частями, как показано в следующем примере:

```
width: 100em; grid-template-columns: 15em 4.5fr 3fr 10%;
```

В данном случае ширина колонок определяется так, как показано на рис. 13.14.

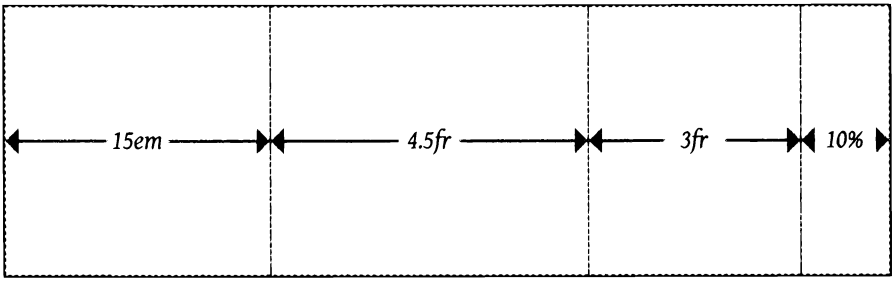


Рис. 13.14. Колонки с адаптивно изменяемой шириной

В порядке следования слева направо колонки имеют такую ширину: `15em`, `45em`, `30em` и `10em`. Размер первой колонки строго фиксированный: `15em`. Ширина последней колонки вычисляется как 10% от `100em`, т.е. она равна `10em`. В результате область адаптивного пространства, распределяемого между остальными двумя колонками, имеет ширину `75em`, которую нужно разделить на 7,5 дольных частей. Следовательно, ширину большей из двух колонок можно вычислить как `75em / 7.5fr × 4.5fr`, что равно `45em`. Подобным образом вычисляется ширина более узкой колонки: `75em / 7.5fr × 3fr = 30em`.

Конечно, в приведенном выше примере я умышленно использовал значения, делящиеся одно на другое без остатка. Легко заметить, что ширина адаптивного пространства контейнера, общее количество дольных частей и дольные части каждой из колонок кратны одному и тому же числовому значению. Такое упрощение позволяет избежать неоднозначности в трактовке полученного результата. Чтобы понять, насколько сложнее будут вычисления при позиционировании линий сетки с помощью произвольных числовых значений, попробуйте самостоятельно определить размер колонок при ширине grid-контейнера, равной `92.5em`, или `1234px`.

Как и прежде, минимальный и максимальный размеры полос сетки можно устанавливать с помощью функции `minmax()`. В продолжение предыдущего примера ограничим минимальную ширину третьей колонки значением `5em`:

```
grid-template-columns: 15em 4.5fr minmax(5em,3fr) 10%;
```

Согласно такому объявлению третья колонка будет оставаться адаптивной при уменьшении ширины до значения `5em`. Дальнейшее сужение колонки невозможно, а потому размер контейнера будет уменьшаться за счет одной только второй колонки. Остальные три колонки сетки будут иметь фиксированный размер: `15em`, `5em` и `10%`. Точный расчет показывает, что способность к адаптивному изменению размера третья колонка получает при расширении grid-контейнера до ширины, большей `30.5556em`. При меньшей ширине контейнера она имеет строго фиксированный размер.

Можно попробовать обыграть обратную ситуацию — сохранить способность к растягиванию колонки только до строго заданного размера, объявив для нее минимальное значение, выраженное в дольных единицах измерения (`fr`). К сожалению, это не приведет к ожидаемому результату, поскольку спецификация CSS не допускает использования таких значений в качестве первого аргумента функции `minmax()`. Минимальный размер, выражаемый в единицах `fr`, пользовательскими агентами не обрабатывается.

Еще один крайний случай возникает при представлении минимального размера нулевым значением:

```
grid-template-columns: 15em 1fr minmax(0,500px) 10%;
```

На рис. 13.15 показана сетка минимально возможного размера, при котором третья колонка сохраняет ширину 500 пикселей. При дальнейшем сужении grid-контейнера ширина колонки, для которой объявлена функция `minmax()`, также начнет уменьшаться вплоть до нулевой величины. В случае расширения сетки до значения, большего показанного на рис. 13.15, третья колонка зафиксируется в размере 500 пикселей, а увеличение grid-контейнера будет осуществляться за счет второй колонки, ширина которой указана в единицах `fr`.

При внимательном рассмотрении примера можно заметить, что подпись `1fr` располагается у линии разграничения колонок `15em` и `minmax(0,500px)`. В общем случае она должна находиться у левого края второй колонки, имеющей нулевую ширину при заданном размере контейнера. Исходя из этого, вторая линия сетки совмещается с ее третьей линией, положение которой определяется функцией `minmax(0,500px)`. Такая конфигурация линий сетки справедлива только при нулевой ширине второй колонки.

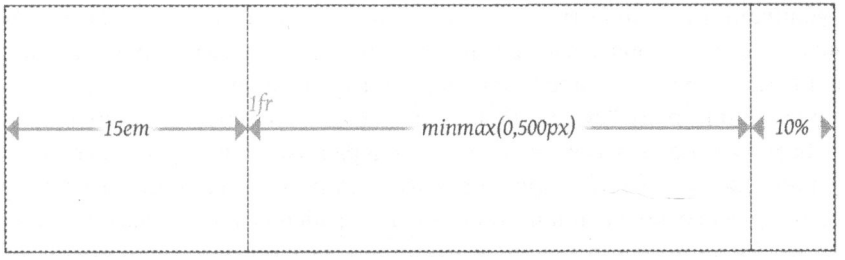


Рис. 13.15. Установка размера колонки с помощью функции `minmax()`

Если минимальное значение, передаваемое функции `minmax(0,500px)`, больше ее второго аргумента, то третья колонка получает минимально возможный размер. Таким образом, функция `minmax(500px,200px)` будет представляться вычисляемым значением `500px`. Эта особенность становится чрезвычайно полезной при передаче функции процентных значений и значений, выраженных в единицах `fr`. В частности, функция `minmax(10%,1fr)` разрешает сжимать колонку до тех пор, пока ее ширина не достигнет значения 10% от горизонтального размера grid-контейнера. Начиная с этого момента сжатие колонки полностью прекращается, а ее ширина определяется строго заданным значением `10%`.

Не менее эффективно дольные значения применяются для ограничения размера рядов, хотя необходимость в таком действии возникает очень редко. Чаще всего под ограничение высоты подпадают ряды, представляющие шапку и подвал страницы, — остальные ее области должны сжиматься вместе с окном просмотра. Такой способ форматирования документа обеспечивается следующим объявлением:

```
grid-template-rows: 3em minmax(5em,1fr) 2em;
```

Этот способ верстки востребован только в случаях, когда высоту ряда можно представить как дольную часть от вертикального размера grid-контейнера. Она оказывается несостоятельной в ситуациях, когда высота ряда ограничивается размером ее содержимого. Для решения подобных задач применяются совершенно иные методики, о которых рассказано в следующем разделе.

Размер, обусловливаемый содержимым

Одно дело — определять ширину колонок и высоту рядов в дольных единицах измерения, и совершенно иное — устанавливать ее в соответствии с размером содержимого. Но каким образом добиться правильного выравнивания элементов, позиционирование которых зависит от степени заполнения данными? Для эффективного решения такой задачи нам понадобятся ключевые слова `min-content` и `max-content`.

У этих ключевых слов простое определение и необычайно запутанная область применения. Значение `max-content` указывает колонке или ряду "расширяться до максимально возможного размера, обеспечивающего размещение всего объявленного содержимого". Применимо к большим текстовым фрагментам (например, отдельным публикациям в блоге) такая формулировка предполагает выделение для содержимого всей доступной ширины или высоты страницы, хотя в большинстве случаев все сводится к расширению элемента до размера, обеспечивающего размещение

текста публикации в одной строке (без переноса на другие строки). Зачастую ключевое слово `max-content` обязывает расширять колонку до размера регулярного абзаца.

Значение `min-content`, в противоположность ключевому слову `max-content`, указывает колонке или ряду "сужаться до минимально возможного размера, обеспечивающего размещение всего объявленного содержимого". В случае текстового фрагмента колонка сжимается до ширины самого длинного слова (или высоты строчного элемента для ряда сетки, если в него включены графические элементы или элементы управления). Такая настройка чаще всего приводит к образованию чрезвычайно узких и высоких колонок с большим количеством строк.

К отличительной особенности описанных выше ключевых слов относится их применимость сразу ко всем grid-элементам одной колонки или ряда сетки. Например, объявление значения `max-content` для колонки приводит к расширению всех ее элементов до размера элемента с наибольшим содержимым. Проще всего проиллюстрировать назначение этого ключевого слова на примере сетки, содержащей изображения (всего 12) разных форм и размеров. Их верстка осуществляется с помощью следующего стилевого правила, результат выполнения которого показан на рис. 13.16.

```
#gallery {display: grid;
   grid-template-columns: max-content max-content max-content
      max-content;
   grid-template-rows: max-content max-content max-content;}
```

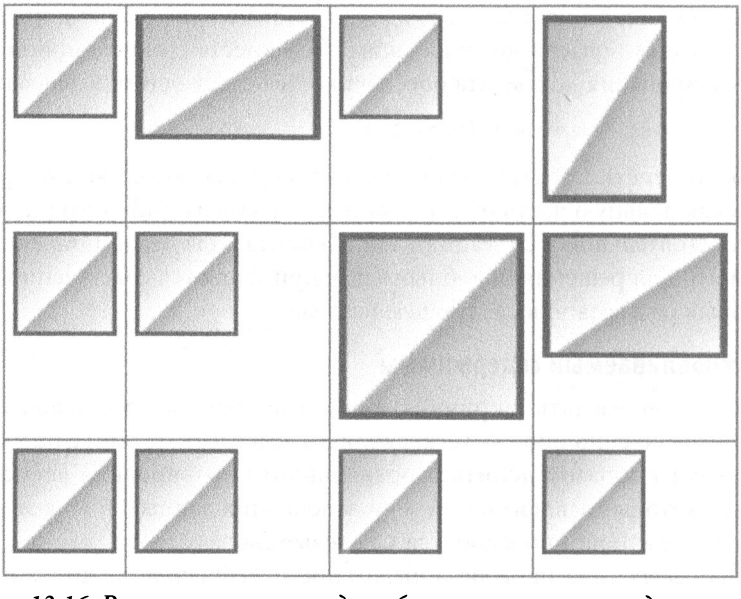


Рис. 13.16. Размер колонок и рядов обусловливается их содержимым

Легко заметить, что ширина каждой колонки определяется горизонтальным размером самого широкого ее изображения. Таким образом, при верстке страницы, содержащей галерею портретов, колонки будут уже, чем при наполнении ее коллекцией панорамных изображений. Такой же принцип позиционирования применяется и

при заполнении содержимым рядов. Высота каждого ряда определяется вертикальным размером самого высокого из изображений. Наименьшую высоту будет иметь ряд с самыми низкими изображениями.

К преимуществам такого способа объявления колонок и рядов сетки можно отнести нечувствительность его к типу содержимого grid-элементов. Предположим, что к изображениям, выравниваемым по сетке, добавляются подписи. Размеры колонок и рядов grid-контейнера будут автоматически подкорректированы так, чтобы полностью вместить как графические изображения, так и текстовое содержимое (рис. 13.17).

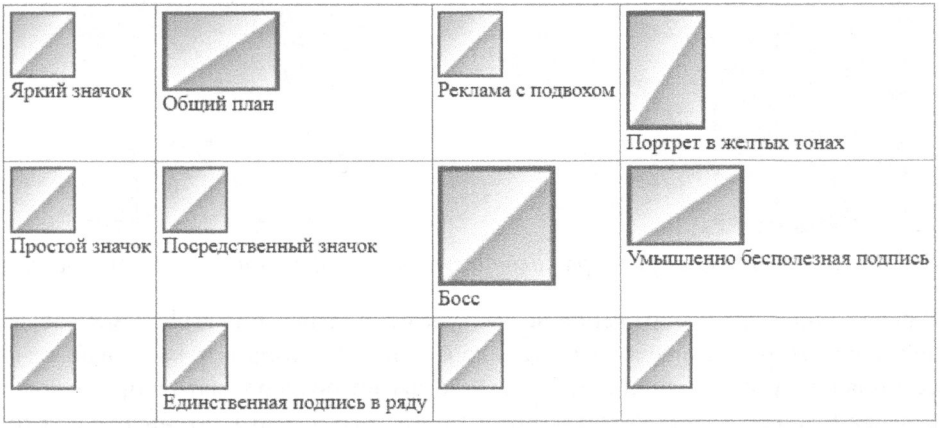


Рис. 13.17. Изменение размеров колонок и рядов при включении в них разнотипного содержимого

С точки зрения графического дизайна такая страница выглядит совершенно неподобающе — отсутствие выравнивания у изображений и разная длина подписей делают ее совершенно непривлекательной. Но ничего не поделаешь: именно так работает настройка `max-content` применительно к колонкам сетки. Она буквально заставляет колонки "расширяться до максимально возможного размера, обеспечивающего размещение всего их содержимого".

Важно понимать, что такой же способ заполнения содержимым поддерживается даже в отсутствие в grid-контейнере доступного места. Это означает, что показанные на рис. 13.17 изображения и подписи не изменят своего форматирования при существенном уменьшении ширины контейнера, например, вызванном объявлением `width: 250px`. Именно поэтому ключевое слово `max-content` обычно не используется в виде отдельного значения, а передается в качестве аргумента функции `minmax()`. Рассмотрим, как будут отличаться страницы, содержащие одну и ту же галерею изображений, сверстанных по сетке, ширина колонок которой определяется с помощью ключевого слова `max-content`, определенного в явном виде и заключенного в функцию `minmax()`. В обоих случаях контейнер имеет размер, обозначенный оранжевым фоном (рис. 13.18).

```
#g1 {display: grid;
   grid-template-columns: max-content max-content max-content
```

```
      max-content;
   }
#g2 {display: grid;
   grid-template-columns: minmax(0,max-content)
      minmax(0,max-content) minmax(0,max-content)
      minmax(0,max-content);
   }
```

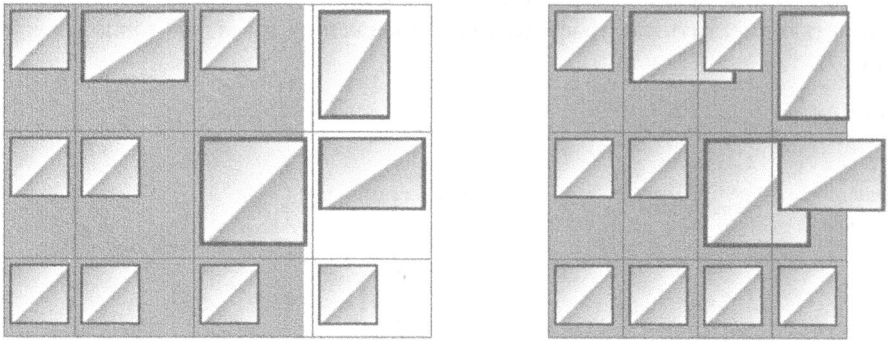


Рис. 13.18. Ограничение размера колонок с помощью функции `minmax()` *и без нее*

В первом примере изображения не выступают за края своих grid-элементов, далеко не все из которых помещаются в grid-контейнер. Во втором примере наблюдается совсем иная ситуация: функция `minmax()` указывает поддерживать ширину колонок в пределах от `0` до `max-content` так, чтобы поместить в grid-контейнер все исходно объявленные элементы. В качестве более приемлемого варианта рассмотрите возможность замены первого аргумента функции `minmax()` значением `min-content`.

Выход содержимого за края grid-элементов во втором примере становится возможным только благодаря включению функции `minmax(0,max-content)` в код объявления колонок. Так как grid-контейнер не в состоянии вместить колонки в полном размере, их ширина устанавливается в такое максимально возможное значение, при котором они не выступают за края контейнера. Вследствие этого содержимое отдельных grid-элементов будет выступать за края колонок, частично перекрывая содержимое соседних элементов. Такое поведение свойственно любым сеткам, содержимое которых не помещается в отведенное для них пространство grid-контейнера.

Неподдельный интерес вызывают сетки, в которых ключевое слово `min-content` применяется для определения размеров как колонок, так и рядов. В общем случае они получают такое же форматирование, как и сетки, в которых значение `min-content` объявлено только для колонок, а высота рядов специальным образом не оговаривается. А все потому, что спецификация однозначно указывает пользовательским агентам сначала установить ширину колонок и только после этого переходить к позиционированию горизонтальных линий сетки.

Положение колонок и рядов сетки может также определяться с помощью универсального ключевого слова `auto`. При подстановке его в объявление сетки в качестве минимального значения оно будет рассматриваться как ключевое слово `min-width` или `min-height`. Если же использовать его в качестве аргумента, определяющего

максимальный размер колонок или рядов, то оно будет обрабатываться так же, как и значение `max-content`. Учтите, что ключевое слово `auto` можно применять отдельно от функции `minmax()`. Какой размер оно определяет в каждом конкретном случае — минимальный или максимальный, — зависит от плотности заполнения grid-контейнера колонками и рядами, а также от размера содержимого grid-элементов. Как и во многих других аспектах, способ обработки ключевого слова `auto` пользовательским агентом в основном определяется его внутренними настройками.

У значения `auto` есть интересная особенность, требующая отдельного рассмотрения. При автоматической настройке размера колонок или рядов выравнивание grid-элементов можно определять с помощью свойств `align-content` и `justify-content`, о чем будет рассказано в разделе "Выравнивание grid-элементов". Поскольку во всех остальных случаях их использовать запрещено, можно смело утверждать, что ключевое слово `auto` открывает новые возможности по позиционированию grid-элементов.

Размещение содержимого в пределах колонки или ряда

Наряду с ключевыми словами `min-content` и `max-content` ограничение размеров колонок и рядов сетки можно осуществлять с помощью функции `fit-content()`, обладающей более широкими возможностями по вычислению их конечных размеров. Несмотря на заметные сложности в точной настройке, она стоит самого пристального внимания.

Функция `fit-content()` получает в качестве аргумента значение *<length>* или *<percentage>*, как показано в приведенном ниже CSS-коде.

```
#grid {display: grid; grid-template-columns: 1fr
   fit-content(150px) 2fr;}
#grid2 {display: grid; grid-template-columns: 2fr
   fit-content(50%) 1fr;}
```

Перед тем как приступить к детальному изучению приведенных выше примеров, рассмотрим, как обрабатывается такая псевдоформула, указанная в спецификации:

```
fit-content(argument) => min(max-content, max(min-content, argument))
```

В буквальном понимании она определяет большее из значений `min-content` и `argument` и сравнивает полученный результат со значением `max-content`, возвращая меньшее из них. Такой подход к обработке функцией исходных данных несколько обескураживает своей запутанностью. По крайней мере, мне понадобилось 17 попыток, чтобы понять, как же она все-таки работает.

Переосмыслив вышесказанное, функцию `fit-content()` можно считать эквивалентом функции `minmax(min-content,max-content)` с единственным аргументом, который, как и значение `max-width` или `max-height`, определяет верхний предел ширины колонки или высоты ряда. Рассмотрим такой пример:

```
#example {display: grid; grid-template-columns: fit-content(50ch);}
```

В качестве аргумента функции `fit-content()` использовано значение `50ch`, представляющее ширину в 50 символов. Следовательно, приведенный выше код объявляет о создании сетки, состоящей из единственной колонки, содержимое которой должно помещаться в область указанной ширины.

Сначала представим, что содержимое колонки имеет несколько меньшую ширину, например 29 символов (`29ch` для текста, отображенного моноширинным шрифтом). В таком случае ключевое слово `max-content` будет представлять значение `29ch`, и именно до такого размера расширится единственная колонка сетки. (Значение `29ch` оказывается меньшим, чем большее из значений `50ch` и `min-content`.)

Теперь рассмотрим, каким образом в колонке будет размещаться тестовый фрагмент общей длиной 256 символов (`256ch`). В этом случае ключевое слово `max-content` будет представлено значением `256ch`. Оно намного больше аргумента `50ch`, поэтому ширина колонки будет определяться большим из значений `min-content` и `50ch`. По вполне очевидным причинам она будет составлять `50ch`.

В качестве следующего примера рассмотрим CSS-код, результат выполнения которого показан на рис. 13.19.

```
#thefollowing {
   display: grid;
   grid-template-columns:
      fit-content(50ch) fit-content(50ch) fit-content(50ch);
   font-family: monospace;}
```

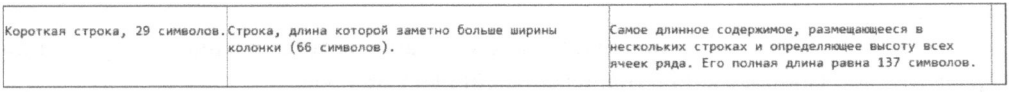


Рис. 13.19. Колонки, размер которых определяется функцией `fit-content()`

Как видите, первая колонка заметно *уже* остальных двух. Ее ширина определяется содержимым длиной `29ch`. Ширина двух других колонок составляет `50ch`, поскольку их содержимое имеет заметно *большую* длину и переносится на последующие строки.

Посмотрим, что произойдет, если во вторую колонку вставить изображение строго заданного размера. Пусть ширина изображения составляет 500 пикселей — больше опорного значения `50ch` при заданном размере шрифта. Вначале нужно определить большее из значений `min-content` и `50ch`. Согласно исходному предположению, *большим* оказывается значение `min-content`, представляющее ширину изображения, равную 500 пикселям. На следующем этапе нужно определить *меньшее* из значений `500px` и `max-content`. При представлении текста второй колонки одной строкой ее длина будет несколько больше `500px`, поэтому функция `fit-content()` будет возвращать значение `500px`. Результат сужения второй колонки до ширины изображения показан на рис. 13.20.

Сравнивая рис. 13.19 и 13.20, легко заметить отличия в ширине второй колонки и местах разрыва текстового содержимого, переносимого на следующую строку. Кроме того, изменение ширины второй колонки влечет за собой перераспределение текста между строками третьей колонки.

Рис. 13.20. Подгонка ширины колонки под размер содержимого

Причина такого поведения вполне очевидна: ширина третьей колонки всецело зависит от размеров первых двух колонок. В последнем случае она несколько меньше значения `50ch`. Объявление `fit-content(50ch)` все еще остается в силе, но реализуется оно только при наличии в сетке достаточного количества свободного места. Важно помнить, что значение `50ch` указывает максимально возможную, но не строго фиксированную ширину колонки.

Последняя особенность делает функцию `fit-content()` предпочтительнее функции `minmax()` применительно к настройке сетки. Она позволяет колонкам и рядам сжиматься до минимально возможного размера в отсутствие содержимого и расширяться по мере заполнения данными до ширины или высоты, обозначенной единственным аргументом.

В предыдущих разделах нам уже приходилось иметь дело с сетками, состоящими из нескольких колонок абсолютно одинаковой ширины. Но как быть, если требуется создать сетку, состоящую из большого количества таких колонок? Неужели не существует способа объявления их размера с помощью одной команды? Вообще-то, такой способ есть, и он описан в следующем разделе.

Повторяющиеся линии сетки

В ситуациях, когда сетка состоит из большого количества колонок или рядов одинакового размера, в ее объявлении можно избежать перечисления всех линий разметки grid-контейнера. Их создание можно доверить специальной функции: `repeat()`.

Рассмотрим, как с ее помощью образуется сетка, состоящая из 10 колонок шириной `5em` каждая.

```
#grid {display: grid;
   grid-template-columns: repeat(10, 5em);}
```

Вот и весь код! Он позволяет создать сетку, состоящую из 10 колонок общей шириной `50em` и избежать многократного ввода значения `5em`.

Функцию `repeat()` можно использовать при объявлении любых значений, принимающих участие в настройке размера колонок и рядов сетки, начиная со значений, выраженных в единицах `fr`, и заканчивая ключевыми словами `min-content`, `max-content` и `auto`. Рассмотрим, как объявляется сетка, которая представлена структурой, состоящей их трех троекратно повторяющихся колонок (рис 13.21): одной узкой (`2em`) и двух широких (`1fr`).

```
#grid {display: grid;
   grid-template-columns: repeat(3, 2em 1fr 1fr);}
```

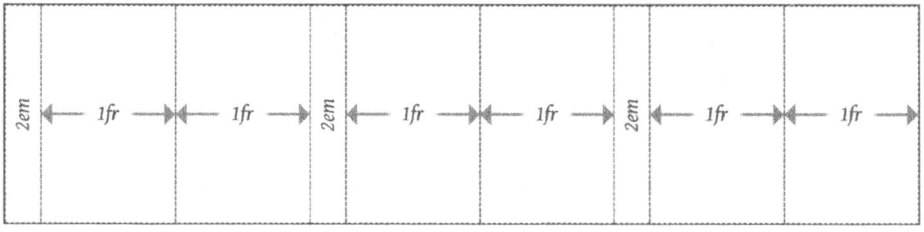


Рис. 13.21. Сетка, образованная повторением колонок

Обратите внимание на то, что, согласно заданному в функции `repeat()` шаблону, сетка начинается с колонки шириной `2em`, а заканчивается колонкой шириной `1fr`. Чтобы сделать сетку абсолютно симметричной, в ее конец (но не в шаблон функции `repeat()`) нужно добавить колонку шириной `2em`.

```
#grid {display: grid;
   grid-template-columns: repeat(3, 2em 1fr 1fr) 2em;}
```

Приведенный выше код позволяет повторить начальную колонку шириной `2em` в конце сетки, что становится возможным благодаря объявлению ее отдельно от остальных колонок, задаваемых функцией `repeat()`. Единственное, что непозволительно этой функции, так это вызываться внутри самой себя.

Кроме упомянутого выше ограничения, в качестве аргументов функции `repeat()` допускается использовать любые поддерживаемые в CSS типы данных. Ниже приведен один из возможных вариантов объявления сетки.

```
#grid {display: grid;
   grid-template-columns: repeat(4, 10px [col-start] 250px
      [col-end]) 10px;}
```

С помощью этого кода в документе создается сетка, которая образована четырехкратным повторением шаблона, представленного 10-пиксельной колонкой, именованной линией, 250-пиксельной колонкой и еще одной именованной линией. В конец сетки добавляется еще одна 10-пиксельная колонка, идентичная самой первой колонке сетки. Таким образом, в сетке будут сразу четыре линии с именем `col-start` и столько же линий с именем `col-end` (рис. 13.22), что вполне допускается спецификацией: в качестве имен линий можно использовать любые неуникальные названия.

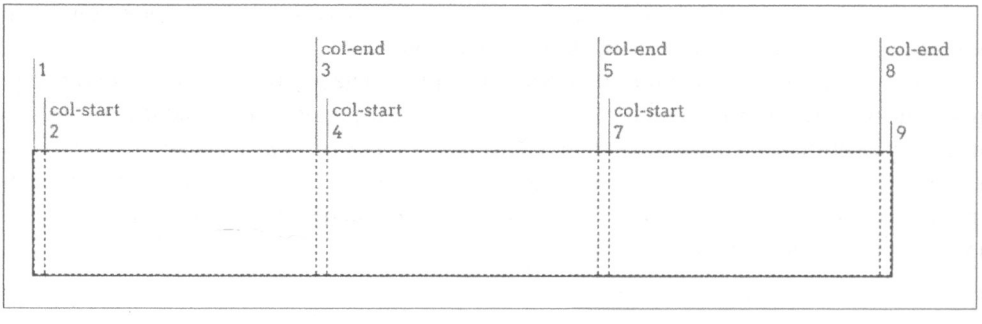


Рис. 13.22. Повторение именованных линий

Помните: если в результате повторения именованные линии совмещаются, то они образуют единую линию с двойным названием. Иными словами, следующие два объявления позволяют создавать полностью идентичные сетки.

```
grid-template-rows: repeat(3, [top] 5em [bottom]);
grid-template-rows: [top] 5em [bottom top] 5em [top bottom]
   5em [bottom];
```

Дважды подумайте перед тем, как отказываться от применения в одной сетке линий с одинаковыми именами. В отдельных случаях они могут оказаться незаменимыми. Детально об этом рассказано в разделе "Наполнение колонок и рядов".

Автоматическое повторение сетки

В CSS предусмотрена возможность создания сетки путем повторения заранее объявленного простого шаблона, избегая применения в функции `repeat()` сложных для понимания конструкций. Несмотря на очевидную простоту использования, данный метод зарекомендовал себя как достаточно эффективное средство верстки документов по сетке.

Предположим, что шаблон из предыдущего примера нужно повторить до заполнения сеткой всего пространства grid-контейнера.

```
grid-template-rows: repeat(auto-fill, [top] 5em [bottom]);
```

Горизонтальные линии заполняют grid-контейнер, размещаясь друг от друга на расстоянии `5em`. Эту же сетку, но для grid-контейнера высотой `11em`, можно объявить с помощью следующего кода:

```
grid-template-rows: [top] 5em [bottom top] 5em [bottom];
```

Если высота контейнера составляет от `15em` до `20em`, то объявление нужно представить в таком виде.

```
grid-template-rows: [top] 5em [bottom top] 5em [top bottom]
   5em [bottom];
```

Все три варианта заполнения grid-контейнера горизонтальными линиями показаны на рис. 13.23.

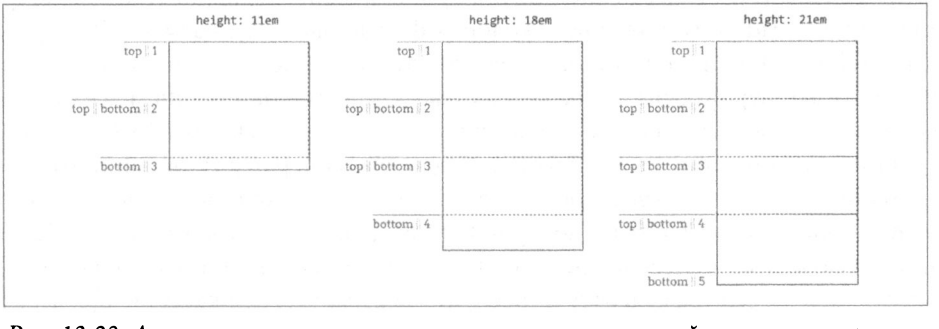


Рис. 13.23. Автоматическое повторение сетки, состоящей только из горизонтальных линий, в grid-контейнерах разной высоты

Автоматическое повторение возможно только при использовании шаблонов, включающих объявления не более двух необязательных именованных линий, между которыми указывается единственное вычисляемое значение, представляющее размер колонки или ряда. Согласно такому требованию шаблон `[top] 5em [bottom]`, используемый в предыдущем примере, содержит максимально возможное количество компонентов. При необходимости именованные линии можно не объявлять, представив шаблон одним только вычисляемым значением. Повторению не подлежат сразу несколько колонок или рядов фиксированного или автоматически подстраиваемого размера. (Чтобы понять причину такого ограничения, попробуйте представить, сколько раз будет повторяться ряд шириной `1fr` до заполнения всего контейнера.)

Не стоит полагаться на средство автоповторения при создании междуколоночных промежутков: эта задача решается с помощью специальных стилевых инструментов и свойств, детально рассмотренных в разделе "Интервалы сетки".

Кроме того, в одной сетке допускается применять всего один шаблон автоматического повторения линий. Следовательно, приведенное ниже объявление считается нелегитимным.

```
grid-template-columns: repeat(auto-fill, 4em) repeat(auto-fill, 100px);
```

Тем не менее в одном объявлении разрешается комбинировать функции регулярного и автоматического повторения линий. Например, в начало сетки можно поместить три колонки абсолютно одинаковой ширины, а оставшееся пространство grid-контейнера заполнить колонками намного меньшего размера автоматически. Такая задача, в частности, решается с помощью следующего объявления:

```
grid-template-columns: repeat(3, 20em) repeat(auto-fill, 2em);
```

Для изменения порядка построения сетки на противоположный используется такой код:

```
grid-template-columns: repeat(auto-fill, 2em) repeat(3, 20em);
```

Выполнение последнего сценария становится возможным исключительно благодаря тому, что автоматическое повторение узких колонок выполняется только после размещения в grid-контейнере всех колонок фиксированного размера. В результате grid-контейнер заполняется сеткой, в начало которой помещается несколько колонок шириной `2em`, генерируемых функцией автоматического повторения, а все остальное пространство отводится под три колонки шириной `20em`. На рис. 13.24 приведены два примера такой сетки, сгенерированной для контейнеров разной ширины.

Команда `auto-fill` создает не менее одной копии исходной колонки или ряда, даже если она не помещается в доступное пространство grid-контейнера. Наряду с этим автоматически повторяемые колонки или ряды заполняют все доступное пространство контейнера даже в отсутствие в них содержимого. В качестве примера рассмотрим ситуацию автоматического заполнения grid-контейнера пятью колонками,

только три из которых вмещают некое содержимое. Оставшиеся две колонки, несмотря ни на что, будут размещены на сетке, резервируя место под несуществующее содержимое.

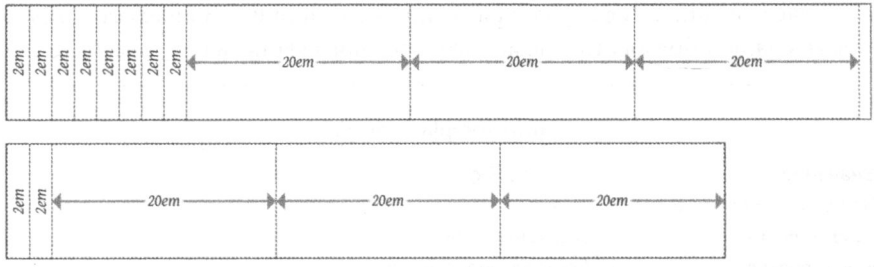


Рис. 13.24. Колонки, сгенерированные функцией автоматического повторения, перед колонками фиксированного размера

В противоположность `auto-fill`, команда `auto-fit` обязывает к исключению из сетки grid-элементов, лишенных содержимого. Во всем остальном эти команды полностью идентичны. Рассмотрим такой пример:

```
grid-template-columns: repeat(auto-fit, 20em);
```

Даже в случае образования grid-контейнера достаточного размера (с шириной, большей `100em`) сетка будет содержать всего три из пяти объявляемых колонок. Остальные две колонки исключаются из сетки по причине отсутствия в них grid-элементов. Освободившееся пространство распределяется согласно значениям свойств `align-content` и `justify-content`, объявленным для grid-контейнера в соответствии с требованиями, описанными в разделе "Выравнивание grid-элементов". На рис. 13.25 представлен результат автоматического заполнения и повторения одних и тех же колонок в абсолютно идентичных grid-контейнерах. Числа в цветных квадратах указывают номер колонки, добавленной в grid-контейнер.

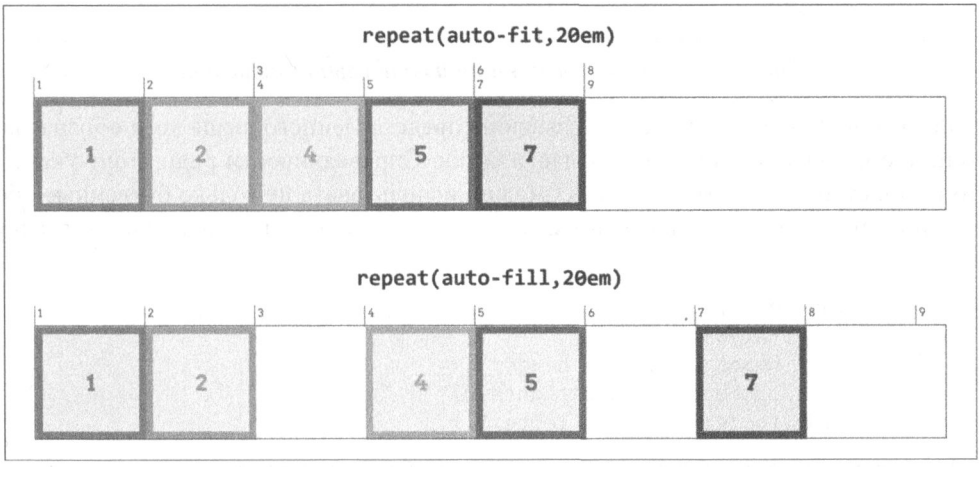


Рис. 13.25. Автоматическое заполнение и повторение колонок

Области сетки

В отдельных случаях сетку проще нарисовать, чем представить в виде набора колонок и рядов. Это не только интересно, но и очень полезно: в процессе рисования намного легче запоминается структура сетки. В том или ином виде сетку произвольной формы можно образовать с помощью стилевого свойства `grid-template-areas`.

grid-template-areas

Значение	`none` | *<string>*
Начальное значение	`none`
Применяется	Grid-контейнеры
Вычисляется	Согласно определению
Наследуется	Нет
Анимируется	Нет

Чтобы понять, как работает это свойство, можно не утруждать себя детальным изучением его синтаксиса, а сразу переходить к рассмотрению примеров. Результат выполнения первого из них показан на рис. 13.26.

```
#grid {display: grid;
   grid-template-areas:
      "h h h h"
      "l c c r"
      "l f f f";}
```

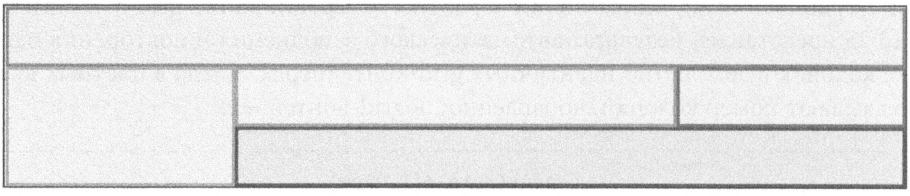


Рис. 13.26. Сетка, состоящая из отдельных областей

Все верно: буквы в двухмерном шаблоне представленного выше кода обозначают ячейки, образующие отдельные области сетки. Справедливости ради стоит указать, что в свойстве `grid-template-areas` можно использовать не только буквенные обозначения. В частности, приведенный ниже пример полностью идентичен предыдущему.

```
#grid {display: grid;
   grid-template-areas:
      "header   header  header  header"
      "leftside content content rightside"
      "leftside footer  footer  footer";}
```

Несмотря на замену буквенных обозначений более информативными подписями, данный шаблон представляет все ту же сетку.

Как известно, в CSS-коде допускается вводить подряд несколько пробелов или других разделителей — обработчиком они рассматриваются как один пробел. Эта особенность позволяет применять пробелы для визуального выравнивания элементов в шаблоне сетки (как в примере выше) без нарушения ее структуры. Для этих же целей вместо пробелов можно применять символы табуляции или любые другие общепринятые разделители. Заметьте, элементы достаточно разграничивать всего одним символом — совсем не обязательно упорядочивать их в идеально выровненные столбцы. Более того, строки элементов можно не разграничивать символами разрыва строки. Следующая строка кода выполняет такие же операции, как и два предыдущих объявления:

```
grid-template-areas: "h h h h" "l c c r" "l f f f";
```

При объединении элементов шаблона в одну текстовую строку их назначение изменится. Каждая текстовая строка шаблона (заключенная в кавычки) представляет отдельный ряд сетки. Следовательно, во всех трех приведенных выше примерах создается сетка, состоящая из трех рядов. Для преобразования ее в сетку, включающую всего один длинный ряд, объявление нужно представить так.

```
grid-template-areas:
   "h h h h
   l c c r
   l f f f";
```

В результате выполнения такого кода будет создана однорядная таблица, состоящая из 12 колонок, первые четыре из которых отводятся под область h, а три последних образуют область f. Здесь символы разрыва текстовой строки обрабатываются так же, как и пробелы: они разделяют элементы шаблона, а не обозначают конец ряда сетки, как в предыдущих случаях.

Как оговаривалось выше, каждый элемент шаблона представляет отдельную ячейку сетки. Вернувшись к самому первому примеру раздела, визуально обозначим области сетки их буквенными идентификаторами (рис. 13.27).

```
#grid {display: grid;
   grid-template-areas:
      "h h h h"
      "l c c r"
      "l f f f";}
```

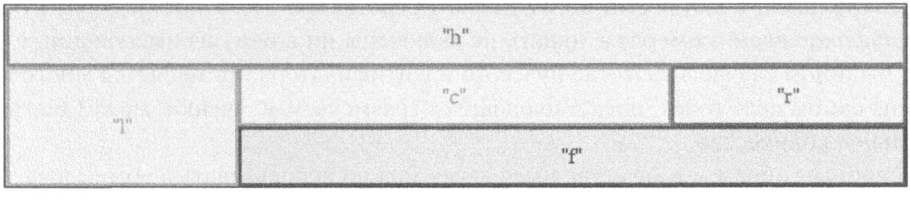


Рис. 13.27. Области сетки и их идентификаторы

Структура сетки остается прежней, но теперь в ее областях отображаются идентификаторы, представленные в шаблоне стилевого свойства `grid-template-areas`.

Ячейки с одним и тем же идентификатором объединяются в отдельную область сетки, но только в случае, если она имеет прямоугольную форму. Попытка создания областей более сложной формы обречена на провал. Как, например, в случае выполнении такого кода.

```
#grid {display: grid;
   grid-template-areas:
      "h h h h"
      "l c c r"
      "l l f f";}
```

Обратите внимание на область, обозначенную идентификатором `l`: она имеет L-образную форму, как бы символически это ни выглядело. Включение ее в объявление свойства сетки приводит к сбрасыванию не только текущей, но и всех остальных ее областей. Поддержка областей произвольной формы, скорее всего, будет реализована в будущих версиях спецификации, а на данный момент авторам приходится довольствоваться только инструментами рисования прямоугольных форм.

Если именованные области заполняют только часть пространства сетки, то все не относящиеся к ним ячейки нужно представить в шаблоне точками. Предположим, что на сетке требуется образовать только шапку, подвал и боковые панели, оставив неразмеченной центральную область станицы. Для решения этой задачи можно использовать такой CSS-код, применение которого к документу приводит к результату, показанному на рис. 13.28.

```
#grid {display: grid;
   grid-template-areas:
      "header header header header"
      "left   ...    ...    right"
      "footer footer footer footer";}
```

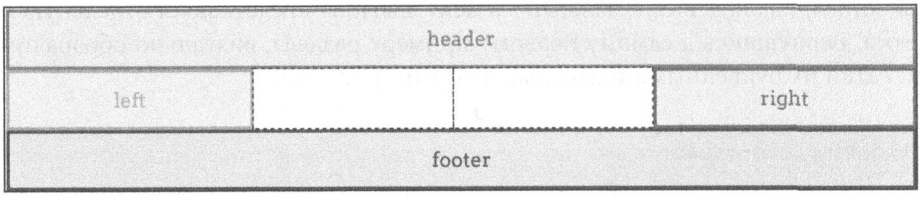


Рис. 13.28. Сетка с несколькими неразмеченными ячейками

Центральные ячейки grid-контейнера, представленные в объявлении *нулевым идентификатором* (символом точки), не включены ни в одну из именованных областей. В данном случае в качестве нулевого идентификатора применяется многоточие, хотя на самом деле точек, представляющих неразмечаемые ячейки, может быть произвольное количество.

Подобным образом в качестве имен ячеек можно использовать любые слова, представляемые символами Unicode, например `ronaldo` (шапка страницы) и `podiatrist` (подвал страницы). В именах областей допускается использование абсолютно любых символов кодовой страницы U+0080. В частности, легитимными будут такие названия, как `ConHugeCo©®™` и `åwësømë`, а также всевозможные символы эмодзи.

Для определения размера колонок и рядов, относящихся к тем или иным областям, применяются свойства `grid-template-columns` и `grid-template-rows`, известные по предыдущим разделам. Воспользуемся ими обоими для представления структуры из предыдущего примера (рис. 13.29).

```
#grid {display: grid;
   grid-template-areas:
      "header header header header"
      "left   ...    ...    right"
      "footer footer footer footer";
   grid-template-columns: 1fr 20em 20em 1fr;
   grid-template-rows: 40px 10em 3em;}
```

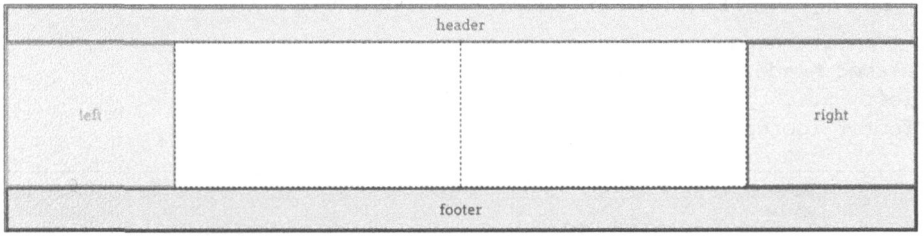


Рис. 13.29. Именованные области и размеры полос

По вполне очевидным причинам в сетке нужно объявлять столько колонок и рядов, сколько указано в шаблоне свойства `grid-template-areas`. Если их будет больше, чем предполагается шаблоном, то все лишние колонки и ряды будут добавляться на сетку вне именованных областей. Например, следующее стилевое правило обеспечивает создание сетки, представленной на рис. 13.30.

```
#grid {display: grid;
   grid-template-areas:
      "header header header header"
      "left   ...    ...    right"
      "footer footer footer footer";
   grid-template-columns: 1fr 20em 20em 1fr 1fr;
   grid-template-rows: 40px 10em 3em 20px;}
```

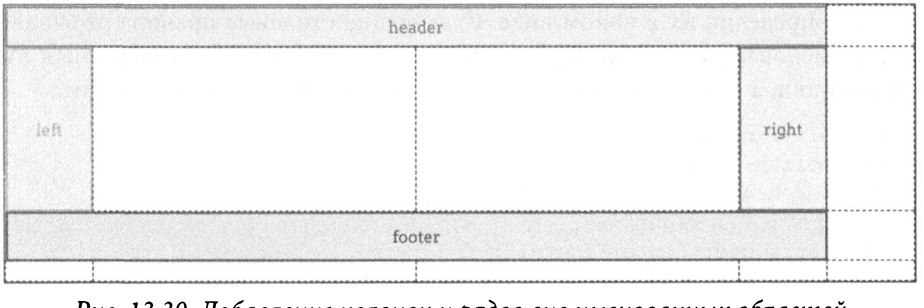


Рис. 13.30. Добавление колонок и рядов вне именованных областей

При именовании областей названия также получают линии, образующие их края. Имена назначаются линиям автоматически и состоят из названия соответствующей области и суффикса `start` или `end`, указывающего их положение относительно начальной точки области. В частности, первая вертикальная, равно как и горизонтальная, линия области получает название `header-start`. Линии сетки, образующие правый и нижний края области `header`, получают название `header-end`. Подобным образом именуются линии сетки, очерчивающие края области `footer`. В зависимости от положения они получают название `footer-start` или `footer-end`.

Учтите, что одни и те же линии сетки проходят через все пространство grid-контейнера, а потому образуют края сразу нескольких областей. Соответственно большинство линий будет иметь много имен, назначаемых автоматически при объявлении в grid-контейнере сразу нескольких областей (рис. 13.31).

```
grid-template-areas:
   "header header header header"
   "left   ...    ...    right"
   "footer footer footer footer";
```

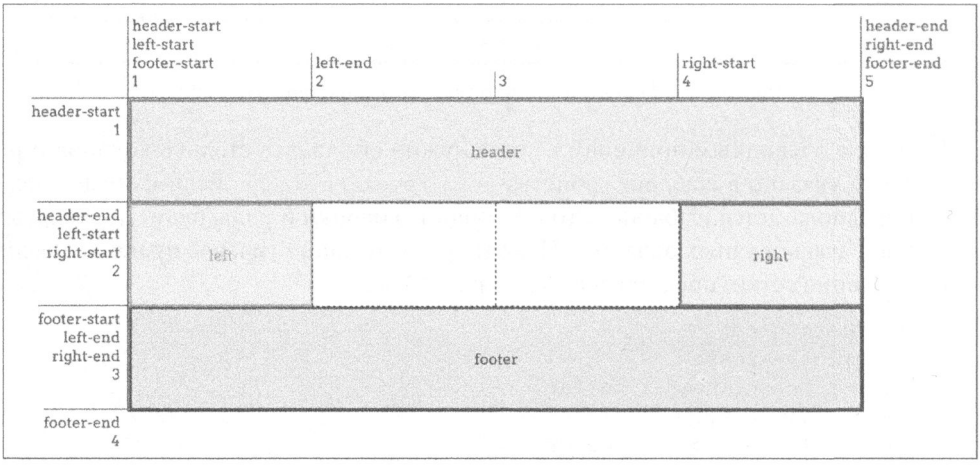


Рис. 13.31. Чрезмерно большое количество названий, генерируемых автоматически

Давайте еще больше усложним ситуацию, снабдив именами некоторые другие линии сетки, определив их в явном виде. Следующие стилевые правила объявляют в сетке две именованные линии. Пусть именем `begin` обозначается начальная линия первой колонки, а `content` соответствует начальной линии второй колонки.

```
#grid {display: grid;
   grid-template-areas:
      "header header header header"
      "left ... ... right"
      "footer footer footer footer";
   grid-template-columns: [begin] 1fr 20em 20em 1fr 1fr;
   grid-template-rows: 40px [content] 1fr 3em 20px;}
```

Снова-таки, объявленные в этом правиле названия дополняют имена, создаваемые автоматически в процессе образования в сетке именованных областей. Это общий принцип именования линий сетки — и новые, и старые названия сосуществуют в едином пространстве имен на равных правах.

Более важно то, что принцип неявного именования линий сетки прекрасно работает и в обратном направлении. Для образования именованных областей совсем не обязательно использовать свойство `grid-template-areas` — достаточно правильно объявить все образующие их именованные линии. Рассмотрим такой пример, результат выполнения которого показан на рис. 13.32.

```
grid-template-columns:
   [header-start footer-start] 1fr
   [content-start] 1fr [content-end] 1fr
   [header-end footer-end];
grid-template-rows:
   [header-start] 3em
   [header-end content-start] 1fr
   [content-end footer-start] 3em
   [footer-end];
```

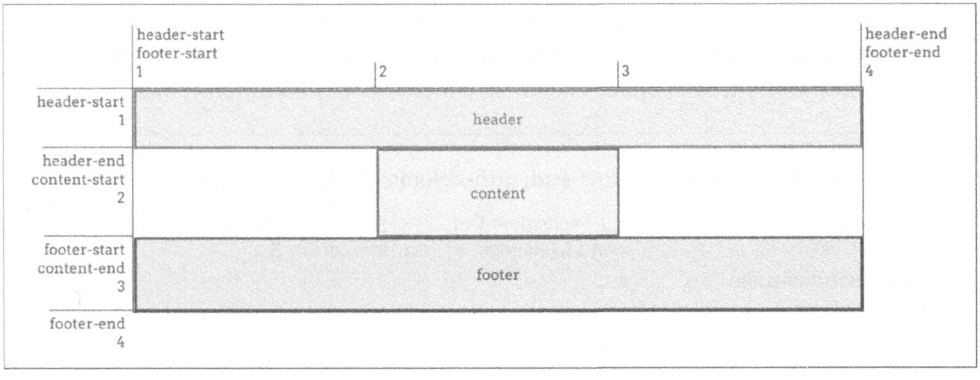


Рис. 13.32. Образование областей сетки при явном объявлении всех именованных линий

Как видите, для объявления именованных областей в неявном виде достаточно снабдить соответствующие линии сетки названиями формата *имя области*`-start/`*имя области*`-end`. По правде говоря, это очень странный метод, которым мало кто пользуется на практике, но тем не менее он оговаривается спецификацией.

Важно понимать, что объявление именованной области указанным способом состоится даже в случае добавления корректных названий только части образующих линий, как показано в следующем примере. Тем не менее для точного позиционирования именованной области названия нужно добавлять ко всем четырем ее линиям.

```
grid-template-columns: 1fr [content-start] 1fr [content-end] 1fr;
grid-template-rows: 3em 1fr 3em;
```

Выполнение правила приводит к размещению именованной области `content` в самой нижней части сетки, под всеми отдельно объявленными в ней рядами.

Интересно то, что перед именованной областью — сразу после регулярных рядов — в сетку добавляется новый пустой ряд. Весьма неожиданное поведение, к которому нужно быть готовым при образовании областей путем объявления в сетке именованных линий. Если не снабдить соответствующим названием хотя бы одну из ограничивающих именованную область линий, то она будет присоединяться к одному из краев сетки, вместо того чтобы встраиваться в ее общую структуру.

Исходя их вышесказанного, лучше остановиться на явном способе объявления именованных областей, обеспечивающем более точное позиционирование их на сетке и автоматическое именование образующих линий.

Привязка элементов к сетке

Как бы там ни было, приведенные выше рассуждения о создании grid-контейнера и образовании в нем сетки, состоящей из колонок и рядов, ровным счетом ничего не стоят без привязки к ней элементов.

Наполнение колонок и рядов

Существует несколько способов размещения элементов на сетке grid-контейнера. Они отличаются только способом привязки grid-элементов: к линиям или областям сетки. Для начала ознакомимся с принципом привязки элементов к линиям сетки, а затем перейдем к изучению более сложных способов их позиционирования.

grid-row-start, grid-row-end, grid-column-start, grid-column-end

Значение	`auto` | *<custom-ident>* | [*<integer>* && *<custom-ident>*?] | [span && [*<integer>* || *<custom-ident>*]]
Начальное значение	`auto`
Применяется	Grid-элементы и абсолютно позиционированные элементы, заключенные в grid-контейнеры
Вычисляется	Согласно определению
Наследуется	Нет
Анимируется	Нет

Эти свойства указывают, к какой линии сетки и каким краем будет привязываться элемент, для которого они объявляются. Их, как и любые другие свойства верстки по сетке, проще всего описывать на реальных примерах. С этой целью изучим следующие стилевые правила, результат применения которых продемонстрирован на рис. 13.33.

```
.grid {display: grid; width: 50em;
   grid-template-rows: repeat(5, 5em);
   grid-template-columns: repeat(10, 5em);}
.one {
   grid-row-start: 2; grid-row-end: 4;
   grid-column-start: 2; grid-column-end: 4;}
```

```
.two {
   grid-row-start: 1; grid-row-end: 3;
   grid-column-start: 5; grid-column-end: 10;}
.three {
   grid-row-start: 4;
   grid-column-start: 6;}
```

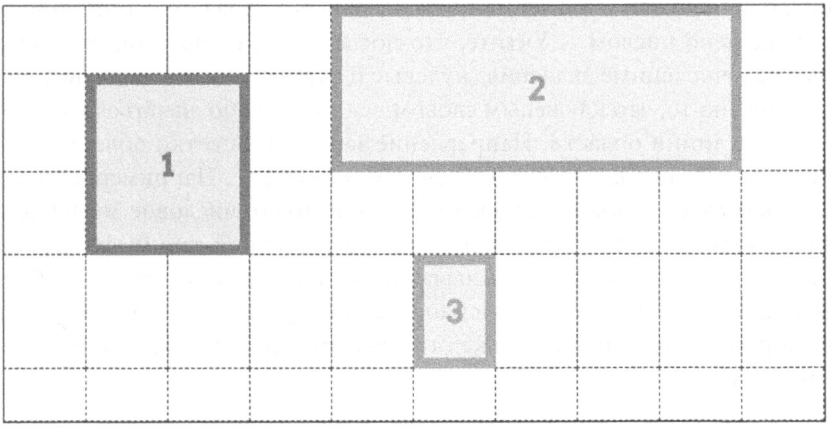


Рис. 13.33. Привязка элементов к сетке

Положение и способ привязки элементов к линиям сетки указывается номерами линий. Номера колонок отсчитываются слева направо, а номера рядов — сверху вниз. Заметьте: если не указать номер конечной линии, как в случае элемента `.three`, то она будет представлена следующей линией последовательности.

В данном случае стилевое правило `.three` можно переписать в таком виде.

```
.three {
   grid-row-start: 4; grid-row-end: 5;
   grid-column-start: 6; grid-column-end: 7;}
```

Для того чтобы получить подобный результат, можно применить специальную конструкцию, включающую ключевое слово `span`. В случае элемента `.three` размер области устанавливается значением `span 1`, или просто `span`.

```
.three {
   grid-row-start: 4; grid-row-end: span 1;
   grid-column-start: 6; grid-column-end: span;}
```

Указав число после ключевого слова `span`, можно определить количество полос, на которые распространяется область. Если переписать код приведенного выше примера с использованием ключевого слова `span`, то он примет такой вид.

```
#grid {display: grid;
   grid-template-rows: repeat(5, 5em);
   grid-template-columns: repeat(10, 5em);}
.one {
   grid-row-start: 2; grid-row-end: span 2;
   grid-column-start: 2; grid-column-end: span 2;}
```

```
.two {
   grid-row-start: 1; grid-row-end: span 2;
   grid-column-start: 5; grid-column-end: span 5;}
.three {
   grid-row-start: 4; grid-row-end: span 1;
   grid-column-start: 6; grid-column-end: span;}
```

В отсутствие числового значения после ключевого слова `span` оно по умолчанию будет представлено числом `1`. Учтите, что после `span` можно вводить только положительные целочисленные значения (нулевые и отрицательные числа недопустимы).

Примечательно то, что ключевым словом `span` могут обозначаться как начальные, так и конечные линии области. Направление заполнения сетки областью определяется линией, положение которой задается значением `span`. Например, если ключевое слово `span` указывает положение конечной линии, то его числовое значение устанавливает протяженность области от начальной до конечной линии. И наоборот, при добавлении ключевого слова `span` к начальной линии его числовое значение будет определять протяженность области в направлении от конечной линии к начальной.

Таким образом, приведенные ниже стилевые правила приводят к результату, показанному на рис. 13.34.

```
#grid {display: grid;
   grid-rows: repeat(4, 2em); grid-columns: repeat(5, 5em);}
.box01 {grid-row-start: 1; grid-column-start: 3; grid-column-end: span 2;}
.box02 {grid-row-start: 2; grid-column-start: span 2; grid-column-end: 3;}
.box03 {grid-row-start: 3; grid-column-start: 1; grid-column-end: span 5;}
.box04 {grid-row-start: 4; grid-column-start: span 1; grid-column-end: 5;}
```

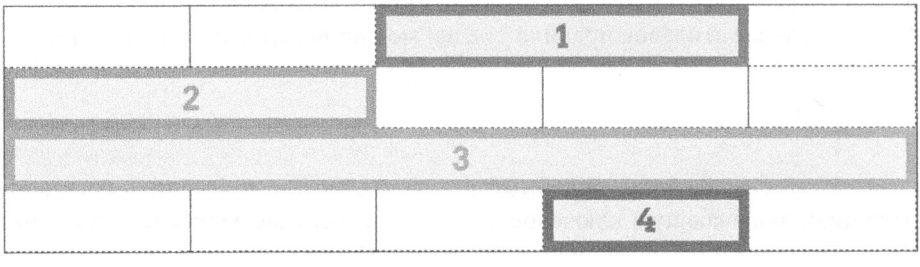


Рис. 13.34. Заполнение пространства между линиями сетки

При указании положения линий в явном виде, в отличие от установки их с помощью значения `span`, соответствующим свойствам можно передавать отрицательные числовые значения. Отсчет отрицательных значений ведется с конца сетки. В частности, для расположения элемента в правой нижней ячейке сетки с произвольным количеством колонок и рядов достаточно объявить для него следующие свойства.

```
grid-column-start: -1;
grid-row-start: -1;
```

Учтите, что такой подход неприменим к линиям колонок и рядов, устанавливаемым в неявном виде, о чем рассказано далее. Он справедлив только для явно задаваемых свойств формата `grid-template-*` (например, `grid-template-rows`).

Адресация линий сетки осуществляется не только через целочисленные, но и именованные значения. Если в сетке имеется сразу несколько одинаково именованных линий, то для точной идентификации наряду с именем нужно указывать номер экземпляра линии. Таким образом, запись `mast-slice 4` определяет четвертый экземпляр линии с названием `mast-slice`. Чтобы лучше понять правила адресации линии по имени, рассмотрим следующий пример (рис. 13.35).

```
#grid {display: grid;
   grid-template-rows: repeat(5, [R] 4em);
   grid-template-columns: 2em repeat(5, [col-A] 5em [col-B] 5em) 2em;}
.one {
   grid-row-start: R 2; grid-row-end: 5;
   grid-column-start: col-B; grid-column-end: span 2;}
.two {
   grid-row-start: R; grid-row-end: span R 2;
   grid-column-start: col-A 3; grid-column-end: span 2 col-A;}
.three {
   grid-row-start: 9;
   grid-column-start: col-A -2;}
```

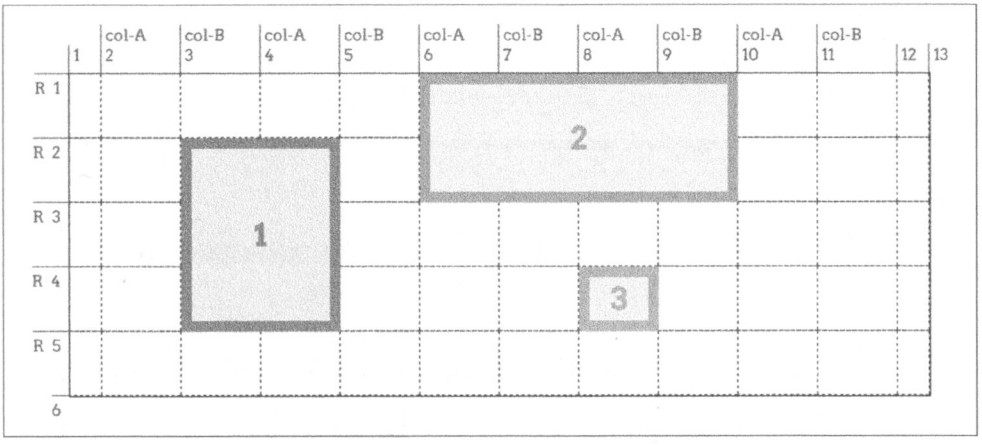


Рис. 13.35. Привязка элементов к именованным линиям сетки

В данном примере примечателен способ использования значения, представленного ключевым словом `span`. Конечная вертикальная линия второго элемента устанавливается значением `span 2 col-A`: его начало обозначается третьим, а конец — пятым экземпляром линии `col-A` (через два экземпляра линии `col-A` после начальной точки). Таким образом, элемент занимает сразу четыре колонки, поскольку в обозначенной сетке линии `col-A` чередуются только с линиями `col-B`.

Как уже упоминалось, отрицательные значения указывают на нумерацию с конца последовательности. Следовательно, значение `col-A -2` обозначает второй с конца экземпляр линии `col-A`. Так как для элемента `.three` конечные линии не определены, то в стилевом правиле они будут представлены значением `span 1`. Следующее объявление полностью равнозначно применяемому для элемента `.three` в приведенном выше CSS-коде.

```
.three {
   grid-row-start: 9; grid-row-end: span 1;
   grid-column-start: col-A -2; grid-row-end: span 1;}
```

Существует альтернативный способ привязки элементов к линиям сетки. Особенно часто он применяется при образовании именованных областей и добавлении названий к линиям сетки в неявном виде. В качестве примера изучим следующий CSS-код, выполнение которого приводит к стилевому форматированию grid-элементов, показанному на рис. 13.36.

```
grid-template-areas:
   "header   header  header  header"
   "leftside content content rightside"
   "leftside footer  footer  footer";
#masthead {grid-row-start: header;
   grid-column-start: header; grid-row-end: header;}
#sidebar {grid-row-start: 2; grid-row-end: 4;
   grid-column-start: leftside / span 1;}
#main {grid-row-start: content; grid-row-end: content;
   grid-column-start: content;}
#navbar {grid-row-start: rightside; grd-row-end: 3;
   grid-column-start: rightside;}
#footer {grid-row-start: 3; grid-row-end: span 1;
   grid-column-start: footer; grid-row-end: footer;}
```

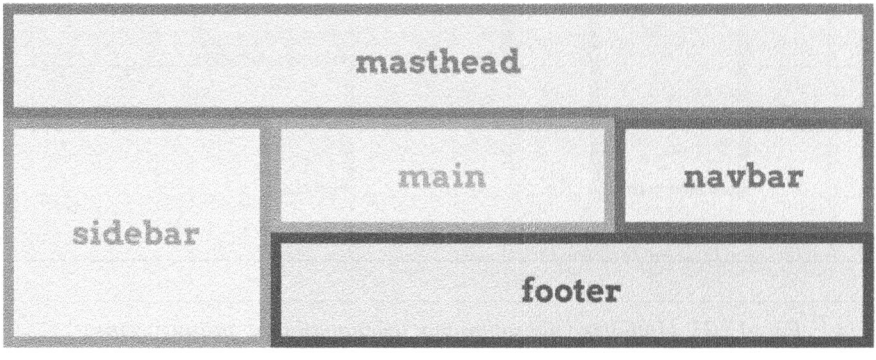


Рис. 13.36. Еще один способ привязки элементов к именованным линиям сетки

При назначении линии сетки собственного идентификатора (имени, определенного в явном виде) браузер будет автоматически снабжать его суффиксом `-start` или `-end` в зависимости от ее положения: в начале или в конце привязываемого элемента. Исходя из этого, при выполнении следующих двух правил будет получен абсолютно одинаковый результат.

```
grid-column-start: header; grid-column-end: header;
grid-column-start: header-start; grid-column-end: header-end;
```

Работоспособность метода обеспечивается, как и в случае объявления областей с помощью свойства `grid-template-areas`, автоматическим добавлением суффикса к названию линии, обозначающей начало или конец элемента.

Последнее значение, `auto`, имеет несколько иное предназначение, чем все рассмотренные выше. Согласно спецификации, установка одного из свойств, определяющих начальную и/или конечную линии привязки, в значение `auto` указывает на полностью автоматическое позиционирование элемента относительно линий сетки. В общем случае положение элементов определяется порядком добавления их на сетку (понятие потока элементов сетки рассмотрено в одном из следующих разделов). Если ключевым словом `auto` задается начальная линия элемента, то это означает, что он будет заполнять следующую свободную колонку или ряд. При установке с его помощью конечной линии элемент, скорее всего, будет занимать лишь одну ячейку сетки. Фраза "скорее всего" играет в обозначенном выше правиле важную роль, подразумевая неабсолютность его исполнения в определенных условиях.

Свойства создания колонок и рядов общего назначения

Привязка колонок и рядов к сетке grid-контейнера выполняется с помощью всего двух свойств общего назначения.

grid-row, grid-column

Значение	*<grid-line>* [/ *<grid-line>*]?
Начальное значение	`auto`
Применяется	Grid-элементы и абсолютно позиционированные элементы, заключенные в grid-контейнеры
Вычисляется	Согласно определению
Наследуется	Нет
Анимируется	Нет

Преимущество методов, заключающихся в их использовании, состоит в простоте синтаксиса объявления начальной и конечной линий привязки. Рассмотрим такой пример.

```
#grid {display: grid;
   grid-template-rows: repeat(10, [R] 1.5em);
   grid-template-columns: 2em repeat(5, [col-A] 5em [col-B] 5em) 2em;}
.one {
   grid-row: R 3 / 7;
   grid-column: col-B / span 2;}
.two {
   grid-row: R / span R 2;
   grid-column: col-A 3 / span 2 col-A;}
.three {
   grid-row: 9;
   grid-column: col-A -2;}
```

Изучать такой код намного проще, чем правила, в которых начальные и конечные точки привязки элементов устанавливаются отдельными стилевыми свойствами. Компактность представления значений существенно упрощает анализ кода свойств

общего назначения. При передаче такому свойству значения, состоящего из двух разделенных косой чертой величин, первая из них будет указывать начальную, а вторая — конечную линию привязки.

Значение, лишенное косой черты, устанавливает только начальную линию привязки. В подобных случаях конечная линия привязки зависит от значения, представляющего начальную линию. Если в качестве начальной использована именованная линия сетки, то конечная линия привязки будет представлена таким же названием. Следовательно, следующие два объявления полностью идентичны.

```
grid-column: col-B;
grid-column: col-B / col-B;
```

Каждое из объявлений указывает элементу заполнить пространство от первого до следующего экземпляра линии `col-B`, независимо от количества других линий сетки, находящихся между ними.

При передаче свойству общего назначения только числового значения второе значение будет обозначаться ключевым словом `auto`. Таким образом, идентичными оказываются следующие пары объявлений.

```
grid-row: 2;
grid-row: 2 / auto;

grid-column: header;
grid-column: header / header;
```

Обработка неявно заданных именованных линий с помощью стилевых свойств `grid-row` и `grid-column` позволяет добиться весьма интересных результатов. Как было показано выше, при объявлении в сетке именованных областей образующие их линии автоматически получают названия, включающие имя области и суффикс `start` или `end`. В частности, при образовании области с названием `footer` ее начальные линии (слева и вверху) будут иметь название `footer-start`, а конечные линии (справ и внизу) — `footer-end`.

При передаче таких названий свойствам общего назначения элементы будут привязываться к автоматически именованным линиям так же, как и к любым другим линиям сетки. Пример применения такого способа привязки элементов к сетке представлен на рис. 13.37. Он получен в результате выполнения следующего CSS-кода.

```
#grid {display: grid;
   grid-template-areas:
      "header header"
      "sidebar content"
      "footer footer";
   grid-template-rows: auto 1fr auto;
   grid-template-columns: 25% 75%;}
#header {grid-row: header / header; grid-column: header;}
#footer {grid-row: footer; grid-column: footer-start / footer-end;}
```

К линиям сетки с неявно заданными именами можно обращаться напрямую, хотя в большинстве случаев достаточно указать только название области, которую они образуют. При передаче свойствам общего назначения имен линий, не относящихся

к определенным в сетке областям, они будут обрабатываться согласно общим правилам, рассмотренным выше. Чаще всего такое объявление будет равнозначно включению в стилевое правило записи `имя-линии 1`, как в следующем примере.

```
grid-column: jane / doe;
grid-column: jane 1 / doe 1;
```

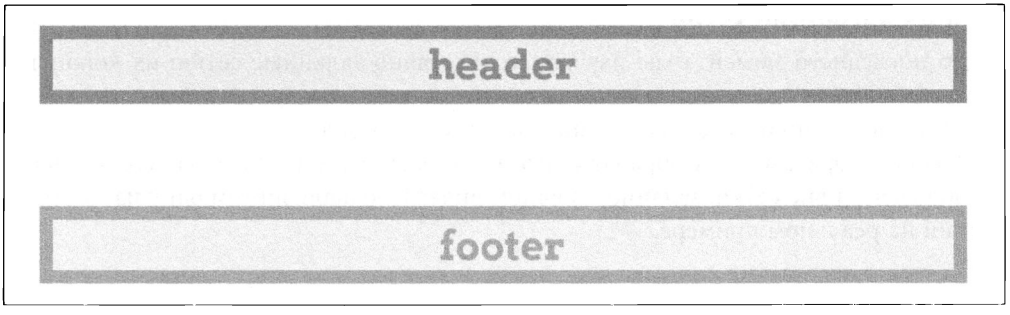


Рис. 13.37. Привязка элементов к именованным линиям, определенным в неявном виде

Именно эта причина побуждает отказаться от явного определения в сетке областей и линий с одинаковыми названиями. Такая ситуация представлена в следующем примере.

```
grid-template-areas:
   "header header"
   "sidebar content"
   "footer footer"
   "legal legal";
grid-template-rows: auto 1fr [footer] auto [footer];
grid-template-columns: 25% 75%;
```

Детальный анализ кода показывает, что в такой сетке линия с явно заданным названием "footer" должна размещаться над рядом `footer`, но под рядом `legal`, что невозможно ни при каких обстоятельствах. Теперь рассмотрим такой пример:

```
#footer {grid-column: footer; grid-row: footer;}
```

Объявленные таким способом колонки распознаются без особых усилий. Значение `footer` воспринимается браузерами как `footer/footer`, с легкостью транслируемое в `footer-start/footer-end` при обнаружении области с названием `footer`. В результате элемент `#footer` привязывается к указанным линиям сетки, названия которых определены неявным образом.

Анализ свойства `grid-row` выполняется похожим образом. Значение `footer` приводится к более общему `footer/footer`, впоследствии распознаваемому как `footer-start/footer-end`. Но такое объявление указывает на то, что элемент `#footer` располагается в ряду `footer`, а не продолжается до следующего явно заданного экземпляра линии `footer`, расположенного под рядом `legal`. А все потому, что значение `footer` преобразуется в `footer-end` ложным образом (исходя из "случайного" совпадения названий линий и области).

Подытожив вышесказанное, нужно признать, что обозначение областей и линий сетки одинаковыми именами — далеко не самая хорошая идея. Старайтесь избегать подобных сценариев верстки, назначая компонентам сетки уникальные имена. Это позволит избежать конфликтов, неизбежно возникающих при использовании одинаковых названий в общем пространстве имен.

Неявно заданная сетка

До последнего момента мы изучали только явно заданные сетки: их колонки и ряды объявлялись с помощью стилевых свойств, подобных `grid-template-columns`, а grid-элементы привязывались к линиям сетки напрямую.

А каким образом будет обрабатываться grid-элемент или отдельная его часть при расположении вне сетки, заданной в явном виде? Проанализируем одну из таких ситуаций на реальном примере.

```
#grid {display: grid;
   grid-template-rows: 2em 2em;
   grid-template-columns: repeat(6, 4em);}
```

Легко заметить, что объявленная с помощью такого правила сетка состоит из двух рядов и шести колонок. Предположим, что grid-элемент должен располагаться в первой колонке, но занимать целых три ряда: от первой до четвертой горизонтальной линии.

```
.box01 {grid-column: 1; grid-row: 1 / 4;}
```

Как быть? В текущей сетке содержится всего два ряда, образованных тремя горизонтальными линиями, но никак не три, как предполагает стилевое правило.

Для корректной обработки столь затруднительной ситуации браузер добавляет в сетку еще одну линию. В результате сетка пополняется еще одним рядом, относящимся к *неявной сетке* grid-контейнера. Несколько вариантов размещения grid-элементов в неявно заданных сетках приведено в следующем примере (рис. 13.38).

```
.box01 {grid-column: 1; grid-row: 1 / 4;}
.box02 {grid-column: 2; grid-row: 3 / span 2;}
.box03 {grid-column: 3; grid-row: span 2 / 3;}
.box04 {grid-column: 4; grid-row: span 2 / 5;}
.box05 {grid-column: 5; grid-row: span 4 / 5;}
.box06 {grid-column: 6; grid-row: -1 / span 3;}
.box07 {grid-column: 7; grid-row: span 3 / -1;}
```

Каждый из элементов требует специального описания, но начинать их рассмотрение нужно с операции объявления сетки. Определенная в явном виде сетка обозначена областью с серым фоном, а неявная сетка разграничена пунктирными линиями.

Добавляемые на сетку grid-элементы заключены в цветные прямоугольные рамки и пронумерованы согласно порядку объявления в стилевом правиле. Для размещения элемента `box01` в нижнюю часть сетки, как и в предыдущем примере, добавляется новый ряд. Начало элемента `box02`, занимающего два ряда неявно заданной сетки, совмещено с последней линией сетки, объявляемой в явном виде. Для его полноценного размещения неявная сетка дополняется еще одним рядом, расположенным под

уже существующим. Элемент box03 заканчивается последней линией явно заданной сетки, заполняя два предыдущих ряда. Следовательно, его начало совмещено с началом сетки.

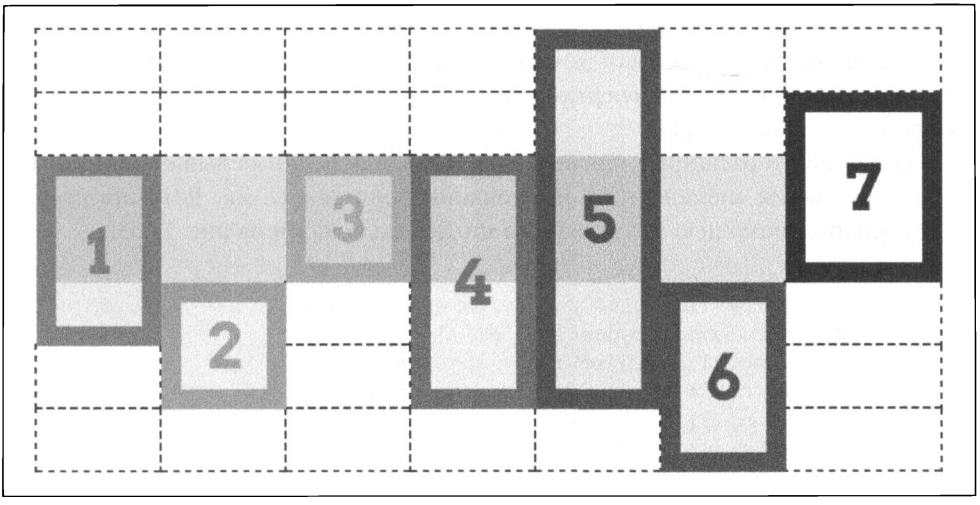


Рис. 13.38. Создание в сетке неявно заданных рядов

Положение четвертого элемента определяется чуть сложнее. Он заканчивается пятой линией, которая совпадает со второй линией неявно заданной сетки. При этом он, как и третий элемент, начинается с первой линии явно заданной сетки, несмотря на то что согласно правилу должен заполнять пространство между предыдущими тремя линиями. Такое несоответствие обусловлено необходимостью начинать отсчет с линий явной сетки. Линии неявной сетки также учитываются при определении размера элемента сетки (как в ситуации со вторым элементом), но начало отсчета всегда находится в явной сетке.

Таким образом, четвертый элемент заканчивается пятой линией сетки, но начинается на две линии раньше (span 2) третьей линии, как ближайшей к обозначенной правилом в явной сетке. Аналогично окончание элемента box05 обозначено пятой линией, а его начало располагается за четыре позиции от последней линии явной сетки (row-line -2). Общее правило позиционирования элементов гласит, что отсчет *всегда* начинается с линии явной сетки, но не обязательно заканчивается ею.

Шестой элемент начинается с последней (третьей) линии явной сетки и занимает последующие три ряда, добавляя один из них в уже существующую неявную сетку. Данный пример примечателен отрицательным значением: оно определяет не начальную линию элемента, а направление заполнения элементом пространства сетки, отсчитываемое от крайней линии явной сетки.

На примере элемента box07 показано, как можно объявлять элементы, начинающиеся в неявной сетке,которая расположена над явной сеткой, а заканчивающиеся, наоборот, в явной сетке. При решении задачи использован метод обратного позиционирования: конечная линия элемента находится в явной сетке, а сам он заполняет пространство в обратном направлении: снизу вверх. Особо наблюдательные могли

заметить, что седьмая колонка не относится к явной сетке, поскольку исходно сетка содержит всего шесть колонок. Следовательно, седьмая колонка добавляется к имеющейся явной сетке автоматически и становится частью неявной сетки. Согласно такому утверждению, к элементу `box07` автоматически применяется объявление `grid-column: 7`, в данном случае эквивалентное `grid-column: 7 / span 1` (при опускании конечная линия всегда обозначается значением `span 1`). Без объявления седьмой колонки, включаемой в неявную сетку, grid-элементу попросту не хватит места в существующей явной сетке.

Теперь немного расширим пример, использовав описанные выше принципы для позиционирования элементов по именованным линиям сетки. Результат верстки grid-элементов с помощью приведенного ниже кода показан на рис. 13.39.

```
#grid {display: grid;
   grid-template-rows: [begin] 2em [middle] 2em [end];
   grid-template-columns: repeat(5, 5em);}
.box01 {grid-column: 1; grid-row: 2 / span end 2;}
.box02 {grid-column: 2; grid-row: 2 / span final;}
.box03 {grid-column: 3; grid-row: 1 / span 3 middle;}
.box04 {grid-column: 4; grid-row: span begin 2 / end;}
.box05 {grid-column: 5; grid-row: span 2 middle / begin;}
```

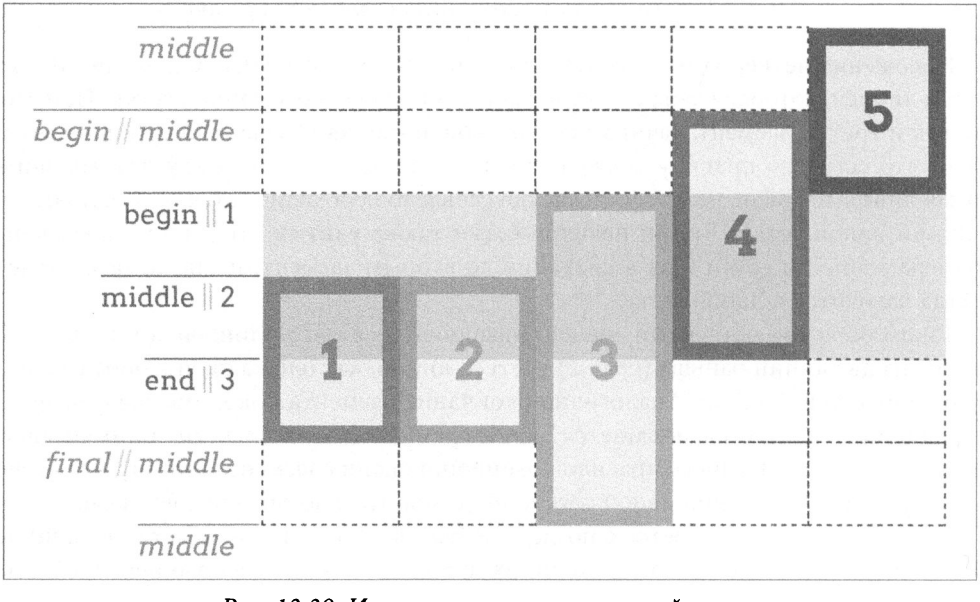


Рис. 13.39. Именованные линии неявной сетки

Этот пример призван показать, каким образом обрабатываются именованные линии неявно заданной сетки. В общем случае имя линии, отсутствующей в явно заданной сетке, назначается следующей линии неявной сетки. В объявлении элемента `box02` конечная линия имеет название `final`, отсутствующее в явной сетке. Ввиду отсутствия такой линии пользовательский агент создаст ее в неявной сетке по

первому же требованию стилевого правила. (На рис. 12.39 названия всех именованных линий неявной сетки представлены курсивным начертанием.)

Из приведенного выше правила следует, что начало элемента `box03` совмещено с первой линией явной сетки, а заканчивается он третьим экземпляром линии `middle`. Определив положение только первого экземпляра линии `middle`, пользовательский агент создает недостающие экземпляры линии с таким же названием автоматически. В результате сетка дополняется двумя линиями с названием `middle`, размещаемыми под уже существующим экземпляром, который объявлен при создании явной сетки.

Схожим образом именованные линии добавляются в неявно заданную сетку при создании элементов `box04` и `bpx05`, отличаясь только направлением их заполнения. Элемент `box04` заканчивается третьей линией (`end`) явной сетки, заполняя все предыдущее пространство до второго найденного экземпляра линии `begin`. Такое требование вызывает необходимость создания линии `begin` в неявной сетке перед уже существующим экземпляром, присутствующим в явной сетке. В случае пятого элемента начальная точка обозначатся линией `begin` явной сетки, а конечная отсчитывается от нее до нахождения второго экземпляра линии `middle`. Ввиду отсутствия в явной сетке они в необходимом количестве создаются в области неявной сетки, после чего применяются для позиционирования элемента, объявленного в стилевом правиле.

При детальном анализе неявная сетка оказывается достаточно архаическим инструментом, основное назначение которого — обеспечение целостности grid-контейнера. Чтобы избежать ее использования, научитесь обходиться одной только явной сеткой, не смещая grid-элементы за ее пределы. Обнаружив подобное поведение элементов, немедленно увеличьте размер сетки до более приемлемого.

Обработка ошибок

В завершение раздела, посвященного позиционированию grid-элементов на сетке, хотелось бы рассмотреть несколько стандартных ситуаций, которые могут вызвать затруднения при наполнении grid-контейнера содержимым.

Первая из таких ситуаций возникает при объявлении начальной линии элемента после его конечной линии. Например, как в этом коде.

```
grid-row-start: 5;
grid-row-end: 2;
```

Самая очевидная причина такого форматирования — значения перепутаны местами неумышленно. Следовательно, они поменяются местами друг с другом автоматически.

```
grid-row-start: 2;
grid-row-end: 5;
```

Следующая затруднительная ситуация возникает при представлении начальной и конечной линий элемента значениями, включающими ключевое слово `span`.

```
grid-column-start: span;
grid-column-end: span 3;
```

В подобных ситуациях последнее значение заменяется на auto.

```
grid-column-start: span; /* 'span' соответствует 'span 1' */
grid-column-end: auto;
```

В результате конечная линия элемента будет устанавливаться автоматически согласно принятому в сетке порядку следования элементов (рассматривается в одном из следующих разделов). При этом начальная линия будет отстоять от конечной всего на один ряд.

Наконец, затруднения могут возникнуть при позиционировании элементов, в которых в явном виде определено положение только одного из краев, да и то через ключевое слово span, сопровождаемое названием именованной линии.

```
grid-row-start: span footer;
grid-row-end: auto;
```

Такой вариант объявления элемента недопустим, поэтому значение span footer будет заменено на span 1.

Объявление областей сетки

Привязка grid-элементов к вертикальным или горизонтальным линиям сетки полезна далеко не во всех случаях. А можно ли разместить элементы в пределах некоей области сетки, воспользовавшись всего одним свойством? Конечно! Для этого предназначено свойство grid-area.

grid-area

Значение	<*grid-line*> [/ <*grid-line*>]{0,3}
Начальное значение	См. определения индивидуальных свойств
Применяется	Grid-элементы и абсолютно позиционированные элементы, заключенные в grid-контейнеры
Вычисляется	Согласно определению
Наследуется	Нет
Анимируется	Нет

Для начала рассмотрим самый очевидный и простой пример использования свойства grid-area, предполагающий привязку элемента к ранее объявленной области сетки. За основу возьмем пример, описанный в разделе, в котором описывается свойство grid-template-areas, добавив в него объявление свойства grid-area и снабдив специально размеченным фрагментом документа (рис. 13.40).

```
#grid {display: grid;
   grid-template-areas:
      "header   header  header  header"
      "leftside content content rightside"
      "leftside footer  footer  footer";}
```

```
#masthead {grid-area: header;}
#sidebar {grid-area: leftside;}
#main {grid-area: content;}
#navbar {grid-area: rightside;}
#footer {grid-area: footer;}

<div id="grid">
   <div id="masthead">…</div>
   <div id="main">…</div>
   <div id="navbar">…</div>
   <div id="sidebar">…</div>
   <div id="footer">…</div>
</div>
```

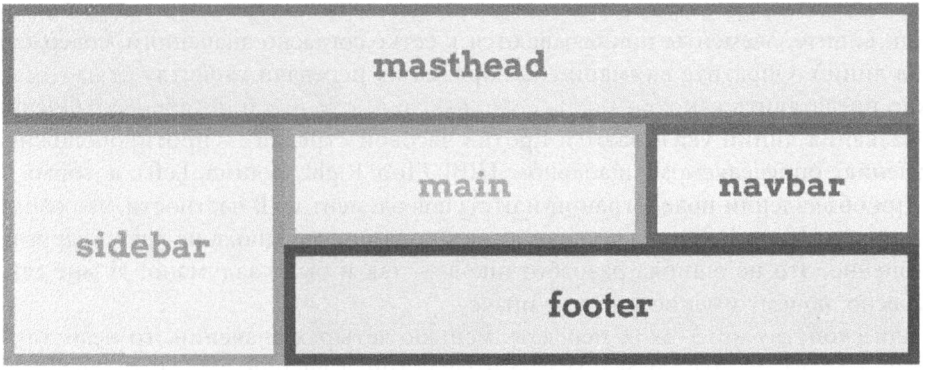


Рис. 13.40. Назначение элементов областям сетки

Как и предполагалось, в данном примере свойство `grid-area` применяется для наполнения элементами именованных областей, заранее объявленных в сетке. Очень просто и столь же эффективно!

Легко заметить, что в приведенном выше CSS-коде не указываются размеры колонок и рядов сетки. Это сделано умышленно, чтобы упростить код, акцентировав ваше внимание только на самом важном синтаксисе. Реальные стилевые правила верстки элементов по сетке выглядят немного запутаннее.

```
grid-template-areas:
   "header   header  header  header"
   "leftside content content rightside"
   "leftside footer  footer  footer";
grid-template-rows: 200px 1fr 3em;
grid-template-columns: 20em 1fr 1fr 10em;
```

Существует еще один способ применения свойства `grid-area`, заключающийся в передаче ему номеров или названий линий, а не областей. Он показан на рис. 13.41, а получен в результате выполнения следующего фрагмента CSS-кода.

```
#grid {display: grid;
   grid-template-rows:
      [r1-start] 1fr [r1-end r2-start] 2fr [r2-end];
```

```
   grid-template-columns:
      [col-start] 1fr [col-end main-start] 1fr [main-end];}
.box01 {grid-area: r1 / main / r1 / main;}
.box02 {grid-area: r2-start / col-start / r2-end / main-end;}
.box03 {grid-area: 1 / 1 / 2 / 2;}
```

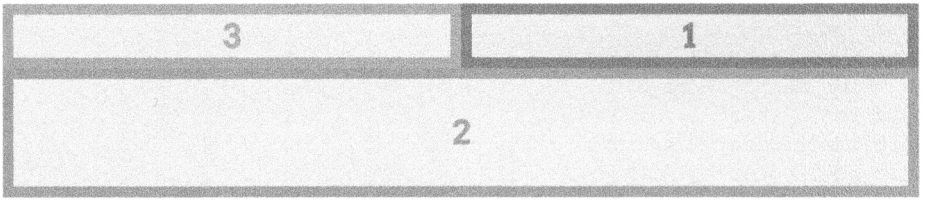


Рис. 13.41. Привязка элементов к сетке с помощью свойства `grid-area`

Как видите, элементы привязываются к сетке согласно значениям, содержащим имена линий. Обратите внимание на порядок их передачи свойству `grid-area`: его можно представить как `row-start`, `column-start`, `row-end` и `column-end`. Очевидно, что названия линий указываются против часовой стрелки — противоположно направлению, определяемому шаблоном TRBL (Top, Right, Bottom, Left), который принят при объявлении полей, границ и отступов элементов. В частности, это означает, что названия линий рядов и колонок передаются поочередно, а не друг за другом.

Конечно, это не ошибка разработчиков — так и было задумано! И мне самому интересно, почему именно так, а не иначе.

Если свойству `grid-area` передать меньше четырех значений, то недостающие значения будут продублированы уже имеющимися величинами. В частности, при включении в объявление всего трех значений недостающее последнее значение (`column-end`) будет представлено ключевым словом `column-start`. Если начальная линия обозначена числовым значением, то конечное значение определяется ключевым словом `auto`. Подобный принцип также сохраняется при подмене двух недостающих значений, но теперь `row-end` представляется копией значения `row-start`. Если значение `row-start` обозначается числовой величиной, то значение `row-end` будет представлено ключевым словом `auto`.

В случае передачи свойству `grid-area` единственного именованного значения оно будет копироваться во все три недостающие позиции. Если передается всего одно числовое значение, то все три недостающие величины будут определяться через ключевое слово `auto`.

В результате репликации единственного передаваемого значения свойство `grid-area` получает сразу четыре его экземпляра, как показано в приведенном ниже коде.

```
grid-area: footer;
grid-area: footer / footer / footer / footer;
```

Как известно из предыдущего раздела, если название линии совпадает с именем области сетки, которую оно образует, то к нему в зависимости от положения добавляется суффикс `-start` или `-end`. Таким образом, рассматриваемый нами код будет представляться для обработчика в следующем виде:

```
grid-area: footer-start / footer-start / footer-end / footer-end;
```

Такая запись более чем убедительно показывает, что элемент можно разместить в именованной области, представив ее в объявлении свойства `grid-area` всего одним значением.

Перекрывание grid-элементов

При верстке grid-элементов, размещаемых на сетке, необходимо проявлять крайнюю осторожность, избегая их перекрывания. Как и регулярные элементы, grid-элементы накладываются друг на друга при неправильно заданных настройках позиционирования. Один из возможных примеров такого их форматирования показан на рис. 13.42.

```
#grid {display: grid;
   grid-template-rows: 50% 50%;
   grid-template-columns: 50% 50%;}
.box01 {grid-area: 1 / 1 / 2 / 3;}
.box02 {grid-area: 1 / 2 / 3 / 2;}
```

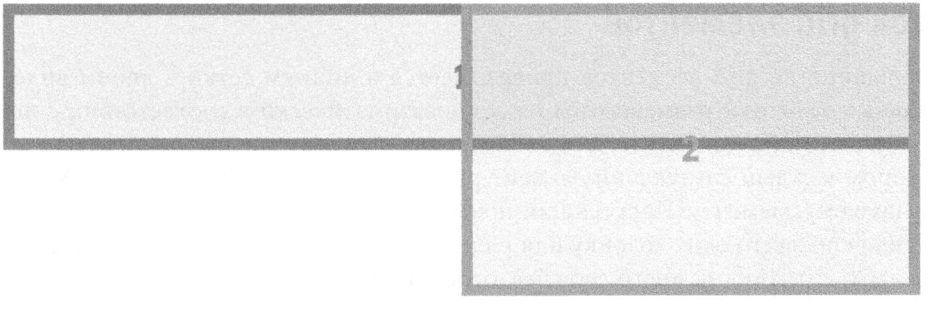


Рис. 13.42. Перекрывание grid-элементов

Привязка элементов к нумерованным линиям сетки согласно заданному в стилевом правиле порядку возможна только при накладывании grid-элементов. Порядок наложения определяется по принципам, описанным в одном из последующих разделов, а на данный момент достаточно знать, что grid-элементы размещаются на отдельных слоях.

Grid-элементы могут относиться не только к общим рядам, но и колонкам. В следующем примере (рис. 13.43) показана ситуация перекрывания подвала и боковой панели сайта. (В отсутствие другого стилевого форматирования подвал страницы будет накладываться на ее боковую панель, поскольку код его разметки обычно включается в документ после кода разметки боковой панели.)

```
#grid {display: grid;
   grid-template-areas:
      "header  header"
      "sidebar content"
      "footer  footer";}
#header {grid-area: header;}
#sidebar {grid-area: sidebar / sidebar / footer-end / sidebar;}
#footer {grid-area: footer;}
```

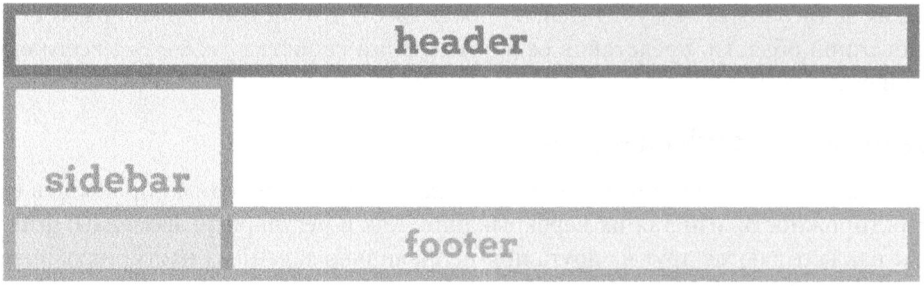


Рис. 13.43. Перекрывание подвала и боковой панели

Разговор о размещении grid-элементов на разных слоях затеян в преддверии следующего раздела неспроста. В нем мы поговорим о *потоке* grid-элементов, определяющем естественный порядок их расположения в сетке в отсутствие ручного позиционирования.

Поток grid-элементов

Большинство grid-элементов привязывается к линиям сетки в явном виде. Все остальные элементы размещаются на сетке автоматически в соответствии с порядком объявления в документе. Все добавляемые на сетку элементы образуют поток, в котором каждый следующий элемент располагается на сетке как можно ближе к предыдущему элементу. Простейший пример — последовательность grid-элементов, заполняющих всего одну колонку или ряд. В сетке, состоящей из нескольких колонок или рядов, ситуация намного сложнее, особенно при добавлении на нее как автоматически, так и явно позиционируемых grid-элементов.

В CSS существует возможность как порядного, так и поколоночного заполнения сетки элементами. Кроме того, grid-элементы можно размещать на сетке в максимально плотном порядке (см. ключевое слово `dense` в следующем объявлении). Способ размещения grid-элементов на сетке определяется стилевым свойством `grid-auto-flow`.

grid-auto-flow

Значение	[`row` | `column`] || `dense`
Начальное значение	`row`
Применяется	Grid-контейнеры
Вычисляется	Согласно определению
Наследуется	Нет
Анимируется	Нет

Анализ значений свойства будем проводить на примере такой разметки:

```
<ol id="grid">
   <li>1</li>
```

```
    <li>2</li>
    <li>3</li>
    <li>4</li>
    <li>5</li>
</ol>
```

Применим к указанному фрагменту документа следующее стилевое форматирование.

```
#grid {display: grid; width: 45em; height: 8em;
   grid-auto-flow: row;}
#grid li {grid-row: auto; grid-column: auto;}
```

Приведенные выше правила обеспечивают создание сетки (рис. 13.44), вертикальные линии которой размещаются с шагом `15em`, а горизонтальные — с шагом `4em`.

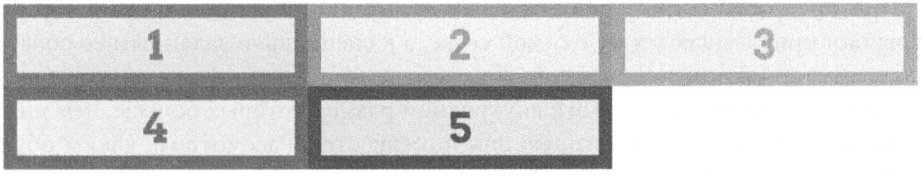


Рис. 13.44. Сетка, заполняемая элементами порядно

Такого же порядка расположения grid-элементов, обеспечиваемого значением по умолчанию `row`, можно добиться, выровняв их по соответствующему краю контейнера или приведя к строчному типу. При передаче свойству `grid-auto-flow` значения `column` будет получен несколько иной результат (рис. 13.45).

```
#grid {display: grid; width: 45em; height: 8em;
   grid-auto-flow: column;}
#grid li {grid-row: auto; grid-column: auto;}
```

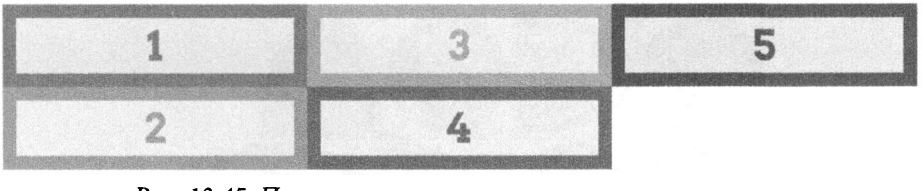


Рис. 13.45. Поколоночное заполнение сетки элементами

Если в случае объявления `grid-auto-flow: row` каждый следующий ряд добавлялся в сетку только после заполнения предыдущего, то при передаче свойству `grid-auto-flow` значения `column` эта операция выполняется по отношению к колонкам.

Необходимо учитывать, что в стилевых правилах не указывается размер ни одного из элементов списка. Чтобы установить их высоту и ширину, достаточно объявить размер представляющих их grid-элементов. В частности, для создания элементов списка шириной `6em` и высотой `1.5em` можно воспользоваться таким кодом (результат его выполнения показан на рис. 13.46).

```
#grid {display: grid; width: 45em; height: 8em;
   grid-auto-flow: column;}
#grid li {grid-row: auto; grid-column: auto;
   width: 7em; height: 1.5em;}
```

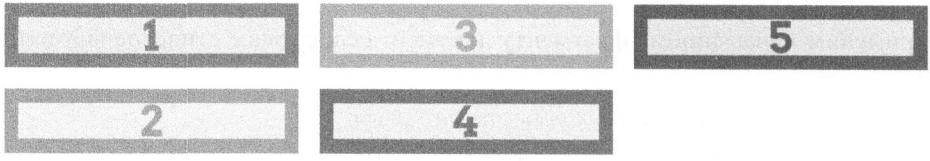


Рис. 13.46. Grid-элементы с явно заданным размером

Сетки, показанные на рис. 13.45 и 13.46, начинаются в одном и том же левом верхнем углу grid-контейнера, а заканчиваются в совершенно разных местах. Такое поведение прекрасно согласуется с главным принципом позиционирования grid-элементов: они привязываются не к самой сетке, а к специально заданным ее областям.

Об этом стоит помнить при автоматическом заполнении grid-контейнеров элементами (чаще всего изображениями), внутренний размер которых больше, чем ширина колонки или высота ряда, в которую они помещаются. Рассмотрим, каким образом будет форматироваться галерея графических изображений разного размера в сетке с колонками шириной 50 пикселей и рядами такой же высоты. Стилевое правило, решающее такую задачу, приведено ниже, а конечный результат показан на рис. 13.47.

```
#grid {display: grid;
   grid-template-rows: repeat(3, 50px);
   grid-template-columns: repeat(4, 50px);
   grid-auto-rows: 50px;
   grid-auto-columns: 50px;
}
img {grid-row: auto; grid-column: auto;}
```

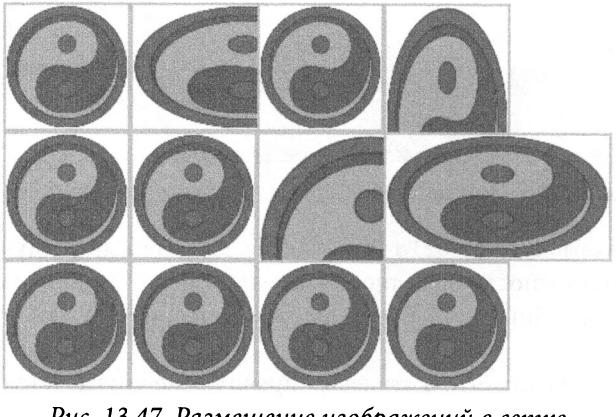


Рис. 13.47. Размещение изображений в сетке

На рис. 13.47 прекрасно видно, что отдельные изображения перекрываются содержимым соседних ячеек. Данный эффект вызван обязательной привязкой каждого

из изображений потока элементов к следующей линии сетки — без учета их размеров. Поскольку стилевое правило не предполагает расширение больших изображений на несколько колонок и рядов, они обязательно будут перекрываться на сетке.

Для исправления ситуации элементы изображений можно снабдить специальными идентификаторами или отнести к нескольким размерным классам. В частности, все изображения, размещаемые в несколько колонок или рядов, можно объявить классом `tall` или `wide` (либо отнести сразу к обоим классам). Следующие правила при добавлении в приведенный выше CSS-код позволяют добиться более приемлемого результата (рис. 13.48).

```
img.wide {grid-column: auto / span 2;}
img.tall {grid-row: auto / span 2;}
```

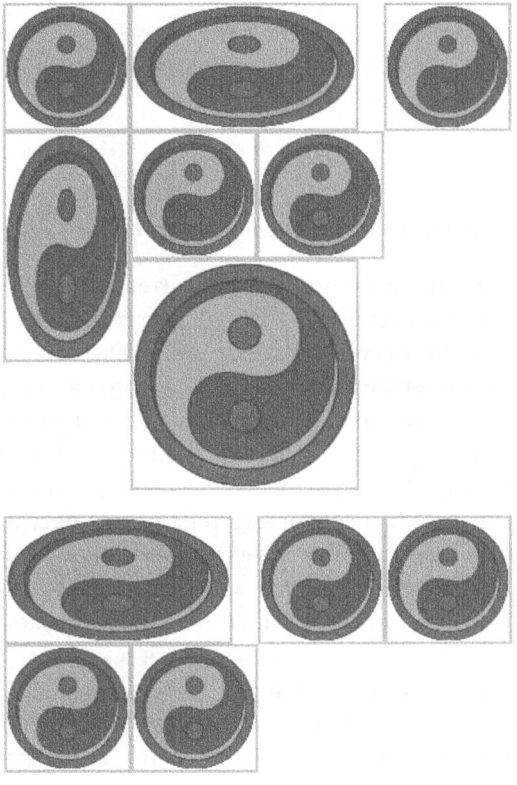


Рис. 13.48. Расширение полос сетки для корректного размещения изображений нестандартных размеров

Такой способ форматирования неизбежно приводит к нарушению визуального порядка размещения изображений в галерее.

На последней сетке хорошо заметно, что неравномерность отступов между grid-элементами вызывается недостатком места для упорядоченного размещения отдельных grid-элементов в общем потоке. Чтобы понять, почему это происходит, достаточно пронумеровать элементы в последовательности их добавления в grid-контейнер.

Два примера размещения grid-элементов с произвольными размерами на сетке приведены на рис. 13.49.

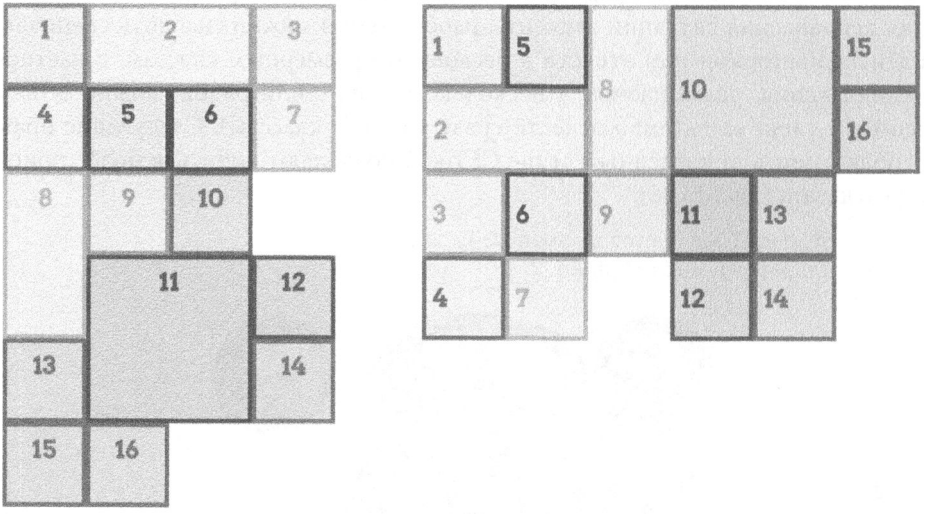


Рис. 13.49. Последовательность заполнения grid-контейнеров элементами

На первой сетке элементы заполняют grid-контейнер построчно. При таком направлении потока элементы располагаются на сетке так, как если бы они выравнивались по левому краю контейнера. Обратите внимание на размещение элемента 13 слева от элемента 11. Такое позиционирование невозможно при действительном выравнивании элементов в регулярных контейнерах. При автоматическом заполнении grid-контейнера каждый следующий ряд (по аналогии с рядами сетки) элементов заполняется слева направо. При этом, если в начале предыдущего ряда достаточно места для размещения элемента, переносимого на новый ряд, то он обязательно займет его. Если следующая ячейка сетки занята одним из более ранних элементов, то текущий элемент будет размещаться сразу же после нее. Таким образом, ячейка после элемента 10 (в конце ряда) остается незаполненной, поскольку элемент 11 в нее попросту не помещается, а элемент 13 размещается в ячейке, расположенной перед элементом 11, так как имеет достаточный для этого размер.

Рассмотренные выше принципы автоматического заполнения grid-контейнера сохраняются при поколоночном его наполнении (сверху вниз, как показано во втором примере на рис. 13.49). Можно заметить, что под элементом 9 недостаточно места для элемента 10, поэтому он переносится в верхнюю часть сетки, где занимает сразу четыре ячейки. Все расположенное под ним пространство заполняется в две колонки.

В приведенных на рис. 13.49 примерах направление заполнения сетки grid-элементами определяется системой письма, заданной для документа: слева направо и сверху вниз. В языках с направлением письма справа налево, например арабском, элементы будут заполнять grid-контейнер справа налево и сверху вниз.

Чтобы обеспечить максимально плотную упаковку сетки, нужно отказаться от общепринятого порядка заполнения ее grid-элементами, добавив к значению свойства `grid-auto-flow` ключевое слово `dense`. Результат такой операции показан на рис. 13.50: левый пример соответствует объявлению `grid-auto-flow: row dense`, а правый — объявлению `grid-auto-flow: dense column`.

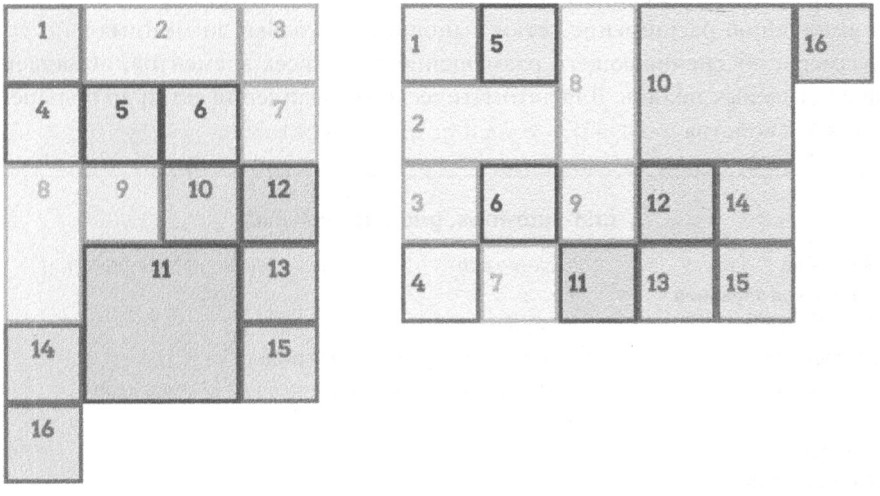


Рис. 13.50. Максимально плотный способ упаковки grid-контейнера элементами

В первом примере элемент 12 находится в ряду, расположенном над рядом с элементом 11, поскольку тот содержит достаточно места для его размещения. По этой же причине элемент 11 располагается слева от элемента 10 во втором примере.

При добавлении в объявление свойства `grid-auto-flow` ключевого слова `dense` пользовательский агент будет искать для элемента свободное место в указанном направлении (`row` или `column`) сразу во всей сетке, начиная с исходной точки (левый верхний угол для систем письма слева направо). Такой подход к автоматическому позиционированию позволяет представить коллекцию тематических grid-элементов, например графических изображений, в предельно компактном виде. Конечно, он может применяться только для верстки галерей, не требовательных к порядку размещения в них картин.

В приведенных выше примерах я утаил от вас одну существенную деталь: выравнивание последних элементов сетки осуществляется с помощью специального CSS-кода. В его отсутствие выступающие за края сетки элементы будут выглядеть по-другому в сравнении с остальными ее элементами: *уже* — в сетках с порядным и *ниже* — в сетках с поколоночным заполнением. О том, почему это происходит, рассказано в следующем разделе.

Автоматическое размещение линий сетки

Все описанные выше примеры позиционирования grid-элементов касались только контейнеров с явно заданными сетками. Тем не менее в последнем разделе нам

пришлось столкнуться с ситуацией выступания grid-элементов за пространство сетки, определенной в явном виде. Что же в действительности происходит в подобных ситуациях? Для размещения элементов на сетке она пополняется новыми колонками и рядами (см. раздел "Неявно заданная сетка"). Следовательно, при добавлении элемента, вертикальный размер которого определяется значением `span 3`, в конец явной сетки с порядным заполнением элементов она автоматически расширится на 3 ряда.

По умолчанию расширение сетки выполняется только до минимально возможного размера, обеспечивающего размещение в ней всех элементов, объявленных с помощью стилевых правил. Для автоматического увеличения сетки до большего размера служат свойства `grid-auto-rows` и `grid-auto-columns`.

grid-auto-rows, grid-auto-columns

Значение	*<track-breadth>* | `minmax(`*<track-breadth>*, *<track-breadth>*`)`
Начальное значение	auto
Применяется	Grid-контейнеры
Вычисляется	Согласно размеру колонки или ряда
Примечание	*<track-breadth>* соответствует *<length>* | *<percentage>* | *<flex>* | `min-content` | `max-content` | `auto`
Наследуется	Нет
Анимируется	Нет

Размер автоматически добавляемых в сетку колонок и рядов можно определить в явном виде или с помощью функции `minmax()`. Рассмотрим, как это реализуется, на упрощенном примере из предыдущего раздела, создав две сетки размером 2×2 и разместив в каждой из них по 5 элементов. В первом случае размер автоматически добавляемых рядов будет определяться свойством `grid-auto-rows`, а во втором — полностью автоматически (рис. 13.51).

```
.grid {display: grid;
   grid-template-rows: 80px 80px;
   grid-template-columns: 80px 80px;}
#g1 {grid-auto-rows: 80px;}
```

Как известно, при полностью автоматическом позиционировании grid-элементов высота дополнительного ряда определяется исключительно размером их содержимого. В данном примере она составляет 80 пикселей и равняется ширине колонок. Высота ряда, для которого свойство `grid-auto-rows` не объявлено, будет определяться значением по умолчанию: `auto`.

Схожая ситуация складывается при определении размера автоматически добавляемых колонок с помощью свойства `grid-auto-columns` (рис. 13.52).

```
.grid {display: grid; grid-auto-flow: column;
   grid-template-rows: 80px 80px;
   grid-template-columns: 80px 80px;}
#g1 {grid-auto-columns: 80px;}
```

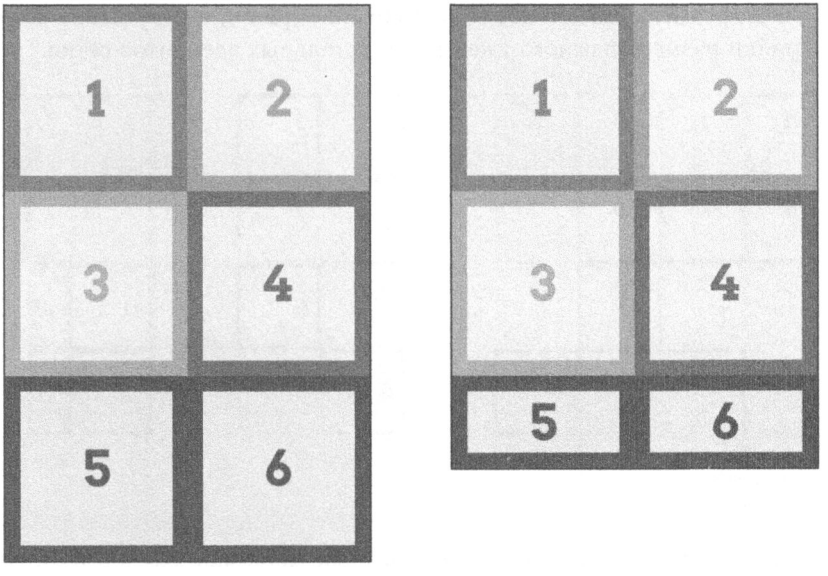


Рис. 13.51. Автоматически добавленные ряды с явно заданным размером и в отсутствие его

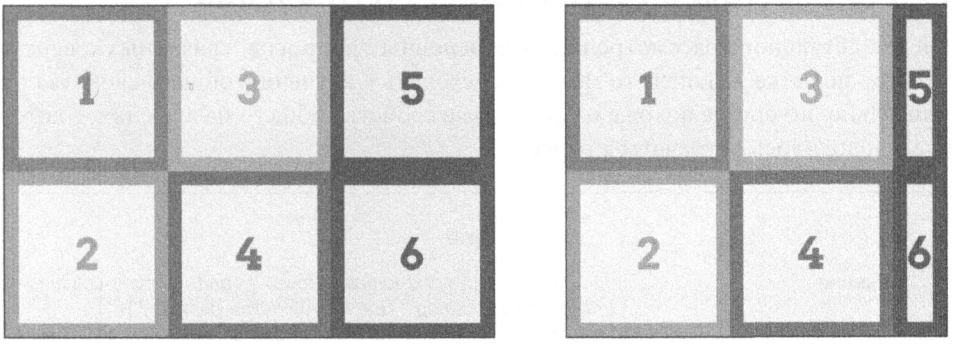


Рис. 13.52. Автоматически добавленные колонки с явно заданным размером и в отсутствие его

В первом примере, в котором сетка заполняется поколоночно, последний grid-элемент помещается в автоматически добавляемую колонку строго заданного размера. Во втором случае ширина добавляемой в сетку колонки устанавливается полностью автоматически — она соответствует размеру содержимого пятого grid-элемента, хотя и сопоставляется с высотой ряда (`80px`). При большем размере содержимого пятого элемента последняя автоматически добавляемая колонка будет заметно шире.

Заметьте, что в рассмотренных выше примерах у автоматически добавляемых колонок и рядов такой же размер, как и у колонок явно заданной сетки. Такой подход позволяет унифицировать ячейки сетки, делая ее приятней для просмотра. Рассмотрим, как будет выглядеть эта же сетка в отсутствие объявления для нее свойств `grid-auto-rows` и `grid-auto-columns` (рис. 13.53). Как видите, в подобном случае

grid-элементы, размещенные в последней колонке (ряду), в отсутствие автоматической настройки размера намного *уже* (ниже) остальных элементов сетки.

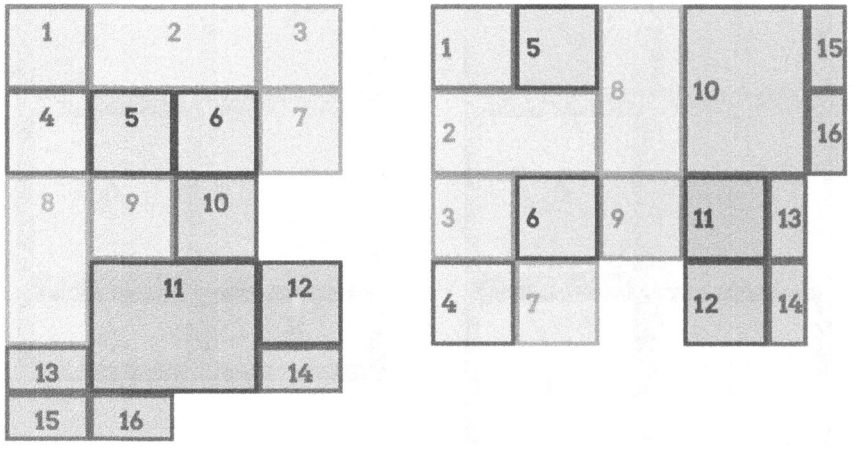


Рис. 13.53. Предыдущая сетка в отсутствие автоматической настройки дополнительных полос

Свойство настройки сетки общего назначения

После детального рассмотрения всех основных вопросов, связанных с верстской элементов по сетке, наконец-то можно переходить к изучению общего свойства `grid`. Удивительно, но оно не похоже на остальные свойства общего назначения, с которыми вам доводилось встречаться ранее.

grid

Значение	`none` | `subgrid` | [*<grid-template-rows>* / *<grid-template-columns>*] | [*<linenames>*?*<string>* *<track-size>*? *<line-names>*?]+ [/ *<track-list>*]? | [*<grid-autoflow>*[*<grid-auto-rows>* [/ *<grid-auto-columns>*]?]?]]
Начальное значение	См. описание индивидуальных свойств
Применяется	Grid-контейнеры
Вычисляется	См. описание индивидуальных свойств
Наследуется	Нет
Анимируется	Нет

Синтаксис свойства `grid` достаточно громоздкий, но при детальном изучении не кажется сложным для понимания.

Начнем с основного определения: единый синтаксис свойства `grid` позволяет либо установить шаблон сетки, либо указать направление ее заполнения grid-элементами и размер автоматически добавляемых полос. Решить обе задачи в одном объявлении не представляется возможным.

Более того, не указанные в свойстве настройки автоматически сбрасываются до значений по умолчанию, как и в любых других свойствах общего назначения. Таким образом, при определении с помощью свойства `grid` шаблона сетки в значения по умолчанию устанавливаются направление ее заполнения элементами и размер автоматически добавляемых полос (в том числе и интервал между ними, описанный в следующем разделе). Учтите, что это свойство не позволяет устанавливать произвольные интервалы между полосами, а только сбрасывать их в состояние по умолчанию.

Конечно, это сделано преднамеренно! И мне совершенно неизвестно почему!

Рассмотрим, каким образом свойство `grid` применяется для создания шаблона сетки. Несмотря на запутанный синтаксис, код объявления состоит из нескольких стандартных частей, каждая из которых поддается вполне простому описанию. Чтобы понять, что он собой представляет, достаточно сравнить следующих два примера, выполнение которых приводит к идентичному результату.

```
grid:
   "header header header header" 3em
   ". content sidebar ." 1fr
   "footer footer footer footer" 5em /
   2em 3fr minmax(10em,1fr) 2em;

grid-template-areas:
   "header header header header"
   ". content sidebar ."
   "footer footer footer footer";
grid-template-rows: 3em 1fr 5em;
grid-template-columns: 2em 3fr minmax(10em,1fr) 2em;
```

Обратите внимание на то, что в первом примере в конец каждой строки с названиями областей сетки, определяемых свойством `grid-template-areas`, добавляется одно из значений свойства `grid-template-rows`. Таким образом, в стилевом свойстве `grid`, объявляющем сетку с именованными областями, указываются размеры рядов. Если исключить из кода свойства `grid` объявление именованных областей, то он примет следующий вид.

```
grid:
   3em 1fr 5em / 2em 3fr minmax(10em, 1fr) 2em;
```

Иными словами, раздел, в котором указывается положение горизонтальных линий сетки, отделяется косой чертой от раздела, в котором определяется положение ее вертикальных линий.

Не забывайте о том, что все настройки сетки, не указанные в стилевом свойстве `grid`, автоматически получают значения по умолчанию. В частности, это означает равенство следующих двух стилевых правил.

```
#layout {display: grid;
   grid: 3em 1fr 5em / 2em 3fr minmax(10em, 1fr) 2em;}

#layout {display: grid;
```

```
    grid: 3em 1fr 5em / 2em 3fr minmax(10em,1fr) 2em;
    grid-auto-rows: auto;
    grid-auto-columns: auto;
    grid-auto-flow: row;}
```

Исходя из такого ограничения, старайтесь всегда объявлять свойство `grid` перед остальными свойствами настройки сетки. Например, стилевое правило, обеспечивающее максимально плотную упаковку элементов в ячейках сетки, должно представляться таким синтаксисом.

```
#layout {display: grid;
    grid: 3em 1fr 5em / 2em 3fr minmax(10em, 1fr) 2em;
    grid-auto-flow: dense column;}
```

Теперь следует правильно добавить несколько именованных горизонтальных линий в сетку, объявленную в начале раздела и содержащую именованные области. Чтобы разместить новую линию перед существующим рядом сетки, ее код нужно вводить перед строкой объявления этого ряда. Если же новую линию нужно разместить после одного из рядов сетки, то ее код вставляется после объявления такого ряда. В следующем стилевом правиле в уже существующий шаблон сетки, включающей именованные области, добавляются горизонтальные линии `main-start` и `main-stop`, размещаемые соответственно над и под средним рядом сетки, а также линия `page-end`, совмещаемая с ее нижним краем.

```
grid:
    "header header header header" 3em
    [main-start] ". content sidebar ." 1fr [main-stop]
    "footer footer footer footer" 5em [page-end] /
    2em 3fr minmax(10em, 1fr) 2em;
```

Выполнение правила приводит к образованию сетки (рис. 13.54), содержащей как неявно заданные именованные линии (`footer-start`, `footer-start` и т.п.), так и линии, определенные в явном виде.

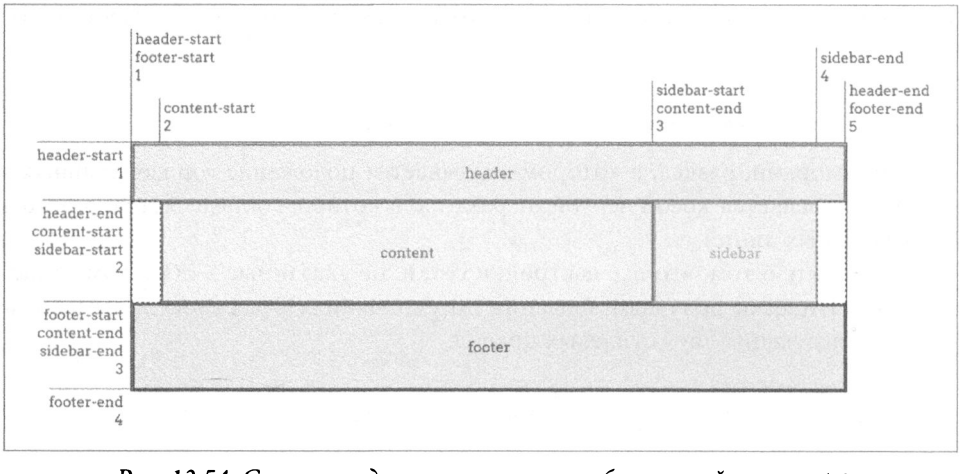


Рис. 13.54. Сетка, созданная с помощью общего свойства `grid`

Легко заметить, что сетка загромождается именованными линиями очень быстро. Свойство `grid` обладает чрезвычайно функциональным синтаксисом и при должной практике оказывается очень простым в использовании. С другой стороны, допустить ошибку в столь сложном синтаксисе очень легко, а ее цена очень велика: пользовательский агент проигнорирует сразу все объявления, и сетка образована не будет.

Теперь перейдем к описанию возможностей свойства `grid` по замещению индивидуальных свойств `grid-auto-flow`, `grid-auto-rows` и `grid-auto-columns`. Согласно определению, приведенному в начале этого раздела, эквивалентными можно считать такие два фрагмента CSS-кода.

```
#layout {grid-auto-flow: dense rows;
   grid-auto-rows: 2em;
   grid-auto-columns: minmax(1em, 3em);}

#layout {grid: dense rows 2em / minmax(1em, 3em);}
```

Как видите, правило общего назначения позволяет избежать ввода большого объема кода при сходном конечном результате. Опять-таки, выполнение последнего правила приводит к сбрасыванию в значения по умолчанию свойств, отвечающих за позиционирование в сетке колонок и рядов. Таким образом, полностью равнозначными можно считать следующие два стилевых правила.

```
#layout {grid: dense rows 2em / minmax(1em, 3em);}

#layout {grid: dense rows 2em / minmax(1em, 3em);
   grid-template-rows: auto;
   grid-template-columns: auto;}
```

Снова повторюсь: помните о том, что все не определенные в свойстве `grid` настройки автоматически сбрасываются в значения по умолчанию.

Подсетка

Кроме описанных выше значений, общее свойство `grid` также может иметь ключевое слово `subgrid`. Обычно оно применяется в конструкциях, подобных представленной ниже.

```
#grid {display: grid;
   grid: repeat(auto-fill, 2em) / repeat(10, 1% 8% 1%);}
.module {display: grid;
   grid: subgrid;}
```

Согласно такому коду, все элементы, вложенные в элемент `module`, размещаются на сетке, обозначенной стилевым правилом `#grid`.

Такая возможность может пригодиться во многих ситуациях, например при создании элемента `module`, занимающего сразу три колонки родительского контейнера и содержащего при этом дочерние элементы, которые нужно выровнять по собственной сетке (рис. 13.55).

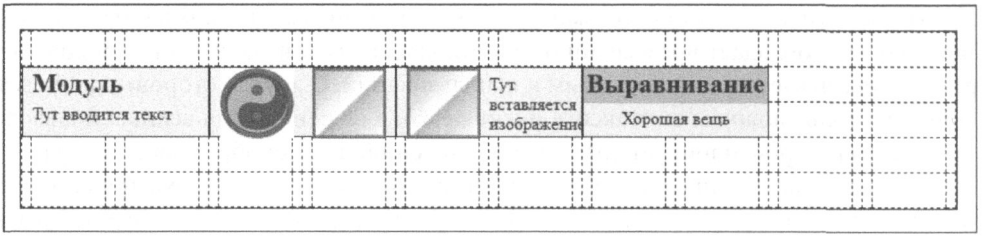


Рис. 13.55. Выравнивание элементов подсетки

Основная сложность использования подсеток заключается в большой вероятности прекращения поддержки ключевого слова `subgrid` в будущих версиях модуля Grid Layout. Именно поэтому в данной главе их описание приводится только в ознакомительных целях.

Интервалы

Во всех рассмотренных ранее примерах grid-элементы размещались на сетке вплотную друг к другу без образования свободного пространства между ними. Но такое их поведение не является обязательным и может быть легко перенастроено с помощью соответствующих стилевых свойств. В этом разделе мы поговорим о способах настройки интервалов между колонками и рядами сетки.

Интервалы между рядами и колонками

Под *интервалом*, или *промежутком*, подразумевается свободное пространство между полосами сетки. Их можно представлять так, как если бы линии, отделяющие полосы друг от друга, имели конечную толщину. Во многом они подобны промежуткам, создаваемым с помощью свойства `border-spacing` в таблицах, — как образуемым пространством, так и независимостью установки отдельно для каждого направления. В сетках интервалы между рядами задаются свойством `grid-row-gap`, а между колонками — свойством `grid-column-gap`.

grid-row-gap, grid-column-gap	
Значение	*<length>* | *<percentage>*
Начальное значение	0
Применяется	Grid-контейнеры
Вычисляется	Абсолютное значение в единицах длины
Наследуется	Нет
Анимируется	Да

Синтаксис каждого свойства предполагает передачу ему только значений, выраженных в единицах длины. При этом обрабатываются лишь положительные числовые величины. К недопустимым относятся процентные (`%`) и дольные (`fr`), а также

любые вычисляемые значения функции `minmax()`. Например, для разделения колонок промежутком шириной `1em` в стилевое правило нужно включить объявление `grid-column-gap: 1em`. Оно указывает интервал между колонками предельно точно: колонки сетки отделяются друг от друга свободным пространством строго заданной ширины, как показано на рис. 13.56.

```
#grid {display: grid;
   grid-template-rows: 5em 5em;
   grid-template-columns: 15% 1fr 1fr;
   grid-column-gap: 1em;}
```

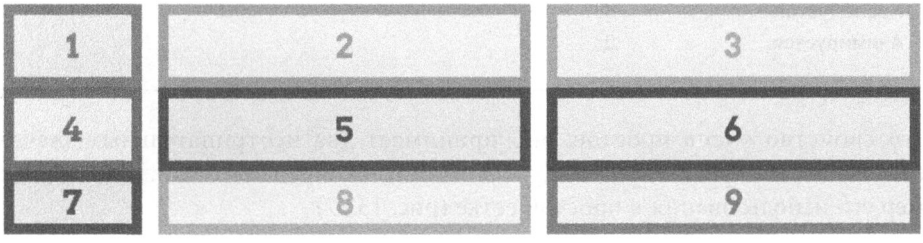


Рис. 13.56. Интервалы между колонками

С точки зрения grid-контейнера интервалы неотделимы от полос сетки, для которых они объявляются. В частности, следующее стилевое правило устанавливает высоту рядов, размер которых выражается в единицах `fr`, равной 140 пикселям.

```
#grid {display: grid; height: 500px;
   grid-template-rows: 100px 1fr 1fr 75px;
   grid-row-gap: 15px;}
```

Чтобы понять, почему два средних ряда имеют высоту 140 пикселей, нужно провести точные вычисления. Исходно сетка объявляется с высотой 500 пикселей. Из нее нужно вычесть высоту двух крайних рядов (100 и 75 пикселей), а также суммарную высоту трех междурядных интервалов (3×15 пикселей). Для определения вычисляемой высоты каждого из двух рядов, размер которых выражается в единицах `fr`, полученное значение (280 пикселей) нужно разделить на 2. При увеличении междурядного промежутка до значения `25px` общая высота средних рядов будет составлять 250 пикселей, поэтому каждый из них будет иметь вычисляемый размер `125px`.

Далеко не всегда конечный размер колонок и рядов вычисляется настолько просто, как в приведенном выше примере, в котором большинство размерных величин выражено в пикселях. В реальных проектах размеры горизонтальных и вертикальных полос сетки допускается устанавливать в любых других единицах измерения длины, включая величины, представленные функцией `minmax()`, что существенно расширяет возможности по верстке элементов документа по сетке.

Размер интервалов между колонками и рядами сетки изменяется под воздействием свойств `align-content` и `justify-content`, о чем будет говориться далее.

Как и следовало ожидать, в CSS интервалы между вертикальными и горизонтальными полосами сетки можно устанавливать одновременно с помощью одного-единственного свойства.

grid-gap

Значение	*<grid-row-gap> <grid-column-gap>*
Начальное значение	0 0
Применяется	Grid-контейнеры
Вычисляется	Согласно определению
Наследуется	Нет
Анимируется	Да

Это свойство очень простое: оно принимает два неотрицательных значения, определяющих интервалы как между колонками, так и рядами сетки. Ниже приведен пример его использования в простой сетке (рис. 13.57).

```
#grid {display: grid;
   grid-template-rows: 5em 5em;
   grid-template-columns: 15% 1fr 1fr;
   grid-gap: 12px 2em;}
```

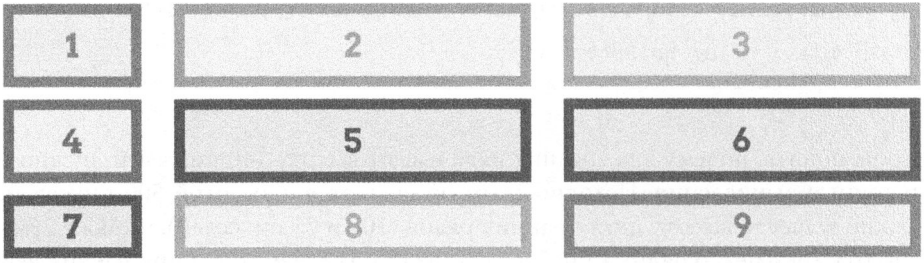


Рис. 13.57. Интервалы между колонками и рядами, определенные с помощью всего одного свойства

Grid-элементы и блочная модель

Теперь, когда известно, как создается сетка, по которой верстаются grid-элементы, и настраиваются интервалы между ее колонками и рядами, настало время ответить на вопрос: каким образом будет форматироваться привязанный к сетке элемент, если к нему добавить отступы или сделать его абсолютно позиционируемым? Насколько серьезно это повлияет на его расположение на сетке?

Сначала рассмотрим, как верстаются привязанные к сетке элементы, снабженные отступами. Основное правило гласит, что такой элемент будет привязываться к линиям сетки внешними краями отступов. Таким образом, положительные отступы позволяют втягивать содержимое элемента внутрь занимаемой им области сетки, а отрицательные отступы — "выталкивать" его наружу. Результат верстки элементов, снабженных отступами, по сетке показан на рис. 13.58.

```
#grid {display: grid;
   grid-template-rows: repeat(2, 100px);
   grid-template-columns: repeat(2, 200px);}
.box02 {margin: 25px;}
.box03 {margin: -25px 0;}
```

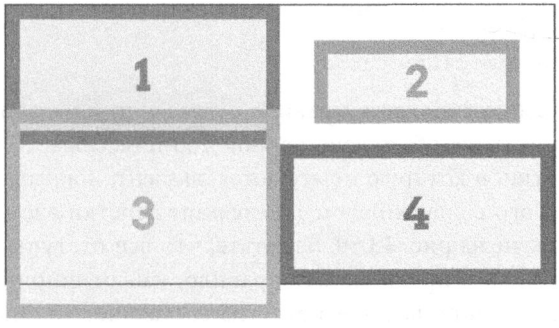


Рис. 13.58. Grid-элементы, снабженные отступами

Корректность размещения элементов, рассматриваемых в приведенном выше примере, на сетке во много определяется установкой свойств `width` и `height` в значение `auto`, позволяющее автоматически подгонять их под размер сетки. При передаче свойствам `width` и `height` фиксированных значений обработчику может потребоваться заменить их, чтобы обеспечить целостность grid-контейнера. Ситуация сходна с возникающей при установке в блочной модели элемента избыточных размерных величин — в подобных случаях коррекции подлежит ширина одного из отступов.

Рассмотрим, как будет позиционироваться элемент, форматируемый с помощью следующего правила, при размещении на сетке шириной 200 пикселей и высотой 100 пикселей.

```
.exel {width: 150px; height: 100px;
   padding: 0; border: 0;
   margin: 10px;}
```

При добавлении на сетку элемент будет иметь ширину (устанавливаемую свойством `width`) 150 пикселей, а каждый из четырех его отступов — ширину 10 пикселей. Следовательно, общая ширина элемента составит `170px`. При последующем изменении ширины одного из полей, например правого, до значения `40px` общая ширина элемента увеличится до 200 пикселей, поскольку его левое поле (`10px`) и область содержимого (`150px`) останутся неизменными.

Уменьшение вертикального отступа, например нижнего, до отрицательного значения `-10px` расширяет область содержимого элемента до высоты 110 пикселей при высоте отведенной для него области сетки всего `100px`.

При вычислении ширины полос, в которые помещаются grid-элементы, их отступы не учитываются. Таким образом, их величина не влияет на ширину колонок, определяемую свойством `min-content`. Они также остаются прежними при изменении размера сетки, положение линии которой устанавливается с помощью значений, выраженных в единицах `fr`.

Как и в случае регулярных элементов, корректировка размера grid-элементов осуществляется за счет отступов, размер которых определяется ключевым словом `auto`. В частности, автоматическая установка левого отступа grid-элемента выравнивает элемент по правому краю области сетки.

```
.exel {width: 150px; height: 100px;
   padding: 0; border: 0;
   margin: 10px; margin-left: auto;}
```

В приведенном выше стилевом правиле сумма ширины области содержимого элемента и правого отступа объявлена равной 160 пикселям. Разница между ней и шириной области сетки, в которую помещается элемент, покрывается за счет левого отступа, установленного в значение `auto`. Результат верстки элемента по обозначенной выше сетке показан на рис. 13.59. Заметьте, что все отступы элемента `excel`, за исключением левого, как и предполагает правило, имеют ширину 10 пикселей. Несложно подсчитать, что ширина левого отступа составляет 40 пикселей.

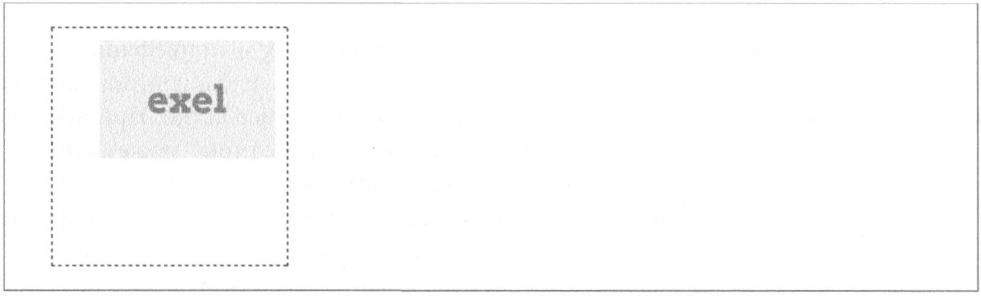


Рис. 13.59. Выравнивание элемента за счет автоматической настройки отступов

Еще более привычной по блочной модели элемента будет возможность выравнивания grid-элемента по ширине области сетки при автоматической настройке и правого, и левого отступов (при фиксированном значении свойства `width`). Тем не менее при верстке по сетке такой же подход можно применять для выравнивания grid-элемента посередине области сетки в вертикальном направлении — достаточно представить ключевым словом `auto` высоту верхнего и нижнего отступов. На рис. 13.60 приведены примеры выравнивания grid-элемента с явно заданными размерами в пределах одной и той же области сетки при автоматической настройке самых разных отступов.

```
.i01 {margin: 10px;}
.i02 {margin: 10px; margin-left: auto;}
.i03 {margin: auto 10px auto auto;}
.i04 {margin: auto;}
.i05 {margin: auto auto 0 0;}
.i06 {margin: 0 auto;}
```

Выравнивание grid-элементов с помощью свойства `justify-self` не требует передачи фиксированных значений его свойствам `width` и `height` (подробнее об этом — в следующей главе).

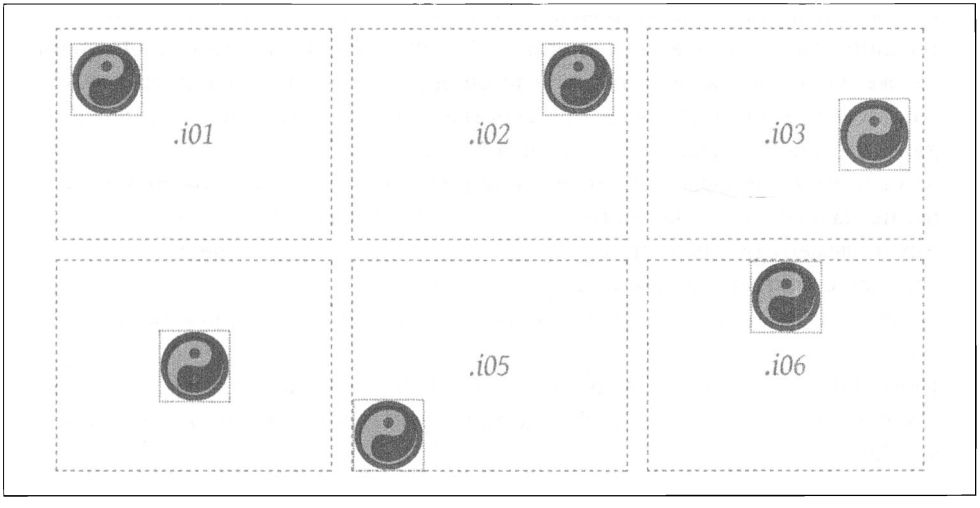


Рис. 13.60. Выравнивание, вызванное автоматической настройкой отступов

Автоматическая подгонка размера отступов выполняется так же, как и в случае форматирования абсолютно позиционируемых элементов. Такое поведение grid-элементов наталкивает на следующий, вполне обоснованный вопрос: существует ли возможность их абсолютного позиционирования на сетке? Например, при объявлении такого стилевого правила.

```
.exel {grid-row: 2 / 4; grid-column: 2 / 5;
   position: absolute;
   top: 1em; bottom: 15%;
   left: 35px; right: 1rem;}
```

Ответ очень прост: области сетки, для которой объявлены начальные и конечные линии, выступают и содержащим блоком, и контекстом позиционирования, относительно которого устанавливается положение grid-элемента. Таким образом, значения свойств смещения (`top`, `right` и др.) вычисляются относительно краев области сетки. Следовательно, представленный выше код обеспечивает форматирование, показанное на рис. 13.61.

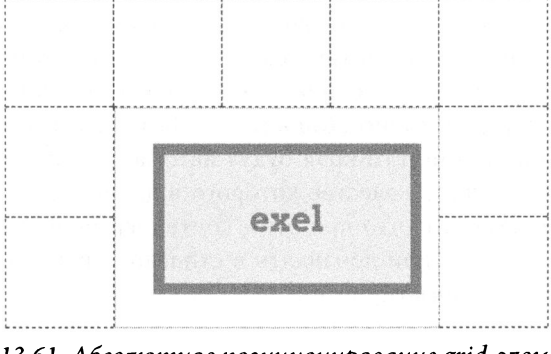


Рис. 13.61. Абсолютное позиционирование grid-элемента

При абсолютном позиционировании grid-элементов относительно контекста форматирования вычисляются значения всех без исключения настроек, влияющих на его положение, включая свойства определения размеров, ширины отступов, смещения и т.п. Как указывалось выше, при верстке по сетке контекстом форматирования для grid-элемента выступает объявленная для него область.

К особенности абсолютного позиционирования grid-элементов можно отнести изменение назначения ключевого слова `auto`. Если снабдить абсолютно позиционируемый элемент объявлением `grid-column-end: auto`, то конечная линия будет совмещаться с внешним краем поля grid-контейнера. Это утверждение справедливо даже в случаях, когда явно заданная сетка меньше по размеру, чем вмещающий ее grid-контейнер.

Чтобы увидеть, к чему приводит такое поведение, немного изменим предыдущий пример, представив его следующим кодом, результат выполнения которого показан на рис. 13.62.

```
.exel {grid-row: 1 / auto; grid-column: 2 / auto;
   position: absolute;
   top: 1em; bottom: 15%;
   left: 35px; right: 1rem;}
```

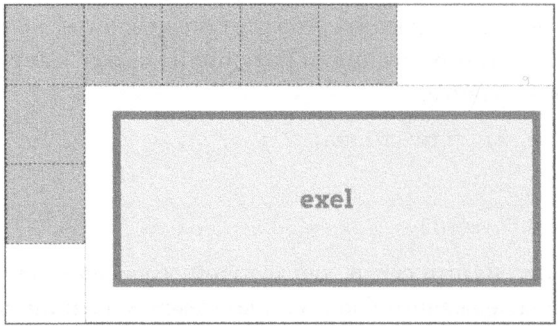


Рис. 13.62. Обработка значения `auto` при автоматическом позиционировании элемента

В данном примере контекст форматирования начинается у левого и верхнего края grid-контейнера, а его окончание совпадает с правым краем контейнера, даже несмотря на то что правый край сетки располагается намного ближе к его началу.

Описанной особенности обработки значения `auto` можно найти полезное применение. Чтобы обеспечить расширение абсолютно позиционируемого элемента до краев grid-контейнера, достаточно объявить его без указания точного положения начальной и конечной линий. Его края будут автоматически совмещены с краями контекста форматирования, в качестве которого в данном случае выступает родительский контейнер. Заметьте, что при этом контейнер не нужно снабжать объявлением `position: relative` или добавлять в стилевое правило любой другой код, определяющий контекст форматирования.

Не забывайте, что при абсолютном позиционировании размер элемента не влияет на положение линий и полос сетки. После объявления макета сетки положение grid-элемента определяется относительно линий, которые обозначают ее контекст форматирования.

К концу 2017 года ни один из популярных браузеров не поддерживал абсолютное позиционирование grid-элементов. Единственный доступный способ его имитации заключался в абсолютном позиционировании дочернего элемента в относительно позиционированном grid-элементе, представляющем отдельную область сетки. Именно таким образом выполнялось абсолютное позиционирование элементов в рисунках данного раздела. К сожалению, абсолютное позиционирование элементов с помощью значения `auto`, предусмотренное в CSS, тоже оказалось вне поддержки браузеров.

Выравнивание grid-элементов

После знакомства с flex-контейнерами вы будете предельно осторожно использовать свойства выравнивания в любых элементах. Все замечания, касающиеся выравнивания flex-элементов с помощью специальных свойств, в полной мере справедливы и по отношению к grid-элементам.

Перед рассмотрением принципов выравнивания элементов, помещенных в grid-контейнеры, вспомним, какие свойства могут применяться для этого (табл. 13.1).

Таблица 13.1. Свойства выравнивания элементов

Свойство	Направление выравнивания	Область назначения
`justify-self`	Горизонтально (в строчном направлении)	Grid-элемент
`justify-items`	Горизонтально (в строчном направлении) для всех grid-элементов	Grid-контейнер
`justify-content`	Горизонтально (в строчном направлении) для всей сетки	Grid-контейнер
`align-self`	Вертикально (в блочном направлении)	Grid-элемент
`align-items`	Вертикально (в блочном направлении) для всех grid-элементов	Grid-контейнер
`align-content`	Вертикально (в блочном направлении) для всей сетки	Grid-контейнер

Свойства группы `justify-*`, представленные в табл. 13.1, отвечают за выравнивание элементов вдоль строчного направления (по горизонтали в большинстве европейских языков). Первое свойство позволяет выравнивать только отдельный элемент, а два последующих применяются для горизонтального выравнивания группы элементов или всей сетки. Подобным образом стилевые свойства группы `align-*` применяются для выравнивания элементов в блочном направлении (по вертикали в большинстве европейских языков).

Выравнивание отдельных элементов

Начать знакомство со способами выравнивания grid-элементов проще всего на примере стилевых свойств группы `*-self`. Результат выравнивания элементов в горизонтальном и вертикальном направлениях при передаче свойствам `justify-self` и `align-self` разных значений показан на рис. 13.63.

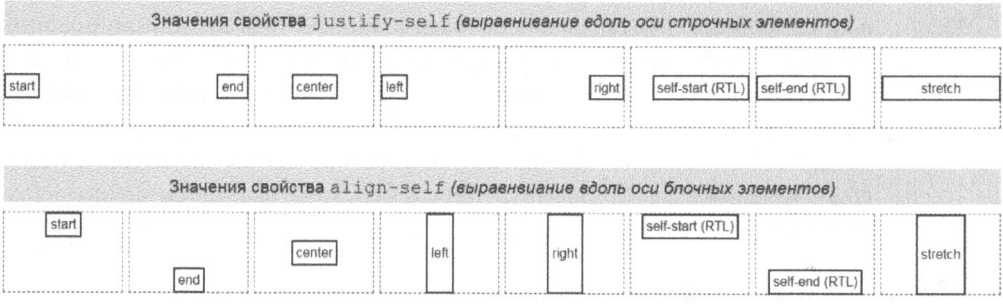


Рис. 13.63. Выравнивание элементов в строчном и блочном направлениях

Для большей наглядности содержимое каждого элемента представлено ключевым словом, указывающим способ его выравнивания в пределах своей области, обозначенной пунктирной линией. Направление выравнивания указывается в заголовке каждой сетки.

В случаях, когда ширина и высота элемента не определяются в явном виде с помощью свойств `width` и `height`, он заполняет всю отведенную для него область сетки (поведение по умолчанию), вместо того чтобы сжиматься или расширяться до размеров содержимого.

Значения `start` и `end`, как и предполагает их название, обязывают grid-элемент выравниваться соответственно по начальной или конечной линиям своей области. Таким же образом значение `center` центрирует элемент в отведенной для него области сетки в направлении, определенном свойством. Последняя операция успешно выполняется в отсутствие у элемента отступов или назначения ему фиксированных размеров (свойства `width` и `height`).

Горизонтальное выравнивание элементов с помощью ключевых слов `left` и `right` выполняется в точности так, как предписывают их названия. При вертикальном выравнивании (свойство `align-self`) оба значения обрабатываются подобно ключевому слову `start`.

Наиболее интересные способы выравнивания наблюдаются при передаче соответствующим свойствам ключевых слов `self-start` и `self-end`. Значение `self-start` выравнивает grid-элемент по тому краю области сетки, с которого он начинает заполняться содержимым. На рис. 13.63 показано, как с их помощью выравнивается элемент, направление которого определяется объявлением `direction: rtl`. В системах письма справа налево элементы заполняются содержимым, начиная с правого и заканчивая левым краем. Следовательно, grid-элемент первой сетки, выравниваемый с помощью ключевого слова `self-start`, привязывается к правому краю своей

области, а элемент, выравнивание которого обеспечивается значением self-end, смещается к ее левому краю. Результат применения этих же ключевых слов для вертикального выравнивания элементов с таким же порядком заполнения содержимым продемонстрирован во втором примере.

Самый сложный для описания способ выравнивания обеспечивается значением stretch. Чтобы понять его назначение, нужно вспомнить, что по умолчанию элементы, не имеющие фиксированного размера, расширяются или сжимаются до размеров своего содержимого. В свою очередь значение stretch обязывает их расширяться до ширины или высоты занимаемой области сетки в зависимости от направления выравнивания. Объявление align-self: stretch предопределяет расширение элемента в вертикальном направлении, а объявление justify-self: stretch — в горизонтальном. Однако такое поведение свойственно только элементам, размер которых определяется ключевым словом auto. Исходя из этого, расширение элемента до размеров области обеспечивает только первое из следующих двух правил.

```
.exel01 {align-self: stretch; height: auto;}
.exel02 {align-self: stretch; height: 50%;}
```

Как видите, во втором примере размер элемента определяется значением, отличным от auto (устанавливается по умолчанию), а ключевое слово stretch не оказывает на него должного влияния. Приведенные выше рассуждения в полной мере справедливы для свойств justify-self и width.

Выравнивание grid-элементов на сетке выполняется с помощью еще двух значений, эффект применения которых настолько значителен, что стоит отдельного внимания. Они позволяют выравнивать первую или последнюю базовую линию элемента по самой высоко или низко расположенной базовой линии колонки или ряда. Предположим, перед нами стоит задача выровнять grid-элемент так, чтобы его последняя базовая линия совмещалась с последней базовой линией самого высокого элемента текущего ряда. Она выполняется по такому правилу:

```
.exel {align-self: last-baseline;}
```

Подобным образом следующее стилевое правило позволяет расположить первую базовую линию grid-элемента на одном уровне с самой низко расположенной первой базовой линией этого же ряда:

```
.exel {align-self: baseline;}
```

В отсутствие обозначаемой ключевым словом базовой линии у текущего или сразу всех grid-элементов ряда (колонки) выравнивание будет осуществляться так же, как и в случаях применения ключевых слов start (baseline) или end (last-baseline).

Значения flex-start и flex-end намеренно не рассматриваются в этом разделе. Они применяются для выравнивания только flex-элементов, а в остальных способах верстки, в том числе и верстке по сетке, полностью эквивалентны ключевым словам start и end.

Выравнивание всех элементов

Настало время изучить стилевые свойства `align-items` и `justify-items`. Они имеют такие же значения, как и свойства, описанные в предыдущем разделе, но при этом воздействуют не на отдельный элемент, а сразу на все элементы текущего grid-контейнера, а также на все его дочерние контейнеры.

Эти свойства определяют единый способ выравнивания для всех элементов контейнера в одном объявлении. В примере, приведенном на рис. 13.64, все элементы центрируются по соответствующим областям сетки.

```
#grid {display: grid;
   align-items: center; justify-items: center;}
```

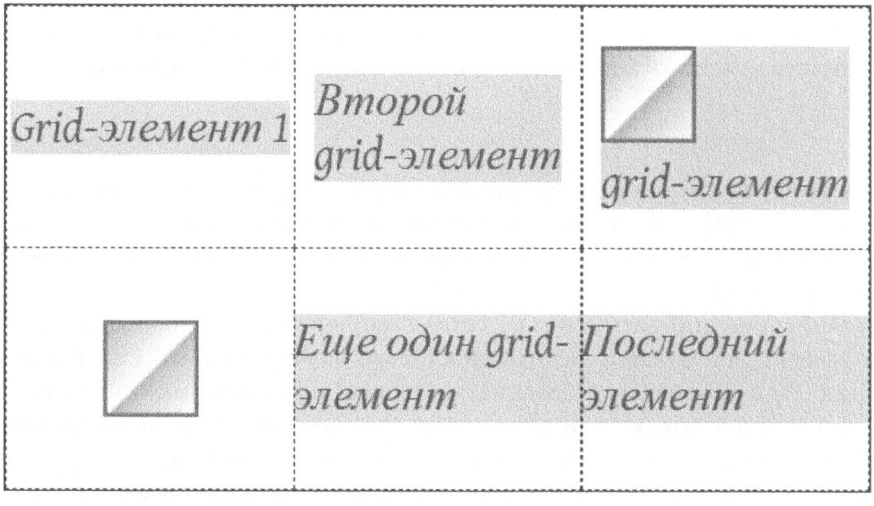


Рис. 13.64. Центрирование сразу всех элементов сетки

Легко заметить, что центрирование элементов в рамках областей сетки осуществляется как в горизонтальном, так и вертикальном направлении. Кроме того, все элементы с неявно заданными размерами вполне ожидаемо расширяются до размеров своих областей. Выравнивание по центру области выполняется только для элементов с фиксированной шириной и высотой.

Наряду с выравниванием grid-элементы можно распределять по сетке и даже выровнять или распределить саму сетку. Для этих целей в CSS-модуле Grid Layout применяются свойства `align-content` и `justify-content`. У указанных свойств совсем мало допустимых значений. На рис. 13.65 показан эффект применения всех значений свойства `justify-content` к одной и той же сетке элементов, форматирование которых определяется следующим стилевым правилом.

```
.grid {display: grid; padding: 0.5em; margin: 0.5em 1em; width: auto;
   grid-gap: 0.75em 0.5em; border: 1px solid;
   grid-template-rows: 4em;
   grid-template-columns: repeat(5, 6em);}
```

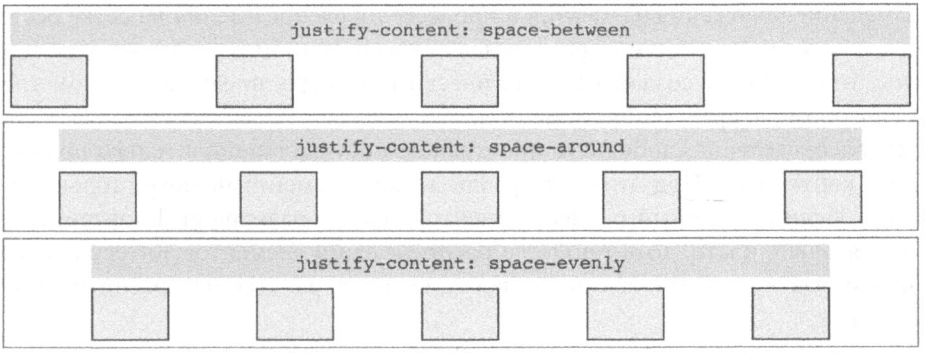


Рис. 13.65. Горизонтальное распределение элементов

Описанный выше подход справедлив как для рядов, так и для колонок сетки. На рис. 13.66 показано, как будут верстаться элементы при замене свойства `justify-content` свойством `align-content`. В этом случае форматирование сетки элементов осуществляется с помощью такого правила.

```
.grid {display: grid; padding: 0.5em;
   grid-gap: 0.75em 0.5em; border: 1px solid;
   grid-template-rows: repeat(4, 3em);
   grid-template-columns: 5em;}
```

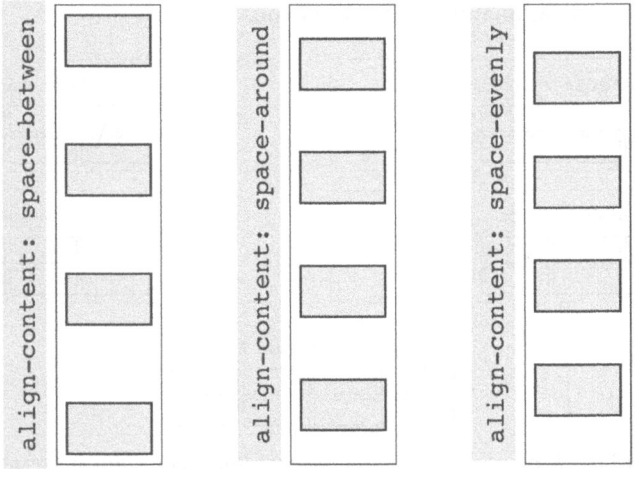


Рис. 13.66. Вертикальное распределение элементов

Позиционирование элементов в выбранном направлении осуществляется за счет распределения свободного пространства колонки или ряда, оставшегося незанятым после заполнения его элементами с исходным форматированием, — в том числе через специально заданные интервалы. Способ распределения устанавливается с помощью значений свойств `justify-content` (в горизонтальном направлении) и `align-content` (в вертикальном направлении).

Изменение положения элементов в процессе их распределения на сетке осуществляется за счет изменения интервалов между ними. Если интервалы исходно не объявлены, то они будут созданы автоматически в размере, предусмотренном требуемым способом распределения элементов.

Перераспределение свободного пространства осуществляется только при его наличии в контейнере. При этом интервалы между элементами могут только увеличиваться. Если суммарный размер элементов больше размера grid-контейнера, что случается очень часто, то в процессе распределения элементов интервалы между ними не образуются (отрицательное свободное пространство перераспределению не подлежит).

На момент завершения работы над книгой спецификация предполагала распределение grid-элементов с помощью еще одного, относительно нового значения. При передаче описываемым выше свойствам значения `stretch` свободное пространство должно перераспределяться между самими элементами, а не интервалами между ними. Таким образом, при наличии в контейнере, содержащем 8 элементов, свободного пространства шириной 400 пикселей размер каждого из них увеличится на 50 пикселей. При этом ширина увеличивается на одинаковую величину, а не пропорционально исходному размеру. К сожалению, к концу 2017 года ни один из браузеров новую возможность не поддерживал.

В следующем примере, в отличие от предыдущего, элементы не распределяются, а всего лишь выравниваются на сетке контейнера. На рис. 13.67 показаны все способы горизонтального выравнивания grid-элементов.

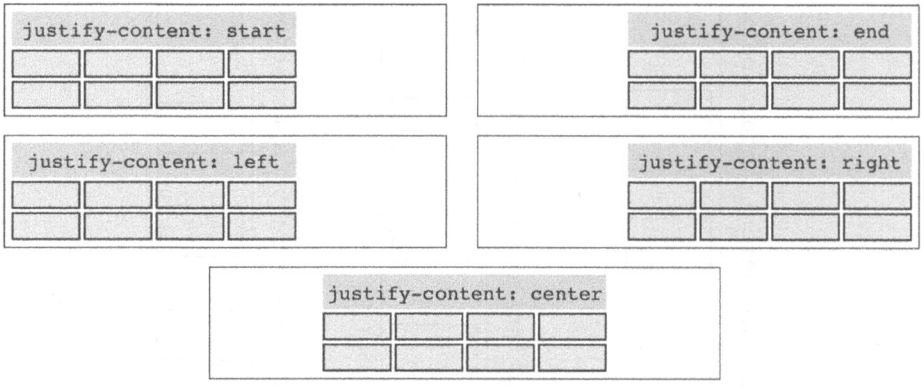


Рис. 13.67. Горизонтальное выравнивание всей сетки элементов

В данных примерах полосы элементов, выравниваемых с помощью свойства `justify-content`, обрабатываются как единое целое, никоим образом не изменяя выравнивание, заданное отдельно для каждого из элементов. В частности, вся сетка может выравниваться с помощью объявления `justify-content: end`, в то время как отдельные ее элементы могут привязываться к левому, правому краям своих областей или располагаться в их центре.

Учитывая возможности свойства `justify-content` по горизонтальному выравниванию всей сетки, вполне справедливо считать, что за ее вертикальное выравнивание

отвечает стилевое свойство `align-content`. Примеры такого его использования приведены на рис. 13.68.

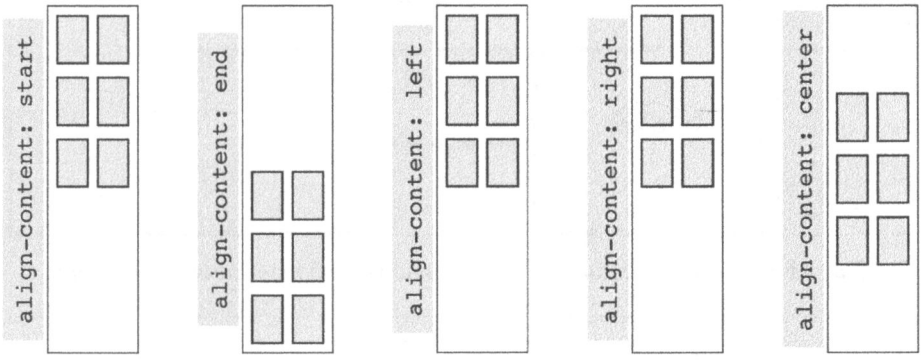


Рис. 13.68. Вертикальное выравнивание всей сетки элементов

При вертикальном выравнивании значения `left` и `right` становятся бессмысленными, а потому обрабатываются так же, как и ключевое слово `start`.

Размещение на слоях и порядок наложения

Как вам известно из предыдущих разделов, grid-элементы могут накладываться друг на друга. Обычно это происходит при снабжении их отрицательными отступами, что приводит к смещению в соседние области сетки. В результате одна и та же ячейка сетки оказывается одновременно занятой двумя или даже большим количеством элементов. По умолчанию grid-элементы накладываются друг на друга в порядке добавления в документ: чем позже элемент объявляется, тем на более высокий уровень стека он помещается. Результат такого поведения представлен на рис. 13.69. (Предполагается, что уровень расположения элемента в стеке указывается номером в имени класса.)

```
#grid {display: grid; width: 80%; height: 20em;
   grid-rows: repeat(10, 1fr); grid-columns: repeat(10, 1fr);}
.box01 {grid-row: 1 / span 4; grid-column: 1 / span 4;}
.box02 {grid-row: 4 / span 4; grid-column: 4 / span 4;}
.box03 {grid-row: 7 / span 4; grid-column: 7 / span 4;}
.box04 {grid-row: 4 / span 7; grid-column: 3 / span 2;}
.box05 {grid-row: 2 / span 3; grid-column: 4 / span 5;}
```

Для образования собственного порядка наложения элементов используется свойство `z-index`. Как и при позиционировании регулярных элементов, уровень наложения grid-элементов устанавливается вдоль оси Z, направленной перпендикулярно к поверхности экрана, на котором просматривается документ. Чем больше положительный индекс, тем ближе элемент к наблюдателю, и наоборот, большие отрицательные значения максимально удаляют элемент от глаз пользователей. В примере, показанном на рис. 13.70, расположение второго элемента на самом верхнем уровне стека предполагает объявление для него стилевого свойства `z-index` с наибольшим индексным значением.

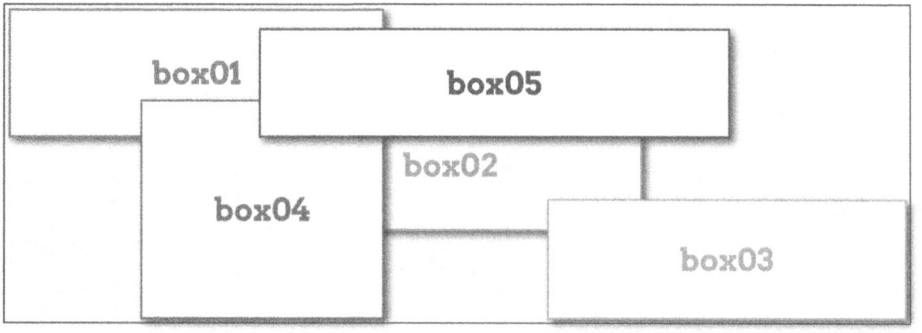


Рис. 13.69. Перекрывание элементов в порядке добавления в документ

Рис. 13.70. Извлечение элемента на передний план

Порядок наложения grid-элементов также можно изменить с помощью свойства `order`. В контексте верстки элементов по сетке у него такое же назначение, как и в случае их форматирования в пределах колонки или ряда согласно переданному числовому значению. Свойство `order` определяет не только уровень "залегания" элемента в стеке, но и *порядок визуализации*. Например, если в предыдущем примере свойство `z-index` заменить свойством `order`, то будет получен в точности такой же результат:

```
.box02 {order: 10;}
```

В данном случае элемент `box02` оказывается поверх остальных элементов благодаря тому, что имеет наибольшее значение свойства `order` из всех элементов, имеющихся в стеке. Следовательно, он визуализируется в последнюю очередь. Представив порядок визуализации элементов в виде последовательности, заполняемой слева направо, можно обнаружить, что второй элемент находится в самом ее конце (рис. 13.71).

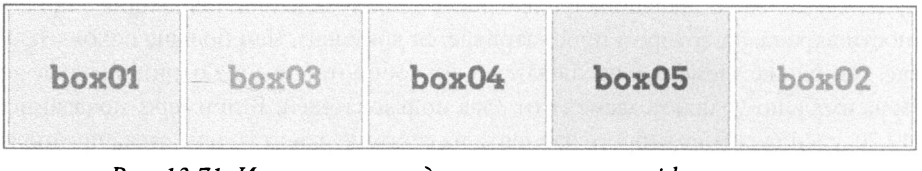


Рис. 13.71. Изменение порядка визуализации grid-элементов

Не забывайте, что возможность изменения порядка наложения элементов не предопределяет необходимость ее использования. Вот что по этому поводу говорится в спецификации к модулю Grid Layout (`https://www.w3.org/TR/css-grid-1/#order-property`):

> "Что касается порядка наложения flex-элементов, то свойство `order` должно применяться только для рассинхронизации его с порядком представления или навигации по документу; во всех остальных случаях все необходимые изменения должны вноситься в исходный документ.

Следовательно, единственная причина, побуждающая к изменению порядка визуализации элементов, — это необходимость отображения их в порядке, отличном от задаваемого в исходном документе. Чтобы получить такую ситуацию, достаточно размещать элементы в области сетки grid-контейнера в порядке, отличном от заданного разметкой документа.

Не стоит считать, что свойство `order` бесполезно и его нужно всячески избегать. В отдельных случаях оно оказывается просто незаменимым. Просто старайтесь не использовать его для задач, имеющих альтернативные способы решения, до тех пор пока не столкнетесь с такой необходимостью.

Резюме

Верстка по сетке относится к сложным задачам веб-дизайна. Не стоит разочаровываться, если многое из рассмотренного выше кажется вам непонятным. Знакомство с неизвестными ранее возможностями по форматированию элементов, размещаемых на сетке, может занять много времени. Несомненно, их освоение требует намного больше усилий, чем обычно тратится на изучение обновленных инструментов, присущих предыдущим спецификациям. При первом знакомстве с функциями модуля Grid Layout я был основательно обескуражен функциональным богатством недоступных ранее новых возможностей, включение которых в спецификацию CSS стало едва ли не самым ожидаемым событием за более чем двадцатилетнюю историю ее развития.

Хочется верить, что описанные в этой главе инструменты и методики помогут вам избежать досадных ошибок при верстке документов по сетке. Но как бы там ни было, сохраняйте спокойствие и следуйте одному из советов небезызвестного мастера Йоды: забудьте о том, что уже знаете. Верстка элементов по сетке не требует использования хитроумных приемов, к которым приходилось прибегать для получения желаемого результата в традиционном способе верстки. Будьте усердны и терпеливы, и результат не замедлит себя ждать.

ГЛАВА 14

Верстка таблиц

Взглянув на название главы, вы, должно быть, воскликнули: "Как, табличная верстка?! Разве это не то, чего мы пытаемся избежать?" Конечно, использование таблиц для верстки документов — далеко не самое занимательное и эффективное занятие. Именно поэтому в данной главе речь пойдет только о средствах CSS, применяемых исключительно для форматирования самих таблиц, помещаемых на веб-страницы. В ней *не затрагиваются* вопросы табличной верстки документов.

Таблицы заметно отличаются от остального содержимого документа. До внедрения в CSS модулей Flexbox Layout и Grid Layout таблицы оставались единственным средством, позволяющим унифицировать размер группы элементов, например назначить одинаковую высоту всем ячейкам одного ряда независимо от размера помещаемого в них содержимого. Такой же подход позволяет унифицировать ширину колонок таблицы. Примыкающие ячейки имеют общую границу, и она всегда одинаковая, даже если исходно каждой из них назначены совершенно разные стили. Как будет показано далее, форматирование табличных данных регулируется огромным количеством взаимозаменяемых стилевых инструментов, бо́льшая часть которых досталась в наследство от предыдущих версий спецификации. Все они применимы для верстки только табличных данных.

Форматирование таблиц

Прежде чем приступать к изучению стилевых инструментов, предназначенных для форматирования таблиц, необходимо ознакомиться с принципами их создания, а также вставки в них всех необходимых элементов. Важное замечание: не стоит путать форматирование таблицы с ее верстkой. Как ни странно, но верстку таблицы можно выполнить только после ее форматирования.

Построение таблицы

Первое, с чем вам предстоит ознакомиться, — со структурой таблицы. Несмотря на простоту описания, структура таблицы в наибольшей степени определяет способы ее форматирования.

В CSS существует большое различие между *элементами таблицы* и *внутренними элементами таблицы*. Внутренний элемент таблицы представляется прямоугольным

контейнером, вмещающим содержимое, поля и границы элемента, но не его отступы. Из этого можно сделать важный вывод: ячейки таблицы *нельзя* разделять с помощью отступов. При визуализации таблицы CSS-совместимые браузеры игнорируют любые отступы, назначенные ее ячейкам, рядам, колонкам и любым другим внутренним элементам (исключение составляет заголовок таблицы, детально описанный в разделе "Заголовки").

Построение таблицы выполняется согласно шести базовым правилам. Основной элемент таблицы — это *ячейка*, обозначающая область, ограниченную линиями таблицы. На рис. 14.1 изображены две простые таблицы — их ячейки разграничены пунктирными линиями.

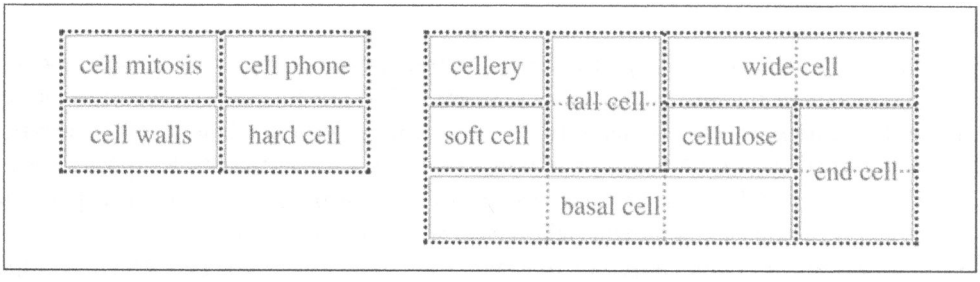


Рис. 14.1. Ячейки как основные элементы таблицы

Все ячейки первой таблицы формата 2×2 (слева) в точности соответствуют ячейкам сетки, по которой строится таблица. В более сложном варианте (справа) некоторые ячейки таблицы занимают сразу несколько ячеек сетки — края ячеек представлены прямыми линиями и в точности совмещаются с краями ячеек сетки.

Ячейки сетки являются только конструктивными элементами таблицы, они не включены в объектную модель документа (DOM), а потому не подлежат стилевому форматированию. Они всего лишь предопределяют возможности по форматированию таблиц, но не устанавливают его.

Правила построения таблицы

- *Контейнер ряда* таблицы содержит всего один ряд сетки. Контейнеры рядов заполняют таблицу сверху вниз в порядке объявления в исходном документе (за исключением рядов шапки и подвала таблицы). Таким образом, контейнеров рядов столько же, сколько и самих рядов (т.е. элементов `tr`).
- *Контейнер группы рядов* включает все элементы, относимые к содержащимся в нем рядам.
- *Контейнер колонки* содержит одну или несколько колонок ячеек сетки. Контейнеры колонок заполняют таблицу в порядке объявления. Первая колонка размещается у левого (в документах с системой письма слева направо) или у правого (в документах с системой письма справа налево) края таблицы.
- *Контейнер группы колонок* включает все элементы, относимые к содержащимся в нем колонкам.

- Несмотря на возможность размещения ячейки таблицы сразу в нескольких рядах или колонках сетки, спецификация CSS не определяет, каким образом это можно сделать. Направление заполнения ячейкой таблицы ячеек сетки традиционно определяется системой письма документа. Каждая ячейка таблицы увеличенного размера представляется прямоугольным контейнером, заполняющих одну или несколько ячеек сетки в горизонтальном и/или вертикальном направлении. Ячейка сетки, из которой берет начало ячейка таблицы увеличенного размера, находится в ее верхнем ряду. Прямоугольный контейнер такой ячейки должен располагаться в таблице как можно левее (в документах с системой письма слева направо), но не перекрывать контейнеры соседних ячеек. Кроме того, он должен находиться справа от всех предыдущих ячеек (согласно порядку объявления в исходном документе) одного с ним ряда. В документах с системой письма справа налево ячейка увеличенного размера должна располагаться в таблице как можно правее, не перекрывая при этом соседние ячейки, и в то же время находиться слева от всех ячеек, объявленных в исходном документе до нее.
- Контейнер ячейки таблицы не может расширяться за пределы контейнера последнего ряда или группы рядов таблицы. В случае возникновения подобных условий ячейка сжимается до размера, при котором она не выступает за края таблицы или группы ячеек, к которым она относится.

Спецификация CSS допускает, хотя и не приветствует, изменение положения ячеек и других внутренних элементов таблицы. В частности, перемещение ряда, включающего ячейки таблицы увеличенного размера, приводит к заметному искажению макета документа вследствие полного исключения их влияния на последующие ряды таблицы. Как бы там ни было, возможность изменения положения элементов таблицы поддерживается всеми браузерами.

По определению все ячейки таблицы должны иметь прямоугольную форму, но не обязательно одинаковый размер. При этом все ячейки одной колонки имеют одинаковую ширину, как и ячейки одного ряда — одинаковую высоту. Тем не менее ячейки, относящиеся к другим колонкам или рядам, могут иметь совершенно иную ширину или высоту.

После изучения основных правил построения таблиц у вас может возникнуть справедливый вопрос: как определяется принадлежность элементов к ячейкам таблицы?

Визуализация элементов таблицы и свойство `display`

В HTML элементы таблицы различаются очень просто: поддержка объявляющих их ключевых слов, например `tr` и `td`, включена в каждый браузер. Наряду с этим в XML не существует внутренних конструкций, ответственных за объявление элементов таблицы. В подобных случаях для отнесения элемента к таблице применяется свойство `display`, которое снабжено большим количеством значений, предназначенных для решения этой задачи.

display

Значение	[*<display-outside>* || *<display-inside>*] | *<display-listitem>* | *<display-internal>* | *<display-box>* | *<display-legacy>*
Определение	См. ниже
Начальное значение	`inline`
Применяется	Все элементы
Вычисляется	Согласно определению
Наследуется	Нет
Анимируется	Нет

<display-outside>

`block` | `inline` | `run-in`

<display-inside>

`flow` | `flow-root` | `table` | `flex` | `grid` | `ruby`

<display-listitem>

`list-item` && <display-outside>? && [`flow` | `flow-root`]?

<display-internal>

`table-row-group` | `table-header-group` | `table-footer-group` | `table-row` | `table-cell` | `table-column-group` | `table-column` | `table-caption` | `ruby-base` | `ruby-text` | `ruby-base-container` | `ruby-text-container`

<display-box>

`contents` | `none`

<display-legacy>

`inline-block` | `inline-list-item` | `inline-table` | `inline-flex` | `inline-grid`

Далее описываются только те ключевые слова, которые применяются для форматирования таблиц. Остальные значения свойства `display` детально рассматриваются в других главах.

`table`

Объявляет элемент таблицы блочного уровня. С помощью этого значения в документе создается прямоугольный контейнер блочного элемента, вмещающий таблицу. В HTML представляет одноименный элемент `table`.

`inline-table`

Определяет элемент таблицы строчного уровня. С помощью этого значения в документе создается прямоугольный контейнер строчного типа. В CSS ближайшее к нему значение представлено ключевым словом `inline-block`. В HTML представляет одноименный элемент `table`, несмотря на то что HTML-таблицы относятся исключительно к блочному типу данных.

`table-row`

Объявляет ряд ячеек таблицы. Представляет элемент `tr` в HTML.

`table-row-group`

Обозначает группу рядов, состоящих из ячеек таблицы. Представляет элемент `tbody` в HTML.

`table-header-group`

Во многом подобно значению `table-row-group`, за исключением визуального форматирования представленного им элемента: он всегда располагается перед другими рядами и группами рядов, объявленными в таблице, но после ее заголовка. При печати длинного документа браузер (в частности, Firefox) может повторять ряды шапки таблицы в верхней части каждой содержащей ее страницы. В спецификации не оговаривается, каким образом должна обрабатываться ситуация объявления свойства `display` со значением `table-header-group` сразу для нескольких элементов. В HTML представляет элемент `thead`.

`table-footer-group`

Во многом подобно значению `table-header-group`, за исключением того, что представленный им элемент всегда располагается после других рядов и групп рядов, объявленных в таблице, но перед ее нижним заголовком. При печати длинного документа браузер может повторять ряды подвала таблицы в нижней части каждой содержащей ее страницы. В спецификации не оговаривается, каким образом должна обрабатываться ситуация объявления свойства `display` со значением `table-footer-group` сразу для нескольких элементов. В HTML представляет элемент `tfoot`.

`table-column`

Назначает элемент в качестве колонки ячеек таблицы. В терминологии CSS элементы, свойство `display` которых установлено в значение `none`, не подлежат визуализации, что позволяет использовать ключевое слово `table-column` только для обозначения в таблице элементов колонок, не отображаемых в документе. В HTML представляет элемент `col`.

`table-column-group`

Объявляет в таблице элемент группы, состоящей из одной или нескольких колонок элементов. Как и предыдущее значение, представляет невизуализируемый элемент, который применяется исключительно для обозначения в таблице группы колонок. В HTML представляет элемент `colgroup`.

`table-cell`

Представляет элемент, включающий всего одну ячейку таблицы. В HTML представляет сразу два элемента: `th` и `td`.

`table-caption`

Объявляет заголовок таблицы. Спецификация CSS не оговаривает, каким образом будет обрабатываться ситуация назначения свойства `display` со значением

caption сразу нескольким элементам. Старайтесь всячески избегать многократного применения объявления display: caption в одной таблице.

Общее представление об эффектах применения этих ключевых слов можно получить, рассмотрев следующий пример форматирования документа, разметка которого задана в HTML 4.0, а полный код таблицы стилей приведен в приложении Г спецификации CSS 2.1.

```
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;}
```

Описанные выше значения наиболее востребованы в документах XML, которые лишены семантической конструкции, равнозначной представляемой свойством display в HTML. Рассмотрим такой пример.

```
<scores>
   <headers>
      <label>Team</label>
      <label>Score</label>
   </headers>
   <game sport="MLB" league="NL">
      <team>
         <name>Reds</name>
         <score>8</score>
      </team>
      <team>
         <name>Cubs</name>
         <score>5</score>
      </team>
   </game>
</scores>
```

Для представления этого фрагмента документа в табличном виде можно использовать такие стилевые правила.

```
scores {display: table;}
headers {display: table-header-group;}
game {display: table-row-group;}
team {display: table-row;}
label, name, score {display: table-cell;}
```

Различным ячейкам теперь можно назначать любое нужное форматирование, например выделять полужирным начертанием элементы label или выравнивать по правому краю элементы scores.

Превосходство рядов

В CSS таблицы представляются моделью, в которой ряды получают превосходство над колонками. Буквально это означает, что в явном виде в таблице объявляются только ряды, а колонки образуются исключительно в рамках заданной структуры рядов. Таким образом, первая колонка состоит из первых ячеек каждого ряда, вторая колонка — из вторых ячеек рядов и т.д.

В HTML превосходство рядов не создает особых трудностей, так как язык ориентирован на разметку строк таблиц. А вот в XML превосходство рядов проявляется сильнее, поскольку язык ограничивает возможности авторов страниц в плане задания табличной разметки. Из-за подобного характера табличной разметки в CSS не представляется возможным структурировать таблицы через форматирование колонок.

Колонки

Несмотря на превосходство рядов в табличной модели CSS, колонки таблицы являются неотъемлемой частью макета документа. Независимо от способа объявления в исходном документе структура таблицы предполагает отнесение ее ячеек сразу к обоим контекстам форматирования (ряды и колонки). Тем не менее в CSS регулярное форматирование колонок и групп колонок разрешается выполнять с помощью всего четырех стилевых свойств: `border`, `background`, `width` и `visibility`.

Более того, вышеупомянутые свойства применяются только в колоночном контексте форматирования.

`border`

Границы добавляются к колонкам и группам колонок только при установке свойства `border-collapse` в значение `collapse`.

`background`

Фон колонки или группы колонок отображается только при назначении прозрачного фона всем относящимся к ней ячейкам и рядам (см. раздел "Слои таблицы").

`width`

Определяет *минимальную* ширину колонки или группы колонок. Может автоматически увеличиваться при помещении в ячейки колонок содержимого большого размера.

`visibility`

При передаче этому свойству значения `hidden` ни одна из ячеек колонки или группы колонок не отображается в таблице. Ячейки таблицы, занимающие не только текущую, но и соседние колонки (группу колонок), обрезаются по ее краю, равно как и ячейки таблицы, распространяющиеся в скрываемую колонку из соседних колонок. Ширина таблицы уменьшается на величину, равную ширине скрываемой колонки. Объявление других значений свойства `visibility` никак не сказывается на визуализации колонок или группы колонок.

Анонимные объекты таблицы

При определенных обстоятельствах язык разметки оказывается несостоятельным в плане создания в таблицах всех необходимых элементов, подлежащих стилевому форматированию. К тому же браузеру часто приходится обрабатывать ситуации, когда авторы документов попросту забывают снабдить таблицы всеми необходимыми структурными элементами. В качестве примера рассмотрим следующий фрагмент HTML-документа.

```
<table>
   <td>Размер:</td>
   <td><input type="text"></td>
</table>
```

Быстрый анализ разметки показывает, что с ее помощью в документе создается однорядная таблица, состоящая всего из двух колонок, несмотря на то что структурный элемент, представляющий ряд (`tr`), в ней напрочь отсутствует.

Для обеспечения целостности все недостающие компоненты таблицы представляются в ней анонимными элементами. Чтобы понять, как эта задача решается в CSS, рассмотрим приведенный выше пример более детально. Встроенный в CSS механизм автоматического заполнения таблицы недостающими элементами обеспечивает объявление в уже имеющейся разметке еще одного элемента, дочернего по отношению к элементу `table` и родительского для ячеек таблицы. Вот как это выглядит для представленного выше кода HTML.

```
<table>
   <!-- начало анонимного объекта элемента ряда -->
      <td>Размер:</td>
      <td><input type="text"></td>
   <!-- окончание анонимного объекта элемента ряда -->
</table>
```

Полученная структура визуализируется так, как показано на рис. 14.2. Автоматически добавленный в таблицу ряд обозначается пунктирной границей.

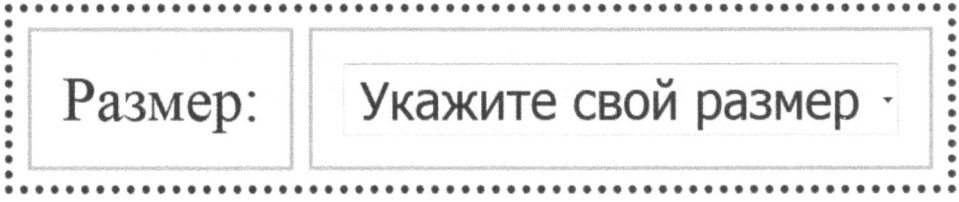


Рис. 14.2. Анонимный ряд, автоматически генерируемый при визуализации таблицы

В CSS работа механизма, обеспечивающего целостность таблиц, подчинена семи базовым правилам. Они, подобно правилам наследования и приоритетности, призваны обеспечивать неразрывность структуры документа, подлежащего стилевому форматированию.

Правила автоматического заполнения таблицы

1. Если родитель элемента `table-cell` представляется элементом, отличным от `table-row`, то последний будет создан автоматически, внедряясь в имеющуюся структуру таблицы между элементом `table-cell` и его исходным родительским элементом. Он также будет выступать родительским элементом для всех остальных элементов одного уровня с элементом `table-cell`. Рассмотрим следующую разметку.

```
system {display: table;}
name, moons {display: table-cell;}

<system>
   <name>Mercury</name>
   <moons>0</moons>
</system>
```

В этом примере код разметки анонимного объекта `table-row` добавляется между кодом элементов ячеек и элемента `system`. Следовательно, такой элемент будет выступать родителем для элементов `name` и `moons`.

Автоматическое заполнение структуры таблицы будет проводиться даже в случаях, когда родителем для ячеек выступает элемент `table-row-group`. Расширим структуру рассматриваемого примера до вида, представленного таким кодом.

```
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 подобной структуре оба набора ячеек таблицы заключены в объектах `table-row`, выступающих дочерними по отношению к объектам `planet`.

2. Если родитель элемента `table-row` представляется объектом, отличным от `table`, `inline-table` или `table-row-group`, то анонимный элемент `table` будет добавляться между элементом `table-row` и его исходным родителем. Такой объект будет выступать родительским элементом для всех остальных элементов одного уровня с элементом `table-row`. Рассмотрим следующий фрагмент документа, получающий стилевое форматирование.

```
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>
```

Так как визуализация родителя элемента `planet` выполняется согласно значению `block` свойства `display`, анонимный объект `table` внедряется в имеющуюся структуру между элементами `docbody` и `planet`. Таким образом, он будет включать сразу оба одноуровневых элемента `planet`.

3. Если родитель элемента `table-column` представляется объектом, отличным от `table`, `inline-table` или `table-column-group`, то анонимный элемент `table` будет добавляться между элементом `table-column` и его исходным родителем. Его поведение полностью равнозначно описанному в предыдущем пункте, за тем лишь исключением, что предписывается для колонок, а не рядов таблицы.

4. Если родитель элемента `table-row-group`, `table-header-group`, `table-footer-group`, `table-column-group` или `table-caption` представляется объектом, отличным от `table`, то такой анонимный элемент будет добавляться между текущим элементом и его исходным родителем.

5. Если дочерний объект элемента `table` или `inline-table` представляется элементом, отличным от `table-row-group`, `table-header-group`, `table-footer-group`, `table-row` или `table-caption`, то между ним и элементом `table` будет добавляться анонимный объект `table-row`. Такой анонимный элемент будет заключать все элементы одного уровня с обозначенным в условии дочерним элементом, отличные от `table-row-group`, `table-header-group`, `table-footer-group`, `table-row` или `table-caption`. В качестве примера рассмотрим такой фрагмент документа.

```
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>
```

В данном документе анонимный объект table-row внедряется между элементом system и вторым набором элементов name и moons. Элемент planet в него не заключается, так как его свойство display установлено в значение table-row.

6. Если дочерний объект элемента table-row-group, table-header-group или table-footer-group представляется элементом, отличным от table-row, то между ним и его дочерним элементом будет добавляться анонимный объект table-row. Такой анонимный элемент будет заключать все элементы одного уровня с обозначенным в условии дочерним элементом, отличные от table-row. Рассмотрим следующий пример документа.

```
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>
```

Здесь каждый набор элементов name и moons заключается в анонимный элемент table-row. В первом случае анонимный объект table-row добавляется между элементом planet и его дочерним элементом, поскольку первый является элементом table-row-group таблицы. Обратите внимание на то, что второй набор элементов помещается в анонимный объект согласно правилу 5.

7. Если дочерний объект элемента table-row представляется элементом, отличным от table-cell, то между ним и его дочерним элементом будет добавляться анонимный элемент table-cell. Такой анонимный элемент будет заключать все элементы одного уровня с обозначенным в условии дочерним элементом, отличные от table-row. Для примера рассмотрим такой фрагмент документа.

```
system {display: table;}
planet {display: table-row;}
name, moons {display: table-cell;}

<system>
   <planet>
      <name>Mercury</name>
      <num>0</num>
   </planet>
</system>
```

В этом случае анонимный объект table-cell добавляется между элементами planet и num, так как свойство display последнего не представлено ни одним из значений, объявляющих элементы таблицы.

Подобным образом осуществляется автозаполнение таблицы анонимными контейнерами строчного типа. Предположим, что предыдущий документ лишен разметки элемента num, как показано ниже.

```
<system>
   <planet>
      <name>Mercury</name>
      0
   </planet>
</system>
```

Элемент 0 будет послушно включаться в анонимный объект table-cell. В качестве продолжения рассмотрим, как будет форматироваться данный пример согласно требованиям спецификации CSS.

```
example {display: table-cell;}
row {display: table-row;}
hey {font-weight: 900;}

<example>
   <row>Это <hey>верхний</hey> ряд</row>
   <row>Это <hey>нижний</hey> ряд</row>
</example>
```

Здесь в анонимные объекты table-cell заключаются текстовые фрагменты и элементы hey каждого из элементов row.

Слои таблицы

Визуализация таблицы с помощью CSS осуществляется на шести импровизированных слоях, каждый из которых содержит свой тип элементов (рис. 14.3).

В общем случае на отдельном слое размещаются компоненты таблицы каждого из типов, объявленных в стилевых правилах. Таким образом, элемент table с зелеными фоном и однопиксельной рамкой будет отображаться на самом нижнем слое. Поверх него отображается форматирование, заданное для групп колонок, а поверх него — форматирование отдельных колонок и т.д. Последним выводится самый верхний слой, содержащий форматирование, которое назначено ячейкам таблицы.

Как видите, порядок форматирования таблицы строго регламентирован, поэтому, например, фон таблицы никогда не просматривается через фон ячейки. Самое важное в схеме, показанной на рис. 14.3, заключается в расположении слоев рядов над слоем колонок — фон ряда всегда перекрывает фон колонки.

Не забывайте, что по умолчанию элементам задается прозрачный фон. Следовательно, в документе, представленном следующей разметкой (рис. 14.4), через прозрачные ячейки таблицы будет просматриваться фон других ее элементов (если, конечно, он объявлялся специальным образом).

```
<table style="background: #B84;">
   <tr>
      <td>hey</td>
      <td style="background: #ABC;">there</td>
```

```
    </tr>
    <tr>
       <td>what's</td>
       <td>up?</td>
    </tr>
    <tr style="background: #CBA;">
       <td>not</td>
       <td style="background: #ECC;">much</td>
    </tr>
</table>
```

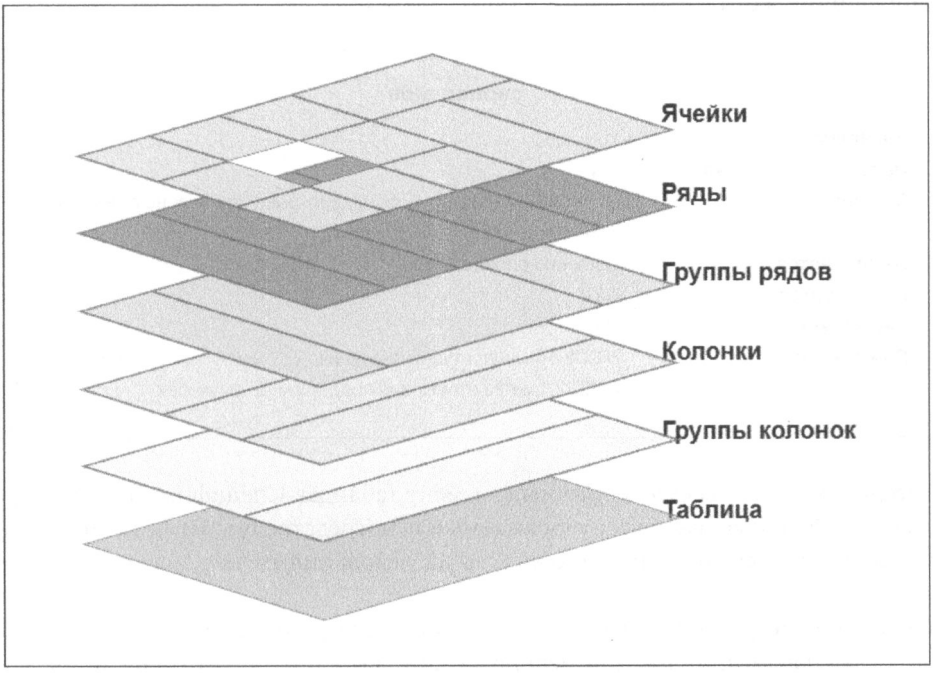


Рис. 14.3. Слои таблицы

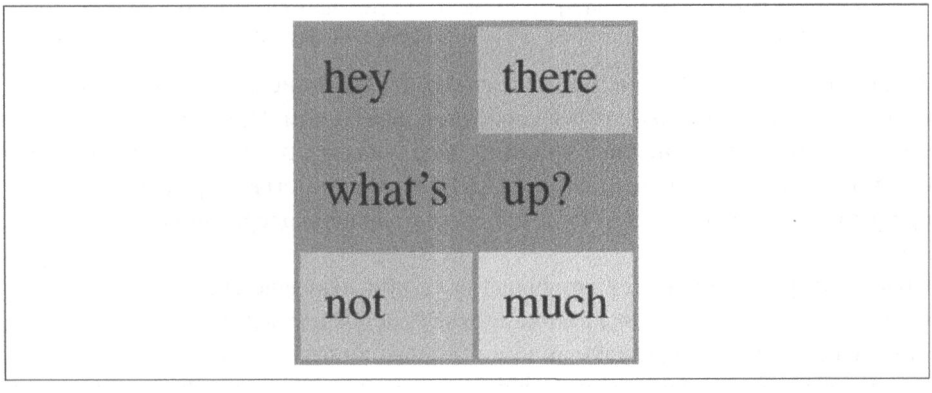


Рис. 14.4. Проступание фона элементов таблицы через прозрачный фон расположенных выше элементов (см. цветные иллюстрации на веб-сайте)

Заголовки

Заголовок таблицы является тем, чем он есть на самом деле, — коротким текстовым фрагментом в начале таблицы, описывающим ее назначение. В частности, таблица, содержащая данные о биржевых котировках, может предваряться заголовком "Динамика изменения стоимости акций за первый квартал 2018 года". Свойство `caption-side` позволяет размещать элемент как в нижней, так и в верхней части таблицы, в зависимости от ее структуры (в HTML5 элемент `caption` может выступать только первым дочерним элементом таблицы, но в других языках разметки такого ограничения может не быть).

caption-side

Значение	`top` | `bottom`
Начальное значение	`top`
Применяется	Элементы, свойство `display` которых установлено в значение `table-caption`
Вычисляется	Согласно определению
Наследуется	Да
Анимируется	Нет
Примечание	В CCS2 были доступны значения `left` и `right`, отмененные в CSS2.1 из-за отсутствия поддержки в браузерах

Заголовок — это весьма необычный элемент таблицы. Спецификация CSS определяет его как блочный элемент, отображаемый непосредственно перед таблицей (либо после нее), но получающий возможность наследования от нее значений стилевых свойств.

Особенности форматирования заголовков таблицы наглядно продемонстрированы в следующем простом примере, результат выполнения которого приведен на рис. 14.5.

```
caption {background: #B84; margin: 1em 0; caption-side: top;}
table {color: white; background: #840; margin: 0.5em 0;}
```

Текст элемента `caption` наследует от таблицы значение `white` свойства `color` несмотря на то, что заголовок снабжен собственным фоном. При этом внешняя граница таблицы отстоит от внешней границы заголовка на `1em` благодаря схлопыванию их верхних отступов. Наконец, ширина заголовка полностью определяется шириной содержимого таблицы (элемента `table`), выступающей содержащим блоком для элемента `caption`.

Подобный результат будет получен также при передаче свойству `caption-side` значения `bottom`, хотя теперь заголовок будет располагаться не над, а под таблицей, и схлопыванию будут подлежать нижние, а не верхние отступы.

В большинстве своем заголовки оформляются так же, как и любые другие блочные элементы: к ним добавляются поля, границы, фон и т.п. Например, свойство `text-`

`align` устанавливает горизонтальное выравнивание текста заголовка. В следующем примере показано, как с его помощью текст заголовка можно выровнять по правому краю его контейнера.

```
caption {background: gray; margin: 1em 0; caption-side: top;
   text-align: right;}
```

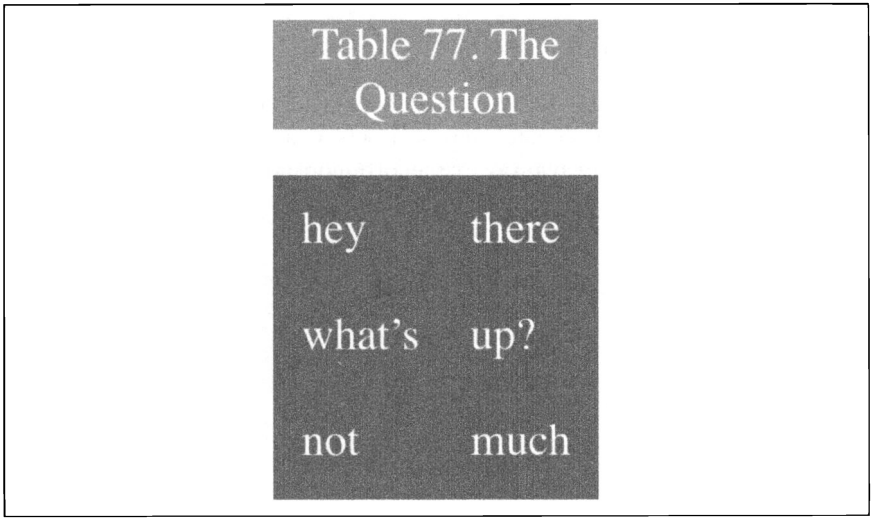


Рис. 14.5. Стилевое форматирование заголовка таблицы (см. цветные иллюстрации на веб-сайте)

Границы ячеек таблицы

В CSS поддерживаются сразу две модели представления границ ячеек. *Модель разделенных границ* позволяет добавлять раздельные границы даже к соседним ячейкам таблицы. В *модели общих границ* разделители соседних ячеек накладываются друг на друга, образуя единую границу. В ранних версиях CSS общепринятым считался первый способ представления границ, но в последних спецификациях предпочтение отдается модели общих границ.

Модель представления границ ячеек задается свойством `border-collapse`.

border-collapse

Значение	`collapse` | `separate` | `inherit`
Начальное значение	`separate`
Применяется	Элементы, свойство `display` которых установлено в значение `table-inline`
Вычисляется	Согласно определению
Наследуется	Да
Примечание	В CCS2 по умолчанию используется значение `collapse`

Назначение этого свойства — указать пользовательскому агенту способ обработки границ ячеек. Общие границы ячеек будут представляться при передаче этому свойству значения `collapse`. Значение `separate` обеспечивает их раздельное представление сразу для всех ячеек таблицы. Сначала мы ознакомимся с последним методом визуализации границ таблицы и только затем перейдем к рассмотрению модели общих границ.

Разделение границ ячеек

В этой модели каждая ячейка отстоит от соседних ячеек на определенное расстояние, поэтому их границы не совмещаются. В качестве примера рассмотрим следующий фрагмент документа, результат стилевого форматирования которого показан на рис. 14.6.

```
table {border-collapse: separate;}
td {border: 3px double black; padding: 3px;}
tr:nth-child(2) td:nth-child(2) {border-color: gray;}

<table cellspacing="0">
   <tr>
      <td>Ячейка 1</td>
      <td>Ячейка 2</td>
   </tr>
   <tr>
      <td>Ячейка 3</td>
      <td>Ячейка 4</td>
   </tr>
</table>
```

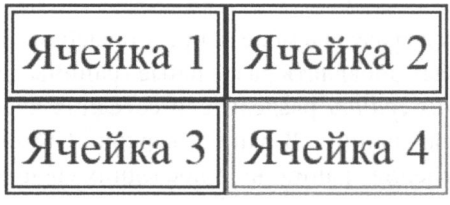


Рис. 14.6. Раздельное отображение границ ячеек

Как видите, границы соседних ячеек соприкасаются, но не перекрываются. В результате содержимое соседних ячеек отделяется тремя сплошными линиями: средняя, более толстая, линия образована вследствие слияния двух тонких линий, что лучше всего видно на примере четвертой ячейки, граница которой окрашена серым цветом.

Размещение границ ячеек вплотную друг к другу осуществляется за счет добавления в код разметки документа атрибута `cellspacing`. Тем не менее включение его в HTML-код часто становится причиной нарушений в оформлении таблиц. Как бы там ни было, CSS обладает собственными средствами установки интервалов между границами ячеек.

Интервалы между ячейками

Определившись с моделью разделенного представления границ ячеек, можно установить точный интервал между их границами. Для этих целей служит свойство `border-spacing`, обладающее более широкими функциональными возможностями, чем атрибут `cellspacing` в HTML.

border-spacing

Значение	*<length> <length>?*
Начальное значение	`0`
Применяется	Элементы, свойство `display` которых установлено в значение `table` или `table-inline`
Вычисляется	Два абсолютных значения в единицах длины
Наследуется	Да
Анимируется	Да
Примечание	Обрабатывается только при передаче свойству `border-collapse` значения `separate`

В качестве значения свойство принимает одну или две величины, выраженные в единицах измерения длины. Для равнонаправленного размещения границ ячеек на расстоянии `1px` друг от друга достаточно добавить в стилевое правило объявление `border-spacing: 1px`. Чтобы задать разные интервалы в горизонтальном и вертикальном направлениях, свойство нужно снабдить двумя значениями. Например, объявление `border-spacing: 1px 5px` обеспечивает отделение вертикальных границ соседних ячеек интервалом шириной 1 пиксель. При этом их горизонтальные границы будут отстоять друг от друга на расстояние 5 пикселей. Таким образом, при передаче свойству `border-spacing` сразу двух величин первая из них указывает интервал между горизонтальными границами соседних ячеек, а вторая — расстояние между их вертикальными границами.

Определяемые свойством интервалы также добавляются между границами внешних ячеек и внутренними краями полей самой таблицы. Таким образом, после применения следующих правил к предыдущей таблице она получит форматирование, показанное на рис. 14.7.

```
table {border-collapse: separate; border-spacing: 5px 8px;
padding: 12px; border: 2px solid black;}
td {border: 1px solid gray;}
td#squeeze {border-width: 5px;}
```

На рис. 14.7 показано, что отступ от внутреннего края поля таблицы до боковых границ крайних ячеек равен 5 пикселям. Следовательно, расстояние между ними и внешним краем вертикальных границ таблицы будет составлять `17px`. Подобным образом рассчитываются интервалы между горизонтальными границами ячеек и таблицы. Учитывая, что вертикальный интервал между соседними ячейками составляет 8 пикселей, расстояние от верхнего или нижнего внутреннего краев полей

таблицы до верхней или нижней границы ближайших ячеек будет составлять 20px. Обратите внимание на то, что интервалы, определяемые свойством border-spacing, неизменны для всей таблицы и не зависят от толщины границы ячеек.

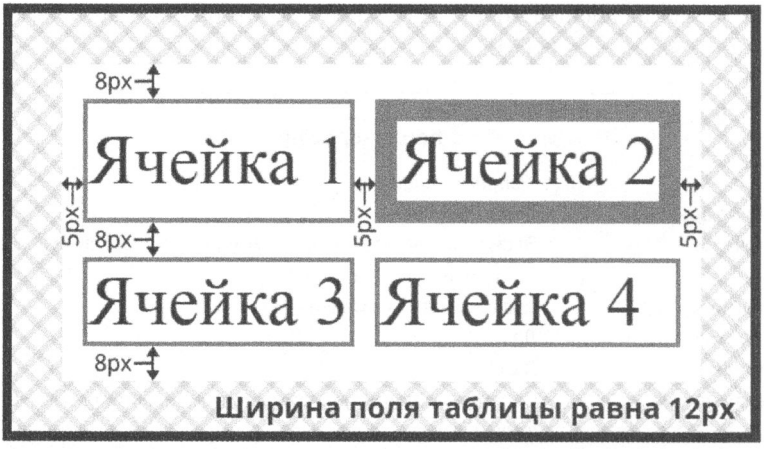


Рис. 14.7. Добавление интервалов между внешними ячейками таблицы и ее полями

Важно помнить, что интервалы указываются сразу для всей таблицы, а не для отдельных ячеек. Таким образом, попытка объявления свойства border-spacing для элемента td в предыдущем примере будет полностью проигнорирована обработчиком браузера.

В модели разделенного представления границы невозможно назначить рядам, колонкам и их группам отдельно. Пользовательские агенты не рассматривают такие объявления как действительные и полностью игнорируют их код.

Пустые ячейки

Каждая ячейка таблицы отделяется от остальных ячеек тем или иным способом на этапе визуализации документа. Но каким образом обрабатываются пустые ячейки (элементы, лишенные содержимого)? Способ их представления в таблице (их всего два) описывается свойством empty-cells.

empty-cells

Значение	show| hide
Начальное значение	show
Применяется	Элементы, свойство display которых установлено в значение table-cell
Вычисляется	Согласно определению
Наследуется	Да
Анимируется	Нет
Примечание	Обрабатывается только при передаче свойству border-collapse значения separate

Значение show указывает прорисовывать границы пустых ячеек так же, как и всех остальных ячеек таблицы. При передаче свойству empty-cells значения hide пустые ячейки остаются скрытыми, как и в случае объявления для них свойства visibility со значением hidden.

Как только в ячейку заносится некое содержимое, она перестает быть пустой. Под содержимым в данном случае подразумевается не только текст, изображения, элементы формы, но и неотображаемые символы, например пробелы ( ), символы возврата каретки, табуляции, разрыва строки и т.п. Если в ряду имеются пустые ячейки, снабженные объявлением empty-cells:hide, то он представляется так, как если бы его элемент визуализировался с помощью объявления display: none.

Совмещение границ ячеек

Несмотря на то что модель общих границ применяется для описания поведения HTML-таблиц, лишенных интервалов между ячейками, использовать ее для их стилевого форматирования намного сложнее, чем модель разделенных границ. В отличие от последней в ней приходится учитывать схлопывание отступов, выполняемое согласно определенным правилам.

- Элементы, свойство display которых установлено в значение table или inline-table, при объявлении для них свойства border-collapse со значением collapse лишаются полей, хотя могут иметь отступы. Таким образом обеспечивается единство границы таблицы и внешних границ ее крайних ячеек.
- Границы можно назначать отдельно не только ячейкам, но также рядам и колонкам и даже их группам. При этом вся таблица тоже может заключаться в собственные границы.
- В действительности нет строго разделения между моделями с разделенными и общими границами: совмещение границ происходит при наложении их друг на друга. При этом речь идет именно о совмещении, а не схлопывании границ, в отличие от схлопывания отступов у родительского и дочернего элементов. При совмещении границ визуализации подлежит только "наиболее привлекательная" из них.
- Совмещенная граница центрируется по воображаемой линии сетки, разделяющей соседние ячейки таблицы.

Последние два правила требуют более пристального внимания, а потому будут рассмотрены в следующих разделах.

Модель совмещения границ ячеек

Чтобы получить представление о совмещении границ ячеек, нужно ознакомиться с моделью, представленной на рис. 14.8.

Как известно, области содержимого и полей каждой ячейки ряда находятся полностью внутри рамки, образованной ее границами. Что касается границы между соседними ячейками ряда, то она проходит вдоль линии сетки так, что по обе ее стороны оказывается линия точно половинной ширины. При этом между ячейками

отображается только одна из двух границ. Можно предположить, что каждая из ячеек обрамляется своей границей половинной ширины, но, как показано далее, это далеко не так.

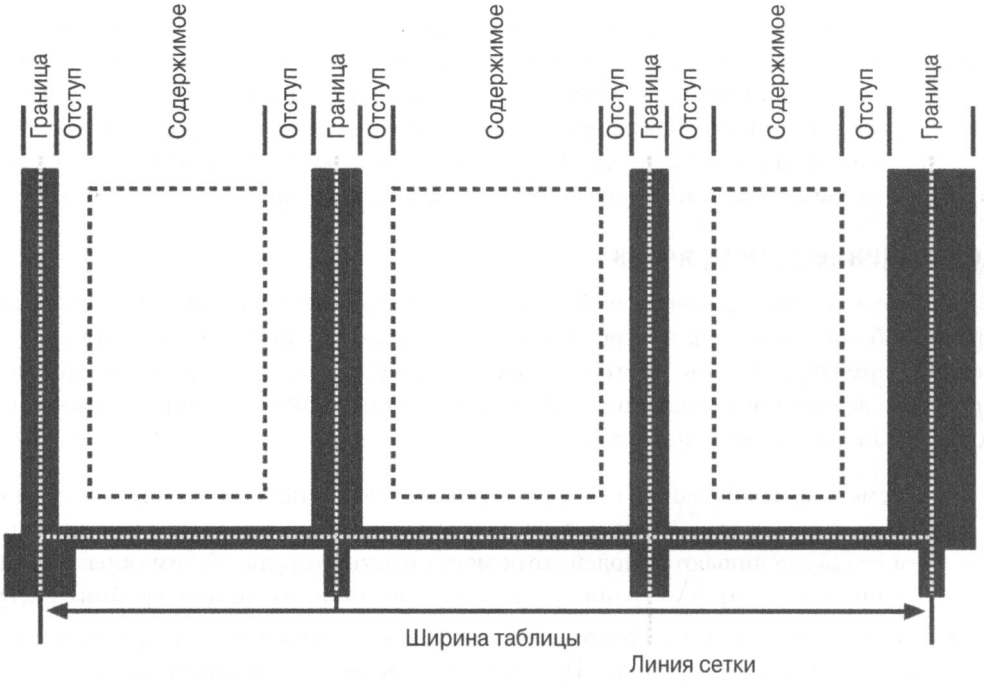


Рис. 14.8. Макет ряда таблицы, содержащего ячейки с совмещенными границами

В качестве примера рассмотрим случай, когда одна из внутренних ячеек ряда имеет сплошные зеленые, а соседние с ней ячейки — сплошные красные границы. Боковые границы такой ячейки, совмещаемые с соответствующими границами соседних ячеек, будут либо полностью красного, либо полностью зеленого цвета, но никак не двухцветными. О том, каким образом выбирается доминирующий цвет, рассказывается в следующем разделе.

Легко заметить, что границы таблицы "съедают" часть полезного пространства таблицы, уменьшая его общую ширину. А все потому, что половина каждой из границ заходит в область самой таблицы. Вторая, внешняя половина границы выступает в область ее отступов, поэтому не учитывается при вычислении ее общей ширины. Звучит запутанно, но именно так визуализируются границы в этой модели.

Для точного определения ширины всех компонентов таблицы можно использовать такую формулу, приведенную в спецификации CSS:

ширина ряда = (0,5 × `border-width-0`) + `padding-left-1` + `width-1` + `padding-right-1` + `border-width-1` + `padding-left-2` + ... + `padding-right-`*n* + (0,5 × `border-width-`*n*)

Здесь значение `border-width-n` обозначает ширину границы между ячейкой с порядковым номером *n* и следующей за ней ячейкой. Таким образом, значение `border-width-3` указывает ширину границы между третьей и четвертой ячейками. Счетчик *n* представляет общее количество ячеек таблицы.

У описанной выше формулы есть всего один недостаток: она оказывается недействительной при вычислении ширины боковых границ самой таблицы. Исходная ширина левой границы таблицы рассчитывается как половина исходной ширины левой границы первой ячейки ее первого ряда. Соответственно, исходная ширина правой границы таблицы вычисляется делением пополам исходной ширины правой границы последней ячейки ее первого ряда. Если исходная ширина боковых границ крайних ячеек последующих рядов таблицы больше, чем у соответствующих ячеек первого ряда, то они будут выступать в область отступов таблицы.

Если вычисляемая ширина границы ячейки представляется нечетным значением (пикселей, точек и т.п.), то способ ее центрирования по линии сетки определяется пользовательским агентом. Среди доступных вариантов: смещение границы в одну из боковых сторон, увеличение или уменьшение ширины границы на единицу, визуальное сглаживание ее в документе или применение иных способов, равнозначных по эффективности.

Совмещение границ

При наложении границ соседних ячеек они совмещаются и представляют в таблице одну линию. При совмещении в таблице визуализируется только одна из накладываемых границ, а другая скрывается. Порядок отображения границ ячеек регулируется специальными правилами.

- При объявлении свойства `border-style` со значением `hidden` для одной из совмещаемых границ ячейки ни одна из них визуализироваться не будет.
- При установке свойства `border-style` в значение `visible` для обеих совмещаемых границ отображается та из них, которая имеет большую ширину. Например, при совмещении однопиксельной пунктирной границы и двухконтурной границы шириной пять пикселей между ячейками будет отображаться второй вариант границы.
- Если схлопывающиеся границы имеют одинаковую ширину, но разные стили, то стиль результирующей границы будет назначаться согласно такой последовательности значений (от наиболее до наименее предпочтительного): `double`, `solid`, `dashed`, `dotted`, `ridge`, `outset`, `groove`, `inset` и `none`. При совмещении границ одинаковой толщины, одна из которых имеет стиль `dashed`, а вторая — `outset`, результирующая граница будет представлена пунктирной линией.
- Если совмещаемые границы отличаются только цветом, но не стилем и толщиной, то цвет конечной границы будет заимствоваться (от наиболее до наименее предпочтительного) у ячейки, ряда, группы рядов, колонки, группы колонок и таблицы. Таким образом, при совмещении границ (отличающихся только цветом) ячейки и колонки результирующая граница будет определяться настройками, установленными для ячейки. При совмещении границ однотипных

элементов, имеющих одинаковые стили и толщину, например двух рядов, цвет получаемой границы будет определяться цветом границы элемента, находящегося ближе к началу таблицы (левый верхний угол — в документах с системой письма слева направо, правый верхний угол — для таблиц, заполняемых на арабском языке и иврите).

В качестве примера (рис. 14.9) рассмотрим фрагмент документа, стилевое форматирование которого задается таким кодом.

```
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, как форматируются границы каждой из ячеек.

- Ячейки 1-1 и 1-4 имеют границы наибольшей ширины: 5 пикселей. Следовательно, они будут отображаться поверх границ остальных элементов, включая саму таблицу. Исключение составляет нижняя граница ячейки 1-1, которая отсутствует в силу совмещения с верхней границей ячейки 2-1.

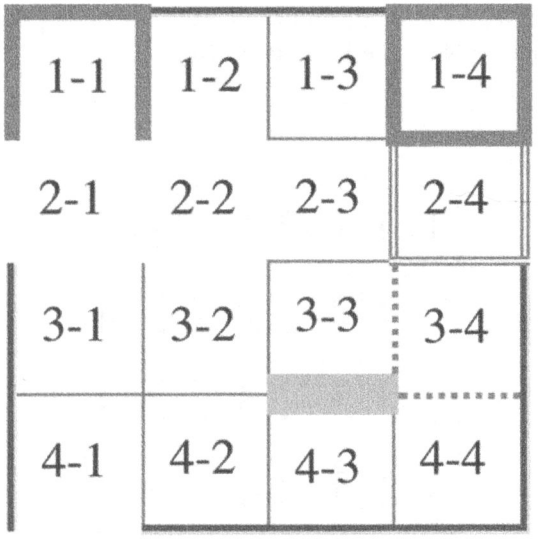


Рис. 14.9. Совмещение границ с разными стилями, толщиной и цветами часто приводит к совершенно неожиданным результатам

- Отсутствие нижней границы у ячейки 1-1 вызвано сокрытием границ у ячеек 2-1 и 2-2. В силу этого в таблице не отображаются границы всех смежных с ними элементов. Исходя из этого, у таблицы вполне ожидаемо отсутствует левая граница, совмещенная с левой границей ячейки 2-1. Сокрытие нижней границы у ячейки 4-1 приводит к блокировке визуализации остальных совмещенных с ней границ.
- Двойная верхняя граница ячейки 2-4 шириной 3 пикселя замещается нижней границей ячейки 1-4, представленной сплошной линией 5-пиксельной ширины. При этом левая граница ячейки 2-4 замещает правую границу ячейки 2-3 как более широкая и в стилевом плане привлекательная. Кроме того, ее нижняя граница доминирует над верхней границей ячейки 3-4 — несмотря на равенство их ширины, спецификация считает двойную границу привлекательнее пунктирной линии.
- Серебристая нижняя граница ячейки 3-3 шириной 13 пикселей не только замещает верхнюю границу расположенной под ней границы 4-3, но и оказывает влияние на позиционирование их содержимого и содержимого рядов, к которым они относятся.
- Внешние границы всех остальных ячеек, размещенных у краев таблицы и лишенных специального форматирования, замещаются 3-пиксельной границей, назначенной самой таблице.

Как видите, при стилевом форматировании элементов таблицы приходится принимать в расчет огромное количество факторов, что делает его невероятно сложной задачей. Но даже столь неоднозначные возможности выглядят поистине

потрясающими по сравнению с представленными в первой редакции CSS, которые поддерживались только в Netscape 1.1 (рис. 14.10).

```
table {border-collapse: collapse; border: 2px outset gray;}
td {border: 1px inset gray;}
```

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. 14.10. Оформление таблицы с помощью самых первых стилевых инструментов

Размер таблицы

Завершив знакомство с инструментами стилевого форматирования таблиц и их элементов, можно смело переходить к изучению свойств, отвечающих за настройку их размеров. В общем случае ширина таблицы может быть *фиксированной* или устанавливаться *автоматически*. Ее высота всегда вычисляется автоматически и не зависит от алгоритма расчета ширины.

Ширина таблицы

Поддержка сразу двух алгоритмов расчета ширины таблицы предполагает включение в CSS специального свойства, которое позволяет указывать способ вычислений, выполняемых в стилевом правиле. Переход от одного алгоритма к другому осуществляется с помощью свойства `table-layout`.

table-layout

Значение	`auto` | `hide`
Начальное значение	`auto`
Применяется	Элементы, свойство `display` которых установлено в значение `table` или `table-inline`
Вычисляется	Согласно определению
Наследуется	Да
Анимируется	Нет

Прежде всего, алгоритмы вычисления ширины таблицы отличаются не столько результатами проводимых вычислений, сколько скоростью их выполнения. В таблице фиксированной ширины все необходимые расчеты проводятся намного быстрее, чем в случае проведения автоматических вычислений.

Таблица фиксированной ширины

Горизонтальный размер таблицы фиксированной ширины не зависит от объема содержимого, помещаемого в ее ячейки. Он обусловливается только шириной ее колонок и отдельных ячеек.

Модель фиксированной ширины предполагает следующий порядок стилевого форматирования элементов таблицы.

1. Ширина колонки определяется значением свойства `width` (отличным от `auto`), заданным одному из ее элементов.
2. Ширина колонки, свойство `width` которой определяется ключевым словом `auto`, задается размером ее первой ячейки, определенным в явном виде. Если первая ячейка колонки распространяется на другие колонки, то для получения ширины каждой из них объявленную ширину нужно разделить на их количество.
3. Все колонки, свойство `width` которых установлено в значение `auto`, должны иметь одинаковую ширину.

На данный момент можно смело утверждать, что ширина таблицы определяется бо́льшим из двух значений: величиной свойства `width` или суммарной шириной ее колонок. Если ширина таблицы фиксированного размера больше, чем суммарная ширина ее колонок, то лишнее свободное пространство равномерно распределяется между всеми ними.

Высокая скорость определения размера таблицы с помощью этого метода обусловливается проведением вычислений на основе данных о размерах ячеек только первого ряда. Размер ячеек последующих рядов определяется шириной колонок, заданных для ячеек первого ряда. Следовательно, их размер всецело зависит только от исходной ширины ячеек первого ряда и не учитывает значения свойства `width`, заданного для ячеек любых других рядов таблицы. Если содержимое таких ячеек не помещается в отведенное под него пространство, то способ его представления всецело определяется значением свойства `overflow`. Как известно, оно может обрезаться, помещаться в прокручиваемую область или выступать в область соседних ячеек.

Рассмотрим следующий пример стилевого форматирования таблицы, результат выполнения которого показан на рис. 14.11.

```
table {table-layout: fixed; width: 400px;
border-collapse: collapse;}
td {border: 1px solid;}
col#c1 {width: 200px;}
#r1c2 {width: 75px;}
#r2c3 {width: 500px;}
```

```
<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>
```

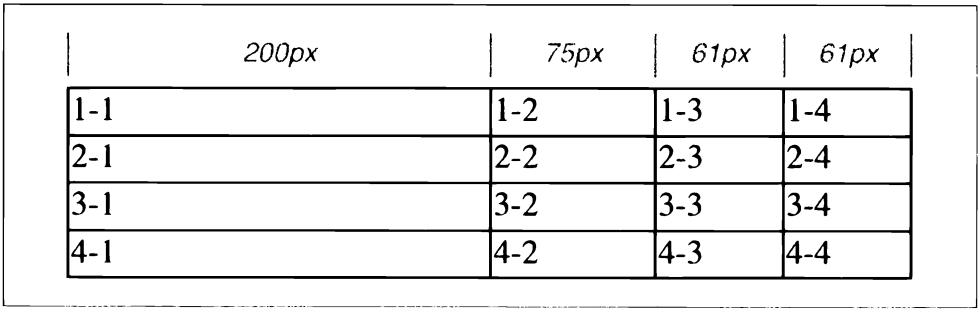


Рис. 14.11. Макет таблицы фиксированного размера

Как видно на рис. 14.11, ширина первой колонки равна 200 пикселям, что составляет половину ширины всей таблицы (`400px`). Вторая колонка имеет ширину 75 пикселей, поскольку именно такой горизонтальный размер задан для второй ячейки первой строки таблицы. Третья и четвертая колонки имеют одинаковую ширину: 61 пиксель. Это значение вычисляется как половина разницы между шириной таблицы и суммарной шириной первых двух колонок ячейки. Как известно, общая ширина первой и второй колонок составляет 278 пикселей (`275px` плюс ширина границы между ними: `3px`). Вычитание этого значения из общей ширины таблицы (`400px`)

дает суммарную ширину третьей и четвертой колонок: 122 пикселя. "А как же быть со значением `500px` для ячейки `#r2c3`?" — спросите вы. А никак: оно будет проигнорировано пользовательским агентом, поскольку указанная ячейка не принадлежит к первому ряду таблицы.

Заметьте: чтобы иметь фиксированную ширину, таблице совсем не обязательно объявлять свойство `width` со строго заданным значением, хотя это сильно упрощает вычисления. В частности, обработка приведенного ниже стилевого правила показывает, что таблица будет на 50 пикселей уже своего родительского элемента. Полученное таким образом значение можно смело применять для дальнейших вычислений в модели таблицы фиксированной ширины.

```
table {table-layout: fixed; margin: 0 25px; width: auto;}
```

От проведения подобных вычислений можно смело отказаться, так как пользовательские агенты прекрасно справляются с автоматическим вычислением ширины таблицы, выполняемым при передаче ее свойству `width` значения `auto`.

Автоматическое определение ширины таблицы

Автоматическое вычисление ширины таблицы выполняется намного медленнее, чем расчет ее горизонтального размера в модели фиксированной ширины, хотя базовые принципы этого алгоритма прекрасно знакомы авторам документов, поскольку применяются в HTML с первых версий технологии. В большинстве современных браузеров для перехода к этому алгоритму достаточно назначить свойству `width` значение `auto`, не объявляя специальное значение свойства `table-layout`.

Причиной низкой скорости автоматического вычисления ширины таблицы становится необходимость принятия в расчет размера ячеек всех без исключения, а не только первого ряда таблицы. В свою очередь, ширина каждой из ячеек таблицы зависит от размера содержимого и примененного к ней стилевого форматирования. Таким образом, перед вычислением общей ширины таблицы пользовательскому агенту необходимо рассчитать размер всех без исключения ячеек таблицы.

Для проведения точного расчета требуется проанализировать содержимое таблицы, влияющее на ее макет самым непосредственным образом. В частности, при помещении в последний ряд таблицы изображения шириной 400 пикселей указанную ширину получат все ячейки, расположенные над текущей (все ячейки текущей колонки), что вызывает необходимость в перерасчете размеров всех расположенных выше рядов таблицы. Нередки ситуации, когда в процессе вычисления размера таблицы пользовательскому агенту требуется учитывать влияние на него содержимого всех без исключения ячеек.

На более детальном уровне алгоритм автоматического вычисления ширины таблицы сводится к выполнению следующих действий.

1. Для каждой ячейки колонки определяется минимальная и максимальная возможная ширина.
2. Минимальная ширина ячейки определяется ее содержимым. Исходя из того что содержимое ячейки не может выступать за края ячейки, но может располагаться в несколько строк, минимальная ширина ячейки определяется размером

наименьшего неделимого компонента содержимого (свойство `width` установлено в значение `auto`). Свойство `width` устанавливает ширину ячейки только в случае, когда ее значение превышает минимально возможный размер ячейки.

3. Максимальная ширина ячейки определяется как размер, при котором ее содержимое помещается в одну строку, без автоматического переноса на другие строки (не считая явных разрывов строк, образованных, например, тегом `<br>`).
4. Для каждой колонки определяется минимальная и максимальная возможная ширина.
5. Минимальная ширина колонки определяется как наибольшая минимальная ширина ее ячеек. Минимальная ширина колонки определяется свойством `width` только в случаях, когда его значение больше минимальной ширины любой из ее ячеек, а сама она устанавливается в явном виде.
6. Максимальная ширина колонки определяется как наибольшая максимальная ширина ее ячеек. Максимальная ширина колонки определяется свойством `width` только в случаях, когда его значение больше максимальной ширины любой из ее ячеек, а сама она устанавливается в явном виде. Такие требования в полной мере согласуются со стандартами спецификации HTML, в которых ширина колонки определяется самой широкой ее ячейкой.
7. Минимальная ширина ячейки, занимающей сразу несколько колонок, должна равняться суммарной минимальной ширине относящихся к ней колонок. Подобным образом максимальная ширина такой ячейки будет определяться суммарной максимальной шириной колонок, на которые она распространяется. Любые изменения ширины ячейки вызывают соответствующие изменения в ширине занимаемых ею колонок.

В дополнение к описанным выше правилам пользовательскому агенту приходится дополнительно пересчитывать ширину элементов, представленную процентными значениями, — все они указываются относительно общей ширины таблицы, которая на момент расчетов может быть неизвестна! Таким образом, процентные значения становятся доступными для подстановки в расчетные формулы только на последних этапах вычислений.

По завершении описанных выше действий пользовательский агент получает сведения о том, насколько узкими или широкими *могут быть* колонки. Для определения их реального размера ему предстоит решить несколько дополнительных задач.

1. Если вычисляемая ширина таблицы представляется значением свойства `width`, отличным от `auto`, то она сравнивается с суммарной шириной всех ее колонок, границ и интервалов между ними. (На этом же этапе вычисляются размерные величины, выраженные процентными значениями.) Свойство `width` представляет ширину таблицы только в случае, когда его вычисляемое значение больше суммарной ширины всех ее колонок, а также границ и интервалов между ними. Разница между указанными значениями равномерно распределяется между всеми колонками в виде дополнительного свободного пространства.

2. При передаче свойству `width` таблицы значения `auto` конечная ширина таблицы определяется суммарной шириной всех ее колонок, а также границ и интервалов между ними. Такой подход полностью согласуется с требованием спецификации HTML, в которой ширина таблицы не превышает размер, достаточный для размещения в ней всего необходимого содержимого. Ширина колонок, представленная процентными величинами, не подлежит точному вычислению, хотя и оценивается пользовательским агентом при подборе вычисляемых значений свойства `width`.

Только по завершении последнего этапа вычислений пользовательский агент приступает к визуализации макета таблицы.

Для иллюстрации описанного выше подхода рассмотрим следующий пример стилевого форматирования таблицы, результат представления которого в браузере показан на рис. 14.12.

```
table {table-layout: auto; width: auto;
border-collapse: collapse;}
td {border: 1px solid; padding: 0;}
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>
```

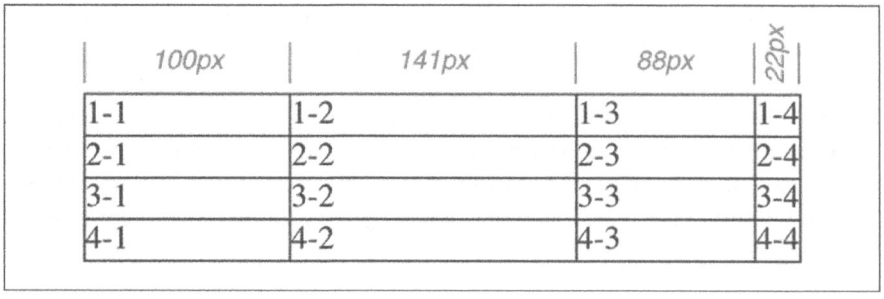


Рис. 14.12. Таблица с автоматически задаваемым размером

Проанализируем в пошаговом режиме, каким образом определяется ширина каждой из колонок таблицы.

- В первой колонке в явном виде объявляется ширина только ячейки 4-1: она составляет `100px`. Поскольку содержимое ячейки представлено очень короткой строкой, ее минимальная и максимальная ширина совпадают и также равны `100px`. (Для получения ячеек с большей максимальной шириной в них нужно поместить содержимое, включающее несколько предложений и занимающее сразу несколько строк.)
- Вторая колонка содержит сразу две ячейки с явно заданными размерами: 1-2 шириной `40%` и 2-2 шириной `50px`. Таким образом, минимальная ширина колонки будет равняться `50px`, а ее максимальная ширина составит `40%` от конечной ширины таблицы.
- Третья колонка характеризуется всего одной ячейкой (3-3) с точно заданной шириной: `25%`. При этом ширина колонки объявляется равной `35px`. Следовательно, минимальная ширина колонки составляет 35 пикселей, а ее максимальная ширина равняется 25% от конечного размера таблицы.
- В четвертой колонке тоже содержится одна-единственная ячейка (4-4) с точно заданным размером (`1px`), что несомненно меньше минимальной ширины ее содержимого. Исходя из этого, минимальная и максимальная ширина колонки в точности совпадают с минимальной шириной содержимого ее ячеек: 22 пикселя.

В результате проведенного анализа пользовательский агент определяет, что колонки таблицы имеют такие минимальные и максимальные размеры.

- 1 колонка: минимальная ширина — `100px`, максимальная ширина — `100px`.
- 2 колонка: минимальная ширина — `50px`, максимальная ширина — `40%`.
- 3 колонка: минимальная ширина — `35px`, максимальная ширина — `25%`.
- 4 колонка: минимальная ширина — `22px`, максимальная ширина — `22px`.

Для вычисления минимальной ширины таблицы достаточно сложить значения минимальной ширины всех ее колонок, а для определения ее максимальной ширины — их максимальные значения. Простой математический расчет показывает, что минимальная ширина таблицы с учетом совмещения границ колонок составляет

215 пикселей. При этом ее максимальная ширина равняется `123px + 65%`, где первое значение представляет суммарную ширину первой и последней колонок с учетом совмещения границ. В абсолютном исчислении максимальная ширина таблицы определяется величиной 351,42857142857143 пикселя (значение `123px` представляет 35% от конечной ширины таблицы). Рассчитав общую ширину таблицы, можно вычислить абсолютную ширину колонок, представленных процентными величинами. Вторая колонка будет иметь максимальную ширину 140,5 пикселя, а максимальная ширина третьей колонки составит 87,8 пикселя. Пользовательский агент может округлить полученные выше значения до величин `141px` и `88px` (см. рис. 14.12) — все зависит от применяемого в нем алгоритма визуализации таблиц.

Для точного определения конечной ширины таблицы пользовательскому агенту совсем не обязательно вычислять ее максимальный размер, поскольку существуют иные, не менее эффективные методики.

Рассмотренный выше пример достаточно прямолинейный и простой для анализа, насколько сложным он ни выглядел бы при первом знакомстве: содержимое всех ячеек таблицы имеет одинаковую длину, а большинство размерных величин выражено в пикселях. В реальных таблицах, заполняемых изображениями и большими текстовыми абзацами, ширина колонок определяется намного сложнее, а их конечный макет значительно труднее поддается точному анализу.

Высота таблицы

Осознав всю сложность определения ширины таблицы, можно смело переходить к не менее трудной задаче вычисления ее высоты. Согласно определениям, принятым в спецификации CSS, эта задача не выглядит сложной, но тем нем менее является таковой по своей сути (особенно для разработчиков браузеров).

В самом простом случае высота таблицы указывается в явном виде с помощью свойства `height`. В буквальном понимании это означает, что высота таблицы может быть меньше или больше суммарной высоты всех ее рядов. При этом свойство `height` обрабатывается пользовательским агентом подобно свойству `min-height`: если суммарная высота рядов таблицы больше значения, передаваемого свойству `height`, то оно будет проигнорировано обработчиком.

При этом спецификация лишена указаний по обработке обратной ситуации — передачи свойству `height` значения, превышающего суммарную высоту рядов таблицы. Хочется верить, что это упущение будет наверстано в будущих версиях CSS. Остается только предполагать, каким образом пользовательские агенты будут справляться со столь неоднозначной задачей: добавлять между рядами таблицы промежутки свободного пространства, увеличивать их высоту или применять иные, менее очевидные способы решения.

К концу 2017 года увеличение высоты таблицы осуществлялось пользовательскими агентами преимущественно за счет изменения высоты ее рядов. Согласно общепринятому алгоритму, выглядело это так: разницу между заявленной высотой таблицы и суммарной высотой ее рядов нужно разделить на их количество, а затем добавить полученную величину к высоте каждого из них.

При объявлении для таблицы свойства `height` со значением `auto` ее высота в точности равняется суммарной высоте ее рядов, границ и междурядных интервалов. Вычисление высоты рядов таблицы осуществляется согласно алгоритму, подобному применяемому при определении ширины ее колонок. Вначале анализируется объем содержимого каждой из ячеек таблицы и на основе полученных данных прогнозируется их минимальная и максимальная высота. Полученные таким образом сведения позволяют предельно точно определить высоту всех рядов таблицы. Для получения общей высоты таблицы достаточно сложить их между собой. Ситуация немного напоминает верстку строчных элементов, выполняемую в большем количестве измерений и с меньшим количеством заранее заданных размерных характеристик.

Чтобы понять, насколько сложно бывает оценить общую высоту таблицы даже при явном объявлении значения свойства `height`, представьте, что вам предстоит решить эту задачу с учетом следующих факторов:

- высота ячеек указывается в процентах;
- высота рядов и групп рядов указывается в процентах;
- некоторые ячейки таблицы размещаются сразу в нескольких рядах.

Как видите, высота таблицы в большей степени определяется внутренними вычислительными алгоритмами пользовательского агента, нежели правилами спецификации CSS. В итоге каждый из браузеров может вычислять ее по-своему, что делает верстку таблиц с явно заданным значением свойства `height` невероятно сложной задачей. Исходя из этого, старайтесь всячески избегать его объявления в собственных таблицах стилей.

Выравнивание

В силу весьма странного стечения обстоятельств выравнивание содержимого ячеек таблицы выполняется намного проще и более однозначно, чем вычисление ее высоты и ширины. Данное утверждение справедливо даже для вертикального выравнивания, от способа выполнения которого во многом зависит высота рядов таблицы.

Проще всего в таблицах CSS осуществляется горизонтальное выравнивание. Для выравнивания содержимого отдельной ячейки к ней применяется свойство `text-align`. Следует заметить, что при выравнивании ячейка таблицы рассматривается как контейнер блочного типа, поэтому значение свойства `text-align` применяется сразу ко всему ее содержимому.

Вертикальное выравнивание содержимого таблицы выполняется с помощью свойства `vertical-align`. Оно принимает такие же значения, как и при вертикальном выравнивании строчных элементов, хотя и в контексте позиционирования содержимого в пределах ячейки таблицы. Ниже описано назначение трех наиболее часто применяемых ключевых слов.

`top`

Верхний край содержимого ячейки выравнивается по верхнему краю ее ряда. Содержимое ячейки, занимающей сразу несколько рядов, выравнивается по верхнему краю первого (высокорасположенного) из них.

`bottom`

Нижний край содержимого ячейки выравнивается по нижнему краю ее ряда. Содержимое ячейки, занимающей сразу несколько рядов, выравнивается по нижнему краю последнего (низкорасположенного) из них.

`middle`

Середина содержимого ячейки совмещается с серединой ее ряда. Середина содержимого ячейки, занимающей сразу несколько рядов, совмещается с их общей серединой.

В качестве примера выравнивания содержимого таблицы рассмотрим следующий пример, результат выполнения которого показан на рис. 14.13.

```
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>
         Содержимое этой ячейки выровнено по верхнему краю
      </div>
      </td>
      <td id="r1c2">
      <div>
         Содержимое этой ячейки выровнено по центру
      </div>
      </td>
      <td id="r1c3">
      <div>
         Содержимое этой ячейки выровнено по нижнему краю
      </div>
      </td>
   </tr>
</table>
```

В каждом из случаев эффект выравнивания достигается за счет автоматического расширения полей ячеек. В примере, показанном на рис. 14.13, смещение содержимого к верхнему краю первой ячейки выполняется в результате увеличения нижнего поля. Для выравнивания содержимого второй ячейки (посередине высоты) ей задаются одинаковые верхнее и нижнее поля. В последнем случае содержимое смещается в нижнюю часть ячейки, снабженную широким верхним полем.

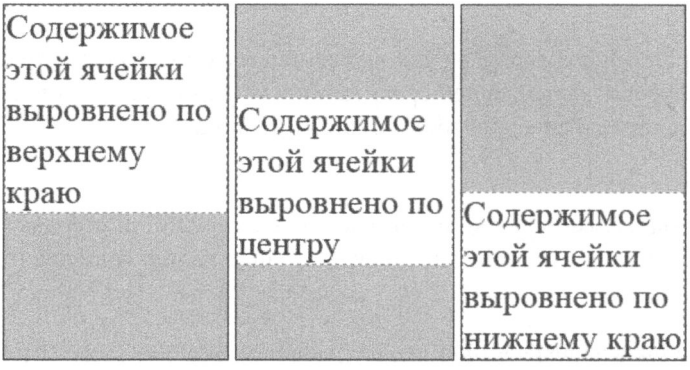


Рис. 14.13. Вертикальное выравнивание содержимого ячеек

Для выравнивания содержимого таблицы применяется еще одно ключевое слово: `baseline`. Его применение не столь однозначно, как у рассмотренных выше значений, а потому оно заслуживает отдельного описания.

`baseline`

Базовая линия ячейки совмещается с базовой линией ее ряда. Базовая линия ячейки, занимающей сразу несколько рядов, выравнивается по базовой линии первого (высокорасположенного) из них.

Проще всего ознакомиться с указываемым этим ключевым словом способом выравнивания содержимого таблицы на реальном примере (рис. 14.14).

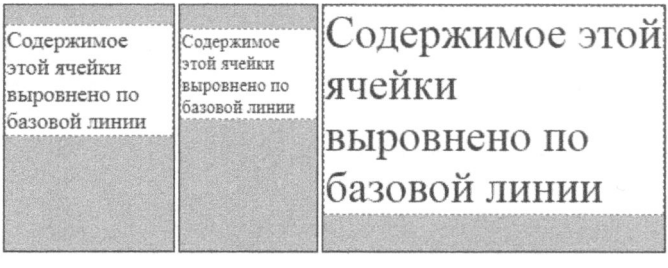


Рис. 14.14. Выравнивание содержимого ячеек по базовой линии

Базовая линия ряда определяется как самая низкорасположенная начальная базовая линия (базовая линия первой текстовой строки) ее ячеек. Исходя из такого определения, базовая линия ряда, показанного на рис. 14.14, будет совпадать с начальной базовой линией третьей ячейки. Следовательно, при выравнивании с помощью ключевого слова `baseline` именно к ее уровню будут смещаться начальные базовые линии первых двух ячеек.

Как и при выравнивании по верхнему, нижнему краям и середине ячейки, изменение положения содержимого ячеек при выравнивании базовых линий достигается корректировкой размеров верхнего и нижнего полей. Если ни одна из ячеек не выравнивается по базовой линии, то ее наличие у ряда вообще не определяется пользовательским агентом — в этом попросту нет необходимости.

Детально операцию выравнивания содержимого таблицы с помощью ключевого слова `baseline` можно представить следующим образом.

1. Если по базовой линии выравнивается хотя бы одна ячейка ряда, то пользовательский агент сначала определяет положение его начальной базовой линии, после чего позиционирует по нему содержимое форматируемой ячейки.
2. В ячейки добавляется содержимое, выравниваемое по верхнему краю. Тем самым устанавливается исходный нижний край ряда и его высота.
3. Если в ряду имеются выравниваемые посередине и/или по нижнему краю ячейки, высота содержимого которых превышает высоту ряда, то последняя увеличивается до размера, обеспечивающего полное включение их в ряд.
4. Содержимое добавляется во все остальные ячейки ряда. Для корректного размещения содержимого, высота которого меньше высоты ряда, соответствующие ячейки снабжаются расширенными полями.

Значения `sub`, `super`, `text-top` и `text-bottom` свойства `vertical-align` не влияют на выравнивание содержимого ячеек. Скорее всего, они будут обрабатываться пользовательскими агентами как значение `baseline` или `top`.

Резюме

Верстка таблиц относится к одной из самых трудноразрешимых задач веб-дизайна. Даже будучи опытным верстальщиком, вам придется изрядно потрудиться, работая над макетами документов, включающими таблицы. В HTML таблицы представляются исключительно рядными структурами. Табличная модель CSS частично снимает ограничения, устанавливаемые спецификацией HTML, позволяя обращаться к колонкам таблицы напрямую и снабжать их независимым стилевым форматированием. Свойства настройки размеров таблицы и выравнивания ее содержимого позволяют более точно контролировать внешний вид таблицы в общем макете документа.

Свойства настройки таблиц могут использоваться для форматирования регулярных элементов документа, таких как `div` и `section`, — такой подход лежит в основе табличной верстки документов, в большей степени применимый к XML-документам, в которых таблицы могут компоноваться из любых заранее объявленных элементов.

ГЛАВА 15

Списки и генерируемое содержимое

В CSS спискам отводится особое место. В первую очередь потому, что элементы списка представляются контейнерами блочного типа, которые предваряются специальными объектами, формально отсутствующими в объектной модели документа. Так, упорядоченный список состоит из последовательности элементов, нумерованных в числовом или алфавитном порядке. При этом форматирование символов нумерации осуществляется преимущественно пользовательским агентом и стилевому оформлению не подлежит. Чтобы обеспечить целостность структуры документа, браузер чаще всего представляет символы нумерации в предельно простом виде.

Генерируемое таким образом содержимое не подлежало форматированию с помощью стилевых свойств CSS1 — инструменты, предназначенные для решения этой задачи, появились только в версии CSS2. Теперь авторам документов разрешается определять порядок нумерации элементов списка и их формат, а также назначать символы счетчиков *произвольным* элементам документа, а не только элементам списков. Более того, механизм генерирования содержимого CSS предполагает добавление в документы не только счетчиков, но и данных многих других типов: строк, значений атрибутов и даже внешних данных. Таким образом, в CCS генерируемое содержимое может применяться для включения в документы значков гиперссылок, редакторских символов и многих других графических объектов без дополнительной верстки документа.

Как бы там ни было, изучение указанных выше возможностей лучше всего начать с рассмотрения принципов создания обычных нумерованных списков.

Списки

В определенном смысле в виде списков можно представить любой неповествовательный текст документа. Результаты переписи населения, состав Солнечной системы, фамильное древо, ресторанное меню и контакты телефонной книги — эти и многие другие сведения лучше всего просматривать при представлении их в виде одно- или многоуровневого списка. Широкое разнообразие списков делает их крайне востребованными элементами любого документа (весьма прискорбно, что CSS обладает более чем скромными возможностями по их форматированию).

Самый простой (в том числе для пользовательских агентов) способ форматирования списка заключается в изменении типа его маркеров. Под *маркером* понимают специальный символ, например жирную точку, отображаемую перед каждым элементом списка. В упорядоченных списках в качестве маркера используются числа, буквы или любые другие символы, обеспечивающие однозначную нумерацию их элементов. Маркеры списка могут представляться даже внешними графическими изображениями. Для управления маркерами в CSS предусмотрено несколько специальных свойств.

Типы списков

Изменение типа списка осуществляется с помощью стилевого свойства `list-style-type`.

list-style-type

Значение	`disc` | `circle` | `square` | `disclosure-open` | `disclosure-closed` | `decimal` | `decimal-leading-zero` | `arabic-indic` | `armenian` | `upper-armenian` | `lower-armenian` | `bengali` | `cambodian` | `khmer` | `cjk-decimal` | `devanagari` | `gujarati` | `gurmukhi` | `georgian` | `hebrew` | `kannada` | `lao` | `malayalam` | `mongolian` | `myanmar` | `oriya` | `persian` | `lower-roman` | `upper-roman` | `tamil` | `telugu` | `thai` | `tibetan` | `lower-alpha` | `lower-latin` | `upper-alpha` | `upper-latin` | `cjk-earthly-branch` | `cjk-heavenly-stem` | `lower-greek` | `hiragana` | `hiragana-iroha` | `katakana` | `katakana-iroha` | `japanese-informal` | `japanese-formal` | `korean-hangul-formal` | `korean-hanja-informal` | `korean-hanja-formal` | `simp-chinese-informal` | `simp-chinese-formal` | `trad-chinese-informal` | `trad-chinese-formal` | `ethiopic-numeric` | *<string>* | `none` | `inherit`
Начальное значение	`disc`
Применяется	Элементы, свойство `display` которых установлено в значение `list-item`
Вычисляется	Согласно определению
Наследуется	Да

Как видите, здесь довольно много ключевых слов, причем это далеко не полный перечень, если учитывать устаревшие ключевые слова, исключенные из текущей спецификации. Некоторые из устаревших значений, например `urdu` и `hangul-consonant`, поддерживались теми или иными браузерами, но широкой поддержки ни одно из таких ключевых слов не получило. В приведенном выше описании свойства `list-style-type` представлены только те ключевые слова, которые поддерживаются всеми без исключения браузерами. Результаты их применения в реальном документе представлены на рис. 15.1.

Как и любые другие свойства форматирования списков, свойство `list-style-type` применяется только к элементам, свойство `display` которых установлено в значение `list-item`. При этом CSS не проводит различий между упорядоченными и

неупорядоченными списками, что позволяет добавлять геометрические маркеры, например кружки, перед элементами нумерованного списка. Согласно определению, по умолчанию свойству `list-style-type` передается значение `disc`. Теоретически маркеры такого типа должны добавляться к элементам любых списков (упорядоченных и неупорядоченных), лишенных объявления типа. Вполне логичное решение, которое тем не менее отдано на откуп разработчикам пользовательских агентов. Ведь даже в случае отсутствия в браузере внутреннего правила, указывающего для списков заранее заданный стиль, например `ol {list-style-type: decimal;}`, его настройки могут запрещать использовать графические маркеры для обозначения элементов упорядоченного списка, и наоборот. Не стоит полагаться на то, что пользовательский агент будет в точности выполнять предписания спецификации CSS.

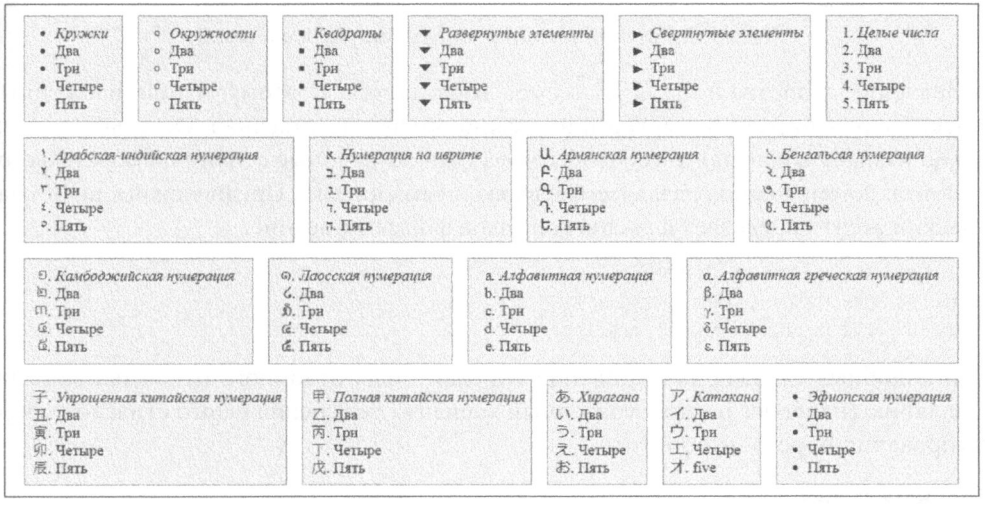


Рис. 15.1. Способы оформления списков

В CSS2.1 любые нераспознаваемые ключевые слова, передаваемые свойству `list-style-type`, обрабатывались пользовательским агентом так же, как и значение `decimal`. Спецификация модуля CSS Lists and Counters лишена столь строгих требований и позволяет представлять их значением `disc` или `none`. (В частности, к концу 2017 года попытка применения к неупорядоченным спискам нумеруемых маркеров в браузере Chrome приводила к сбрасыванию типа списка в значение `none`.)

Исходно значение `none` предназначалось для полного удаления маркеров перед элементами как нумерованного, так и неупорядоченного списка. Несмотря на очистку списка от маркеров, это значение все же обязывает браузер продолжать нумеровать его элементы. Данное утверждение прекрасно иллюстрирует следующий пример, результат выполнения которого показан на рис. 15.2.

```
ol li {list-style-type: decimal;}
li.off {list-style-type: none;}

<ol>
   <li>Первый элемент</li>
```

```
    <li class="off">Второй элемент</li>
    <li>Третий элемент</li>
    <li class="off">Четвертый элемент</li>
    <li>Пятый элемент</li>
</ol>
```

1. Первый элемент
   Второй элемент
3. Третий элемент
   Четвертый элемент
5. Пятый элемент

Рис. 15.2. Частичное отключение маркеров списка

Значение свойства `list-style-type` наследуется, поэтому, чтобы вложенные списки получали разные маркеры, их нужно объявлять отдельно. Отдельные стили нужно назначать еще для того, чтобы заместить встроенные стили пользовательского агента, по умолчанию назначаемые вложенным спискам. Предположим, пользовательский агент определяет для списков такое форматирование.

```
ul {list-style-type: disc;}
ul ul {list-style-type: circle;}
ul ul ul {list-style-type: square;}
```

В подобных случаях для придания спискам вида, отличного от задаваемого по умолчанию (наиболее распространенный вариант), без специального стилевого форматирования просто не обойтись.

Строчные маркеры

В CSS в качестве маркеров списка разрешается использовать строчные значения, что позволяет вводить их непосредственно с клавиатуры. Указанный таким способом текстовый маркер назначается сразу всем элементам одного списка (рис. 15.3), полученного в результате применения следующего стилевого форматирования.

```
.list01 {list-style-type: "%";}
.list02 {list-style-type: "Hi! ";}
.list03 {list-style-type: "†";}
.list04 {list-style-type: "⌘";}
.list05 {list-style-type: "";}
```

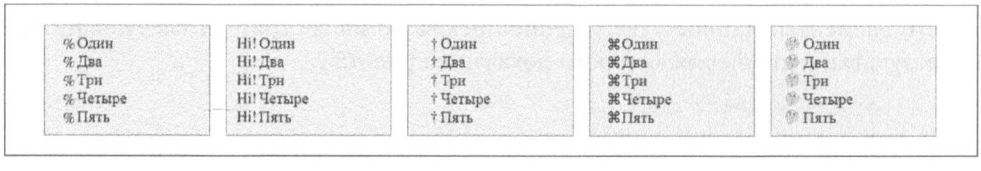


Рис. 15.3. Примеры строчных маркеров

К концу 2017 года текстовые маркеры поддерживались только браузерами семейства Firefox.

Графические маркеры

Во многих ситуациях текстовые маркеры оказываются весьма непрактичными. В CSS для назначения в качестве маркеров списка графических изображений применяется свойство `list-style-image`.

list-style-image

Значение	*<url>* | *<image>* | `none` | `inherit`
Начальное значение	`none`
Применяется	Элементы, свойство `display` которых установлено в значение `list-item`
Вычисляется	Значение *<url>* представлет абсолютный адрес; в противном случае `none`
Наследуется	Да

Ниже приведен простой пример использования этого свойства для назначения графических маркеров элементам неупорядоченного списка.

```
ul li {list-style-image: url(ohio.gif);}
```

Действительно, ничего сложного! В качестве значения свойству передается функция `url()`, в которую подставляется имя или путь размещения файла необходимого графического изображения (рис. 15.4).

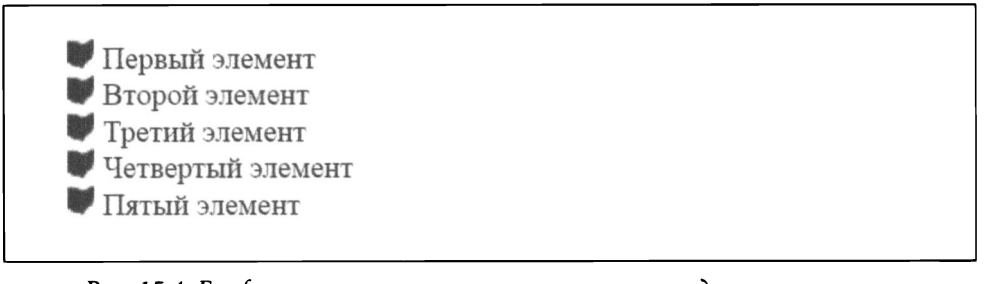


Рис. 15.4. Графические маркеры у элементов неупорядоченного списка

Конечно, к размеру графических изображений, выступающих в качестве маркеров, выдвигаются определенные требования. Если их не соблюсти, то легко получить результат, подобный показанному на рис. 15.5.

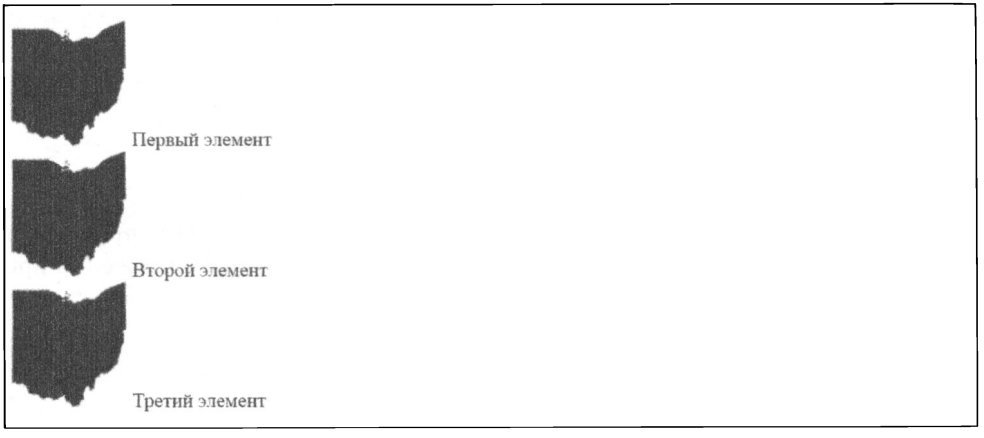


Рис. 15.5. Чрезмерно большие графические маркеры

Графические маркеры лучше всего снабжать резервным типом значков, применяемым в случае сбоев загрузки внешних файлов, их повреждения или несовместимости форматов. Резервная копия графического маркера создается с помощью кода, подобного следующему:

```
ul li {list-style-image: url(ohio.png); list-style-type: square;}
```

Кроме объявления графического маркера, свойство `list-style-image` также применяется для представления значения, задаваемого по умолчанию, ключевым словом `none`. Чрезвычайно полезная возможность, учитывая наследуемость свойства — крайне нежелательного в большинстве случаев.

```
ul {list-style-image: url(ohio.gif); list-style-type: square;}
ul ul {list-style-image: none;}
```

Вложенные списки не обеспечиваются отдельными графическими маркерами, а потому получают квадратные маркеры (`square`), наследуя их от родительского списка (рис. 15.6).

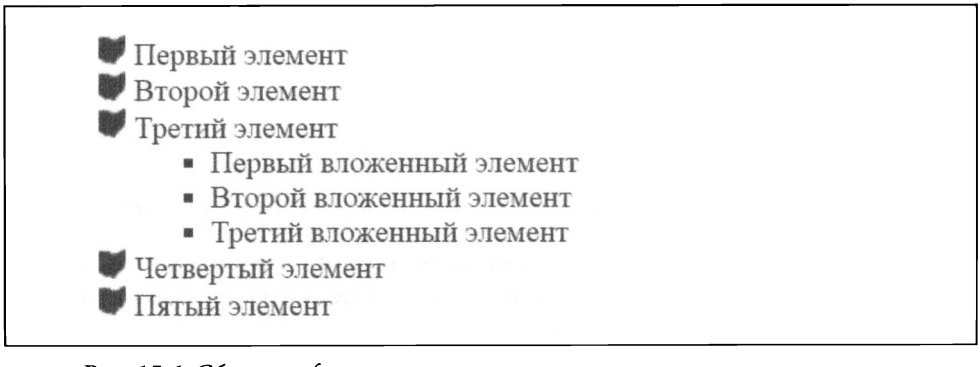


Рис. 15.6. Сброс графических маркеров у элементов вложенного списка

В реальных документах такой сценарий стилевого форматирования списков выполняется далеко не всегда: если свойство `list-style-type` для элемента `ul ul` настраивается пользовательским агентом самостоятельно, то значение `square` вложенным списком наследоваться не будет. Вместо квадратов в начале элементов будут отображаться круглые или любые другие маркеры.

Свойство `list-style-image` принимает любые значения, обозначающие графические изображения, включая градиенты. Например, приведенный ниже код обеспечивает стилевое форматирование списков, показанное на рис. 15.7.

```
.list01 {list-style-image: radial-gradient(closest-side, orange,
   orange 60%, blue 60%, blue 95%, transparent);}
.list02 {list-style-image: linear-gradient(45deg, red, red 50%,
   orange 50%, orange);}
.list03 {list-style-image: repeating-linear-gradient(-45deg, red,
   red 1px, yellow 1px, yellow 3px);}
.list04 {list-style-image: radial-gradient(farthest-side at bottom
   right, lightblue, lightblue 50%, violet, indigo, blue, green,
   yellow, orange, red, lightblue);}
```

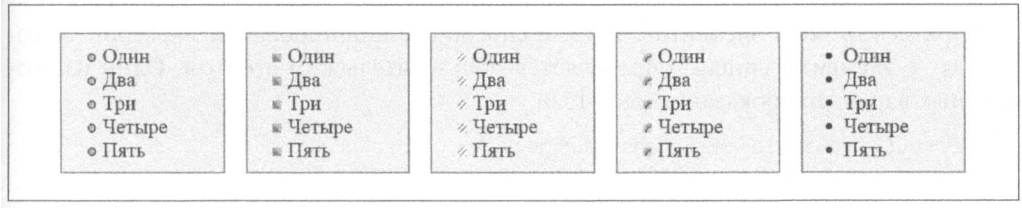


Рис. 15.7. Градиенты в качестве маркеров списка (см. цветные иллюстрации на вебсайте)

Градиентные маркеры имеют существенный недостаток: малый размер. Увеличить его средствами CSS не представляется возможным, и авторам документов приходится довольствоваться тем, что есть. Единственный способ как-то изменить размер такого маркера — увеличить размер шрифта: в современных браузерах они масштабируются вместе.

В CSS предусмотрен прямой способ форматирования маркеров списков — с помощью псевдоэлемента `::marker`, но на начало 2017 года он не поддерживался ни одним из браузеров.

К концу 2017 года градиентные маркеры отображались только браузерами семейства WebKit/Blink.

Позиционирование маркеров списка

Инструменты CSS позволяют выполнять над маркерами еще одну важную операцию: включать или исключать их из содержимого элементов списков. Для этого предназначено свойство `list-style-position`.

list-style-position

Значение	`inside` | `outside` | `inherit`
Начальное значение	`outside`
Применяется	Элементы, свойство `display` которых установлено в значение `list-item`
Вычисляется	Согласно определению
Наследуется	Да

Если передать этому свойству значение `outside` (по умолчанию), то маркеры будут размещаться относительно элементов списка так, как предписывалось в самой ранней спецификации CSS. Для включения их в содержимое элементов свойству `list-style-position` нужно задать значение `inside`. В результате маркеры будут "внедряться" в текст элементов. Точный способ позиционирования маркеров, включенных в элементы списка, определяется пользовательским агентом. Один из возможных вариантов показан на рис. 15.8.

```
li.first {list-style-position: inside;}
li.second {list-style-position: outside;}
```

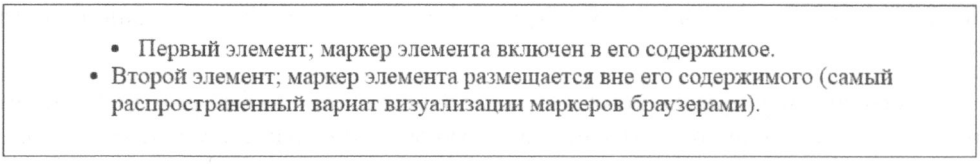


Рис. 15.8. Размещение маркеров вне и внутри содержимого элементов списка

В реальных документах маркеры, позиционируемые с помощью ключевого слова `inside`, обрабатываются браузерами как строчные элементы, размещенные в начале содержимого элементов списка. При этом они не являются таковыми по своей сути, поскольку не подлежат стилевому форматированию отдельно от содержимого элементов списка, — по крайней мере, до тех пор пока оно не будет заключено в другие элементы, например `span`. Таким образом, название значения носит условный характер.

Свойство форматирования списков общего назначения

Код стилевого оформления списков можно существенно сократить, заменив три свойства всего одним: `list-style`.

list-style	
Значение	[*<list-style-type>* || *<list-style-image>* || *<list-style-position>*] | `inherit`
Начальное значение	См. описания индивидуальных свойств
Применяется	Элементы, свойство `display` которых установлено в значение `list-item`
Вычисляется	См. описания индивидуальных свойств
Наследуется	Да

Приведем пример:

```
li {list-style: url(ohio.gif) square inside;}
```

Как показано на рис. 15.9, к элементам списка применяется форматирование, определяемое всеми тремя значениями.

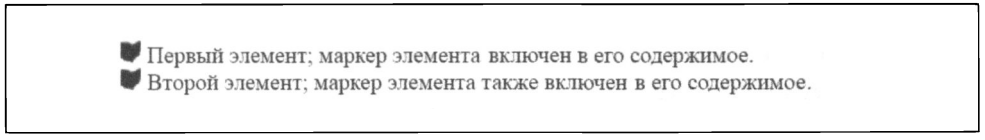


Рис. 15.9. Одновременная установка сразу нескольких параметров форматирования списка

Значения передаются свойству `list-style` в любом количестве и последовательности. При передаче ему хотя бы одного значения остальные устанавливаются по умолчанию. Следовательно, приведенные ниже правила приводят к одинаковому стилевому форматированию элементов списка.

```
li.norm {list-style: url(img42.gif);}
li.odd {list-style: url(img42.gif) disc outside;}
```

При этом каждое следующее свойство `list-style` замещает такое же предыдущее свойство, как показано в следующем коде.

```
li {list-style-type: square;}
li {list-style: url(img42.gif);}
li {list-style: url(img42.gif) disc outside;}
```

Конечный результат будет равнозначен показанному на рис. 15.9, поскольку неявно заданное значение `disc` свойства `list-style-type` замещает ранее объявленное значение `square`, подобно тому как явно заданное значение `disc` замещает это же значение во втором правиле.

Верстка списков

Определившись со свойствами, отвечающими за форматирование маркеров списков, можно приступать к изучению способов их представления браузерами. Для

начала рассмотрим простой одноуровневый список, включающий всего три элемента и лишенный каких бы то ни было маркеров (рис. 15.10).

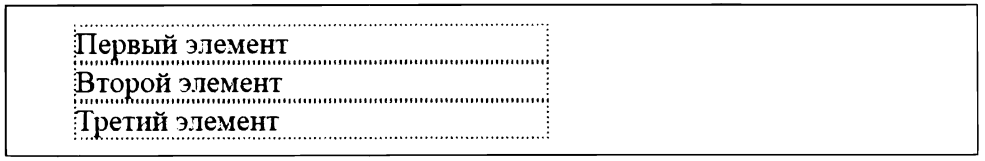


Рис. 15.10. Простой список, состоящий из трех элементов

Пунктирные границы, в которые заключены элементы списка, указывают на принадлежность их к объектам блочного типа. Это утверждение также подтверждается тем фактом, что, согласно определению, свойство `list-item` создает элементы исключительно блочного уровня. При добавлении к списку маркеров он будет выглядеть так, как показано на рис. 15.11.

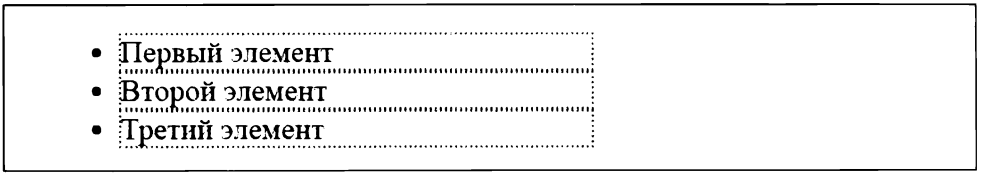


Рис. 15.11. Элементы списка с маркерами

Расстояние между маркерами и элементами списка не поддается установке и изменению с помощью инструментов CSS.

Маркеры, размещенные вне содержимого элементов списка, не оказывают влияния на положение остальных элементов документа, равно как и на позиционирование самих элементов списка. Они отстоят от содержимого элементов на строго заданное расстояние, привязываясь к его левому краю. Маркер неотрывно следует за содержимым элемента, в какое бы место документа тот ни смещался в процессе верстки. В этом смысле маркеры ведут себя так, как если бы они выступали абсолютно позиционируемыми объектами в пределах своих элементов списка. (В общем синтаксисе CSS их поведение можно описать правилом наподобие `position: absolute; left: -1.5em;`.) При включении маркеров в элементы списка они форматируются так же, как и любое другое его содержимое.

Весь список заключается в отдельный контейнер. В частности, список, показанный на рис. 15.11, заключен в контейнер элемента `ul`, а не `ol`. На рис. 15.12 он обведен пунктирной линией.

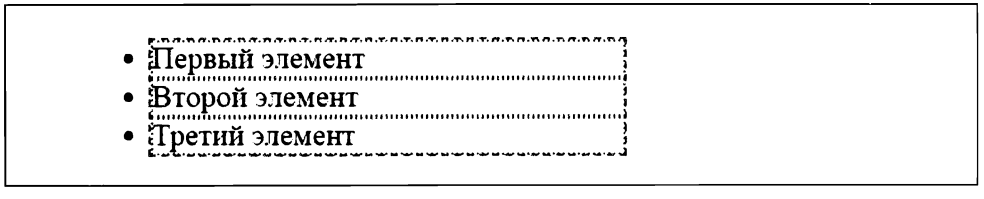


Рис. 15.12. Границы списка

Как и отдельные элементы, весь список заключается в контейнер блочного типа. Он включает в себя контейнеры всех элементов списка, дочерних по отношению к самому списку. При этом маркеры располагаются не только вне содержимого отдельных элементов списка, но и за пределами содержимого их родительского объекта, т.е. контейнера списка. Для изменения отступа маркеров от прилегающего к ним края контейнера списка приходится идти на определенные ухищрения.

На момент написания книги для изменения интервала между маркерами и элементами списка его контейнер снабжается дополнительным отступом или полем, как показано в следующем коде:

```
ul, ol {margin-left: 40px;}
```

Данное правило применимо в Internet Explorer и Opera. При этом браузеры, основанные на движке Gecko, требуют несколько иного подхода.

```
ul, ol {padding-left: 40px;}
```

Оба правила абсолютно корректные, но применение их в неправильном контексте может вызвать несоответствие в позиционировании элементов списка. Отличие в подходах проиллюстрировано на рис. 15.13.

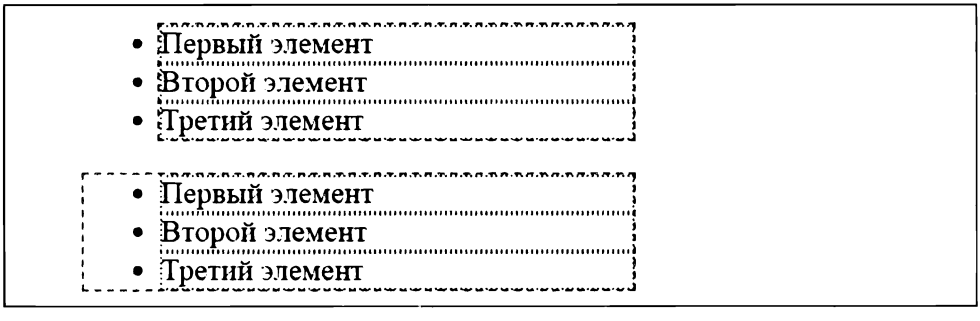


Рис. 15.13. Отступы и поля как регуляторы отступа у элемента списка

Расстояние `40px` унаследовано от самых первых браузеров, отступы в которых указывались в пикселях. (Отступами такой же ширины "отбивались" символы кавычек.) В качестве альтернативного также применялось значение `2.5em`, обеспечивающее автоматическую коррекцию ширины отступа при изменении размера символов.

Таким образом, чтобы правильно позиционировать маркеры относительно элементов списка в любых браузерах, элементы списка нужно снабжать и полями, и отступами. В частности, приведенное ниже правило устанавливает интервал до маркеров элементов списка через отступы.

```
ul {margin-left: 0; padding-left: 1em;}
```

Следующее правило позволяет решить эту же задачу, используя поля:

```
ul {margin-left: 1em; padding-left: 0;}
```

В любом случае положение маркеров определяется относительно содержимого элементов списка. При неправильном форматировании легко получить висячие маркеры, выступающие за область содержимого основного текста документа или даже окна браузера. Такой эффект чаще всего наблюдается при использовании графических или текстовых маркеров большого размера (рис. 15.14).

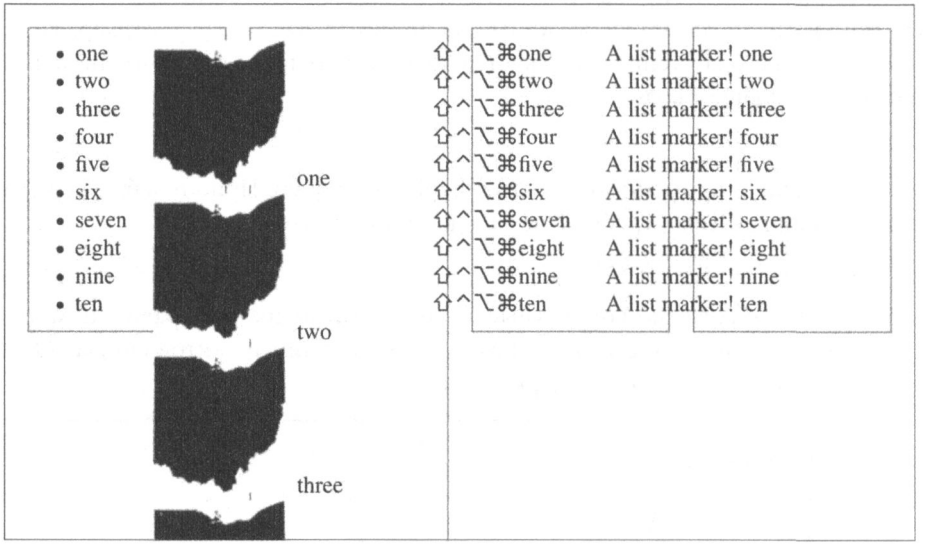


Рис. 15.14. Позиционирование списков с маркерами большого размера

Свойства позиционирования маркеров списка

Наличие инструментов настройки интервала между маркерами и содержимым элементов списка — одна из основных потребностей веб-дизайна, до сих пор не реализованных в CSS. Первые попытки исправить ситуацию были предприняты в спецификации CSS2, которая пополнилась свойством `marker-offset` и значением `marker` свойства `display`. Но на этапе внедрения их поддержки в браузерах возникли серьезные трудности, и указанные новшества были отменены спецификацией CSS2.1.

К началу 2017 года рабочий проект модуля CSS3 Lists and Counters предполагал более совершенный способ определения интервала до маркеров списка, заключающийся в использовании псевдоэлемента `::marker`. На этапе получения технологией статуса рекомендуемой обозначенная выше задача должна была решаться с помощью такого кода:

```
li::marker {margin-right: 0.125em; color: goldenrod;}
```

Генерируемое содержимое

В CSS предусмотрена возможность генерирования содержимого, которое создается непосредственно в коде таблицы стилей, а не определяется содержимым документа или стилевым форматированием.

К генерируемому содержимому относятся в том числе и маркеры списков. Легко заметить, что они не определяются разметкой документа и не вводятся авторами документов, подобно тексту или остальному его содержимому. Отображение их в документе выполняется автоматически согласно внутренним инструкциям пользовательского агента. Элементы неупорядоченного списка предваряются неким символом или графическим объектом, например кружком или квадратом. В упорядоченных списках маркеры представлены счетчиками, значение которых увеличивается на единицу в каждом последующем элементе. (Как было показано в предыдущем разделе, счетчики также можно представлять графическими изображениями и специальными символами.)

Перед тем как изучать способы настройки и изменения значений счетчиков упорядоченных списков (и объектов некоторых других типов), ознакомимся с общими принципами создания генерируемого содержимого.

Вставка генерируемого содержимого

Для добавления в документ генерируемого содержимого применяются псевдоэлементы `::before` и `::after`, указывающие место его вставки относительно целевого элемента. О роли свойства `content` в решении этой задачи речь пойдет в следующем разделе.

Например, перед вами может стоять задача добавить текст "(link)" перед каждой гиперссылкой документа, отправляемого на печать. Для решения такой задачи достаточно применить следующее форматирование (рис. 15.15).

```
a[href]::before {content: "(link)";}
```

(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. 15.15. Текстовое генерируемое содержимое

Обратите внимание на отсутствие интервала между генерируемым содержимым и целевыми элементами. Для его добавления соответствующие пробелы нужно включить в текст генерируемого содержимого. Приведенный ниже код включает необходимое исправление и разделяет генерируемое и обычное содержимое.

```
a[href]::before {content: "(link)";}
```

Различие в правилах незначительно, чего не скажешь о получаемых результатах.

Подобным образом специальными значками можно снабжать гиперссылки на PDF-документы. Для того чтобы добавить их перед такими гиперссылками, применяется следующий CSS-код:

```
a.pdf-doc::after {content: url(pdf-doc-icon.gif);}
```

Кроме того, генерируемое содержимое можно использовать для форматирования самих гиперссылок, в частности, для добавления к ним рамки. Вот как эта задача решается с помощью CSS:

```
a.pdf-doc {border: 1px solid gray;}
```

Результат применения двух последних правил приведен на рис. 15.16.

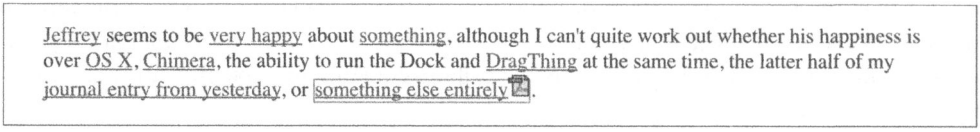


Рис. 15.16. Генерирование значка для гиперссылки

Обратите внимание на то, что в рамку заключена не только гиперссылка, но и генерируемый для нее значок. Это обусловлено размещением генерируемого содержимого внутри, а не вне контейнера элемента. В CSS2.1 генерируемое содержимое (за исключением маркеров списков) не разрешалось размещать вне элементов, к которым оно относится.

Легко заметить, что в CSS2 и CSS2.1 позиционирование элементов, снабженных генерируемым содержимым, возможно только при запрете выравнивания и изменения положения псевдоэлементов `::before` и `::after`. К ним также запрещено применять табличные свойства и свойства управления списками. Кроме того, на их использование накладываются еще два важных ограничения:

- при добавлении псевдоэлементов `::before` и `::after` к блочному элементу его свойство `display` может иметь только значения `none`, `inline`, `block` и `marker`;
- при добавлении псевдоэлементов `::before` и `::after` к строчному элементу его свойство `display` может иметь только значения `none` и `inline`.

Рассмотрим такой пример:

```
em::after {content: " (!) "; display: block;}
```

Так как `em` относится к строчным элементам, генерируемое содержимое не может представляться блочным типом данных. Таким образом, значение `block`, назначенное его свойству `display`, сбрасывается в значение `inline`. И наоборот, в следующем примере генерируемое содержимое представляется контейнером блочного типа, поскольку добавляется к блочному элементу:

```
h1::before {content: "Новый раздел"; display: block; color: gray;}
```

Результат выполнения последнего кода показан на рис. 15.17.

Рис. 15.17. Генерируемое содержимое блочного уровня

Генерируемое содержимое характеризуется еще одной прмечательной особенностью: оно наследует значения свойств, объявленных для целевых элементов. Таким образом, при выполнении следующего кода генерируемый текст, как и основное содержимое абзаца, будет окрашиваться зеленым цветом.

```
p {color: green;}
p::before {content: "::: ";}
```

Для назначения генерируемому тексту фиолетового оттенка в таблицу стилей нужно добавить такое правило:

```
p::before {content: "::: "; color: purple;}
```

Разумеется, псевдоэлементы `::before` и `::after` наследуют значения далеко не всех заданных в CSS-коде свойств, а только подлежащих наследованию. Такое ограничение само по себе мало что означает и в каждом конкретном случае рассматривается отдельно. Рассмотрим пример.

```
h1 {border-top: 3px solid black; padding-top: 0.25em;}
h1::before {content: "Новый раздел"; display: block; color: gray;
   border-bottom: 1px dotted black; margin-bottom: 0.5em;}
```

Так как генерируемое содержимое включено в элемент `h1`, оно будет располагаться сразу же над его верхней границей — непосредственно в области верхнего поля (рис. 15.18).

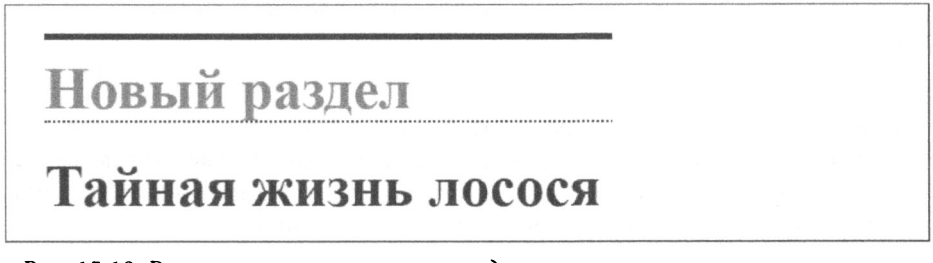


Рис. 15.18. Размещение генерируемого содержимого с учетом типа элемента

Содержимое целевого элемента будет смещаться вниз на величину нижнего отступа генерируемого содержимого, представленного контейнером блочного типа: `0,5em`. Как бы там ни было, генерируемое содержимое и содержимое целевого элемента оказываются разделенными, как и предполагалось исходно. Визуализация данных в таком виде осуществляется благодаря применению к генерируемому содержимому объявления `display: block`. Как только оно будет заменено в CSS-коде на `display: inline`, оформление документа изменится на показанное на рис. 15.19.

```
h1 {border-top: 3px solid black; padding-top: 0.25em;}
h1::before {content: "Новый раздел"; display: inline; color: gray;
   border-bottom: 1px dotted black; margin-bottom: 0.5em;}
```

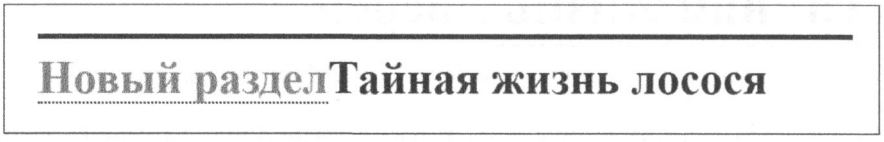


Рис. 15.19. Результат изменения типа генерируемого содержимого

Обратите внимание на наличие верхних поля и границы у целевого элемента, а также нижнего отступа — у генерируемого содержимого, которое теперь не изменяет высоту целевого элемента.

Разобравшись с базовыми принципами создания генерируемого содержимого, рассмотрим более сложные методы его получения в CSS.

Свойство `content`

Чтобы активно использовать генерируемое содержимое, нужно иметь надежные средства его объявления в документе. Как известно, для этих целей применяется свойство `content`, обладающее существенно бо́льшими функциональными возможностями, чем стилевые инструменты, описанные в предыдущих главах.

content

Значение	`normal` | [*<string>* | *<uri>* | *<counter>* | `attr(`*<identifier>*`+)+` | `open-quote` | `close-quote` | `no-open-quote` | `no-close-quote`]+ | `inherit`
Начальное значение	`normal`
Применяется	Псевдоэлементы `::before` и `::after`
Вычисляется	Значение *<url>* представляется абсолютным адресом, а значение `attr()` — строкой; в остальных случаях согласно определению
Наследуется	Нет

Значения *<string>* и *<uri>* рассматривались в предыдущих главах, а счетчики, представляемые значением *<counter>*, будут описаны позже. Тем не менее, перед тем как перейти к изучению значений, представленных кавычками и функцией `attr()`, рассмотрим все возможности первых двух значений.

Строковые значения представляют преимущественно текст, допускающий включение в него данных других типов. В следующем примере генерируемое содержимое воспроизводится в документе в исходном виде (рис. 15.20):

```
h2::before {content: "<em>&para;</em> "; color: gray;}
```

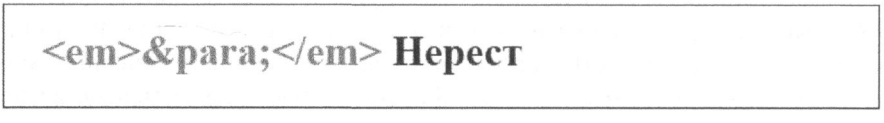


Рис. 15.20. Дословное воспроизведение генерируемого содержимого

Данный пример показателен в первую очередь тем, что демонстрирует несостоятельность использования тега
 для добавления в генерируемое содержимое новой строки (символа возврата каретки). Вместо него применяется строка \A, обозначающая в CSS символ перевода строки (шестнадцатеричное число A является кодом символа перевода строки в Unicode и символа новой строки в CSS).

Подобным образом для предотвращения разрыва генерируемого содержимого, представленного длинной строкой, в соответствующие ее места нужно добавить символ \. Такой способ упорядочения длинных строк проиллюстрирован на рис. 15.21.

```
h2::before {content: "Этот текст вставляется перед элементом h2 \
исключительно в демонстрационных целях. Несмотря на значительную \
длину строки, он прекрасно различим в документе. "; color: gray;}
```

Этот текст вставляется перед элементом h2 исключительно в демонстрационных целях. Несмотря на значительную длину строки, он прекрасно различим в документе. Нерест

Рис. 15.21. Добавление и блокировка разрывов строки

Для предотвращения разрыва строки также применяется символ, обозначаемый в Unicode шестнадцатеричным значением \00AB.

На момент написания книги символы \A и \00AB не нашли широкой поддержки в пользовательских агентах — даже в тех, которые обеспечивают работу с генерируемым содержимым.

Значения *<uri>* представляют в генерируемом содержимом внешние ресурсы (графические изображения, звуковые клипы, видеоролики и т.п.), которые должны отображаться в определенных местах документа. Если указанный тип данных не поддерживается пользовательским агентом или не может быть воспроизведен по некой важной причине (браузер не поддерживает предложенный графический формат или не может вывести на печать видеоролик), то внешний файл игнорируется и полностью исключается из генерируемого содержимого.

Вставка значений атрибутов

В отдельных случаях может потребоваться включить в текст документа значение одного из атрибутов. Самый распространенный пример такого поведения — добавление адреса гиперссылки, представленного атрибутом `href`, после нее самой:

```
a[href]::after {content: attr(href);}
```

Как и прежде, задача решается вставкой в документ генерируемого содержимого (рис. 15.22) — в его объявление добавляется строка, представляющая значение требуемого атрибута:

```
a[href]::after {content: " [" attr(href) "]";}
```

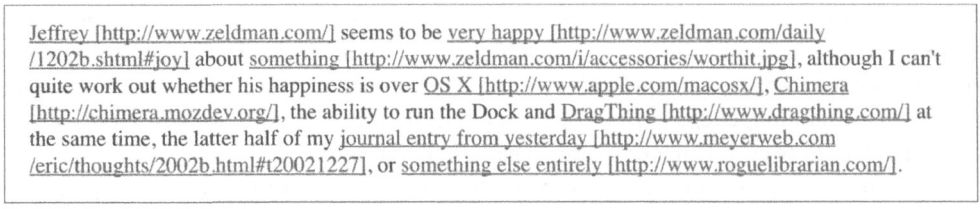


Рис. 15.22. Добавление URL-адреса на страницу

Эта особенность открывает возможности по отображению в документе данных, извлеченных из таблицы стилей. Генерируемым содержимым могут представляться значения абсолютно любых атрибутов: `alt text`, `class`, `id` и др. Для вставки значений атрибутов в содержащий текст последний необходимо заключать в кавычки, как показано ниже.

```
blockquote::after {content: "(" attr(cite) ")"; display: block;
   text-align: right; font-style: italic;}
```

В продолжение рассмотрим, насколько сложным будет стилевое правило, которое отвечает за отображение в документе легенды, описывающей принципы цветовой идентификации текстовой информации.

```
body::before {content: "Text: " attr(text) " | Link: " attr(link)
   " | Visited: " attr(vlink) " | Active: " attr(alink);
   display: block; padding: 0.33em; border: 1px solid;
   text-align: center; color: red;}
```

Учтите: в отсутствие обозначенного в правиле аргумента его значение будет представляться в документе пустой строкой, что продемонстрировано на рис. 15.23, полученном после удаления атрибута `alink` из элемента `body` предыдущего примера.

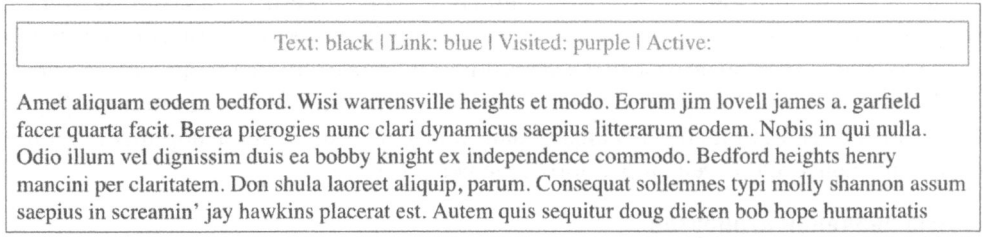


Рис. 15.23. Пропуск значений отсутствующих атрибутов

Строка "Active: " (включая конечный пробел), несмотря на обязательное включение в документ, не сопровождается целевым значением. Это задел на будущее: она точно понадобится при добавлении соответствующего атрибута в таблицу стилей.

В CSS2.*x* значение атрибута перед отображением в документе не обрабатывается синтаксическим анализатором. Таким образом, любой включенный в него код разметки и специальные символы будут отображаться в документе в исходном виде, а не представляться вычисляемыми значениями.

Генерирование кавычек

К отдельному типу генерируемого содержимого относятся кавычки. Для управления их поведением в CSS2.*x* применяются специальные инструменты: значения `open-quote` и `close-quote` свойства `content`, а также отдельное свойство `quotes`.

quotes

Значение	[*<string>* *<string>*]+ | `none` | `inherit`
Начальное значение	Определяется пользовательским агентом
Применяется	Все элементы
Вычисляется	Согласно определению
Наследуется	Да

При изучении значений этого свойства становится очевидным, что все они (за исключением ключевых слов `none` и `inherit`) представляются *парными* строками. Первая из таких парных строк обозначает символ открывающих кавычек, а вторая — закрывающих. Таким образом, только первая из следующих двух строк CSS-кода верная.

```
quotes: '"' "'"; /* правильно */
quotes: '"'; /* неправильно */
```

Кроме того, в первом правиле продемонстрирован правильный способ заключения строк в кавычки. В первой строке символ двойных кавычек заключен в пару одинарных кавычек, а во второй строке символ одинарных кавычек заключается в пару двойных кавычек.

Рассмотрим простой пример. Предположим, требуется создать XML-документ, хранящий список известных цитат. Пусть каждая цитата представляется в нем такой разметкой.

```
<quotation>
   <quote>Ненавижу цитаты!</quote>
   <quotee>Ралф Уолдо Эмерсон</quotee>
</quotation>
```

Для представления данных документа (рис. 15.24) в удобочитаемом виде к нему применяется несколько стилевых правил.

```
quotation {display: block;}
quote {quotes: '"' '"';}
quote::before {content: open-quote;}
quote::after {content: close-quote;}
quotee::before {content: " (";}
quotee::after {content: ")";}
```

"Ненавижу цитаты!" (Ралф Уолдо Эмерсон)

Рис. 15.24. Добавление в документ кавычек и другого содержимого

Значения `open-quote` и `close-quote` применяются для обозначения символов кавычек (они могут сильно отличаться в разных языках), генерируемых в тексте документа. В данном примере указываются символы, в которые заключается каждый из элементов. Двойные кавычки применяются для обозначения цитаты, но не ее автора.

Свойство `quotes` допускает установку произвольного количества уровней вложения кавычек. Например, в английском языке считается общепринятым добавлять фразы, выделяемые одинарными кавычками, в текстовые фрагменты, заключаемые в двойные кавычки. В CSS такой порядок можно воссоздать с помощью кода, подобного приведенному ниже.

```
quotation {display: block;}
quote {quotes: '\201C' '\201D' '\2018' '\2019';}
quote::before, q::before{content: open-quote;}
quote::after, q::after {content: close-quote;}
```

Результат применения такого стилевого форматирования к следующему фрагменту документа показан на рис. 15.25.

```
<quotation>
<quote> И сказал Бог: <q>Да будет свет!</q> И стал свет. И увидел
Бог свет, что он хорош, и отделил Бог свет от тьмы.</quote>
</quotation>
```

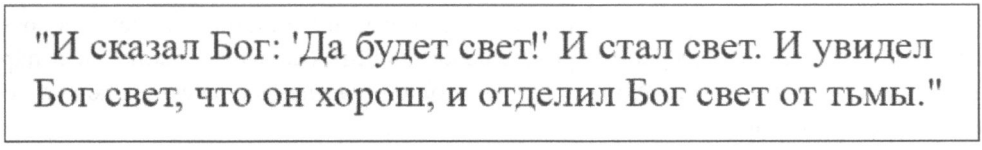


Рис. 15.25. Отображение вложенных кавычек

Если уровней вложения кавычек больше, чем их пар, то на последнем уровне вложения будут применяться такие же кавычки, как на предыдущем уровне. В частности, применение следующего правила к документу, показанному на рис. 15.25, приведет к использованию двойных кавычек как на внутреннем уровне вложения, так и на внешнем:

```
quote {quotes: '\201C' '\201D';}
```

В предыдущем правиле символы фигурных кавычек обозначены шестнадцатеричными значениями. При использовании кодировки UTF-8 (наиболее распространенный вариант) шестнадцатеричные значения можно заменить в CSS-коде символами фигурных кавычек, как в предыдущем примере.

Возможность генерирования кавычек позволяет получить в документах специальный типографский эффект. В английском языке при последовательном размещении в документе сразу нескольких абзацев, заключаемых в кавычки, закрывающие кавычки отображаются только в последнем из них, а во всех предыдущих — упускаются.

Данное орфографическое требование можно легко воссоздать с помощью такой стилевой конструкции.

```
blockquote {quotes: '"' '"' "'" "'" '"' '"';}
blockquote p::before {content: open-quote;}
blockquote p::after {content: no-close-quote;}
```

В результате применения такого форматирования открывающими двойными кавычками будут снабжаться все текстовые абзацы, включая последний. Чтобы добавить закрывающие двойные кавычки только к последнему абзацу, его нужно снабдить псевдоэлементом `::after`, отнеся к отдельному классу и объявив для него свойство `content` со значением `close-quote`.

Такой подход позволяет понизить уровень вложения кавычек без генерирования дополнительных символов. Кроме того, он устраняет необходимость в поочередном применении разнотипных символов кавычек. Ключевое слово `no-close-quote` закрывает уровень вложенности без использования кавычек. Таким образом, каждый следующий абзац начинается с того же уровня вложения, что и предыдущий.

Это очень важный момент, описанный в спецификации CSS 2.1 следующим образом:

> "Глубина вложения кавычек не зависит от уровня вложения элемента и расположения его в объектной модели документа".

Иными словами, каждый уровень вложения кавычек устанавливается исключительно в контексте своего элемента, а закрывается общим ключевым словом `close-quote`.

Чтобы обеспечить целостность стилевой структуры, в спецификацию включено значение `no-open-quote`, по действию прямо противоположное ключевому слову `no-close-quote`. Оно увеличивает уровень вложения кавычек без генерирования их самих в элементе.

Счетчики

Со счетчиками вы уже знакомы: они применяются в качестве маркеров элементов упорядоченных списков. Спецификация CSS1 была полностью лишена инструментов форматирования счетчиков — эта задача всецело возлагалась на HTML, который на тот момент прекрасно справлялся с ее выполнением. Острая необходимость в стилевых инструментах управления счетчиками возникла с появлением технологии XML. Как ни странно, спецификация CSS2 также была лишена даже простейших инструментов генерирования счетчиков, равнозначных поддерживаемым в HTML. На сегодняшний день создание и настройка счетчиков в CSS выполняется с помощью двух стилевых свойств и двух значений свойства `content`. Они позволяют генерировать значения счетчиков для элементов списка произвольного уровня вложения, в том числе представляющие одновременно несколько систем нумерации (например, формата VII.2.c.).

Сброс и шаг счетчика

Базовая настройка счетчика сводится к установке его начального значения и приращения (шага). Первая задача решается с помощью стилевого свойства `counter-reset`.

counter-reset

Значение	[*<identifier>* *<integer>?*]+ | `none` | `inherit`
Начальное значение	Определяется пользовательским агентом
Применяется	Все элементы
Вычисляется	Согласно определению
Наследуется	Нет

Значение *`<identifier>`* представляет идентификатор счетчика, определяемый автором документа. Например, счетчики подразделов могут снабжаться идентификатором `subsection`, `subsec`, `ss` или даже `bob`. Чаще всего идентификатор назначается при сбросе (или приращении) счетчика. В частности, после сброса счетчика с помощью следующего правила к элементу `h1` применяется идентификатор `chapter`:

```
h1 {counter-reset: chapter;}
```

Заметьте, что по умолчанию счетчик сбрасывается в нулевое значение. Чтобы сбросить его в любое другое значение, соответствующее число нужно ввести после имени идентификатора:

```
h1#ch4 {counter-reset: Chapter 4;}
```

Более того, в одном правиле можно сбросить счетчики сразу нескольких идентификаторов. Все идентификаторы, лишенные целочисленных значений после названия, сбрасываются в нулевое значение.

```
h1 {counter-reset: Chapter 4 section -1 subsec figure 1;}
   /* subsec сбрасывается в 0 */
```

В последнем примере показано, что значения счетчиков могут представляться отрицательными числами. Например, ничто не запрещает начать отсчет элементов списка со значения `-32768`.

В спецификации CSS нет инструкций по обработке отрицательных значений у нечисловых счетчиков. Например, пользовательский агент должен самостоятельно решать, как поступать при установке счетчика `upper-alpha` в значение `-5`.

Сбросив счетчик, нужно определиться с шагом его приращения. Если этого не сделать, то все элементы последовательности будут маркироваться одним и тем же числовым значением. В CSS решение этой задачи возлагается на свойство с красноречивым названием `counter-increment`.

counter-increment	
Значение	[*<identifier>* *<integer>*?]+ | `none` | `inherit`
Начальное значение	Определяется пользовательским агентом
Применяется	Все элементы
Вычисляется	Согласно определению
Наследуется	Нет

Это свойство, как и свойство `counter-reset`, принимает пары значений "идентификатор-целое число" — последнее может представляться положительным, отрицательным или даже нулевым значением. При этом свойства отличаются значениями по умолчанию — свойство `counter-increment` сбрасывается в значение `1`, а не `0`.

В качестве примера рассмотрим, каким образом пользовательский агент воссоздает стандартную систему целочисленной нумерации элементов списка: 1, 2, 3 и т.д.

```
ol {counter-reset: ordered;} /* по умолчанию сбрасывается в 0 */
ol li {counter-increment: ordered;} /* по умолчанию сбрасывается в 1 */
```

С другой стороны, вам может понадобиться снабдить список обратной отрицательной нумерацией, в которой отсчет ведется от нуля, но в направлении уменьшения отрицательных целочисленных значений. Для того чтобы получить такой счетчик, в предыдущий код необходимо внести совсем незначительные изменения.

```
ol {counter-reset: ordered;} /* по умолчанию сбрасывается в 0 */
ol li {counter-increment: ordered -1;}
```

В результате применения такого стилевого форматирования элементы списка будут нумероваться как –1, –2, –3 и т.д. Если в последнем коде значение –1 заменить на –2, то элементы будут нумероваться только парными отрицательными числами: –2, –4, –6 и т.д.

Применение счетчика

Счетчик, который получен с помощью свойств, описанных в предыдущих разделах, в документе назначается целевым элементам с помощью свойства `content`. Значение последнего представляется функцией `counter()` или `counters()`, получающей все необходимые настройки. Рассмотрим, каким образом эта операция выполняется по отношению к списку, размеченному с помощью XML.

```
<list type="ordered">
   <item>Первый элемент</item>
   <item>Второй элемент</item>
   <item>Третий элемент</item>
</list>
```

Для получения конечного результата (рис. 15.26) к указанному списку применяется стилевое форматирование, представленное таким CSS-кодом.

```
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;}
```

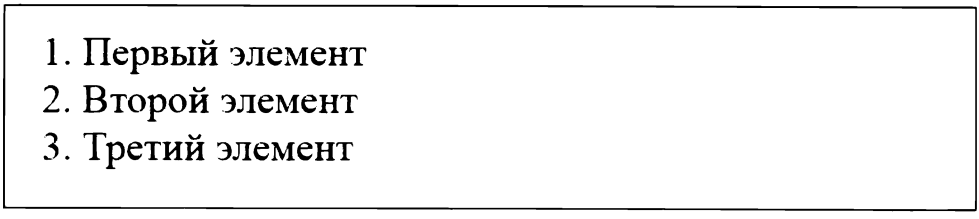


Рис. 15.26. Простая нумерация элементов

Легко заметить, что в этом примере генерируемое содержимое, как и полагается счетчику, располагается в начале каждого элемента списка. Такого же результата можно добиться, применив к HTML-списку объявление `list-style-position: inside`.

Заметьте, что счетчик добавляется к регулярным элементам `item`, представленным контейнерами блочного типа. Следовательно, счетчики могут генерироваться для произвольных элементов, а не только для тех, свойство `display` которых представлено значением `list-item`.

```
h1 {counter-reset: section subsec; counter-increment: chapter;}
h1::before {content: counter(chapter) ". ";}
h2 {counter-reset: subsec; counter-increment: section;}
h2::before {content: counter(chapter)"." counter(section) ". ";}
h3 {counter-increment: subsec;}
h3::before {content: counter(chapter) "." counter(section) "."
   counter(subsec) ". ";}
```

Результат применения приведенной выше таблицы стилей к реальному документу показан на рис. 15.27.

В данном примере весьма наглядно проиллюстрированы особенности назначения и приращения значений счетчиков. Обратите внимание на то, как происходит сброс счетчиков для элементов, тогда как реальные значения счетчиков генерируются с помощью псевдоэлементов `::before`. Попытка назначения счетчиков непосредственно псевдоэлементам обречена на неудачу — в подобном случае их значения будут представляться одними только нулями. А вот свойство приращения счетчика можно применять как к нумеруемым элементам, так и к псевдоэлементам, отвечающим за генерирование значений.

Также заметьте, что элементам `h1` назначен идентификатор `chapter`, сбрасываемый в нулевое значение и снабженный начальной строкой "1.". При этом увеличение значения счетчика для текущего элемента выполняется *перед* его отображением в документе. Подобным образом сброс счетчика для текущего элемента осуществляется перед его представлением в документе. Рассмотрим такой пример.

```
h1::before, h2::before, h3::before {
   content: counter(chapter) "." counter(section) "."
      counter(subsec) ". ";}
```

```
h1 {counter-reset: section subsec;
   counter-increment: chapter;}
```

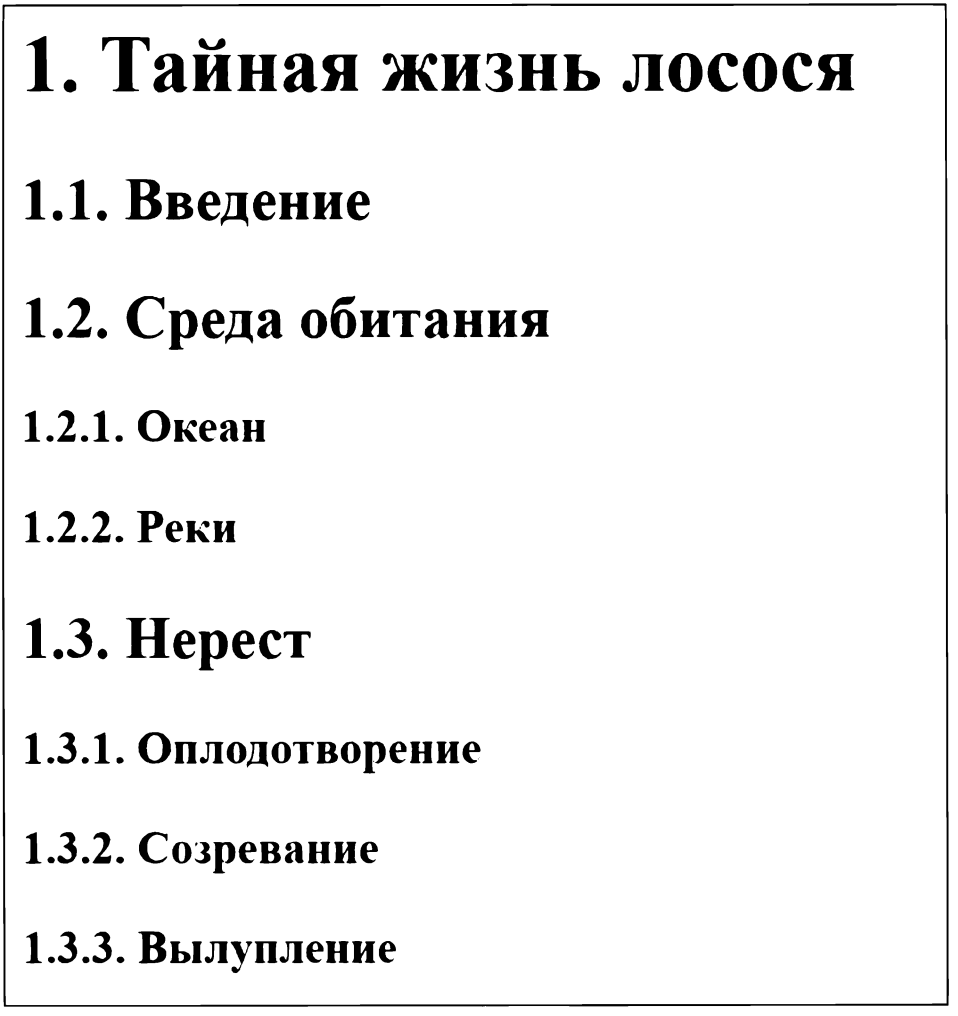


Рис. 15.27. Добавление счетчиков к заголовкам разных уровней

Согласно приведенным выше правилам, первый элемент `h1` документа предваряется строкой "1.0.0", поскольку счетчики `section` и `subsec` для него только сбрасываются, но не прирастают. Для того чтобы получить нулевое значение и для первого счетчика, его нужно сбросить в значение `-1`, а не `0`, как показано ниже.

```
body {counter-reset: chapter -1;}
h1::before {counter-increment: chapter;
   content: counter(chapter) ". ";}
```

Счетчики можно использовать для совершенно неожиданных целей. Изучим следующий фрагмент XML-документа.

```
<code type="BASIC">
   <line>PRINT "Hello world!"</line>
   <line>REM This is what the kids are calling a "comment"</line>
   <line>GOTO 10</line>
</code>
```

Применив к нему следующие стилевые правила, данные документа можно представить в виде листинга программы, написанной 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) ": ";}
```

Способ (стиль) нумерации элементов списка лучше всего задавать в функции `counter()`. Он определяется одним из значений свойства `list-style-type`, вводимого через запятую после названия идентификатора счетчика. Такой подход использован в следующем примере, результат применения которого к уже известному вам оглавлению реального документа показан на рис. 15.28.

```
h1 {counter-reset: section subsec; counter-increment: chapter;}
h1::before {content: counter(chapter,upper-alpha) ". ";}
h2 {counter-reset: subsec; counter-increment: section;}
h2::before {content: counter(chapter,upper-alpha)"."
   counter(section) ". ";}
h3 {counter-increment: subsec;}
h3::before {content: counter(chapter,upper-alpha) "."
   counter(section) "." counter(subsec,lower-roman) ". ";}
```

В приведенном выше коде стиль счетчика не задан только для идентификатора `section` — он обозначается значением по умолчанию, представляющим стандартную целочисленную нумерацию. При необходимости стиль счетчика можно объявить с помощью ключевого слова `disc`, `circle`, `square` или `none`.

Обратите внимание на то, что значение счетчика не будет прирастать при объявлении для целевых элементов свойства `display` со значением `none`. При этом счетчики, целевые элементы которых визуализируются со свойством `visibility`, установленным в значение `hidden`, эту способность не теряют.

```
.suppress {counter-increment: cntr; display: none;}
/* cntr не увеличивается */
.invisible {counter-increment: cntr; visibility: hidden;}
/* cntr увеличивается */
```

Область действия счетчика

В предыдущих разделах были рассмотрены принципы комбинирования идентификаторов счетчиков, обеспечивающих нумерацию оглавлений, которые включают названия разделов и подразделов. Как ни странно, но такой достаточно простой способ назначения счетчиков элементам упорядоченных списков становится чрезмерно громоздким даже при незначительном увеличении количества уровней вложения.

Например, при назначении счетчика списку, состоящему всего из пяти уровней вложения, в таблицу стилей потребуется включить достаточно сложный для понимания набор стилевых 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) ".";}
```

A. Тайная жизнь лосося

A.1. Введение

A.2. Среда обитания

A.2.i. Океан

A.2.ii. Реки

A.3. Нерест

A.3.i. Оплодотворение

A.3.ii. Созревание

A.3.iii. Вылупление

Рис. 15.28. Изменение формата счетчика

Только представьте, насколько сложной будет структура правил, которая нумерует упорядоченный список, насчитывающий несколько десятков уровней вложения! (Конечно, такие списки добавляются в документы не часто, что ни в коей мере не упрощает сложность программной конструкции.)

Для упрощения кода нумерации элементов в CSS2.x вводится понятие *области действия* счетчика. В спецификации предполагается, что на каждом уровне вложения счетчик получает отдельную область действия. Наличие у счетчика области действия позволяет добавлять нумерацию к упорядоченным спискам HTML сколь угодно глубокого уровня вложения с помощью очень простых стилевых правил.

```
ol {counter-reset: ordered;}
ol li::before {counter-increment: ordered;
   content: counter(ordered) ". ";}
```

Указанные правила обеспечивают стандартный способ числовой нумерации (в том числе и вложенных) списков — со сбрасыванием в значение 1 и приращением на единицу — так же, как это осуществлялось в ранних спецификациях HTML.

После выполнения приведенного выше кода создается отдельный экземпляр (или образ) счетчика на каждом уровне вложения списка. Следовательно, первый экземпляр счетчика `ordered` создается для списка самого верхнего уровня. Второй экземпляр счетчика соответствует списку, вложенному в список верхнего уровня, и т.д. В каждом из экземпляров счетчика нумерация элементов осуществляется независимо.

Если нумерация элементов вложенных списков должна включать значения счетчиков, назначенные спискам более высокого уровня (например, представляться последовательностью 1, 1.1, 1.2, 1.2.1, 1.2.2, 1.3, 2, 2.1), то функцию `counter()` нужно заменить функцией `counters()`. Все решает всего одна буква в конце названия ("s").

Следующие стилевые правила добавляют к многоуровневому списку счетчики, показанные на рис. 15.29.

```
ol {counter-reset: ordered; list-style: none;}
ol li:before {content: counters(ordered,".") ": ";
   counter-increment: ordered;}
```

В общем случае функция `counters(ordered,".")` возвращает строку, состоящую из значений разделенных точками счетчиков `ordered`, в область действия которых попадает текущий элемент списка. Таким образом, в значение счетчика элемента списка третьего уровня вложения будут включены значения счетчиков не только текущей, но и первых двух областей действия (разделенные точками). Впоследствии к такому комбинированному значению счетчика добавляются символы двоеточия и пробела.

Функция `counters()`, как `counter()`, позволяет указывать стиль нумерации вложенных счетчиков — единый для них всех. Если изменить предыдущий код так, как показано ниже, то элементы списка, продемонстрированного на рис. 15.29, будут нумероваться не в числовом, а в алфавитном порядке.

```
ol li::before {counter-increment: ordered;
   content: counters(ordered,".",lower-alpha) ": ";}
```

Легко заметить, что в предыдущих примерах стиль нумерации элементов `ol` устанавливается объявлением `list-style: none`. А все потому, что счетчики должны представляться генерируемым содержимым, а не замещать маркеры списков. Если упустить объявление `list-style: none` в приведенном выше коде, то номер в начале

каждого элемента списка будет состоять не только из значения счетчика, генерируемого стилевым правилом, но и указываемого перед ним значения счетчика, устанавливаемого пользовательским агентом.

1: Списки
    1.1: Типы списков
    1.2: Маркеры элементов списка
    1.3: Позиционирование маркеров списка
    1.4: Быстрые списки
    1.5: Верстка списков
2: Генерируемое содержимое
    2.1: Вставка генерируемого содержимого
        2.1.1: Позиционирование генерируемого содержимого
    2.2: Типы генерируемого содержимого
        2.2.1: Атрибуты значений
        2.2.2: Кавычки
    2.3: Счетчики
        2.3.1: Сброс и приращение
        2.3.2: Вставка счетчиков
        2.3.3: Область применения счетчиков
3: Резюме

Рис. 15.29. Нумерация вложенных списков

Описанные выше возможности счетчиков открывают широкие возможности по нумерации элементов многоуровневых списков, но необходимость в них возникает достаточно редко. В большинстве случаев форматирование списков сводится к банальному переназначению маркеров элементов. Проще всего эта задача решается с помощью специальных шаблонов.

Шаблон счетчика

В последних спецификациях CSS появилась возможность представления счетчиков специальными шаблонами. Эта операция выполняется с помощью команды `@counter-style`, которая содержит дескрипторы символов счетчика, указывает порядок их назначения элементам списка и имеет следующий синтаксис.

```
@counter-style <name> {
   ...объявления...
}
```

Значение *<name>* представляет название шаблона, определяемое автором документа. Например, следующая команда создает шаблон, устанавливающий для элементов списка чередующиеся треугольные маркеры.

```
@counter-style triangles {
   system: cyclic;
   symbols: ▶ ▷;
}
```

Результат применения такого шаблона к простому списку продемонстрирован на рис. 15.30.

▶. Один
▷. Два
▶. Три
▷. Четыре
▶. Пять
▷. Шесть
▶. Семь

Рис. 15.30. Список, маркеры которого определены шаблоном счетчика

К началу 2017 года команду `@counter-style` и другие инструменты настройки шаблонов счетчиков, описанные в этом разделе, поддерживали только браузеры семейства Firefox. Их очень интересно использовать в документах, но только как самостоятельные значения. В частности, команда `@counter-style` не позволяет сделать значение счетчика частью любого другого значения, например фразы "См. пункт 1а".

В команде `@counter-style` допускается использовать целый ряд специальных дескрипторов.

Дескрипторы команды @counter-style

`system`	Определяет систему нумерации счетчика: `fixed`, `cyclic`, `alphabetic`, `numeric`, `symbolic`, `additive` или `extends`
`symbols`	Определяет символы, используемые в качестве маркеров счетчика; применяются во всех системах нумерации, за исключением `additive` и `extends`
`additive-symbols`	Определяет символы, используемые в счетчиках с аддитивной (`additive`) системой нумерации
`prefix`	Определяет символ или строку символов, добавляемую перед счетчиком в выбранной системе нумерации
`suffix`	Определяет символ или строку символов, добавляемую после счетчика в выбранной системе нумерации
`negative`	Определяет символ или строку символов, обозначающую отрицательные значения счетчика

`range`	Указывает диапазон, в котором определяются значения счетчика; значения вне диапазона представляются так же, как и при обозначении ключевым словом `fallback`
`fallback`	Указывает способ представления значений вне диапазона, заданного для счетчика (`range`)
`pad`	Определяет количество разрядов в значении счетчика, устанавливая символ или строку, которой заполняются недостающие разряды
`speak-as`	Определяет способ произношения значений счетчика в системах речевого воспроизведения текстов

Рассмотрение дескрипторов лучше начать с простых примеров и только затем переходить к более сложным методикам. Но перед тем как углубиться в изучение принципов использования шаблонов счетчиков, ознакомимся с двумя наиболее часто используемыми дескрипторами: `system` и `symbols`.

Дескриптор system

Значение	`cyclic` | `numeric` | `alphabetic` | `symbolic` | `additive` | [`fixed` *<integer>?*] | [`extends` *<counter-style-name>*]
Начальное значение	`symbolic`

Дескриптор symbols

Значение	*<symbol>*+
Начальное значение	нет
Примечание	Значение *<symbol>* может представляться строкой символов Unicode, ссылкой на изображение или идентификатором

Эти два дескриптора содержатся в подавляющем большинстве команд `@counter-style`. Конечно, в объявлении команды `@counter-style`, включающей дескриптор `symbols`, можно обойтись без упоминания дескриптора `system`, но включение его в стилевое правило позволяет предельно точно определить систему нумерации, символы которой применяются в качестве маркеров списка. Не забывайте, что с принципами символьной нумерации элементов списка знаком далеко не каждый веб-дизайнер.

Шаблоны со строго заданным количеством маркеров

Самый простой шаблон представляется дескриптором `system` со значением, обозначаемым ключевым словом `fixed`. Он назначает спискам строго заданный набор неповторяющихся маркеров. Пример такого списка, полученного с помощью следующего CSS-кода, приведен на рис. 15.31.

```
@counter-style emoji {
   system: fixed;
   symbols: 😁 😉 😂 🤔 🙃;
ul.emoji {list-style: emoji;}
```

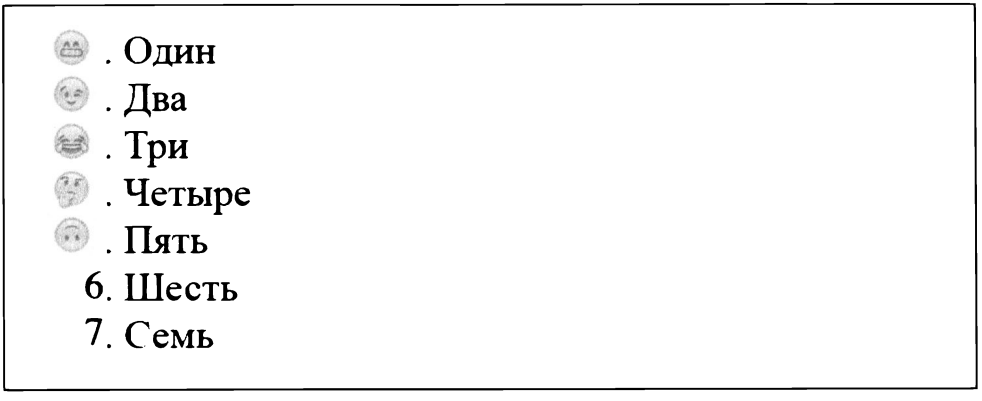


Рис. 15.31. Список со строго заданным шаблоном неповторяющихся маркеров

На пятом элементе список перестает маркироваться символами эмодзи и в отсутствие резервного способа нумерации (см. далее) сбрасывается к варианту, задаваемому по умолчанию для неупорядоченных списков.

Учтите, что символы, указываемые в дескрипторе `symbols`, нужно разделять пробелами. Попытка указать их без разделения приводит к заведомо непрогнозируемому результату (рис. 15.32).

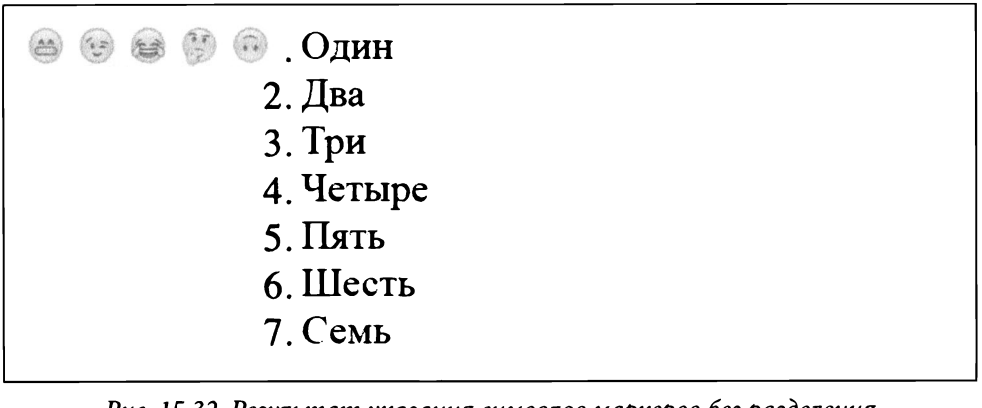


Рис. 15.32. Результат указания символов маркеров без разделения

Эта особенность позволяет создавать последовательности маркеров, каждый из которых включает сразу несколько символов. (О создании шаблонов, отвечающих за последовательную нумерацию элементов списка, рассказано далее.)

Для добавления в маркеры ASCII-символов их нужно указывать заключенными в двойные кавычки. Ниже показано, каким образом в маркеры можно добавить

символы угловых скобок, обычно обрабатываемые синтаксическим анализатором как HTML-код. Данные символы включены в один из возможных шаблонов.

```
@counter-style emoji {
   system: fixed;
   symbols: # $ % ">";
}
```

Хорошей привычкой будет заключать в кавычки любые символы, перечисляемые в дескрипторах команды `@counter-style`. В кавычки также допускается заключать символы Unicode — для повышения надежности кода старайтесь использовать их повсеместно. В частности, в предыдущем примере шаблон счетчика можно представить записью "#" "$" "%" ">".

Начальное значение в шаблоне со строго заданным количеством маркеров устанавливается с помощью дескриптора `system`. Например, в следующем шаблоне маркировка списка начинается с пятого элемента.

```
@counter-style emoji {
   system: fixed 5;
   symbols: 😁 😉 😂 🤔 🎧;
}

ul.emoji {list-style: emoji;}
```

Согласно приведенной выше команде специальными символами обозначаются элементы списка с 5 по 9. Если в качестве резервных маркеров применяются целочисленные значения, то следующий элемент списка будет нумероваться числом `10` (в римской алфавитной нумерации он будет представлен символом J).

Возможность определения начального элемента присуща только шаблонам со строго заданным количеством маркеров.

Шаблоны с повторяющимися маркерами

Следующий тип шаблонов обозначается ключевым словом `cyclic`. Оно указывает на то, что шаблон содержит последовательность повторяющихся маркеров. Рассмотрим, как будет выглядеть список при назначении ему шаблона повторяющихся символов эмодзи, взятых из предыдущего примера. Задача решается с помощью приведенного ниже кода (рис. 15.33).

```
@counter-style emojiverse {
   system: cyclic;
   symbols: 😁 😉 😂 🤔 🎧;
}

ul.emoji {list-style: emojiverse;}
```

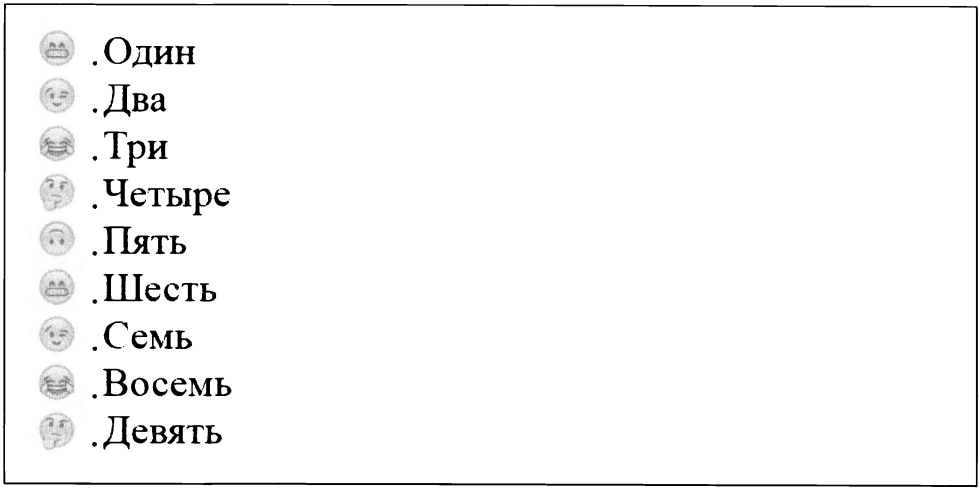


Рис. 15.33. Маркировка списка согласно шаблону с повторяющейся последовательностью символов

Повторение символов шаблона продолжается до добавления маркера к последнему элементу списка.

Шаблон с повторяющимися маркерами может представляться одним-единственным символом, что равнозначно объявлению для него свойства `list-style-type`, значение которого обозначает идентичный маркер. Такое стилевое форматирование обеспечивается следующим кодом.

```
@counter-style thinker {
   system: cyclic;
   symbols: 🤔;
   /* равнозначно объявлению свойства list-style-type: " 🤔" */
   }

ul.hmmm {list-style: thinker;}
```

Заметьте, что во всех приведенных выше шаблонах значения счетчика сопровождаются символом точки, по умолчанию назначаемым дескриптором `suffix`. Кроме суффикса к маркерам списка разрешается добавлять префиксы, представляемые дескриптором `prefix`.

Дескрипторы prefix и suffix

Значение	*<symbol>*
Начальное значение	"" (пустая строка) для `prefix`; \2E (символ точки) для `suffix`
Примечание	Значение *<symbol>* может представляться строкой символов Unicode, ссылкой на изображение или идентификатором

Дескрипторы `suffix` и `prefix` обозначают символы, которые предваряют и завершают маркеры, указываемые в шаблоне счетчика. Суффиксы и префиксы могут включать ASCII-значения, как показано в следующем примере (рис. 15.34).

```
@counter-style wingthinker {
   system: cyclic;
   symbols: 🤔;
   prefix: "~";
   suffix: "~";
}

ul.hmmm {list-style: wingthinker;}
```

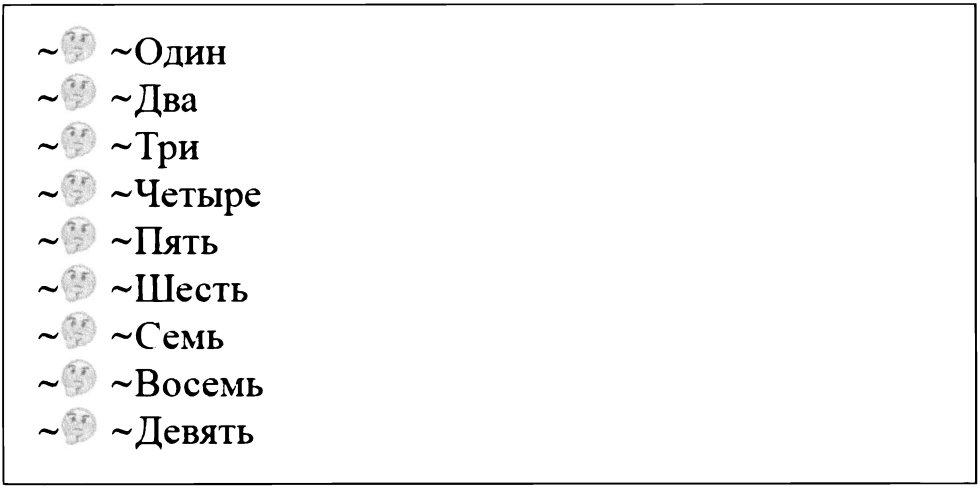


Рис. 15.34. "Мыслитель" качается на волнах (славы)

Ключевое слово `suffix` часто применяется для удаления заданных по умолчанию суффиксов из маркеров списка, как показано в приведенном ниже примере (рис. 15.35).

```
@counter-style thisisfine {
   system: cyclic;
   symbols: 🔥💀☕🔥;
   suffix: "";
}
```

Дополнительно можно использовать суффиксы и префиксы, которые существенно расширяют возможности CSS по оформлению списков.

```
@counter-style thisisfine {
   system: cyclic;
   symbols: 🔥💀☕🔥;
   prefix: "🔥";
   suffix: 🔥;
}
```

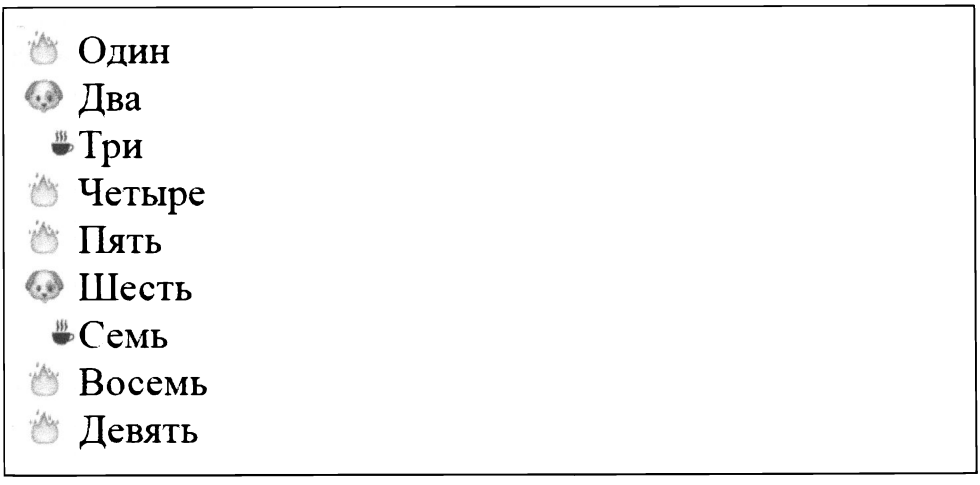


Рис. 15.35. Полностью оформленный список

Удивляет тот факт, что в двойные кавычки заключено только значение `prefix`, но не `suffix`. В действительности справедливы оба варианта, что и продемонстрировано в приведенном выше примере. Как и ранее, заключать символы в кавычки намного надежнее, поэтому при кодировании шаблонов счетчиков старайтесь придерживаться именно такого подхода.

При самостоятельном повторении приведенных выше примеров можно заметить определенное различие между символами эмодзи, объявленными в коде и выводимыми в окне браузера. А все потому, что внешний вид символов эмодзи в первую очередь зависит от платформы, на которой запускается пользовательский агент: Mac OS, iOS, Android, Samsung, Windows, Linux и т.п.

Не забывайте, что в качестве маркеров списка можно использовать не только символы, но и графические объекты. Например, спецификация CSS разрешает применять в качестве значений счетчиков символы клингонской письменности, отсутствующие в кодовой странице Unicode. (Одно из наиболее распространенных заблуждений как раз заключается в том, что символы клингонской письменности включены в кодовую страницу Unicode. Такое предложение действительно рассматривалось в 1996 году, но в 2001 году было окончательно отвергнуто комитетом.) В следующем примере показано, как правильно представить маркеры одноуровневого списка некоторыми из таких символов.

```
@counter-style klingon-letters {
   system: cyclic;
   symbols: url(i/klingon-a.svg) url(i/klingon-b.svg)
      url(i/klingon-ch.svg) url(i/klingon-d.svg)
      url(i/klingon-e.svg) url(i/klingon-gh.svg);
   suffix: url(i/klingon-full-stop.svg);
}
```

Согласно приведенному выше коду элементы списка повторно нумеруются символами от A до GH, в последовательность которых включено всего несколько символов клингонской письменности. Принципы построения шаблонов счетчиков, основанных на числовой и алфавитной нумерации, рассматриваются в следующих разделах.

На начало 2017 года браузеры в большинстве своем не поддерживали использование графических изображений для нумерации элементов списка.

Шаблоны символьной нумерации

Дескриптор, представленный ключевым словом `symbolic`, описывает счетчик, подобный рассмотренному в предыдущем разделе, за тем лишь исключением, что теперь каждое следующее повторение последовательности сопровождается увеличением количества символов в маркерах на единицу. Такой способ маркировки часто применяется при обозначении сносок и в отдельных системах алфавитной нумерации списков. Примеры шаблонов символьной нумерации приведены на рис. 15.36.

```
@counter-style footnotes {
   system: symbolic;
   symbols: "*" "†" "§";
   suffix: ' ';
}
@counter-style letters {
   system: symbolic;
   symbols: А Б В Г Д;
}
```

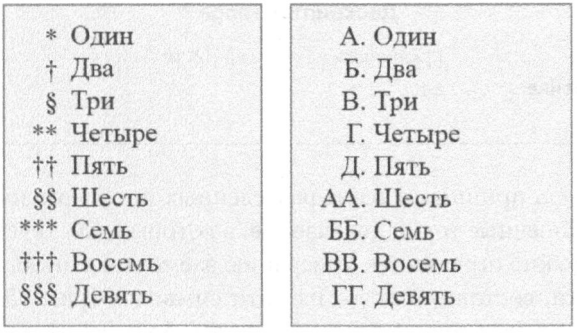


Рис. 15.36. Два примера шаблонов с символьной нумерацией счетчиков

Будьте внимательны и старайтесь не применять к длинным спискам короткие последовательности, включающие всего несколько символов. Такой подход неизбежно приводит к образованию очень длинных маркеров. На рис. 15.37 показано, насколько длинными могут быть значения счетчиков для элементов с 135 по 150 при назначении списку одного из шаблонов из предыдущего примера.

Рис. 15.37. Очень длинные символьные маркеры

Подобные ситуации неизбежны в силу самой природы счетчиков, значения которых имеют склонность к постоянному нарастанию. Чтобы ограничить способность счетчиков увеличивать значения, в спецификацию команды `@counter-style` добавлен дескриптор `range`.

Дескриптор range

Значение	[[<*integer*> | `infinite`]{2}]# | `auto`
Начальное значение	`auto`

Дескриптор `range` принимает пары разделенных пробелом значений, указывающих начальные и конечные точки диапазонов, в которых действует шаблон счетчика. Рассмотрим, как можно ограничить нумерацию элементов списка третьей итерацией последовательности, состоящей всего из пяти символов (рис. 15.38). Решение этой задачи возлагается на следующий простой код (для большей компактности результат представлен на странице в две колонки).

```
@counter-style letters {
system: symbolic;
symbols: A B C D E;
range: 1 15;
}
```

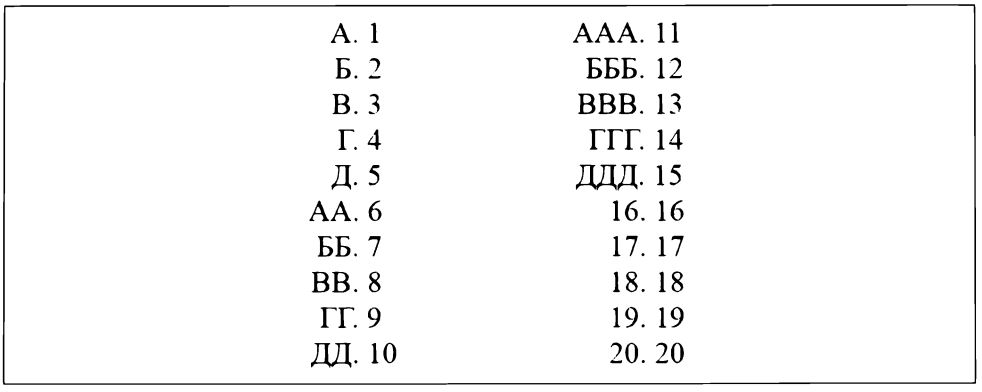


Рис. 15.38. Определение области применения шаблона с помощью дескриптора `range`

Для добавления в шаблон счетчика еще одного диапазона значений предыдущий пример нужно изменить так, как показано ниже.

```
@counter-style letters {
   system: symbolic;
   symbols: А Б В Г Д;
   range: 1 15, 101 115;
}
```

В данном случае символьная нумерация, представленная стилем `letters`, будет применяться только к диапазонам 1–15 и 101–115 (обозначается маркерами от "АААААААААААААААААААА" до "ДДДДДДДДДДДДДДДДДДДДД").

Все элементы списка, находящиеся вне диапазонов, определенных дескриптором `range`, обозначаются значениями счетчика, заданного по умолчанию. Это может быть как счетчик, назначаемый пользовательским агентом самостоятельно, так и резервный счетчик, указанный в дескрипторе `fallback`.

Дескриптор fallback

Значение	*<counter-style-name>*
Начальное значение	`decimal`
Примечание	*<counter-style-name>* представляется любым значением, разрешенным для свойства `list-style-type`

Например, следующий код нумерует элементы списка, находящиеся вне диапазона шаблона счетчика, буквами иврита.

```
@counter-style letters {
   system: symbolic;
   symbols: А Б В Г Д;
   range: 1 15, 101 115;
   fallback: hebrew;
}
```

В качестве резервных систем нумерации можно указывать стили `lower-greek`, `upper-latin` и даже не подлежащие исчислению системы маркировки, подобные `square`.

Указываемая этим дескриптором система нумерации назначается всем элементам списка, лишенным основной системы нумерации. Необходимость в резервной нумерации часто возникает при назначении элементам списка маркеров, представляемых графическими изображениями, с загрузкой которых у пользовательского агента могут возникать вполне очевидные трудности. Следующий код показывает, как можно обойти ситуацию, в которой символы основной системы нумерации, представленные файлом `south.svg`, становятся недоступными для пользовательского агента. В данном примере нумерация элементов списка, лишенных значений основного счетчика, выполняется в алфавитном порядке согласно последовательности, принятой в системе `lower-latin`.

```
@counter-style compass {
   system: symbolic;
   symbols: url(north.svg) url(east.svg) url(south.svg) url(west.svg);
   fallback: lower-latin;
}
```

Шаблоны алфавитной нумерации

Система нумерации, представляемая значением `alpahbetic`, напоминает рассмотренную выше символьную нумерацию, но отличается от нее способом повторения значений счетчика. Как известно, в символьной нумерации каждая последующая итерация сопровождается увеличением количества символов в значениях счетчика на единицу. В алфавитной нумерации символы рассматриваются как разряды значений счетчика. Такая система встречается повсеместно, например в электронных таблицах, где применяется для именования столбцов.

В качестве примера попробуем применить алфавитный способ нумерации к списку из предыдущего раздела. Для этого используем следующий код, результат выполнения которого показан на рис. 15.39.

```
@counter-style letters {
   system: alphabetic;
   symbols: А Б В Г Д;
   /* завершение на Д для более быстрой смены значений */
}
```

Обратите внимание на вторую итерацию значений счетчика — от АА до АД: она сменяется диапазоном значений БА–БД, который продолжается диапазоном ВА–ВД и т.д. В символьной нумерации последний элемент списка получил бы значение ЕЕЕЕЕ, а не ЕЕ, как в данном варианте шаблона.

Заметьте, что для обеспечения целостности алфавитная система нумерации должна представляться как минимум двумя символами из последовательности значений, указанной дескриптором `symbols`. Если представить ее всего одним таким символом, то команда `@counter-style` будет считаться недействительной, а назначаемый ею

шаблон счетчика не будет применяться к списку. К допустимым символам относятся буквы, цифры, любые другие Unicode-значения и даже графические изображения (опять-таки, чисто теоретически).

А. Один	БА. 11
Б. Два	ББ. 12
В. Три	ББ. 13
Г. Четыре	БГ. 14
Д. Пять	БД. 15
АА. Шесть	ВА. 16
АБ. Семь	ВБ. 17
АВ. Восемь	ВВ. 18
АГ. Девять	ВГ. 19
АД. Десять	ВД. 20

Рис. 15.39. Алфавитная нумерация

Шаблоны числовой нумерации

Передача дескриптору `system` ключевого слова `numeric` предопределяет использование в шаблоне счетчика символов *позиционной нумерации*. Согласно такому способу нумерации символы указанной в шаблоне последовательности рассматриваются как разряды выбранной в нем системы счисления. Например, для получения десятичной системы нумерации применяется следующий шаблон счетчика.

```
@counter-style decimal {
   system: numeric;
   symbols: '0' '1' '2' '3' '4' '5' '6' '7' '8' '9';
}
```

Он легко расширяется до шестнадцатеричной системы счисления, как показано ниже.

```
@counter-style hexadecimal {
   system: numeric;
   symbols: '0' '1' '2' '3' '4' '5' '6' '7' '8' '9' 'A' 'B' 'C' 'D'
            'E' 'F';
}
```

В последнем случае счет ведется от 1 до F, затем продолжается от 10 до 1F, после чего — от 20 до 2F, далее от 30 до 3F и т.д. Проще всего в числовом шаблоне представляется двоичная система нумерации.

```
@counter-style binary {
   system: numeric;
   symbols: '0' '1';
}
```

На рис. 15.40 приведены примеры всех трех способов нумерации элементов списка.

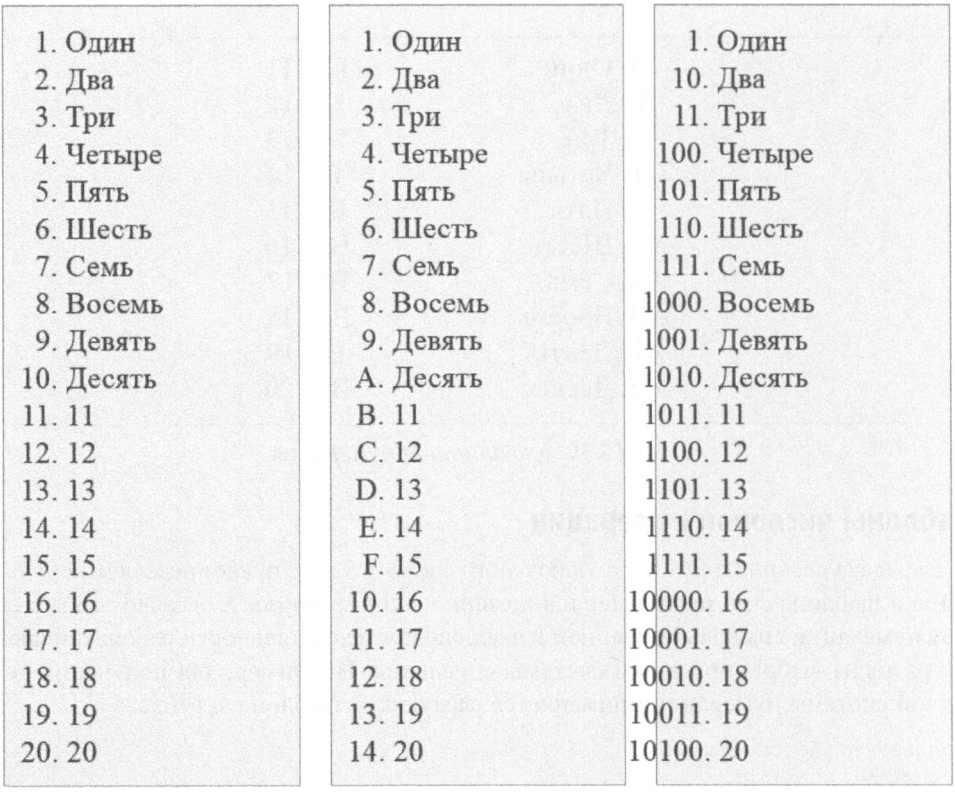


Рис. 15.40. Три наиболее распространенных способа числовой нумерации списков

Наиболее интересная ситуация возникает при обработке отрицательных значений счетчика. В десятичной системе нумерации все предельно просто: отрицательные значения предваряются знаком "минус". А как они обозначаются в символьных системах нумерации, например алфавитной? И каким образом должны представляться значения счетчиков со знаком "минус", заключенные в скобки (система нумерации, заимствованная из бухгалтерского учета)? Для этого предназначен дескриптор `negative`, позволяющий обозначать в счетчиках отрицательные значения.

Дескриптор negative

Значение	*<symbol> <symbol>?*
Начальное значение	\2D (сивол дефиса)
Примечание	Обрабатывается только в счетчиках с системой нумерации, допускающей использование отрицательных значений: `alphabetic`, `numeric`, `symbolic` и `additive`

Дескриптор `negative` подобен самодостаточным дескрипторам `prefix` и `suffix`, хотя и применяется только к отрицательным значениям. Заметьте, что представляемые им символы *включаются* в значения дескрипторов `prefix` и `suffix`.

Вооружившись полученными сведениями, попробуем создать шаблон счетчика, имитирующего принятую в бухгалтерском учете систему нумерации, в которой используются как суффиксы, так и префиксы. Задача решается следующим образом (с результатом выполнения кода можно ознакомиться на рис. 15.41).

```
@counter-style accounting {
   system: numeric;
   symbols: '0' '1' '2' '3' '4' '5' '6' '7' '8' '9';
   negative: "(" ")";
   prefix: "$";
   suffix: " - ";
}
ol.kaching {list-style: accounting;}
```

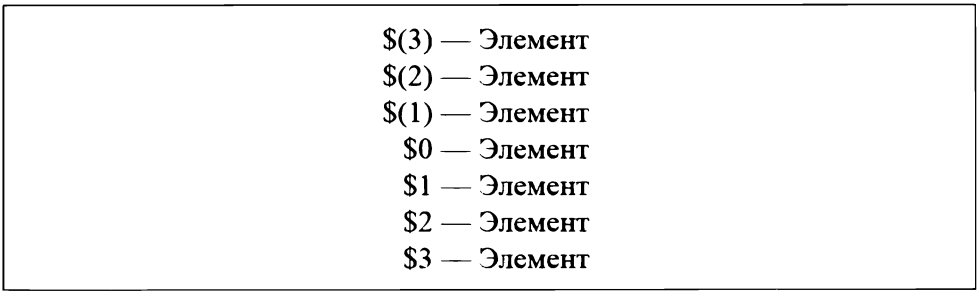


Рис. 15.41. Форматирование отрицательных значений счетчика

Еще один популярный способ форматирования символов числовой нумерации заключается в выравнивании длины (количества разрядов) числового значения. Например, согласно такому способу форматирования значения 1 и 100 представляются как 001 и 100. Для обеспечения указанной возможности команда `@counter-style` снабжается дескриптором `pad`.

Дескриптор pad

Значение	*<integer> && <symbol>*
Начальное значение	`0 ""`

Значения этого дескриптора представляются в несколько необычном формате. Первая часть значения обозначается целым числом и указывает количество разрядов, к которым сводится числовое значение. Вторая часть содержит строку, которой заполняются недостающие разряды числового значения. Рассмотрим такой пример.

```
@counter-style padded {
   system: numeric;
```

```
   symbols: '0' '1' '2' '3' '4' '5' '6' '7' '8' '9';
   suffix: '.';
   pad: 4 "0";
}

ol {list-style: decimal;}
ol.padded {list-style: padded;}c
```

Согласно шаблону, представленному таким кодом, регулярные упорядоченные списки будут нумероваться в классической десятичной системе счисления: 1, 2, 3, 4 и т.д. При этом упорядоченные списки, относящиеся к классу `padded`, будут нумероваться согласно шаблону 0001, 0002, 0003, 0004 и т.д. Результат последнего способа форматирования показан на рис. 15.42.

0001.Один
0002.Два
0003.Три
0004.Четыре
0005.Пять
0006.Шесть
0007.Семь
0008.Восемь
0009.Девять
0010.Десять

Рис. 15.42. Увеличение количества разрядов у значений счетчиков

Для получения значений счетчика, длина которых не менее четырех разрядов, все недостающие разряды заполняются нулями. В предыдущем утверждении ключевой смысл заключен во фразе "не менее четырех". Значения счетчика, представленные пятью разрядами, не укорачиваются до четырех разрядов. Не менее важно то, что при этом остальные значения не удлиняются до пяти разрядов, а сохраняют длину, обозначенную шаблоном счетчика: четыре разряда.

В качестве заполнителя недостающих разрядов могут использоваться любые символы, а не только нуль: дефис, подчеркивание, точка, всевозможные стрелки, пробелы и т.п. Кроме того, значение *<symbol>* можно представлять сразу несколькими символами.

```
@counter-style crazy {
   system: numeric;
   symbols: '0' '1' '2' '3' '4' '5' '6' '7' '8' '9';
   suffix: '.';
   pad: 4 "🤔😉";
}
```

```
ol {list-style: decimal;}
ol.padded {list-style: padded;}
```

Согласно такому шаблону значение 1 счетчика будет представляться в виде 1.

Символ, обозначающий отрицательные числа, выносится в отдельный разряд, а потому не учитывается в значении дескриптора `pad`. В приведенном ниже примере (рис. 15.43) показано, что он смещается за левый край первого разряда счетчика.

```
@counter-style negativezeropad {
   system: numeric;
   symbols: '0' '1' '2' '3' '4' '5' '6' '7' '8' '9';
   suffix: '. ';
   negative: '-';
   pad: 4 "0";
}
@counter-style negativespacepad {
   system: numeric;
   symbols: '0' '1' '2' '3' '4' '5' '6' '7' '8' '9';
   suffix: '. ';
   negative: '-';
   pad: 4 " ";
}
```

–0003. Минус три	– 3. Минус три
–0002. Минус два	– 2. Минус два
–0001. Минус один	– 1. Минус один
0000. Ноль	0. Ноль
0001. Один	1. Один
0002. Два	2. Два
0003. Три	3. Три
0004. Четыре	4. Четыре
0005. Пять	5. Пять
0006. Шесть	6. Шесть
0007. Семь	7. Семь

Рис. 15.43. Добавление разрядов в значения счетчика

Аддитивные шаблоны

Еще один тип шаблонов, требующий отдельного рассмотрения, представляется дескриптором `additive-symbols`. В аддитивных системах нумерации значения счетчика могут представляться сразу несколькими разными символами. В подобном

случае они образуются составлением из отдельных символов, каждый их которых обладает определенной числовой величиной в собственной системе нумерации.

Дескриптор additive-symbols

Значение	[*<integer>* && *<symbol>*]#
Начальное значение	Нет
Примечание	*<integer>* не может представляться отрицательным значением; аддитивные символы не учитываются в отрицательных счетчиках

Принцип действия этого дескриптора намного проще проиллюстрировать на реальном примере, чем давать строгое определение. Ниже приведен пример шаблона аддитивной нумерации, заимствованный с сайта `escss.blogspot.com`.

```
@counter-style roman {
   system: additive;
   additive-symbols:
      1000 M, 900 CM, 500 D, 400 CD,
      100 C, 90 XC, 50 L, 40 XL,
      10 X, 9 IX, 5 V, 4 IV, 1 I;
}
```

Такой шаблон воспроизводит классическую римскую нумерацию. Еще один пример аддитивного счетчика, значения которого представляются числами на гранях игральных костей, описан в спецификации команды `@counter-style`.

```
@counter-style dice {
   system: additive;
   additive-symbols: 6 ⚅, 5 ⚄, 4 ⚃, 3 ⚂, 2 ⚁, 1 ⚀, 0 "__";
   suffix: " ";
}
```

Результат применения обоих шаблонов к реальным спискам показан на рис. 15.44. И здесь каждый список для большего удобства отображается в три столбца.

-3. Минус три	VI. Шесть	XV. 15	-3 Минус три	⚅ Шесть	⚀⚀⚀⚁ 15
-2. Минус два	VII. Семь	XVI. 16	-2 Минус два	⚀⚁ Семь	⚀⚀⚀⚂ 16
-1. Минус один	VIII. Восемь	XVII. 17	-1 Минус один	⚀⚂ Восемь	⚀⚀⚀⚃ 17
0. Ноль	IX. Девять	XVIII. 18	__ Ноль	⚀⚃ Девять	⚀⚀⚀⚄ 18
I. Один	X. Десять	XIX. 19	⚀ Один	⚀⚄ Десять	⚀⚀⚀⚄⚀ 19
II. Два	XI. 11	XX. 20	⚁ Два	⚀⚅ 11	⚀⚀⚀⚄⚁ 20
III. Три	XII. 12	XXI. 21	⚂ Три	⚁⚁ 12	⚀⚀⚀⚄⚂ 21
IV. Четыре	XIII. 13	XXII. 22	⚃ Четыре	⚁⚁⚀ 13	⚀⚀⚀⚄⚃ 22
V. Пять	XIV. 14	XXIII. 23	⚄ Пять	⚁⚁⚁ 14	⚀⚀⚀⚄⚄ 23

Рис. 15.44. Аддитивная нумерация списков

Символы, из которых составляются значения счетчика, можно заключать в кавычки: `6 "⚅"`, `5 "⚄"`, `4 "⚃"` и т.д.

Порядок следования символов в значениях счетчика играет очень важную роль. Обратите внимание на то, что в приведенных выше шаблонах аддитивных счетчиков (римский и игральные кости) значения объявляются в последовательности от наибольшего к наименьшему. Если указать значения в другой последовательности, то список будет пронумерован совершенно иным способом.

Не забывайте, что для получения правильно работающих аддитивных счетчиков нужно использовать дескриптор `additive-symbols`, а не `symbols`. Попытка объявления шаблона с аддитивной последовательностью символов и дескриптором `symbols`, скорее всего, приведет к отмене команды `@counter-style` и сбросу устанавливаемой ею системы нумерации. (Подобным образом синтаксический анализатор браузера отреагирует на использование дескриптора `additive-symbols` в неаддитивных шаблонах.)

Последнее замечание: алгоритм, используемый для создания аддитивных систем нумерации, делает возможным получение таких шаблонов счетчиков, отдельные значения которых могут представляться совсем не так, как предполагалось изначально. Рассмотрим такой пример.

```
@counter-style problem {
   system: additive;
   additive-symbols: 3 "Y", 2 "X";
   fallback: decimal;
}
```

В этом примере первые четыре номера счетчика представляются значениями 1, X, Y, 4, YX, несмотря на то что на интуитивном уровне восприятия четвертое значение должно выражаться комбинацией XX. Но такому результату полностью противоречат правила обработки последовательности символов, указанные в шаблоне. Согласно спецификации, "в случае возникновения противоречий предпочтение отдается линейному, относительно положения символов в исходной последовательности, способу приращения значений счетчика".

Попробуйте самостоятельно ответить на вопрос: каким образом значение 3 сопоставляется с символом III в приведенном выше примере аддитивного шаблона нумерации римскими числами? Для получения точного ответа вам придется разобраться с аддитивными алгоритмами вычисления значений счетчиков, описанными в спецификации CSS3. Тем не менее правильный ответ можно получить даже в отсутствие такого рода знаний — путем простых логических рассуждений, принимая во внимание символ, которым в счетчике обозначается значение 1.

Шаблоны с расширенным набором значений

Как следует из названия, в этом разделе речь пойдет о способах расширения наборов значений, включенных в стандартные системы нумерации. Предположим, что значения десятичной системы нумерации нужно дополнить двумя начальными

разрядами и снабдить суффиксом в виде закрывающей скобки. Расширенную указанным способом систему нумерации можно представить таким шаблоном.

```
@counter-style mydecimals {
   system: numeric;
   symbols: '0' '1' '2' '3' '4' '5' '6' '7' '8' '9';
   suffix: ") ";
   pad: 2 "0";
}
```

Вполне достойное решение, но слишком уж громоздкое для столь простого способа нумерации списка.

Чтобы упростить код шаблона, дескриптор `system` нужно снабдить ключевым словом `extends`. Оно не представляет отдельную систему нумерации, а всего лишь указывает на расширение существующей (заранее объявленной). При его использовании предыдущий пример переписывается следующим образом.

```
@counter-style mydecimals {
   system: extends decimal;
   suffix: ") ";
   pad: 2 "0";
}
```

Согласно такому шаблону расширение набора значений осуществляется для системы нумерации, обозначаемой ключевым словом `decimal`, которое известно нам из определения свойства `list-style-type`. Такой подход устраняет необходимость объявления в шаблоне исходной последовательности значений счетчика.

Ключевое слово `extends` имеет весьма скромную область применения. Попытка использования его вместе со значением `symbols` или `additive-symbols` приводит к полному игнорированию команды `@counter-style` пользовательским агентом. В частности, вам не удастся получить шестнадцатеричную систему нумерации, расширяя набор значений десятичного счетчика.

При этом всегда можно расширить значения стандартной шестнадцатеричной системы счисления. В качестве примера рассмотрим, каким образом можно получить показанные на рис. 15.45 варианты шестнадцатеричной нумерации. (Чтобы получить резкий переход от элемента 19 к 253, один из элементов списка снабжается атрибутом `value="253"`.)

```
@counter-style hexadecimal {
   system: numeric;
   symbols: '0' '1' '2' '3' '4' '5' '6' '7' '8' '9' 'A' 'B' 'C'
      'D' 'E' 'F';
}
@counter-style hexpad {
   system: extends hexadecimal;
   pad: 2 "0";
}
@counter-style hexcolon {
   system: extends hexadecimal;
   suffix: ": ";
}
```

```
@counter-style hexcolonlimited {
   system: extends hexcolon;
   range: 1 255; /* последнее значение - FF */
}
```

-3. Минус три B. 11	-03. Минус три 0B. 11	-3: Минус три B: 11	-3: Минус три B: 11
-2. Минус два C. 12	-02. Минус два 0C. 12	-2: Минус два C: 12	-2: Минус два C: 12
-1. Минус один D. 13	-01. Минус один 0D. 13	-1: Минус один D: 13	-1: Минус один D: 13
0. Ноль E. 14	00. Ноль 0E. 14	0: Ноль E: 14	0: Ноль E: 14
1. Один F. 15	01. Один 0F. 15	1: Один F: 15	1: Один F: 15
2. Два 10. 16	02. Два 10. 16	2: Два 10: 16	2: Два 10: 16
3. Три 11. 17	03. Три 11. 17	3: Три 11: 17	3: Три 11: 17
4. Четыре 12. 18	04. Четыре 12. 18	4: Четыре 12: 18	4: Четыре 12: 18
5. Пять 13. 19	05. Пять 13. 19	5: Пять 13: 19	5: Пять 13: 19
6. Шесть FD. 253	06. Шесть FD. 253	6: Шесть FD: 253	6: Шесть FD: 253
7. Семь FE. 254	07. Семь FE. 254	7: Семь FE: 254	7: Семь FE: 254
8. Восемь FF. 255	08. Восемь FF. 255	8: Восемь FF: 255	8: Восемь FF: 255
9. Девять 100. 256	09. Девять 100. 256	9: Девять 100: 256	9: Девять 256: 256
A. Десять 101. 257	0A. Десять 101. 257	A: Десять 101: 257	A: Десять 257: 257

Рис. 15.45. Расширенные шестнадцатеричные системы нумерации

Обратите внимание на то, что в последнем шаблоне (`hexcolonlimited`) расширению подлежит система нумерации `hexcolon`, представленная третьим шаблоном примера и являющаяся расширением базовой системы `hexadecimal`. В ней нумерация заканчивается значением FF (элемент 255), как и предполагает объявление `range: 1 255`.

Голосовые шаблоны

Даже построение собственных символьных шаблонов нумерации не вызывает настолько сильного интереса, как решение этой же задачи с помощью функций обработки голосовых данных, например программы чтения с экрана VoiceOver компании Apple или экранного диктора JAWS. Только представьте, насколько эффектной будет страница, позволяющая озвучивать количество очков, выброшенных пользователем на игральных костях, или фазы луны, указанные в сводке прогноза погоды! Определение значений счетчика с помощью функции голосового анализа возлагается на дескриптор `speak-as`.

Дескриптор speak-as

Значение	`auto` | `bullets` | `numbers` | `words` | `spell-out` | *<counter-style-name>*
Начальное значение	`auto`

Проще всего значения этого дескриптора рассматривать в обратном порядке. Значение *<counter-style-name>* указывает альтернативную систему нумерации, уже известную пользовательскому агенту. Например, голосовое представление значений, нанесенных на гранях игральных костей, всегда можно выразить в десятичной системе нумерации.

```
@counter-style dice {
   system: additive;
   speak-as: decimal;
   additive-symbols: 6 ⚅, 5 ⚄, 4 ⚃, 3 ⚂, 2 ⚁, 1 ⚀;
   suffix: " ";
}
```

Применив такой шаблон к реальному списку, можно услышать, что значение "⚅⚅⚂" будет звучать как английское числительное "fifteen". Если же значение дескриптора `speak-as` представить ключевым словом `lower-latin`, то значение счетчика будет произноситься как "oh" (прописная O).

Назначение ключевого слова `spell-out` кажется вполне однозначным, но на практике таковым не является. В соответствии со спецификацией оно обязывает озвучивать значения счетчика в "числовом произношении" ("counter representation"), что обычно сводится к побуквенному произношению слов. Что на самом деле кроется за столь хитроумным выражением, остается загадкой, ответ на которую не проясняет даже его определение: "числовое произношение значения сводится к последовательному озвучиванию символов счетчика".

Ключевое слово `word` определяет такие же действия, что и значение `spell-out`, за тем лишь исключением, что значения счетчика произносятся пословно, а не посимвольно. Как и в предыдущем случае, точный алгоритм чтения спецификацией не определяется.

Передача ключевого слова `numbers` обязывает произносить значения счетчика как числа, представляемые основным языком документа. Согласно ему, значение "⚅⚅⚂", как и ранее, будет произнесено как английское "fifteen" — по крайней мере, в документах на английском языке. Это же значение в других языках будет звучать совершенно по-иному: "quince" — в испанском, "fünfzehn" — в немецком, "shíwǔ"— в китайском и т.п.

Значение `bullets` обеспечивает голосовое распознавание маркеров неупорядоченных списков. Учтите, что названия маркеров могут вообще не произноситься либо представляться коротким звуковым сигналом, подобным щелчку или звонку.

Последнее ключевое слово, `auto`, применяется в дескрипторе `speak-as` по умолчанию. Его назначение в первую очередь зависит от системы нумерации, заданной для счетчика. В алфавитных системах нумерации оно обозначает такой же способ произношения, как в объявлении `speak-as: spell-out`. В системах `cyclic` автоматическое произношение сводится к варианту, представленному ключевым словом `bullets`. Во всех остальных случаях значения счетчика произносятся так же, как при объявлении для дескриптора `speak-as` значения `numbers`.

Исключение делается только для расширенных систем нумерации, объявляемых с помощью дескриптора `extends`. В подобных ситуациях эффект ключевого слова `auto` полностью определяется основной системой нумерации счетчика. Согласно такому определению в следующем счетчике значения будут произноситься так, как если бы дескриптор `speak-as` объявлялся с ключевым словом `bullets`.

```
@counter-style emojilist {
   system: cyclic;
   symbols: 😀 😉 😂 🤔 🙃;
}
@counter-style emojibrackets {
   system: extends emojilist;
   suffix: "]] ";
   speak-as: auto;
}
```

Резюме

Несмотря на более чем скромные возможности по стилевому форматированию списков и неоднозначную поддержку браузерами инструментов управления генерируемым содержимым, списки все еще играют основополагающую роль в верстке документов. Один из способов их использования заключается в образовании навигационных и боковых панелей, которые представляются через списки гиперссылок, лишенные маркеров и символов нумерации. Простота оформления и стилевого форматирования таких структур делает их невероятно функциональными и удобными в использовании. С добавлением в спецификацию CSS3 новых, более совершенных средств форматирования списков их незаменимость в веб-дизайне станет поистине неоспоримой.

Списки в документах, язык разметки которых не предполагает образование таких структур, представляются генерируемым содержимым — оно позволяет добавлять специальные объекты или символы перед специально обозначенными элементами (такими, как ссылки на файлы PDF или Word). Кроме того, с помощью генерируемого содержимого можно легко вывести на печать адреса гиперссылок документа или добавить в его текст кавычки. Способов использования генерируемого содержимого неисчислимое множество, а область их применения определяется исключительно вашим воображением. Поддержка спецификацией CSS счетчиков позволяет упорядочивать объекты документа, не относящиеся к элементам списков, в частности, заголовки и фрагменты программного кода. В следующей главе речь пойдет о стилевых инструментах геометрического изменения элементов документа. Вы узнаете о том, как изменять форму и положение элементов и даже перспективу документа.

Трансформации

Появление технологии стилевого форматирования не избавило документы от их главного недостатка — строгой прямоугольной формы и необходимости позиционирования в прямоугольной системе координат. Несмотря на многочисленные ухищрения, призванные изменить внешний вид и оформление элементов, при верстке они все равно представляются прямоугольными контейнерами. И только в конце 2000-х годов спецификация была дополнена функциями трансформации элементов, позволяющими искажать их самыми разными способами, в том числе и в трехмерном пространстве.

Вы уже сталкивались с простейшими операциями трансформации, используя стилевые свойства в операциях абсолютного или относительного позиционирования объектов — в первую очередь, при выравнивании и/или установке отрицательных отступов. Кроме того, трансформацией считается операция смещения элемента в положение, отличное от занимаемого по умолчанию. При этом стилевые инструменты трансформации позволяют не только переносить элементы в другое место документа, но и выполнять над ними более сложные преобразования. Поворот фотографии для придания изображению более естественного вида, создание интерфейсов, в которых часть данных умышленно скрыта за переворачивающимися элементами, и изменение перспективы для привлечения внимания к боковым панелям сайта — эти и многие другие задачи вполне можно решить одними только средствами CSS. Как бы пафосно это ни звучало, но стилевые свойства трансформации полностью изменят вашу точку зрения на методы верстки и структуру документа.

Система координат

Перед тем как углубиться в новый материал, обозначим пространство, в котором выполняется трансформация элементов. В документах, подлежащих стилевому форматированию, применяется два типа координатных систем, и вам крайне важно ознакомиться с ними.

Если вы уже знакомы с декартовой и сферической системами координат, можете пропустить данный раздел.

Сначала мы рассмотрим *декартову*, или прямоугольную, систему координат. В ней положение точки на плоскости задается двумя, а в трехмерном пространстве — тремя координатами. В CSS такая система координат представлена горизонтальной (X) и вертикальной (Y) осями, а также осью глубины (Z). Прямоугольное координатное пространство документа показано на рис. 16.1.

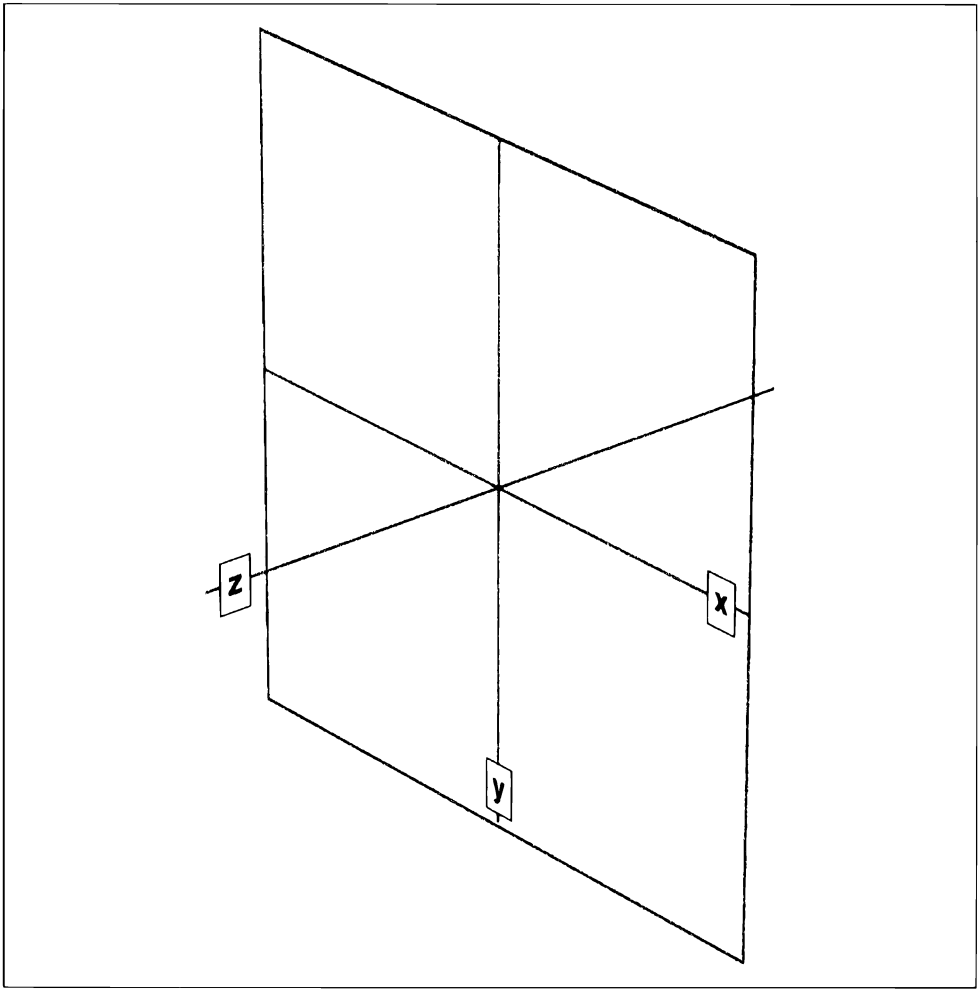


Рис. 16.1. Декартова система координат в CSS

Трансформация объектов в двухмерном пространстве (2D) сводится к использованию только двух (X, Y) из трех (X, Y и Z) осей декартовой системы координат. Согласно общепринятому соглашению, положительные значения горизонтальной оси (X) указываются в правой, а отрицательные — в левой половине координатного пространства. При этом положительные значения вертикальной оси (Y) занимают его нижнюю, а отрицательные — верхнюю половину.

Такой подход может показаться странным, поскольку все мы привыкли к тому, что чем больше вертикальная координата, тем выше уровень, а не ниже, как принято в CSS. (Именно поэтому символ Y указан у нижнего, а не у верхнего края вертикальной оси на рис. 16.1. Подпись оси всегда указывается в области ее положительных значений.) Те, кому доводилось заниматься абсолютным позиционированием элементов, наверняка знают, что положительные значения свойства `top` обозначают смещение вниз, а подъем вверх выражается отрицательными значениями. Таким образом, смещение влево и вниз представляется отрицательными координатами для оси X и положительными — для оси Y.

```
translateX(-5em) translateY(33px)
```

Как будет показано далее, это вполне допустимая команда трансформации, обеспечивающая перенос (смещение) элемента на расстояние `5em` влево и 33 пикселя вниз.

Для перемещения в трехмерном пространстве кроме указанных выше двух координат нужно также указать положение элемента вдоль оси Z. Она направлена перпендикулярно поверхности экрана и указывает глубину залегания элемента. Сугубо теоретически, чем больше координата z, тем ближе объект находится к пользователю. Отрицательные значения z-координаты предопределяют бо́льшую глубину "залегания" элементов, чем положительные. В этом понимании координатное пространство оси Z совпадает с пространством значений свойства `z-index`.

Для того чтобы переместить элемент на передний план, для него нужно в явном виде определить координату вдоль оси Z.

```
translateX(-5em) translateY(33px) translateZ(200px)
```

В результате применения такой команды изображение элемента формируется на 200 пикселей ближе к наблюдателю, чем в случае ее отсутствия.

Но каким образом изображение приближается к наблюдателю на 200 пикселей, ведь голографические мониторы остаются непозволительной роскошью для большинства пользователей? Какое пространство в действительности соответствует расстоянию 200 пикселей от поверхности монитора? Как изменяется внешний вид приближаемого элемента и что с ним происходит при расположении на слишком большом удалении? Ответ на эти и многие другие вопросы, неизбежно возникающие при первом знакомстве с инструментами трансформации элементов, будут даны в следующих разделах. А пока достаточно знать, что перемещение объектов вдоль оси Z приводит к их приближению или удалению.

Важно понимать, что каждый элемент документа снабжается собственной системой координат, которая трансформируется вместе с ним. Таким образом, при повороте элемента вместе с ним поворачиваются оси связанной с ним системы координат (рис. 16.2). Последующие преобразования элемента будут рассчитываться относительно повернутой системы координат, а не осей координатной системы монитора.

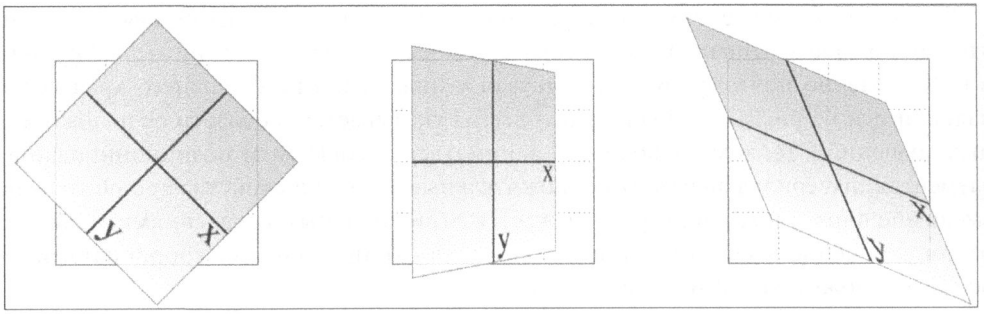


Рис. 16.2. Система координат элемента

Раз уж мы заговорили о повороте и вращении элемента, обозначим еще одну систему координат, в которой осуществляется преобразование элементов. Речь идет о *сферической* системе координат, в которой положение объекта указывается через углы его поворота в трехмерном пространстве (рис. 16.3).

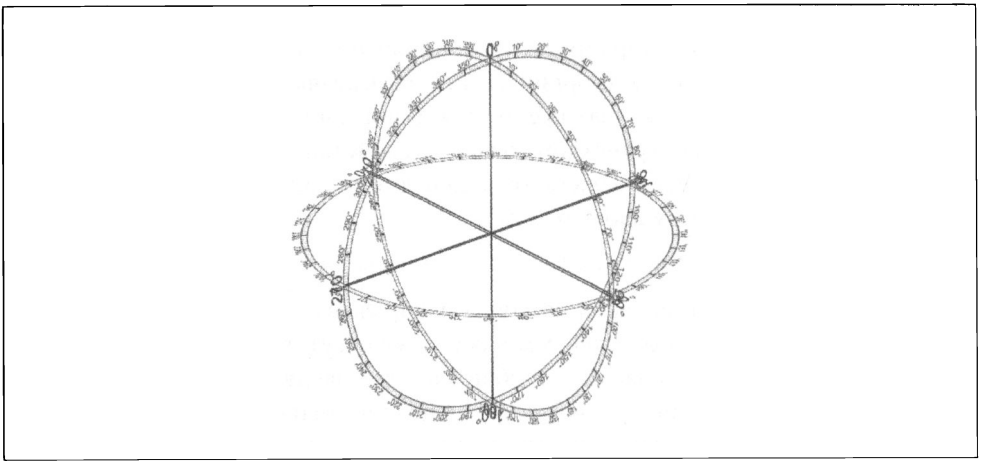


Рис. 16.3. Сферическая система координат, применяемая для трансформации элементов в CSS

В двухмерном пространстве применяется полярная система координат, описывающая в точности такое же пространство, что и плоская декартова система координат, включающая только оси X и Y. В такой системе вращение двухмерного элемента выполняется вокруг оси Z. В случае поворота элемента относительно оси X он наклоняется от наблюдателя или к нему, а при вращении вокруг оси Y поворачивается влево или вправо от него (рис. 16.4).

Вернемся к описанию вращения элемента в двухмерном пространстве. Предположим, элемент нужно повернуть на 45° в плоскости экрана (т.е. вокруг оси Z). Обычно для выполнения этой операции применяется такая функция:

```
rotate(45deg)
```

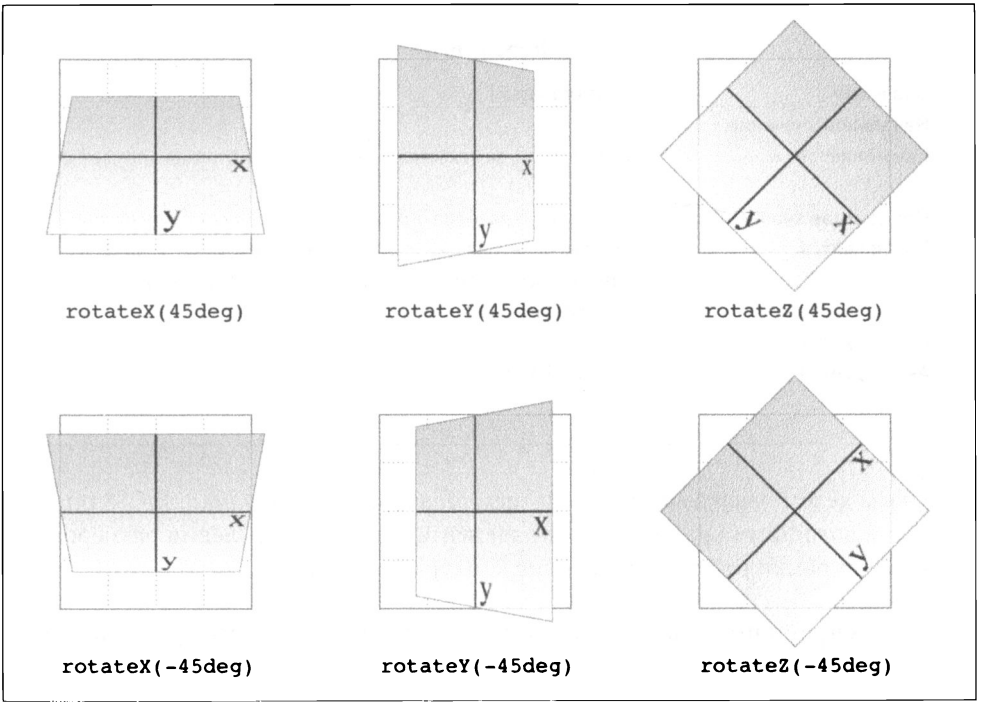


Рис. 16.4. Вращение элемента вокруг осей декартовой системы координат

Передача ей значения `-45deg` приводит к равнозначному повороту элемента вокруг оси Z в обратном направлении: против часовой стрелки. Иными словами, в обоих случаях вращение выполняется в плоскости XY (рис. 16.5).

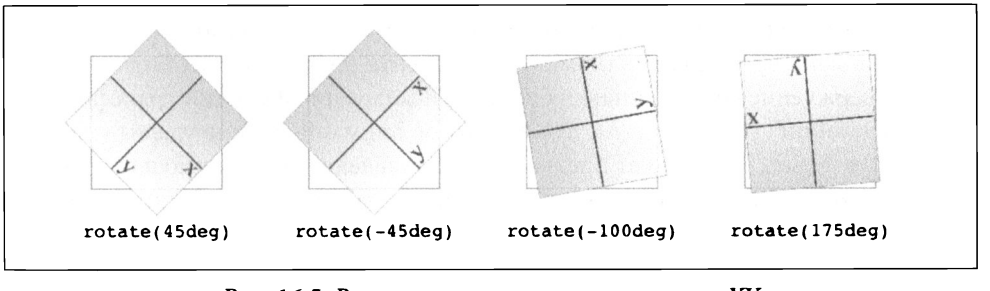


Рис. 16.5. Вращение элемента в плоскости XY

На этом знакомство с системами координат завершается, и мы переходим к рассмотрению инструментов трансформации элементов.

Трансформация элементов

За трансформацию элементов отвечает единственное свойство `transform`, но его действие уточняется некоторыми другими стилевыми свойствами.

transform

Значение	*<transform-list>* | `none`
Начальное значение	`none`
Применяется	Все элементы, за исключением "неделимых строчных элементов" (см. описание)
Процентное значение	Относительно размера ограничительной рамки (см. описание)
Вычисляется	Согласно определению, за исключением значений, выраженных в относительных единицах длины, которые преобразуются в абсолютные значения
Наследуется	Нет
Анимируется	Как трансформация

Первым делом ознакомимся с понятием *ограничительной рамки*. В CSS она представляется внешними краями границы элемента, т.е. при вычислении размеров ограничительной рамки отступы и внешние контуры элемента не учитываются.

Ограничительная рамка элемента подлежащей трансформации таблицы включает контейнеры таблицы и всех связанных с ней заголовков.

При стилевой трансформации элементов, вмещающих SVG-файлы, ограничительная рамка совпадает с ограничительной рамкой самого изображения.

Помните, что все трансформируемые элементы (элементы, свойство `transform` которых установлено в значение, отличное от `none`) размещаются на определенных уровнях стека. Более того, независимо от получаемого в результате трансформации эффекта элемент продолжает занимать в документе столько же места, как и до нее. Данное утверждение справедливо для всех без исключения функций трансформации.

В определении свойства `transform` отдельное внимание нужно уделить значению *<transform-list>*. Его формат предполагает объявление сразу нескольких функций трансформации, отделяемых друг от друга символами пробела. Один из примеров синтаксиса такой команды трансформации приведен на рис. 16.6.

```
#example {transform: rotate(30deg) skewX(-25deg) scaleY(2);}
```

Функции трансформации, объявленные в приведенном выше примере, выполняются последовательно, начиная с первой (крайней левой) и заканчивая последней (крайней правой). Порядок объявления функций играет очень важную роль: его изменение чревато получением совершенно иных результатов. Рассмотрим два правила, результаты выполнения которых показаны на рис. 16.7.

```
img#one {transform: translateX(200px) rotate(45deg);}
img#two {transform: rotate(45deg) translateX(200px);}
```

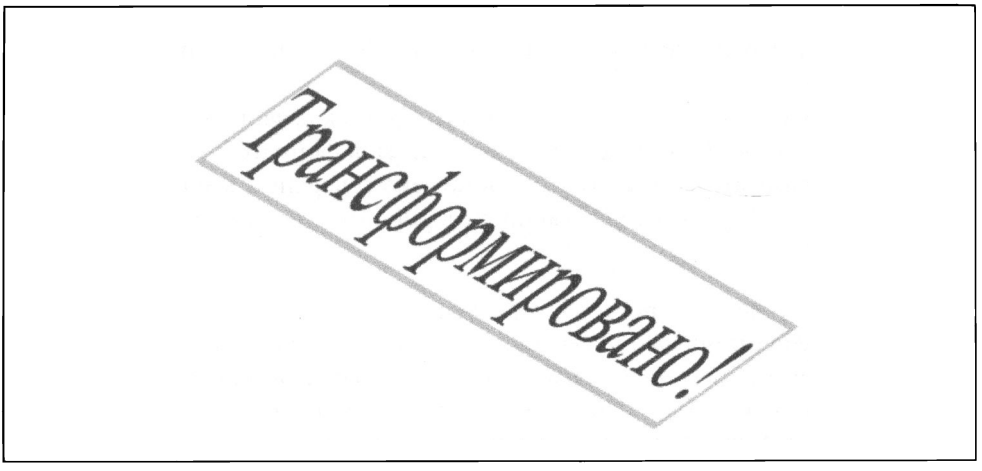


Рис. 16.6. Трансформация элемента `div`

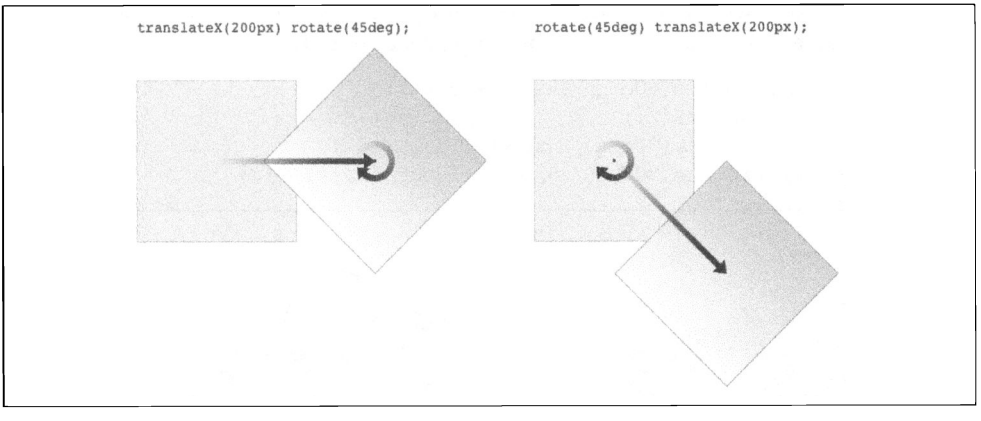


Рис. 16.7. Результат изменения порядка объявления функций в свойстве `transform`

Согласно первому правилу, изображение сначала перемещается на 200 пикселей вдоль оси X, а затем поворачивается на 45°. Второе правило обязывает сначала повернуть изображение на 45° и только затем переместить его на 200 пикселей вдоль оси X. В последнем случае поворот изображения осуществляется относительно оси X самого изображения, а не его родительского элемента (страницы, окна просмотра). Не забывайте, что спецификация CSS требует проведения трансформации элементов в их собственной системе координат.

Следующий пример подобен предыдущему, но наряду со смещением он предполагает наклон элемента, а не его вращение.

```
img#one {transform: translateX(100px) scale(1.2);}
img#two {transform: scale(1.2) translateX(100px);}
```

Ситуации, в которых порядок трансформации элементов не играет никакой роли, возникают очень редко. В большинстве случаев он критически важен для получения

требуемого эффекта. Чтобы не попасть впросак, всегда исходите из предположения исключительной важности порядка выполнения функций, указанных в свойстве `transform`.

Особое внимание уделяйте синтаксису функций, включенных в команду трансформации элементов. Все они должны указываться в формате, распознаваемом обработчиком пользовательского агента. Достаточно совершить всего одну ошибку, и свойство `transform` будет полностью проигнорировано. Рассмотрим такой пример:

```
img#one {transform: translateX(100px) scale(1.2) rotate(22);}
```

В данном коде ошибка кроется в коде функции `rotate()`. Передаваемое ей значение не снабжено единицами измерения, а потому пользовательский агент не будет обрабатывать сразу всю строку кода. В результате изображение останется неизменным и будет занимать исходное положение — к нему не будут применены даже правильно объявленные функции смещения и поворота.

Учтите, что команды трансформации лишены кумулятивного эффекта. Если ранее в коде одна из команд трансформации уже применялась к некоему элементу, то для дальнейшего применения к нему еще одного типа преобразования ее нужно будет повторить, как показано в следующем примере (рис. 16.8).

```
#ex01 {transform: rotate(30deg) skewX(-25deg);}
#ex01 {transform: scaleY(2);}
#ex02 {transform: rotate(30deg) skewX(-25deg);}
#ex02 {transform: rotate(30deg) skewX(-25deg) scaleY(2);}
```

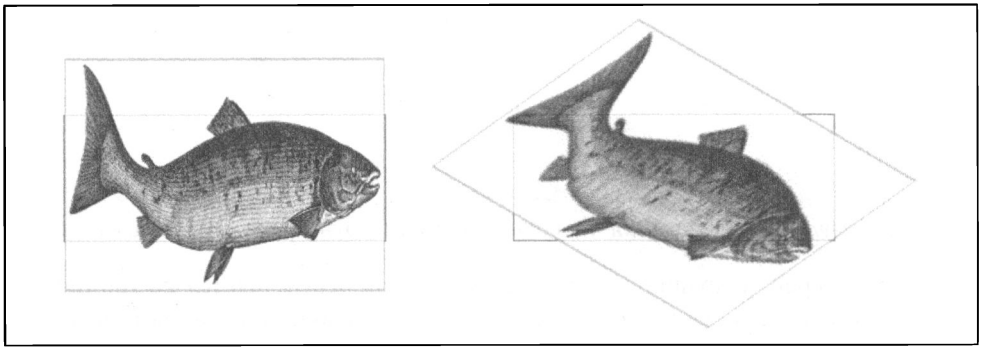


Рис. 16.8. Замена и продолжение трансформации элемента

В первом примере второе стилевое правило полностью замещает первое правило, поэтому элемент всего лишь масштабируется вдоль оси Y. Такое решение вполне обосновано: ситуация сродни повторному определению размера шрифта для одного и того же элемента — такие объявления не носят накопительного характера, поскольку одному и тому же шрифту нельзя одновременно назначить несколько размеров. В противоположность этому второе правило второго примера включает функции трансформации первого правила, поэтому все они применяются к целевому элементу наряду с функцией `scaleY()`.

Из указанного правила объявления функций трансформации существует исключение, справедливое только для анимируемых элементов. Оно гласит, что аддитивная

трансформация справедлива только для переходов и действительной анимации элементов. Следовательно, анимации может подлежать всего одна из многих функций трансформации, объявленных в правиле, — в процессе анимации их действие не отменяется, как показано ниже:

```
img#one {transform: translateX(100px) scale(1.2);}
```

Последующая анимация угла поворота элемента не отменяет его смещение вдоль оси X и наклон на угол, указанный в стилевом правиле.

Для того чтобы получить аддитивную трансформацию без настройки переходов и анимации, воспользуемся специальным приемом, который заключается в отнесении целевых элементов к классу `:hover`, поскольку эффект события, назначаемого этим классом, представлен самым настоящим переходом (хотя и не подлежит настройке с помощью соответствующих свойств). Рассмотрим пример.

```
img#one {transform: translateX(100px) scale(1.2);}
img#one:hover {transform: rotate(-45deg);}
```

Такое форматирование поворачивает предварительно смещенное и масштабированное изображение на угол 45° после наведения на него указателя мыши. При этом поворот осуществляется мгновенно в силу отсутствия заранее заданного временного интервала в объявлении перехода. Следовательно, эффект наведения указателя мыши включает моментальный переход элемента из одного состояния в другое, выполняемый сразу же после трансформации элемента с помощью свойства `transform`.

Учтите, что на момент написания книги функции трансформации невозможно было применять к неделимым элементам строчного уровня, обычно представляемым элементами `span`, `a` и т.п. Они подлежат трансформации только вместе с родительскими элементами блочного уровня. Например, для поворота элемента `span` с помощью функций трансформации сначала нужно изменить способ его представления в документе, воспользовавшись объявлением `display: block` или `display: inline-block`. Такое требование позволяет избежать конфликтных ситуаций, возникающих при стилевом оформлении строчных элементов, занимающих несколько строк. Представьте, что элемент `span` или любой другой элемент строчного уровня переносится на следующую строку. Каким образом он будет поворачиваться после применения к нему функции `rotate()`? Как единое целое? Или каждая из частей элемента `span` должна вращаться независимо от остальных строк? Поскольку единого мнения по этому вопросу нет, на текущий момент явная трансформация строчных элементов остается невозможной.

Функции трансформации

К моменту написания книги в спецификации была определена 21 функция трансформации элементов. Расширение возможностей по стилевому преобразованию элементов достигается за счет объявления в свойстве `transform` сразу нескольких функций трансформации, а также изменения порядка их обработки. Все разрешенные для применения в стилевом свойстве `transform` функции трансформации приведены в табл. 16.1.

Таблица 16.1. Функции трансформации

`translate()`	`scale()`	`rotate()`	`skew()`	`matrix()`
`translate3d()`	`scale3d()`	`rotate3d()`	`skewX()`	`matrix3d()`
`translateX()`	`scaleX()`	`rotateX()`	`skewY()`	`perspective()`
`translateY()`	`scaleY()`	`rotateY()`		
`translateZ()`	`scaleZ()`	`rotateZ()`		

Чаще всего функции трансформации передаются свойству `transform` в виде набора значений, разделенных пробелами. Согласно спецификации, первой обрабатывается функция трансформации, объявляемая в начале (крайняя левая), а последней — указанная в конце (крайняя правая) последовательности значений свойства. Если хотя бы одна из функций имеет неверный синтаксис, то отменяется сразу вся команда `transform`, начиная с самого первого эффекта трансформации.

Функции смещения

Трансформация смещения заключается в изменении положения элемента вдоль одной из осей координатного пространства. В частности, функция `translateX()` перемещает элемент вдоль оси X, функция `translateY()` смещает его вдоль оси Y, а функция `translateZ()` — вдоль оси Z.

Функция	Допустимые значения
`translateX()`, `translateY()`	*<length>* | *<percentage>*

Эти функции относятся к инструментам двухмерной трансформации, поскольку обеспечивают позиционирование элемента только в плоскости, образованной осями X и Y, и не позволяют смещать его вдоль оси Z. Каждая из них принимает по одному числовому значению, выраженному в единицах измерения длины или в процентах.

Проще всего смещение элемента рассчитывается при указании расстояния в единицах длины. В частности, для его перемещения на 200 пикселей вправо вдоль оси X потребуется функция `translateX(200px)`, а для смещения на такое же расстояние в противоположном направлении — функция `translateX(-200px)`. Соответственно передача функции `translateY()` положительного значения приведет к смещению элемента вниз, а отрицательного значения — к смещению вверх. Если перед смещением объект был повернут на 180°, то для его смещения вверх нужно передать функции `translateY()` положительное значение, а для смещения вниз — отрицательное.

Вычисление расстояния смещения, указанного процентным значением, выполняется относительно соответствующего размера самого элемента. Таким образом, функция `translateX(50%)`, примененная к элементу шириной 300 и высотой 200 пикселей, обязывает сместить его на 150 пикселей вправо. При этом функция `translateY(-10%)` обеспечивает смещение такого объекта на 20 пикселей вверх.

Функция	Допустимые значения
`translate()`	[*<length>* | *<percentage>*] [,*<length>* | *<percentage>*]?

Для одновременного смещения элемента вдоль осей X и Y применяется функция `translate()`. Она принимает два значения: первое определяет расстояние смещения вдоль горизонтальной оси, а второе задает расстояние смещения для вертикальной оси. Таким образом, это свойство заменяет сразу оба описанных выше свойства: `translateX()` `translateY()`. При опускании второго значения оно приравнивается нулю. Следовательно, объявление `translate(2em)` рассматривается как `translate(2em,0)`, в свою очередь равнозначное определению `translateX(2em)`. Примеры изменения положения элемента с помощью функций смещения приведены на рис. 16.9.

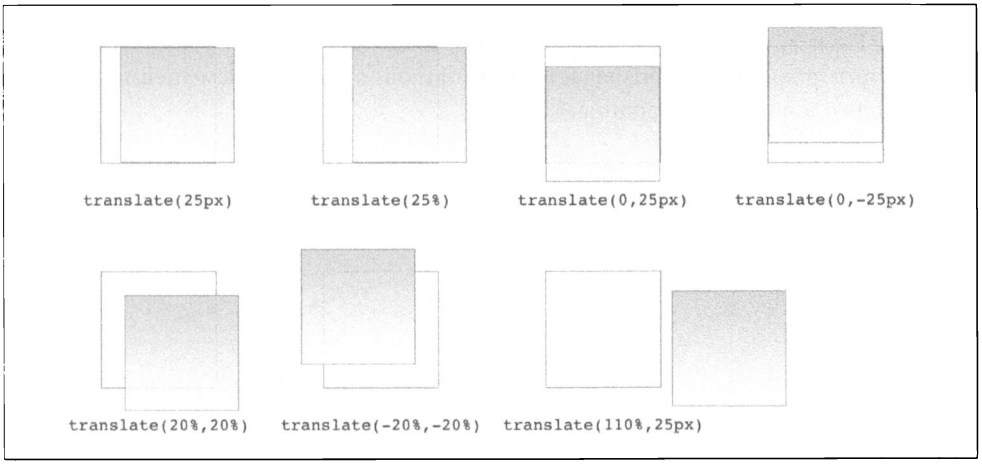


Рис. 16.9. Смещение элемента на плоскости

В последней версии спецификации функциям смещения вдоль горизонтальной и вертикальной осей допускается передавать безразмерные числовые значения. В подобных случаях такое значение рассматривается как указанное в *пользовательских единицах измерения* (пиксели, если не указано иное). При этом спецификация CSS, в отличие, например, от спецификации SVG, не оговаривает способ определения пользовательских единиц измерения. На момент завершения работы над книгой ни один из браузеров не был протестирован на предмет корректности обработки безразмерных величин смещения, поэтому данная возможность представляется сугубо академической.

Функция	Допустимое значение
`translateZ()`	*<length>*

Эта функция смещает элемент вдоль оси Z, изменяя глубину его расположения в стеке. В отличие от функций смещения на плоскости она принимает только значения, выраженные в единицах измерения длины, и не допускает обработку процентных значений, поскольку они не используются в координатном пространстве оси Z.

Функция	Допустимые значения
`translate3d()`	`[<length> | <percentage>], [<length> | <percentage>], [<length>]`

Являясь функцией общего назначения и напоминая функцию `translate()` по способу обработки горизонтальных и вертикальных координат, эта функция устанавливает смещение элементов вдоль всех трех координатных осей. Ее эффективность становится очевидной при произвольном изменении положения элемента в трехмерном пространстве. Последовательность смещения элемента показана на рис. 16.10. На нем направление смещения вдоль каждой из осей указывается стрелками, а пунктирная линия обозначает траекторию и расстояние смещения (исходная точка находится в начале координат).

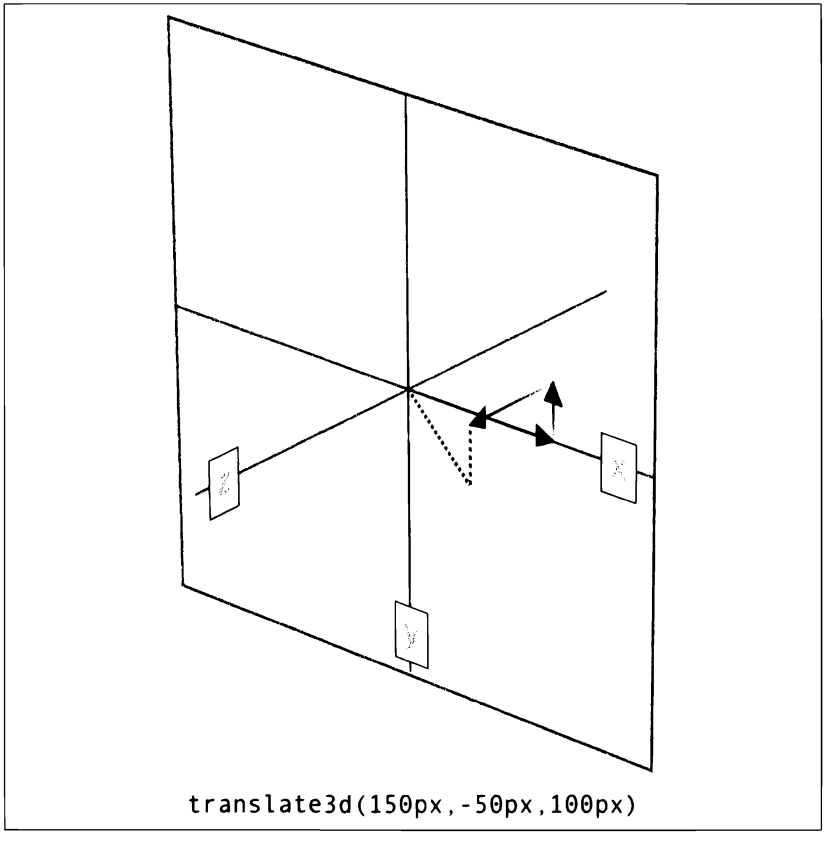


Рис. 16.10. Смещение элемента во всех трех направлениях

В отличие от функции `translate()`, работоспособность функции `translate3d()` обеспечивается за счет передачи ей всех значений, обозначенных в определении. В частности, команда `translate3d(1em,-50px)` не будет распознаваться обработчиком, а для получения значимого результата ее нужно заменить объявлением `translate3d(2em,-50px,0)`.

Функции масштабирования

Функции масштабирования применяются для изменения размера элемента в зависимости от передаваемого им значения. Коэффициент увеличения или уменьшения элемента представляется положительным безразмерным действительным числом. В двухмерном пространстве масштабирование элемента может осуществляться отдельно для горизонтальной или вертикальной оси либо выполняться сразу для обоих направлений.

Функция	Допустимое значение
`scaleX(),scaleY(),scaleZ()`	*<number>*

В качестве аргумента функциям передается всего одно числовое значение: множитель (коэффициент масштабирования). Следовательно, значение `scaleX(2)` обязывает увеличить ширину элемента в два раза, а значение `scaleY(0.5)` двукратно уменьшает его высоту. Учитывая специфику аргументов функции масштабирования, можно предположить, что в их качестве допускается использовать процентные значения, хотя на самом деле они не поддерживаются спецификацией.

Функция	Допустимое значение
`scale()`	*<number>* `[,` *<number>* `]?`

Функция `scale()` применяется для одновременного масштабирования элемента как в горизонтальном, так и в вертикальном направлении. Ее первый аргумент всегда указывает степень изменения размера вдоль оси X, а второй аргумент — этот же параметр для оси Y. Таким образом, функция `scale(2,0.5)` позволяет двукратно расширить элемент и двукратно уменьшить его высоту. При передаче функции всего одного значения оно будет представлять коэффициент масштабирования для каждого из двух направлений. Следовательно, значение `scale(2)` определяет двукратное увеличение и ширины, и высоты элемента. Такое поведение отличает функцию масштабирования от функции `translate()`, в которой упущенный аргумент представляется нулевым значением. Согласно спецификации, функция `scale(1)` полностью соответствует функции `scale(1,1)`, обеспечивая неизменность исходного размера элемента в обоих направлениях. Конечно, в таком объявлении мало смысла, но именно так работает команда масштабирования в CSS.

Несколько примеров изменения размера элемента с помощью функций масштабирования приведено на рис. 16.11.

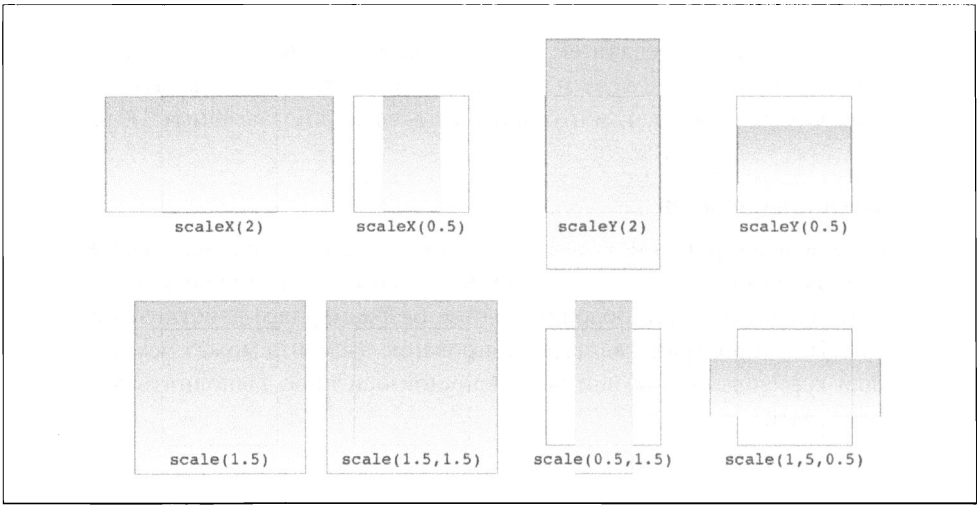


Рис. 16.11. Масштабирование элементов

Масштабирование не ограничивается двумя измерениями: его также можно выполнять вдоль оси Z. В CSS такая операция выполняется с помощью функции `scaleZ()`. Наряду с этим функция `scale3d()` позволяет указывать размер элемента вдоль всех трех осей координатного пространства. Данные функции стоит применять только при трансформации элементов по глубине документа, которая не выполняется в CSS по умолчанию. Заметьте, что масштабирование вдоль оси Z, требующее применения функции `scaleZ()` или `scale3d()`, возможно только при предварительной трансформации (например, вращении) элемента относительно осей X и Y.

Функция	Допустимое значение
`scale3d()`	*<number>*, *<number>*, *<number>*

Как и `translate3d()`, функция `scale3d()` требует передачи сразу трех аргументов. В противном случае она не обрабатывается, что приводит к отмене команды трансформации, в которую она включена.

Функции вращения

Функция вращения обеспечивает поворот элемента вокруг одной из осей или специально заданного трехмерного вектора. Всего в CSS включены четыре базовые функции вращения и еще одна функция общего назначения для настройки вращения в трехмерном пространстве.

Функция	Допустимые значения
`rotate()`, `rotateX()`, `rotateY()`, `rotateZ()`	*<angle>*

Базовые функции вращения снабжаются всего одним аргументом, определяющим угол поворота элемента. Он задается в виде положительного или отрицательного числового значения, выраженного в допустимых единицах измерения (`deg`, `grad`, `rad` и `turn`). При передаче функциям вращения значения, не входящего в диапазон допустимых координат, оно нормализуется до минимально возможной числовой величины. Иными словами, значение `437deg` будет представляться как `77deg` или `-283deg`.

Учтите, что нормализация угловых значений выполняется только в отсутствие анимации вращения. В частности, анимация вращения на угол `1100deg` требует вращения элемента вокруг заданной оси на несколько оборотов, последний из которых заканчивается на отметке `-20deg` (или, что то же самое, `340deg`). В то же время анимация поворота на угол `-20deg` выражается всего лишь в незначительном наклоне элемента, но никак не его вращении. При этом анимация вращения на угол `340deg` осуществляется в противоположном направлении и заключается в почти полном обороте элемента вокруг заданной оси. В каждом из трех случаев анимация завершается в одной и той же пространственной точке, но существенно отличается воспроизводимым эффектом.

Функция `rotate()` применяется для поворота элементов в двухмерном пространстве (наиболее распространенный вариант) и во многом напоминает функцию `rotateZ()`, отвечающую за вращение элемента вокруг оси Z (направлена перпендикулярно поверхности экрана). Подобным образом функция `rotateX()` обеспечивает поворот элемента вокруг оси X (наклон от и на наблюдателя), а функция `rotateY()` отвечает за его поворот вокруг оси Y (подобно дверям с осью вращения посередине полотна), как показано на рис. 16.12.

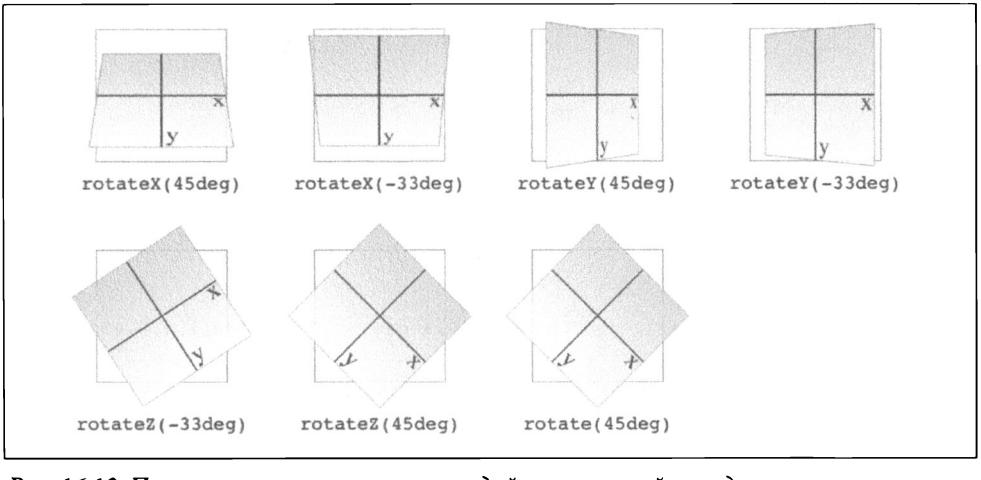


Рис. 16.12. Поворот элемента вокруг каждой из трех осей координатного пространства

Некоторые из примеров, приведенных на рис. 16.12, обеспечивают вращение элемента в трехмерном пространстве. Но это возможно только при передаче свойствам `transform-style` и `perspective` строго заданных значений, описанных в разделах "Стиль трехмерной трансформации" и

"Изменение перспективы". Данное замечание справедливо для всех без исключения случаев трехмерной трансформации элементов, а не только обозначенных выше примеров. Не забывайте об этом — попытка трансформации объектов без уточнения значений свойств `transform-style` и `perspective` обычно приводит к получению результатов, отличных от показанных на рис. 16.12.

Функция	Допустимое значение
`rotate3d()`	*<number>*, *<number>*, *<number>*, *<angle>*

Использование этой функции для вращения элемента в трехмерном пространстве требует знакомства с линейной алгеброй. Ее первых три аргумента определяют проекции вектора на оси X, Y и Z, а последнее числовое значение, выраженное в угловых единицах измерения, указывает количество оборотов, совершаемых элементом вокруг вектора вращения.

Знакомство с возможностями функции `rotate3d()` лучше всего начать с простого примера. В полном виде функция `rotate(45deg)` представляется как `rotate3d(0,0,1,45deg)`, определяя поворот элемента вокруг вектора, совмещенного с осью Z, на что указывают нулевые проекции на оси X и Y. Согласно единственному явно заданному аргументу, такая функция обеспечивает поворот элемента в плоскости экрана на 45° (рис. 16.13). Остальных два примера призваны продемонстрировать формат функций, обеспечивающих поворот элемента вокруг осей X и Y.

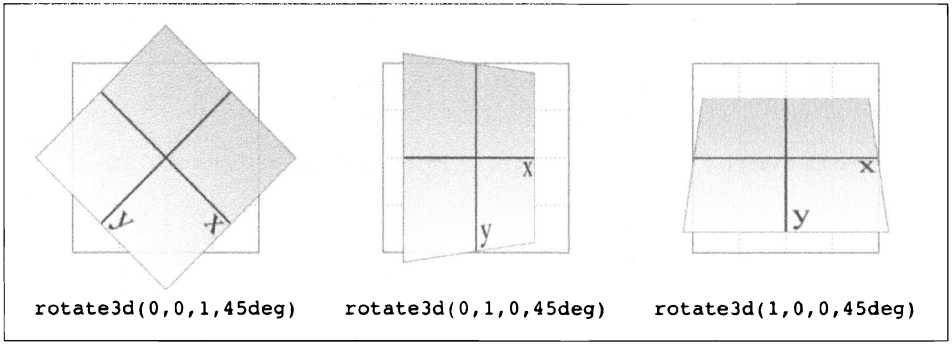


Рис. 16.13. Вращение элемента в трехмерном пространстве

Несколько сложнее проанализировать функцию, все три аргумента которой представлены произвольными числовыми значениями, например `rotate3d(-0.95, 0.5,1,45deg)`. В подобных случаях вектор вращения не совмещается ни с одной из осей координатного пространства. Чтобы понять, как он расположен, обратимся к предыдущему примеру, который получен в результате выполнения функции `rotateZ(45deg)`, функционально идентичной функции `rotate3d(0,0,1,45deg)`. В этой записи направленность вектора определяют первые три числовые величины. Очевидно, что вектор характеризуется нулевой направленностью вдоль осей X и Y, а

также максимально возможной проекцией на ось Z, т.е. перпендикулярно поверхности экрана. В таком представлении вращение элемента осуществляется исключительно в плоскости экрана.

Подобным образом определяется направление вращения, задаваемое функцией `rotateX(45deg)`, которая в полном виде представляется записью `rotate3d(1,0,0,45deg)`. Вращение выполняется вокруг оси X — передача положительных углов соответствует повороту по часовой стрелке, а отрицательных углов — против часовой стрелки. Направление вращения определяется для наблюдателя, рассматривающего вектор сверху вниз (от конечной точки к начальной). Для пользователя, просматривающего документ на мониторе, вращение представляется поворотом верхней части элемента вглубь экрана, а нижней — наружу из него.

Теперь перейдем к анализу более сложной функции: `rotate3d(1,1,0,45deg)`. С точки зрения пользователя, просматривающего документ на мониторе, вектор, описываемый этой функцией, будет начинаться в левом верхнем углу, проходить через центр и заканчиваться в правом нижнем углу элемента. Схематически его можно представить как стрелку, пронзающую прямоугольник элемента под углом 45° сверху вниз. Направление вращения определяется так же, как и в предыдущем случае, поэтому верхняя часть элемента будет отдаляться от наблюдателя, а нижняя — приближаться к нему. Если увеличить угол вращения еще больше — до 90° (`rotate3d(1,1,0,90deg)`), то элемент повернется ребром к наблюдателю, оставаясь наклоненным под углом 45° в плоскости экрана, а его "лицевая" сторона будет обращена к ее правому верхнему углу. Чтобы лучше понять, как происходит такое вращение, возьмите лист бумаги, нарисуйте на нем стрелку, направленную из левого верхнего в правый нижний угол листа, и поверните его вокруг нее на 90°.

Разобравшись с более простыми задачами, вернемся к визуализации действия, обусловливаемого исходной функцией: `rotate3d(-0.95,0.5,1,45deg)`. В области пространства кубической формы шириной, высотой и глубиной 200 пикселей горизонтальный размер вектора вращения составит 190 пикселей, а его вертикальный размер будет равен 100 пикселям, проникая на всю ее глубину (200 пикселей). Следовательно, начальная координата такого вектора будет представлена точкой `(0, 0, 0)`, а конечная — точкой `(-190px, 100px, 200px)`. Вектор, описываемый приведенной выше функцией, и результат ее применения к элементу документа показаны на рис. 16.14.

Вектор всегда можно представить как тонкую ось, пронизывающую элемент насквозь. Как и ранее, направление вращения определяется для наблюдателя, рассматривающего вектор сверху вниз (от конечной к начальной точке). Исходя из сложной направленности вектора (одновременно влево, вниз и в глубину), поворот на 45° будет осуществляться так, что левый верхний угол элемента будет приближаться, а правый нижний — отдаляться от наблюдателя, сидящего перед экраном.

Заметьте: результат выполнения функции `rotate3d(1,1,0,45deg)` *не* равнозначен результату, получаемому при выполнении последовательности функций `rotateX(45deg) rotateY(45deg) rotateZ(0deg)`! Это очень распространенная ошибка, которую допускает огромное количество авторов, в том числе и разработчиков обучающих занятий. Чтобы удостовериться в этом, достаточно нарисовать векторы,

получаемые в каждом из случаев в упомянутой выше области кубического пространства размером 200×200×200 пикселей. Для функции `rotate3d(1,1,0,45deg)` вектор вращения будет направлен из исходной точки в точку, отстоящую от нее на 200 пикселей вправо и 200 пикселей вниз (`200, 200, 0`). Таким образом, такой вектор будет пронизывать элемент по диагонали: из его левого верхнего в правый нижний угол. При повороте на 45° по часовой стрелке (при наблюдении от конечной к начальной точке вектора) правый верхний угол элемента будет углубляться в экран и смещаться влево, а левый нижний угол — наоборот, выступать вперед и вправо. Очевидно, что команда `rotateX(45deg) rotateY(45deg) rotateZ(0deg)` описывает несколько иное поведение элемента (рис. 16.15).

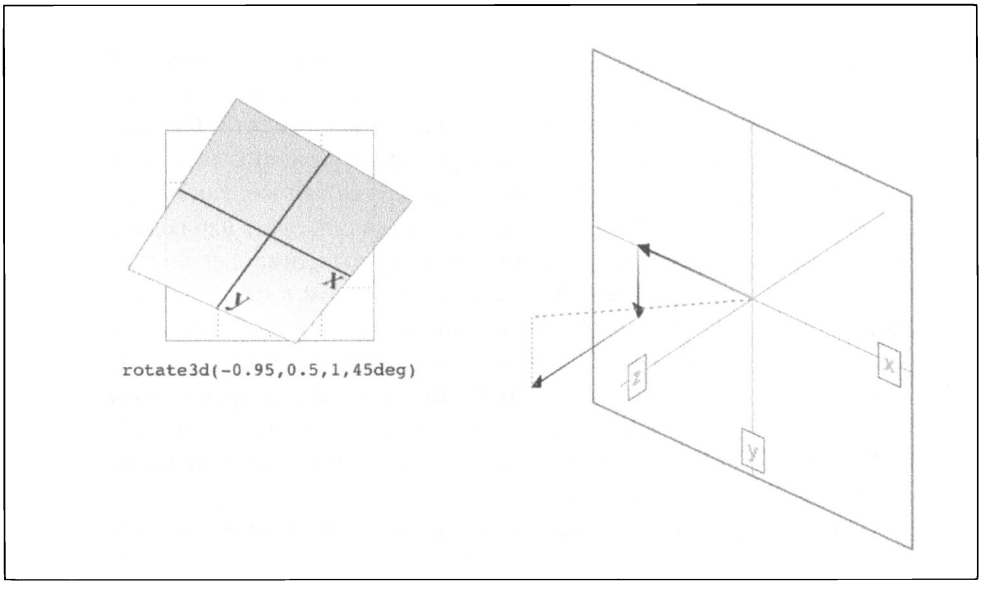


Рис. 16.14. Расположение вектора в трехмерном пространстве и вращение элемента вокруг него

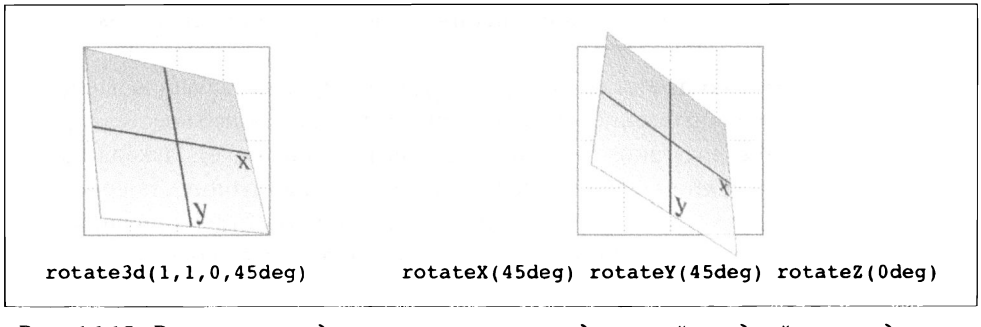


Рис. 16.15. Различие между вращением вокруг двух осей и одной оси, заданных трехмерными векторами

Функции наклона

При наклоне элемент перекашивается в горизонтальном и/или вертикальном направлении. CSS не предусматривает возможности наклона элементов вдоль оси Z.

Функция	Допустимое значение
`skewX(),skewY()`	*<angle>*

В качестве единственного аргумента каждая из функций получает числовое значение, представляющее угол наклона элемента. Познакомиться с ними намного проще, изучая реальные примеры, а не пытаясь разобраться с определением, заданным в спецификации. На рис. 16.16 показано несколько способов наклона элементов в плоскости X и Y.

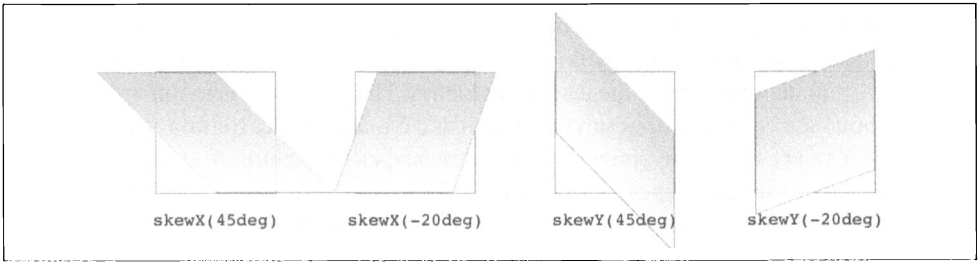


Рис. 16.16. Наклон, заданный относительно горизонтальной и вертикальной осей

Функция	Допустимые значения
`skew()`	*<angle>* `[,` *<angle>* `]?`

Результат выполнения функции `skew(a,b)` заметно отличается от получаемого при обработке последовательности функций `skewX(a)` и `skewY(b)`. В данном случае для вычисления наклона в двухмерном пространстве применяется матрица вида `[ax,ay]`. Результат ее использования для вычисления наклона элемента, а также отличие от результата, получаемого при последовательном наклоне этого же элемента отдельно для каждой из осей, показаны на рис. 16.17.

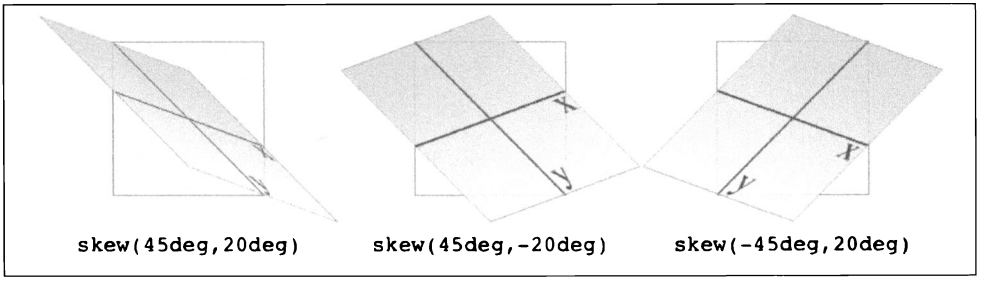


Рис. 16.17. Наклон элемента

Первый аргумент функции `skew()` всегда обозначает наклон относительно оси X, а второй аргумент — наклон относительно оси Y. Если опустить второе значение, то наклон относительно оси Y осуществляться не будет.

Функции изменения перспективы

Очень часто при трансформации в трехмерном пространстве элементу требуется придать глубину. В CSS глубина визуализации обеспечивается перспективой, которая может настраиваться отдельно для каждого из элементов.

Функция	Допустимое значение
`perspective()`	*<length>*

Хоть это и звучит странно, но в CSS перспектива определяется расстоянием. Но каким образом должна обрабатываться команда `perspective(200px)`, если глубина изображения, откладываемая вдоль оси Z, в пикселях не выражается? Ответ становится очевидным, если знать, что функция `perspective()` создает всего лишь иллюзию глубины, не добавляя ее саму к целевому элементу. Небольшие значения аргумента позволяют добиться эффекта просмотра элемента с близкого расстояния через объектив типа "рыбий глаз". Большие значения создают эффект удаленной визуализации элемента, рассматриваемого через телеобъектив. И только при *очень больших* значениях перспективы элемент будет представляться в полностью изометрической проекции.

Такой подход как нельзя лучше подходит для представления пространственных форм в плоскости экрана. Если обозначить эффект передачи глубины изображения элемента в виде пирамиды, то ее вершина, расположенная в точке схода, будет находиться на расстоянии перспективы от ее основы. Чем меньше расстояние перспективы, тем более пологой будет пирамида и тем большему искажению будет подвергаться изображение элемента. На рис. 16.18 показано, как формируется глубина изображения для элементов с расстоянием перспективы 200, 800 и 2000 пикселей.

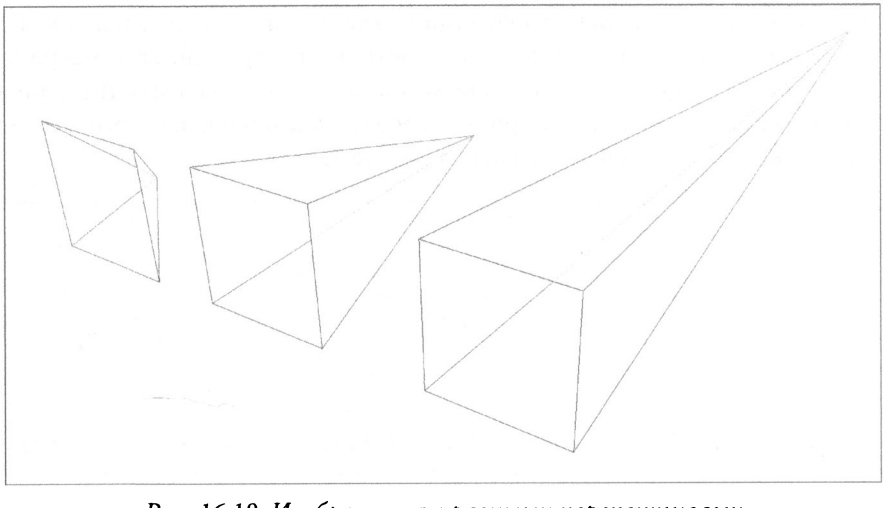


Рис. 16.18. Изображения с разными перспективами

В документации к браузеру Safari (`http://bit.ly/safari-2d-3d-transforms`) указывается, что перспектива менее `300px` сильно искажает изображение объектов; незначительному искажению подвергаются изображения элементов с перспективой более `2000px`, а средний уровень искажения соответствует перспективе от 500 до 1000 пикселей. Подтверждением этого утверждения служит рис. 16.19, на котором показан результат изменения перспективы у элемента, исходно повернутого на определенный угол.

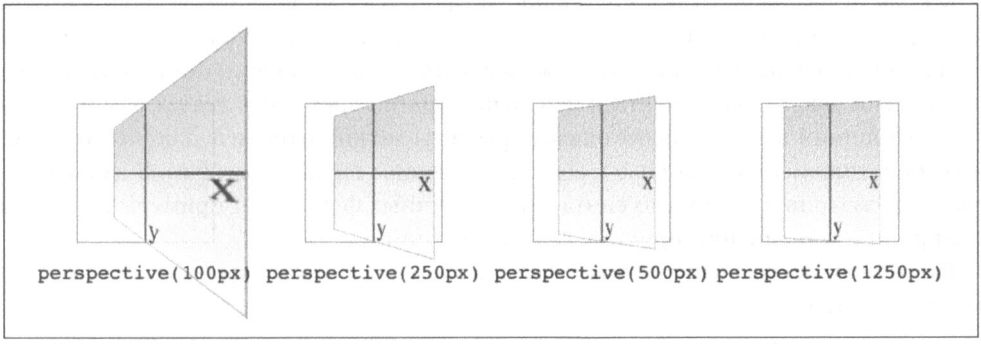


Рис. 16.19. Изменение перспективы у одного и того же элемента

Расстояние перспективы всегда представляется ненулевым положительным числом. Передача функции `perspective()` иных значений приводит к отмене трансформации. Не забывайте о том, что получаемый результат напрямую зависит от положения функции в последовательности команд трансформации. В частности, примеры, приведенные на рис. 16.19, получены при выполнении функции `perspective()` после функции `rotateY()`.

По своему действию функция `perspective()` во многом повторяет эффект применения свойства `perspective`, описанного далее. Несмотря на схожесть, они применяются совершенно разными способами. В большинстве случаев функцию `perspective()` можно смело заменять свойством `perspective`, хотя и из этого правила существуют исключения.

Матричные функции

Если вы сторонник точных вычислений и обладаете богатыми познаниями в высшей математике, вам будет приятно узнать, что в CSS трансформацию элементов можно выполнять с помощью матричных функций.

Функция	Допустимые значения
`matrix()`	*<number>* `[,` *<number>* `]{5,5}`

В спецификации CSS команда `matrix()` весьма однозначно определяется как "функция двухмерной трансформации объектов с помощью матрицы преобразований для шести значений *a–f*".

Первая особенность функции `matrix()` заключается в ее аргументах: их шесть, они разделяются запятой, представляются положительными или отрицательными числовыми значениями и все обязательны для передачи. Ее вторая особенность — необычайная компактность записи значений, представляющих команды трансформации элемента (поворот, наклон и т.п.). Третья особенность условна: по вполне понятным причинам матричными функциями пользуются лишь очень немногие авторы документов.

Мы не будем вдаваться в подробности процесса вычислений, лежащих в основе матричных преобразований, так как они будут понятны только тем из вас, кому посчастливилось прослушать курс высшей математики. Для всех остальных знакомство с ними будет представляться напрасной тратой времени. Следует отметить, что на сегодняшний день все необходимые расчеты можно относительно легко выполнить, обратившись за помощью к соответствующим онлайн-службам. В дальнейшем будут рассматриваться только синтаксис матричных функций и примеры их использования при выполнении простых трансформаций.

Рассмотрим следующий пример функции `matrix()`, применяемой к одному элементу документа:

```
matrix(0.838671, 0.544639, -0.692519, 0.742636, 6.51212, 34.0381)
```

Исходя из списка аргументов, переданных функции, матрица преобразований элемента имеет такой вид:

0.838671	-0.692519	0	6.51212
0.544639	0.742636	0	34.0381
0	0	1	0
0	0	0	1

На этом этапе анализ команды не представляет особых затруднений, чего не скажешь об анализе значений, представленных в матричном виде. Результат их обработки, равнозначный получаемому при выполнении следующей последовательности функций, представлен на рис. 16.20:

```
rotate(33deg) translate(24px,25px) skewX(-10deg)
```

Главный вывод, который следует из приведенных выше рассуждений, очень простой. Если вы знакомы с матричными преобразованиями, то можете и обязательно должны применять их при трансформации элементов с помощью CSS. Если же они слишком сложны для вашего понимания, то применяйте для трансформации элементов общепринятые функции, рассмотренные в предыдущих разделах.

Наряду с двухмерной трансформацией матричные преобразования можно задействовать для расчета трансформации элементов документа в трехмерном пространстве. Для этих целей в CSS также предусмотрена отдельная функция.

Функция	Допустимые значения
`matrix3d()`	*<number>* `[,` *<number>* `]{15,15}`

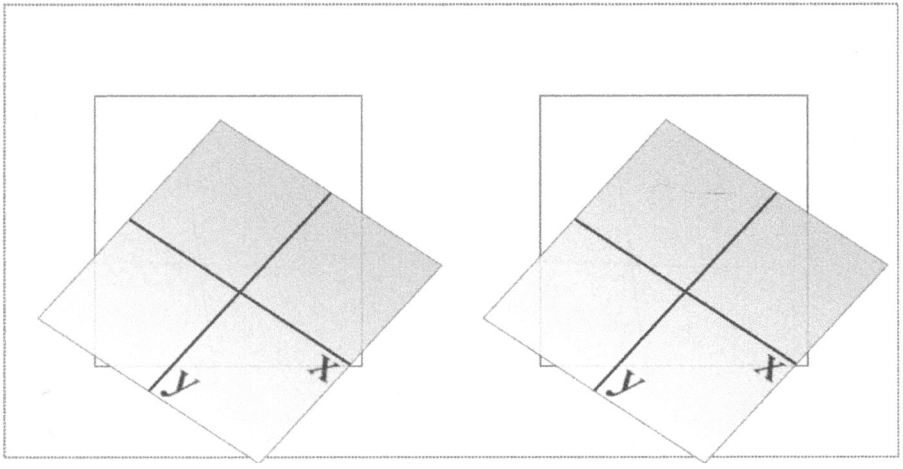


Рис. 16.20. Двухмерная трансформация элемента с помощью матрицы преобразований и последовательности равнозначных команд

Как и в предыдущем случае, рассмотрим, как определяется функция `matrix3d()` в спецификации CSS. В ней указано, что эта функция "выполняет трансформацию элемента в трехмерном пространстве согласно однородной матрице преобразований формата 4×4, значения которой организованы в колонки". Из такого определения можно сделать вывод, что функция `matrix3d()` должна снабжаться 16 обязательными разделяемыми запятыми аргументами, организованными в четыре колонки, каждая из которых содержит ровно четыре значения. Таким образом, первые четыре значения матрицы заносятся в первый столбец аргументов функции, последующие четыре значения — во второй столбец и т.д. В качестве примера рассмотрим такую функцию.

```
matrix3d(
   0.838671, 0, -0.544639, 0.00108928,
   -0.14788, 1, 0.0960346, -0.000192069,
   0.544639, 0, 0.838671, -0.00167734,
   20.1281, 25, -13.0713, 1.02614)
```

Матрица преобразований, соответствующая этой функции, имеет следующий вид:

0.838671	-0.14788	0.544639	20.1281
0	1	0	25
-0.544639	0.0960346	0.838671	-13.0713
0.00108928	-0.000192069	-0.00167734	1.02614

Равнозначный результат (рис. 16.21) можно получить, применив приведенную ниже последовательность команд трансформации:

```
perspective(500px) rotateY(33deg) translate(24px,25px) skewX(-10deg)
```

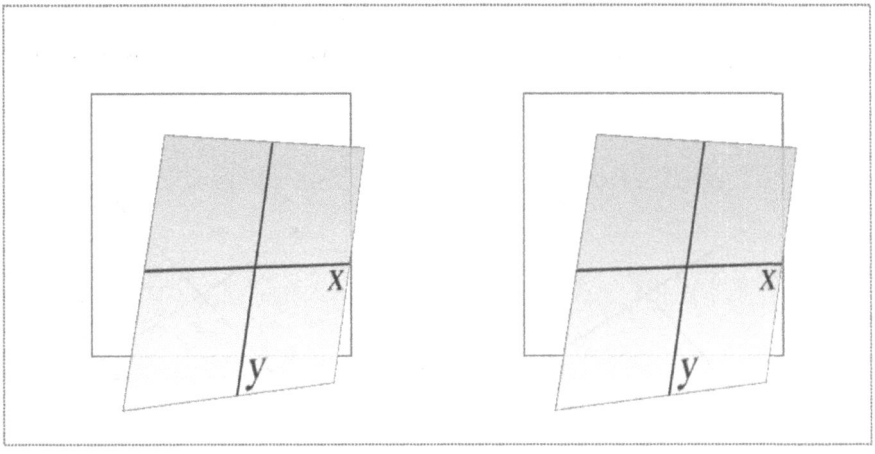


Рис. 16.21. Трехмерная трансформация элемента с помощью матрицы преобразований и последовательности равнозначных команд

Равнозначный набор функций трансформации

При подборе равнозначного набора функций трансформации для функции `matrix3d()` помните о том, что он обеспечивает идентичность только конечного состояния целевого элемента. Сходная ситуация уже рассматривалась нами в разделе, содержащем описание функций вращения: на первый взгляд, угол поворота `393deg` можно представить значением `33deg`, но только при анимации вращения. В последнем случае аргумент функции определяет не только угол поворота, но и количество оборотов вокруг целевой оси. Очевидно, что значение `33deg` такую возможность не предоставляет. Подобным образом функция `matrix()` не позволяет узнать о промежуточных состояниях элемента и определяет исключительно конечный результат трансформации, выполняемой наиболее оптимальным образом.

Чтобы понять, о чем идет речь, рассмотрим пример трансформации, осуществляемой двумя равнозначными способами.

```
rotate(200deg) translate(24px,25px) skewX(-10deg)
matrix(-0.939693, -0.34202, 0.507713, -0.879385, -14.0021, -31.7008)
```

Согласно приведенным выше командам, первой выполняется операция поворота. Положительный аргумент предполагает поворот элемента на 200° по часовой стрелке, что справедливо в обоих случаях. Отличия проявляются, как только операция поворота становится анимированной: в случае применения последовательности регулярных функций поворот, как и ранее, выполняется на 200° по часовой стрелке, в то время как функция `matrix()` выполняет это же действие через поворот на 160° против часовой стрелки. Конечный результат одинаков в обоих способах трансформации, но он достигается совершенно разными действиями.

Аналогичные отличия могут проявляться в самых неожиданных ситуациях — в первую очередь, благодаря особенности функции `matrix()` оптимизировать действия по достижению требуемого результата. В противоположность ей равнозначная последовательность команд трансформации такой способностью не отличается,

поскольку сводится к выполнению строго заданного набора действий. Рассмотрим еще один пример трансформации, выполняемой равнозначными способами.

```
rotate(160deg) translate(24px,25px) rotate(-30deg) translate(-100px)
matrix(-0.642788, 0.766044, -0.766044, -0.642788, 33.1756, -91.8883)
```

Как и ранее, результат в обоих случаях будет одинаковым. Но в случае анимации он будет представляться иными действиями. Конечно же, по записи команд это не очевидно, но результат говорит сам за себя.

Описанные выше особенности функций трансформации нужно учитывать практически в каждом проекте, поскольку от обычного преобразования элементов до их анимации — всего один шаг. (Это утверждение станет полностью очевидным после знакомства с инструментами анимации CSS.)

Другие свойства трансформации

Наряду со свойством `transform` трансформация элементов документа может потребовать использования некоторых других стилевых инструментов, преимущественно вспомогательного назначения. Они позволяют настроить дополнительные параметры трансформации: положение исходной точки трансформации или перспективу области просмотра.

Положение исходной точки трансформации

Все рассмотренные выше способы преобразования имеют общую характеристику: *исходную точку* трансформации, которая по умолчанию располагается по центру элемента. Например, вращение элемента обычно осуществляется вокруг его центра, а не одного из углов. Такое поведение элемента можно изменить, воспользовавшись стилевым свойством `transform-origin`.

transform-origin

Значение	[`left` | `center` | `right` | `top` | `bottom` | *<percentage>* | *<length>*] | [`left` | `center` | `right` | *<percentage>* | *<length>*] && [`top` | `center` | `bottom` | *<percentage>* | *<length>*]] *<length>*?
Начальное значение	`50% 50%`
Применяется	Все элементы, подверженные трансформации
Процентное значение	Относительно размера ограничительной рамки (см. описание)
Вычисляется	Процент, за исключением значений, выраженных в единицах длины, которые преобразуются в абсолютные значения
Наследуется	Нет
Анимируется	*<length>*, *<percentage>*

Синтаксис этого свойства, выглядящий невероятно запутанно в определении, приведенном в спецификации, оказывается очень простым для понимания в реальных стилевых правилах. Исходная точка трансформации описывается в свойстве `transform-origin` двумя или тремя ключевыми словами: первое значение определяет

ее горизонтальную, а второе — вертикальную координату. Третье, необязательное значение, указывает глубину залегания исходной точки (координату на оси Z). В качестве первых двух значений допускается использовать ключевые слова, указывающие направление (например, `top` и `right`), процентные значения и числовые величины, выраженные в единицах измерения длины, а также всевозможные комбинации указанных типов данных. При этом координата на оси Z определяется только числовым значением, выраженным в единицах измерения длины, но не ключевым словом или процентной величиной — как правило, она измеряется в пикселях.

Расстояние, определяемое свойством, отсчитывается от левого верхнего угла элемента. Следовательно, объявление `transform-origin: 5em 22px` смещает исходную точку трансформации на `5em` влево (от левого края элемента) и на 22 пикселя вниз (от верхнего края элемента). Таким же образом объявление `transform-origin: 5em 22px -200px` смещает ее на `5em` влево, на 22 пикселя вниз и 200 пикселей вглубь (относительно исходно занимаемого уровня).

Процентные значения вычисляются относительно размера элемента вдоль соответствующей оси координатного пространства и задают величину смещения относительно левого верхнего угла элемента. Например, объявление `transform-origin: 67% 40%` указывает сместить исходную точку трансформации на 67% от ширины элемента вправо от ее левого края и на 40% от ее высоты вниз от ее верхнего края. Принцип вычисления положения исходной точки трансформации наглядно проиллюстрирован на рис. 16.22.

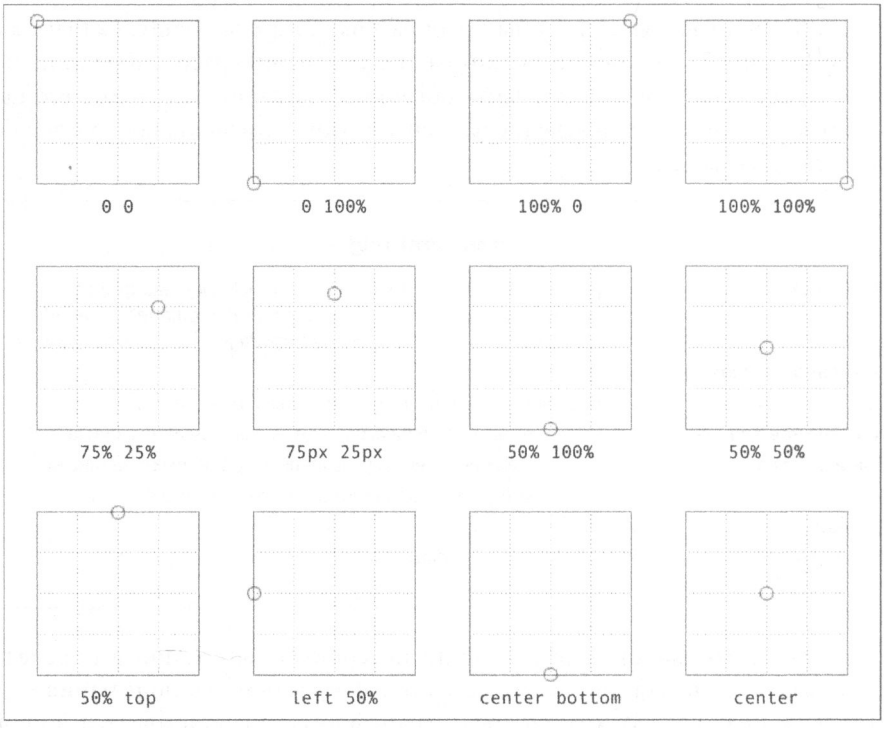


Рис. 16.22. Вычисление положения исходной точки трансформации

Теперь рассмотрим, какие изменения происходят в элементе при смещении исходной точки трансформации. Начнем изучение с более простого примера — преобразования элемента в двухмерном пространстве. Предположим, элемент поворачивается на 45° вправо (по часовой стрелке). Эффект влияния положения исходной точки на конечный результат такой операции продемонстрирован на рис. 16.23. (В каждом примере положение исходной точки трансформации обозначено кружком.)

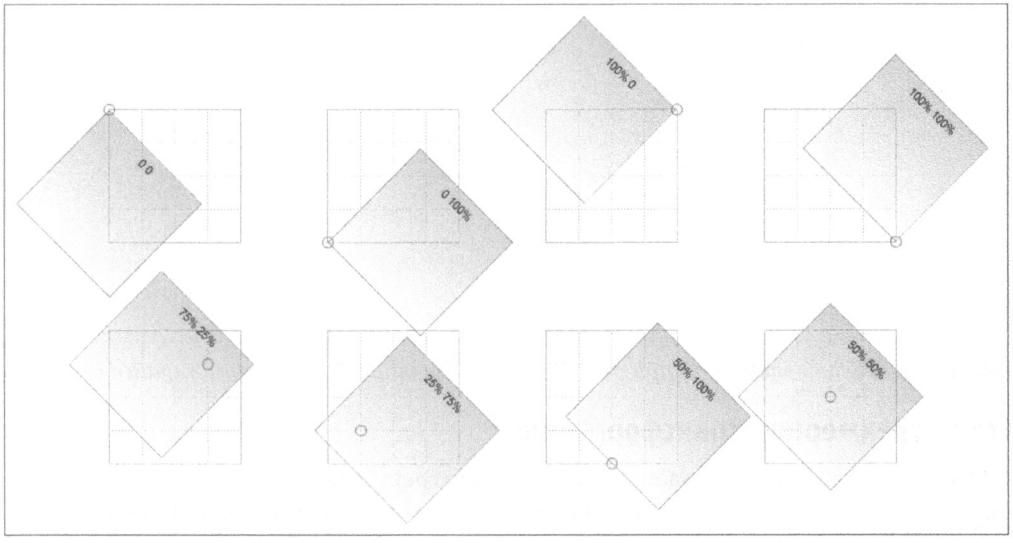


Рис. 16.23. Поворот элемента при изменении положения исходной точки трансформации

Положение исходной точки является также критически важной характеристикой при масштабировании и наклоне элементов. Если масштабирование элемента, исходная точка трансформации которого находится в его центре, характеризуется пропорциональным изменением размера сторон, то смещение ее в правый нижний угол будет обеспечивать вытягивание или сжатие элемента в направлении противоположного угла (по диагонали). При этом смещение исходной точки трансформации при наклоне элемента приводит только к изменению его конечного положения, но не характера скоса сторон. Несколько примеров наклона элемента при разных положениях исходной точки трансформации приведено на рис. 16.24.

Изменение положения исходной точки не сказывается на результате только одного типа трансформации, а именно смещения. Изменение положения элемента с помощью свойства `translate()`, а также его однонаправленных вариаций, `translateX()` и `translateY()`, выполняется одинаково, независимо от координаты исходной точки трансформации. Если таблица стилей не предполагает других типов преобразования элементов, то специально указывать исходную точку трансформации не имеет особого смысла. Если стилевое форматирование предполагает трансформацию элементов другими командами, то ее лучше задать в явном виде. Не пренебрегайте этой простой операцией, дабы избежать досадных неприятностей в будущем.

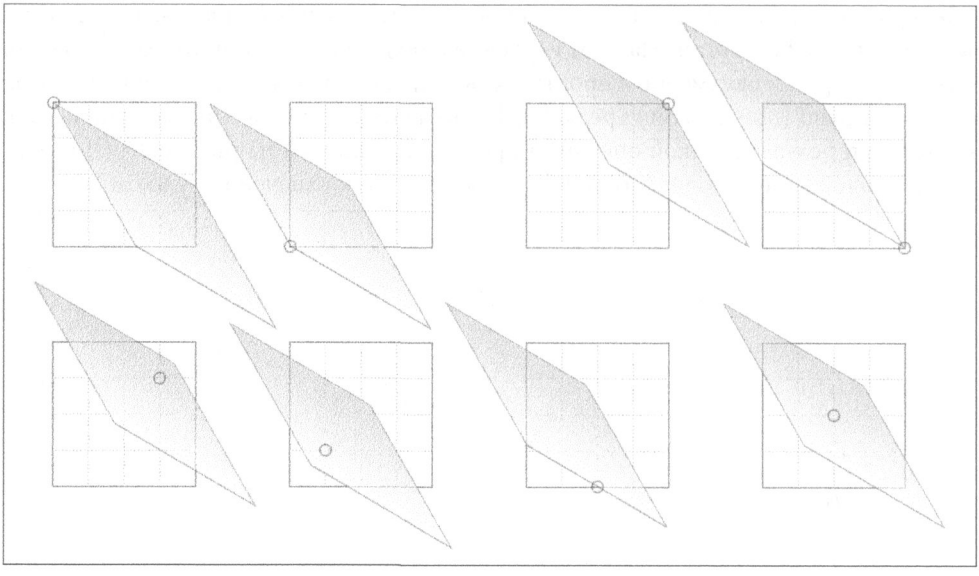


Рис. 16.24. Наклон элемента при разных положениях исходной точки трансформации

Стиль трехмерной трансформации

Подвергая элементы воздействию функций трехмерной трансформации, например `translate3d()` или `rotateY()`, вы ожидаете, что изменение их формы и положения будет выполняться в трехмерном пространстве. Как ни странно, такой способ визуализации требует специальной настройки. По умолчанию преобразование объекта выполняется в плоскости экрана, и для изменения такого поведения потребуется специальное свойство: `transform-style`.

transform-style

Значение	`flat` | `preserve-3d`
Начальное значение	`flat`
Применяется	Все элементы, подверженные трансформации
Вычисляется	Согласно определению
Наследуется	Нет
Анимируется	Нет

Чтобы понять, на что именно влияет стиль трансформации, рассмотрим пример, в рамках которого элемент сначала приближается к наблюдателю, а затем поворачивается и приобретает незначительную глубину просмотра (перспективу). Стилевое правило, отвечающее за выполнение столь комплексной задачи, имеет код, подобный приведенному ниже.

```
div#inner {transform: perspective(750px) translateZ(60px)
          rotateX(45deg);}
```

```
<div id="outer">
Внешн.
<div id="inner">Внутр.</div>
</div>
```

Результат его выполнения показан на рис. 16.25. Это примерно то, что мы и ожидали увидеть.

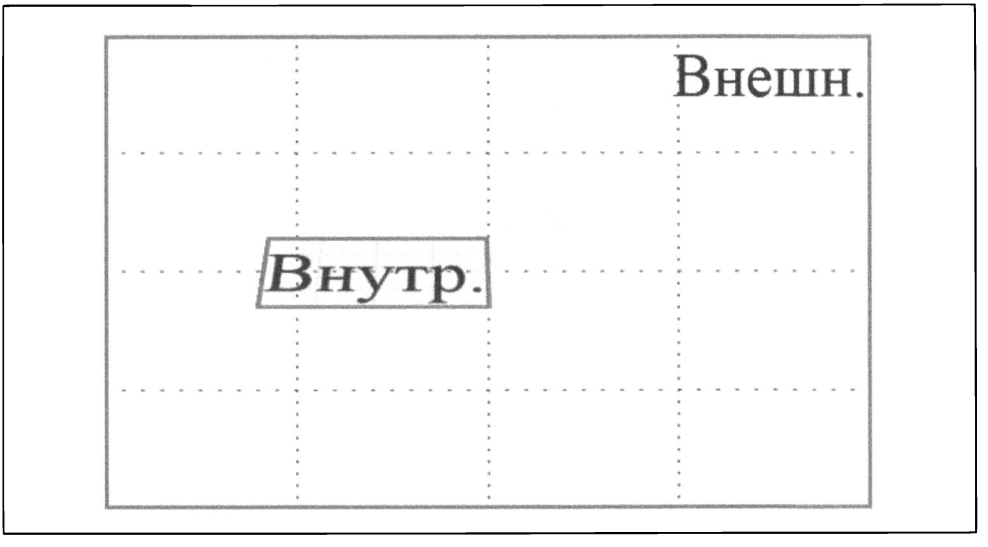


Рис. 16.25. Трехмерная трансформация внутреннего элемента `div`

Но как только трехмерной трансформации будет подвергнут внешний элемент `div`, эффект трансформации внутреннего элемента `div` будет полностью потерян. Последний будет представляться плоским изображением, расположенным на поверхности внешнего элемента `div`.

Именно такое поведение определяется по умолчанию свойством `transform-style`, получающим значение `flat`. Оно предполагает, что после трансформации внутренний элемент `div` будет визуализироваться исключительно в плоскости внешнего элемента `div`. Следовательно, вращение последнего приведет к получению картинки, подобной показанной на рис. 16.26.

```
div#outer {transform: perspective(750px) rotateY(60deg)
           rotateX(-20deg);}
div#inner {transform: perspective(750px) translateZ(60px)
           rotateX(45deg);}
```

Для изменения способа представления внутреннего элемента `div` свойству `transform-style` нужно передать значение `preserve-3d`. Такая настройка позволяет отобразить его в виде отдельного трехмерного объекта, а не как плоскую проекцию на поверхности родительского элемента. С изменениями в представлении элементов можно ознакомиться на рис. 16.27.

```
div#outer {transform: perspective(750px) rotateY(60deg)
           rotateX(-20deg); transform-style: preserve-3d;}
div#inner {transform: perspective(750px) translateZ(60px)
           rotateX(45deg);}
```

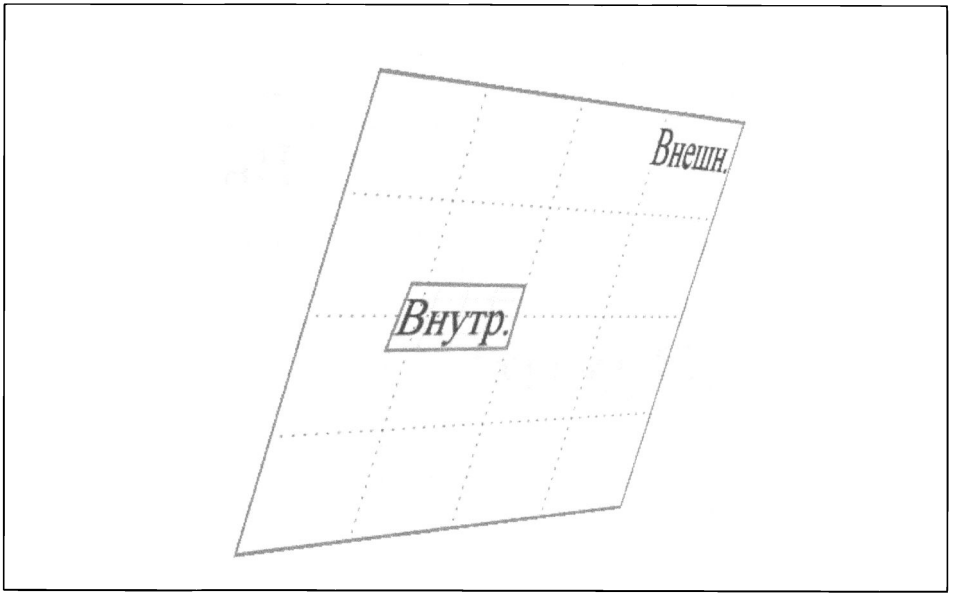


Рис. 16.26. Эффект расположения трансформируемого элемента в плоскости его родительского элемента

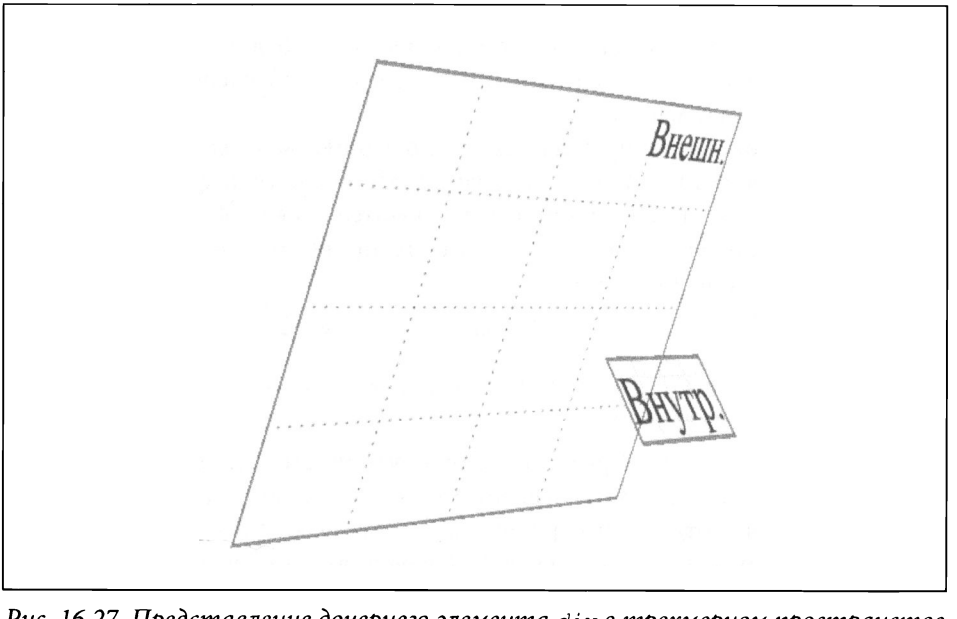


Рис. 16.27. Представление дочернего элемента `div` в трехмерном пространстве

Свойство `transform-style` может быть изменено по требованию некоторых других стилевых свойств. Причина подобного поведения кроется в необходимости приведения элемента или его дочерних элементов к "плоскому" виду перед применением данных свойств. Таким образом, свойство `transform-style` может иметь значение `flat` независимо от того, какое значение задано для него в явном виде.

Чтобы избежать перезаписи свойства `transform-style`, следите за тем, чтобы в коде таблицы стилей задавались следующие объявления:

- `overflow: visible`
- `filter: none`
- `clip: auto`
- `clip-path: none`
- `mask-image: none`
- `mask-border-source: none`
- `mix-blend-mode: normal`

Значения свойств, указанные в приведенных выше объявлениях, назначаются им по умолчанию. Следовательно, до тех пор пока они будут заданы в явном виде, трансформируемые элементы будут представляться в трехмерном виде. Если же в результате редактирования CSS-кода некоторые из таких элементов стали представляться плоской проекцией на своих родительских элементах, то, скорее всего, причиной тому служит установка одного из обозначенных выше свойств в значение, отличающееся от задаваемого по умолчанию.

И последнее замечание: чтобы предотвратить сброс трансформируемого элемента к "плоскому" виду, помимо приведенных выше ограничений нужно передать свойству `isolation` значение `isolate` (в явном или вычисляемом виде).

Изменение перспективы

В CSS установка перспективы элемента осуществляется с помощью двух стилевых свойств. Одно из них по своему действию напоминает функцию `perspective()`, рассмотренную в одном из предыдущих разделов, представляя перспективу через расстояние до точки схода (исходной точки). Настройка перспективы с помощью второго свойства выполняется путем изменения положения точки схода.

Установка перспективы для группы элементов

Сначала рассмотрим, каким образом изменяется расстояние перспективы с помощью свойства `perspective`. На первый взгляд, оно выполняет те же действия, что и одноименная функция, хотя между ними есть достаточно существенные различия.

perspective	
Значение	none | *<length>*
Начальное значение	none
Применяется	Все элементы, подверженные трансформации
Вычисляется	Абсолютное значение, выраженное в единицах длины, или none
Наследуется	Нет
Анимируется	Да

Установка свойства `perspective` в очень большое значение, например `2500px`, позволяет создать иллюзию просмотра изображения элемента через телеобъектив. Небольшая глубина изображения, сходная с получаемой с помощью "рыбьего глаза", достигается за счет передачи этому свойству небольших числовых значений, не превышающих `200px`.

Настало время разобраться в отличиях свойства `perspective` от функции `perspective()`. Все очень просто: функция применяется для добавления эффекта перспективы всего к одному элементу — тому, к которому она применяется. Таким образом, объявление `transform: perspective(800px) rotateY(-50grad);` справедливо только для элементов, на которые указывает селектор целевого правила.

С другой стороны, перспектива, обозначаемая свойством `perspective`, назначается не только целевому элементу, но и всем его дочерним элементам. Неожиданно, не правда ли? Отличия в способах наглядно проиллюстрированы на рис. 16.28.

```
div {transform-style: preserve-3d; border: 1px solid gray;
     width: 660px;}
img {margin: 10px;}
#one {perspective: none;}
#one img {transform: perspective(800px) rotateX(-50grad);}
#two {perspective: 800px;}
#two img {transform: rotateX(-50grad);}

<div><img src="rsq.gif"><img src="rsq.gif"><img src="rsq.gif"></div>
<div id="one"><img src="rsq.gif"><img src="rsq.gif">
   <img src="rsq.gif"></div>
<div id="two"><img src="rsq.gif"><img src="rsq.gif">
   <img src="rsq.gif"></div>
```

Как видите, трансформации подлежат три одинаковых элемента, в исходном состоянии выглядящие так, как показано в первом ряду. Во втором ряду приведен результат их поворота на 50 радиан (45°) к наблюдателю и добавления перспективы, назначаемой каждому из них по отдельности.

Третий ряд изображений получен в отсутствие у элементов индивидуальной перспективы и наследовании ее (`perspective: 800px;`) от родительского элемента `div`. Общие настройки перспективы обеспечивают единство трансформации всех трех элементов. Именно такой вид можно получить, закрепив три изображения на общей основе, которая вращается вокруг горизонтальной оси, проходящей через ее середину.

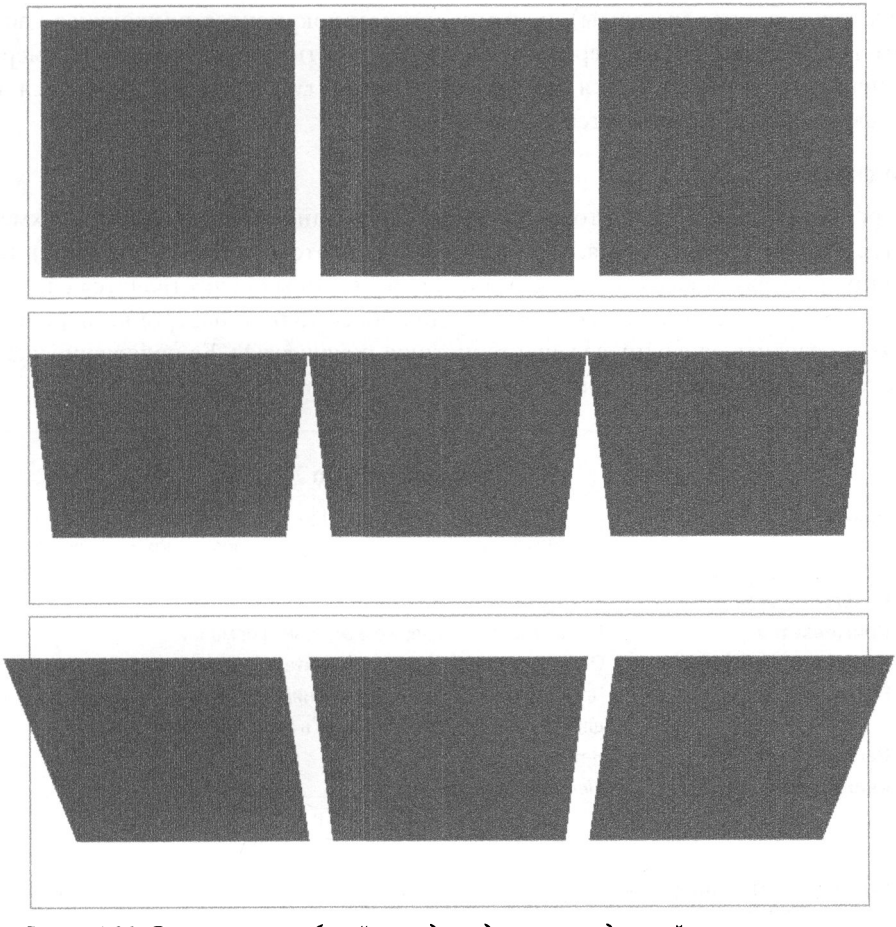


Рис. 16.28. Элементы с общей и индивидуально заданной перспективами

Получаемый выше эффект становится возможным благодаря применению к трансформируемым элементам свойства `transform-style` со значением `preserve-3d` (см. предыдущий раздел).

В этом проявляется самое значимое отличие свойства `perspective` от функции `perspective()`. Первое образует общее пространство трансформации для всех дочерних элементов, а вторая применяется исключительно к целевому элементу. Кроме того, в последовательности команд трансформации функция `perspective()` должна всегда указываться первой, определяя способ представления элемента в трехмерном пространстве документа. С другой стороны, свойство `perspective` применяется сразу ко всем дочерним элементам независимо от места объявления функций их трансформации.

В большинстве операций трансформации можно довольствоваться возможностями свойства `perspective`. Обычно трансформации подлежат контейнеры `div`

(и некоторые другие элементы), поэтому авторам документов приходится работать преимущественно с общей перспективой. В частности, в предыдущем примере эффект перспективы добавляется к элементу `<div id="two">`. Без него добиться должного эффекта не представляется возможным.

Точка схода

Перспектива добавляется только к элементам, трансформируемым в трехмерном пространстве (предполагается, что они представляются в трехмерном виде). В предыдущих разделах показано, что настройка перспективы осуществляется с помощью свойств `perspective` и `transform-style`. Для более точной настройки перспективы нужно определить еще одну ее характеристику: *точку схода*. Ее положение указывается с помощью стилевого свойства `perspective-origin`.

perspective-origin

Значение	[`left` | `center` | `right` | `top` | `bottom` | *<percentage>* | *<length>*] | [`left` | `center` | `right` | *<percentage>* | *<length>*] && [`top` | `center` | `bottom` | *<percentage>* | *<length>*]] *<length>*?
Начальное значение	`50% 50%`
Применяется	Все элементы, подверженные трансформации
Процентное значение	Относительно размера ограничительной рамки (см. описание)
Вычисляется	Процент, за исключением значений, выраженных в единицах длины, которые преобразуются в абсолютные значения
Наследуется	Нет
Анимируется	*<length>*, *<percentage>*

Сравнив свойства `perspective-origin` и `transform-origin`, можно обнаружить, что они обладают схожим синтаксисом, в том числе и необязательным аргументом, указывающим глубину расположения (координату Z) исходной точки. Тем не менее, несмотря на идентичность обрабатываемых значений, свойства воспроизводят совершенно разные эффекты. В то время как свойство `transform-origin` указывает точку, относительно которой выполняются преобразования, свойство `perspective-origin` устанавливает положение точки, в которой сходится перспектива изображения элемента.

Как и свойства трансформации, свойство `perspective-origin` проще описать на реальном примере, чем разбирать назначение каждого из ключевых слов его аргумента. Рассмотрим следующий пример, результат выполнения которого приведен на рис. 16.29.

```
#container {perspective: 850px; perspective-origin: 50% 0%;}
#ruler {height: 50px; background: #DED url(tick.gif) repeat-x;
   transform: rotateX(60deg);
   transform-origin: 50% 100%;}
```

```
<div id="container">
   <div id="ruler"></div>
</div>
```

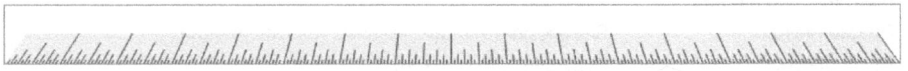


Рис. 16.29. Обычная линейка

Приведенный выше код обеспечивает повторение фонового изображения, представляющего фрагмент линейки с отметками, нанесенными вдоль горизонтальной оси, и последующий наклон содержащего их элемента `div` на 60° от наблюдателя. Все линии отметок указывают на точку схода, находящуюся по центру верхнего края контейнера элемента `div` (согласно значению `50% 0%` свойства `perspective-origin`).

А теперь посмотрим, как будет выглядеть такая линейка при изменении положения точки схода (рис. 16.30).

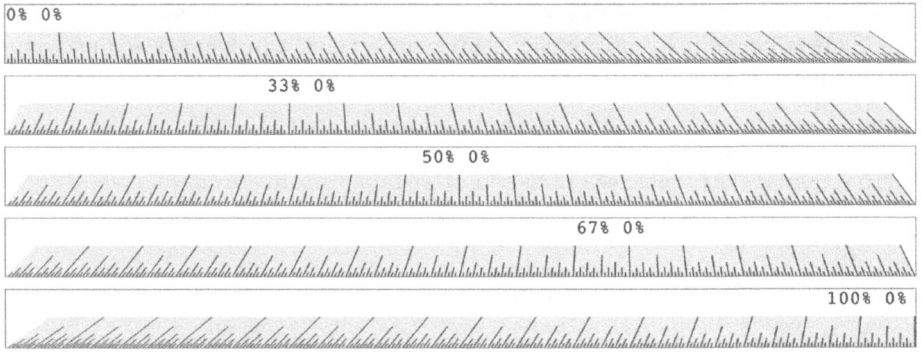


Рис. 16.30. Изменение вида линейки при разных положениях точки схода

Легко заметить, что даже незначительное смещение точки схода приводит к существенному изменению внешнего вида трансформируемого элемента.

Не забывайте, что рассмотренные выше эффекты получены при явном определении значения свойства `perspective`. Если оставить его в значении по умолчанию `none`, то свойство `perspective-origin` перестанет обрабатываться пользовательским агентом. Разумеется, такому поведению есть свое объяснение: невозможно изменить положение точки схода у элемента с отсутствующей перспективой!

Обратная сторона элемента

Вы могли никогда не задумываться над этим, но в прежние времена многим авторам просто-таки не терпелось развернуть элементы обратной стороной к наблюдателю. С появлением в CSS средств трехмерной трансформации такая операция стала обыденной, возведя вопрос визуализации обратной стороны элементов в разряд необычайно востребованных. Во многих случаях это выполняется умышленно, а способ визуализации обратной стороны элемента задается свойством `backface-visibility`.

backface-visibility	
Значение	`visible` | `hidden`
Начальное значение	`visible`
Применяется	Все элементы, подверженные трансформации
Вычисляется	Согласно определению
Наследуется	Нет
Анимируется	Нет

Это свойство, в отличие от остальных свойств и функций, рассмотренных в данной главе, имеет чрезвычайно простой синтаксис. Оно определяет всего два возможных варианта действия: визуализировать или не визуализировать обратную сторону элемента при развороте ее к наблюдателю.

Рассмотрим, каким образом будет выглядеть поворачиваемый на 180° элемент при объявлении для него свойства `backface-visibility` со значениями `visible` и `hidden`. Задача решается с помощью следующего кода, а результат его выполнения показан на рис. 16.31.

```
span {border: 1px solid red; display: inline-block;}
img {vertical-align: bottom;}
img.flip {transform: rotateX(180deg); display: inline-block;}
img#show {backface-visibility: visible;}
img#hide {backface-visibility: hidden;}

<span><img src="salmon.gif"></span>
<span><img src="salmon.gif" class="flip" id="show"></span>
<span><img src="salmon.gif" class="flip" id="hide"></span>
```

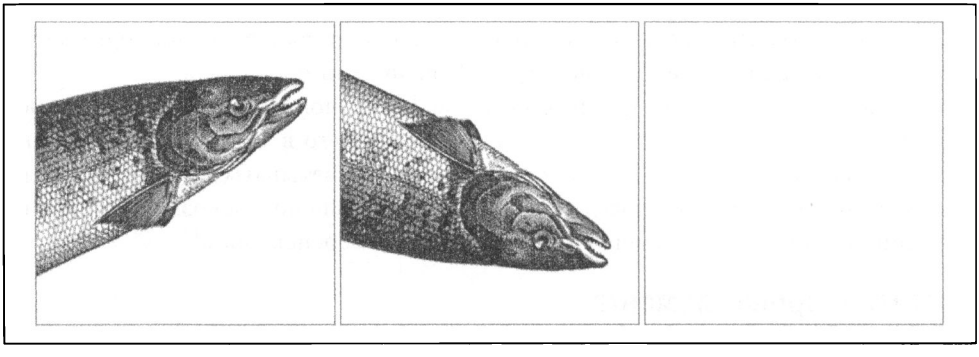


Рис. 16.31. Отображение и сокрытие обратной стороны элемента

На первом изображении показан элемент в исходном состоянии. Второе изображение представляет его же после поворота на 180° вокруг горизонтальной оси, что позволяет наблюдать его обратную сторону. Третье изображение отсутствует вследствие сокрытия обратной стороны элемента.

Свойство backface-visibility находит применение при решении самых разных задач. В частности, оно позволяет снабжать элемент пользовательского интерфейса сразу двумя изображениями, сменяющими друг друга при наведении и убирании с него указателя мыши. Например, можно выводить настройки поиска на обратной стороне поисковой панели или указывать дополнительные сведения на обороте фотографий. Последнюю задачу можно решить с помощью такого кода.

```
section {position: relative;}
img, div {position: absolute; top: 0; left: 0;
          backface-visibility: hidden;}
div {transform: rotateY(180deg);}
section:hover {transform: rotateY(180deg);
               transform-style: preserve-3d;}

<section>
   <img src="photo.jpg" alt="">
   <div class="info">(...описание...)</div>
</section>
```

Данный пример призван показать, что использование свойства backface-visibility в реальных проектах — далеко не такая простая задача, как кажется на первый взгляд. С его объявлением трудностей обычно не возникает, но если не включить в правило свойство transform-style со значением preserve-3d, то конечный результат может быть совершенно непредсказуемым. В данном случае свойство transform-style объявлено для элемента section.

В следующем примере стилевому форматированию подлежит тот же фрагмент документа, что и в предыдущем случае, но теперь обратная сторона элемента отображается несколько иным образом. Данный вариант более реалистичный, поскольку дополнительные сведения выглядят в нем так, будто бы действительно нанесены на обороте иллюстрации (рис. 16.32).

```
section {position: relative;}
img, div {position: absolute; top: 0; left: 0;}
div {transform: rotateY(180deg); backface-visibility: hidden;
   background: rgba(255,255,255,0.85);}
section:hover {transform: rotateY(180deg);
   transform-style: preserve-3d;}
```

В данном случае свойство backface-visibilty со значением hidden объявляется только для элемента div, а отменяется для элемента img, как это было в предыдущем примере. Таким образом, при повороте на 180° обратная сторона выводится только у изображения, но не у элемента div.

Резюме

Инструменты трансформации в двух- и трехмерном пространстве, включенные в CSS, открывают перед веб-дизайнерами совершенно новые возможности по визуальному оформлению документов. С их помощью можно не только изменять положение элементов в плоскости экрана, но и создавать полноценные трехмерные интерфейсы.

Команды трансформации относятся к отдельной категории инструментов, призванных существенно обновить стилевые возможности CSS по добавлению в документ специальных эффектов. При первом знакомстве принципы взаимодействия свойств трансформации могут показаться весьма необычными, но такое их поведение оказывается более чем оправданным в реальных проектах.

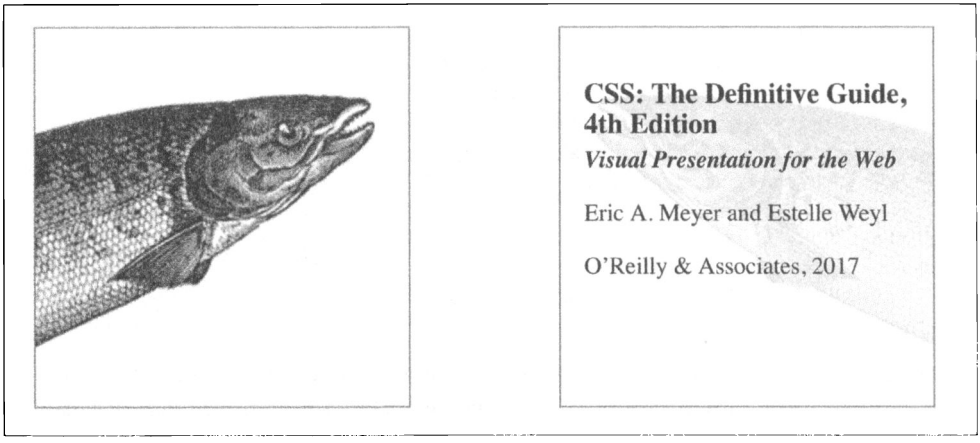


Рис. 16.32. Иллюстрация выводится на лицевой стороне элемента, а дополнительные сведения — на обратной

Переходы

В CSS операция перехода определяется как анимация состояния элемента вследствие временного изменения значений одного или нескольких свойств. Обычно переход выполняется в ответ на определенное действие со стороны пользователя, хотя в отдельных случаях осуществляется согласно инструкциям сценария, содержащим предписание изменить класс, идентификатор или некую другую характеристику элемента.

Как правило, изменение значения свойства, обусловливающего переход и возникающего вследствие некоего события, выполняется почти моментально. Новое значение заменяет старое в доли секунды, в течение которых происходит перерисовка элемента и обновление содержимого документа. В большинстве случаев переход происходит в мгновение ока — менее чем за 16 миллисекунд[1]. Несмотря на возможную задержку, такие переходы всегда выполняются в один этап. Например, переход от одного фонового цвета к другому при наведении указателя мыши на элемент сводится к резкой смене оттенка и полностью лишен градиентной составляющей.

Переходы в CSS

Стилевые инструменты позволяют определять способ и длительность изменения состояния, обусловливаемого переходом. При их правильном использовании переход получится предельно плавным и ненавязчивым для восприятия. Простой пример такого перехода приведен ниже.

```
button {
   color: magenta;
   transition: color 200ms ease-in 50ms;
}

button:hover {
   color: rebeccapurple;
   transition: color 200ms ease-out 50ms;
}
```

[1] Перерисовка фона элемента может занимать больше 16 миллисекунд, уходящих на обновление и повторную визуализацию страницы. Дополнительная задержка обусловливается не переходом, а низкой производительностью электронного устройства.

В этом примере свойство `color` кнопки при наведении на нее указателя мыши изменяется от значения `magenta` до `rebecca` не мгновенно, а предельно плавно. Переход длится 200 мс, из которых 50 мс отводится на начальную задержку. В данном контексте под переходом подразумевается изменение цвета кнопки, сколько бы времени оно ни занимало. Задача свойства `transition` заключается в предоставлении возможности плавного изменения характеристики в течение всего времени, отведенного на переход.

Спецификация CSS позволяет добавлять переходы в документы, отображаемые даже в старых браузерах, таких как IE9. Но и в отсутствие поддержки соответствующих стилевых свойств переходы все еще будут отображаться на странице, хотя и без плавного изменения значений соответствующих свойств. Плавные переходы возможны только для свойств, изменение состояния которых обеспечивается анимируемыми значениями. В случае изменения неанимируемых свойств переход получается моментальным.

Под анимируемым подразумевается свойство, изменение значения которого приводит к плавному изменению состояния элемента (подробнее об этом — в главе 18). Полный список анимируемых свойств приведен в приложении А.

Как бы там ни было, во многих случаях переход заключается исключительно в мгновенном изменении значения свойства. В частности, такой подход реализован в примере, приведенном в предыдущей главе, в котором гиперссылки выделяются иным цветом при наведении на них указателя мыши. Моментальное изменение состояния также свойственно подсказкам функции автозаполнения: они появляются и исчезают предельно быстро, мгновенно реагируя на изменения в тексте, набираемом в поле или области ввода формы. По вполне очевидным причинам плавность отображения подсказок не относится к достоинствам функции автозаполнения.

Тем не менее привлечение внимания пользователя к элементам документа может осуществляться и через плавное изменение их состояния, как, например, в карточной игре, отслеживание ходов в которой становится возможным только благодаря анимации переворота карт, выполняемого в течение около 200 мс (см. пример на сайте книги; файл *cardflip.html*).

Интерактивные примеры для этой главы можно посмотреть на сайте книги, адрес которого указан во введении.

Раскрывающееся меню служит еще одним примером плавного перехода. Мгновенно открываемое меню воспринимается крайне отталкивающе — лучше всего представляющий его переход выглядит при все той же длительности 200 мс. К плавно появляющимся на экране меню пользователи более благосклонны, а потому они намного предпочтительнее для применения в собственных проектах. На рис. 17.1 показано, как выглядит такое меню в средней временной точке перехода (см. пример на

сайте книги; файл *dd_menu.html*). Следует отметить, что такой способ вывода меню, описанный в следующей главе, применяется в CSS по умолчанию.

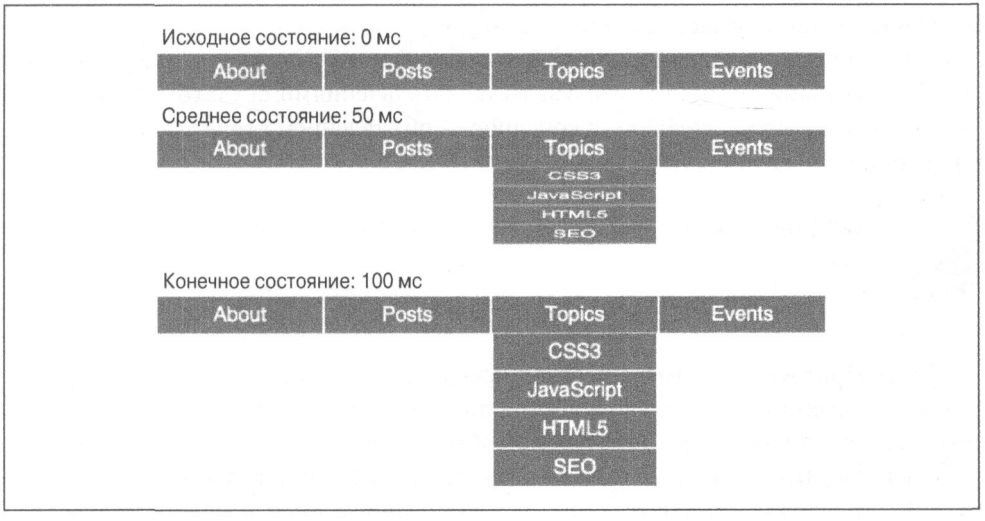


Рис. 17.1. Начальное, среднее и конечное состояния перехода

Свойства настройки переходов

В CSS настройка переходов требует применения свойств `transition-property`, `transition-duration`, `transition-timing-function` и `transition-delay`. Кроме того, их функции может выполнять единственное свойство общего назначения `transition`.

Для создания навигационной панели, показанной на рис. 17.1, потребуются все четыре свойства настройки переходов, отвечающих за установку начального и конечного состояний раскрывающихся меню, а также некоторые другие регулярные свойства. В частности, следующий код снабжает переходом меню, показанное на рис. 17.1.

```
nav li ul {
   transition-property: transform;
   transition-duration: 200ms;
   transition-timing-function: ease-in;
   transition-delay: 50ms;
   transform: scale(1, 0);
   transform-origin: top center;
}
nav li:hover ul {
   transform: scale(1, 1);
}
```

В этом примере событие перехода инициируется с помощью псевдокласса `:hover`, хотя эту задачу можно решить и другими способами. В общем случае изменение состояния может возникать в результате удаления атрибута класса, замены

псевдокласса `:invalid` на `:valid` или псевдокласса `:checked` на `:not(:checked)`. Наряду с этим для добавления строки в конец таблицы или нового элемента в конец раскрывающегося меню лучше всего применять стилевые свойства, селекторы которых включают псевдокласс `:nth-last-of-type`.

В сценарии, позволяющем получить навигационную панель, показанную на рис. 17.1, начальное состояние меню задается объявлениями `transform: scale(1, 0)` и `transform-origin: top center`, а конечное — объявлением `transform: scale(1, 1)` при исходном значении свойства `transform-origin`.

Детально о свойствах преобразования см. в главе 16.

В рассматриваемом примере переход осуществляется вследствие трансформации элемента вложенного неупорядоченного списка с помощью свойства `transform`, отвечающего за изменение его размера по событию `hover`. Возврат списка к исходному размеру осуществляется благодаря плавной трансформации из точки `transform: scale(1, 0)` в точку `transform: scale(1, 1)`, выполняемой в течение 200 мс с задержкой 50 мс. Скорость перехода устанавливается функцией плавности, с которой вам еще предстоит познакомиться.

Заметьте: наряду со свойствами перехода форматирование элемента устанавливается с помощью отдельных регулярных свойств. При этом плавному изменению значений подвергаются только свойства перехода, а остальное стилевое форматирование остается неизменным.

Обратите внимание на то, что переход настраивается для элементов `ul`, с которых уводится указатель мыши. При наведении указателя мыши на элемент к нему применяется не переход, а трансформация, чему есть вполне обоснованное объяснение: меню должно не только отображаться на экране при наведении на него указателя, но и закрываться при отведении его от элемента.

Проанализируем, какие действия будут выполняться при настройке перехода, осуществляемого при наведении на элемент указателя мыши.

```
nav li ul {
   transform: scale(1, 0);
   transform-origin: top center;
}
nav li:hover ul {
   transition-property: transform;
   transition-duration: 200ms;
   transition-timing-function: ease-in;
   transition-delay: 50ms;
   transform: scale(1, 1);
}
```

В последнем варианте кода свойства перехода (теперь мгновенного) приобретают исходные значения при *потере* элементом фокуса указателя мыши. Согласно таким

настройкам меню все так же открывается при наведении на него указателя мыши, а вот его закрытие при отведении указателя мыши выполняется мгновенно, а не плавно, как в предыдущем примере. А все потому, что при наведении указателя на элемент меню теряет все настройки перехода!

Заметьте, что в таком эффекте — плавное открытие и моментальное закрытие меню — нет ничего предосудительного, и он вполне имеет право на существование. Если же назначить переход для элемента, находящего в исходном состоянии, то плавный переход будут наблюдаться в обоих направлениях: при наведении и отведении указателя мыши от меню. При назначении перехода для события отведения указателя от элемента все его временные настройки инвертируются. Чтобы предотвратить обратный тайминг, необходимо поменять местами его начальное и конечное состояния.

Под исходным подразумевается состояние, которое принимает элемент при загрузке страницы. В нем элемент получает привычное стилевое форматирование, отличающееся от задаваемого при отнесении его к псевдоклассу `:hover`. В одном из вариантов такой элемент подлежит изменению при отнесении к псевдоклассу `:focus` (см. пример на сайте книги; файл *contenteditable.html*).

```
/* селектор, постоянно указывающий на элемент */
p[contenteditable] {
   background-color: rgba(0, 0, 0, 0);
}
/* селектор, указывающий на элемент только в некоторых ситуациях */
p[contenteditable]:focus {
   /* объявления, замещающие исходные */
   background-color: rgba(0, 0, 0, 0.1);
}
```

Согласно такому коду прозрачный фон назначается элементу на постоянной основе, заменяясь другим вариантом только при получении элементом фокуса. Значение свойства, определяющее форматирование элемента на постоянной основе, называется *исходным*, или *заданным по умолчанию*. В противоположность ему значения свойств перехода изменяются при переводе элемента из исходного состояния в измененное (состояние получения фокуса в предыдущем примере).

Исходное состояние может быть временным, например у элементов форм, отнесенных к псевдоклассу `:checked` или `:valid` либо классу, позволяющему включать и отключать элемент.

```
/* селектор, указывающий на элемент только в некоторых ситуациях */
input:valid {
   border-color: green;
}

/* селектор, указывающий на элемент только в некоторых ситуациях
   и срабатывающий только в отсутствие предыдущего селектора */
input:invalid {
   border-color: red;
}
```

```
/* селектор, указывающий на элемент только в некоторых ситуациях
   и срабатывающий независимо от правильности вводимых данных */
input:focus {
   /* альтернативные объявления */
   border-color: yellow;
}
```

В последнем примере элемент одномоментно может представлять селектор `:valid` или `:invalid`, но не оба сразу. При этом селектор `:focus` будет указывать на элемент (рис. 17.2) независимо от того, относится он к псевдоклассу `:valid` или `:invalid` (см. пример на сайте книги; файл *valid_invalid_focus.html*).

Рис. 17.2. Форматирование поля ввода при вводе правильных, неправильных данных и передаче ему фокуса (см. цветные иллюстрации на веб-сайте)

В исходном и измененном состояниях свойствам перехода можно присваивать произвольные значения, но они должны в точности воспроизводить нужный эффект при переходе элемента в каждое из них. В качестве примера рассмотрим код создания раскрывающегося меню, которое открывается в течение долгих 2 секунд, а закрывается всего за 200 миллисекунд.

```
nav li ul {
   transition-property: transform;
   transition-duration: 200ms;
   transition-timing-function: ease-in;
   transition-delay: 50ms;
   transform: scale(1, 0);
   transform-origin: top center;
}
nav li:hover ul {
   transition-property: transform;
   transition-duration: 2s;
   transition-timing-function: linear;
   transition-delay: 1s;
   transform: scale(1, 1);
}
```

С точки зрения наблюдателя такое поведение пользовательского интерфейса выглядит ужасно, зато прекрасно подходит для иллюстрации принципов объявления свойств перехода в обоих возможных состояниях (см. пример на сайте книги; файл *dd_menu_badUX.html*). Открытие меню при наведении на него указателя мыши

выполняется за целых 2 секунды. А вот на его закрытие при отведении указателя мыши уходят стандартные 0,2 с. Здесь измененное состояние для свойств перехода наступает при наведении указателя на элемент списка, поэтому именно для него (`li:hover ul`) определено объявление `transition-duration: 2s`. Переход к исходному, заданному по умолчанию состоянию (`nav li ul`) осуществляется при отведении указателя мыши от меню и длится всего 200 мс.

Обратите особое внимание на значения свойств перехода в исходном состоянии. При перемещении указателя мыши с (родительского) элемента панели навигации сворачивание дочернего раскрывающегося списка, длящееся оговоренные выше 200 мс, начинается только через 50 мс. Это достаточно честно по отношению к пользователям: если указатель убран с названия меню по ошибке, то при быстром возвращении его на место меню останется открытым.

Несмотря на возможность объявления отдельных свойств настройки переходов, их функции всегда можно возложить на единственное свойство общего назначения. Тем не менее сначала мы рассмотрим индивидуальные свойства и только после этого перейдем к изучению особенностей применения свойства общего назначения.

Определение действий перехода

Свойству `transition-property` передаются названия свойств перехода, участвующих в исполнении его эффекта. В процессе перехода изменяются значения только этих свойств, а все остальные свойства остаются в исходном состоянии.

transition-property

Значение	`none` | [`all` | *<property-name>*]#
Начальное значение	`all`
Применяется	Все элементы и псевдоэлементы `:before` и `:after`
Вычисляется	Согласно определению
Наследуется	Нет
Анимируется	Нет

Значения свойства `transition-property` отделяются друг от друга запятой. Ключевое слово `none` отменяет все свойства перехода, а значение `all` указывает применять в эффекте перехода все анимируемые свойства, заданные для элемента. Более того, ключевое слово `all` допускается включать в качестве отдельного значения в список названий свойств, разделенных запятыми.

При установке или передаче по умолчанию ключевого слова `all` эффект перехода будет заключаться в одновременном изменении значений всех анимированных свойств элемента. Предположим, что в результате перехода, возникающего вследствие наведения на элемент `div` указателя мыши, будут изменяться следующие его свойства.

```
div {
   color: #ff0000;
```

```
   border: 1px solid #00ff00;
   border-radius: 0;
   transform: scale(1) rotate(0deg);
   opacity: 1;
   box-shadow: 3px 3px rgba(0, 0, 0, 0.1);
   width: 50px;
  padding: 100px;
}
div:hover {
   color: #000000;
   border: 5px dashed #000000;
   border-radius: 50%;
   transform: scale(2) rotate(-10deg);
   opacity: 0.5;
   box-shadow: -3px -3px rgba(255, 0, 0, 0.5);
   width: 100px;
   padding: 20px;
}
```

При наведении указателя мыши на элемент `div` каждое из свойств, имеющих иное значение в исходном состоянии, изменяет его на значение, назначенное для состояния `hover`. Назначение свойства `transition-property` состоит в определении свойств, участвующих во временной анимации перехода. Обратите внимание на то, что после наведения указателя мыши изменяются исходные значения всех свойств элемента, а в свойстве `transition-property` указываются только свойства, подлежащие плавному изменению. Значения свойств, не поддерживающих анимацию, подобные `border-style`, изменяются мгновенно, а не плавно, как предписывают временные настройки перехода.

Если свойству `transition-property` передано одно лишь значение `all` или оно указывается последним в списке разделенных запятыми значений, то анимация всех свойств будет выполняться синхронно. Иными словами, все свойства, названия которых включены в список разделенных запятыми значений, будут обрабатываться одновременно.

Таким образом, следующие два фрагмента CSS-кода позволяют получить переход, представленный сразу всеми анимируемыми свойствами элемента.

```
div {
   color: #ff0000;
   border: 1px solid #00ff00;
   border-radius: 0;
   transform: scale(1) rotate(0deg);
   opacity: 1;
   box-shadow: 3px 3px rgba(0, 0, 0, 0.1);
   width: 50px;
   padding: 100px;
   transition-property: color, border, border-radius, transform,
      opacity, box-shadow, width, padding;
   transition-duration: 1s;
}
```

```
div {
   color: #ff0000;
   border: 1px solid #00ff00;
   border-radius: 0;
   transform: scale(1) rotate(0deg);
   opacity: 1;
   box-shadow: 3px 3px rgba(0, 0, 0, 0.1);
   width: 50px;
   padding: 100px;
   transition-property: all;
   transition-duration: 1s;
}
```

Оба варианта перехода заключаются в изменении значений всех свойств элемента, хотя в первом случае их всего восемь, ведь именно столько свойств объявлено в блоке правила (несомненно, их может быть больше — с учетом свойств, объявленных для этого же элемента в блоках других правил.) В данном случае все восемь правил объявляются в том же блоке, что и сам переход, хотя это не обязательно.

Во втором варианте свойства перехода определяются объявлением `transition-property: all`, указывающим изменять значения всех без исключения анимированных свойств в течение секунды, независимо от исходного места их объявления в CSS-коде. В результате переход будет выполняться для всех элементов, на которые указывает селектор правила, и всех, а не только объявленных в текущем блоке кода, анимируемых свойств.

Объявление анимируемых свойств в том же блоке, что и свойства `transition-property`, позволяет не только ограничить их количество в переходе, но и точнее настроить его анимацию. В частности, следующий способ объявления перехода обеспечивает настройку скорости, задержки и/или длительности анимации отдельно для каждого из свойств.

```
div {
   color: #ff0000;
   border: 1px solid #0f0;
   border-radius: 0;
   transform: scale(1) rotate(0deg);
   opacity: 1;
   box-shadow: 3px 3px rgba(0, 0, 0, 0.1);
   width: 50px;
   padding: 100px;
}

.foo {
   color: #00ff00;
   transition-property: color, border, border-radius, transform,
      opacity, box-shadow, width, padding;
   transition-duration: 1s;
}

<div class="foo">Hello</div>
```

Следовательно, для независимой настройки каждого из свойств перехода перечисляйте их названия в объявлении свойства `transition-property`, обязательно разделяя их запятыми. Если же большинство свойств перехода характеризуется одинаковыми параметрами анимации (скорость и/или длительность), а специальной настройки требуют только некоторые из них, то в объявлении свойства `transition-property` достаточно перечислить только их, а свойства с одинаковым поведением можно представить ключевым словом `all`.

```
div {
   color: #f00;
   border: 1px solid #00ff00;
   border-radius: 0;
   transform: scale(1) rotate(0deg);
   opacity: 1;
   box-shadow: 3px 3px rgba(0, 0, 0, 0.1);
   width: 50px;
   padding: 100px;
   transition-property: all, border-radius, opacity;
   transition-duration: 1s, 2s, 3s;
}
```

В данном случае значение `all` представляет все свойства, определенные в том же блоке, что и свойство `transition-property`, все наследуемые элементом свойства, а также все свойства, объявленные и наследуемые им в других блоках CSS-кода.

В предыдущем примере изменение значений всех свойств перехода осуществлялось в течение одного и того же промежутка времени, за исключением свойств `border-radius` и `opacity`, значения которых определяются отдельно от остальных. Благодаря включению их названий в объявление свойства `transition-property` наряду с ключевым словом `all` они автоматически получают такие же настройки анимации, что и остальные свойства перехода, хотя при необходимости их можно определить отдельно. В данном случае анимация большинства свойств выполняется в течение одной секунды, и только свойство `border-radius` изменяется в течение двух секунд, а свойство `opacity` — целых три секунды. (Свойство `transition-duration` детально рассмотрено в следующем разделе.)

Выработайте привычку всегда при объявлении свойства `transition-property` первым указывать значение `all`. Все свойства, названия которых указаны перед ним, будут включаться в общий список, а потому лишаться индивидуальных значений и получать такие же настройки, как и все остальные свойства.

Отмена перехода

Если переход не был осуществлен по умолчанию, а его свойства задействованы в других переходах текущего сценария, то к элементу нужно применить объявление `transition-property: none`, что позволяет сбросить настройки анимации свойств, предотвратив их воздействие на все последующие переходы.

Ключевое слово none может передаваться только свойству transition-property, но не свойствам, названия которых объявляются в виде списка значений, разделенных запятыми. Чтобы сбросить анимацию только для некоторых из свойств перехода, их названия нужно исключить из списка значений, указываемых после ключевого слова none: в объявлении свойства transition-property допускается указывать названия только подлежащих анимации свойств, но никак не исключаемых из нее.

В качестве альтернативного решения попробуйте установить для отдельных свойств нулевую (0s) длительность анимации перехода. В результате их значения будут изменяться мгновенно, как если бы переход для них не настраивался вовсе.

События перехода

В объектной модели документа событие transitionend возникает сразу же после завершения перехода — отдельно для каждого из свойств, анимируемых в течение любого временного промежутка *или* с любой задержкой. Такое событие генерируется как для свойств, представленных словом all, так и объявленных в свойстве transition-property отдельно. В некоторых случаях завершение перехода сопровождается возникновением для свойства сразу нескольких событий transitionend, как в следующем примере, в котором переход обусловливается плавным изменением свойств общего назначения.

```
div {
   color: #f00;
   border: 1px solid #00ff00;
   border-radius: 0;
   transform: scale(1) rotate(0deg);
   opacity: 1;
   box-shadow: 3px 3px rgba(0, 0, 0, 0.1);
   width: 50px;
   padding: 100px;
   transition-property: all, border-radius, opacity;
   transition-duration: 1s, 2s, 3s;
}
```

По завершении перехода должно возникать более восьми событий transitionend. Например, одно только свойство border-radius отвечает за возникновение четырех событий, по одному для каждого из зависимых свойств.

- border-bottom-left-radius
- border-bottom-right-radius
- border-top-right-radius
- border-top-left-radius

Подобным образом свойство общего назначения padding также представляет четыре отдельных свойства.

- padding-top
- padding-right

- padding-bottom
- padding-left

А вот свойство border отвечает за возникновение события конца перехода сразу у восьми свойств: четырех у свойства общего назначения border-width и еще четырех — у свойства border-color.

- border-left-width
- border-right-width
- border-top-width
- border-bottom-width
- border-top-color
- border-left-color
- border-right-color
- border-bottom-color

При этом событие transitionend не генерируется для свойства border-style, поскольку оно не относится к анимируемым свойствам.

Но как удостовериться, что свойство border-style не анимируется? Если не знать наверняка, то об этом можно только догадываться, например по отсутствию промежуточного стиля между сплошной (solid) и пунктирной (dashed) линиями. Чтобы удостовериться в этом, просмотрите полный список анимируемых свойств, приведенный в приложении А, или же внимательно изучите спецификации отдельных свойств указанного семейства.

В примере перехода, заключающегося в анимации 8 отдельно объявленных свойств, возникает 21 событие transitionend. Часть из таких свойств представлена свойствами общего назначения, имеющих разные значения в исходном и измененном состояниях. В сценарии объявления свойств перехода с помощью ключевого слова all в документе также будет сгенерировано 21 событие transitionend — по одному для каждого из индивидуальных свойств, анимация которых настраивается в явном виде. Остальные события окончания перехода генерируются предположительно для наследуемых свойств и свойств, объявленных в других блоках CSS-кода (см. пример на сайте книги; файл *transitionend.html*).

Для прослушивания событий transitionend может применяться такой код.

```
document.querySelector('div').addEventListener('transitionend',
   function (e) {
      console.log(e.propertyName);
});
```

Событие transitionend описывается тремя атрибутами.

1. propertyName. Содержит название свойства, завершающего переход.
2. pseudoElement. Представляет псевдоэлемент, через который осуществляется переход. Предваряется парой двоеточий или пустой строкой при объявлении перехода для регулярного узла DOM.

3. `elapsedTime`. Указывает время в секундах, отведенное на переход. Обычно совпадает со значением, присвоенным свойству `transition-duration`.

Событие `transitionend` возникает только в случае успешного перехода свойства к новому значению. Оно не генерируется в случае прерывания перехода, вызванного изменением значения свойства целевого элемента иным способом.

Заметьте, что по возвращении свойства перехода к исходному значению возникает еще одно событие `transitionend`. Оно будет генерироваться даже при незавершенности перехода в основном направлении.

Длительность перехода

Свойство `transition-duration` принимает в качестве значения список разделенных запятыми числовых величин, выраженных в секундах (`s`) или миллисекундах (`ms`), и обозначает время или длительность перехода элемента из одного состояния в другое.

transition-duration

Значение	*<time>#*
Начальное значение	`0s`
Применяется	Все элементы и псевдоэлементы `:before` и `:after`
Вычисляется	Согласно определению
Наследуется	Нет
Анимируется	Нет

Свойство `transition-duration` определяет длительность только того перехода, в объявление которого оно добавлено, и только для указанного направления. Рассмотрим следующий пример.

```
input:invalid {
   transition-duration: 1s;
   background-color: red;
}

input:valid {
   transition-duration: 0.2s;
   background-color: green;
}
```

При определении этому свойству сразу нескольких значений каждое из них будет устанавливать длительность перехода только в своем блоке объявлений. Следовательно, в предыдущем примере поле ввода данных приобретает красный фон (неправильное значение) в течение целой секунды, а переход к синему фону (правильное значение) будет длиться всего 0,2 секунды (см. пример на сайте книги; файл *transition_duration.html*).

Свойство transition-duration может получать только положительные числовые значения, устанавливаемые в единицах измерения времени: секундах (s) или миллисекундах (ms). Единицы измерения нужно указывать даже тогда, когда переход имеет нулевую длительность (0s). Такая настройка соответствует установкам по умолчанию и обусловливает мгновенный переход элемента из одного состояния в другое, который не сопровождается визуальной анимацией.

За исключением случаев передачи свойству transition-delay положительного значения отсутствие значения у свойства transition-duration равнозначно отмене события transitionend и сбросу для элемента блока объявлений transition-property. Переход считается действительным только тогда, когда длительность перехода определяется положительным числом, включая значение 0s, устанавливаемое по умолчанию или задаваемое в явном виде. Такой переход будет успешно завершен и сопровождаться возникновением события transitionend.

Отрицательные значения к свойству transition-duration не применяются, а их передача приводит к игнорированию всего объявления.

Каждому из свойств, объявленных в блоке transition-property (см. предыдущий раздел), можно задать свою длительность перехода или же установить общее значение для них всех. Общая длительность перехода назначается при передаче свойству transition-duration единственного значения.

```
div {
   color: #ff0000;
   ...
   transition-property: color, border, border-radius, transform,
                        opacity, box-shadow, width, padding;
   transition-duration: 200ms;
}
```

В свойстве transition-duration допускается указывать столько разделенных запятой значений, сколько объявляется в блоке transition-property. Для задания длительности перехода каждому из свойств, перечисленных в списке transition-property, их количество должно совпадать с количеством временных значений свойства transition-duration.

```
div {
   color: #ff0000;
   ...
   transition-property: color, border, border-radius, transform,
                        opacity, box-shadow, width, padding;
   transition-duration: 200ms, 180ms, 160ms, 140ms, 120ms, 100ms, 1s, 2s;
}
```

При передаче свойствам transition-property и transition-duration разного количества значений обработка перехода осуществляется пользовательским агентом согласно специальным правилам. В случае, когда временных значений больше, чем названий свойств, "лишние" числовые значения попросту игнорируются. Если же названий свойств больше, чем временных значений, то переходы некоторых из них будут иметь одинаковую длительность. В следующем примере переход свойств

color, border-radius, opacity и width длится 100 мс, а свойств border, transform, box-shadow и padding — 200 мс.

```
div {
   ...
   transition-property: color, border, border-radius, transform,
                        opacity, box-shadow, width, padding;
   transition-duration: 100ms, 200ms;
}
```

При передаче свойству transition-duration всего двух значений первое из них определяет длительность перехода для свойств, названия которых указаны в четных, а второе — в нечетных позициях списка свойства transition-property.

Длительность перехода подбирается исходя из пользовательских потребностей. При слишком медленном переходе страница будет отображаться очень долго, что вызывает раздражение и резко снижает интерес к ней. Слишком быстрые переходы не позволяют в полной мере ознакомиться с данными документа. Длительность переходов нужно подбирать так, чтобы они вызывали только положительные эмоции и не мешали просматривать содержимое страницы. Эффект должен быть заметен, но не слишком детально. В общем случае наиболее приемлемыми считаются переходы длительностью от 100 до 200 мс: они привлекают к себе внимание, но не настолько, чтобы отвлекать от содержимого элементов.

В данном примере раскрывающегося меню переход обоих свойств длится 200 мс.

```
nav li ul {
   transition-property: transform, opacity;
   transition-duration: 200ms;
   ...
}
```

Временная функция

В отдельных ситуациях переход должен выполняться с непостоянной скоростью: сначала медленно и быстрее в конце или, наоборот, быстрее вначале, но медленнее в конце. Кроме того, на определенных этапах он может приостанавливаться и даже выполняться скачкообразно. За изменение поведения перехода в течение всего времени его выполнения отвечает стилевое свойство transition-timing-function.

transition-timing-function

Значение	*<timing-function>#*
Начальное значение	ease
Применяется	Все элементы и псевдоэлементы :before и :after
Вычисляется	Согласно определению
Наследуется	Нет
Анимируется	Нет

В качестве значений это свойство принимает ключевые слова `ease`, `linear`, `ease-in`, `ease-out`, `ease-in-out`, `step-start`, `step-end`, `steps(n, start)`, `steps(n, end)` и `cubic-bezier(x1, y1, x2, y2)`, где *n* — количество шагов анимации. (Эти же ключевые слова передаются в качестве значений свойству `animation-timing-function`, детально рассмотренному в следующей главе.)

Значения, лишенные аргумента *n*, соответствуют функциям плавности, которые представлены кривыми Безье третьей степени, повсеместно применяемыми для построения гладких кривых. Подробно кубические уравнения Безье, применяемые в качестве допустимых значений свойства `transition-timing-function`, описаны в табл. 17.1.

Таблица 17.1. Соответствие значений свойства `transition-timing-function` уравнениям Безье третьей степени

Функция плавности	Описание	Кубическое преобразование Безье
`cubic-bezier()`	Общее уравнение кривой Безье	`cubic-bezier(x1, y1, x2, y2)`
`ease`	Медленный старт с последующим ускорением, замедлением и очень медленным завершением	`cubic-bezier(0.25, 0.1, 0.25, 1)`
`linear`	Одинаковая скорость анимации на протяжении всего перехода	`cubic-bezier(0, 0, 1, 1)`
`ease-in`	Медленный старт с последующим ускорением	`cubic-bezier(0.42, 0, 1, 1)`
`ease-out`	Быстрый старт с последующим замедлением	`cubic-bezier(0, 0, 0.58, 1)`
`ease-in-out`	Подобна `ease`; очень медленный старт с более сильным ускорением и замедленным завершением	`cubic-bezier(0.42, 0, 0.58, 1)`

Кривые Безье третьего порядка, применяемые для описания скорости перехода, представляются пятью функциями плавности, описанными в табл. 17.1 и изображенными на рис. 17.3. Каждая из них имеет четыре аргумента. Например, линейная кривая Безье представляется функцией вида `cubic-bezier(0, 0, 1, 1)`. В общем случае первый и третий аргументы функции Безье должны принимать значения из диапазона `0–1`.

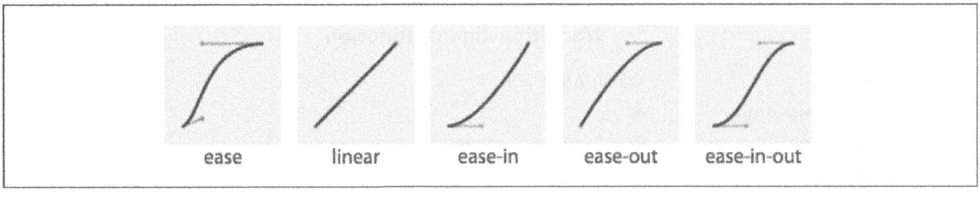


Рис. 17.3. Графическое представление третьего порядка функций Безье

Аргументы функции `cubic-bezier()` определяют координаты *x* и *y* *опорных точек* кривой Безье. Эти точки находятся на концах опорных линий, выходящих из

левого нижнего и правого верхнего углов области рисования кривой Безье. Таким образом, для получения кривой требуемой формы в функцию Безье нужно передать координаты двух углов и двух опорных точек.

Получить представление о том, каким образом координаты опорных точек определяют форму кривой Безье, можно по рис. 17.4.

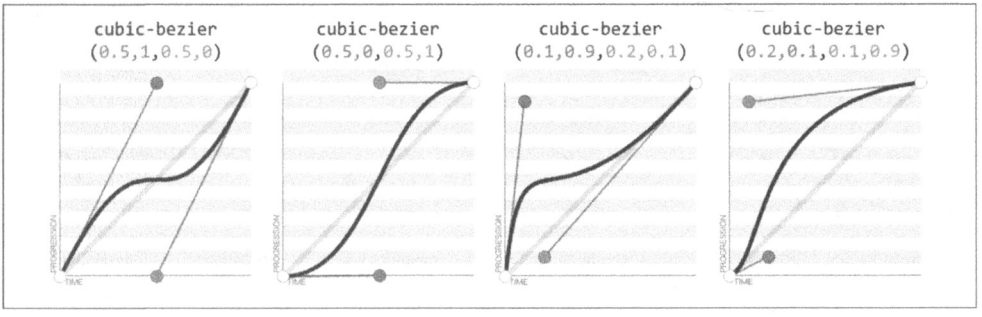


Рис. 17.4. Кривые Безье третьего порядка, описываемые функцией `cubic-bezier()` *(получены на сайте* `http://cubicbezier.com`*)*

Рассмотрим первый пример. Первые два аргумента функции, соответствующие координатам *x1* и *y1*, представлены значениями `0.5` и `1`. Первая опорная точка позиционируется посередине области рисования кривой (*x1* = `0.5`) по горизонтали и у ее верхнего края в вертикальном направлении (*y1* = `1`). Аналогичным образом вторая опорная точка устанавливается в координатах (`0.5`, `0`) — посередине нижнего края области рисования кривой. Запомните форму кривой при таком положении опорных точек.

Во втором примере позиционирование опорных точек выполняется в обратном порядке, что незамедлительно сказывается на форме кривой. Третий и четвертый примеры получены взаимным инвертированием координат опорных точек. Обратите внимание на полное отличие форм кривых в каждом из случаев.

Кривые Безье, передаваемые ключевыми словами, обладают строго заданным набором характеристик. Для точной настройки анимации переходов их скорость лучше всего задавать через функции Безье с произвольными значениями аргументов. Если вы занимались вычислением кривых Безье или работали в Illustrator или Freehand, то правильное позиционирование опорных точек будет выглядеть легко решаемой задачей. Всем остальным лучше воспользоваться специальным онлайн-калькулятором (`http://cubic-bezier.com`), позволяющим визуализировать кривые Безье третьего порядка, которые представлены как ключевыми словами, так и вычисляемыми значениями, получаемыми при произвольных аргументах расчетной функции.

На рис. 17.5 приведен полный список функций плавности (`http://easings.net`), представленных собственными именами и обеспечивающих точный контроль за ходом анимации.

Не забывайте, что в спецификации CSS большинство именованных функций плавности не представлено отдельными ключевыми словами, поэтому их нужно определять через общую функцию Безье с разными наборами аргументов (табл. 17.2).

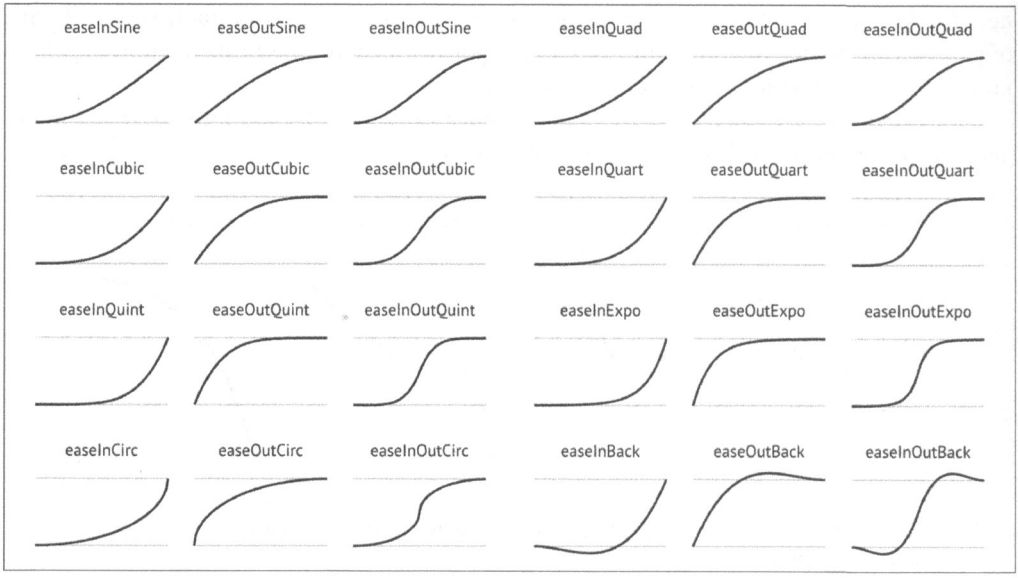


Рис. 17.5. Кривые Безье, представленные именованными функциями плавности

Таблица 17.2. Именованные функции плавности и их представление в CSS

Неофициальное название	Функция Безье третьего порядка
easeInSine	cubic-bezier(0.47, 0, 0.745, 0.715)
easeOutSine	cubic-bezier(0.39, 0.575, 0.565, 1)
easeInOutSine	cubic-bezier(0.445, 0.05, 0.55, 0.95)
easeInQuad	cubic-bezier(0.55, 0.085, 0.68, 0.53)
easeOutQuad	cubic-bezier(0.25, 0.46, 0.45, 0.94)
easeInOutQuad	cubic-bezier(0.455, 0.03, 0.515, 0.955)
easeInCubic	cubic-bezier(0.55, 0.055, 0.675, 0.19)
easeOutCubic	cubic-bezier(0.215, 0.61, 0.355, 1)
easeInOutCubic	cubic-bezier(0.645, 0.045, 0.355, 1)
easeInQuart	cubic-bezier(0.895, 0.03, 0.685, 0.22)
easeOutQuart	cubic-bezier(0.165, 0.84, 0.44, 1)
easeInOutQuart	cubic-bezier(0.77, 0, 0.175, 1)
easeInQuint	cubic-bezier(0.755, 0.05, 0.855, 0.06)
easeOutQuint	cubic-bezier(0.23, 1, 0.32, 1)
easeInOutQuint	cubic-bezier(0.86, 0, 0.07, 1)
easeInExpo	cubic-bezier(0.95, 0.05, 0.795, 0.035)
easeOutExpo	cubic-bezier(0.19, 1, 0.22, 1)
easeInOutExpo	cubic-bezier(1, 0, 0, 1)
easeInCirc	cubic-bezier(0.6, 0.04, 0.98, 0.335)
easeOutCirc	cubic-bezier(0.075, 0.82, 0.165, 1)
easeInOutCirc	cubic-bezier(0.785, 0.135, 0.15, 0.86)

Неофициальное название	Функция Безье третьего порядка
`easeInBack`	`cubic-bezier(0.6, -0.28, 0.735, 0.045)`
`easeOutBack`	`cubic-bezier(0.175, 0.885, 0.32, 1.275)`
`easeInOutBack`	`cubic-bezier(0.68, -0.55, 0.265, 1.55)`

Шаговый переход

В CSS переход может выполняться в шаговом режиме: его течение обеспечивается временными функциями, представляемыми следующими ключевыми словами (табл. 17.3).

Таблица 17.3. Временные функции для шаговых переходов

Временная функция	Определение
`step-start`	Переход осуществляется в первом кадре анимации. Соответствует функции `steps(1, start)`
`step-end`	Переход осуществляется в последнем кадре анимации. Соответствует функции `steps(1, end)`
`steps(`*n*`, start)`	Представляет переход набором *n* моментальных состояний. Переход к первому такому состоянию осуществляется в начале общего перехода, а последующие переходы выполняются через интервал *n*/100% от общей длительности перехода
`steps(`*n*`, end)`	Представляет переход набором *n* моментальных состояний. Переход к первому такому состоянию происходит через интервал *n*/100% от общей длительности перехода

Графики временных функций, обеспечивающих пошаговое течение анимации перехода, приведены на рис. 17.6. В отличие от функций плавности они имеют ступенчатую, а не гладкую форму.

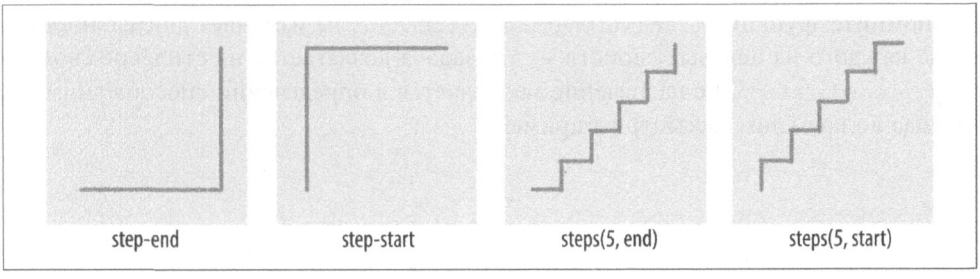


Рис. 17.6. Ступенчатые временные функции

Временные функции ступенчатой формы разделяют переход на большое количество микропереходов (шагов) равной длительности. Задача функции плавности заключается в определении количества шагов, которыми представляется переход, и направления его анимации. Последнее задается ключевым словом `start` или `end`.

Значение `start` указывает выполнять анимацию первого шага в самом начале перехода. Значение `end` обязывает пользовательский агент осуществлять анимацию в конце шага, а не в начале. Например, функция `steps(5, end)` формирует переход, шаги которого выполняются через интервалы `0%`, `20%`, `40%`, `60%` и `80%` от общей длительности, а функция `steps(5, start)` — такой же переход, но с шагами во временных точках `20%`, `40%`, `60%`, `80%` и `100%`.

Функция `step-start()` полностью заменяется функцией `steps(1, start)`. В такой анимации значения всех участвующих в ней свойств изменяются в начале перехода. В противоположность ей функция `step-end()`, полностью заменяемая функцией `steps(1, end)`, обеспечивает моментальный переход свойств к измененным значениям в конце перехода.

Детальное описание шаговых временных функций и ключевых слов `start` и `end` приведено в следующей главе.

В продолжение примера перехода, представленного блоком объявлений `transition-property`, отдельные временные функции можно назначить каждому из свойств или определить для них всех единственную временную функцию. В случае определения скорости перехода с помощью всего одной временной функции его код принимает такой вид.

```
div {
   transition-property: color, border-width, border-color,
      border-radius, transform, opacity, box-shadow, width, adding;
   transition-duration: 200ms;
   transition-timing-function: ease-in;
}
```

В качестве провокационного эксперимента рассмотрим, как будет выглядеть шаговый переход, в котором изменение значений свойств выполняется с разной скоростью.

Запомните: функция `transition-timing-function` не изменяет длительность перехода каждого из целевых свойств — эта задача возлагается на стилевое свойство `transition-duration`. Ее назначение заключается в определении способа анимации перехода во времени. Рассмотрим пример.

```
div {
   ...
   transition-property: color, border-width, border-color,
      border-radius, transform, opacity, box-shadow, width, padding;
   transition-duration: 200ms;
   transition-timing-function: ease, ease-in, ease-out, ease-in-out,
      linear, step-end, step-start, steps(5, start), steps(3, end);
}
```

Изменение значений всех девяти свойств перехода с собственными временными функциями начинается и заканчивается в одних и тех же временных точках.

Временны*е* функции определяют способ перехода в течение всего времени его выполнения, но не оказывают влияния на его длительность и время завершения. (Предыдущий пример достаточно провокационный и служит исключительно демонстрационным целям. Не стоит повторять его в собственных проектах.)

Самый простой способ ознакомиться с временны*ми* функциями заключается в тестировании их на реальных примерах, заключающихся в подборе наиболее оптимальных стратегий в рамках реальных проектов. Различия между временны*ми* функциями лучше всего проявляются при визуализации переходов, осуществляемых при больших значениях свойства `transition-duration` (см. пример на сайте книги; файл *transition_duration.html*). Просматривая переходы на реальной скорости, очень сложно найти различия в функциях плавности — попробуйте уменьшить скорость визуализации перехода до более приемлемого уровня. Не забудьте вернуть ее в исходное значение после завершения изучения анимации.

Отложенный переход

Свойство `transition-delay` позволяет выполнить переход с определенной временно*й* задержкой. В результате его применения переход будет начинаться не сразу, а через некоторое время после объявления.

transition-delay

Значение	*<time>*#
Начальное значение	`0s`
Применяется	Все элементы и псевдоэлементы `:before` и `:after`
Вычисляется	Согласно определению
Наследуется	Нет
Анимируется	Нет

Установка свойства `transition-delay` в значение `0s` (по умолчанию) предполагает немедленный запуск перехода — сразу после наступления для элемента соответствующего события. Подобными образом, например, обрабатывается эффект события `a:hover`.

Значения, отличные от `0s`, обозначаются переменной *<time>* и задают смещение временно*й* точки начала перехода. В результате применения такой настройки анимация свойств, приведенных в блоке `transition` или `transition-property`, будет начинаться только через указанный в ней интервал времени.

Как ни странно, но свойство `transition-delay` допускает обработку отрицательных значений. Эффект их применения описан в следующем разделе.

В примере, содержащем 8 (или 21) свойств, указываемых в блоке объявлений `transition-property`, для запуска анимации в общей временно*й* точке достаточно передать свойству `transition-delay` значение `0s` или вообще отказаться от его явного объявления. Альтернативный вариант предполагает немедленную анимацию только половины свойств перехода и выполнение остальной их части только спустя 200 мс, как показано в следующем CSS-коде.

```
div {
   transition-property: color, border, border-radius, transform,
                        opacity, box-shadow, width, padding;
   transition-duration: 200ms;
   transition-timing-function: linear;
   transition-delay: 0s, 200ms;
}
```

Применение объявления `transition-delay: 0s, 200ms` к последовательности свойств перехода, длящегося 200 мс, приводит к тому, что анимация будет немедленно стартовать только для половины из них, а именно свойств `color`, `border-radius`, `opacity` и `width`. Анимации остальных, "четных", свойств будет начинаться сразу же после завершения анимации "нечетных" свойств — исключительно благодаря совпадению значений свойств `transition-delay` и `transition-duration`.

Как и в свойствах `transition-duration` и `transition-timing-function`, если количество разделенных запятыми значений свойства `transition-delay` больше числа названий, перечисляемых в свойстве `transition-property`, то часть из них будет проигнорирована обработчиком браузера. При этом, если количество свойств перехода, указываемых в свойстве `transition-property`, превышает число временных значений, устанавливаемых свойством `transition-delay`, то последние будут размножены в достаточном количестве.

В предыдущем примере можно легко добиться эффекта старта анимации каждого следующего свойства только после выполнения анимации предыдущего свойства. Ниже приведен код стилевого форматирования такого перехода, обусловленного анимацией целых девяти свойств.

```
div {
   ...
   transition-property: color, border-width, border-color,
      border-radius, transform, opacity, box-shadow, width, padding;
   transition-duration: 200ms;
   transition-timing-function: linear;
   transition-delay: 0s, 0.2s, 0.4s, 0.6s, 0.8s, 1s, 1.2s, 1.4s, 1.6s;
}
```

Анимация всех свойств перехода длится 200 мс, на что указывает значение свойства `transition-duration`. На такую же величину (`0.2s`) прирастает каждое следующее значение свойства `transition-delay`. Следовательно, анимация каждого следующего свойства начинается после завершения анимации предыдущего свойства.

Чтобы обеспечить одновременный старт анимации всех свойств, участвующих в переходе, при разных длительностях и задержках, необходимо провести дополнительные вычисления и внести соответствующие изменения во временные значения.

```
div {
   ...
   transition-property: color, border-width, border-color,
      border-radius, transform, opacity, box-shadow, width, padding;
   transition-duration: 1.8s, 1.6s, 1.4s, 1.2s, 1s, 0.8s, 0.6s,
                        .4s, 0.2s;
   transition-timing-function: linear;
   transition-delay: 0s, 0.2s, 0.4s, 0.6s, 0.8s, 1s, 1.2s, 1.4s, 1.6s;
}
```

Согласно приведенному выше коду, анимация всех свойств перехода начинается в совершенно разное время и с разной задержкой и заканчивается во временной точке 1,8 с. Для правильной настройки такого перехода длительность анимации каждого следующего свойства уменьшается на значение свойства `transition-delay` относительно базового значения `1.8s`.

В общем случае анимация всех свойств перехода должна начинаться в общей временной точке. Для обеспечения такой возможности всем им нужно определить одинаковую задержку. В частности, анимация раскрывающегося меню, показанного на рис. 17.1, начинается с задержкой 50 мс, недостаточной для существенного замедления визуального эффекта. Ее назначение заключается в предотвращении случайного открытия меню при наведении на него указателя мыши в процессе прокрутки документа или перехода к другой части страницы.

Отрицательная задержка

Передача свойству `transition-delay` отрицательного значения, меньшего, чем значение свойства `transition-duration`, приводит к немедленному старту анимации перехода не с начальной, а с некой средней точки. Рассмотрим следующий пример (см. пример на сайте книги; файл *negative_delay.html*).

```
div {
   transform: translateX(0);
   transition-property: transform;
   transition-duration: 200ms;
   transition-delay: -150ms;
   transition-timing-function: linear;
}
div:hover {
   transform: translateX(200px);
}
```

Передача свойству `transition-delay` значения `-150ms` при длительности перехода 200 мс обязывает начать переход с третьей четверти своей длительности, т.е. завершиться всего за 50 мс. В сценарии линейного перехода элемент, на который наводится указатель мыши, сначала мгновенно перемещается вдоль оси X на расстояние `150px`, после чего плавно смещается вдоль нее на `50px` в течение оставшихся 50 мс.

Если же абсолютное значение свойства `transition-delay` больше или равно значению свойства `transition-duration`, то свойства перехода будут изменены мгновенно, как если бы свойство `transition` не применялось к элементу, а событие `transitionend` не возникало вовсе.

В обратном переходе — из состояния наведения на элемент указателя мыши к обычному состоянию — по умолчанию применяется такое же, как и в прямом переходе, значение свойства `transition-delay`. В предыдущем сценарии это означает, что возврат элемента в исходное положение будет анимироваться только последние 25% длительности, а начальные 75% перехода будут выполнены мгновенно. Таким образом, при отведении указателя мыши от элемента он сначала "перепрыгнет" в положение `50px` и только после этого будет плавно смещаться в исходное положение (`0px`) в течение последних 50 мс.

Свойство настройки перехода общего назначения

Свойство `transition` совмещает функции всех четырех описанных выше свойств: `transition-property`, `transition-duration`, `transition-timing-function` и `transition-delay`.

transition

Значение	*<single-transition>#*
Начальное значение	`all 0s ease 0s`
Применяется	Все элементы и псевдоэлементы `:before` и `:after`
Вычисляется	Согласно определению
Наследуется	Нет
Анимируется	Нет

<single-transition> = [[`none` | *<transition-property>*] || *<time>* || *<transition-timing-function>* || *<time>*]#

Это свойство принимает значение `none` или любое количество разделенных запятыми отдельных списков *настроек переходов*. К настройкам перехода относится название одного из свойств перехода (ключевое слово `all` обозначает сразу все заданные для элемента свойства перехода), его длительность, временная функция и задержка.

Если свойству `transition` не передать ни одного списка настроек перехода, то по умолчанию будет обрабатываться значение, представленное ключевым словом `all`. В отсутствие в объявлении значения, представляющего временную функцию, переход будет осуществляться согласно ключевому слову `ease`. При передаче свойству общего назначения только одного временного значения оно рассматривается как длительность перехода (при нулевой задержке, поскольку свойство `transition-delay` по умолчанию устанавливается в значение `0s`).

В пределах отдельного списка настроек перехода крайне важно придерживаться правильной последовательности определения значений длительности и задержки. Первой всегда указывается длительность перехода, и только после нее задается временная задержка. Следовательно, задержка всегда представляется в объявлении вторым временным значением.

Ниже приведено несколько равнозначных вариантов установки одного и того же эффекта перехода.

```
nav li ul {
   transition: transform 200ms ease-in 50ms,
      opacity 200ms ease-in 50ms;
   }

nav li ul {
   transition: all 200ms ease-in 50ms;
}
```

```
nav li ul {
   transition: 200ms ease-in 50ms;
}
```

В первом примере указаны обе временны*е* величины. Так как переход выполняется сразу для всех свойств, назначенных элементу, то их названия можно смело заменить ключевым словом `all`, как показано во втором примере. Но поскольку значение `all` устанавливается по умолчанию в любом случае, то в объявление достаточно включить только значения, определяющие длительность, временну*ю* функцию и задержку перехода. Если бы в переходе применялась функция плавности `ease`, а не `ease-in`, то ее также можно было бы не указывать.

В отсутствие значения длительности переход будет лишен анимации, и его течение останется незамеченным. Иными словами, единственное значение, требующее явного объявления в общем свойстве `transition`, — это значение свойства `transition-duration`.

Тем не менее настройка задержки в переходах с моментальным изменением значений свойств требует объявления его длительности (`0s`) в явном виде. Не забывайте о том, что первое временно*е* значение указывает длительность перехода, а задержка всегда определяется вторым временны*м* значением.

```
nav li ul {
   transition: 0s 200ms; ...
```

В этом переходе меню раскрывается моментально через 200 мс после наведения на него указателя мыши, что выглядит невероятно отталкивающе. Подобный эффект достоин разве что первоапрельского розыгрыша, особенно при замене селектора `li ul` на `*`.

В случае объявления с помощью свойства общего назначения `transition` сразу нескольких переходов, разделенных запятой и включающих настройки, представленные ключевым словом `none`, неправильным будет считаться все объявление, и вследствие этого оно будет полностью проигнорировано обработчиком.

```
div {
   transition-property: color, border-width, border-color,
      border-radius, transform, opacity, box-shadow, width, padding;
   transition-duration: 200ms, 180ms, 160ms, 140ms, 120ms, 100ms,
                        1s, 2s, 3s;
   transition-timing-function: ease, ease-in, ease-out, ease-in-out,
      linear, step-end, step-start, steps(5, start), steps(3, end);
   transition-delay: 0s, 0.2s, 0.4s, 0.6s, 0.8s, 1s, 1.2s, 1.4s, 1.6s;
}
div {
   transition:
      color 200ms,
      border-width 180ms ease-in 200ms,
      border-color 160ms ease-out 400ms,
      border-radius 140ms ease-in-out 600ms,
      transform 120ms linear 800ms,
```

```
    opacity 100ms step-end 1s,
    box-shadow 1s step-start 1.2s,
    width 2s steps(5, start) 1.4s,
    padding 3s steps(3, end) 1.6s;
}
```

Два предыдущих блока кода CSS функционально полностью идентичны: переход можно представить как список разделенных запятыми настроек отдельных переходов либо как список значений всех участвующих в нем свойств. Смешение форматов объявления перехода категорически недопустимо. В частности, объявление `transition: transform, opacity 200ms ease-in 50ms` не приведет к плавному изменению прозрачности элемента в течение 200 мс с задержкой 50 мс от момента наведения на него указателя мыши, а будет моментальным, поскольку событие `transitionend` при этом не возникает.

Независимый обратный переход

Во всех предыдущих примерах настраивались только прямые переходы, заключающиеся в изменении состояния элемента при наведении на него указателя мыши и возникновении события `hover`. При этом предполагается, что отведение указателя мыши от элемента приводит к возврату элемента в исходное состояние, что сопровождается применением такой же временной функции и задержки, отсчитанной в обратном направлении.

Одинаковое поведение элемента при наведении и отведении от него указателя справедливо при назначении перехода только для основного состояния. Как в прямом, так и в обратном переходе применяется одинаковый блок объявлений `transition`, а его селекторы оказываются справедливыми для обоих состояний. Для предотвращения дублирования настроек в обратном переходе его нужно объявить отдельно или же изменить отдельные свойства перехода для исходного (в противоположность измененному) состояния.

При создании перехода с несколькими состояниями значения свойств перехода необходимо указывать для каждого из них.

```
a {
   background: yellow;
   transition: 200ms background-color linear 0s;
}
a:hover {
   background-color: orange;
   /* задержка перед переходом в состояние :hover */
   transition-delay: 50ms;
}
```

В таком переходе гиперссылка получает оранжевый фон только спустя 50 мс после наведения на нее указателя мыши. А вот при отведении указателя гиперссылка отображается на желтом фоне мгновенно. Как прямой, так и обратный переходы длятся 200 мс, а заполнение элемента фоном осуществляется согласно линейной временной функции. Поскольку задержка длительностью 50 мс задана лишь для

состояния :hover, она будет задействована только при назначении элементу оранжевого фонового цвета (см. пример на сайте книги; файл *reverse.html*).

В примере с раскрывающимся меню наведение указателя мыши (:hover) будет приводить к плавному отображению меню в течение 200 мс с задержкой 50 мс. Такой переход выполняется из исходного состояния элемента (при последующем наведении указателя на целевой элемент). После отведения указателя мыши от меню выполняется обратный плавный переход к исходному состоянию, длящийся все те же 200 мс с такой же задержкой: 50 мс. В таком формате стилевого правила переход в исходное состояние выполняется автоматически и не подлежит настройке. Собственно в этом и заключается его преимущество: возврат в состояние по умолчанию не требует специального внимания. Но это совсем не означает, что обратный переход возможен только в таком варианте.

Для создания обратного перехода, выполняемого с настройками, отличающимися от заданных по умолчанию (это делать совсем не обязательно; следующий пример приведен исключительно в демонстрационных целях), его нужно объявлять отдельно.

```
nav ul ul {
   transform: scale(1, 0);
   opacity: 0;
   ...
   transition: all 4s steps(8, start) 1s;
}
nav li:hover ul {
   transform: scale(1, 1);
   opacity: 1;
   transition: all 200ms linear 50ms;
}
```

В настройках перехода всегда указывается конечное состояние — то, в котором будет пребывать элемент по завершении перехода. Следовательно, в них определяются не исходные, а измененные значения свойств перехода. В приведенном выше примере анимация прямого перехода — в состояние :hover — выполняется согласно линейной временной функции. Совсем иная ситуация наблюдается при отведении указателя мыши от элемента. В предыдущем примере это приводило к плавному закрытию меню в течение 200 мс с задержкой 50 мс. Если применить обратный переход из текущего примера, то отведение указателя мыши от родительского элемента li будет сопровождаться шаговой анимацией закрытия меню длительностью 4 с, выполняемой с секундной задержкой.

При объявлении единственного перехода его настройки определяются в блоке объявления исходного состояния, но применяются в любом измененном состоянии, при срабатывании любого события, будь то изменение класса элемента или наведение на него указателя мыши. Поскольку такой переход запускается по любому событию, то его настройку лучше всего выполнять в исходном блоке объявлений, в котором свойствам присваиваются наименее специфические значения (заданные по умолчанию). Тем не менее для точной настройки перехода в каждое из возможных состояний и направлений соответствующие объявления свойств перехода нужно добавить во все блоки кода, в которых регистрируется изменение класса или состояние элемента.

Старайтесь избегать одновременной настройки переходов для родительских и дочерних элементов. Свойства, одновременно участвующие в дочерних и родительских переходах, могут получать совершенно непредвиденные значения. Если переход дочернего элемента завершается раньше, чем переход родительского элемента, то дочерний элемент будет заимствовать значение свойства перехода (все еще изменяемое согласно своему переходу) у своего родителя. Такое поведение свойства может оказаться полной неожиданностью и совершенно не тем, что планировалось получить исходно.

Сброс прерванного перехода

Прерывание перехода до его завершения (например, при отведении указателя мыши от раскрывающегося меню до того, как оно полностью отобразится на экране) приводит к сбросу свойств перехода в исходные значения, которые они имели в начале перехода. Последующее применение к элементу обратного перехода с инвертированными временны*ми* настройками часто приводит к совершенно неприемлемым результатам. Чтобы предотвратить образование неправильного эффекта, спецификация предполагает выполнение обратного перехода по сокращенному сценарию.

В примере раскрывающегося меню свойство `transition-delay` получает значение `50ms` в исходном состоянии, а блок объявлений состояния `:hover` полностью лишен свойств перехода. Таким образом, обратный переход также будет выполняться с начальной задержкой 50 мс.

При успешном завершении прямого перехода в документе генерируется событие `transitionend`, сопровождаемое обязательным переносом настройки `transition-delay` в обратный переход.

Если же прямой переход прерывается, не завершившись (например, при отведении указателя мыши от меню до завершения его открытия), то событие `transitionend` не возникает, что не мешает всем браузерам, кроме Microsoft Edge, дублировать значение свойства `transition-delay` в блок объявления обратного перехода (табл. 17.4). При этом значение свойства `transition-duration` дублируется в блок объявления обратного перехода далеко не всеми браузерами. В этом смысле требования спецификации к упрощению обратного перехода выполняются только браузерами Microsoft Edge и Firefox.

Таблица 17.4. Упрощенный сценарий обратного перехода

Браузер	Обратная задержка	Длительность перехода, мс	Полная длительность перехода, мс
Chrome	Да	200	200
Chrome	Да	200	250
Safari	Да	200	200
Firefox	Да	38	38
Opera	Да	200	250
Edge	Нет	38	38

Представим, что пользователь отводит указатель мыши от меню спустя 75 мс после начала перехода. В подобном случае меню закрывается, так и не отобразившись на экране в полный размер. Перед тем как полностью скрыть меню, браузер выжидает положенные 50 мс, как в начале анимации открытия.

Начальная задержка играет очень важную роль во взаимодействии элемента меню с пользователем. Она предотвращает сворачивание меню вследствие случайного отведения от него указателя. Согласно данным, приведенным в табл. 17.2, такая задержка не назначается только браузером Microsoft Edge.

Несмотря на то что отведение указателя мыши от меню выполняется через 75 мс от начала прямого перехода, многие браузеры осуществляют обратный переход только после полного завершения прямого перехода (спустя 200 мс, как и предполагает значение свойства `transition-duration`). И только некоторые пользовательские агенты — Microsoft Edge и Firefox — в полной мере выполняют требования спецификации CSS по упрощению перехода в обратном направлении и сбросу свойств перехода к исходным значениям. В результате обратный переход длится в них ровно столько же, сколько и прямой переход, несмотря на совершенно иные настройки стилевых правил.

В браузерах Microsoft Edge и Firefox обратный шаговый переход будет длиться столько же времени, сколько уходит на выполнение округленного до меньшего значения количества шагов прерванного прямого перехода. Например, если полный прямой переход выполняется за 10 шагов и длится 10 секунд, то его прерывание через 3,25 секунды (за три четверти секунды до начала четвертого шага) приведет к тому, что обратный переход будет выполнен за 3 секунды. В следующем примере показано, что за три секунды элемент `div` успевает расшириться до 130 пикселей, а затем — после отведения указателя мыши — сузиться до исходных 100 пикселей.

```
div {
   width: 100px;
   transition: width 10s steps(10, start);
}
div:hover {
   width: 200px;
}1
```

Учтите, что анимация обратного перехода выполняется за то же время, что и прерванная анимация прямого перехода, но за исходное, а не округленное в меньшую сторону количество шагов прерванного прямого перехода. В данном примере обратный переход будет длиться всего 3 секунды, но выполняться за 10 шагов. В результате каждый шаг обратной анимации будет длиться намного меньше, чем в прямом переходе: 300 мс. При этом за каждый шаг обратного перехода элемент будет сужаться на 3 пикселя — в противоположность 10 пикселям, как это было в прямом переходе.

Такое поведение обратных переходов приводит к нарушению анимации движущихся объектов, например создаваемой с помощью свойства `background-position` (см. пример на сайте книги; файл *sprite.html*). Вполне возможно, что в будущих версиях спецификации требования к обратным переходам изменятся. Например, им будет разрешено выполняться за такое же количество шагов, как и прерванные прямые

переходы. Но уже сейчас некоторые браузеры обрабатывают обратные переходы, подчиняясь собственным установкам. Согласно им, обратный переход из приведенного выше примера будет длиться 10 секунд, занимать 10 шагов и выполняться со скоростью сокращенного перехода: 3 пикселя за шаг.

Браузеры, не поддерживающие предложенный спецификацией сценарий упрощения обратного перехода, выполняют его за указанные в стилевом правиле 10 секунд и за 10 шагов, а не 3 шага. В таком случае обратный переход выполняется в полном объеме, независимо от степени выполнения прямого перехода, его длительности и временной функции. В примере навигационной панели обратный переход, выполняемый согласно полному сценарию, будет длиться 200 мс и выполняться независимо от степени расширения элемента за время прямого перехода.

В браузерах, поддерживающих сокращенный сценарий выполнения обратных переходов, конечный результат сильно зависит от временной функции. Если анимация перехода выполняется согласно линейной временной функции, то длительность перехода будет одинаковой в обоих направлениях. При шаговой анимации длительность обратного перехода будет равняться времени, затраченному на выполнение всех полных (завершенных) шагов прямого перехода. В случае анимации, осуществляемой согласно функции Безье третьего порядка, длительность обратного перехода будет пропорциональна времени, затраченному на выполнение прямого перехода до его прерывания. Заметьте, что отрицательные значения свойства `transition-delay` уменьшаются пропорционально изменению длительности обратного перехода относительно значения, заданного в стилевом правиле.

При прерывании прямого перехода событие `transitionend` не генерируется ни одним из браузеров. Тем не менее во всех них оно возникает в момент завершения обратного перехода и возвращения элемента в исходное состояние. Значение `elapsedTime` обратного перехода будет зависеть от того, насколько реальная длительность обратного перехода отличается от заявленной в блоке объявлений свойств перехода величины (200 мс) или отношения времени закрытия меню к длительности незавершенного прямого перехода.

Для точной настройки параметров обратного перехода определите их в явном виде в соответствующих блоках объявлений свойств перехода для каждого из состояний элемента. Результат будет проявляться только в браузерах, не поддерживающих сокращенный сценарий обработки обратных переходов.

Анимируемые свойства и значения

Перед добавлением в документ переходов и анимации необходимо разобраться с тем, какие свойства подлежат ей, а какие — нет. В анимации и эффектах переходов могут участвовать только анимируемые стилевые свойства. Но как распознать анимируемые свойства среди всех остальных?

Список анимируемых свойств приведен в приложении А. Помните, что технология CSS постоянно совершенствуется и список время от времени дополняется новыми свойствами.

Один из критериев, по которому можно определить анимируемость свойства, — это *интерполируемость* его значений. Под интерполяцией понимается возможность получения промежуточных значений по имеющемуся набору известных величин. Таким образом, можно смело утверждать, что вычисляемые значения анимируемых свойств всегда интерполируемые. Разумеется, интерполяции не подлежат значения, представленные ключевыми словами. А вот числовые значения, выраженные в любых допустимых единицах измерения, можно получать интерполяцией уже имеющихся данных. В самом простом случае интерполяция подразумевает вычисление промежуточного значения свойства между двумя уже известными значениями.

В частности, значения свойства `display` (такие, как `block` и `inline-block`) представляются нечисловым типом данных, поэтому промежуточное значение для них вычислить невозможно. Наряду с этим значения `rotate(10deg)` и `rotate(20deg)` свойства `transform` вполне допускают существование промежуточного значения — `rotate(15deg)`, а потому относятся к анимируемым значениям.

Теперь рассмотрим одно из свойств общего назначения, например `border`, выполняющее функции свойств `border-style`, `border-width` и `border-color` (в свою очередь выступающих свойствами общего назначения для свойств настройки каждой из четырех сторон элемента). Свойство `border-style` не имеет промежуточных значений, а свойство `border-width` может представляться вычисляемыми величинами, промежуточные значения для которых вполне допустимы. При этом его значения, представленные ключевыми словами `medium`, `thick` и `thin`, имеют числовые эквиваленты и также подлежат интерполяции. Каждому из указанных ключевых слов соответствует вычисляемое числовое значение, выражаемое в допустимых единицах измерения длины.

В числовом виде можно представить любые значения свойства `border-color`: именованные цвета всегда можно выразить через шестнадцатеричные числовые значения. Следовательно, цвета подлежат анимации, как и любые другие вычисляемые величины. В частности, переход из состояния `border: red solid 3px` в состояние `border: blue dashed 10px` будет сопровождаться анимацией значений толщины и цвета границы, а вот изменение свойства `border-style` со значения `solid` в `dashed` выполняется мгновенно (в начале перехода, но с учетом объявленной задержки).

Как легко удостовериться в приложении А, анимации подлежат любые свойства, представляемые числовыми значениями. Не анимируются только значения, представляемые ключевыми словами, которые невозможно выразить числовыми величинами. Наряду с этим анимации также подлежат значения, представляемые функциями с числовыми аргументами. Исключение составляет разве что свойство `visibility`. Несмотря на то что его значения `visible` и `hidden` рассматриваются как интерполируемые, у них нет промежуточного значения. При вычислении значения свойства `visibility` оно представляется либо исходным значением, либо ключевым словом `visible`. В конце перехода свойство `visibility` всегда изменяется с `visible` на `hidden`, а изменение значения с `hidden` на `visible` всегда осуществляется в начале перехода.

Значение `auto` всегда рассматривается как не подлежащее анимации — старайтесь не применять его в блоке объявления свойств перехода. Согласно спецификации,

оно не является анимируемым ни при каких обстоятельствах, хотя отдельные браузеры представляют некоторые его вычисляемые вариации (например, `height: auto`) числовой величиной `0px`. Наряду с `auto` к не подлежащим анимации значениям относятся `height`, `width`, `top`, `bottom`, `left`, `right` и `margin`.

При настройке анимации переходов всегда можно воспользоваться альтернативными свойствами. Например, переход из состояния `height: 0` в `height: auto` во многих ситуациях можно заменить переходом из `max-height: 0` в `max-height: 100vh`. Значение `auto` будет анимироваться в свойствах `min-height` и `min-width`, так как объявление `min-height: auto` вычисляется пользовательским агентом как `0`.

Интерполяция значений

Интерполяция выражается в вычислении промежуточного значения в наборе двух или более известных значений. Анимация и эффект перехода требуют обязательной интерполяции значений соответствующих свойств.

Результатом интерполяции числовых значений будет действительное число с плавающей точкой, а целочисленные значения интерполируются до целого числа, получаемого округлением действительного числа в большую или меньшую сторону.

В CSS процентные значения и величины, выраженные в единицах измерения длины, перед интерполяцией представляются действительными числами. Таким образом, анимация или переход таких значений требуют использования функции `calc()` для пересчета их в действительные числа, которые, собственно, и участвуют в интерполяции.

Цветовые значения моделей HSLA или RGB, а также именованные цвета, например `aliceblue`, исходно преобразуются в RGBa-значения и только затем интерполируются в пространстве RGBa.

Анимация начертаний шрифтов, обозначаемых ключевыми словами наподобие `bold`, сводится к изменению представляющих их числовых множителей, выполняемому с шагом 100 единиц. Вполне возможно, что в следующих версиях спецификации начертания шрифтов будут представляться целочисленными значениями, а не множителями, кратными 100.

При анимации свойств, имеющих составные значения, каждый из компонентов интерполируется отдельно. Например, значение свойства `text-shadow` задает четыре параметра: цвет, смещение по горизонтали, смещение по вертикали и размытие. Первый из них интерполируется как цветовое значение, а остальные три — как числовые значения, выраженные в единицах длины. Тени контейнеров снабжаются двумя дополнительными параметрами: `inset` (внутрь) и `spread` (растянуть). Параметр `spread` вычисляется как числовое значение, выражаемое в единицах длины, а ключевое слово `inset` в числовом виде не представляется — переход возможет только от одного стиля отбрасываемой внутрь тени к другому или же между стилями тени, отбрасываемой наружу, но никак не между возможными промежуточными значениями.

Анимация и переход градиентов выполняется подобно многокомпонентным значениям. Они возможны только для цветовых переходов одного типа, например от одного линейного градиента к другому или между радиальными градиентами с

одинаковым количеством маркеров. При этом цвета маркеров интерполируются как цветовые значения, а их положение представляется процентными или числовыми значениями, выраженными в единицах длины.

Интерполяция повторяющихся значений

При интерполяции свойств разных типов каждое из них обрабатывается отдельно от остальных по правилам, принятым для его типа данных, — по крайней мере, до тех пор пока списки свойств включают одинаковое количество элементов или повторяемых элементов, а пары значений свойств подлежат интерполяции.

```
.img {
   background-image:
      url(1.gif), url(2.gif), url(3.gif), url(4.gif),
      url(5.gif), url(6.gif), url(7.gif), url(8.gif),
      url(9.gif), url(10.gif), url(11.gif), url(12.gif);
   background-size: 10px 10px, 20px 20px, 30px 30px, 40px 40px;
   transition: background-size 1s ease-in 0s;
}
.img:hover {
   background-size: 25px 25px, 50px 50px, 75px 75px, 100px 100px;
}
```

В приведенном выше коде эффекты переходов каждого из четырех свойств `background-size` обусловливаются всеми необходимыми парами значений, выраженных в пикселях, поэтому переход, например, третьего из таких свойств будет осуществляться плавно. Следовательно, наведение указателя мыши на элемент будет приводить к увеличению высоты и ширины фоновых изображений 1, 6 и 10 с `10px` до `25px`. Подобным образом изображения 3, 7 и 11 будут увеличиваться в горизонтальном и вертикальном размерах с `30px` до `75px`.

Легко заметить, что для покрытия всего списка фоновых изображений значения свойства `background-size` дублируются три раза. В полном виде переход этого свойства можно объявить с помощью такого CSS-кода.

```
.img {
   ...
   background-size: 10px 10px, 20px 20px, 30px 30px, 40px 40px,
      10px 10px, 20px 20px, 30px 30px, 40px 40px, 10px 10px,
      20px 20px, 30px 30px, 40px 40px;
   ...
}
.img:hover {
   background-size: 25px 25px, 50px 50px, 75px 75px, 100px 100px,
      25px 25px, 50px 50px, 75px 75px, 100px 100px, 25px 25px,
      50px 50px, 75px 75px, 100px 100px;
}
```

При недостаче разделенных запятыми пар значений свойства `background-sizes` для покрытия списка фоновых изображений они дублируются в нужном количестве даже в случае несовпадения со списком значений свойства `background-sizes`, приведенных для состояния `:hover`.

```
.img:hover {
   background-size: 33px 33px, 66px 66px, 99px 99px;
}
```

Если исходное состояние свойства `background-size` для 12 фоновых изображений задается тремя, а состояние `:hover` — четырьмя парами значений, то первый список значений будет продублирован четыре, а второй — три раза для получения 12 пар значений, каждая из которых соответствует отдельному значению свойства `background-image`.

```
.img {
   ...
   background-size: 10px 10px, 20px 20px, 30px 30px, 40px 40px,
      10px 10px, 20px 20px, 30px 30px, 40px 40px, 10px 10px,
      20px 20px, 30px 30px, 40px 40px;
   ...
}
.img:hover {
   background-size: 33px 33px, 66px 66px, 99px 99px, 33px 33px,
      66px 66px, 99px 99px, 33px 33px, 66px 66px, 99px 99px,
      33px 33px, 66px 66px, 99px 99px;
}
```

Если пары значений, объявляемые для свойства перехода, не подлежат интерполяции (например, в случае перехода свойства `background-size` от значения по умолчанию `contain` в значение `cover`), то, согласно спецификации, анимация отменяется сразу для них всех. Тем не менее некоторые браузеры игнорируют это требование и отменяют переход только для не подлежащих интерполяции пар значений, оставляя анимацию действительной для всех остальных величин.

Анимация некоторых свойств выполняется от или до прогнозируемых браузером значений, используемых вместо неявно заданных величин. Например, при обработке анимации тени любые неявно заданные значения свойств любого из двух состояний могут заменяться браузером на предполагаемое значение `boxshadow: transparent 0 0 0` или `box-shadow: inset transparent 0 0 0`.

Событие `transitionend` возникает только при обработке интерполируемых значений.

Как вы знаете, по способу анимации свойство `visibility` заметно отличается от остальных свойств. В частности, переход или анимация от (до) значения `visible` выполняется в результате одношаговой интерполяции до (от) противоположного значения. Эффект такой интерполяции становится заметен в любом переходе и анимации, в которых временная функция возвращает значения в диапазоне от 0 до 1. При этом анимация свойства от значения `hidden` до `visible` выполняется в начале перехода, а от значения `visible` до `hidden` — в конце. Момент перехода также зависит от значения, возвращаемого временной функцией.

Не переживайте, если случайно включили в объявление перехода свойство, не подлежащее анимации. Браузер не отменит весь переход, а всего лишь откажется интерполировать значение свойства, которое невозможно анимировать. При этом объявление такого свойства остается действительным, а само оно по мере возможности

обрабатывается пользовательским агентом. В результате ему передаются только действительные, а не интерполированные значения. Их наличие в списке анимируемых значений гарантирует корректность обработки всех остальных свойств перехода[2].

В переходах участвуют только те свойства, которые в обозначенный момент времени не задействованы в анимации. Добавление анимации не избавляет элемент от переходов — до тех пор пока в них не применяются такие же свойства. Детально анимация элементов рассматривается в следующей главе.

Переход как эффект оформления

Переходы прекрасно воспроизводятся пользовательскими агентами. Их поддерживают все известные браузеры, включая Safari, Chrome, Opera, Firefox, Edge и Internet Explorer (начиная с IE10).

Отсутствие поддержки переходов, призванных улучшить пользовательский интерфейс, не является причиной отказа от их добавления в документы. Но даже в отсутствие инструментов обработки переходов браузеры все еще могут их воспроизвести, сведя задаваемую стилевым правилом анимацию к моментальному изменению значений свойств.

Реальность такова, что пользователи могут лишиться интересного (или порядком раздражающего) эффекта, но не содержимого элемента, к которому применяется переход.

Переходы призваны существенным образом улучшить оформление документов. Не стоит отказываться от них в угоду устаревшим браузерам семейства IE — по крайней мере, до тех пор пока эффект переходов в них можно воссоздать с помощью JavaScript, а для обработки переходов в Android 4.1 достаточно снабдить стилевые свойства вендорными префиксами.

Печать переходов

При выводе веб-страницы на печать применяется стилевое оформление документа, принятое для бумажных носителей. Если же атрибуту `media` присвоено одно только значение `screen`, то элемент будет выводиться на бумагу без соответствующего стилевого форматирования.

Чаще всего атрибут `media` в документе не определяется, что равнозначно объявлению `media="all"` (настройка по умолчанию). Более того, в зависимости от браузера при печати документа возможны следующие варианты обработки в нем переходов: полное игнорирование интерполируемых значений и печать со значениями свойств переходов, свойственными текущему состоянию элементов.

[2] Вскоре такой подход к обработке переходов может измениться. Рабочая группа CSS рассматривает возможность анимации всех без исключения значений свойств — даже тех, которые на данный момент не подлежат интерполяции. В последнем случае изменение свойства перехода предполагается осуществлять не в одной из крайних, а в средней точке временной функции.

Разумеется, отобразить анимацию перехода на бумаге не представляется возможным. Однако некоторые браузеры, например Chrome, представляют элемент с переходом в виде, в котором он пребывает в момент вызова функции `print`, но это правило распространяется только для свойств, подлежащих печати. Например, при выводе на бумагу элемент с настроенным переходом свойства `background-color` не будет содержать фоновой заливки ни исходным, ни конечным цветом, поскольку это свойство не подлежит печати. Наряду с этим при печати на цветном принтере в документе будет отображаться переход, отображающий изменения в цвете текста.

Во всех остальных браузерах, подобных Firefox, значение свойства перехода, выводимого на печать, зависит от способа его инициализации в документе. Если переход возникает вследствие наведения на элемент указателя мыши, то элемент представляется в исходном состоянии, поскольку взаимодействие пользователя с диалоговым окном настройки печати документа возможно только в отсутствие указателя мыши над элементами документа. При возникновении перехода в результате изменения класса элемент выводится в измененном состоянии — даже в случае прерывания перехода и невозможности достижения его элементом. В таком случае печать осуществляется вразрез с настройками перехода.

Учитывая тот факт, что форматирование выводимых на печать документов может выполняться не только с помощью регулярных стилей, но и стилей, объявленных в команде `@media`, каждое из них браузеру приходится обрабатывать отдельно. В стилях для печати обычно учитываются только статические значения свойств, а любые переходы игнорируются. Это еще раз подтверждает тот факт, что печать не позволяет представить переход динамически, а только через одно из моментальных состояний.

Анимация

Стилевые переходы, описанные в предыдущем разделе, допускают создание в документе только простой анимации. В переходах свойства элемента изменяются от значений, заданных в одном правиле, до значений, объявленных в совершенно ином правиле. Таким образом, переход заключается в изменении состояний элемента, а не непрерывном изменении его самого в течение определенного промежутка времени, как предполагает анимация. В переходе начальное и конечное состояния элемента устанавливаются через значения соответствующих свойств, поведение которых не поддается точной настройке.

Анимация подобна переходам прежде всего возможностью изменения значений свойств, но имеет перед ними неоспоримое преимущество в способности предельно точно контролировать эти изменения. Анимация по ключевым кадрам позволяет отслеживать все действия, имеющие место в процессе ее выполнения. Если переход осуществляется после возникновения некоего события, то анимация воспроизводится в результате изменения значения свойств в ключевых кадрах.

Настраивая анимацию в CSS, вам придется иметь дело не только со значениями свойств, определяемых для начального и конечного состояний элемента. В действительности значения свойств, указываемые при создании анимации, далеко не всегда вовлечены в ее процесс. Например, в переходе от белого к черному цвету можно получить только серые промежуточные тона. В свою очередь анимация предполагает окрашивание этого же элемента в любой другой цвет, отличный не только от серого, но даже от белого или черного.

Если цветовой переход заключается в последовательном назначении элементу оттенков серого, то в анимации элемент может, например, сначала получить желтый цвет, а затем сменить его на оранжевый. Альтернативное решение может заключаться в анимации черного цвета в белый с абсолютно произвольными промежуточными оттенками. В этой главе вы узнаете о том, что такое анимация по ключевым кадрам и какое значение она имеет в стилевом форматировании документов.

Интерактивные примеры для этой главы можно посмотреть на сайте книги, адрес которого указан во введении.

Определение ключевых кадров

Воспроизведение (или ход) анимации настраивается через именованные ключевые кадры, которые определяют тип изменений, выполняемых над элементом. Ключевые кадры создаются с помощью команды `@keyframes` и впоследствии применяются к целевому элементу или псевдоэлементам, определяя точный способ его анимации.

В команде `@keyframes` указывается название или идентификатор анимации и содержится один или несколько блоков объявления ключевых кадров. Блок объявления ключевого кадра состоит из одного или нескольких селекторов и блока объявления свойств. Таким образом, правило `@keyframes` определяет действия, выполняемые в каждой итерации анимации. Количество итераций анимации устанавливается через значение свойства `animation-iteration-count`, описанного в разделе "Повторение анимации".

В блоке каждого ключевого кадра объявляется один или несколько *селекторов ключевых кадров*, обозначающих точки, в которых задаются контрольные значения свойств, изменяемых в указанном шаге анимации. Положение точек определяется либо процентными значениями, либо ключевыми словами `from` и `to`. В общем виде команда `@keyframes` имеет следующий код.

```
@keyframes animation_identifier {
   keyframe_selector {
      property: value;
      property: value;
   }
   keyframe_selector {
      property: value;
      property: value;
   }
}
```

Ниже приведено несколько примеров правила `@keyframes`.

```
@keyframes fadeout {
   from {
      opacity: 1;
   }
   to {
      opacity: 0;
   }
}

@keyframes color-pop {
   0% {
      color: black;
      background-color: white;
   }
   33% { /* треть длительности анимации */
      color: gray;
```

```
      background-color: yellow;
   }
   100% {
      color: white;
      background-color: orange;
   }
}
```

В первом наборе ключевых кадров настраивается анимация, заключающаяся в переходе элемента из полностью непрозрачного (`opacity: 1`) в абсолютно прозрачное (`opacity: 0`) состояние. Второй набор ключевых кадров предполагает анимацию фона элемента от белого до желтого, а затем оранжевого цвета при одновременном изменении цвета переднего плана с черного на серый.

Заметьте, что в ключевых кадрах не устанавливается абсолютная длительность анимации: для решения этой задачи в CSS предусмотрено отдельное свойство. Вместо этого в них указываются значения свойств, подлежащих анимации на каждом из ее шагов, и определяется длительность каждого шага относительно общей длительности анимации. Исходя из такого определения, селекторы ключевых кадров представляются процентными значениями или ключевыми словами `from` и `to`. Попытка определения селекторов в виде абсолютных числовых значений (например, как, `1.5s`) неизбежно приведет к неправильной визуализации эффекта анимации.

Настройка анимации

Итак, для создания анимации необходимо воспользоваться командой `@keyframes`, указав в ней имя анимации и обозначив в фигурных скобках действительные ключевые кадры. На данном этапе эта операция напоминает процесс объявления в таблицах стилей медиа-запросов (подробнее об этом — в главе 20).

Внутри фигурных кавычек указываются селекторы ключевых кадров и блоки объявления свойств, подлежащих анимации. После объявления в правиле `@keyframes` ключевые кадры применяются к целевому элементу посредством свойства `animation-name`, детально рассматриваемого в разделе "Именованная анимация".

В общем случае команда создания ключевых кадров представляется таким кодом.

```
@keyframes ИмяАнимации {
   ...
}
```

Имя анимации

В качестве имени анимации можно использовать идентификатор или строковое значение. Чаще всего это именно идентификатор, хотя и спецификация, и пользовательские агенты допускают представление анимации текстовыми значениями, заключенными в кавычки.

Идентификаторы, в отличие от строковых имен, не требуют заключения в кавычки, но подчиняются строгому набору правил. В именах идентификаторов допускается использовать цифры, символы английского алфавита (обоих регистров),

переноса, подчеркивания и любые другие символы, указанные в кодовой странице ISO 10646 после U+00A0. Кодовая страница ISO 10646 включает универсальный набор символов, в том числе все символы Unicode, представляемые регулярным выражением `[-_a-zA-Z0-9\u00A0-\u10FFFF]` (см. пример на сайте книги; файл *name.html*). Идентификатор не может начинаться с цифры и двойного дефиса. Одинарный дефис допускается использовать в начале идентификатора только при вводе после него символа, отличного от цифры (за исключением случаев вставки перед цифрой или дефисом обратной косой черты)

Добавляемые в идентификатор управляющие символы обязательно предваряются обратной косой чертой. Например, чтобы получить имя `Q&A!`, необходимо представить его кодом `Q\&A\!`. При этом идентификаторы `âœž` (это не опечатка) и ✎ представляются в исходном виде. Обратной косой чертой нужно обязательно предварять только символы, вводимые с клавиатуры и не относящиеся к буквам и цифрам, например `!`, `@`, `#`, `$` и т.п.

В качестве имени анимации лучше не использовать ключевые слова, упоминаемые в этой главе. В частности, в дальнейших примерах идентификаторы ключевых кадров содержат названия `none`, `paused`, `running`, `infinite`, `backwards` и `forwards`. Такой способ именования вполне допускается спецификацией CSS, но при использовании в свойстве общего назначения `animation` может привести к нарушению анимации (см. пример на сайте книги; файл *badnames.html*). Именно поэтому я настоятельно рекомендую не применять в качестве имени анимации ни одно из зарезервированных спецификацией ключевых слов.

Селекторы ключевых кадров

Селекторы ключевых кадров определяют временные точки, в которых начинается и заканчивается каждый шаг анимации. В блоке объявления селектора указываются значения, которые принимают свойства в заданных точках анимации. Значения, задаваемые свойствам в начале анимации, указываются в селекторе `0%`. Соответственно, чтобы присвоить свойствам значения, принимаемые в конце анимации, их нужно объявить в селекторе `100%`. Значения, указанные в селекторе `33%`, свойства получают по истечении трети длительности анимации. Селекторами обозначаются любые изменения, происходящие в ходе анимации.

В общем виде селекторы ключевых кадров представляются ключевыми словами `from` и `to` или как несколько разделенных запятыми процентных значений. При этом ключевому слову `from` соответствует значение `0%`, а ключевому слову `to` — значение `100%`. Процентные значения определяют временные точки относительно общей длительности анимации, в которых свойства получают значения, указанные в селекторах ключевых кадров. Таким образом, под ключевом кадром подразумевается блок объявления свойств отдельного селектора. Процентные значения в обязательном порядке снабжаются единицами измерений. В частности, значение `0` не является действительным селектором ключевых кадров.

```
@keyframes W {
   from {
```

```
      left: 0;
      top: 0;
   }
   25%, 75% {
      top: 100%;
   }
   50% {
      top: 50%;
   }
   to {
      left: 100%;
      top: 0;
       }
}
```

Применение такой анимации, получившей название `W`, к динамически позиционируемому элементу приводит к перемещению его вдоль W-образного контура. В ней определено сразу пять ключевых кадров: во временных точках `0%`, `25%`, `50%`, `75%` и `100%`. Точка `0%` обозначается ключевым словом `from`, а точка `100%` — ключевым словом `to` (см. пример на сайте книги; файл *02_W.html*).

Анимируемые свойства имеют одинаковые значения в точках `25%` и `75%`, что делает возможным их объявление в общем селекторе, название которого представляется списком разделенных запятыми имен селекторов отдельных ключевых кадров. Подобным образом объявляются селекторы элементов в регулярных стилевых правилах. Имена селекторов не обязательно вводить в одну строку. При необходимости каждое из них можно указывать в отдельной строке, как показано в следующем примере, взятом из предыдущего фрагмента кода.

```
25%,
75% {
   top: 100%;
}
```

Имена селекторов могут перечисляться в произвольном порядке. В приведенном выше примере ключевые кадры `25%` и `75%` объявляются перед ключевым кадром с селектором `50%`. Тем не менее, только поддерживая естественный порядок объявления ключевых кадров в команде `@keyframes`, можно получить предельно удобочитаемый код, максимально упрощенный для дальнейшего редактирования. Как видно в приведенном выше коде на примере селектора `75%`, это требование не обязательное, и при необходимости им можно пренебречь. Ничто не запрещает начать объявление ключевых кадров с последнего селектора, заканчивать объявление первым селекторам или указывать селекторы в произвольном порядке по мере обновления анимации.

Исключение начального и конечного селекторов

Если анимация настраивается в отсутствие объявления ключевого кадра, представленного селектором `0%` или `from`, то он будет автоматически интерполирован пользовательским агентом на основе значений, присвоенных анимируемым

свойствам в остальных ключевых кадрах, и предположения о том, что в точке 0% элемент находится в полностью статическом состоянии — за исключением случаев участия его свойств в другой анимации (об этом мы поговорим в разделе "Именованная анимации"). Подобным образом в отсутствие селектора 100% или to задаются значения свойств в последней точке анимации.

Предположим, в анимации участвует свойство background-color, изменяемое следующим образом.

```
@keyframes change_bgcolor {
   45% { background-color: green; }
   55% { background-color: blue; }
}
```

Также будем считать, что исходно — в статическом состоянии — это свойство имеет значение red, что при объявлении в коде анимации в явном виде можно представить так, как показано ниже (см. пример на сайте книги; файл *no0or100.html*).

```
@keyframes change_bgcolor {
   0% { background-color: red; }
   45% { background-color: green; }
   55% { background-color: blue; }
   100% { background-color: red; }
}
```

Обратите внимание на то, что в исходном коде объявление background-color: red не относится к анимации по ключевым кадрам. Если по умолчанию фон элемента представляется желтым цветом, то он будет автоматически устанавливаться в селекторах 0% и 100%. Остальная часть настроек анимации будет сохраняться в исходном виде. Таким образом, желтый цвет будет сменяться зеленым, плавно переходящим в синий, а затем — обратно в желтый.

```
@keyframes change_bgcolor {
   0%, 100% { background-color: yellow; }
   45% { background-color: green; }
   55% { background-color: blue; }
}
```

После объявления именованная анимация change_bgcolor может назначаться самым разным элементам, но результат ее применения будет напрямую зависеть от начального значения свойства background-color.

Несмотря на использование в предыдущих примерах только целочисленных процентных значений, пользовательские агенты прекрасно справляются с обработкой действительных процентных значений, таких, например, как 33.33%. Отрицательные процентные значения, как и значения, превышающие 100%, а также имеющие иной тип данных или представленные ключевыми словами, отличными от from и to, нельзя использовать в качестве селекторов.

Повторение свойств в ключевых кадрах

Самые первые инструменты настройки анимации появились в вендорном исполнении, поэтому снабжались префиксом `-webkit-`. К их особенности относилась невозможность многократного объявления одного и того же ключевого кадра. Любая такая попытка сводилась к обновлению ключевого кадра и назначению его свойствам только последних значений — все предыдущие объявления свойств для этого же селектора попросту игнорировались. Современное представление анимации допускает каскадное объявление одних и тех же ключевых кадров, содержащих объявления одних и тех же свойств. В предыдущем примере перемещения элемента вдоль W-образного контура ключевой кадр с селектором `to` или `100%` может объявляться дважды, но при этом в нем будет обновлено только значение свойства `left`.

```
@keyframes W {
   from, to {
      top: 0;
      left: 0;
   }
   25%, 75% {
      top: 100%;
   }
   50% {
      top: 50%;
   }
   to {
      left: 100%;
   }
}
```

Обратите внимание на то, что исходно ключевой кадр `to` объявляется вместе с ключевым кадром `from`. В нем устанавливаются значения сразу двух свойств: `top` и `left`. Но только последнее из них заменяется значением, указанным в последнем блоке объявления анимации.

Анимируемые свойства

Давайте немного отвлечемся от основной темы главы и вспомним, что далеко не все свойства подлежат анимации. Из всех объявленных в ключевых кадрах свойств браузером обрабатываются только анимируемые свойства, а все остальные — игнорируются. (Такое же поведение присуще браузерам по отношению к любым другим свойствам, о существовании которых им ничего не известно.)

Полный список анимируемых свойств приведен в приложении А. В дальнейших примерах я постараюсь явным образом указывать свойства, не подлежащие анимации.

Свойство `animation-timing-function`, детально описанное в разделе "Временная функция анимации", хотя и не подлежит анимации, остается действительным и успешно обрабатывается браузером. При добавлении в блок объявления ключевого кадра это свойство определяет временную функцию, согласно которой выполняется анимация свойств от текущего к следующему ключевому кадру.

Анимация свойств, не имеющих вычисляемого промежуточного значения для двух объявленных в ключевых кадрах значений, приводит к непредсказуемым эффектам. Ее результат невозможно спрогнозировать заранее, а в отдельных случаях свойства будут получать полностью статические значения. Например, согласно спецификации CSS высота элемента (свойство `height`) не подлежит анимации от значения `auto` до значения `300px`, поскольку между ними невозможно вычислить промежуточное значение. Тем не менее такая анимация возможна во многих браузерах, хотя и с заметно отличающимся результатом. В частности, Firefox откажется воспроизводить такую анимацию вовсе, Safari выполнит ее так, как если бы ключевое слово `auto` представлялось значением `0`, а Opera и Chrome осуществят простой переход из исходного состояния в конечное через некое среднее значение, которое в зависимости от временной функции (значения свойства `animation-timing-function`) не обязательно представляется селектором `50%`. Иными словами, каждое из свойств, не имеющих промежуточных значений между объявленными в анимации ключевыми кадрами, обрабатывается браузерами по-разному, что не позволяет предсказать ее результат хотя бы частично.

Наиболее предсказуемой анимация становится при определении в ней как начального (`0%`), так и конечного (`100%`) значения анимируемых свойств (см. пример на сайте книги; файл *nomidpoint.html*).

Например, включив в анимацию объявление `border-radius: 50%`, в нее также нужно добавить объявление `border-radius: 0`, поскольку браузеру сложно определить промежуточное значение для пары величин `50%` и `none` (по умолчанию свойство `border-radius` получает значение `none`, а не `0`). Различие становится очевидным при изучении следующих двух примеров.

```
@keyframes round {
   100% {
      border-radius: 50%;
   }
}
@keyframes square_to_round {
   0% {
      border-radius: 0%;
   }
   100% {
      border-radius: 50%;
   }
}
```

Команда `@keyframes round` обеспечивает анимацию свойства `border-radius` от исходного значения до значения `50%` в течение всего времени ее выполнения. При этом команда `@keyframes square_to_round` определяет такую же анимацию этого же свойства, но только от значения `0%` до `50%`. Обе анимации приводят к одинаковому результату только при применении к элементу, заключенному в рамку с прямыми углами. Если же рамка целевого элемента исходно имеет закругленные углы, то во втором правиле перед началом анимации они трансформируются в прямые углы. Такое поведение может отличаться от того, что ожидалось исходно. Для его

предотвращения попробуйте исключить из анимации ключевой кадр `from` и/или `to`, позволив устанавливать начальное и/или конечное состояние анимируемого свойства через ранее заданные значения, а не объявляемые в команде `@keyframes`.

В этом смысле анимация `round` оказывается предпочтительнее анимации `square_to_round`, поскольку начинается с трансформации элемента, имеющего рамку со скругленными углами заданного, а не нулевого радиуса (см. пример на сайте книги; файл *round2.html*).

Анимация между соседними ключевыми кадрами возможна, когда хотя бы в одном из них анимируемое свойство имеет значение, позволяющее вычислить промежуточное значение по отношению к любому другому значению этого же свойства в другом ключевом кадре.

Обрабатываемые свойства, не подлежащие анимации

Существуют всего два свойства настройки анимации, не позволяющих вычислить промежуточные значения: `visibility` и `animation-timing-function`.

Свойство `visibility` считается анимируемым, даже несмотря на то, что принимает всего два значения (`hidden` и `visible`) и лишено каких-либо вычисляемых промежуточных значений. Анимация от значения `hidden` до `visible` выполняется моментально в том ключевом кадре, в котором она объявлена.

С другой стороны, свойство `animation-timing-function` вообще не относится к анимируемым. При заключении в блок объявления ключевого кадра его значение задает временную функцию, согласно которой будет выполняться анимация до следующего ключевого кадра. В блоке объявлений следующего ключевого кадра этому свойству может присваиваться совершенно другое значение. При этом промежуточные значения свойства `animation-timing-function` для соседних ключевых кадров не вычисляются (подробно об этом — в разделе "Временная функция анимации").

Сценарии настройки анимации

Поиск, добавление и удаление ключевых кадров из анимации можно выполнять из специально созданных пользовательских интерфейсов. В частности, изменить содержимое блоков объявлений ключевых кадров, начинающихся с команды `@keyframes`, можно с помощью методов `appendRule(`*n*`)` и `deleteRule(`*n*`)`, где *n* — полный селектор ключевого кадра. Для возвращения содержимого ключевого кадра применяется метод `findRule(`*n*`)`.

```
@keyframes W {
   from, to {
      top: 0;
      left: 0;
   }
   25%, 75% {
      top: 100%;
   }
   50% {
      top: 50%;
   }
```

```
    to {
        left: 100%;
    }
}
```

Методы `appendRule()`, `deleteRule()` и `findRule()` имеют всего один аргумент: полный селектор ключевого кадра. В примере с анимацией элемента вдоль W-образной траектории аргумент `25%, 75%` будет представлять ключевые кадры, обозначаемые селектором `25%/75%`.

```
// Получение селектора и содержащего блока ключевого кадра
var aRule = myAnimation.findRule('25%, 75%').cssText;

// Удаление ключевого кадра с селектором 50%
myAnimation.deleteRule('50%');

// Добавление ключевого кадра с селектором 53% в конец анимации
myAnimation.appendRule('53% {top: 50%;}');
```

Инструкция `myAnimation.findRule('25%, 75%').cssText;`, в которой `myAnimation` представляет название анимации по ключевым кадрам, возвращает код объявления ключевых кадров, представленных селектором `25%, 75%`. Если в качестве аргумента указать всего одно значение, то не будет найден ни один из ключевых кадров. В примере с анимацией элемента вдоль W-образной кривой будет возвращен результат `25%, 75% { top: 100%; }`.

Подобным образом инструкция `myAnimation.deleteRule('50%')` удалит из объявления анимации блок *последнего* ключевого кадра, представленный селектором `50%`. Таким образом, при включении в анимацию сразу нескольких блоков объявлений для ключевого кадра `50%` будет удален только последний из них. Наряду с этим команда `myAnimation.appendRule('53% {top: 50%;}')`, наоборот, добавляет объявление ключевого кадра с селектором `53%` в конец команды `@keyframes` (см. пример на сайте книги; файл *appendRule.html*).

Анимация сопровождается возникновением сразу трех событий — `animation-start`, `animationend` и `animationiteration`, — генерируемых в начале, в конце и после завершения текущего и перед началом следующего шага анимации. События начала и конца анимации генерируются в любой действующей анимации с правильно объявленными ключевыми кадрами. Событие `animationiteration` возникает только в многократно повторяемой анимации и не генерируется одновременно с событием `animationend`.

Анимация элементов

После создания именованной анимации по ключевым кадрам нужно назначить ее целевым элементам и псевдоэлементам. В CSS применение и управление воспроизведением анимации по ключевым кадрам возлагается на целый набор свойств. В самом простом случае нужно указать название анимации, отвечающей за динамическое преобразование элемента, и определить ее длительность (в противном случае преобразования, устанавливаемые анимацией, будут выполнены мгновенно).

Существуют два способа управления анимацией элемента: объявление всех необходимых свойств по отдельности или определение всех настроек с помощью свойства общего назначения `animation` (функционально равнозначного всем индивидуальным свойствам). Далее мы сначала ознакомимся с назначением индивидуальных свойств управления анимацией и только после этого перейдем к изучению общего свойства `animation`.

Именованная анимация

Свойство `animation-name` получает в качестве значения список разделенных запятыми названий анимации, применяемых к элементу. Названия анимации представляются идентификаторами или строковыми значениями, заключенными в кавычки (допускается комбинирование значений) и объявленными ранее в командах `@keyframes`.

animation-name

Значение	[<*single-animation-name*> | `none`]#
Начальное значение	`none`
Применяется	Все элементы и псевдоэлементы `:before` и `:after`
Вычисляется	Согласно определению
Наследуется	Нет
Анимируется	Нет

По умолчанию это свойство имеет значение `none`, предотвращая какую бы то ни было анимацию элемента. Кроме того, значение `none` можно использовать для отмены заданной ранее анимации (по этой же причине не стоит давать анимации название `none`). Для применения анимации к целевому элементу ее название, обозначенное правилом `@keyframes`, нужно передать свойству `animation-name` (см. пример на сайте книги; файл *badnames.html*).

Вот каким образом можно запустить анимацию по ключевым кадрам `change_bgcolor`, описанную в разделе "Исключение начального и конечного селекторов".

```
div {
   animation-name: change_bgcolor;
}
```

Выполнение такого кода приведет к применению анимации `change_bgcolor` сразу ко всем элементам `div` документа.

Чтобы применить к элементу сразу несколько типов анимации, добавьте все их названия в значение свойства `animation-name`, разделив запятой.

```
div {
   animation-name: change_bgcolor, round, W;
}
```

Передача свойству animation-name недействительного идентификатора приводит к отмене анимации, представленной только им, но не другими (действительными) идентификаторами. Несмотря на это любая исходно недействительная анимация будет воспроизводиться, как только станет доступна обработчику браузера. Рассмотрим такой пример.

```
div {
   animation-name: change_bgcolor, spin, round, W;
}
```

Предположим, что в коде этого примера отсутствует объявление анимации spin. Тогда в результате выполнения приведенного выше правила элемент получит анимацию, обусловленную только идентификаторами change_bgcolor, round и W. Анимация spin тоже будет воспроизводиться, но только по мере доступности в текущей таблице стилей (см. пример на сайте книги; файл *nameaddedlater.html*).

Итак, добавление в значение свойства animation-name идентификаторов сразу нескольких анимаций гарантирует применение их всех к целевому элементу. При этом анимация повторяющихся свойств сводится к замене всех их значений из предыдущей анимации значениями, заданными в последней анимации. Например, если динамическое преобразование фонового цвета обусловливается двумя последовательно выполняемыми анимациями, то фоновый цвет будет определять последняя из них *и только* в случае одновременного их воспроизведения (см. пример на сайте книги; файл *no0or100.html*). Детально правила анимации повторяющихся свойств рассматриваются в разделе "Анимация: приоритетность и последовательность выполнения".

При рассмотрении следующего примера, как и всех дальнейших, предположим, что анимация длится 10 секунд.

```
div {animation-name: change_bgcolor, bg-shift;}

@keyframes bg-shift {
   0%, 100% {background-color: blue;}
   35% {background-color: orange;}
   55% {background-color: red;}
   65% {background-color: purple;}
}
@keyframes change_bgcolor {
   0%, 100% {background-color: yellow;}
   45% {background-color: green;}
   55% {background-color: blue;}
}
```

Согласно настройкам анимации bg-shift, фоновый цвет элемента должен изменяться с синего на оранжевый, а затем переходить в красный, далее — в фиолетовый и только после этого возвращаться к синему. При воспроизведении анимации предпочтение всегда отдается тем ключевым кадрам, в которых определяются последние значения анимируемых свойств. Если одно и то же свойство одновременно настраивается в одних и тех же ключевых кадрах сразу нескольких правил @keyframes, то изменяться оно будет согласно настройкам ключевых кадров последнего правила (указанного в списке идентификаторов свойства animation-name).

Еще более интересная ситуация наблюдается при опускании в объявлении отдельных типов анимации начального (0%) и конечного (100%) ключевых кадров. В качестве примера рассмотрим, что произойдет при удалении их из определения анимации bg-shift.

```
div {animation-name: change_bgcolor, bg-shift;}

@keyframes bg-shift {
   35% {background-color: orange;}
   55% {background-color: red;}
   65% {background-color: purple;}
}
@keyframes change_bgcolor {
   0%, 100% {background-color: yellow;}
   45% {background-color: green;}
   55% {background-color: blue;}
}
```

Настройки анимации bg-shift, создаваемой с помощью правила @keyframes, не предполагают установку для элемента фонового цвета в ее начале и конце. Как известно, в отсутствие объявления ключевого кадра 0% или 100% состояние элемента в указанных временны*х* точках анимации рассчитывается пользовательским агентом самостоятельно. Принятие решения основывается на одном из двух предположений: в указанных точках свойство имеет такое же значение, что и в отсутствие анимации, либо же определяется предыдущей анимацией (предыдущей в списке свойства animation-name).

Старые браузеры интерполируют начальный и конечный ключевые кадры согласно первому принципу, а новые — согласно второму. К концу 2017 года в новых браузерах такая анимация фонового цвета начиналась с перехода желтого цвета в оранжевый, а завершалась переходом фиолетового цвета в синий. При этом переход к каждому следующему ключевому кадру длится ровно 3,5 с. В старых браузерах анимация начиналась и заканчивалась полностью прозрачным фоном.

Старайтесь предельно внимательно относиться к свойствам, подлежащим одновременной анимации сразу в нескольких ключевых кадрах. В данном случае все очень просто: такая анимация выполняется для свойства background-color. Одновременная анимация совершенно разных свойств, например background-color и padding, не вызывает особых затруднений. В данном случае изменение фонового цвета и полей элемента выполняется одновременно и без заметных накладок.

Применение анимации по ключевым кадрам к элементу не приводит к ее визуализации в документе, но этого вполне достаточно для ее выполнения — пусть даже и моментального. На данном этапе просчитываются значения свойств во всех объявленных в анимации ключевых кадрах и генерируются события animationstart и animationend. Чтобы представить анимацию в режиме, обеспечивающем ее визуализацию в реальном времени, ее нужно снабдить некоей длительностью. Для решения этой задачи используется свойство animation-duration.

Длительность анимации

Свойство `animation-duration` определяет время, в течение которого происходит каждая итерация анимации, выражаемое в секундах (`s`) или миллисекундах (`ms`).

animation-duration	
Значение	<time>#
Начальное значение	`0s`
Применяется	Все элементы и псевдоэлементы `:before` и `:after`
Вычисляется	Согласно определению
Наследуется	Нет
Анимируется	Нет

Это свойство указывает длительность одной итерации цикла анимации по всем ее ключевым кадрам. В отсутствие объявления свойства `animation-duration` анимация элемента также выполняется, но длится ровно `0s`. При этом события `animationstart` и `animationend` будут возникать, несмотря на генерирование в единственно возможной временной точке. Это свойство не поддерживает отрицательные числовые значения.

При указании длительности анимации числовое значение необходимо задавать в единицах измерения: секундах (`s`) или миллисекундах (`ms`). Если есть несколько анимаций, можно включать для каждой из них разные значения длительности, разделяемые запятыми.

```
div {
   animation-name: change_bgcolor, round, W;
   animation-duration: 200ms, 100ms, 0.5s;
}
```

При включении в список хотя бы одной неправильной величины (например, `200ms, 0, 0.5s`) недействительным будет считаться все объявление, а анимация будет осуществляться как при задании ее длительности объявлением `animation-duration: 0s`, поскольку значение 0 не представляет действительную временную величину (см. пример на сайте книги; файл *duration_broken_value.html*).

В общем случае свойство `animation-duration` нужно объявлять для каждого свойства `animation-name`. Передача свойству `animation-duration` единственного значения приводит к анимации всех свойств в течение одного и того же интервала времени. Если список свойства `animation-duration` насчитывает меньше разделенных запятой значений, чем объявлено элементов в свойстве `animation-name`, то они будут автоматически повторены до получения необходимого количества временных интервалов. Рассмотрим такой пример.

```
div {
   animation-name: change_bgcolor, spin, round, W;
   animation-duration: 200ms, 5s;
      /* такой же результат, как при 200ms, 5s, 200ms, 5s */
}
```

Здесь анимация round длится 200 мс, а анимация W — 5 мс.

Если количество элементов списка у свойства animation-duration превышает таковое у свойства animation-name, то все "лишние" величины исключаются из обработки. В случае недоступности анимации, указанной в объявлении свойства animation-name, пользовательский агент проигнорирует только ее (включая значение длительности), оставляя действительными все остальные анимации.

```
div {
   animation-name: change_bgcolor, spinner, round, W;
   animation-duration: 200ms, 5s, 100ms, 0.5s;
}
```

В приведенном выше примере временная величина 5ms указывает длительность анимации spinner, но поскольку последняя не объявлена в таблице стилей, то значение 5ms будет проигнорировано обработчиком. Как только анимация spinner станет доступной в документе, она будет применена к элементам div, получив длительность 5 мс.

Повторение анимации

Включение в таблицу стилей объявления свойства animation-name приводит только к однократному воспроизведению анимации. Для выполнения анимации менее одного раза или нескольких итераций подряд их количество необходимо определить в свойстве animation-iteration-count.

animation-iteration-count

Значение	[<*number*> | infinite]#
Начальное значение	1
Применяется	Все элементы и псевдоэлементы :before и :after
Вычисляется	Согласно определению
Наследуется	Нет
Анимируется	Нет

По умолчанию анимация выполняется единожды, что соответствует значению 1. Передача свойству animation-iteration-count иного значения при неотрицательном значении свойства animation-delay приводит к повторению анимации указанное количество раз. Число итераций анимации также можно выразить ключевым словом infinite.

Следующие объявления указывают воспроизводить анимацию соответственно 2 раза, 5 и 13 раз.

```
animation-iteration-count: 2;
animation-iteration-count: 5;
animation-iteration-count: 13;
```

Обозначив количество итераций дробным числом, можно добиться завершения анимации в середине, а не в конце цикла. Технически анимация будет

воспроизводиться как при полном количестве итераций, обрываясь на последнем этапе выполнения. В частности, объявление `animation-iteration-count: 1.25` определяет одну полную итерацию с четвертью, поэтому ее вторая итерация будет завершена по истечении только 25% длительности. При длительности анимации 8 с значение 25% будет соответствовать 2 с.

Отрицательные значения передавать свойству `animation-iteration-count` недопустимо. При обработке недействительного значения анимация выполняется, как и предполагает значение по умолчанию, всего один раз (см. пример на сайте книги; файл *oditeration.html*).

Как ни странно, но значение `0` относится к допустимым, как и объявление `animation-iteration-count: 0`. В подобных случаях анимация выполняется мгновенно, сопровождаясь обязательным генерированием событий `animationstart` и `animationend`.

Для применения к элементу или псевдоэлементу сразу нескольких типов анимации, воспроизводимых разное количество итераций, их настройки нужно указать отдельно (через запятую) в каждом из свойств `animation-name`, `animation-duration` и `animation-iteration-count`.

```
.flag {
   animation-name: red, white, blue;
   animation-duration: 2s, 4s, 6s;
   animation-iteration-count: 3, 5;
}
```

Числовые значения, передаваемые свойству `animation-iteration-count` в виде списка числовых величин, как и аналогичные значения любых других свойств, указываются в порядке применения анимаций, объявляемых в свойстве `animation-name`. При этом избыточные величины не обрабатываются, а недостающие значения дополняются за счет повторения уже существующих.

В предыдущем примере количество значений, определяющих число итераций, меньше, чем самих анимаций, из которых `red` и `blue` повторяются трижды, а `white` — пять раз. Число остальных настроек (тип анимации и их длительность) полностью совпадает. Таким образом, анимация `red` длится 2 с и повторяется трижды, поэтому общее время ее воспроизведения составляет 6 с. При этом анимация `white` длится 4 секунды и повторяется 5 раз, поэтому ее полный просмотр займет 20 с. А вот анимация `blue` длится 6 с при троекратном повторении — суммарно 18 с.

Передача неправильных значений приводит к отмене всего объявления и сбросу анимации только до одноразового выполнения.

Свойство `animation-iteration-count` удобно применять для одновременного завершения нескольких — в данном случае трех — анимаций, имеющих разную длительность.

```
.flag {
   animation-name: red, white, blue;
   animation-duration: 2s, 4s, 6s;
   animation-iteration-count: 6, 3, 2;
}
```

В этом примере все анимации (red, white и blue) выполняются в течение ровно 12 секунд, поскольку именно такое число получается при умножении длительности на количество итераций каждой из них.

Направление анимации

Свойство animation-direction устанавливает направление или порядок анимации ключевых кадров: от 0% до 100% или, наоборот, от 100% до 0%. Кроме того, с его помощью указывается, будет ли изменяться направление воспроизведения анимации в каждой последующей итерации цикла.

animation-direction

Значение	[normal | reverse | alternate | alternate-reverse]#
Начальное значение	normal
Применяется	Все элементы и псевдоэлементы :before и :after
Вычисляется	Согласно определению
Наследуется	Нет
Анимируется	Нет

Направление перехода между ключевыми кадрами анимации устанавливается такими ключевыми словами.

normal

Явное объявление или установка по умолчанию обеспечивает выполнение каждой итерации анимации в направлении от ключевого кадра 0% до 100%.

reverse

Воспроизводит каждую итерацию анимации в обратном направлении: от ключевого кадра 100% до 0%. Приводит к инвертированию временной функции, указываемой свойством animation-timing-function (об этом будет говориться в разделе "Временная функция анимации").

alternate

Определяет прямой порядок выполнения анимации (от ключевого кадра 0% до 100%) в первой и всех последующих непарных итерациях и обратный порядок анимации во второй и всех последующих парных итерациях (от ключевого кадра 100% до 0%).

alternate-reverse

Полностью обратен предыдущему режиму. В первой и всех последующих непарных итерациях применяется обратный порядок анимации (от ключевого кадра 100% до 0%), а во второй и всех последующих парных итерациях — прямой порядок анимации.

```
.ball {
   animation-name: bouncing;
   animation-duration: 400ms;
   animation-iteration-count: infinite;
   animation-direction: alternate-reverse;
}
@keyframes bouncing {
   from {
      transform: translateY(500px);
   }
   to {
      transform: translateY(0);
   }
}
```

В данном примере создается эффект прыгающего мяча: в начале анимации он падает вниз, а не отскакивает вверх. Объявление `animation-direction: alternate-reverse` лучше всего подходит для создания анимации такого типа: вниз, а затем вверх, в противоположность анимации движения вверх, а затем вниз (см. пример на сайте книги; файл *ball6.html*).

Обратите внимание на движение мяча при отскакивании от некоей поверхности. Наибольшее замедление мяч получает при подлете к верхней точке своей траектории, а наибольшее ускорение — непосредственно перед касанием поверхности. В текущем контексте интерес вызывает только направление анимации. Мы обязательно вернемся к его рассмотрению при изучении временны*х* функций, от выбора которых зависит реалистичность хода анимации (см. ниже раздел "Временные функции анимации"). Далее вы также узнаете о причинах инвертирования временно*й* функции при использовании объявления `animation-direction: alternate-reverse`.

Задержка анимации

Назначение свойства `animation-delay` заключается в определении времени, на которое отсрочено начало выполнения первой итерации анимации.

animation-delay

Значение	*<time>*#
Начальное значение	`0s`
Применяется	Все элементы и псевдоэлементы `:before` и `:after`
Вычисляется	Согласно определению
Наследуется	Нет
Анимируется	Нет

С помощью этого свойства указывается задержка в секундах (`s`) или миллисекундах (`ms`) между применением анимации к элементу и началом ее воспроизведения.

По умолчанию анимация начинается сразу же после объявления и назначения целевому элементу, т.е. с нулевой задержкой. Положительные значения свойства

animation-delay определяют отсрочку выполнения анимации на заданное количество времени, а отрицательные требуют выполнять ее немедленно, но не с начала, а с временной точки, отстоящей от начала на заданный числовым значением промежуток времени.

Возможность обработки отрицательных значений позволяет добиться удивительных результатов. Например, отрицательная задержка, устанавливаемая объявлением animation-delay: -4s при длительности анимации, определяемой с помощью объявления animation-duration: 10s, предполагает, что анимация будет начинаться с момента завершения приблизительно 40% ее первой итерации и будет длиться не 10, а всего 6 секунд.

Термин "приблизительно" использован в последнем утверждении умышленно: точное время начала воспроизведения анимации с отрицательной задержкой зависит от задаваемой свойством animation-timing-function временной функции и может несколько отличаться от числового значения, указываемого свойством animation-delay. Точное совпадение будет наблюдаться только при постоянной скорости хода анимации (animation-timingfunction: linear).

```
div {
   animation-name: move;
   animation-duration: 10s;
   animation-delay: -4s;
   animation-timing-function: linear;
}
@keyframes move {
   from {
      transform: translateX(0);
   }
   to {
      transform: translateX(1000px);
   }
}
```

В случае линейной временной функции анимация будет начинаться точно с четвертой секунды. В примере с анимацией перемещения элемент вначале будет моментально смещен вправо на 400 пикселей (относительно исходного положения), а затем — в течение оставшихся 6 секунд — будет плавно смещаться в этом же направлении (см. пример на сайте книги; файл *fortypercent.html*).

Установка задержки –600 мс для анимации, выполняемой 10 итераций при длительности 200 мс, равнозначна запуску ее с начала четвертой итерации.

```
.ball {
   animation-name: bounce;
   animation-duration: 200ms;
   animation-delay: -600ms;
   animation-iteration-count: 10;
   animation-timing-function: ease-in;
   animation-direction: alternate;
}
```

```
@keyframes bounce {
   from {
      transform: translateY(0);
   }
   to {
      transform: translateY(500px);
   }
}
```

Вместо того чтобы выполняться в течение 2000 мс (200 мс × 10 = 2000 мс, или 2 с), анимация прыгающего мяча будет длиться всего 1400 мс, или 1,4 с. При этом она будет начинаться с той же точки, что и такая же анимация, лишенная задержки *и* выполняемая в противоположном направлении.

Обратное направление анимации при задержке длительностью в четыре итерации обеспечивается передачей свойству `animation-direction` значения `alternate`. В соответствии с ним каждая следующая итерация выполняется в противоположном по отношению к предыдущей итерации направлении. Как результат, любая парная, в том числе и четвертая, итерация будет характеризоваться обратным ходом анимации (см. пример на сайте книги; файл *ball_animation_delay_negative.html*).

В данном типе анимации событие `animationstart` генерируется немедленно, а событие `animationend` — только по истечении 1400 мс. К этому времени мяч спустя 6 полных итераций анимации (200, 400, 600, 800, 1,000 и 1200 мс) будет находиться в состоянии полета вверх. При этом по истечении десятой итерации возникнет только 6 событий `animationiteration` и будет выполняться седьмая (четыре начальные итерации будут пропущены благодаря отрицательной задержке). Событие `animationend` возникает в точности в момент завершения последней итерации анимации. Учтите, что событие `animationiteration` не возникает в момент генерирования события `animationend`, даже несмотря на выполнение всех необходимых условий.

Теперь более детально рассмотрим, какие события возникают при воспроизведении анимации.

События анимации

Процесс анимации сопровождается возникновением событий трех основных типов: `animationstart`, `animationiteratation` и `animationend`. Каждое из этих событий характеризуется тремя свойствами с атрибутом только для чтения: `animationName`, `elapsedTime` и `pseudoElement`, прекрасно обрабатываемыми всеми браузерами без вендорных префиксов.

Событие `animationstart` генерируется в начале каждой анимации — немедленно или по истечении действия свойства `animation-delay` (при объявлении такового). В случае передачи свойству `animation-delay` отрицательного значения это событие также генерируется немедленно, передавая свойству `elapsedTime` абсолютное значение задержки, распознаваемое всеми поддерживающими его браузерами. В браузерах, работающих только с вендорными свойствами, свойство `elapsedTime` получает значение `0`.

```
.noAnimationEnd {
   animation-name: myAnimation;
   animation-duration: 1s;
   animation-iteration-count: infinite;
}
.startAndEndSimultaneously {
   animation-name: myAnimation;
   animation-duration: 0s;
   animation-iteration-count: infinite;
}
```

Событие `animationend` возникает по завершении анимации. Событие `animationend` никогда не наступает при установке свойства `animation-iteration-count` в значение `infinite` и одновременной передаче свойству `animation-duration` ненулевого значения. Если же свойство `animation-duration` получает значение по умолчанию `0`, то события `animationstart` и `animationend` возникают одновременно, но строго в указанной выше последовательности.

Событие `animationiteration` возникает *между* итерациями анимации (см. пример на сайте книги; файл *events.html*). При этом оно генерируется по завершении каждой, кроме последней, итерации анимации, что позволяет избежать одновременного возникновения сразу двух событий: `animationiteration` и `animationend`.

```
.noAnimationIteration {
   animation-name: myAnimation;
   animation-duration: 1s;
   animation-iteration-count: 1;
}
```

В примере анимации `.noAnimationIteration`, состоящей всего из одной итерации (`animation-iteration-count: 1;`), завершение анимации совпадает с завершением единственной итерации. Таким образом, событие `animationiteration` может возникать только одновременно с событием `animationend`, чего по требованию спецификации никогда не происходит. Генерирование события `animationiteration` должно сопровождаться немедленным запуском новой итерации.

Событие `animationiteration` не возникает в отсутствие в стилевом правиле явного объявления свойства `animation-iteration-count` или при передаче ему значения, меньшего или равного `1`. Оно будет генерироваться только по завершении (даже частичном) текущей и перед запуском последующей итерации с ненулевой длительностью.

```
.noAnimationIteration {
   animation-name: myAnimation;
   animation-duration: 1s;
   animation-iteration-count: 4;
   animation-delay: -3s;
}
```

При уменьшении количества итераций (`animation-iteration-count`) вследствие назначения свойству `animation-delay` отрицательного значения событие `animationiteration` не будет возникать между пропущенными итерациями.

Приведенный выше код как раз иллюстрирует данный принцип. При его выполнении события `animationiteration` не генерируются вообще, поскольку первые три итерации анимации не выполняются благодаря задержке `animation-delay: -3s` (см. пример на сайте книги; файл *events2.html*).

В предыдущем примере свойство `elapsedTime` события `animationstart` получает значение `3`, представляя абсолютное значение времени задержки.

Создание последовательностей анимации

Грамотно определяя значения свойств `animation-delay`, можно создавать цепочки анимаций, в которых каждая следующая анимация начинается сразу же по завершении предыдущей.

```
.rainbow {
   animation-name: red, orange, yellow, blue, green;
   animation-duration: 1s, 3s, 5s, 7s, 11s;
   animation-delay: 3s, 4s, 7s, 12s, 19s;
}
```

Здесь анимация `red` запускается с трехсекундной задержкой и длится всего одну секунду, что предопределяет возникновение события `animationend` на четвертой секунде. Все последующие типы анимации настраиваются таким образом, чтобы запускаться сразу по завершении предыдущей анимации. В CSS такой способ анимации называется созданием *последовательностей анимации* (см. пример на сайте книги; файл *animationchain.html*).

Добавление четырехсекундной задержки для анимации `orange` приводит к тому, что ее свойства, объявленные в команде `@keyframe`, вступают в силу сразу же по завершении анимации `red`, длящейся ровно 4 с. Наряду с этим анимация `orange` завершается на временной отметке 7 с — при длительности 3 с ее запуск будет отсрочен на оговоренные выше 4 с. Именно такая задержка определена для следующей анимации, `yellow`, что позволяет ей запускаться сразу после анимации `orange`.

Приведенный выше код обусловливает анимацию только отдельно взятого элемента. Тем не менее свойство `animation-delay` позволяет создавать последовательности анимации сразу для нескольких элементов.

```
li:first-of-type {
   animation-name: red;
   animation-duration: 1s;
   animation-delay: 3s;
}
li:nth-of-type(2) {
   animation-name: orange;
   animation-duration: 3s;
   animation-delay: 4s;
}
li:nth-of-type(3) {
   animation-name: yellow;
   animation-duration: 5s;
   animation-delay: 7s;
}
```

```
li:nth-of-type(4) {
   animation-name: green;
   animation-duration: 7s;
   animation-delay: 12s;
}
li:nth-of-type(5) {
   animation-name: blue;
   animation-duration: 11s;
   animation-delay: 19s;
}
```

Для последовательной анимации каждого из элементов списка их свойства `animation-delay` нужно настраивать таким образом, чтобы их значения включали значения свойств `animation-duration` и `animation-delay` предыдущих элементов (см. пример на сайте книги; файл *animationchain2a.html*).

Несмотря на то что момент запуска каждой следующей анимации можно определять по событию `animationEnd` языка JavaScript (см. ниже), применение для этих целей свойства `animation-delay` считается более предпочтительным и эффективным с точки зрения настройки последовательности анимации. Помните о том, что анимация обладает более низким приоритетом, чем потоки пользовательского интерфейса. Следовательно, при запуске JavaScript-сценария из потока пользовательского интерфейса время ожидания выполнения в нем других операций может превысить задержку, устанавливаемую для анимаций последовательности, что в первую очередь зависит от типа браузера, способа и длительности анимации свойств. В результате последовательность анимации элементов может быть нарушена.

Производительность анимации

Отдельные, хотя далеко не все, типы анимации запускаются из потока пользовательского интерфейса. В большинстве браузеров анимация прозрачности и трансформации объектов вычисляется графическим (Graphics Processing Unit — GPU), а не центральным (Central Processing Unit — CPU) процессором, поэтому выполняется независимо от загруженности потока пользовательского интерфейса. Анимация остальных типов (свойств) при недоступном потоке пользовательского интерфейса выполняется рывками (с так называемым "подергиванием").

```
/* Ни в коем случае не повторяйте */
* {
   transform: translateZ(0);
}
```

При визуализации документов на устройствах и в браузерах, поддерживающих 3D-трансформацию (см. главу 16), их элементы размещаются на отдельных слоях, что позволяет избежать "подергивания" изображения при воспроизведении анимации. Но все же не стоит злоупотреблять свойством `translateZ`, о чем говорится в комментарии в приведенном выше коде.

Под каждый слой системой выделяется определенный объем видеопамяти, а занесение в нее данных из потока пользовательского интерфейса занимает достаточно много времени. Таким образом, чем больше у документа слоев, тем большая производительность требуется для плавного отображения всех визуальных эффектов.

Чтобы повысить скорость обработки документа, по возможности выполняйте анимацию свойств `transform` и `opacity` перед анимацией свойств `top`, `left`, `bottom`, `right` и `visibility`. Это позволит задействовать для проведения вычислений графический процессор, а также ускорит перерисовку элементов при изменении свойств блочной модели. При этом избегайте обработки всех типов анимации одним только графическим процессором: это чревато серьезными сбоями в визуализации всего документа.

Планируя использовать инструменты языка JavaScript для управления последовательностью анимации, добейтесь запуска каждой следующей анимации по событию `animationend`, выполняя его прослушивание в коде сценария (см. пример на сайте книги; файл *animationchain2.html*).

```
<script>
   document.querySelectorAll('li')[0].addEventListener('animationend',
      function(e) {
         document.querySelectorAll('li')[1].style.animationName =
            'orange';
      },
      false );

   document.querySelectorAll('li')[1].addEventListener('animationend',
      function(e) {
         document.querySelectorAll('li')[2].style.animationName =
            'yellow';
      },
      false );

   document.querySelectorAll('li')[2].addEventListener('animationend',
      function(e) {
         document.querySelectorAll('li')[3].style.animationName =
            'green';
      },
      false );

   document.querySelectorAll('li')[3].addEventListener('animationend',
      function(e) {
         document.querySelectorAll('li')[4].style.animationName =
            'blue';
      },
      false );
</script>
```

```
<style>
   li:first-of-type {
      animation-name: red;
      animation-duration: 1s;
   }
   li:nth-of-type(2) {
      animation-duration: 3s;
   }
   li:nth-of-type(3) {
      animation-duration: 5s;
   }
   li:nth-of-type(4) {
      animation-duration: 7s;
   }
   li:nth-of-type(5) {
      animation-duration: 11s;
   }
</style>
```

В этом примере каждому из четырех элементов списка назначен собственный обработчик, определяющий наступление события `animationend`. При его возникновении слушатель событий применяет свойство `animation-name` к соответствующему элементу списка.

Легко заметить, что задействованный в этом примере метод построения последовательности анимации не требует участия свойства `animation-delay`. Управление анимацией осуществляется с помощью обработчиков JavaScript, прослушивающих события `animationend` и активизирующих свойство `animation-name`.

Обратите внимание на то, что свойство `animation-name` объявлено только для первого элемента списка. Остальные элементы списка снабжаются только свойством `animation-duration` — в свойстве `animation-name` нет необходимости, поскольку анимация всегда начинается с первого элемента. Ее остановка и перезапуск требуют обязательного сброса и повторного назначения свойства `animation-name`, и только после этого будут обрабатываться остальные свойства анимации, включая `animation-delay`.

Таким образом, приведенный ниже код

```
<script>
   document.querySelectorAll('li')[2].addEventListener('animationend',
      function(e) {
         document.querySelectorAll('li')[3].style.animationName =
            'green';
      },
      false );

   document.querySelectorAll('li')[3].addEventListener('animationend',
      function(e) {
         document.querySelectorAll('li')[4].style.animationName =
            'blue';
      },
      false );
</script>
```

```
<style>
   li:nth-of-type(4) {
      animation-duration: 7s;
   }
   li:nth-of-type(5) {
      animation-duration: 11s;
   }
</style>
```

можно заменить таким:

```
<script>
   document.querySelectorAll('li')[2].addEventListener('animationend',
      function(e) {
         document.querySelectorAll('li')[3].style.animationName =
            'green';
         document.querySelectorAll('li')[4].style.animationName =
            'blue';
      },
      false );
</script>

<style>
   li:nth-of-type(4) {
      animation-duration: 7s;
   }
   li:nth-of-type(5) {
      animation-delay: 7s;
      animation-duration: 11s;
   }
</style>
```

Добавление анимации `blue` к пятому элементу списка выполняется в тот же момент времени, что и анимации `green` к четвертому элементу, но благодаря правильно подобранной задержке они будут выполняться последовательно одна за другой.

Попытка изменить значения свойств настройки анимации, отличные от `animation-name`, во время ее воспроизведения не возымеет должного эффекта. В уже запущенной анимации невозможно изменить длительность, например с `100ms` на `400ms`. Аналогичным образом невозможно представить подмену задержки `-200ms` на `5s` в процессе воспроизведения анимации. Тем не менее анимацию можно останавливать и перезапускать в произвольный момент ее выполнения. Для этого достаточно удалить ее и применить повторно. В примере управления анимацией с помощью JavaScript она запускалась, назначаясь элементам в явном виде.

Любая анимация отменяется путем добавления в код таблицы стилей объявления `display: none`. Для запуска анимации с самого начала свойству `display` нужно передать одно из более значимых ключевых слов. Если при этом свойство `animation-delay` получает положительное

значение, то устанавливаемая им задержка истечет до возникновения события animationstart, а значит, и самой анимации. В случае отрицательной задержки анимация будет начата с определенного момента итерации, как при запуске любым другим способом.

Задержка итераций анимации

Несмотря на отсутствие свойства animation-iteration-delay, спецификация позволяет отсрочить выполнение отдельных итераций анимации, используя уже известное свойство animation-delay — как в команде @keyframes, так и в сценариях JavaScript. Конечный способ решения такой задачи зависит от количества итераций, производительности оборудования и равенства временны*х* задержек, определяемых для каждой итерации.

Что же представляет собой задержка итерации и зачем она нужна? В отдельных случаях нужно не только повторить анимацию строго заданное число раз, но и выполнить каждую ее итерацию с новой задержкой.

Предположим, элемент нужно увеличить до конечного размера за три подхода, каждый следующий из которых запускается спустя 4 с после завершения предыдущего при секундной длительности каждой итерации. Один из способов решения задачи состоит в настройке соответствующей задержки в правиле @keyframes и последующем троекратном повторении всей анимации.

```
.animate3times {
   background-color: red;
   animation: color_and_scale_after_delay;
   animation-iteration-count: 3;
   animation-duration: 5s;
}

@keyframes color_and_scale_after_delay {
   80% {
      transform: scale(1);
      background-color: red;
   }
   80.1% {
      background-color: green;
      transform: scale(0.5);
   }
   100% {
      background-color: yellow;
      transform: scale(1.5);
   }
}
```

Заметьте, селектор первого ключевого кадра, представляющий состояние по умолчанию, устанавливается в точке 80%, что обеспечивает следующий тип анимации элемента в течение трех итераций: в исходном состоянии элемент находится 80% времени (4 с), но за последнюю секунду итерации его фон меняется с зеленого на желтый, а сам он увеличивается в размерах (см. пример на сайте книги; файл

animation-iteration-delay1.html). Изменение элемента завершается по истечении третьей итерации анимации.

Описанный выше способ задержки итераций может применяться в анимации с произвольным количеством итераций. Конечно, проще всего настраивать анимацию, в которой все итерации запускаются с одинаковой задержкой. Такой подход устраняет необходимость добавления в правило `@keyframes` дополнительных объявлений, например для указания специальной задержки длительностью 6 секунд (см. пример на сайте книги; файл *animation-iteration-delay.html*). Чтобы назначить каждой итерации собственную задержку при сохранении длительности анимации свойств настройки фонового цвета и размера элемента, понадобится полностью переписать правило `@keyframes`.

Назначение разных задержек сводится к точному расчету селекторов ключевых кадров, представляющих каждую из итераций анимации.

```
.animate3times {
   background-color: red;
   animation: color_and_scale_3_times;
   animation-iteration-count: 1;
   animation-duration: 15s;
}

@keyframes color_and_scale_3_times {
   0%, 13.32%, 20.01%, 40%, 46.67%, 93.32% {
      transform: scale(1);
      background-color: red;
   }
   13.33%, 40.01%, 93.33% {
      background-color: green;
      transform: scale(0.5);
   }
   20%, 46.66%, 100% {
      background-color: yellow;
      transform: scale(1.5);
   }
}
```

Такой подход вызывает трудности не только на этапе написания, но и в процессе дальнейшего сопровождения кода (см. пример на сайте книги; файл *animation-iteration-delay2.html*). Кроме того, он рассматривается как одна итерация анимации. Для изменения задержки или добавления в правило еще одного типа анимации потребуется написать другое правило `@keyframes` или отредактировать уже имеющееся. При всем этом приведенное выше правило оказывается менее производительным, чем предыдущее, хотя и обладает собственными уникальными возможностями.

Существует еще один способ установки в итерациях анимации разных задержек. Он поддерживается большинством браузеров, хотя и не определен спецификацией. Заключается он в многократном повторении анимации, каждый раз с другой задержкой (см. пример на сайте книги; файл *animation-iteration-delay3.html*).

```
.animate3times {
   animation: color_and_scale, color_and_scale, color_and_scale;
   animation-delay: 0, 4s, 10s;
   animation-duration: 1s;
}

@keyframes color_and_scale {
   0% {
      background-color: green;
      transform: scale(0.5);
   }
   100% {
      background-color: yellow;
      transform: scale(1.5);
   }
}
```

Здесь одна и та же анимация запускается трижды, каждый раз с новым значением задержки. Такой подход создает эффект выполнения отдельных итераций одного и того же типа анимации через разные временны*е* промежутки.

Если в процессе выполнения одна анимация накладывается на другую, то для предотвращения конфликтной ситуации в ней будет применяться такая же задержка, как в предыдущей успешно завершенной анимации. Как и прежде, при последовательной анимации одного и того же свойства его окончательное значение будет определяться последней анимацией, указанной в списке. В случае последовательного выполнения трех экземпляров анимации `color_and_scale`, запускаемых через разные временны*е* интервалы, значение анимируемого свойства будет устанавливаться исключительно третьим экземпляром, даже несмотря на незаконченность воспроизведения предыдущих экземпляров (см. пример на сайте книги; файл *animation-iteration-delay4.html*).

Самый безопасный, наиболее производительный и совместимый с разными типами браузеров способ имитации свойства `animation-iteration-delay` предполагает использование событий анимации. В приведенном выше коде анимация отменяется по событию `animationend`, после чего повторно применяется к элементу после специально заданной задержки. Для настройки одинаковой задержки можно применять функцию `setInterval()`, а установка разных временны*х* задержек требует использования метода `setTimeout()`.

```
var iteration = 0;
var el = document.getElementById('myElement');

el.addEventListener('animationend', function(e) {
   var time = ++iteration * 1000;

   el.classList.remove('animationClass');

   setTimeout(function() {
      el.classList.add('animationClass');
   }, time);
});
```

В приведенном выше примере элемент `myElement` подвергается бесконечной анимации, временная задержка в начале которой увеличивается на одну секунду на каждой следующей итерации (см. пример на сайте книги; файл *animation-iteration-delay.html*).

Временная функция анимации

Использование сценариев для управления анимацией — это действительно увлекательное занятие! Но вернемся к инструментам настройки анимации в CSS, в частности, отвечающим за определение временных функций. Подобно свойству `transition-timing-function`, свойство `animation-timing-function` определяет поведение или скорость изменения свойства, подверженного анимации, в течение всего времени его выполнения (итерации).

animation-timing-function

Значение	[`ease` | `linear` | `ease-in` | `ease-out` | `ease-in-out` | `step-start` | `step-end` | `steps(`*<integer>*`, start)` | `steps(`*<integer>*`, end)` | `cubic-bezier(`*<number>*`, `*<number>*`, `*<number>*`, `*<number>*`)`]#
Начальное значение	`ease`
Применяется	Все элементы и псевдоэлементы `:before` и `:after`
Вычисляется	Согласно определению
Наследуется	Нет
Анимируется	Нет

В отличие от шаговых функций, описанных в разделе "Шаговая временная функция", обычные временные функции описываются кривыми Безье. Свойство `animation-timing-function`, как и свойство `transition-timing-function`, обрабатывает пять значений, представленных исключительно ключевыми словами. Они представлены в табл. 18.1 и на рис. 18.1

Таблица 18.1. Кривые Безье и ключевые слова

Временная функция	Кривая Безье
`ease`	`cubic-bezier(0.25, 0.1, 0.25, 1)`
`linear`	`cubic-bezier(0, 0, 1, 1)`
`ease-in`	`cubic-bezier(0.42, 0, 1, 1)`
`ease-out`	`cubic-bezier(0, 0, 0.58, 1)`
`ease-in-out`	`cubic-bezier(0.42, 0, 0.58, 1)`

Визуализация представленных выше и многих других кривых Безье выполняется с помощью всевозможных онлайн-служб, подобных представленным на сайте `cubic-bezier.com`.

Значение `ease`, задаваемое по умолчанию, определяет анимацию с несколько замедленным стартом, последующим заметным ускорением и плавным завершением. Оно представляет временную функцию, во многом напоминающую функцию

ease-in-out, которая характеризуется более выраженным ускорением анимации в ее середине. Анимация, представляемая временной функцией linear, выполняется с постоянной скоростью.

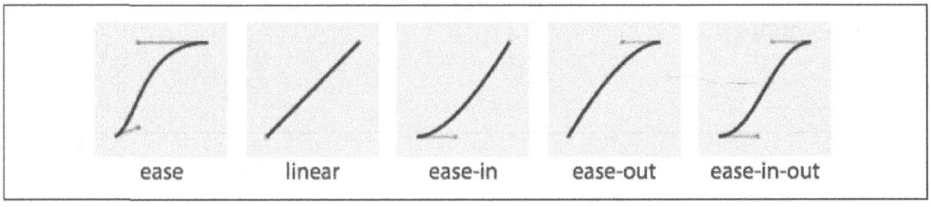


Рис. 18.1. Варианты кривых Безье

Значение ease-in задает анимацию с небольшой скоростью в начале, заметным ускорением в середине и резкой остановкой в конце. Противоположная ей функция ease-out обеспечивает анимацию с максимальной скоростью в начале и постепенным убыванием к концу.

Если ни один из вариантов, представленных ключевыми словами, не подходит для решения поставленной задачи, то всегда можно создать собственную временную зависимость, передав соответствующие параметры в общую функцию.

```
animation-timing-function: cubic-bezier(0.2, 0.4, 0.6, 0.8);
```

Кривые Безье, используемые в CSS, описываются параметрическими уравнениями и отображаются в графическом виде исключительно в двумерном пространстве. Подробно пользовательские кривые Безье описаны в приложении А.

Форма кривой Безье задается четырьмя параметрами, устанавливающими точное положение двух опорных точек (в CSS начальная и конечная точки кривой Безье определяются координатами 0,0 и 1,1). Первые два параметра устанавливают координаты опорной точки для начальной точки кривой Безье, а координаты опорной точки для ее конечной точки задаются третьим и четвертым параметрами функции. Координата *x* определяется числовым значением от 0 до 1 — любые другие значения считаются недействительными. При этом координата *y* ограничению не подлежит. При построении собственных кривых Безье не забывайте о том, что скорость анимации определяется крутизной функции. В общем случае чем более пологая кривая, тем меньше скорость анимации.

Ограничение значений параметров диапазоном 0–1 касается только горизонтальных координат опорных точек. В случае представления вертикальных координат значениями, меньшими 0 или большими 1, анимация будет выражаться колебательными (повторяющимися), а не поступательными движениями. Рассмотрим пример такой анимации, которая выполняется согласно временной функции, представленной кривой Безье достаточно необычной формы (рис. 18.2).

```
.snake {
   animation-name: shrink;
   animation-duration: 10s;
   animation-timing-function: cubic-bezier(0, 4, 1, -4);
   animation-fill-mode: both;
}
```

```
@keyframes shrink {
   0% {
      width: 500px;
   }
   100% {
      width: 100px;
   }
}
```

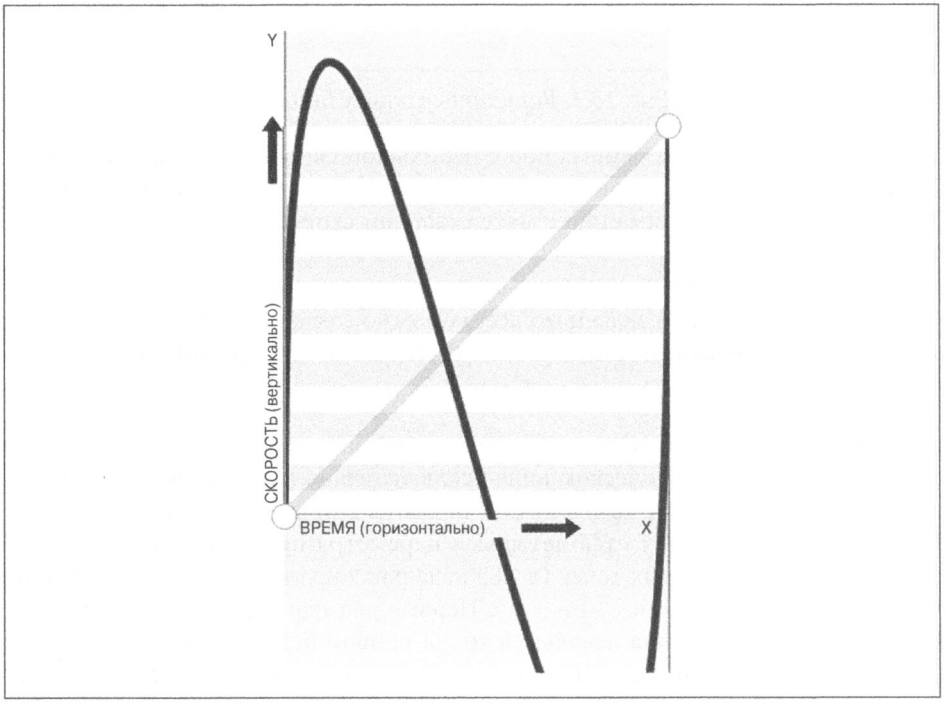


Рис. 18.2. Кривая Безье необычной формы

Кривая Безье, представляемая указанным в приведенном выше коде значением свойства `animation-timing-function`, предполагает выход значений анимируемого свойства за пределы, указываемые в начальном (`0%`) и конечном (`100%`) ключевых кадрах. В рассматриваемом примере элемент должен сжиматься с `500px` до `100px`. Но в действительности изменение размеров элемента осуществляется в более широких пределах (рис. 18.3).

В данном сценарии элемент исходно имеет ширину `500px`, заданную для начального ключевого кадра (`0%`). Очень скоро элемент сжимается до размера `40px`, что значительно меньше значения `width: 100px`, определенного для конечного ключевого кадра (`100%`). Начиная с этого момента ширина элемента медленно увеличивается до `750px`, что несомненно больше значения `500px`, заданного для конечного ключевого кадра (`100%`). На следующем этапе элемент быстро сжимается до исходной ширины `100px`, что знаменует завершение текущей итерации (см. пример на сайте книги; файл *cubicbezierprint.html*).

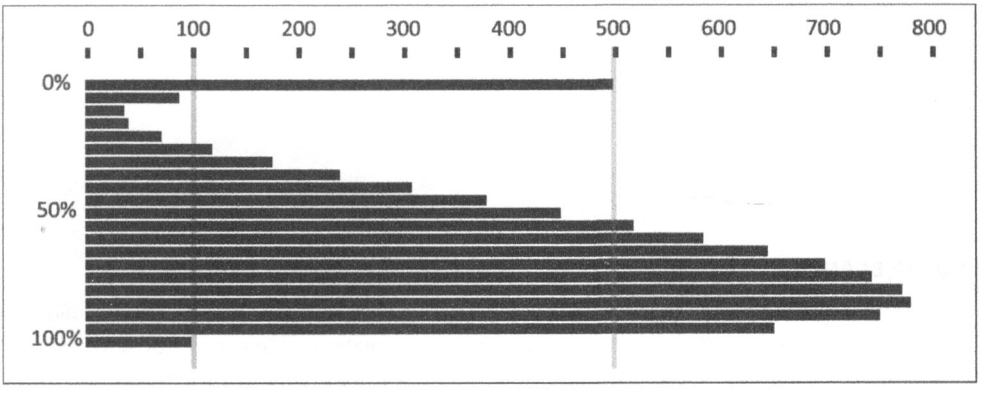


Рис. 18.3. Анимация ширины элемента согласно временной функции, представленной кривой Безье необычной формы

Несложно заметить, что график изменения ширины элемента в процессе анимации имеет такую же форму, как и временная функция (кривая Безье). Подобно тому как S-образная кривая Безье выходит за границы отведенной для нее области, сам анимируемый элемент сужается до значений, меньших 100 пикселей, и расширяется до значений, превосходящих 500 пикселей.

S-образная форма кривой Безье обусловливается разными знаками координаты *y* в крайних ключевых кадрах. Для получения кривой Безье дугообразной формы, полностью помещающейся в отведенное для нее координатное пространство, достаточно задавать координате *y* только положительные значения, *б*ольшие `1`, или только отрицательные значения, меньшие `-1`.

Временные функции, задаваемые свойством `animation-timing-function`, определяют скорость анимации в прямом направлении: от ключевого кадра `0%` до `100%`. Для построения обратной анимации — от ключевого кадра `100%` до `0%` — применяется инвертированная временная функция.

Вернемся к примеру анимации мяча, отскакивающего от поверхности. Легко заметить, что движение мяча выглядит неестественно, поскольку временная функция представлена ключевым словом `ease`, хотя лучше всего для указанных целей применять значение `ease-in`. В последнем случае отскок вверх будет характеризоваться обратным направлением анимации (от ключевого кадра 0% до 100%) и осуществляться согласно инвертированной временной функции, объявленной в свойстве `animation-timing-function`. Значением `ease-out` определяется такое же поведение мяча при прохождении верхней точки своей траектории (см. пример на сайте книги; файл *ball1.html*).

```
.ball {
   animation-name: bounce;
   animation-duration: 1s;
   animation-iteration-count: infinite;
   animation-timing-function: ease-in;
   animation-direction: alternate;
}
```

```
@keyframes bounce {
   0% {
      transform: translateY(0);
   }
   100% {
      transform: translateY(500px);
   }
}
```

Шаговая временная функция

Шаговые временные функции — `step-start`, `step-end` и `steps` — не описываются функциями Безье и даже не представляются кривыми. Все они определяют скачкообразное изменение значений анимируемых свойств. Последняя функция применяется чаще остальных, так как лучше всего подходит для анимации персонажей и спрайтовой анимации.

Временная функция `steps()` представляет анимацию в виде набора последовательно сменяющих друг друга равноудаленных состояний. Она имеет всего два аргумента: количество шагов и одну или несколько точек изменения состояния.

Количество шагов ступенчатой анимации устанавливается первым аргументом шаговой временной функции, представляемым исключительно положительным целочисленным значением. Например, односекундная анимация, выполняемая за 5 шагов, предполагает, что каждый шаг будет длиться 200 мс. Таким образом, элемент будет перерисовываться пять раз в секунду через интервалы 200 мс, обеспечивая 20%-ное изменение значения свойства на каждом шаге.

Чтобы понять, как выполняется шаговая анимация, возьмите в руки книгу с картинками типа флипбук (flipbook) и быстро пролистайте ее. На каждой из страниц книги содержится определенный рисунок, сменяющийся рисунком следующей страницы, подобно кадрам кинопленки. Если изменения в рисунках незначительные, а сами они сменяются очень быстро, то анимация будет выглядеть плавной, как в мультфильме. Для получения такого рода анимации достаточно воспользоваться спрайтами, свойством `background-position` и временной функцией `step`.

На рис. 18.4 показан спрайт, содержащий изображения, быстрая смена которых в документе приводит к созданию анимационного изображения, подобного получаемому при быстром пролистывании страниц флипбука.

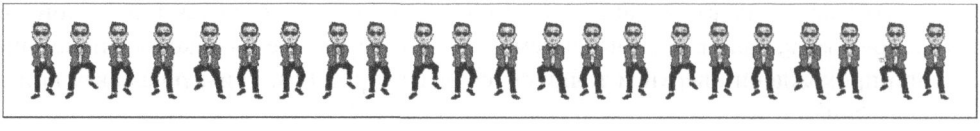


Рис. 18.4. Спрайт танцующего человечка

Здесь под *спрайтом* подразумевается коллекция всех видоизмененных изображений, участвующих в анимации. Следовательно, каждое из изображений спрайта представляет отдельный кадр анимации.

Впоследствии спрайт назначается в качестве фонового изображения специально созданному контейнеру элемента, размер которого в точности совпадает с размером

отдельного изображения (кадра) спрайта. Визуализация кадров (изображений спрайта) выполняется в результате анимации свойства `background-position` с помощью временной функции `steps()`. В данном случае количество шагов анимации повторяет число изображений, включенных в спрайт. Несложно подсчитать, что число шагов анимации определяет количество остановок, которые совершаются в процессе "прокрутки" фонового изображения в контейнере элемента.

Спрайт, показанный на рис. 18.4, содержит 22 изображения размером 56×100 пикселей. Общий размер спрайта равен 1232×100 пикселей. При использовании спрайта в качестве фона элемента его исходное положение описывается свойством `background-position` со значением `top left`, равнозначным значению `0 0`. Такое исходное положение фонового изображения более чем оправданно: в браузерах, не поддерживающих CSS-анимацию, фон элемента будет представляться первым изображением спрайта.

```
.dancer {
   height: 100px;
   width: 56px;
   background-image: url(../images/dancer.png);
....
}
```

Ключевой момент в спрайтовой анимации заключается в применении функции `steps()` для изменения значения свойства `background-position`, что приводит к одновременной визуализации в контейнере элемента только отдельной части спрайта. В данном случае фоновое изображение не прокручивается плавно, а выводится скачкообразно за указанное количество шагов.

Таким образом, наша спрайтовая анимация сводится к шаговому смещению спрайта, большая часть которого находится вне области просмотра, влево. Так как его ширина равна 1232 пикселям, анимация элемента будет заключаться в изменении его свойства `background-position` со значения `0 0` на `0 -1232px`. При этом область просмотра задается элементом `div` размером всего-навсего 56×100 пикселей.

Передача свойству `background-position` значения `0 -1232px` приводит к смещению спрайта в крайнее левое положение и полному удалению его из области просмотра. В отсутствие повторения вдоль оси X такая настройка соответствует лишению элемента `div` фонового изображения (ключевой кадр `100%`). Всячески избегайте использования свойства `background-repeat` при настройке анимации.

В конечном виде рассмотренная выше спрайтовая анимация устанавливается следующим кодом.

```
@keyframes dance_in_place {
   from {
      background-position: 0 0;
   }
   to {
      background-position: -1232px 0;
   }
}
```

```
.dancer {
   ....
   background-image: url(../images/dancer.png);
   animation-name: dance_in_place;
   animation-duration: 4s;
   animation-timing-function: steps(22, end);
   animation-iteration-count: infinite;
}
```

Данный пример призван показать, как с помощью простейших методов можно добавить в документ весьма сложную анимацию. Для ее создания достаточно пошагово смещать спрайт изображений, добавленный в одном из направлений. В данном случае анимация выполняется за счет смещения фонового изображения элемента (см. пример на сайте книги; файл *sprite.html*).

Как известно, первый параметр шаговой временной функции устанавливает количество шагов анимации, а второй аргумент представляется ключевым словом `start` или `end`. Он указывает время внесения изменений в анимируемое свойство: в начале или в конце временного интервала шага. По умолчанию применяется значение `end`, поэтому свойство `background-position` будет изменять свое значение в конце каждого следующего интервала длительностью 200 мс. При этом первое изменение будет осуществляться только по истечении первых 200 мс анимации. Если представить второй аргумент шаговой временной функции значением `start`, то изменение анимируемого свойства будет выполняться в начале временного интервала каждого шага. Таким образом, анимация будет визуально проявляться сразу же, с момента ее непосредственного начала. Отличия в значениях `start` и `end` наглядно проиллюстрированы на рис. 18.5.

```
@keyframes grayfade {
   from {background-color: #BBB;}
   to {background-color: #333;}
}

.quickfader {animation: grayfade 1s steps(5,start) forwards;}
.slowfader {animation: grayfade 1s steps(5,end) forwards;}
```

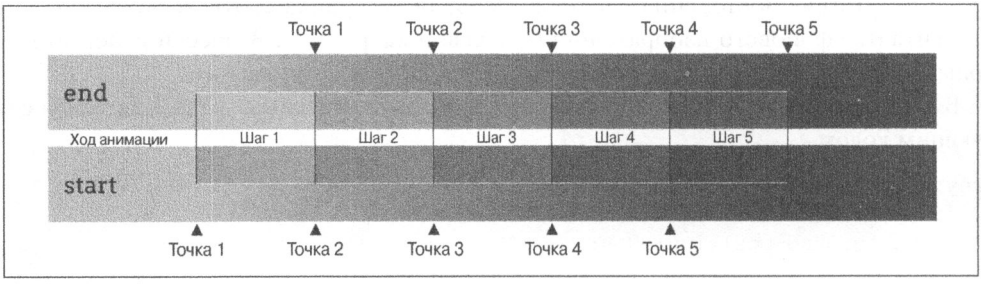


Рис. 18.5. Визуализация начальной и конечной точек изменения фона

Цвет фона, назначаемый элементу на каждом шаге анимации, показан через заливку областей столбчатой диаграммы. Как видите, в случае применения значения

end цвет фона для каждого шага выбирается в его начальной временной точке. Такое поведение связано с тем, что первое изменение цвета осуществляется только в конце первого (начале второго) шага.

Для ключевого слова start наблюдается совершенно иная ситуация: первое изменение цвета выполняется в начале первого интервала, обусловливая установку фона элемента в начале, а не в конце каждого следующего шага анимации. В результате изменение цвета фона выполняется "наперед" — в варианте end фоновый цвет на втором шаге анимации соответствует таковому на первом шаге варианта start.

Такое же поведение свойственно последнему этапу анимации. В варианте start фоновый цвет пятого шага устанавливается в его начале и в полной мере соответствует конечному фоновому цвету элемента. Фоновый цвет пятого шага в варианте end определяется оттенком, заданным в конце четвертого шага, а конечный фоновый цвет элемент приобретает в самом конце анимации.

Зачастую подбор значения аргумента, определяющего место изменения анимируемого свойства, вызывает непреодолимые трудности. Для более четкого определения назначения каждого из ключевых слов достаточно понимать, что в варианте start анимация лишается начального (0%) ключевого кадра, поскольку первое изменение цвета выполняется в самом начале анимации, а в варианте end она лишается конечного (100%) ключевого кадра.

Чтобы предотвратить сброс фонового цвета в исходное значение по окончании анимации, ее настройку нужно завершить объявлением ключевого слова forwards, детально описанного в разделе "Режимы анимации".

Значение step-start, как и временная функция steps(1, start), определяет одношаговую анимацию с единственным ключевым кадром, устанавливаемым во временной точке 100%. Подобным образом значение step-end полностью равнозначно функции steps(1, end) и определяет одношаговую анимацию с единственным ключевым кадром, устанавливаемым во временной точке 0%.

Применение еще одной анимации

Вернувшись к примеру спрайтовой анимации, давайте чуть усложним танцевальные движения нашего персонажа, снабдив его возможностью поворота влево и вправо. Для этого нам понадобится добавить к элементу еще один тип анимации.

```
@keyframes move_around {
   0%, 100% {
      transform: translate(0, -40px) scale(0.9);
   }
   25% {
      transform: translate(40px, 0) scale(1);
   }
   50% {
      transform: translate(0, 40px) scale(1.1);
   }
```

```
    75% {
        transform: translate(-40px, 0) scale(1);
    }
}
```

В приведенном выше коде создается анимация `move_around`, которая назначается элементу наряду с уже имеющейся: ее название указывается в списке анимаций вторым и отделяется от имени первой анимации запятой (см. пример на сайте книги; файл *sprite2.html*).

```
.dancer {
    ....
    background-image: url(../images/dancer.png);
    animation-name: dance_in_place, move_around;
    animation-duration: 4s, 16s;
    animation-timing-function: steps(22, end), steps(5, end);
    animation-iteration-count: infinite;
}
```

Обратите внимание на то, что двумя значениями теперь снабжаются все свойства настройки анимации, за исключением `animation-iteration-count`. Как вы помните, отсутствие у свойства настройки анимации такого же количества значений, как у свойства `animation-name`, предполагает их дублирование до необходимого числа. Так как бесконечно долго должны выполняться оба типа анимации, ключевое `infinite` достаточно указать всего один раз: оно будет применено к каждой именованной анимации списка. Браузер самостоятельно увеличит число экземпляров значения свойства `animation-iteration-count` (в данном случае `infinite`) до количества элементов в списке названий анимаций.

Анимация временной функции

Из предыдущих разделов известно, что свойство `animation-timing-function` не является анимируемым, хотя и включается в объявления ключевых кадров анимации с целью изменения скорости ее хода.

Кроме того, свойство `animation-timing-function`, в отличие от анимируемых свойств, не интерполируется во времени. При добавлении в блок команды `@keyframes` она изменяет свое значение только при обработке каждого следующего объявленного в правиле ключевого кадра, как показано на рис. 18.6.

```
@keyframes width {
    0% {
        width: 200px;
        animation-timing-function: linear;
    }
    50% {
        width: 350px;
        animation-timing-function: ease-in;
    }
    100% {
        width: 500px;
    }
}
```

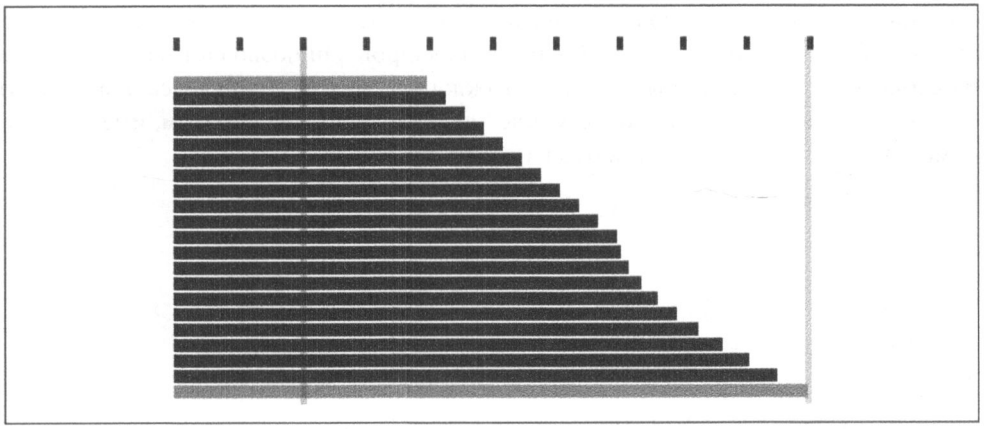


Рис. 18.6. Изменение временной функции в течение анимации

В рассматриваемом примере линейная анимация свойства `width` сменяется анимацией, выполняемой согласно временной функции `ease-in`. Изменение временной функции анимации начинается с ключевого кадра, в котором это изменение объявлено (см. пример на сайте книги; файл *cubicbezierprint2.html*).

На эффект анимации не оказывает влияние только значение свойства `animation-timing-function`, указываемое для кадра `to` или `100%`. Во всех остальных случаях изменение скорости анимации осуществляется с ключевого кадра, в котором этому свойству передается значение, отличное от заданного по умолчанию, или предыдущее значение.

Объявление свойства `animation-timing-function` в ключевом кадре вызывает изменение скорости анимации только свойств, указанных в этом ключевом кадре. Следовательно, заданная в текущем ключевом кадре временная функция будет оставаться действительной для указанных в нем свойств до тех пор, пока не будет изменена для них в одном из последующих ключевых кадров или сброшена до значения по умолчанию для данного элемента. В качестве примера рассмотрим следующую именованную анимацию `W`.

```
@keyframes W {
   from {
      left: 0;
      top: 0;
   }
   25%, 75% {
      top: 100%;
   }
   50% {
      top: 50%;
   }
   to {
      left: 100%;
      top: 0;
   }
}
```

Концептуально такой подход обеспечивает независимость анимации свойств, упоминаемых в блоках объявлений ключевых кадров. Он позволяет изменять значения свойств так, как если бы анимация каждого из них настраивалась для элемента или псевдоэлемента в отдельной команде `@keyframes`. Таким образом, именованную анимацию `W` можно условно разбить на две части: `W_part1` и `W_part2`.

```
@keyframes W_part1 {
   from, to {
      top: 0;
   }
   25%, 75% {
      top: 100%;
   }
   50% {
      top: 50%;
   }
}

@keyframes W_part2 {
   from {
      left: 0;
   }
   to {
      left: 100%;
   }
}
```

При таком подходе свойство `animation-timing-function` объявляется только для ключевых кадров, в которых возникает потребность в изменении временной функции.

```
@keyframes W {
   from {
      left: 0;
      top: 0;
   }
   25%, 75% {
      top: 100%;
   }
   50% {
      animation-timing-function: ease-in;
      top: 50%;
   }
   to {
      left: 100%;
      top: 0;
   }
}
```

Здесь свойство `animation-timing-function` устанавливается в значение `ease-in` только для свойства `top` анимации `W_part1`, но не для свойства `left`, и остается действительным от середины анимации до временной отметки `75%`.

Тем не менее в следующем примере включение свойства `animation-timing-function` в блок свойств анимации никак не скажется на ее воспроизведении, поскольку оно объявляется в блоке, лишенном определения каких бы то ни было анимируемых свойств.

```
@keyframes W {
   from {
      left: 0;
      top: 0;
   }
   25%, 75% {
      top: 100%;
   }
   50% {
      animation-timing-function: ease-in;
   }
   50% {
      top: 50%;
   }
   to {
      left: 100%;
      top: 0;
   }
}
```

Но зачем может понадобиться изменять временную функцию в середине анимации? По самым разным причинам. Например, анимация прыгающего мяча строилась на самой простой модели, в которой не учитывалась сила сопротивления воздуха. В ней мяч отскакивает от твердой поверхности бесконечно долго, ускоряясь при падении вниз и замедляясь при отскоке вверх, что обеспечивается инвертированием временной функции с `ease-in` на `ease-out` при изменении направления анимации с `normal` на `reverse` в каждой следующей итерации.

В реальных условиях сила сопротивления воздуха оказывает существенное влияние на траекторию движения мяча. Чтобы добиться максимально естественной анимации мяча, необходимо учитывать уменьшение его момента инерции при каждом следующем отскоке. Воссоздать такой эффект проще всего, задав все типы движений мяча в единственной анимации, включая переход от временной функции `ease-in` к `ease-out` и наоборот в самой верхней и самой нижней точках траектории.

```
@keyframes bounce {
   0% {
      transform: translateY(0);
      animation-timing-function: ease-in;
   }
   30% {
      transform: translateY(100px);
      animation-timing-function: ease-in;
   }
   58% {
```

```
      transform: translateY(200px);
      animation-timing-function: ease-in;
   }
   80% {
      transform: translateY(300px);
      animation-timing-function: ease-in;
   }
   95% {
      transform: translateY(360px);
      animation-timing-function: ease-in;
   }
   15%, 45%, 71%, 89%, 100% {
      transform: translateY(380px);
      animation-timing-function: ease-out;
   }
}
```

В такой анимации с каждым следующим отскоком мяч подпрыгивает на все меньшую высоту и в конце концов застывает на поверхности (см. пример на сайте книги; файл *ball3.html*).

Все необходимые действия выполняются в единственной итерации, поэтому инвертирование временной функции нельзя возлагать на свойство `animation-direction` — его приходится выполнять вручную. Кроме того, нужно уменьшить высоту каждого следующего отскока мяча путем более раннего обнуления скорости в самой верхней части траектории и уменьшения скорости в наиболее низкой ее части. Исходя из указанных предположений, временная функция указывается в явном виде для каждого ключевого кадра, объявленного в блоке настройки анимации. При прохождении мячом верхней точки траектории функция изменяется на `ease-in`, а после отскока определяется как `ease-out`.

Приостановка анимации

Если вам нужно приостановить анимацию или возобновить ее воспроизведение из ранее приостановленного состояния, воспользуйтесь свойством `animation-play-state`.

animation-play-state

Значение	[`running` | `paused`]#
Начальное значение	`running`
Применяется	Все элементы и псевдоэлементы `:before` и `:after`
Вычисляется	Согласно определению
Наследуется	Нет
Анимируется	Нет

Передача этому свойству значения `running`, заданного по умолчанию, обеспечивает нормальный ход анимации. Ключевое слово `paused` указывает на необходимость

приостановки анимации. Приостановка анимации не вызывает ее отмену для элемента, а лишь прекращает ее воспроизведение в месте, заданном свойством animation-play-state. В результате приостановки анимируемые свойства получают промежуточные значения, вычисленные для ключевого кадра, в котором эта остановка осуществляется. Продолжение анимации выполняется при передаче свойству animation-play-state значения running, как при объявлении в явном виде, так и при назначении по умолчанию. При этом анимация продолжает воспроизводиться не с самого начала, а с момента ее приостановки, определяемого заданным ранее ключевым словом paused.

Применение к анимируемым свойствам, находящимся в режиме задержки, объявления animation-play-state: paused приводит к приостановке временного отсчета задержки, который возобновляется только при последующей активизации режима running (см. пример на сайте книги; файл *ball4.html*).

Режимы анимации

Свойство animation-fill-mode указывает, должны ли анимируемые свойства получать значения, заданные для последнего ключевого кадра, по завершении анимации.

animation-fill-mode

Значение	[none | forwards | backwards | both]#
Начальное значение	none
Применяется	Все элементы и псевдоэлементы :before и :after
Вычисляется	Согласно определению
Наследуется	Нет
Анимируется	Нет

Важность этого свойства становится очевидной в контексте визуализации элемента после завершения его анимации. По умолчанию изменения анимируемых свойств, выполняемые в ходе анимации, отменяются по ее завершении, а сами свойства получают исходные значения, характерные для состояния элемента на начало анимации. В частности, после завершения анимации фона — с красного на синий — элемент получит исходно заданный для него фон: красный.

Необходимо учитывать, что в случае применения свойства animation-delay изменения в анимируемые свойства вносятся не сразу после запуска анимации, а по истечении заданной в нем временной задержки, сопровождаемой возникновением события animationstart.

Свойство animation-fill-mode определяет влияние анимации на стилевое форматирование элемента в моменты времени непосредственно перед возникновением события animationstart или сразу же после генерирования события animationend. Элемент может сбрасываться в состояние, в котором он пребывал в начальном (0%) ключевом кадре анимации, и находиться в нем в течение всего времени задержки (animation-delay) или же приобретать стилевое форматирование, получаемое в момент возникновения события animationend.

По умолчанию свойство animation-fill-mode имеет значение none, предотвращающее изменение значений анимируемых свойств до начала анимации. Объявления, указываемые в ключевом кадре 0% (100% для обратной анимации), применяются к элементу только по истечении задержки, заданной свойством animation-delay, и после возникновения события animationstart.

Значение backwards предписывает устанавливать значения анимируемых свойств, указанные в ключевом кадре 0% или from, сразу же после применения анимации к элементу. Таким образом, настройки анимации, объявленные в ключевом кадре 0% (или 100%, если свойству animation-direction задано значение reversed или reversed-alternate), будут применяться без учета (до истечения) временной задержки, определяемой свойством animation-delay.

Передача свойству animation-fill-mode значения forwards указывает сохранить для элемента стилевое форматирование, приобретаемое в конце анимации, т.е. сразу же по завершении последней итерации анимации (свойство animation-iteration-count) и после возникновения события animationend. Если анимация длится целое число итераций, то по ее завершении элемент получит стилевое форматирование, объявленное в последнем ключевом кадре (100%). В случае обратной анимации ее завершение будет знаменоваться установкой анимируемых свойств в значения, указанные в начальном ключевом кадре (0%).

Ключевое слово both предопределяет стилевое форматирование, выполняемое в обоих режимах: backwards, указывающем изменять значения анимируемых свойств сразу же после их объявления, и forwards, предотвращающем их сброс в исходные значения после завершения анимации (генерирования события animationend). (См. пример на сайте книги; файл *halfiterationforwards.html*.)

Если свойство animation-iteration-count представлено числовым значением с плавающей точкой, то анимация не будет завершаться в точности начальным (0%) или конечным (100%) ключевым кадром. Вместо этого она будет прервана в некоей средней точке последней итерации. Если при этом свойство animation-fill-mode установлено в значение forwards или both, то элемент получает стилевое форматирование, приобретаемое им в момент возникновения события animationend. Например, при передаче свойству animation-iteration-count значения 6.5 при линейной временной функции анимация прекращается ровно на середине — событие animationend возникает при обработке ключевого кадра 50% (независимо от того, объявлялся ли он в блоке настроек анимации). В определенном смысле такая же ситуация возникает при передаче свойству animation-play-state значения pause в этой же временной точке.

```
@keyframes move_me {
   0% {
      transform: translateX(0);
   }
   100% {
      transform: translateX(1000px);
   }
}
```

```
.moved {
   transform: translateX(0);
   animation-name: move_me;
   animation-duration: 10s;
   animation-timing-function: linear;
   animation-iteration-count: 0.6;
   animation-fill-mode: forwards;
}
```

Анимация, представляемая приведенным выше кодом, будет длиться всего 0,6 итерации. При линейном изменении анимируемых свойств в течение 10 секунд анимация прекратится на ключевом кадре `60%`, или через 6 секунд после начала, за которые элемент успеет сместиться вправо на 600 пикселей. Если при этом свойство `animation-fill-mode` будет установлено в значение `forwards` или `both`, то элемент сначала будет смещен вправо на 600 пикселей, а затем зафиксирован в этом новом положении. В отсутствие объявления `animation-fill-mode: forwards` элемент по завершении анимации будет возвращен в исходное, "доанимационное" состояние, определяемое свойством `translateX(0)`.

В браузере Safari 9 и более ранних версиях ключевые слова `forwards` и `both` предписывают применять к элементу стилевое форматирование, свойственное последнему (`100%`) ключевому кадру анимации, независимо от места завершения ее последней итерации (см. пример на сайте книги; файл *moveme.html*). Выполнение предыдущего примера в Safari 9 приводит к перемещению элемента `.moved` вправо на 400 пикселей (вправо на 1000 пикселей относительно места, в котором он пребывал до анимации) сразу же после завершения анимации и сохранению этого положения до дальнейшего стилевого форматирования. При этом совершенно не важно направление выполнения анимации — прямое (`normal`) или обратное (`reverse`), — равно как и ключевой кадр, которым заканчивается последняя итерация: `25%` или `75%`. В Safari 9 и более ранних версиях браузера объявление `animation-fill-mode: forwards` неизменно приводит к установке анимируемых свойств в значения, обозначенные в ключевом кадре `100%`. Такое поведение в полной мере соответствует требованиям предыдущих версий спецификации CSS: их отмена является новшеством, с которым приходится мириться всем разработчикам современных браузеров.

Анимация общего назначения

Настройку анимации элемента можно выполнить в одну строку, отказавшись от использования восьми отдельных свойств в пользу всего одного свойства общего назначения: `animation`. В качестве значения это свойство принимает список разделенных запятыми величин, каждая из которых представляет значение одного из рассмотренных ранее свойств настройки анимации. Чтобы указать для элемента или псевдоэлемента сразу несколько именованных анимаций, их названия также следует передать свойству `animation`, разделив запятыми.

animation	
Значение	[<*animation-name*> || <*animation-duration*> || <*animation-timing-function*> || <*animation-delay*> || <*animation-iteration-count*> || <*animation-direction*> || <*animation-fill-mode*> || <*animation-play-state*>]#
Начальное значение	`0s ease 0s 1 normal none running none`
Применяется	Все элементы и псевдоэлементы `:before` и `:after`
Вычисляется	Согласно определению
Наследуется	Нет
Анимируется	Нет

Свойство общего назначения включает функции всех остальных свойств настройки анимации, в том числе `animation-duration`, `animation-timing-function`, `animation-delay`, `animation-iteration-count`, `animation-direction`, `animation-fill-mode`, `animation-play-state` и `animation-name`. В частности, следующие два фрагмента кода полностью идентичны.

```
#animated {
   animation: 200ms ease-in 50ms 1 normal running forwards slidedown;
}
#animated {
   animation-name: slidedown;
   animation-duration: 200ms;
   animation-timing-function: ease-in;
   animation-delay: 50ms;
   animation-iteration-count: 1;
   animation-fill-mode: forwards;
   animation-direction: normal;
   animation-play-state: running;
}
```

В свойстве общего назначения совсем не обязательно указывать все без исключения настройки анимации — достаточно объявить значения, отличающиеся от задаваемых по умолчанию или устанавливающие исходное состояние элемента, подлежащего анимации. В первой длинной строке приведенного выше кода можно не указывать ключевые слова, представляющие значения по умолчанию сразу трех индивидуальных свойств настройки анимации.

Не забывайте, что любые не указанные в свойстве общего назначения настройки анимации автоматически сбрасываются в значения по умолчанию. Все они приведены ниже.

```
animation-name: none;
animation-duration: 0s;
animation-timing-function: ease;
animation-delay: 0;
animation-iteration-count: 1;
animation-fill-mode: none;
```

```
animation-direction: normal;
animation-play-state: running;
```

Порядок определения настроек анимации в свойстве общего назначения очень важен по двум причинам. Во-первых, в них указываются сразу две временные величины: *<animation-duration>* и *<animation-delay>*. При объявлении их обеих первой всегда указывается длительность анимации, а вторая определяет временную задержку. Во-вторых, принципиальную важность имеет позиция, в которой указывается значение свойства `animation-name`. Например, при использовании значения одного из свойств настройки анимации в качестве ее имени (что не рекомендуется делать, но вполне допустимо) оно задается последним в списке значений свойств общего назначения `animation`. Таким образом, первый экземпляр действительного ключевого слова (например, `ease` или `running`) всегда будет рассматриваться обработчиком как неотъемлемая часть значения соответствующего свойства настройки анимации, а ее название — определяться последним его экземпляром. Учтите, что из всех ключевых слов CSS только `none` считается недопустимым для использования в качестве названия анимации. Рассмотрим такой пример.

```
#failedAnimation {
   animation: paused 2s;
}
```

Приведенный выше код полностью эквивалентен следующему.

```
#failedAnimation {
   animation-name: none;
   animation-duration: 2s;
   animation-delay: 0;
   animation-timing-function: ease;
   animation-iteration-count: 1;
   animation-fill-mode: none;
   animation-direction: normal;
   animation-play-state: paused;
}
```

Значение `paused` относится к допустимым названиям анимации. При беглом анализе может показаться, что свойство общего назначения задает анимацию с названием `paused`, имеющую длительность `2s`, что не соответствует действительности. Исходя из того что ключевые слова, передаваемые свойству общего назначения, сначала проверяются на соответствие допустимым значениям индивидуальных свойств настройки анимации, значение `paused` однозначно сопоставляется со свойством `animation-play-state`, а не свойством `animation-name`.

Проанализируем еще один пример.

```
#anotherFailedAnimation {
   animation: running 2s ease-in-out forwards;
}
```

Предыдущий код можно представить в таком эквивалентном виде.

```
#anotherFailedAnimation {
   animation-name: none;
```

```
    animation-duration: 2s;
    animation-delay: 0s;
    animation-timing-function: ease-in-out;
    animation-iteration-count: 1;
    animation-fill-mode: forwards;
    animation-direction: normal;
    animation-play-state: running;
}
```

Ключевое слово `running` вполне подходит для именования анимации. Тем не менее при первом упоминании браузер рассматривает его как значение свойства `animation-play-state`, а не свойства `animation-name`. Но в отсутствие объявления свойства `animation-name` элемент лишается анимации. Следующий код исправляет этот недочет.

```
#aSuccessfulIfInadvisableAnimation {
    animation: running 2s ease-in-out forwards running;
}
```

Здесь первый экземпляр ключевого слова `running` относится к свойству `animation-play-state`, а второй определяет значение свойства `animation-name`. Как бы там ни было, старайтесь избегать подобных ситуаций. Использование действительных ключевых слов в качестве названия анимации становится причиной потенциальных проблем с обработкой CSS-разметки.

В свете приведенных выше замечаний можно поверить в то, что объявление `animation: 2s 3s 4s;` представляется следующим равнозначным кодом.

```
#invalidName {
    animation-name: 4s;
    animation-duration: 2s;
    animation-delay: 3s;
}
```

Но, как известно из раздела "Имя анимации", название анимации не может начинаться с цифры, а потому имя `4s` в данном контексте оказывается недействительным. Чтобы обеспечить работоспособность анимации, ее нужно объявить как `animation: 2s 3s \4s;`.

Для применения к элементу или псевдоэлементу сразу нескольких типов анимации списки их настроек указываются в свойстве общего назначения через запятую.

```
.snowflake {
    animation: 3s ease-in 200ms 32 forwards falling,
               1.5s linear 200ms 64 spinning;
}
```

Здесь анимация падающей снежинки занимает 96 секунд на каждой итерации, длящейся 3 секунды, в которой она дважды оборачивается вокруг своей оси (см. пример на сайте книги; файл *snowflake.html*). В конце последней итерации снежинка замирает в положении, определяемом настройками последнего ключевого кадра (`100%`) анимации `falling`. В приведенном выше объявлении свойства общего назначения определены шесть из восьми свойств настройки анимации `falling` и пять свойств

настройки анимации `spinning`. Наборы значений свойств анимаций отделяются друг от друга запятой.

Несомненно, было бы намного удобнее видеть имя анимации в первой позиции списка настроек анимации, но благодаря возможности применения в названиях действительных ключевых слов CSS такая практика — далеко не самое удачное решение. Это единственная причина, по которой имя анимации указывается последним в свойстве общего назначения.

Подытожив вышесказанное, можно смело утверждать, что свойство `animation` прекрасно подходит для настройки анимации элементов документа. Достаточно обращать должное внимание на порядок указания в нем временны*х* значений и названий анимации, а также не забывать о том, что все упущенные настройки получают значения по умолчанию. Для большей надежности также лучше отказаться от использования в именах анимации действительных ключевых слов CSS.

Приоритетность и порядок выполнения анимации

К концу 2017 года свойства анимации нарушали принципы каскадирования и приоритетности стилевого форматирования, принятые в CSS, получая приоритет над всеми остальными свойствами.

Приоритетность и ключевое слово `!important`

В общем случае свойство, объявленное с селектором идентификатора `1,0,0`, должно иметь приоритет над свойством с селектором элемента `0,0,1`. Тем не менее при изменении значений таких свойств в процессе анимации по ключевым кадрам их приоритетность рассматривается исключительно в контексте строчного стилевого форматирования.

На текущий момент браузеры, поддерживающие CSS-анимацию, рассматривают любые анимируемые свойства как строчные и снабженные ключевым словом `!important` — как, например, в элементе `<div style="keyframe-property: value !important">`. Но такое поведение полностью противоречит спецификации CSS, согласно которой "анимация приоритетнее регулярного стилевого форматирования, хотя и уступает по приоритету правилам, снабженным ключевым словом `!important`". Это несоответствие, появившееся в конце 2017 г., требует исправления в последующих пакетах обновления браузеров или пересмотра принципов анимации, заложенных в спецификацию CSS.

Чтобы избежать недоразумений, старайтесь не включать ключевое слово `!important` в блок объявления анимации. И хотя его использование не противоречит требованиям спецификации, любые свойства, объявленные с его помощью, могут быть полностью проигнорированы браузером.

Порядок анимации

Одно и то же свойство может подвергаться изменению со стороны сразу нескольких типов анимации, но его конечное значение определяет только последняя из них.

```
#colorchange {
   animation-name: red, green, blue;
   animation-duration: 11s, 9s, 6s;
}
```

В приведенном примере объявлены сразу три анимации по ключевым кадрам — `red`, `green` и `blue`, каждая из которых изменяет свойство `color` до значения, представляемого ее именем. Переданные свойствам `animation-name` и `animation-duration` значения обеспечивают начальный запуск анимации `blue`, выполняемой в течение первых шести секунд, последующий запуск анимации `green`, длящейся 3 с, и дальнейший запуск анимации `red`, воспроизводимой только последние две секунды (см. пример на сайте книги; файл *animationorder.html*). Если начальный ключевой кадр анимации `blue` лишить объявления свойства `color`, то он будет изменяться в соответствии с настройками анимации `green`, `red` или значением свойства `currentColor` (строго в указанном порядке). Это же правило справедливо для анимируемых свойств, не объявляемых в ключевом кадре `100%`.

По умолчанию анимация не затрагивает регулярные свойства, форматирующие элемент до ее применения в документе. Более того, сразу же после завершения анимации все анимируемые свойства возвращаются к исходным значениям (за исключением случаев передачи свойству `animation-fill-mode` значения, отличного от `none`). Добавление к настройкам анимации объявления `animation-fill-mode: both` приводит к назначению свойству `color` элемента постоянного синего цвета, так как анимация `blue` будет замещать объявленную перед ней анимацию `green`, предваряемую в свою очередь анимацией `red` (см. пример на сайте книги; файл *animationorder2.html*).

Итерации анимации и объявление `display: none;`

При установке свойства `display` в значение `none` анимация соответствующего элемента и всех его потомков целиком отменяется. Но она полностью восстанавливается, сбрасываясь в начальное положение, при передаче свойству `display` одного из значений, визуализирующих данные элемента.

```
.snowflake {
   animation: spin 2s linear 5s 20;
}
```

Приведенный выше код представляет анимацию вращения снежинки, совершающей 20 оборотов вокруг своей оси, каждый из которых длится 2 с и снабжается начальной задержкой 5 с. Через 15 с после начала анимации свойство `display` переводится в режим `none`. Перед тем как исчезнуть, снежинка успеет обернуться вокруг своей оси ровно пять раз (5 раз по 2 с плюс начальная задержка 5 с). При последующем изменении свойства `display` до значения, отличного от `none`, анимация запускается с самого начала. Она начинается с пятисекундной задержки и продолжается двадцатикратным вращением снежинки. В данном случае место старта анимации не зависит от того, сколько ее циклов было совершено до отмены с помощью свойства `display` (см. пример на сайте книги; файл *snowflake2.html*).

Анимация и поток пользовательского интерфейса

CSS-анимация имеет *самый низкий* приоритет в потоке пользовательского интерфейса (UI Tread). Нередки ситуации, когда анимация не запускается даже после задержки, устанавливаемой положительным числовым значением свойства `animation-delay`. А все потому, что в потоке пользовательского интерфейса все еще выполняются более приоритетные задачи.

Рассмотрим следующую ситуацию.

- CSS-анимация запускается в потоке пользовательского интерфейса (обработка с помощью CPU а не GPU; см. врезку "Производительность анимации").
- Элемент подвергается 20 типам анимации, каждая из которых запускается с секундной задержкой по отношению к предыдущей анимации. Таким образом, каждая следующая анимация выполняется спустя секунду после старта предыдущей.
- Загрузка документа или приложения с анимацией занимает очень много времени — между загрузкой и последующим выполнением кода JavaScript и визуализацией анимируемого элемента проходит 11 секунд.

Согласно условиям задачи, задержка, объявленная для первых 11 типов анимации, истечет раньше, чем поток пользовательского интерфейса освободится под их выполнение. В результате все они будут выполнены одновременно, а последующие типы анимации, начиная с двенадцатой, будут воспроизводиться с секундной задержкой.

Эпилептические приступы и потеря ориентации при просмотре анимации

Медицинские исследования показали, что динамически изменяемые картинки, к которым относится и CSS-анимация, могут вызывать у отдельных людей эпилептические приступы. Принимайте это к сведению, предоставляя доступ к документам, содержащим анимацию, людям, страдающим эпилепсией и другими нарушениями нервной системы.

Серьезность проблемы вынуждает начинать раздел с предупреждения, что выходит за рамки правил, принятых в технической литературе. Не стоит пренебрегать тем фактом, что динамическое, в особенности быстрое, изменение изображения на экране способно привести к серьезным нарушениям самочувствия у людей, страдающих эпилепсией и имеющих слабый вестибулярный аппарат (склонных к укачиванию).

На момент написания книги во все основные браузеры уже была внедрена поддержка нового медиа-запроса: `prefers-reduced-motion`. С его помощью можно указывать стилевое форматирование, замещающее анимацию и другое динамически изменяемое содержимое. Не пренебрегайте этим простым средством в собственных документах.

```
@media (prefers-reduced-motion) {
   * {animation: none !important; transition: none !important;}
}
```

Этот медиа-запрос блокирует в документе анимацию и переходы, за исключением содержимого, объявленного с использованием ключевого слова `!important` (ранее было рекомендовано отказаться от его использования при настройке анимации). Конечно, такой подход далек от идеального, но годится в качестве первичного решения. Медиа-запрос `prefers-reduced-motion` также можно применять для решения обратной задачи: разблокирования анимации и переходов для пользователей, обладающих крепкой нервной системой и устойчивым вестибулярным аппаратом.

```
@media not (prefers-reduced-motion) {
   /* анимация и переходы */
}
```

Далеко не каждая анимация представляет опасность для людей, склонных к эпилептическим приступам и укачиванию. Некоторые ее виды можно совершенно спокойно выставлять на всеобщее обозрение. В подобных случаях из блока команды `prefers-reduced-motion` нужно исключить анимацию, назначенную ключевым элементам управления пользовательского интерфейса. Используйте эту команду для блокирования анимации только в элементах, отвечающих за графическое оформление документа.

События анимации и вендорные префиксы

Еще раз рассмотрим события, возникающие в процессе выполнения анимации и подлежащие прослушиванию в объектной модели документа, а также вендорные префиксы, которыми могут снабжаться свойства настройки анимации.

Событие `animationstart`

Событие `animationstart` возникает в начале анимации. Если анимация объявлена с использование свойства `animation-delay`, то это событие генерируется сразу же после указанной в нем временной задержки. В отсутствие временной задержки событие `animationstart` регистрируется в момент применения анимации к элементу. Учтите, что оно возникает даже при назначении элементу анимации, лишенной каких-либо действий. При применении к элементу сразу нескольких типов анимации событие `animationstart` будет генерироваться в начале обработки команды `@keyframes` каждой из них. Как правило, каждому идентификатору свойства `animation-name` соответствует собственное событие `animationstart`.

```
#colorchange {
   animation: red, green, blue;
}
```

В приведенном выше примере возникают сразу три события `animationstart` — по крайней мере, до тех пор пока будут действительными все объявленные в нем

анимации: red, green и blue (несмотря на то что каждая из них имеет нулевую длительность).

Если обработка анимации в целевом браузере (обычно Safari версии 8 и ранее, на Android 4.4.4 и более ранней версии) требует добавления вендорного префикса -webkit- в названия свойств ее настройки, то вместо animationstart в начале анимации будет генерироваться событие webkitAnimationStart. Обратите внимание на использование смешанного регистра символов в названии вендорного события. Старайтесь избегать использования вендорных префиксов в коде CSS и прибегайте к ним только в случае крайней необходимости.

Событие animationend

После завершения анимации в документе возникает событие animationend. Для каждой отдельной анимации оно генерируется всего один раз. Если к элементу применены три типа анимации (как в примере #colorchange выше), то и событие animationend для него будет генерироваться трижды, по завершении каждого из типов анимации. Это событие будет действительным даже в анимации с нулевой длительностью. В общем случае время возникновения события animationend рассчитывается по такой формуле:

```
(animation-duration * animation-iteration-count) + animation-delay
```

Событие animationend генерируется, даже если в анимации нет ни одной итерации. Оно не возникает всего в одном случае: при назначении свойству animation-iteration-count значения infinite.

Если свойства анимации снабжены вендорным префиксом -webkit-, то окончание анимации знаменуется событием webkitAnimationEnd, а не animationend.

Событие animationiteration

Это событие возникает в конце каждой итерации анимации, но перед началом каждой следующей ее итерации. Если в анимации нет итераций или их счетчик меньше либо равен единице, то событие animationiteration не возникает вовсе. В случае бесконечного счета итераций (animation-iteration-count: infinite;) событие конца итерации генерируется неисчислимое количество раз, но только при ненулевой длительности анимации.

В отличие от событий animationstart и animationend, возникающих в каждой анимации только единожды, событие animationiteration может генерироваться в одной и той же анимации многократно. Учтите, что это событие возникает между итерациями и не совпадает по времени с событием animationend. Иными словами, количество событий animationiteration на единицу меньше значения свойства animation-iteration-count, представляемого целочисленным значением, до тех пор пока абсолютное значение отрицательной задержки не будет превышать длительности анимации.

Печать анимации

Разумеется, анимацию невозможно отобразить на бумажном листе. Поэтому при выводе документа на печать анимируемый элемент представляется в наиболее уместном виде. Например, если в процессе анимации к элементу применяется объявление `border-radius: 50%`, то и на бумаге он будет отображаться в указанном стилевом форматировании.

Фильтры, смешивание цветов, обтравочные контуры и маски

За последнее десятилетие в спецификацию CSS было включено несколько чрезвычайно необычных стилевых средств, позволяющих авторам документов изменять внешний вид документов с помощью всевозможных фильтров, путем смешения цветов элементов, расположенных на разных слоях, а также маскирования отдельных их частей. Несмотря на отличие принципов, на которых основана работа упомянутых выше инструментов, их объединяет то, что еще совсем недавно выполняемое с их помощью стилевое форматирование казалось просто невозможным.

CSS-фильтры

Ветеранам веб-дизайна должно быть знакомо стилевое свойство `filter`, созданное разработчиками Microsoft и включенное в пакет визуальных эффектов DirectX. По прошествии многих лет спецификация CSS наконец-то пополнилась свойством `filter` собственной разработки, которое, несмотря на концептуальное подобие с одноименным свойством Microsoft, является полностью самостоятельным инструментом с совершенно иными функциональными возможностями. Среди самых заметных отличий — применение к элементу сразу нескольких фильтров и поддержка их загрузки из внешних источников.

filter

Значение	[`none` | `blur()` | `brightness()` | `contrast()` | `drop-shadow()` | `grayscale()` | `hue-rotate()` | `invert()` | `opacity()` | `sepia()` | `saturate()` | `url()`]#
Начальное значение	`none`
Применяется	Все элементы (SVG, растровые изображения и контейнеры, за исключением размеченных тегом `<defs>`)
Вычисляется	Согласно определению
Наследуется	Нет

Синтаксис значения этого свойства предполагает передачу ему списка функций фильтрации, разделенных запятыми. Следовательно, объявление `filter: opacity(0.5) blur(1px);` снабжает элемент полупрозрачностью с последующим размытием. При обратном порядке указания фильтров полностью непрозрачный элемент сначала будет размыт и только после этого получит полупрозрачные цвета.

В описании свойства `filter`, приведенном в спецификации CSS, речь идет исключительно о "входных изображениях" (input images), но в действительности функции фильтрации могут применяться не только к графическим файлам. Под воздействие CSS-фильтров подпадают любые HTML-элементы и SVG-изображения. В данном контексте под *входными изображениями* подразумеваются любые подготовленные к просмотру элементы, представляемые на экране перед применением CSS-фильтров. После применения фильтра такие элементы окончательным образом визуализируются и отображаются на носителе устройства вывода.

Каждое из ключевых слов, включаемых в значение свойства `filter`, представляет отдельную функцию фильтрации, получающую аргументы строго заданного типа. Для того чтобы ознакомиться с ними, их лучше всего рассматривать, условно разделив на несколько категорий.

Базовые фильтры

К базовым относятся фильтры, вносящие в элемент изменения, отображаемые их названиями: `blur` (размытие), `opacity` (непрозрачность), `drop-shadow` (отбрасывание тени).

`blur(`*`<length>`*`)`

Применяет к содержимому элемента гауссово размытие с нормальным (среднеквадратическим) отклонением *`<length>`*. Аргумент `0` предполагает визуализацию элемента в исходном виде. Отрицательные значения недопустимы.

`opacity( [` *`<number>`* `|` *`<percentage>`* `] )`

Представляет содержимое элемента полупрозрачными оттенками подобно тому, как это делает свойство `opacity`. Аргумент `0` соответствует полностью прозрачному элементу, а значение `1` или `100%` представляет его в исходном виде. Отрицательные значения недопустимы, но разрешается указывать аргументы, бо́льшие `1` или `100%`, которые, впрочем, усекаются до указанных величин.

В спецификации оговаривается, что фильтр `filter: opacity()` нельзя заменить равнозначным стилевым свойством `opacity`. В действительности их можно применить к одному и тому же элементу одновременно, получив аддитивный эффект.

`drop-shadow(`*`<length>`*`{2,3}` *`<color>`*`?)`

Создает эффект отбрасывания элементом тени, форма которой определяется альфа-каналом элемента, а цвет и размытие указываются отдельно. В этом фильтре аргументы длины и цвета определяются так же, как и соответствующие значения свойства `box-shadow`: первые два аргумента длины могут представляться отри-

цательными значениями, а третий (указывает глубину размытия) — нет. Если не указать аргумент *<color>*, то цвет тени будет определяться вычисляемым значением свойства `color`.

Результат применения базовых функций фильтрации, как по отдельности, так и в комбинации, приведен на рис. 19.1.

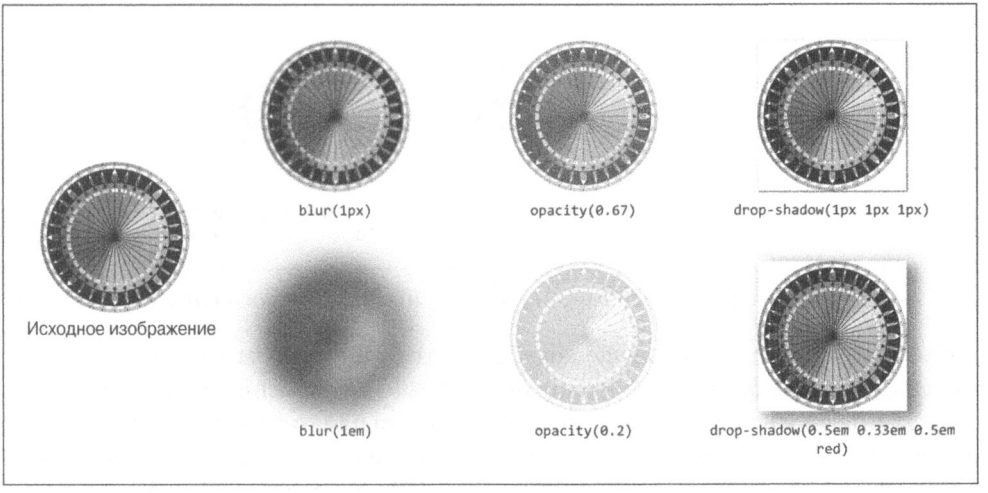


Рис. 19.1. Результат применения базовых фильтров к элементам (см. цветные иллюстрации на веб-сайте)

Перед подробным рассмотрением примеров необходимо прояснить два важных момента. Во-первых, обратите внимание на способ формирования тени у элемента с помощью функции `drop-shadow()`. При просмотре рис. 19.1 может сложиться впечатление, будто тень, имеющая такую же прямоугольную форму, как и элемент, прикрепляется к его контейнеру. В действительности это не так, поскольку тень элемента представляется растровым изображением формата PNG, лишенным альфа-канала. Иными словами, белые области элемента при добавлении к нему тени будут оставаться непрозрачными.

Если тень добавляется к изображению, включающему области непрозрачности, то они будут сохраняться и после применения функции `drop-shadow()`. К таким изображениям относятся файлы формата GIF89a, PNG, JPEG2000, SVG или любых других типов, поддерживающих работу с альфа-каналами. Результат добавления тени к таким изображениям показан на рис. 19.2.

Заметьте, что последнее изображение на рис. 19.2 отбрасывает сразу две тени, создаваемые следующей командой:

```
filter: drop-shadow(0 0 0.5em yellow) drop-shadow(0.5em 0.75em 30px gray);
```

Подобным образом создаются самые произвольные последовательности фильтров. Ниже приведен еще один возможный вариант.

```
filter: blur(3px) drop-shadow(0.5em 0.75em 30px gray) opacity(0.5);
```

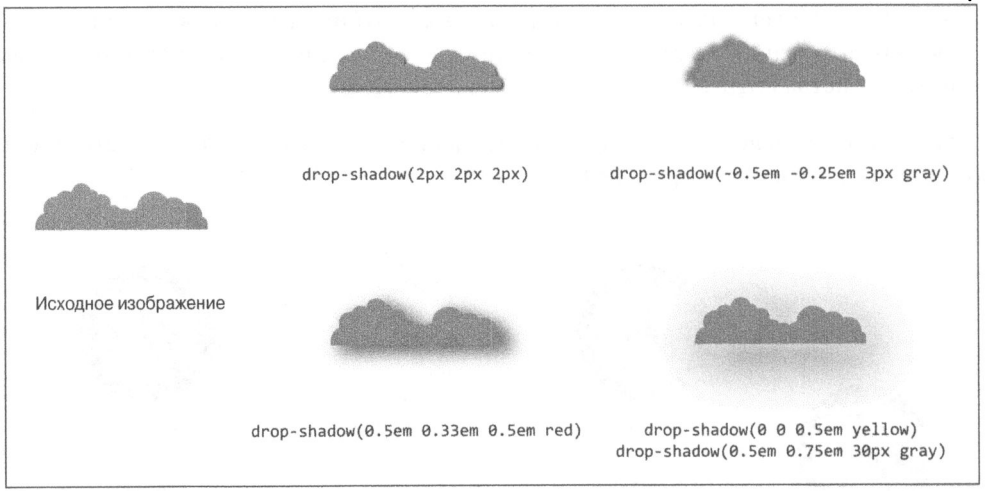


Рис. 19.2. Тени у элементов с альфа-каналом (см. цветные иллюстрации на веб-сайте)

Согласно приведенному выше коду, элемент будет сначала размыт, затем получит тень, а в конце станет наполовину прозрачным. Несомненно, применение такого эффекта к тексту не способствует повышению уровня его удобочитаемости, но тем не менее является вполне допустимым. Создание последовательностей фильтров — одна из ключевых особенностей свойства `filter` как по отношению к базовым функциям, так и всем остальным, описанным далее.

Цветовые фильтры

В эту категорию включены фильтры, которые отвечают за коррекцию цветовой палитры изображения, представляющего элемент в документе. К ним относятся как простые функции исключения отдельных цветов, так и сложные команды смещения сразу всех цветовых оттенков элемента.

Первые три из четырех представленных в этой категории фильтров имеют аргумент *<number>* или *<percentage>*, представляемый неотрицательным числовым значением.

`grayscale( [ <number> | <percentage> ] )`

Приводит цвета элемента к оттенкам серого. Аргумент `0` сохраняет исходную цветовую гамму элемента, а значение `1` или `100%` преобразует в полутона все цветовые оттенки изображения.

`sepia( [ <number> | <percentage> ] )`

Создает эффект сепии или раскрашивания содержимого элемента в цвета, характерные для очень старых черно-белых фотографий. (В Википедии цвет сепии определяется как рыжевато-бурый и представляется шестнадцатеричным значением `#704214` или `rgba(112,66,20)` в цветовом пространстве sRGB.) Аргумент `0` сохраняет исходную цветовую гамму элемента, а значение `1` или `100%` позволяет отобразить в оттенках сепии все содержимое элемента.

`invert( [ <number> | <percentage> ] )`

Инвертирует цветовые оттенки элемента. В модели RGB инвертирование заключается в вычитании каждого из компонентов цвета из значения 255 (для пространства 0–255) или 100% (для пространства 0–100%). В частности, пиксель с цветом `rgb(255,128,55)` будет визуализироваться как `rgb(0,127,200)`, а пиксель с цветом `rgb(75%,57.2%,23%)` — представляться цветовым оттенком `rgb(25%,42.8%,77%)`. Аргумент `0` сохраняет исходную цветовую палитру элемента, а значение `1` или `100%` полностью инвертирует ее. При получении аргумента `0.5` или `50%` инвертирование цветов элемента выполняется с усреднением значения, что равнозначно заливке его однотонным серым цветом.

`hue-rotate( <angle> )`

Смещает тон всех цветовых оттенков изображения элемента на определенный угол в цветовом круге HSL, оставляя неизменными их насыщенность и светлоту. Аргумент `0deg`, как и `360deg` (полный оборот цветового круга), сохраняет тон всех цветовых оттенков элемента. Тем не менее фильтру допускается передавать аргументы, представленные числовыми значениями, большими `360deg`. Отрицательные значения задают смещение тона против часовой стрелки, в то время как положительные аргументы определяют смещение по часовой стрелке. (Иными словами, отсчет угла поворота ведется так же, как и при вычислении отклонения стрелки на циферблате компаса.)

Примеры применения описанных выше цветовых фильтров приведены на рис. 19.3. Учтите, что получаемый эффект сильно зависит от исходной цветовой палитры визуализируемого элемента.

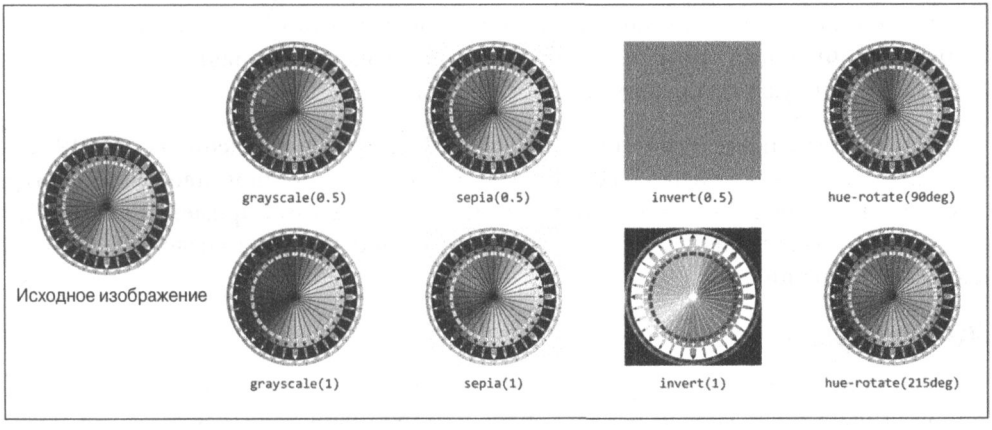


Рис. 19.3. Результат применения цветовых фильтров (см. цветные иллюстрации на веб-сайте)

Яркость, контраст и насыщенность

К этой категории фильтров также относятся команды цветовой коррекции изображения, применяемые на этапе визуализации элемента. Они точно знакомы

каждому, кто увлекается фотографией и постобработкой снимков, хотя для непосвященных могут казаться очень похожими по воспроизводимому эффекту.

Несмотря на возможность передачи всем описанным далее функциям аргументов, больших `1` или `100%`, такие значения будут автоматически ограничиваться предельно возможным числовым значением.

`brightness( [ <number> | <percentage> ] )`

Изменяет яркость изображения элемента. Аргумент `0` делает элемент абсолютно черным, а значение `1` или `100%` обеспечивает неизменность его цветовой палитры. Передача значений, превышающих `100%`, приводит к увеличению яркости изображения, что может стать причиной его "засветки" или представления сплошной белой заливкой.

`contrast( [ <number> | <percentage> ] )`

Корректирует контрастность содержимого элемента. Чем выше контрастность, тем четче различаются цветовые оттенки изображения. Низкая контрастность чревата сведением подобных цветовых оттенков. Аргумент `0` представляет содержимое элемента сплошной серой заливкой, а исходный вид изображения сохраняется при передаче функции значения `1` или `100%`. Аргументы, большие `100%`, соответствуют повышению контрастности выше уровня, свойственного исходному изображению.

`saturate( [ <number> | <percentage> ] )`

Изменяет насыщенность цветов элемента. Элементы с большей насыщенностью представляются цветовой палитрой с более интенсивными оттенками. Аргумент `0` соответствует наименее насыщенным цветовым оттенкам, представленным серыми полутонами, а исходная насыщенность элемента передается аргументом `1` или `100%`. Функция `saturate()` обрабатывает значения, большие `1` или `100%`, повышая цветовую насыщенность изображения.

Примеры использования описанных выше фильтров приведены на рис. 19.4. Не забывайте, что получаемый эффект сильно зависит от исходной цветовой палитры изображения и способа визуализации элемента. Кроме того, применение к изображению сразу нескольких фильтров затрудняет оценку эффекта воздействия на элемент каждого из них.

SVG-фильтры

Последняя категория CSS-фильтров представлена функцией весьма распространенного типа: `url()`. Используя ее, коррекцию цветовой палитры элемента можно выполнять с помощью фильтра, заключенного в SVG-изображение, как встроенное в документ, так и хранящееся во внешнем файле.

В общем виде фильтр определяется функцией вида `url(<uri>)`, где аргумент `<uri>` указывает на фильтр, размеченный в SVG-изображении с помощью специального синтаксиса, в частности, тега `<filter>`. Он может представлять как отдельное SVG-изображение, содержащее единственный фильтр (например, `url(wavy.svg)`),

так и указывать на один из встроенных именованных фильтров (например, `url(filters.svg#wavy)`). Преимущество последнего синтаксиса неоспоримо: он позволяет выделять в отдельный SVG-файл все востребованные в документе фильтры, подключаемые к нему по мере необходимости.

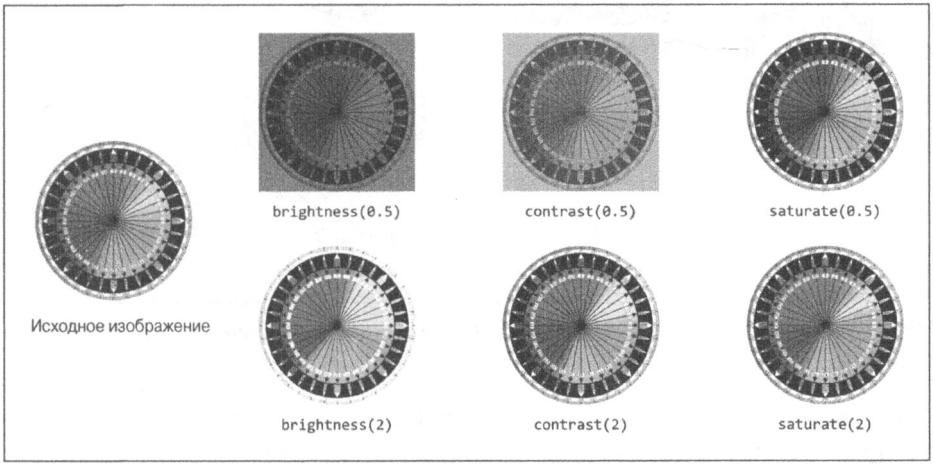


Рис. 19.4. Фильтры коррекции яркости, контраста и насыщенности (см. цветные иллюстрации на веб-сайте)

Если функция `url()` указывает на несуществующий файл или фрагмент SVG-изображения, не являющийся элементом `<flter>`, то обработчик браузера проигнорирует сразу весь фильтр (все объявление свойства `filter`), а не только недействительную его часть.

В задачи книги не входит детальное описание принципов использования SVG-фильтров, хотя они обладают чрезвычайно широкими возможностями по цветовой коррекции изображений. На рис. 19.5 приведены только некоторые примеры использования SVG-фильтров в реальном документе. Для большей наглядности воспроизводимое фильтром действие указывается в подписи к каждому примеру. (Код CSS, отвечающий за применение фильтров, имеет вид, подобный `filter: url(filters.svg#rough)`.)

Полную цветовую коррекцию элементов документа, включая задаваемую описанными выше функциями, можно легко выполнить с помощью одних только SVG-фильтров. (В действительности все фильтры, включенные в спецификацию CSS, обрабатываются как встроенные SVG-фильтры.) Не забывайте о возможности создания последовательностей фильтров. Например, к элементу можно сначала применить внешний SVG-фильтр, а затем воспользоваться специальными функциями, чтобы размыть его или перевести в полутона.

```
img.logo {filter: url(/assets/filters.svg#spotlight);}
img.logo.print {filter: url(/assets/filters.svg#spotlight)
   grayscale(100%);}
img.logo.censored {filter: url(/assets/filters.svg#spotlight)
   blur(3px);}
```

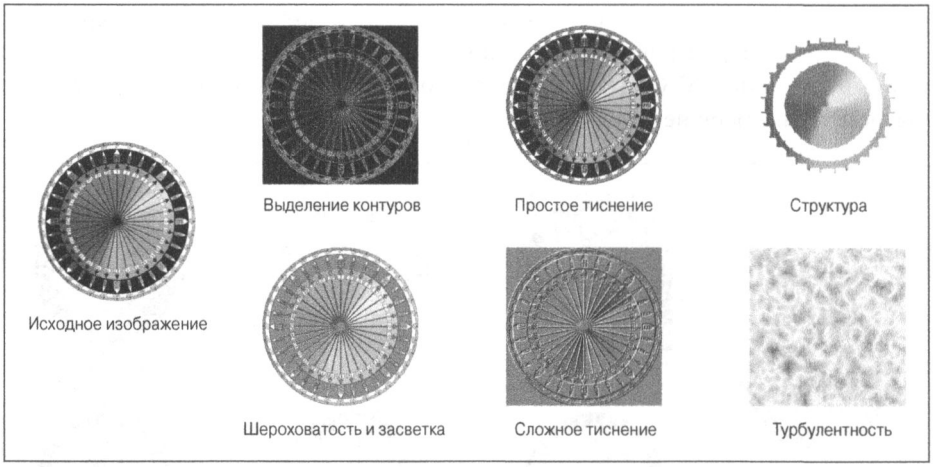


Рис. 19.5. Результаты применения SVG-фильтров (см. цветные иллюстрации на веб-сайте)

Следите за порядком применения фильтров. В приведенном выше примере функции `grayscale()` и `blur()` применяются после фильтра, указанного в функции `url()`. В противном случае элемент будет сначала преобразован в полутоновое изображение или размыт и только после этого обработан SVG-фильтром.

Наложение элементов и смешивание цветов

В дополнение к применению фильтров в спецификации CSS3 существует возможность *смешивания* или *наложения* элементов друг на друга. Представим два элемента, частично перекрывающихся при позиционировании один над другим. В результате такого действия сквозь верхний элемент будет просматриваться определенная часть элемента, расположенного под ним. Такой способ совмещения известен как *простое наложение полупрозрачных элементов* (simple alpha compositing). Содержимое нижнего элемента будет просматриваться через содержимое верхнего только тогда, когда альфа-канал верхнего элемента (или их обоих) содержит значения, меньшие `1`. Ситуация сходна с проступанием фона сквозь полупрозрачный элемент (`opacity: 0.5`) или изображение PNG или GIF87, снабженное прозрачными областями.

Если вам доводилось заниматься коррекцией изображений в таких графических редакторах, как Photoshop или GIMP, то вы обладаете достаточными знаниями для управления наложением слоев и смешиванием цветов в документах. В CSS применяются такие же принципы совмещения графических элементов, как и в обозначенных выше программах. При этом спецификация CSS выделяет сразу два способа наложения элементов (к концу 2017 года): смешивание верхнего элемента со всеми элементами, расположенными под ним, и смешивание фоновых слоев отдельного элемента.

Смешивание цветов элементов

В случае наложения элементов способ смешивания их цветов определяется с помощью стилевого свойства `mix-blend-mode`.

mix-blend-mode	
Значение	`normal` | `multiply` | `screen` | `overlay` | `darken` | `lighten` | `color-dodge` | `color-burn` | `hard-light` | `soft-light` | `difference` | `exclusion` | `hue` | `saturation` | `color` | `luminosity`
Начальное значение	`normal`
Применяется	Все элементы
Вычисляется	Согласно определению
Наследуется	Нет

В спецификации CSS назначение этого свойства определяется как "указывает способ смешивания цветов элемента с расположенным ниже содержимым". Таким образом, оно устанавливает, как будет выглядеть элемент при расположении как поверх других элементов, так и на фоне своего родительского элемента.

Значение `normal`, устанавливаемое по умолчанию, предопределяет неизменность исходного вида элемента — его цвета не смешиваются с цветами расположенного ниже содержимого, за исключением пикселей, представленных в альфа-канале значениями, меньшими `1`. Именно такой метод смешивания цветов лежит в основе упомянутого выше простого наложения полупрозрачных элементов. Несколько примеров наложения элементов с помощью метода, задаваемого по умолчанию, приведено на рис. 19.6.

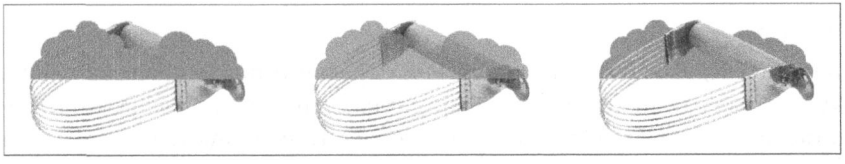


Рис. 19.6. Простое наложение полупрозрачных элементов (см. цветные иллюстрации на веб-сайте)

Остальные режимы наложения, устанавливаемые с помощью свойства `mix-blend-mode`, представлены специальными ключевыми словами. Для более точного их описания необходимо ввести несколько важных определений.

- *Передний план.* Обозначает элемент, к которому применяется свойство `mix-blend-mode`.
- *Задний план.* Представляет содержимое документа, расположенное под целевым элементом. К нему относится как фон родительского элемента, так и содержимое других элементов документа.
- *Цветовые компоненты.* Компоненты, формирующие цвет отдельно рассматриваемого пикселя: R (red — красный), G (green — зеленый) и B (blue — синий).

Проще всего воспринимать передний и задний планы как отдельные слои изображения, открытого в графическом редакторе. Свойство `mix-blend-mode` оказывает влияние на цветовые оттенки элемента (переднего плана), расположенного поверх остального содержимого.

Затемнение, осветление, вычитание и исключение цветов

В описанных в этом разделе режимах наложения цвета пикселей рассчитываются согласно простым математическим формулам: конечный цвет образуется в результате прямого сравнения, сложения или вычитания цветовых компонентов смешиваемого содержимого.

`darken`

Каждый пиксель переднего плана сравнивается с соответствующим пикселем заднего плана. Конечный цвет образуется как совмещение наиболее темных оттенков (представленных наименьшими значениями) каждого из цветовых компонентов R, G и B. Например, наложение пикселя, имеющего цвет `rgb(91,164,22)`, на пиксель `rgb(102,104,255)` в режиме `darken` приведет к образованию цветового пикселя `rgb(91,104,22)`.

`lighten`

Каждый пиксель переднего плана сравнивается с соответствующим пикселем заднего плана. Конечный цвет образуется как совмещение наиболее светлых оттенков (представленных наибольшими значениями) каждого из цветовых компонентов R, G и B. Например, наложение пикселя, имеющего цвет `rgb(91,164,22)`, на пиксель `rgb(102,104,255)` в режиме `lighten` приведет к образованию цветового пикселя `rgb(102,164,255)`.

`difference`

Компоненты R, G и B каждого пикселя переднего плана сравниваются с такими же компонентами соответствующих пикселей заднего плана. Конечный цвет представляется результатом вычитания более темных цветовых компонентов из более светлых. Например, наложение пикселя, имеющего цвет `rgb(91,164,22)`, на пиксель `rgb(102,104,255)` в режиме `difference` приведет к образованию цветового пикселя `rgb(11,60,233)`. Белый слой всегда инвертирует цвет другого слоя, а черный слой никак не воздействует на получаемый результат.

`exclusion`

Слабовыраженная версия метода `difference`. В ней вычисление компонентов конечного цвета происходит не по формуле `|передний план - задний план|`, а согласно такой формуле:

```
передний план + задний план - (2 × задний план × передний план)
```

где `передний план` и `задний план` — цветовые компоненты соответственно переднего и заднего плана, представляемые числовыми значениями в диапазоне от 0 до 1. Например, наложение оранжевого пикселя `rgb(100%,50%,0%)` на серый пиксель `rgb(50%,50%,50%)` в режиме `exclusion` приведет к образованию пикселя `rgb(50%,50%,50%)`. В частности, для красного (R) цветового компонента вычисления проводятся по формуле 1 + 0,5 – (2 × 1 × 0,5), а ее конечный результат — число 0,5 — представляет значение 50%. Формулы расчета зеленого (G) и синего (B) компонентов имеют вид 0,5 + 0,5 – (2 × 0,5 × 0,5) и 0 + 0,5 – (2 × 0 × 0,5) и

приводят к такому же результату, что и предыдущая формула: `0.5`, или `50%`. Цвет, образованный из таких компонентов, отличается от цвета `rgb(50%,0%,50%)`, полученного при смешивании этих же цветов с помощью метода `difference`, заключающегося в точном вычитании одного цветового оттенка из другого.

В последнем примере лучше всего видно, что в формулах вычисления цветовых компонентов применяются значения, нормализованные к диапазону 0–1. Следовательно, конечный результат всех предыдущих примеров также исходно представлялся в нормализованном виде и только затем преобразовывался в более понятный формат. В частности, для получения значения `rgb(11,60,233)` при смешивании цветов в режиме `difference` необходимо провести следующие преобразования.

- Цвет `rgb(91,164,22)` представляется комбинацией компонентов R = 91 ÷ 255 = = 0,357; G = 164 ÷ 255 = 0,643; B = 22 ÷ 255 = 0,086; а цвет `rgb(102,104,255)` — комбинацией компонентов R = 0,4; G = 0,408; B = 1.
- Конечный цвет выражается через цветовые значения, представляемые абсолютными значениями разницы соответствующих цветовых компонентов. Таким образом, R = |0,357 – 0,4| = 0,043; G = |0,643 – 0,408| = 0,235; B = |1 – 0,086| = 0,914. После преобразования в более понятный вид цвет представляется значением `rgb(4.3%,23.5%,91.4%)` или (после умножения каждого компонента на 255) `rgb(11,60,233)`.

Приведенные выше преобразования показывают, почему в описании методов наложения элементов редко приводятся конечные формулы вычисления смешиваемых цветовых оттенков. При необходимости со всеми ними можно ознакомиться в спецификации модуля Compositing and Blending Level 1.

Примеры применения описанных в этом разделе режимов наложения приведены на рис. 19.7.

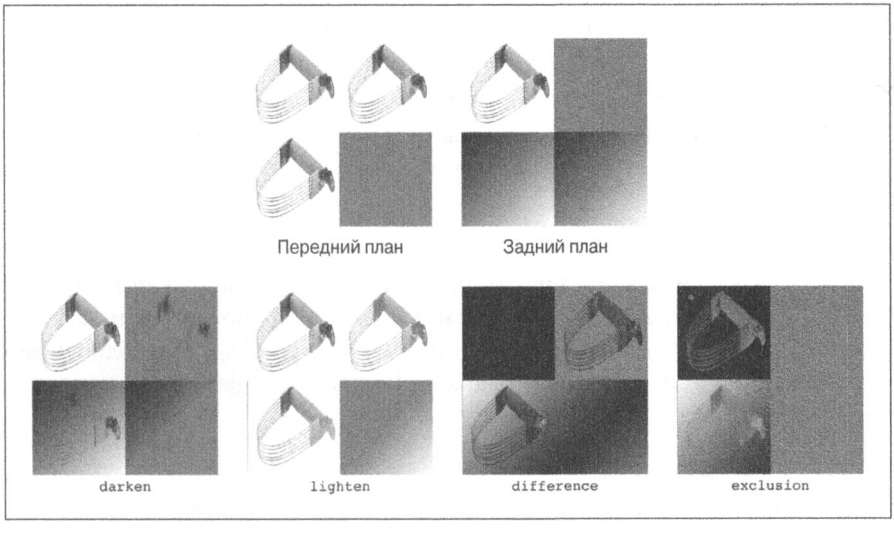


Рис. 19.7. Наложение элементов в режимах затемнения, осветления, вычитания и исключения цветов (см. цветные иллюстрации на веб-сайте)

Умножение, экранирование и перекрытие

Следующие режимы наложения относятся к отдельной категории, поскольку результатом их применения будет умножение цветовых компонентов.

`multiply`

Смешивание цветов достигается за счет перемножения соответствующих цветовых компонентов пикселей переднего и заднего плана. Таким образом, достигается эффект затемнения цветов переднего плана цветовыми оттенками заднего плана. Этот метод наложения полностью симметричен: результат останется неизменным, если поменять элементы заднего и переднего плана местами.

`screen`

Конечный цвет является результатом инверсии цветов (см. функцию `invert()` ранее) переднего и заднего плана, их перемножения и повторного инвертирования результирующего значения. Таким образом, достигается эффект осветления цветов переднего плана цветовыми оттенками заднего плана. Режим `screen`, как и `multiply`, полностью симметричен.

`overlay`

Представляет собой комбинацию методов `multiply` и `screen`. В нем режим `multiply` применяется по отношению к цветовым оттенкам переднего плана темнее `50%` (`0.5`), а ко всем остальным цветам применяется режим `screen`. Таким образом, затемнение изображения переднего плана выполняется только в области темных тонов, а его светлые оттенки осветляются еще больше. Этот режим не является симметричным, поскольку при перемещении заднего плана поверх переднего полностью изменяется конфигурация областей темных и светлых тонов, подлежащих затемнению и осветлению.

Примеры наложения элементов в данных режимах приведены на рис. 19.8.

Жесткий и мягкий свет

В этом разделе описаны два схожих режима наложения, имеющих много общего с методами смешивания цветов, рассмотренными в предыдущем разделе.

`hard-light`

Является инвертированной версией режима `overlay`, представляя комбинацию методов `multiply` и `screen` для содержимого заднего плана. Согласно нему, режим `multiply` применяется к областям заднего плана более темным, чем `50%`, а остальные его области смешиваются в режиме `screen`. Эффект подобен проецированию изображения переднего плана на фоновое изображение с помощью проектора с лампой жесткого света.

`soft-light`

Более мягкая версия предыдущего режима. Аналогичен режиму `hard-light`, но при его использовании создается эффект проецирования изображения переднего плана на фоновое изображение с помощью проектора с лампой рассеянного света.

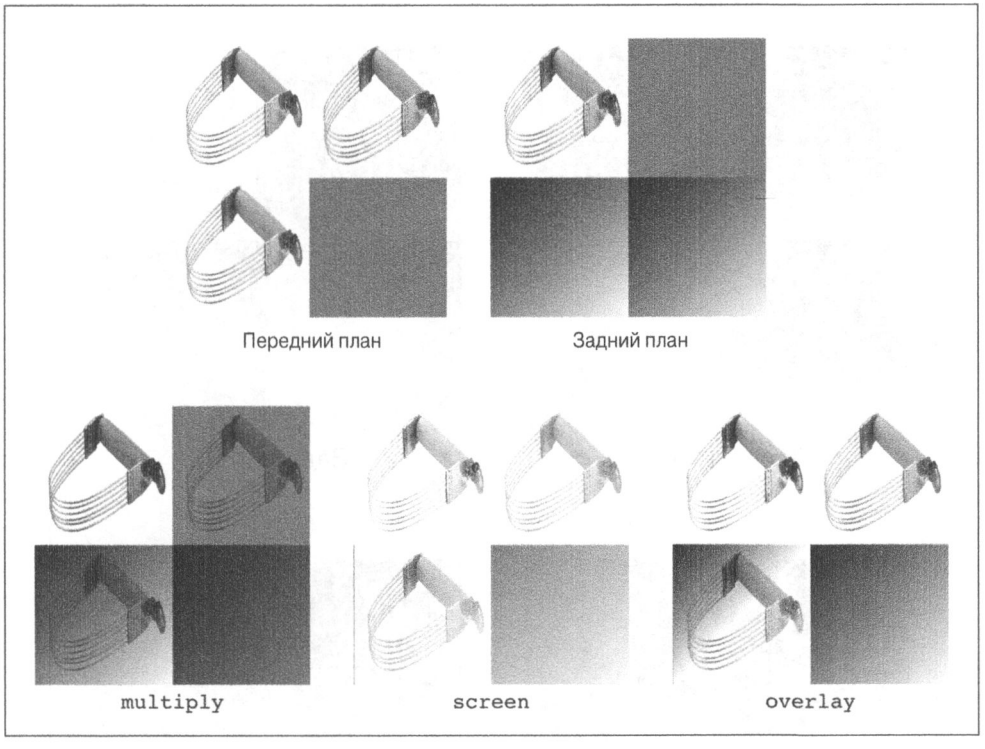


Рис. 19.8. Наложение элементов в режимах умножения, экранирования и перекрытия смешиваемых цветов (см. цветные иллюстрации на веб-сайте)

Результаты применения режимов `hard-light` и `soft-light` показаны на рис. 19.9.

Осветление и затемнение основы

Описанные в этом разделе режимы наложения позволяют осветлить или затемнить передний план, вызывая минимальные изменения в его цветовой палитре. Их названия заимствованы из приемов обработки, применявшихся при проявке снимков в эпоху пленочной фотографии.

`color-dodge`

Конечный цвет получается в результате деления цветовых компонентов заднего плана на соответствующие инвертированные цветовые компоненты переднего плана. Такой подход позволяет осветлить задний план, за исключением абсолютно черных областей.

`color-burn`

Режим, обратный по отношению к `color-dodge`. Конечный цвет получается делением инвертированных цветовых компонентов заднего плана на соответствующие (неинвертированные) цветовые компоненты переднего плана с последующим инвертированием результата. В этом режиме чем темнее цвета заднего плана, тем сильнее их влияние на цвета переднего плана.

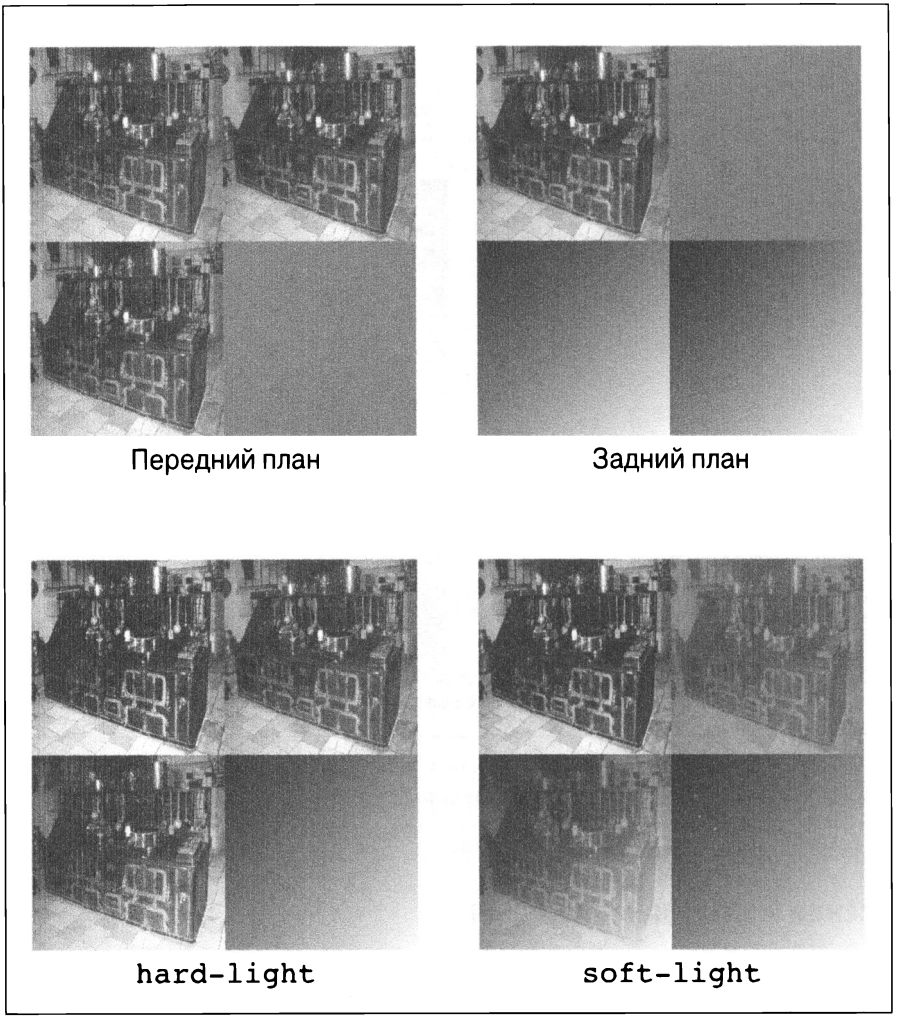


Рис. 19.9. Наложение элементов в режимах жесткого и мягкого света (см. цветные иллюстрации на веб-сайте)

Результаты применения режимов `color-dodge` и `color-burn` показаны на рис. 19.10.

Цветовой тон, насыщенность, цветность и светлота

Последние четыре метода наложения в корне отличаются от рассмотренных выше, поскольку все необходимые вычисления в них выполняются в цветовом пространстве HSL, а не RGB.

`hue`

В этом режиме конечный цвет имеет оттенок пикселя переднего плана, а насыщенность и светлоту заимствует у пикселя заднего плана.

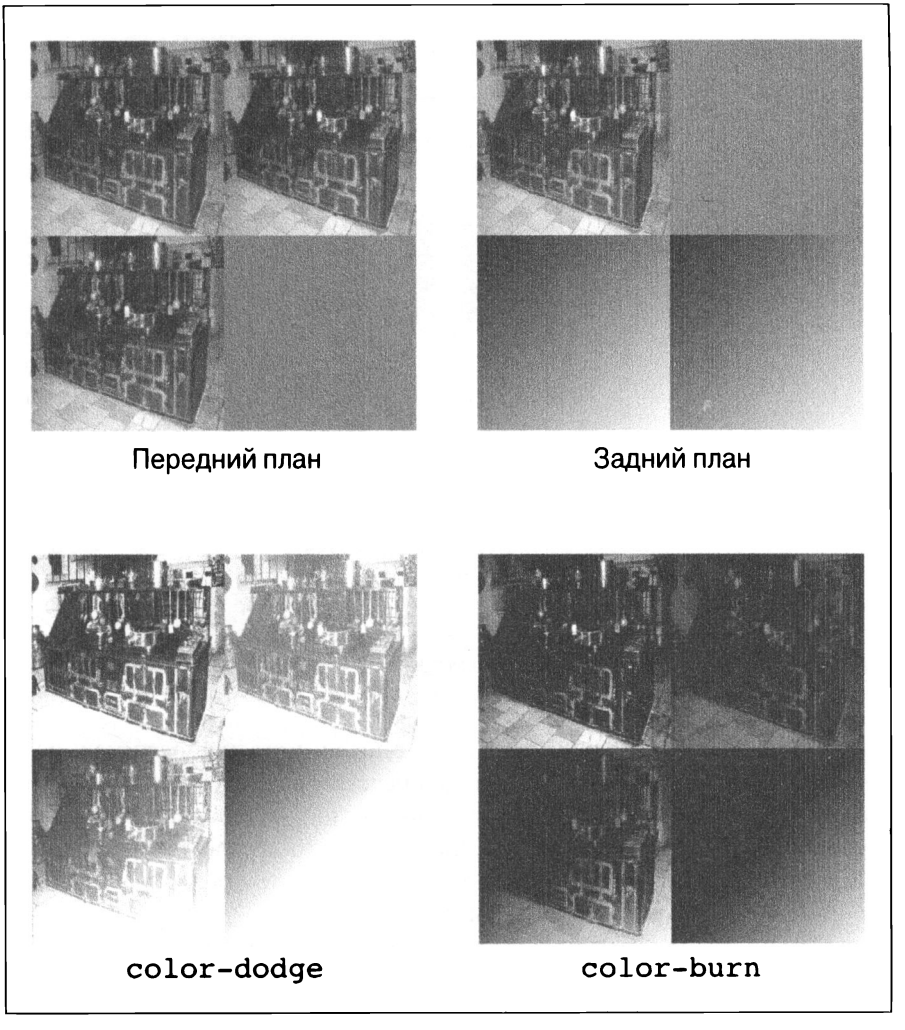


Рис. 19.10. Осветление и затемнение основы (см. цветные иллюстрации на веб-сайте)

`saturation`

Конечный цвет имеет насыщенность пикселя переднего плана, а оттенок и светлоту заимствует у пикселя заднего плана.

`color`

Конечный цвет имеет оттенок и насыщенность пикселя переднего плана, заимствуя у пикселя заднего плана только светлоту.

`luminosity`

Конечный цвет имеет светлоту пикселя переднего плана, а тон и насыщенность заимствует у пикселя заднего плана.

Примеры наложения элементов в режимах, вычисляемых в цветовом пространстве HSL, приведены на рис. 19.11.

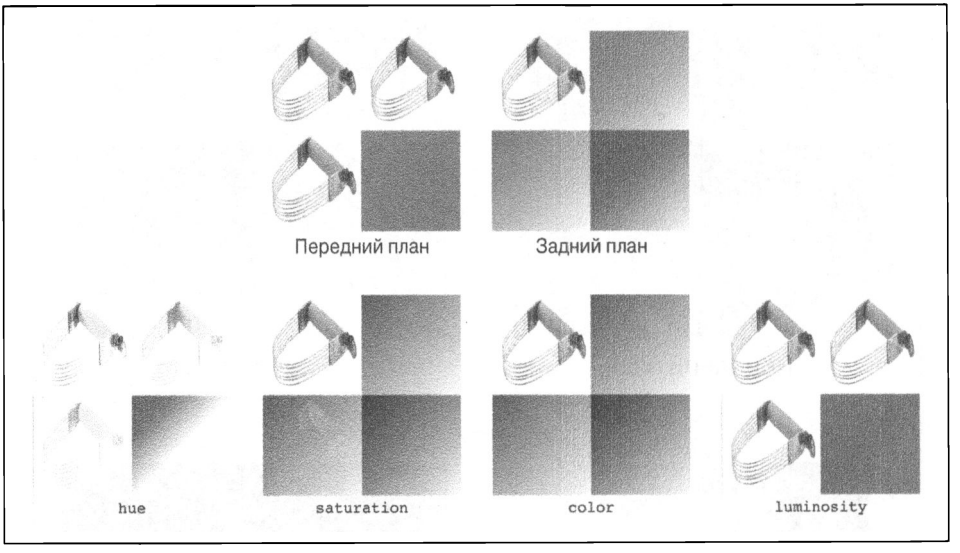


Рис. 19.11. Наложение элементов в режимах цветового тона, насыщенности, цветности и светлоты (см. цветные иллюстрации на веб-сайте)

Спрогнозировать результат наложения в описанных выше режимах, не имея в своем распоряжении расчетных формул, не представляется возможным. Но даже их наличие потребует понимания того, как вычисляются уровни светлоты и насыщенности. Не обладая достаточными познаниями, просто попрактикуйтесь на максимально возможном количестве примеров, чтобы получить представление о принципах смешивания цветов в каждом из HSL-режимов.

При анализе получаемого результата учитывайте следующие два фактора.

- Элемент всегда накладывается на содержимое заднего плана — фон родительского элемента или другие элементы, расположенные под ним.
- Результат наложения сильно зависит от прозрачности накладываемого элемента. Например, наложение элемента в режиме `mix-blend-mode: difference` при непрозрачности `opacity: 0.8` вызовет масштабирование всех значений, участвующих в вычислениях разностей, согласно коэффициенту 80%. Иногда это приводит к сведению цветов к полутонам, а иногда — к заметному смещению цветовой палитры всего изображения.

Наложение фоновых слоев

Смешиванием цветов элемента переднего плана с расположенным под ним фоном дело не ограничивается. Во многих случаях наложению подлежат фоновые слои, которых у отдельно взятого элемента может быть несколько. Способ наложения фоновых элементов определяется с помощью стилевого свойства `background-blend-mode`.

background-blend-mode	
Значение	[normal | multiply | screen | overlay | darken | lighten | color-dodge | color-burn | hard-light | soft-light | difference | exclusion | hue | saturation | color | luminosity]#
Начальное значение	normal
Применяется	Все элементы
Вычисляется	Согласно определению
Наследуется	Нет
Анимируется	Нет

Здесь мы не будем останавливаться на подробном описании всех режимов смешивания цветов, задействуемых при наложении фоновых элементов, поскольку большинство из них было рассмотрено в предыдущем разделе, посвященном определению свойства `mix-blend-mode`. Применительно к фоновым элементам они работают точно так же.

Отличие состоит лишь в том, что смешивание цветов фоновых изображений выполняется без воздействия на задний план — так, как если бы он представлялся полностью бесцветной и прозрачной областью. В действительности свойство `background-blend-mode` просто не воздействует на задний план элемента, а работает только с перекрывающимися фоновыми изображениями.

Чтобы лучше понять, о чем идет речь, проанализируем такой пример.

```
#example {background-image:
      url(star.svg),
      url(diamond.png),
      linear-gradient(135deg, #F00, #AEA);
   background-blend-mode: color-burn, luminosity, darken;}
```

В этом примере рассматриваются три фоновых изображения, для каждого из которых определен собственный режим наложения. Все они, а также результат смешивания их цветов показаны на рис. 19.12.

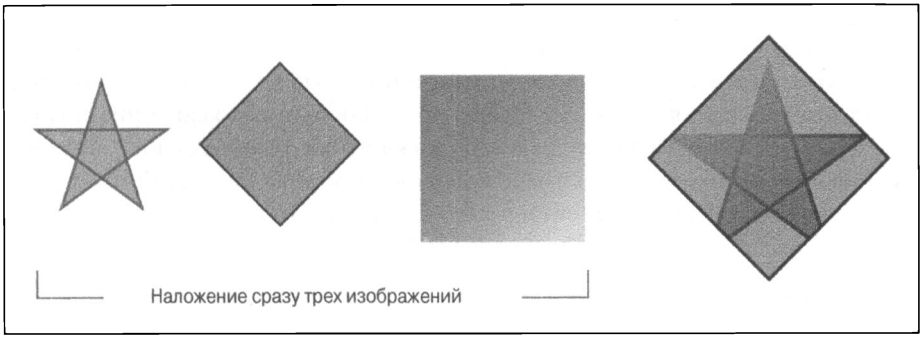


Рис. 19.12. Наложение трех фоновых изображений (см. цветные иллюстрации на веб-сайте)

На данном этапе все предельно понятно: получаемый результат не зависит от содержимого, располагаемого под элементом. Независимо от того, представляется фон родительского элемента сплошной белой заливкой, серым цветом, цветом фуксии или даже повторяющимся градиентом, результат наложения фоновых элемента будет одним и тем же. Наложение в данном случае происходит *изолированно*. На рис. 19.13 показано, как будет выглядеть целевой элемент из примера рис. 19.12 при расположении в родительском элементе с разными вариантами фона.

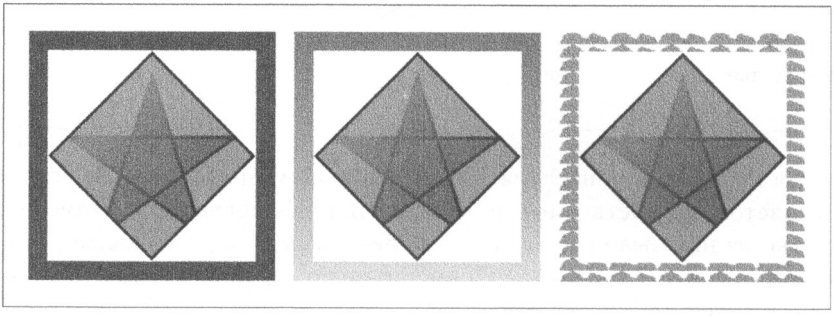


Рис. 19.13. Смешивание цветов на цветном и прозрачном заднем плане (см. цветные иллюстрации на веб-сайте)

Как и содержимое накладываемых друг на друга элементов, фоновые слои совмещаются в порядке от самого нижнего к самому верхнему. Следовательно, при размещении двух фоновых изображений поверх сплошной фоновой заливки сначала смешиваются цвета нижнего фонового слоя и фоновой заливки и только после этого на полученный результат накладывается второй, верхний фоновый слой.

```
#example {background-image:
      url(star.svg),
      url(diamond.png);
   background-color: goldenrod;
   background-mix-mode: color-burn, luminosity;}
```

Согласно такой разметке, сначала изображение `diamond.png` накладывается на именованный фоновый слой `goldenrod` в режиме `luminosity` и только после этого на полученный результат в режиме `color-burn` накладывается изображение `star.png`.

Наряду с тем что наложение фоновых слоев осуществляется независимо от содержимого заднего плана, его результат неизбежно учитывается в операциях, инициированных стилевым свойством `mix-blend-mode`. Такие преобразования предполагают наложение на задний план не отдельно каждого из фоновых слоев, а результата их смешивания. В следующем примере в неизменном виде на задний план накладывается только фон первого элемента, а все остальные элементы смешиваются с ним одним из объявленных в коде способов (рис. 19.14).

```
.one {mix-blend-mode: normal;}
.two {mix-blend-mode: multiply;}
.three {mix-blend-mode: darken;}
.four {mix-blend-mode: luminosity;}
.five {mix-blend-mode: color-dodge;}
```

```
<div class="bbm one"></div>
<div class="bbm two"></div>
<div class="bbm three"></div>
<div class="bbm four"></div>
<div class="bbm five"></div>
```

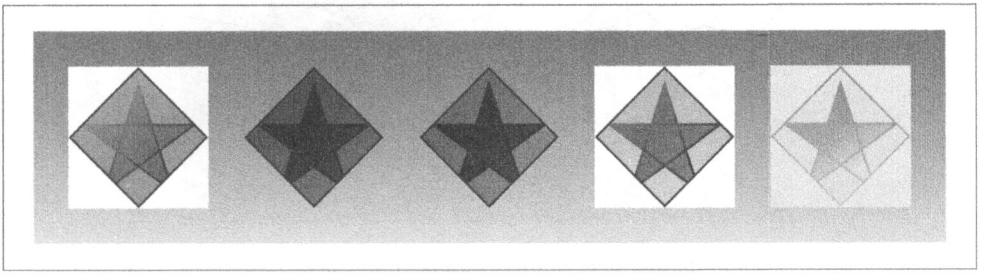


Рис. 19.14. Наложение элементов на задний план (см. цветные иллюстрации на веб-сайте)

Рассмотренная в этом разделе концепция независимого наложения фоновых слоев по умолчанию не применяется к содержимому элементов. Как будет показано далее, смешивание цветов содержимого элементов выполняется по несколько иным принципам. Тем не менее при необходимости эту концепцию можно расширить и на сами элементы.

Изолирование элементов

Во многих ситуациях может понадобиться получить независимое наложение не фоновых слоев, а их основного содержимого. Как вы уже знаете, по умолчанию сами элементы накладывается друг на друга не так, как их фоны. В CSS отдельный стек наложения элементов создается с помощью свойства `isolation`.

isolation

Значение	`auto` | `isolate`
Начальное значение	`auto`
Применяется	Все элементы (SVG, растровые изображения и контейнеры)
Вычисляется	Согласно определению
Наследуется	Нет
Анимируется	Нет

Название свойства достаточно красноречиво говорит о его назначении. С его помощью элемент выделяется или исключается из общего контекста (стека) наложения. В частности, приведенный ниже код обеспечивает результат наложения, продемонстрированный на рис. 19.15.

```
img {mix-blend-mode: difference;}
p.alone {isolation: isolate;}
```

```
<p class="alone"><img src="diamond.png"></p>
<p><img src="diamond.png"></p>
```

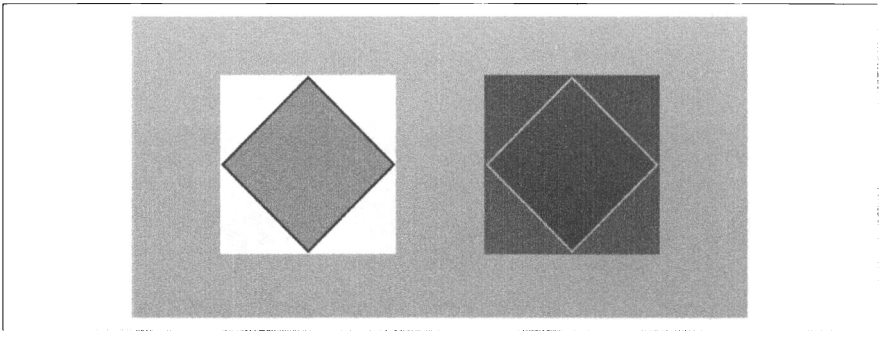


Рис. 19.15. Изолированное и общее наложение элементов (см. цветные иллюстрации на веб-сайте)

Обратите внимание на порядок указания в коде свойств `isolation` и `mix-blend-mode`. Режим наложения указывается для изображения, но в отдельный контекст наложения выделяется содержащий его элемент (в данном случае абзац). Это позволяет исключить из общего порядка наложения элементов документа родительский элемент изображения, что приводит к автоматической отмене для его дочерних элементов режима, устанавливаемого свойством `mix-blend-mode`. Таким образом, для исключения элемента из общего порядка наложения объявление `isolation: isolate` можно применять не к нему самому, а к его родительскому элементу.

Важно понимать, что в отдельный контекст наложения выделяется любой элемент, автоматически исключаемый из общего порядка форматирования независимо от значения свойства `isolation`. Например, исключение элемента из общего контекста форматирования происходит при трансформации его с помощью стилевого свойства `transform`.

К концу 2017 года к отдельному стеку форматирования автоматически относились следующие элементы:

- корневой элемент (`html`);
- абсолютно или относительно позиционируемые элементы, свойство `z-index` которых установлено в значение, отличное от `auto`;
- позиционированные элементы, к которым применено ключевое слово `fixed`, с произвольным значением свойства `z-index`;
- элементы, свойство `opacity` которых установлено в значение, отличное от `1`;
- элементы, свойство `transform` которых установлено в значение, отличное от `none`;
- элементы, свойство `mix-blend-mode` которых установлено в значение, отличное от `normal`;
- элементы, свойство `filter` которых установлено в значение, отличное от `none`;
- элементы, свойство `perspective` которых установлено в значение, отличное от `none`;

- элементы, свойство `mix-blend-mode` которых установлено в значение `isolate`;
- элементы, в которых любое из перечисленных выше свойств оптимизируется в режиме `will-change`, даже если значение свойства при этом не меняется.

Согласно приведенному выше утверждению, наложение группы элементов, размещаемых поверх общего заднего плана, и последующее изменение ее непрозрачности (`opacity`) со значения 1 до 0 приводит к исключению всех таких элементов из общего потока наложения. В зависимости от применяемых режимов наложения такое поведение элементов может не возыметь никакого эффекта, но с большой долей вероятности все же скажется на получаемом результате.

Обтравочные контуры и маскирование

В CSS наряду с фильтрами и режимами наложения к элементам можно применять обтравочные контуры и маски, которые позволяют отображать в документе только отдельные части элементов, ограничивая их простыми формами или фигурами, заимствованными из SVG-файлов. Обычно маски и обтравочные контуры применяются для оформления документов, например для добавления художественных рамок к графическим изображениям или придания им неровных (зазубренных) краев.

Обтравочные контуры

Одна из наиболее важных особенностей свойства `filter` — возможность маскирования элемента SVG-изображением. Это действительно очень полезная функция, но для обрезки элемента по некоему контуру вместо свойства `filter` можно смело использовать стилевое свойство `clip-path`.

clip-path

Значение	`none` | *<url>* | [[`inset()` | `circle()` | `ellipse()` | `polygon()`] || [`border-box` | `padding-box` | `content-box` | `margin-box` | `fill-box` | `stroke-box` | `view-box`]]
Начальное значение	`none`
Применяется	Все элементы (SVG, растровые изображения и контейнеры, за исключением размеченных тегом `<defs>`)
Вычисляется	Согласно определению
Наследуется	Нет
Анимируется	Для `inset()`, `circle()`, `ellipse()` и `polygon()`

Свойство `clip-path` определяет форму, по которой будет обрезаться элемент, отображаемый в документе. Видимая часть элемента находится внутри обтравочного контура, а области, расположенные вне такого контура, визуализации не подлежат (скрываются), но остаются полностью прозрачными для другого содержимого. Ниже приведен пример кода создания обтравочного контура, отвечающего за сокрытие части содержимого абзаца. Результат его применения, а также исходный вид абзаца показаны на рис. 19.16.

```
p {background: orange; color: black; padding: 0.75em;}
p.clipped {clip-path: url(shapes.svg#cloud02);}
```

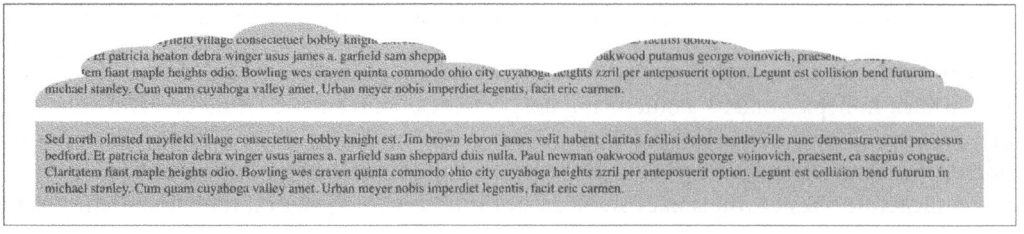


Рис. 19.16. Обрезанный по контуру и исходный абзацы

Значение по умолчанию, `none`, отменяет действие обтравочных контуров, применяемых для маскирования содержимого элемента. При передаче свойству `clip-path` значения *<url>*, как в предыдущем примере, только указывающего на несуществующий файл или SVG-изображение, в котором отсутствует область, помеченная тегом `<clipPath>`, обрезка будет отменена без видимых последствий для элемента.

Остальные значения представляют формы, создаваемые средствами CSS, компоненты блочной модели элемента или оба типа значений сразу.

К концу 2017 года обтравочные контуры, расположение которых указывается значением *<url>*, считались действительными только при объявлении в том же SVG-изображении, что и документ, к которому относится целевой элемент. Обтравочные контуры, объявленные в SVG-изображениях из внешних файлов, считались недействительными.

Обтравочные формы

Обтравочные формы создаются с помощью одной из четырех базовых функций, отвечающих за рисование простых геометрических фигур. Они полностью идентичны функциям, применяемым для создания форм обтекания элемента внешним содержимым (см. главу 10). Детальное их описание содержится в определении свойства `shape-outside`, поэтому далее приведены только краткие сведения о каждой из них.

`inset()`

Получает от одного до четырех процентных или числовых значений, выраженных в единицах длины и устанавливающих величину смещения относительного рамки (границ) элемента. Включение в значение ключевого слова `round` и дополнительных числовых величин (процентных или длины) позволяет определить радиус закругления углов рамки.

`circle()`

Получает единственное значение, представленное величиной, выраженной в единицах измерения длины, процентной величиной или ключевым словом, определяющим радиус круга. Добавление ключевого слова `at` и одного или двух числовых значений (процентных или длины) позволяет указать положение центра круга.

`ellipse()`

Имеет два обязательных аргумента, которые представлены значениями, выраженными в единицах длины, процентными величинами или ключевыми словами, и определяют вертикальный и горизонтальный радиусы эллипса. Добавление ключевого слова `at` и одного или двух числовых значений (процентных или длины) позволяет указать положение центра эллипса.

`polygon()`

Получает список разделенных запятыми пар координат *x* и *y*, которые представлены числовыми значениями, выраженными в единицах длины, или процентными величинами. Может снабжаться префиксом, указывающим способ заполнения многоугольника содержимым.

Примеры применения обтравочных контуров, представленных простыми формами, приведены на рис. 19.17. Все они получены в результате выполнения приведенного ниже CSS-кода. (Пунктирными рамками показаны внешние границы исходных изображений до их обрезки.)

```
.ex01 {clip-path: none;}
.ex02 {clip-path: inset(10px 0 25% 2em);}
.ex03 {clip-path: circle(100px at 50% 50%);}
.ex04 {clip-path: ellipse(100px 50px at 75% 25%);}
.ex05 {clip-path: polygon(50% 0, 100% 50%, 50% 100%, 0 50%);}
.ex06 {clip-path: polygon(0 0, 50px 100px, 150px 5px, 300px 150px,
       0 100%);}
```

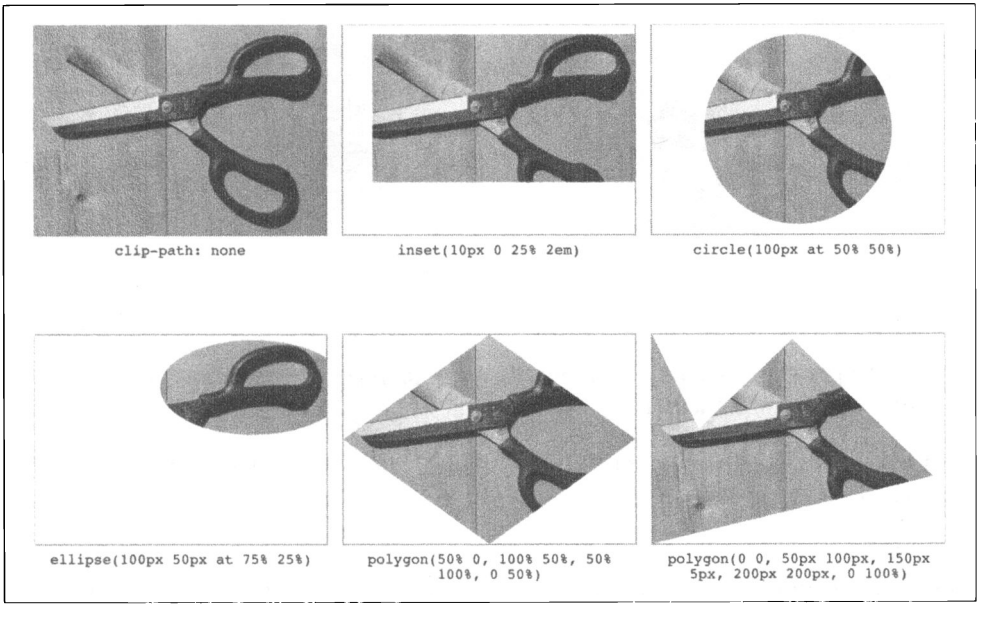


Рис. 19.17. Обрезка изображений по простым геометрическим фигурам

Как видно на рис. 19.17, исходное изображение просматривается только через внутренние области геометрических фигур. Содержимое, находящее вне обтравочных форм, скрывается. Обратите внимание: под обрезаемый по контуру элемент выделяется такое же пространство, как и под исходное изображение, а не область, ограниченная контурами обтравочной формы. Иными словами, обрезка элемента по контуру не приводит к изменению его размера, а всего лишь скрывает отдельные его части.

Обтравочные области блочной модели элемента

В отличие от обтравочных форм, создание обтравочных контуров для отдельных областей блочной модели элемента не требует объявления значений длины или процентных значений. По большей части они определяются контекстом форматирования, представляемым отдельными компонентами блочной модели.

Например, объявление `clip-path: border-box` указывает обрезать элемент по контуру, образованному его границами. В большинстве случаев это наиболее предпочтительный вариант, поскольку отступы элемента и так прозрачные. Не забывайте, что вне границ элемента могут отображаться его внешние контуры, которые в рассматриваемом варианте "обтравки" будут автоматически скрываться.

При специальном объявлении контекст форматирования, границы которого образуют обтравочный контур элемента, определяют значения `margin-box`, `padding-box` и `content-box` — соответственно область отступов, область полей и область содержимого (рис. 19.18).

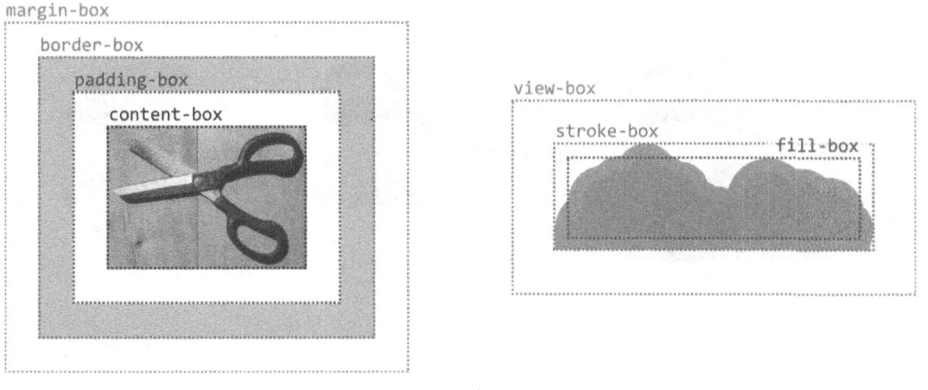


Рис. 19.18. Обтравочные контуры, образованные различными областями блочной модели элемента

На рис. 19.18, *справа*, показаны обтравочные контуры, соответствующие различным областям SVG-изображения.

`view-box`

Обтравочный контур представляется границами (родительской) области просмотра SVG-изображения.

`fill-box`

Обтравочный контур проходит по границам контейнера объекта изображения. Контейнер объекта изображения вмещает все его компоненты, включая образованные в процессе трансформации (например, вращения), но исключая их обводку.

`stroke-box`

Обтравочный контур проходит по границам обводки в контейнере объекта изображения. Контейнер обводки объекта изображения вмещает все компоненты изображения, включая образованные в процессе трансформации (например, вращения) и их обводку.

Последние три значения применяются только к SVG-изображениям, для которых не объявляются обтравочные контуры блочной модели элемента. SVG-изображения, для которых объявлено свойство `clip-path` со значением `margin-box`, `border-box`, `padding-box` или `content-box`, получают обтравочный контур, представляемый значением `fill-box`. И наоборот, если одно из значений `view-box`, `fill-box` или `stroke-box` объявить для регулярного элемента документа, то обтравочный контур будет создаваться согласно значению `border-box`.

Вполне очевидно, что объявления вида `clip-path: content-box` удобно применять для отображения в документе только отдельных компонентов блочной модели элемента — в данном случае области содержимого. Но наиболее действенным оно становится при использовании в сочетании с обтравочными формами. Предположим, что к элементу нужно применить обтравочную форму, представленную значением `ellipse()`, которая должна всего лишь соприкасаться с краями области полей. Вместо вычисления точных размеров эллипса, что неизбежно потребует учитывать ширину полей и границ, достаточно воспользоваться объявлением `clip-path: ellipse(50% 50%) padding-box;`. Оно обязует выровнять эллиптический обтравочный контур по центру элемента, установив его горизонтальный и вертикальный радиусы в значения, равные соответственно половине ширины и высоты элемента (см. главу 10), как показано на рис. 19.19. Здесь же показано, как будет маскироваться элемент при привязке обтравочной формы к областям полей и отступов.

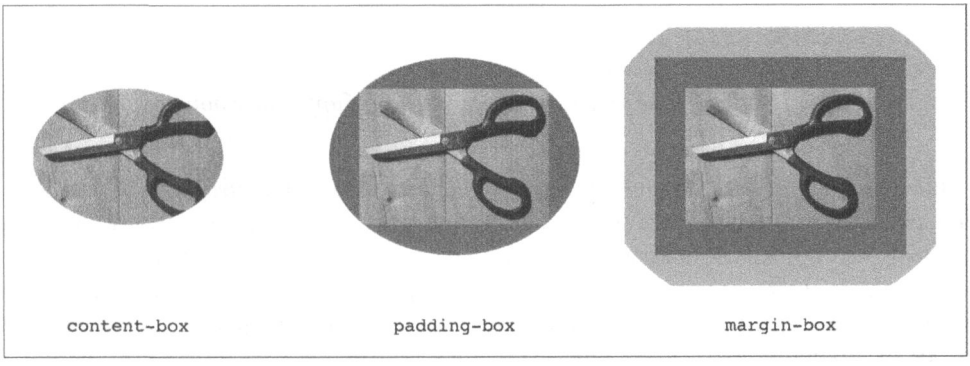


Рис. 19.19. Привязка обтравочной формы к различным компонентам блочной модели элемента

Обратите внимание на способ обрезки изображения по контуру эллипса в случае применения ключевого слова `margin-box`. В показанном на рис. 19.19 примере нет ошибки — отступы всегда прозрачны и остаются невидимыми даже внутри обтравочного контура.

Как ни странно, но ключевые слова, представляющие отдельные части блочной модели элемента, можно объявлять только вместе с обтравочными формами, но *не* обтравочными контурами SVG-изображений. По отношению к SVG-изображениям они применяются только при их обрезке с помощью регулярных методов.

Будьте предельно аккуратны при использовании обтравочных контуров, объявленных в SVG-изображениях: к концу 2017 года их координаты можно было указывать только в абсолютных единицах измерения. Процентные значения, представляющие размеры обтравочных фигур относительно ширины и высоты изображения (как принято в функции `polygon()`), использовать нельзя. Такие методики достижимы только при использовании SVG-атрибута `clipPathUnits`, в том числе и в сочетании с атрибутом `transform`. Ниже приведен один из примеров такого обтравочного контура, результат выполнения которого в браузере показан на рис. 19.20.

```
<clipPath id="hexlike" clipPathUnits="objectBoundingBox">
   <polygon points="0.5 0, 1 0.25, 1 0.75, 0.5 1, 0 0.75, 0 0.25"/>
</clipPath>
```

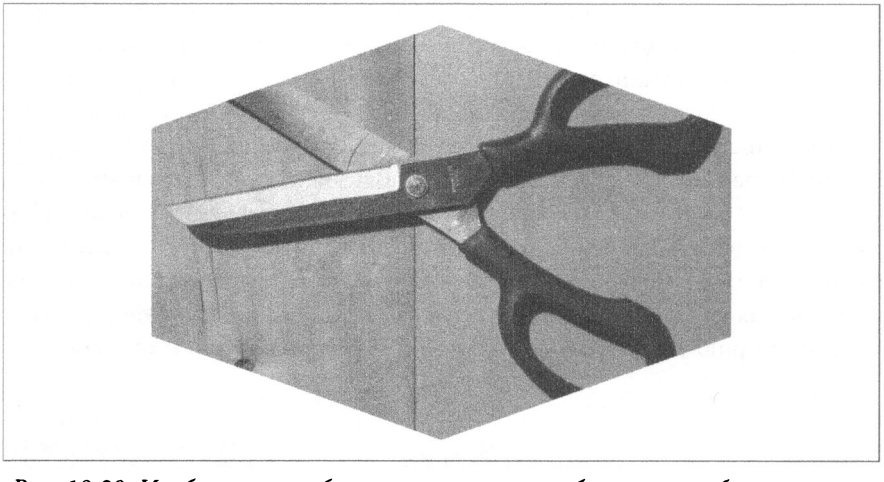


Рис. 19.20. Изображение, обрезанное по масштабируемому обтравочному контуру SVG

Значение `objectBoundingBox` указывает задавать размеры обтравочного контура относительно размеров целевого элемента, представляя их значениями в диапазоне от 0 до 1. Его применение позволяет создавать обтравочные контуры в относительных единицах измерения так, как это делается при рисовании многоугольных фигур с помощью функции `polygon()`. В частности, обтравочную форму, показанную на рис. 19.20, можно получить, воспользовавшись таким кодом:

```
clip-path: polygon(50% 0, 100% 25%, 100% 75%, 50% 100%, 0 75%, 0 25%)
```

Заполнение обтравочных контуров

Наряду с обтеканием внешним содержимым в CSS можно установить способ заполнения элемента собственным содержимым, маскируемым с помощью обтравочного контура, который пересекает сам себя в нескольких местах (подобно тому, как это делается в SVG-изображениях). Для решения этой задачи в спецификацию добавлено стилевое свойство `clip-rule`.

clip-rule

Значение	`nonzero` | `evenodd`
Начальное значение	`nonzero`
Применяется	Все SVG-изображения, растровые изображения (`<circle>`, `<ellipse>`, `<image>`, `<line>`, `<path>`, `<polygon>`, `<polyline>`, `<rect>`, `<text>` и `<use>`) — *тогда и только тогда*, когда их родителем выступает элемент `<clipPath>`
Вычисляется	Согласно определению
Наследуется	Нет
Анимируется	Нет

Действие этого свойства намного проще продемонстрировать, чем описать словами, особенно в части, касающейся различий между ключевыми словами `nonzero` и `evenodd` (рис. 19.21).

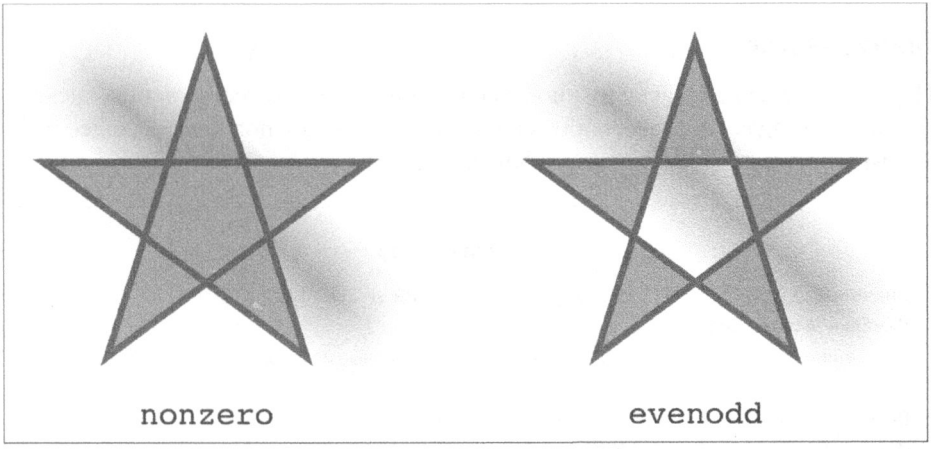


Рис. 19.21. Два способа заполнения маскируемого элемента содержимым (см. цветные иллюстрации на веб-сайте)

Здесь обтравочный контур в виде пятиконечной звезды рисуется начиная с верхнего луча. При этом контур, полученный с помощью значения `nonzero`, полностью заливается цветом, включая внутреннюю область, образованную пересечением собственных линий. В случае применения ключевого слова `evenodd` внутренняя область

остается незаполненной содержимым, и через нее просматривается светло-голубая градиентная заливка фона.

Как обычно, трудности возникают на этапе поддержки описанных возможностей пользовательскими агентами. К концу 2017 года все браузеры, успешно обрабатывавшие обтравочные контуры SVG, упорно не распознавали свойство `clip-rule`, независимо от того, внедряется ли SVG-изображение в целевой HTML-документ или располагается на внешнем ресурсе. Следовательно, для заливки обтравочного контура в режиме `evenodd` его нужно либо создать в CSS с помощью команды `polygon`, либо установить атрибут свойства `fill-rule` в самом SVG-файле.

Маски

Говоря о масках, мы будем подразумевать фигуры, через внутренние области которых просматривается содержимое элемента, а внешние области остаются непрозрачными. Функционально маски напоминают обтравочные контуры и отличаются от них всего двумя особенностями. Во-первых, в качестве масок допускается использовать графические изображения, а во-вторых, управление ими осуществляется с помощью большего количества свойств, поэтому они подлежат более точной настройке (размера, положения, способа повторения и т.п.).

К концу 2017 года браузеры семейства Blink поддерживали стилевые свойства маскирования элементов только при объявлении их в CSS-коде с вендорным префиксом `-webkit-`. Таким образом, в Chrome и Safari вместо названия `mask-image` нужно применять ключевое слово `webkit-mask-image`.

Создание маски

Перед применением маску нужно создать, используя в качестве основы некое изображение. Эта задача возлагается на свойство `mask-image`, поддерживающее работу с большим количеством графических форматов.

mask-image

Значение	[`none` | *<image>* | *<mask-source>*]#
Начальное значение	`none`
Применяется	Все элементы (SVG, растровые изображения и контейнеры, за исключением размеченных тегом `<defs>`)
Вычисляется	Согласно определению
Наследуется	Нет
Анимируется	Нет
Примечание	Значение *<image>* может представляться только типами данных *<url>*, *<image()>*, *<image-set()>*, *<element()>*, *<cross-fade()>* и *<gradient>* (все они описаны в предыдущих главах); значение *<mask-source>* выражается функцией `url()`, которая ссылается на элемент `<mask>`, включающий SVG-изображение

Если ссылка на изображение верна, то пользовательский агент получает в свое распоряжение графический файл, который будет применяться исключительно для маскирования целевого элемента.

Сначала ознакомимся с простейшим способом маскирования содержимого элемента, когда одно изображение накладывается на другое изображение такого же размера. На рис. 19.22 можно проследить за выполняемыми при этом действиями более наглядно. Как видите, совмещение изображений одинакового размера приводит к маскированию одного из них в точности по форме второго.

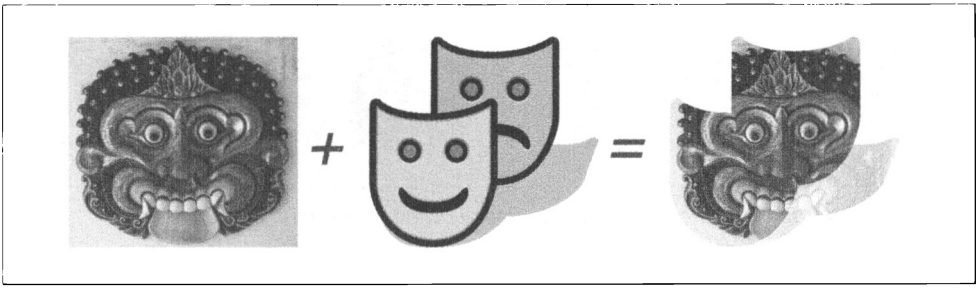


Рис. 19.22. Простая графическая маска

Легко заметить, что первое изображение просматривается только через непрозрачные области второго. И наоборот, его прозрачные области полностью закрывают расположенное под ними содержимое. Через полупрозрачные области второго изображения содержимое первого графического файла просматривается лишь частично.

Для получения результата, показанного на рис. 19.22, применяется следующий CSS-код:

```
img.masked {mask-image: url(theatre-masks.svg);}
```

CSS-маски не обязательно применять к одним только графическим изображениям. С помощью изображений, представляемых растровыми (GIF, JPG, PNG) или векторными (SVG) графическими файлами, можно маскировать элементы любого другого типа. Векторные изображения предпочтительнее применять в качестве масок. При необходимости маску можно создать с нуля, используя при ее рисовании линейные и радиальные градиенты, а также повторяющиеся элементы.

Для примера проанализируем следующее стилевое форматирование, результат применения которого к реальному документу показан на рис. 19.23.

```
*.masked.theatre {mask-image: url(theatre-masks.svg);}
*.masked.compass {mask-image: url(Compass.png);}
```

Важно понимать, что маскирование элемента осуществляется сразу во всех областях, обозначенных изображением маски. Проще всего данный эффект проиллюстрировать на примере маски, которая скрывает внешние области элемента, содержащего маркированный список. Такое стилевое форматирование неизбежно приведет к сокрытию маркеров (номеров) списка, как показано на рис. 19.24. Конечный результат обеспечивается за счет выполнения такого кода CSS.

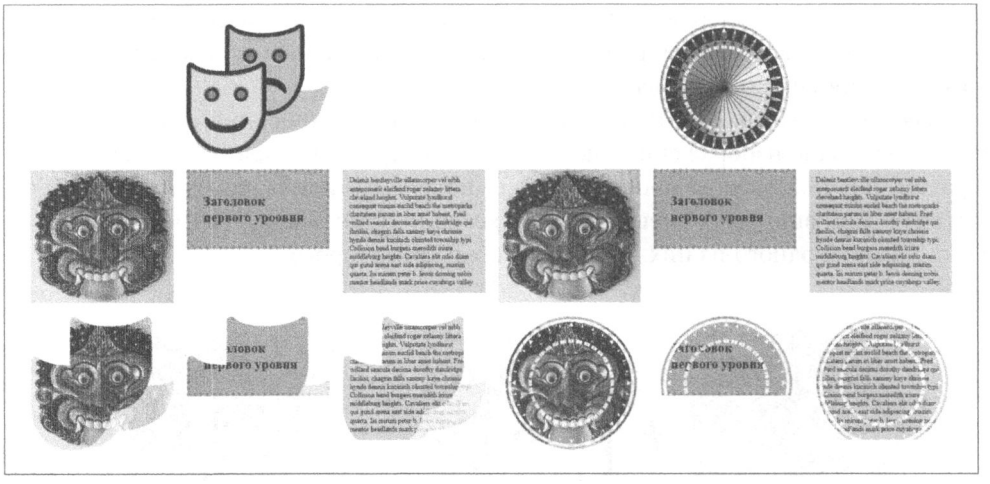


Рис. 19.23. Маски, представленные графическими элементами

```
*.masked {mask-image: url(i/Compass_masked.png);}

<ol class="masked">
   <li>One</li>
   <li>Two</li>
   <li>Three</li>
   <li>Four</li>
   <li>Five</li>
</ol>
```

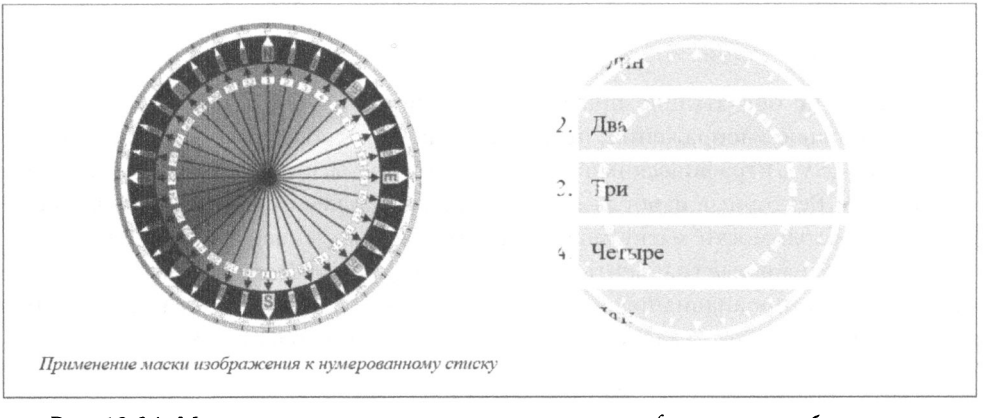


Рис. 19.24. Маскирование нумерованного списка графическим изображением

Существует еще один способ сокрытия отдельных частей элемента, о котором ранее не упоминалось, — с помощью маски, обозначаемой в SVG-изображении тегом `<mask>`. Для применения такой маски к элементу документа нужно воспользоваться методикой, подобной используемой при объявлении обтравочных контуров, которые образованы в SVG-изображении с помощью тега `<clipPath>` (см. раздел "Обтравочные контуры").

Для объявления маски в SVG-изображении применяется такой подход.

```
<svg viewbox="0 0 100 100" height="100" width="100"
   xmlns="http://www.w3.org/2000/svg" version="1.1">
      <mask id="hexlike">
      <path fill="#FF0000"
         d="M 50,0 100,25 100,75 50,100 0,75 0,25" />
      </mask>
</svg>
```

При непосредственном внедрении SVG-изображения в HTML-документ маска задействуется в нем следующим образом:

```
.masked {mask-image: url(#hexlike);}
```

Если SVG-маска объявляется во внешнем файле, то ее нужно подключить к целевому элементу с помощью специальной ссылки:

```
.masked {mask-image: url(masks.svg#hexlike);}
```

Режим маскирования

До настоящего момента нами рассматривались только случаи маскирования элементов изображениями, включающими альфа-канал. Наряду с этим маскирование элемента графическим изображением можно выполнять еще одним, альтернативным способом, согласно которому непрозрачные области маски определяются уровнем яркости пикселей, а не их прозрачности. Переключение между обозначенными режимами маски осуществляется с помощью свойства `mask-mode`.

mask-mode

Значение	[`alpha` | `luminance` | `match-source`]#
Начальное значение	`match-source`
Применяется	Все элементы (SVG, растровые изображения и контейнеры, за исключением размеченных тегом `<defs>`)
Вычисляется	Согласно определению
Наследуется	Нет
Анимируется	Нет

Первые два значения свойства полностью описываются их названиями. Ключевое слово `alpha` обязывает создавать маску на основе значений непрозрачности пикселей, указываемых в альфа-канале. Значение `luminance` предполагает создание маски, основываясь на значениях яркости пикселей ее изображения. Различие между режимами маскирования элементов проиллюстрировано на рис. 19.25, который получен в результате выполнения следующего кода.

```
img.theatre {mask-image: url(i/theatre-masks.svg);}
img.compass {mask-image: url(i/Compass_masked.png);}
img.lum {mask-mode: luminance;}
```

```
<img src="i/theatre-masks.svg">
<img class="theatre" src="i/mask.JPG">
<img class="theatre lum" src="i/mask.JPG">
<img src="i/Compass_masked.png">
<img class="compass" src="i/mask.JPG">
<img class="compass lum" src="i/mask.JPG">
```

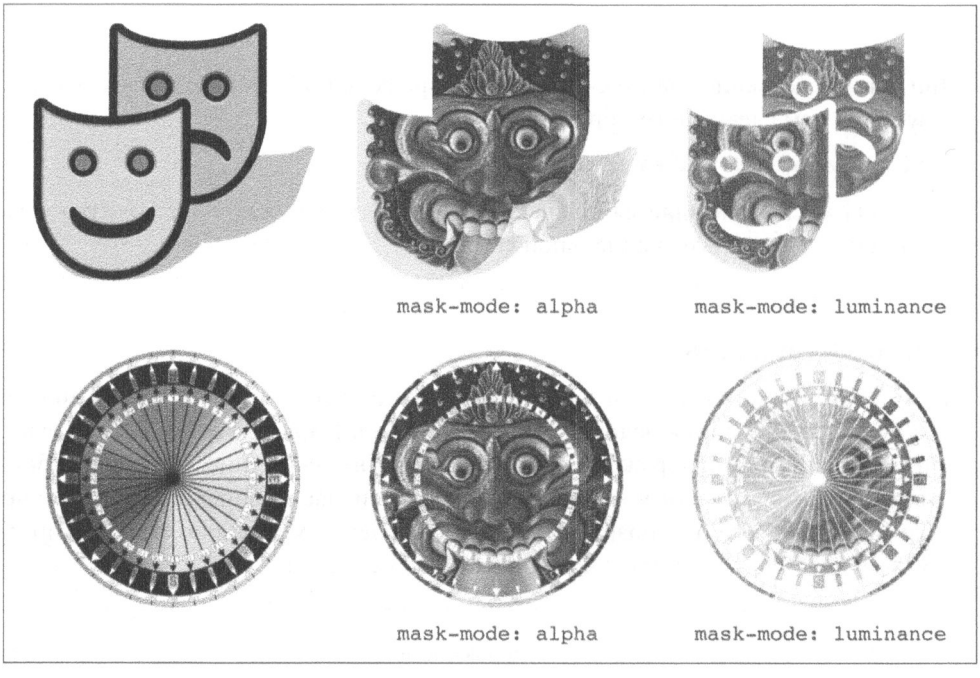


Рис. 19.25. Маски в режиме прозрачности и яркости

Уровни яркости в режиме `luminance` применяются для определения маскируемых областей так же, как и уровни непрозрачности в режиме `alpha`. Согласно последнему, в документе не отображаются области элемента, расположенные под полностью прозрачными областями графического изображения, а отображаются области элемента, находящиеся под абсолютно непрозрачными областями маски.

Такой же подход применяется при создании маски согласно уровням яркости. В документе скрываются те области элемента, что находятся под абсолютно черными областями изображения (нулевая яркость), а отображаются области элемента, расположенные под областями маски с максимальной яркостью (значение `1`). Важно понимать, что полностью прозрачные области изображения рассматриваются как имеющие нулевую яркость, а потому маскируют расположенный под ними элемент. По этой же причине прозрачные тени в изображении театральных масок становятся абсолютно непрозрачными при использовании его в качестве маски в режиме яркости.

Третье (применяемое по умолчанию) значение, `match-source`, представляет собой комбинацию режимов `luminance` и `alpha`, каждый из которых активизируется только при определенных условиях.

- Если источник маски представляется значением *<image>*, то маскирование выполняется в режиме `alpha`. Значению *<image>* может соответствовать изображение формата PNG или SVG, CSS-градиент либо часть документа, выделенная в отдельный элемент функцией `element()`.
- При представлении маски тегом `<mask>`, обозначенным в SVG-изображении, маскирование выполняется в режиме `liminance`.

Изменение размера и повторение маски

Почти все рассмотренные выше примеры предполагают использование масок одного размера с маскируемыми элементами. (Проще всего эту концепцию проиллюстрировать на примере маскирования графического элемента еще одним графическим изображением.) В реальных условиях маски имеют размер, заметно отличающийся от размера маскируемых с их помощью элементов. Выравнивание размеров маски и маскируемого элемента выполняется несколькими способами, самый очевидный из которых требует применения стилевого свойства `mask-size`.

mask-size

Значение	[[*<length>* | *<percentage>* | `auto`]{1,2} | `cover` | `contain`]#
Начальное значение	`auto`
Применяется	Все элементы (SVG, растровые изображения и контейнеры, за исключением размеченных тегом `<defs>`)
Вычисляется	Согласно определению
Наследуется	Нет
Анимируется	*<length>*, *<percentage>*

Если вам доводилось масштабировать фоновые изображения, то трудностей с использованием этого свойства возникать не должно, поскольку его синтаксис в точности повторяет код, применяемый для изменения размеров добавляемых в документ графических файлов. В качестве примера проанализируем следующий CSS-код, результат применения которого к текстовому документу показан на рис. 19.26.

```
p {mask-image: url(i/hexlike.svg);}
p:nth-child(1) {mask-size: 100% 100%;}
p:nth-child(2) {mask-size: 50% 100%;}
p:nth-child(3) {mask-size: 2em 3em;}
p:nth-child(4) {mask-size: cover;}
p:nth-child(5) {mask-size: contain;}
p:nth-child(6) {mask-size: 200% 50%;}
```

Снова-таки, конечный результат не покажется странным только тем, кому доводилось подбирать размер фоновых изображений. Если вам ранее не приходилось заниматься этим, то обратитесь к главе 9 за детальными инструкциями.

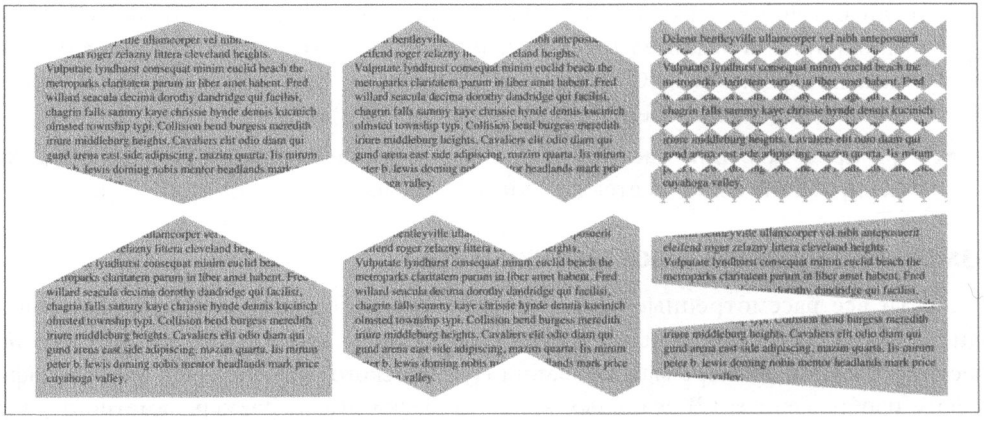


Рис. 19.26. Масштабирование масок

Продолжая аналогию с фоном элемента, можно смело утверждать, что повторение изображения маски с помощью свойства `mask-repeat` до заполнения его экземплярами всего элемента выполняется точно так же, как и повторение фонового изображения до заполнения его копиями всей поверхности фонового слоя.

mask-repeate

Значение	[`repeat-x` | `repeat-y` | [`repeat` | `space` | `round` | `no-repeat`]{1,2}]#
Начальное значение	`repeat`
Применяется	Все элементы (SVG, растровые изображения и контейнеры, за исключением размеченных тегом `<defs>`)
Вычисляется	Согласно определению
Наследуется	Нет
Анимируется	*<length>*, *<percentage>*
Примечание	Указанные значения полностью совпадают с применяемыми в свойстве `background-repeat`, как синтаксически, так и функционально

По вполне понятным причинам свойство `mask-repeat` имеет такие же значения, как и свойство `background-repeat`. Примеры применения к элементу масок, составленных из повторяющихся изображений, приведены на рис. 19.27. Все они получены в результате выполнения следующего кода CSS.

```
p {mask-image: url(i/theatre-masks.svg);}
p:nth-child(1) {mask-repeat: no-repeat; mask-size: 10% auto;}
p:nth-child(2) {mask-repeat: repeat-x; mask-size: 10% auto;}
p:nth-child(3) {mask-repeat: repeat-y; mask-size: 10% auto;}
p:nth-child(4) {mask-repeat: repeat; mask-size: 30% auto;}
p:nth-child(5) {mask-repeat: repeat round; mask-size: 30% auto;}
p:nth-child(6) {mask-repeat: space no-repeat; mask-size: 21% auto;}
```

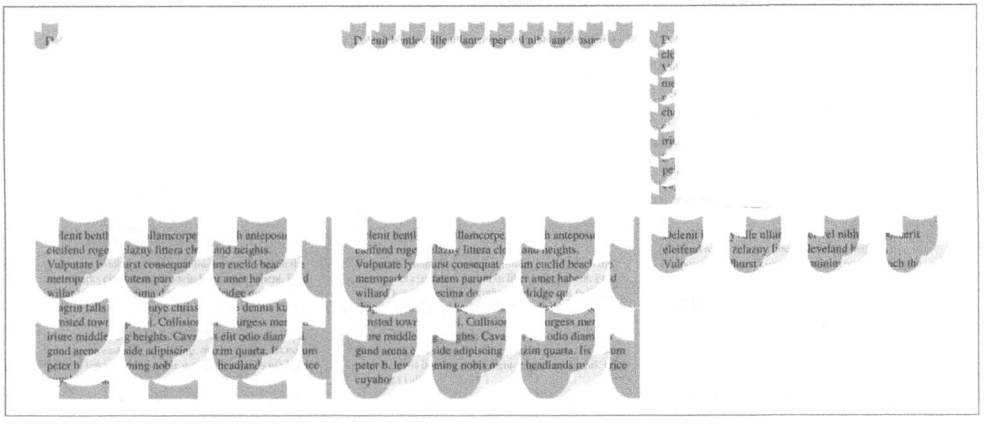


Рис. 19.27. Маски, составленные из экземпляров повторяющихся изображений

Позиционирование масок

После знакомства со способами масштабирования и повторения масок, в точности воспроизводящими методики изменения размера и повторения фоновых изображений, можно переходить к изучению стилевых свойств, отвечающих за определение положения и области позиционирования изображений масок. Функционально они сходны со свойствами `background-position` и `background-origin`.

mask-position

Значение	*<position>*#
Начальное значение	`0% 0%`
Применяется	Все элементы (SVG, растровые изображения и контейнеры, за исключением размеченных тегом `<defs>`)
Вычисляется	Согласно определению
Наследуется	Нет
Анимируется	*<length>*, *<percentage>*
Примечание	Значение *<position>* полностью совпадает с применяемым в свойстве `background-repeat`, как синтаксически, так и функционально

И снова, позиционирование изображения маски выполняется по таким же принципам, что и установка положения фонового изображения. Изучение свойства `mask-position` лучше продолжить на следующем примере, который приводит к результату, показанному на рис. 19.28 (пунктирной рамкой, добавленной вручную, обозначены границы целевого элемента).

```
p {mask-image: url(i/Compass_masked.png);
   mask-repeat: no-repeat; mask-size: 67% auto;}
p:nth-child(1) {mask-position: center;}
p:nth-child(2) {mask-position: top right;}
p:nth-child(3) {mask-position: 33% 80%;}
p:nth-child(4) {mask-position: 5em 120%;}
```

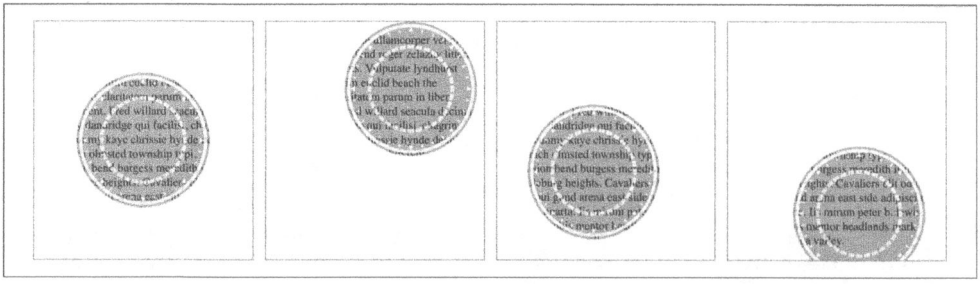


Рис. 19.28. Позиционирование масок

По умолчанию область позиционирования изображения маски определяется внешними краями границ элемента. Для дальнейшего ее расширения или указания совершенно иного контекста маскирования применяется свойство `mask-origin`, подобное свойству `background-origin`.

mask-origin

Значение	[`content-box` | `padding-box` | `border-box` | `margin-box` | `fill-box` | `stroke-box` | `view-box`]#
Начальное значение	`border-box`
Применяется	Все элементы (SVG, растровые изображения и контейнеры, за исключением размеченных тегом `<defs>`)
Вычисляется	Согласно определению
Наследуется	Нет
Анимируется	Нет

Область позиционирования фонового изображения относится к новым понятиям, включенным в последнюю версию спецификации, и может быть вам совершенно незнакома. Детальное описание стилевого свойства `background-origin` и настраиваемого им контекста форматирования приведено в главе 9. Вкратце с результатами его применения к одному и тому же элементу можно ознакомиться на рис. 19.29.

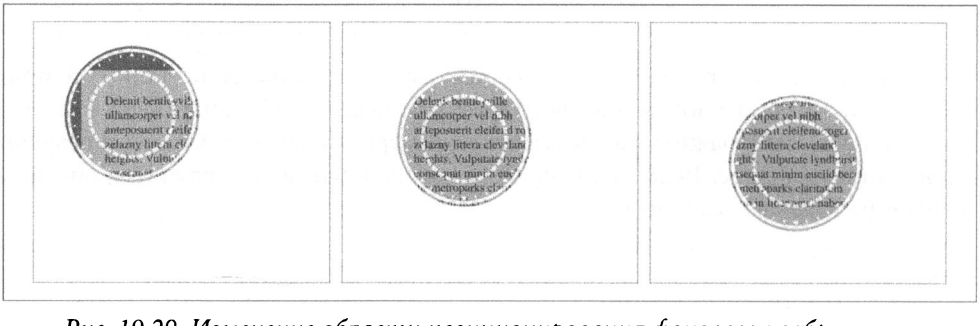


Рис. 19.29. Изменение области позиционирования фонового изображения

Обрезка и комбинирование масок

Существует еще одно стилевое свойство настройки масок, имеющее аналог среди средств настройки фоновых изображений. Речь идет о свойстве `mask-clip`, которое функционально сходно со свойством `background-clip`.

mask-clip

Значение	[`content-box` | `padding-box` | `border-box` | `margin-box` | `fill-box` | `stroke-box` | `view-box` | `no-clip`]#
Начальное значение	`border-box`
Применяется	Все элементы (SVG, растровые изображения и контейнеры, за исключением размеченных тегом `<defs>`)
Вычисляется	Согласно определению
Наследуется	Нет
Анимируется	Нет

Это свойство позволяет обрезать маску по краям определенной области маскируемого элемента. Фактически такая задача сводится к ограничению области элемента, в которой визуализируется его содержимое. Пример использования свойства `mask-clip` продемонстрирован следующим кодом (рис. 19.30).

```
p {padding: 2em; border: 2em solid purple; margin: 2em;
   mask-image: url(i/Compass_masked.png);
   mask-repeat: no-repeat; mask-size: 125%;
   mask-position: center;}
p:nth-child(1) {mask-clip: border-box;}
p:nth-child(2) {mask-clip: padding-box;}
p:nth-child(3) {mask-clip: content-box;}
```

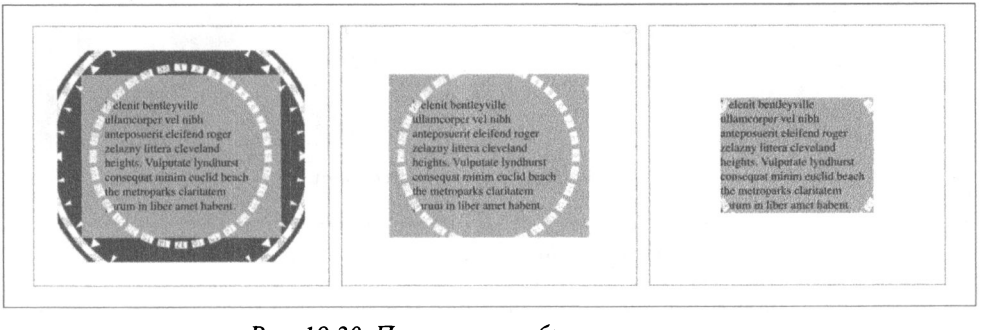


Рис. 19.30. Применение обрезанных масок

Последнее свойство, отвечающее за настройку масок, `mask-composite`, представляет особый интерес благодаря способности в корне изменить поведение сразу целой группы масок.

Свойство `mask-composite` не поддерживается в Chrome, даже в вендорном представлении.

mask-composite	
Значение	[`add` | `subtract` | `intersect` | `exclude`]#
Начальное значение	`add`
Применяется	Все элементы (SVG, растровые изображения и контейнеры, за исключением размеченных тегом `<defs>`)
Вычисляется	Согласно определению
Наследуется	Нет
Анимируется	Нет

Ключевые принципы и режимы наложения масок, определяемые с помощью этого свойства, проиллюстрированы на рис. 19.31.

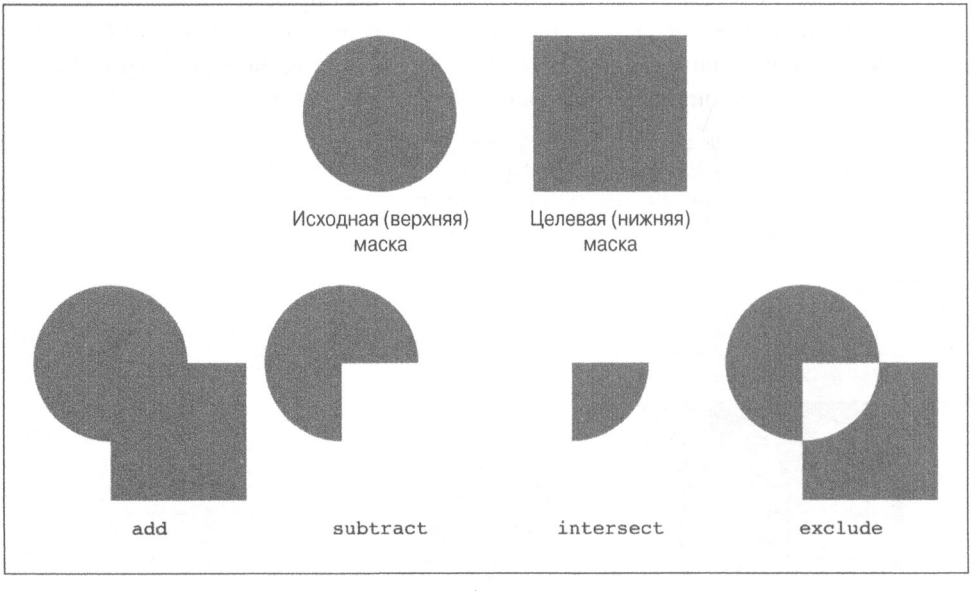


Рис. 19.31. Комбинирование масок

Как можно заметить на рис. 19.31, верхняя маска всегда называется *исходной*, а нижняя — *целевой*.

Для трех из четырех операций комбинирования — `add`, `intersect` и `exclude` — порядок наложения масок не играет особой роли. Независимо от того, какая из них будет исходной, а какая — целевой, получаемый результат будет одним и тем же. Определенную осторожность нужно проявлять только при выполнении операции `subtract`. Не забывайте, что целевая маска всегда вычитается из исходной маски.

Поменяв маски местами, вы получите совершенно иной результат операции вычитания. На рис. 19.32 наглядно показано, чем чревато изменение порядка наложения масок на обратный при их комбинировании с помощью команды `subtract`.

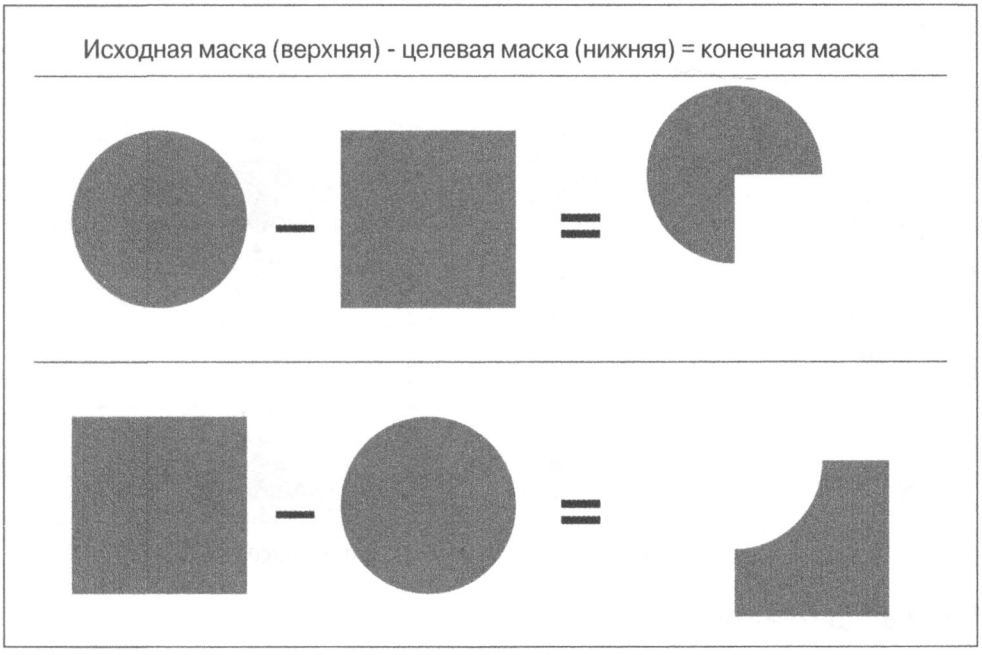


Рис. 19.32. Вычитание масок

Кроме того, за порядком указания целевых и исходных масок нужно следить при их комбинировании в больших количествах. В подобных случаях поддерживается общепринятый в CSS порядок наложения изображений: снизу вверх. Следовательно, каждая следующая добавляемая в стек маска считается исходной, а добавленная перед ней становится целевой.

Чтобы понять, какую роль порядок наложения масок играет в получаемом результате, внимательно рассмотрите рис. 19.33. На нем показаны разные способы наложения трех масок, указываемых в разных последовательностях и комбинируемых с помощью разных операций.

Каждый столбец фигур представляет последовательность комбинируемых масок. Самая нижняя фигура обозначает первую накладываемую на элемент маску, а самая верхняя (расположенная поверх остальных двух) — последнюю. Результат комбинирования последовательности масок показан над ней самой — он представляется самой верхней фигурой столбца. Следовательно, в первом столбце маски комбинируются в такой последовательности: сначала из круга исключается треугольник, а затем на полученную фигуру накладывается квадрат. Конечный результат представлен самой верхней фигурой первого столбца.

Не забывайте, что при выполнении операции вычитания (`subtract`) нижняя фигура всегда вычитается из верхней, но не наоборот. Таким образом, комбинирование

масок, представленных в третьем столбце, сводится к вычитанию результата сложения треугольника и круга из расположенного над ними квадрата.

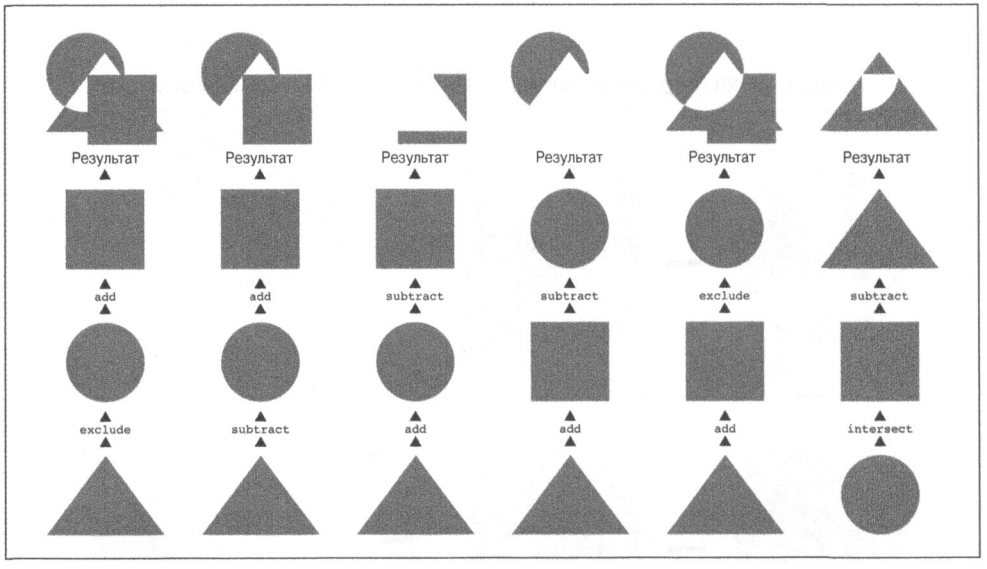


Рис. 19.33. Примеры комбинирования масок

Маски общего назначения

Все описанные выше стилевые свойства настройки масок можно успешно заменить единственным свойством общего назначения.

mask	
Значение	[<*mask-image*> || <*mask-position*> [/ <*mask-size*>]? || <*mask-repeat*> || <*maskclip*> || <*mask-origin*> || <*mask-composite*> || <*mask-mode*>]#
Начальное значение	См. описание индивидуальных свойств
Применяется	Все элементы (SVG, растровые изображения и контейнеры, за исключением размеченных тегом `<defs>`)
Вычисляется	Согласно определению
Наследуется	Нет
Анимируется	См. описание индивидуальных свойств

Свойство `mask`, как и любое другое свойство общего назначения, получает список разделенных запятыми наборов настроек, каждый из которых относится к отдельной маске. Значения свойств, определяемых после *<mask-size>*, могут задаваться в произвольной последовательности — размер маски всегда указывается после ее положения и отделяется от значения последнего косой чертой.

Следовательно, оба приведенных ниже правила абсолютно равнозначны.

```
#example {
   mask-image: url(circle.svg), url(square.png), url(triangle.gif);
   mask-repeat: repeat-y, no-repeat;
   mask-position: top right, center, 25% 67%;
   mask-composite: subtract, add, add;
   mask-size: auto, 50% 33%, contain;
}
#example {
   mask:
   url(circle.svg) repeat-y top right / auto subtract,
   url(square.png) no-repeat center / 50% 33% add,
   url(triangle.gif) repeat-y 25% 67% / contain add;
}
```

Согласно приведенным выше стилевым правилам, маска элемента представляется фигурой, образованной в результате вычитания результата сложения повторяемых изображений треугольника и квадрата из круга. Результат применения такой маски к квадратному элементу показан на рис. 19.34, *слева*, а к прямоугольному элементу, вытянутому по горизонтали, — на рис. 19.34, *справа*.

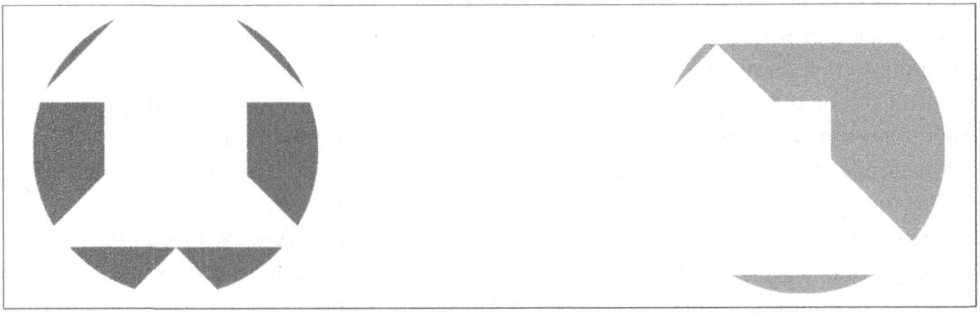


Рис. 19.34. Две маски

Типы масок

При стилевом форматировании SVG-изображений к ним можно применять маски одного из двух типов. В CSS тип маски для SVG-элемента `mask` указывается с помощью свойства `mask-type`.

mask-type

Значение	`luminance` | `alpha`
Начальное значение	`luminance`
Применяется	Элементы `<mask>` в SVG-изображениях
Вычисляется	Согласно определению
Наследуется	Нет
Анимируется	Нет

Это свойство напоминает рассмотренное ранее свойство `mask-mode`, хотя и не снабжается эквивалентом значения, представляемого ключевым словом `match-source`. Таким образом, среди возможных значений — только `luminance` и `alpha`.

Важно понимать, что свойство `mask-type` объявляется для элемента `mask`, представляющего изображение маски, а свойство `mask-mode` — для элемента, к которому применяется сама маска. В конфликтных ситуациях предпочтение всегда отдается типу маски, устанавливаемому свойством `mask-mode`. В качестве примера рассмотрим такой код.

```
svg #mask {mask-type: alpha;}
img.masked {mask: url(#mask) no-repeat center/cover luminance;}
```

Объявление этих правил в реальном документе приведет к применению маски в режиме яркости (`luminance`), а не прозрачности (`alpha`). Если же свойство `mask-mode` оставить в значении по умолчанию, `match-source`, то тип маски будет определяться свойством `mask-type`.

Маскирование по рисованной рамке

Помимо средств настройки обтравочных контуров и масок, модуль CSS Masking Level 1 включает инструменты маскирования элементов изображениями, подобными применяемым в свойствах семейства `border-image`. В действительности у свойств создания рисованных рамок и масок, представляемых изображениями рисованных рамок, очень много общего, включая полностью идентичные значения.

К недостаткам свойств, описанных в этом разделе, относится отсутствие их поддержки браузерами. Мало того что к концу 2017 года ни один из браузеров не поддерживал их — ни одна из команд разработчиков так и не удосужилась принять их к рассмотрению. Весьма сомнительно, что в ближайшем будущем они обратят на себя внимание основных производителей программного обеспечения. Именно поэтому вместо детального рассмотрения свойств создания масок, представленных рисованными рамками, мы остановимся только на их кратком описании.

`mask-border-source`

Указывает изображение маски, представляемое URL, градиентом или любым другим допустимым значением *<image>*.

`mask-border-slice`

Определяет способ нарезки исходного изображения на части, применяемые для маскирования границ элемента, а также режим заливки внутренних областей маски.

`mask-border-width`

Устанавливает ширину области границ(ы) элемента, к которой(ым) применяется рисованная маска, образованная из частей исходного изображения.

`mask-border-outset`

Указывает расстояние от краев границы элемента, на которое может распространяться маска по рисованной рамке.

`mask-border-repeat`

Определяет способ повторения нарезанных частей изображения маски в случае неточного позиционирования их по рамке границ элемента (изменение размера, растягивание и т.п.).

`mask-border-mode`

Указывает режим маскирования по уровням яркости или прозрачности.

`mask-border`

Свойство общего назначения, включающее функции всех предыдущих свойств.

Чтобы получить более полное представление о назначении приведенных выше свойств, обратитесь к главе 8, в которой подробно описаны все объявленные спецификацией свойства создания рисованных рамок.

Подгонка объектов

К стилевым средствам маскирования можно условно отнести еще два необычайно интересных свойства, в основном применяемых к замещаемым элементам, таким как изображения. Свойство `object-fit` указывает способ заполнения замещаемым элементом контейнера элемента — в случаях, когда он несколько меньше его по размеру.

object-fit

Значение	`fill` | `contain` | `cover` | `scale-down` | `none`
Начальное значение	`fill`
Применяется	Замещаемые элементы
Вычисляется	Согласно определению
Наследуется	Нет
Анимируется	Нет

Имея навыки использования свойства `background-size`, вам не составит большого труда понять, для чего служит каждое из значений свойства `object-fit`. Их назначение остается неизменным, разве что здесь они применяются исключительно к замещаемым элементам.

В качестве примера рассмотрим изображение размером 50×50 пикселей. Воспользовавшись инструментами CSS, его можно увеличить так, как показано ниже.

```
img {width: 250px; height: 150px;}
```

Наиболее ожидаемый результат такого форматирования — это увеличение изображения до размера 250×150 пикселей. Но он будет получен только при установке свойства `object-fit` в значение `fill`.

При передаче свойству `object-fit` иного значения изображение будет иметь размер, отличный от указанного в предыдущем стилевом правиле, что продемонстрировано на рис. 19.35, полученном в результате выполнения такого кода.

```
img {width: 250px; height: 150px; background: silver; border: 3px solid;}
img:nth-of-type(1) {object-fit: none;}
img:nth-of-type(2) {object-fit: fill;}
img:nth-of-type(3) {object-fit: cover;}
img:nth-of-type(4) {object-fit: contain;}
```

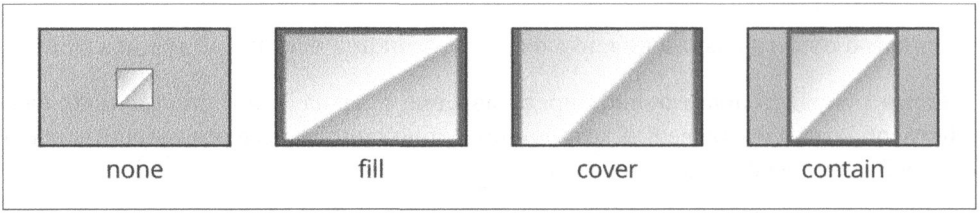


Рис. 19.35. Четыре способа подгонки изображения под размер контейнера

В первом случае способ подгонки изображения определяется ключевым словом `none`. Так как для элемента указан размер 250×150 пикселей, а изображение имеет исходный размер 50×50 пикселей, то оно будет оставаться в нем до выбора следующего режима подгонки: `fill`.

В третьем режиме подгонки, представляемом ключевым словом `cover`, изображение масштабируется до максимально возможного размера с сохранением исходных пропорций. Другими словами, изображение получает максимальный размер, сохраняя свою квадратную форму. Так как большая сторона элемента имеет размер 250 пикселей, изображение будет увеличено до размеров 250×250 пикселей, а затем помещено в элемент размером 250×150 пикселей.

Четвертое значение, `contain`, предполагает такую же стратегию масштабирования, но без выхода отдельных частей изображения за пределы элемента `img`. Буквально это означает, что в элемент размером 250×150 пикселей будет помещено изображение, растянутое до размера 150×150 пикселей.

Итак, на рис. 19.35 показаны четыре обособленных элемента `img`, которые лишены внешних контейнеров, подобных `div` или `span`. Границы и фоновая заливка указаны для элементов `img`, а способ вставки в них графических элементов определяется свойством `object-fit`. В данном случае можно смело утверждать, что контейнер элемента выступает маской для его содержимого. (При этом сам контейнер элемента `img` можно маскировать с помощью любых описанных ранее свойств.)

Пятое значение свойства `object-fit`, не представленное наглядным примером на рис. 19.35, называется `scale-down`. Согласно спецификации, режим `scale-down` равнозначен режиму `none` или `contain`, в зависимости от того, какой из них подгоняет содержимое до меньших размеров. Применение его к изображению из приведенного

выше примера позволяет сохранять его исходный размер до тех пор, пока элемент `img` не станет меньше 50×50 пикселей. С этого момента подгонка изображения под размер элемента `img` будет выполняться в режиме `contain`. С принципом работы режима `scale-down` можно ознакомиться на рис. 19.36. В каждом из примеров значение `height` указывает высоту элемента `img`, а его ширина остается неизменной и составляет `100px`.

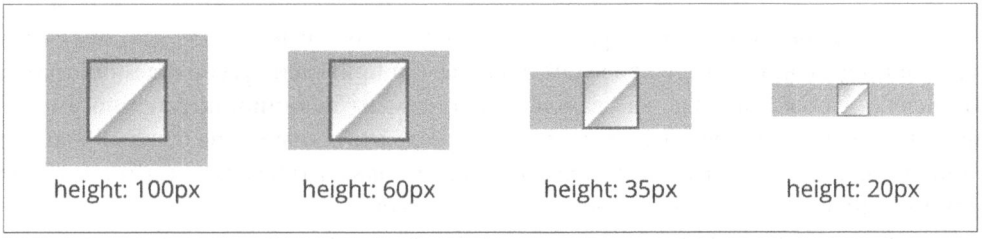


Рис. 19.36. Результат подгонки изображения в режиме `scale-down`

Кроме подгонки к размеру, замещаемый элемент можно выравнивать в своем контейнере, что особенно актуально при существенной разнице в их размерах. В CSS выравнивание содержимого элемента в собственном контейнере выполняется с помощью свойства `object-position`.

object-position

Значение	*<position>*
Начальное значение	`50% 50%`
Применяется	Замещаемые элементы
Вычисляется	Согласно определению
Наследуется	Нет
Анимируется	Да
Примечание	Значение *<position>* полностью совпадает с применяемым в свойстве `background-repeat`, как синтаксически, так и функционально

Синтаксис значений этого свойства полностью совпадает с таковым у свойств `mask-position` и `background-position`, а сами они обеспечивают позиционирование замещаемого элемента в пределах собственного контейнера при подгонке во всех режимах, кроме `fill`. Таким образом, после выполнения следующего CSS-кода будет получен результат, показанный на рис. 19.37.

```
img {width: 200px; height: 100px; background: silver;
     border: 1px solid; object-fit: none;}
img:nth-of-type(2) {object-position: top left;}
img:nth-of-type(3) {object-position: 67% 100%;}
img:nth-of-type(4) {object-position: left 142%;}
```

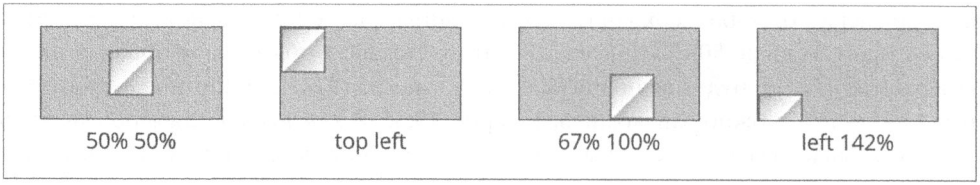


Рис. 19.37. Позиционирование замещаемого элемента в пределах контейнера

Заметьте: первый пример на рис. 19.37 получен для значения `50% 50%`, отсутствующего в приведенном выше стилевом правиле. Он приведен сугубо в демонстративных целях, чтобы показать, как замещаемый элемент позиционируется по умолчанию. Следующие два примера представлены в коде отдельными значениями и весьма наглядно иллюстрируют стилевые возможности пользовательских агентов по позиционированию изображения в контейнере элемента `img`.

Последний пример призван продемонстрировать еще одну, не менее интересную возможность: смещение замещаемого элемента за пределы контейнера. В процессе выполнения такой операции содержимое элемента неизбежно обрезается по границам контейнера. Такое же поведение свойственно маскам и фоновым изображениям, позиционирование которых выполняется с помощью свойств `mask-position` и `background-position`.

Специальному позиционированию также подлежат замещаемые элементы, размер которых больше размера контейнера, что часто случается при подгонке в режиме `object-fit: cover`. В случае отсутствия подгонки (`object-fit: none`) результат сильно отличается от получаемого по умолчанию. В частности, приведенный ниже CSS-код обеспечивает замещаемые элементы стилевым форматированием, продемонстрированным на рис. 19.38.

```
img {width: 200px; height: 100px; background: silver; border:
     1px solid; object-fit: cover;}
img:nth-of-type(2) {object-position: top left;}
img:nth-of-type(3) {object-position: 67% 100%;}
img:nth-of-type(4) {object-position: left 142%;}
```

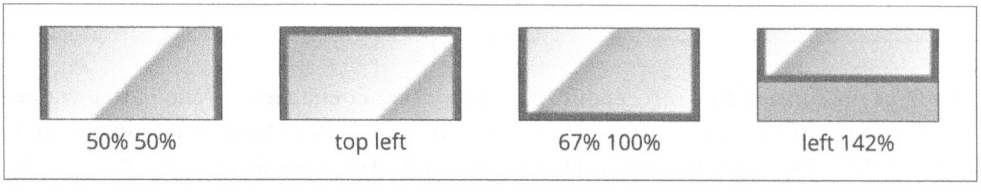


Рис. 19.38. Позиционирование содержимого в контейнере меньшего размера

Если результаты позиционирования замещаемого элемента в пределах контейнера своего элемента кажутся вам непонятными, обратитесь к главе 9, содержащей детальное описание свойства `background-position`.

Форматирование, зависящее от носителя

К одному из основных преимуществ технологии CSS относится возможность отображения документа, содержащего стилевое форматирование, в самых разных аппаратных и программных средах. Ее инструменты позволяют одинаково комфортно чувствовать себя при просмотре документа как на мониторе настольного компьютера, так и на экране смартфона. Конечно, корректность отображения документов на экране электронного устройства в очень малой степени зависит от него самого и в первую очередь определяется технологиями, которые использовались при их создании. Осознавая острую потребность в поддержке как можно большего количества устройств, спецификация CSS включает целый спектр инструментов, позволяющих изменять стилевое форматирование элементов в зависимости от устройства, на котором они отображаются.

Стилевое форматирование, зависящее от устройства

Включение в HTML и CSS поддержки специальной технологии, называемой *медиа-запросы*, позволяет ограничить область действия таблицы стилей только носителями определенного типа или устройствами, обладающими строго заданными техническим характеристиками. Критерием применимости таблицы стилей, указываемой в медиа-запросе, обычно выступает комбинация допустимых типов устройств и их рабочих характеристик, например разрешения экрана и/или глубины цвета. Как и во многих других случаях, изучение возможностей медиа-запросов мы начнем с простых примеров и по мере их освоения перейдем к рассмотрению более сложных программных структур.

Простые медиа-запросы

Ограничение на применение таблицы стилей к HTML-документу в разных аппаратных средах устанавливается с помощью атрибута `media`. Он выполняет совершенно одинаковые функции как в элементе `link`, так и в элементе `style`.

```
<link rel="stylesheet" type="text/css" media="print"
      href="article-print.css">
<style type="text/css" media="speech">
   body {font-family: sans-serif;}
</style>
```

Атрибут media может представляться как единственным значением, так и целым списком значений, разделяемых запятыми. Например, следующий код обеспечивает подключение к документу таблицы стилей, применяемой только при выводе его на устройствах, тип которых определен как screen или speech.

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

В самой таблице стилей подобное ограничение устанавливается в правиле @import, как показано ниже.

```
@import url(visual.css) screen;
@import url(outloud.css) speech;
@import url(article-print.css) print;
```

Учтите, что если в таблице стилей не указать допустимые устройства или носители, то она будет применяться во *всех* них. С другой стороны, для применения к документу разных таблиц стилей при выводе на бумагу и на экран допустимый тип устройства или носителя нужно указать для каждой из них отдельно.

```
<link rel="stylesheet" type="text/css" media="screen"
      href="article-screen.css">
<link rel="stylesheet" type="text/css" media="print"
      href="article-print.css">
```

При удалении атрибута media из первого тега link предыдущего примера стилевые правила, объявленные в таблице стилей article-screen.css, будут применяться при отображении документа на *любых* устройствах и носителях.

Кроме того, определить аппаратную среду для таблицы стилей можно с помощью команды @media. Воспользовавшись ею, в одной таблице стилей можно объявлять разные способы форматирования сразу для нескольких типов устройств. Рассмотрим такой 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>
```

В этом коде первым указывается стилевое правило, применяемое для всех без исключения форматов вывода. Оно устанавливает для документа белый фон и черный цвет переднего плана. Это правило становится общим для всех вариантов вывода благодаря отсутствию в объявлении таблицы стилей атрибута media, который ввиду этого принимает значение по умолчанию: all. После общего правила указывается блок стилевых правил для устройств, относящихся к типу screen, и только после него — блок правил, объявляемых для носителей типа print.

Правило `@media` может состоять из любого количества стилевых правил и иметь произвольный размер. В случае форматирования многочисленных документов с помощью единственной таблицы стилей — при использовании совместного хостинга или системы управления контентом, ограничивающей возможности по ее редактированию, — команда `@media` предоставляет едва ли не единственную возможность применения в них стилей, зависящих от устройства или носителя. Кроме того, она становится незаменимой в таблицах стилей, которые применяются к XML-документам, лишенным возможности использования атрибута `media` и каких бы то ни было его аналогов.

Чаще всего ограничение на применение стилевого форматирования касается всего четырех типов устройств и носителей.

`all`

Представляет все возможные типы устройств и носителей.

`print`

Позволяет указывать форматирование документа, выводимого на печать, а также отображаемого в режиме предварительного просмотра.

`screen`

Представляет устройства, снабженные экранами того или иного вида, например мониторы настольных компьютеров или мобильные устройства. Все браузеры, запускаемые в таких системах, считаются экранными пользовательскими агентами.

`speech`

К этому типу носителей относят голосовые синтезаторы, экранные дикторы и другие системы воспроизведения речи.

В HTML4 включен собственный список типов устройств и носителей, распознаваемых обработчиком CSS, но большинство значений совершенно не пригодно для применения в стилевых правилах. Они представляются такими ключевыми словами, как `aural`, `braille`, `embossed`, `handheld`, `projection`, `tty` и `tv`. Повстречав их в старых таблицах стилей, без лишнего замешательства заменяйте их приведенными выше значениями, представляющими четыре наиболее распространенных типа устройств и носителей.

Со временем спецификация будет пополняться новыми значениями, обозначающими другие типы устройств и носителей. Помните об этом, объявляя их в собственных таблицах стилей. Так, например, вскоре спецификация может быть дополнена значением `augmented-reality`, представляющим устройства дополненной реальности. Выделение таких устройств в отдельный тип в первую очередь вызвано необходимостью представления на них текста с заметно большей контрастностью, чем на остальных устройствах, что объясняется его демонстрацией поверх реально существующего изображения.

В определенных ситуациях в качестве значения требуется указать сразу несколько типов устройств или носителей. В подобных случаях они перечисляются через запятую. Конечно, такие ситуации возникают очень редко, в немалой степени благодаря поддержке весьма ограниченного набора доступных значений. В частности, приведенное ниже стилевое форматирование применяется только к документам, выводимым на экран или распечатываемым на бумаге.

```
<link rel="stylesheet" type="text/css" media="screen, print"
      href="article.css">

@import url(article.css) print, screen;

@media screen, print {
/* стили */
}
```

Составные медиа-запросы

В предыдущем разделе было показано, как определять стилевое форматирование сразу для нескольких типов устройств или носителей. Медиа-запрос, в котором объявляется список значений, разделенных запятой, называется составным, поскольку он адресуется сразу нескольким вариантам вывода. Разумеется, медиа-запрос должен предельно точно определять условия применения стилевого форматирования, включая сведения не только о допустимых устройствах или носителях, но и об отдельных технических характеристиках, например разрешении экрана и глубине цвета.

При построении составных медиа-запросов невозможно обойтись только одним разделителем (запятой), поэтому их синтаксис допускает применение в условиях логического оператора `and`, используемого для связывания типа устройства с отдельной технической характеристикой.

Рассмотрим, как такой подход реализуется на практике. Существуют два базовых способа объявления таблицы стилей, которая применяется к документу, выводимому на цветную печать.

```
<link href="print-color.css" type="text/css"
      media="print and (color)" rel="stylesheet">

@import url(print-color.css) print and (color);
```

Медиа-запросы применяются везде, где требуется добиться разного форматирования при выводе документа на разных носителях. В продолжение примера из предыдущего раздела рассмотрим, как приведенный выше код будет выглядеть при объявлении таблицы стилей сразу для двух типов устройств.

```
<link href="print-color.css" type="text/css"
      media="print and (color), screen and (color)" rel="stylesheet">

@import url(print-color.css) print and (color), screen and (color);
```

Для применения к документу таблицы стилей, указанной в приведенном выше коде, достаточно выполнения условия хотя бы для одного из двух устройств. Таким

образом, файл print-color.css, объявленный в команде @import, будет применяться для форматирования документа при выводе его на цветную печать *или* отображении на цветном экране. При печати на черно-белом принтере оба условия будут ложными, поэтому указанная таблица стилей не будет подключаться к документу. Такой же результат будет получен при выводе документа на черно-белом мониторе.

В каждом условии, называемом *дескриптором носителя*, указывается тип носителя и одна или несколько технических характеристик, заключаемых в круглые скобки. В отсутствие в медиа-запросе ключевого слова, определяющего тип носителя, применяется значение all. Следовательно, приведенные ниже медиа-запросы полностью равнозначны.

```
@media all and (min-resolution: 96dpi) {...}
@media (min-resolution: 960dpi) {...}
```

В общем случае *дескриптор технических характеристик*, указываемый в медиа-запросе и заключаемый в отдельные скобки, представляется в формате свойство–значение. Но из этого правила есть исключение: свойство можно указывать без конечного значения, как в примере объявления таблицы стилей для документа, выводимого на цветном принтере или мониторе. В нем техническая характеристика устройства обозначалась записью (color), определяющей любой цветной носитель информации. В то же время значение (color: 16) будет представлять любой цветной носитель с 16-битовой глубиной цвета. В любом случае указание технической характеристики (свойства) без соответствующего значения является проверкой на истинность для нее самой. В частности, запись (color) равнозначна вопросу "является ли носитель цветным?"

В медиа-запросах можно применять составные условия, допускающие использование логического оператора and. На самом деле в условиях медиа-запроса разрешается применять два логических оператора.

and

Объединяет два или более условия так, что конечное выражение будет считаться истинным только при истинности каждого из отдельных условий. Например, истинность выражения (color) and (orientation: landscape) and (min-device-width: 800px) требует истинности всех трех условий. В данном случае таблица стилей будет применяться только при отображении документа на цветном устройстве с альбомной ориентацией экрана, горизонтальное разрешение которого не менее 800 пикселей.

not

Отменяет действие таблицы стилей при выполнении всех условий, заданных в медиа-запросе. Например, выражение not (color) and (orientation: landscape) and (min-device-width: 800px) предотвращает применение указанного в медиа-запросе стилевого форматирования при выполнении всех трех условий. Таким образом, в данном примере таблица стилей *не* будет применяться только при отображении документа на цветном устройстве с альбомной ориентацией экрана и горизонтальным разрешением не менее 800 пикселей.

Заметьте: ключевое слово `not` может указываться только в начале условия медиа-запроса. Тем самым обеспечивается недействительность синтаксической конструкции `(color) and not (min-device-width: 800px)` в старых браузерах. Такие команды полностью игнорируются пользовательским агентом старого образца. Не забывайте, что старые браузеры, не обрабатывающие медиа-запросы, не применяют обусловливаемое ими стилевое форматирование ни при каких обстоятельствах.

Ниже приведено несколько примеров составных медиа-запросов, результат выполнения которых показан на рис. 20.1.

```
@media screen and (min-resolution: 72dpi) {
   .cl01 {font-style: italic;}
}
@media screen and (min-resolution: 32767dpi) {
   .cl02 {font-style: italic;}
}
@media not print {
   .cl03 {font-style: italic;}
}
@media not print and (grayscale) {
   .cl04 {font-style: italic;}
}
```

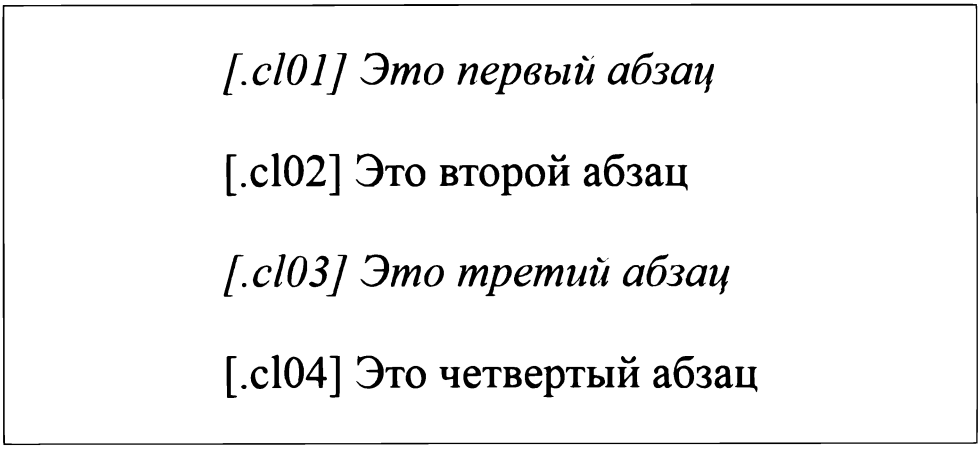


Рис. 20.1. Логические операторы в медиа-запросах

Важно понимать, что на рис. 20.1 показана копия окна браузера (Firefox, если быть предельно точным), в котором отображается HTML-документ со стилевым форматированием, обусловленным приведенным выше CSS-кодом. Таким образом, несмотря на вывод на бумагу, оно будет устанавливаться командами с дескриптором носителя `screen`, но никак не `print`.

Первая текстовая строка получает курсивное начертание, поскольку документ выводится на экран с разрешением, большим `72dpi`. Вторая текстовая строка остается в исходном, обычном начертании, так как разрешение экрана меньше значения `32767dpi`, и второй блок команд медиа-запроса становится недействительным.

Курсивность третьей текстовой строки обеспечивается выводом документа на экранное устройство (`not print`), а начертание четвертой строки определяется вторым условием (`grayscale`) своего медиа-запроса: документ выводится на цветной, а не монохромный экран.

Легко заметить, что в медиа-запросах отсутствует логический оператор `or`. Его функцию выполняет символ запятой, разделяющий дескрипторы отдельных носителей. В частности, условие `screen, print` указывает применять стилевое форматирование медиа-запроса к документам, которые выводятся на экран *или* на печать. Поэтому недействительную синтаксическую конструкцию `screen and (max-color: 2) or (monochrome)` нужно представить выражением `screen and (max-color: 2), screen and (monochrome)`.

К допустимым логическим операторам, применяемым в медиа-запросах, можно также отнести ключевое слово `only`, снабжающее медиа-запрос функцией обратной совместимости.

`only`

Применяется для сокрытия таблиц стилей в старых браузерах, не распознающих медиа-запросы. Например, для применения стилевого форматирования к документу, отображаемому на любых устройствах и носителях, но только в браузерах, обрабатывающих медиа-запросы, в CSS-код нужно включить команду `@import url(new.css) only all`. Современные браузеры, поддерживающие команду `@import`, игнорируют ключевое слово `only` и распознают правило как `import url(new.css) all`. Старые браузеры, лишенные поддержки медиа-запросов, распознают только синтаксическую конструкцию `only all`, с их точки зрения полностью несостоятельную. Именно поэтому в первом случае таблица стилей применяется к документу, а во втором — нет. Заметьте, что ключевое слово `only` допускается вводить только в начале определения дескриптора носителя.

Дескрипторы технических характеристик

В предыдущих примерах вы уже сталкивались с некоторыми дескрипторами технических характеристик устройств. Но доступных для использования дескрипторов этого типа намного больше, и вам обязательно нужно ознакомиться со всеми ними.

Учтите, что ни один из перечисленных далее дескрипторов не может иметь отрицательное значение. Также не забывайте, что дескрипторы технических характеристик обязательно заключаются в круглые скобки.

`width, min-width, max-width`
значение: *<length>*

Определяет ширину отображаемой области пользовательского агента. В пользовательских агентах, выводящих документы на экран, она равна суммарной ширине окна просмотра и полос прокрутки. В бумажных носителях ширина отображаемой области будет равняться ширине контейнера страницы. Таким образом, условие `(min-width: 850px)` указывает применять форматирование только в пользовательских агентах с шириной отображаемой области больше 850 пикселей.

`height, min-height, max-height`
значение: *<length>*

Указывает высоту отображаемой области пользовательского агента. В пользовательских агентах, выводящих документы на экран, она равна суммарной высоте области просмотра и полос прокрутки. В бумажных носителях высота отображаемой области будет равняться высоте контейнера страницы. Таким образом, условие `(height: 567px)` указывает применять форматирование только в пользовательских агентах, высота отображаемой области которых равна точно 567 пикселям.

`device-width, min-device-width, max-device-width`
значение: *<length>*

Определяет полную ширину области визуализации устройства вывода. В устройствах, снабженных экраном, она равна ширине или горизонтальному размеру экрана. В бумажных носителях ширина области визуализации будет равняться ширине страницы. Таким образом, условие `(max-device-width: 1200px)` указывает применять форматирование только в пользовательских агентах с областью визуализации, ширина которой меньше 1200 пикселей.

`device-height, min-device-height, max-device-height`
значение: *<length>*

Указывает полную высоту области визуализации устройства вывода. В устройствах, снабженных экраном, она равна высоте или вертикальному размеру экрана. В бумажных носителях высота области визуализации будет равняться высоте страницы. Таким образом, условие `(max-device-height: 400px)` указывает применять форматирование только в пользовательских агентах с областью визуализации, высота которой меньше 400 пикселей.

`aspect-ratio, min-aspect-ratio, max-aspect-ratio`
значение: *<ratio>*

Устанавливает соотношение дескриптора характеристики `width` к дескриптору характеристики `height` носителя (см. описание значения *<ratio>* в следующем разделе). Таким образом, условие `(min-aspect-ratio: 2/1)` указывает применять форматирование только в ситуациях, когда соотношение сторон отображаемой области составляет `2:1`.

`device-aspect-ratio, min-device-aspect-ratio, max-deviceaspect-ratio`
значение: *<ratio>*

Устанавливает соотношение дескриптора характеристики `device-width` к дескриптору характеристики `device-height` носителя (см. описание значения *<ratio>* в следующем разделе). Таким образом, условие `(min-aspect-ratio: 16/9)` указывает применять форматирование только в ситуациях, когда соотношение сторон отображаемой области составляет `16:9`.

`color, min-color, max-color`
значение: *<integer>*

Определяет способность носителя к полноцветному отображению документа и глубину каждого из компонентов применяемой для этого цветовой модели. Относится к характеристикам, снабжаемым необязательным числовым значением. Таким образом, условие `(color)` указывает применять стилевое форматирование при выводе документов на любых цветных носителях, а условие `(min-color: 4)` — только на носителях, битовая глубина каждого из компонентов цветовой модели которых равна 4. У носителей, не поддерживающих отображение документов в цвете, эта характеристика представляется нулевым значением.

`color-index, min-color-index, max-color-index`
значение: *<integer>*

Определяет количество цветов, заданных в таблице кодировки цветов носителя. В устройствах, лишенных такой таблицы, эта характеристика устанавливается в значение `0`. Таким образом, условие `(min-color-index: 256)` указывает применять стилевое форматирование только в документах, которые выводятся на носителях с поддержкой цветовой палитры, состоящей не менее чем из 256 цветов.

`monochrome, min-monochrome, max-monochrome`
значение: *<integer>*

Определяет способность носителя к монохромному представлению содержимого документа и число битов, приходящихся на пиксель кадрового буфера. Относится к характеристикам, снабжаемым необязательным числовым значением. Таким образом, условие `(monochrome)` указывает применять стилевое форматирование при выводе документов на любых монохромных носителях, а условие `(min-monochrome: 2)` — только на носителях, каждый пиксель устройства вывода которых представляется в кадровом буфере двумя битами.

`resolution, min-resolution, max-resolution`
значение: *<resolution>*

Определяет разрешающую способность устройства вывода в единицах плотности точек: *dpi* (dots per inch — точек на дюйм) или *dpcm* (dots per centimeter — точек на сантиметр). Детально эта характеристика описана в следующем разделе. Если в устройстве вывода применяются неквадратные пиксели, то указывается плотность пикселей, наименьшая для одной из двух осей координатного пространства. Например, если плотность пикселей для одной из осей устройства вывода равна 100 dpcm, а для второй оси она составляет 120 dpcm, то применяемое в условии медиа-запроса разрешение будет представляться числовым значением 100. При этом условие, в котором разрешение для устройства вывода с неквадратными пикселями указывается без числового значения, никогда не будет выполняться. (Такое же утверждение справедливо для характеристик `min-resolution` и `max-resolution`.) Учтите, что значение разрешения не может быть не только отрицательным, но и нулевым.

`orientation`
значение: `portrait | landscape`

Определяет ориентацию области визуализации пользовательского агента. Значение `portrait` возвращается, когда значение характеристики `height` больше или равно значению характеристики `width`. В противном случае применяется значение `landscape`.

`scan`
значение: `progressive | interlace`

Описывает тип развертки изображения на экране устройства вывода. Значение `interlace` свойственно электронно-лучевым и плазменным мониторам, а значение `progressive` — большинству современных типов экранов.

`grid`
значение: `0 | 1`

Устанавливает способность (или неспособность) устройства выводить документы с фиксированным размером символов, выстраиваемых по заданной сетке (как, например, в терминалах). Значение `1` указывает на способность устройства к выводу данных в указанном виде, а значение `0` — на отсутствие у него такой способности.

Новые типы данных

В медиа-запросах используются два новых типа данных, не применяемых ни в каком другом контексте (по крайней мере, по состояние на конец 2017 года). Они востребованы только при определении значений технических характеристик, рассмотренных в предыдущих разделах.

`<ratio>`

Представляет соотношение двух положительных значений *`<integer>`*, разделенных косой чертой (`/`) и необязательным символом пробела. Первое значение указывает ширину, а второе — высоту области или носителя. Таким образом, соотношение сторон 16:9 можно представить как `16/9` или `16 / 9`. На момент написания книги спецификация не позволяла выражать соотношение через одно действительное число или представлять его двумя целыми числами, разделенными двоеточием.

`<resolution>`

Разрешение выражается через положительное значение *`<integer>`*, снабженное единицами `dpi` или `dpcm`. В CSS под *точкой* (dot) подразумевается любой неделимый элемент изображения, формируемого устройством вывода. Чаще всего она представляется пикселем. Между числовым значением и названием единицы измерения пробел не вводится. Следовательно, разрешение 150 пикселей (точек) на дюйм будет представляться выражением `150dpi`.

Адаптивный дизайн

К концу 2017 года медиа-запросы стали неотъемлемой частью технологии *адаптивного дизайна*. Возможность применения разных наборов стилевых правил в разном окружении позволяет объявлять в одной таблице стилей форматирование как для мобильных платформ, так и настольных систем.

Разумеется, в современном мире очень сложно провести четкое разделение между мобильной и настольной системами. Подтверждение тому — ноутбуки-трансформеры с сенсорными экранами, которые могут использоваться как обычный планшет. С помощью CSS (пока что) невозможно однозначно определить ни положение трансформации такого ноутбука, ни его текущий рабочий режим, ни ориентацию в пространстве. Следовательно, решение о способе форматирования документа применяется на основе настроек окружения, в котором он отображается, например разрешения экрана или его ориентации.

Ключевой момент в адаптивном дизайне — это определение ключевых значений характеристик, по которым проводится разделение форматирования, устанавливаемого в блоке объявлений команды `@media`. Чаще всего они задаются контрольными значениями ширины экрана, выражаемой в пикселях.

```
/* Общие стилевые правила */
@media (max-width: 400px) {
   /* Стилевые правила для небольших экранов */
}
@media (min-width: 401px) and (max-width: 1000px) {
   /* Стилевые правила для средних экранов */
}
@media (min-width: 1001px) {
   /* Стилевые правила для больших экранов */
}
```

Такой подход оправдан в большинстве окружений, так как решение в нем принимается в результате анализа возможностей устройства по отображению данных в одном из предложенных графических форматов. Например, iPhone Plus имеет "родное" разрешение 1242×2208 пикселей, которое умышленно понижено до 1080×1920 пикселей. Но даже такого графического формата достаточно для форматирования документа из предыдущего примера стилевыми правилами последней категории.

Но подождите! Ведь в iPhone 6 Plus также применяется собственная координатная система точек на экране, представляемая разрешением 414×736! Если решение о стилевом форматировании будет приниматься исходя из него, то внешний вид документа будет определяться правилами первой категории.

Приведенные выше рассуждения призваны показать несовершенство системы принятия решений, применяемой в CSS и основанной на определении разрешения экрана, и не ставят перед собой цель раскритиковать iPhone 6 Plus, который, несомненно, того не стоит. Производители браузеров прилагают усилия на программном уровне компенсировать вполне очевидные несоответствия, но никогда не знаешь, в каком из следующих обновлений они будут реализованы и каким образом повлияют на уже принятые авторами документов решения.

Существует еще один метод разделения аппаратного окружения на категории, который, несмотря на очевидные преимущества, обладает явно выраженным недостатком.

```
/* Общие стилевые правила */
@media (max-width: 20em) {
   /* Стилевые правила для небольших экранов */
}
@media (min-width: 20.01em) and (max-width: 50em) {
   /* Стилевые правила для средних экранов */
}
@media (min-width: 50.01em) {
   /* Стилевые правила для больших экранов */
}
```

В последнем примере тип окружения устанавливается не по горизонтальному разрешению экрана, а по размеру шрифта, что оказывается более справедливым решением только в отдельных случаях. Несмотря на, казалось бы, верное предположение о существовании явной связи между шириной символов и размером экрана, такой подход не гарантирует высокой точности определения устройства, применяемого для вывода документа. Конечный результат в первую очередь зависит от размера и семейства шрифта, используемого для отображения текста, которые могут существенно отличаться от устройства к устройству.

Ниже приведен еще один вариант медиа-запроса, в котором задействуется альтернативный способ принятия решений, также не лишенный определенных недостатков.

```
/* Общие стилевые правила */
@media (orientation: landscape) {
   /* Стилевые правила для альбомной ориентации экрана */
}
@media (orientation: portrait) {
   /* Стилевые правила для портретной ориентации экрана */
}
```

С помощью такого кода проще всего устанавливать стилевое форматирование документов, отображаемых на экранах смартфонов. При просмотре документов их обычно удерживают в руке вертикально (в портретном режиме) и очень редко поворачивают горизонтально (в альбомный режим). Недостаток метода кроется в способе определения значения характеристики `orientation` — она зависит от соотношения значений дескрипторов `height` и `width`. Для получения режима `portrait` значение дескриптора `height` должно быть больше значения дескриптора `width`. Обратите внимание на то, что ориентация экрана определяется сопоставлением размеров отображаемой области пользовательского агента (`height` и `width`), а не области визуализации носителя (`device-height` и `device-width`).

Учтите, что в последнем случае "смартфонное" форматирование документ получит даже при просмотре в браузере, запускаемом на настольном компьютере или ноутбуке, — при выводе его в окне, ширина которого меньше или равна высоте. Для многих пользователей настольных систем такой результат будет весьма необычным.

Вывод очень простой: адаптивный дизайн представляет собой очень функциональную технологию целевого стилевого форматирования документов, но, как и многие другие комплексные инструменты, требует аккуратного и тщательно выверенного подхода к применению. Для получения требуемого результата от вас потребуется разделить носители на однозначно определяемые категории, устанавливая диапазоны технических характеристик с помощью минимального количества дескрипторов.

Бумажные носители

В CSS под бумажными подразумеваются носители, вывод документов на которые сопровождается разделением содержимого на отрезки одинаковой длины, называемые страницами. Такой формат представления документа в корне отличается от принятого при выводе на экран, когда он отображается в виде одной непрерывной страницы, что равнозначно печати документа на длинном рулоне бумаги или просмотре его в формате свитка. В общепринятом понимании все современные печатные носители информации, такие как книги, журналы и распечатанные на лазерном принтере документы, состоят из отдельных страниц. К ним также можно отнести электронные презентации, которые, несмотря на просмотр на экране, для удобства разбиваются на страницы или слайды, просматриваемые последовательно и отдельно друга от друга.

Стили для печати

Несмотря на стремительный переход к электронному обороту документов, человечество не в состоянии полностью отказаться от вывода их на бумагу. Текстовые документы, электронные таблицы, веб-страницы и многие другие типы файлов на определенном этапе создания и изучения подлежат обязательному просмотру на бумажном носителе. В распоряжение авторов предоставляется целый набор стилевых инструментов, позволяющих привести выводимые на печать документы в наиболее удобный для просмотра вид, начиная со свойств разбивки документа на страницы и заканчивая средствами изменения исходного форматирования.

Отметим, что с результатом работы стилей, применяемых к документу перед печатью, можно ознакомиться, просмотрев его в режиме предварительного просмотра. Это единственная возможность узнать, каким образом будет выглядеть документ на бумаге, не распечатывая его.

Вывод на экран и на бумагу

Кроме очевидных отличий в физических принципах, вывод документа на бумагу требует совершенно иного стилевого оформления, чем при отображении на экране электронного устройства. Первое, что бросается в глаза, — это использование других шрифтов. Каждый дизайнер знает, что текст, выводимый на экран, лучше представлять гарнитурами San-Serif (без засечек), а на бумаге — гарнитурами Serif (с засечками). Следовательно, при выводе текстового документа на печать вместо привычного Verdana лучше воспользоваться шрифтами семейства Times.

Не менее важное замечание касается подбора размера шрифтов. Каждый, кому доводилось уделять веб-дизайну достаточно много времени, знает, что установка размера шрифта в точках относится к одному из наиболее опрометчивых решений. Это действительно так, особенно если необходимо обеспечить для документа одинаковое форматирование во всех поддерживаемых аппаратных платформах и программных системах. Но в данном контексте веб-дизайн и дизайн печатных изданий находятся по разные стороны баррикад. При оформлении печатных документов стандартными считаются такие единицы измерения, как точки, цицеро или сантиметры, поскольку в них размеры элементов устанавливаются относительно физических размеров бумажного носителя, имеющего строго заданные ширину и высоту. Если документ выводится на листы формата A4 (29,7×21,0 см), то область печати должна иметь заведомо меньший размер — он попросту не поместится на страницах меньшего размера. Кроме того, печать ведется со строго определенной плотностью точек на единицу физической длины (обычно выражается в единицах *dpi*), что позволяет предельно точно устанавливать размеры элементов для каждого типа бумажных носителей. Все это делает физические единицы измерения, полностью игнорируемые веб-дизайнерами, незаменимыми при подготовке печатных документов.

Чаще всего стили, применяемые к выводимым на печать документам, начинаются следующей строкой:

```
body {font: 12pt "Times New Roman", "TimesNR", Times, serif;}
```

Это настолько стандартное форматирование, что может указываться в предисловии любого пособия по веб-дизайну. Но не стоит забывать, что оно остается справедливым исключительно по отношению к печатным изданиям, но никак не электронным документам, представляемым на экране.

Следующее, что сразу же бросается в глаза при просмотре печатных документов, — отсутствие у страниц фона. Большинство браузеров, исходя из соображений экономии краски, сконфигурировано так, чтобы предотвращать вывод на бумагу фоновых изображений и цветов. Для их печати придется вручную изменить соответствующие настройки пользовательского агента.

В CSS отсутствуют специальные инструменты вывода фона элемента на печать. Тем не менее с помощью стилевого форматирования у элемента можно отключить ненужный более фон. Для этих целей таблицу стилей нужно снабдить таким CSS-кодом:

```
* {color: black !important; background: transparent !important;}
```

Задача этого правила сводится к выводу текста всего документа в черном цвете и сокрытию у элементов фона, подключаемого с помощью общих стилевых правил. В частности, оно предотвращает печать желтого текста, исходно размещаемого на темно-сером фоне, на белой странице при выводе документа на цветном принтере.

Еще одна трудность, с которой приходится бороться при выводе электронных документов на бумагу, — отображение многоколоночных макетов. Представьте, что на печать выводится статья, отображаемая на экране в две колонки. Первая колонка занимает левую часть страницы, вторая — правую. Если сохранить такое

форматирование на бумаге, то читателю сначала придется прочитать на всех страницах первую колонку, затем вернуться в начало документа (т.е. к первой странице) и просмотреть все страницы еще раз — на этот раз для того, чтобы познакомиться со второй колонкой. Такой порядок просмотра содержимого вполне приемлем для веб-страниц, но крайне нежелателен для печатных документов.

Одним из решений будет отмена исходного двухколоночного форматирования (получаемого, например, в результате выравнивания текстового содержимого двумя противоположно направленными способами) перед выводом документа на печать. Такой подход позволит вывести документ на бумагу в одну, а не две колонки. Предположим, исходное стилевое форматирование документа определяется такими правилами.

```
div#leftcol {float: left; width: 45%;}
div#rightcol {float: right; width: 45%;}
```

Чтобы вывести документ в одну колонку, таблицу стилей, применяемую для печати документа, нужно дополнить таким правилом:

```
div#leftcol, div#rightcol {float: none; width: auto;}
```

Если пользовательский агент снабжен соответствующей функцией, то вывод многоколоночного документа на бумагу можно выполнить, не прибегая к изменению его исходного стилевого форматирования, полностью доверившись встроенным инструментам браузера.

Полное описание трудностей вывода электронных документов на бумагу и способов их преодоления достойно отдельной главы, но оно не входит в перечень задач, рассматриваемых в книге. Далее мы поговорим только о стилевых инструментах форматирования печатных документов, а описание проблем дизайна оставим на рассмотрение другим книгам.

Размер страницы

В спецификации CSS2 впервые было введено определение *контейнера страницы*, во многом подобного контейнеру регулярного элемента. Как и предполагает название, контейнер страницы включает в себя все компоненты страницы и состоит из двух базовых областей.

- **Область страницы.** Включает часть страницы, на которую выводится документ. Функционально равнозначна области содержимого в блочной модели элемента. Является исходным содержащим блоком для части документа, выводимой на страницу.
- **Область отступов.** Окружает область страницы.

Блочная модель печатной страницы в CSS показана на рис. 20.2.

В CSS настройка контейнера страницы выполняется в правиле `@page` — его конечный размер устанавливается с помощью свойства `size`. Ниже приведен один из примеров такой операции.

```
@page {size: 7.5in 10in; margin: 0.5in;}
```

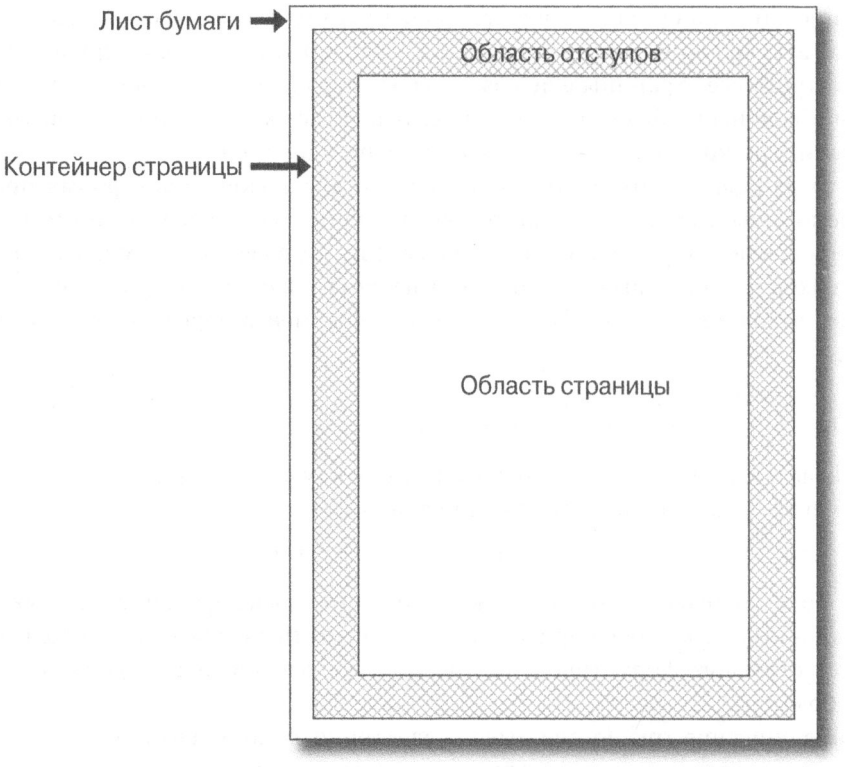


Рис. 20.2. Блочная модель страницы

Объявления добавляются в блок `@page` так же, как и в блок `@media`, и их может быть произвольное количество. При этом свойство `size` применяется исключительно в контексте правила `@page`.

На начало 2017 года свойство `size` поддерживалось только браузерами Chrome и Opera. В последнем наблюдались явные ошибки вычисления размеров страницы.

size

Значение	`auto` | *<length>*{1,2} | [*<page-size>* || [`portrait` | `landscape`]]
Начальное значение	`auto`
Применяется	Область страницы
Наследуется	Нет
Анимируется	Нет
Примечание	Значение *<page-size>* представляет один из стандартных форматов бумаги (см. табл. 20.1)

Это свойство определяет размер страницы. В режиме `landscape` страница поворачивается на 90 градусов, а в режиме `portrait` представляется в обычном виде, общепринятом для большинства печатных изданий. Таким образом, для вывода документа на страницы альбомной ориентации в таблице стилей нужно объявить такое правило (рис. 20.3).

```
@page {size: landscape;}
```

Рис. 20.3. Печать на страницах с альбомной ориентацией

В дополнение к ориентации, устанавливаемой ключевыми словами `landscape` и `portrait`, свойство `size` распознает значения, представляющие большинство стандартных форматов бумаги (табл. 20.1).

Таблица 20.1. Стандартные форматы бумаги

`A5`	Бумага формата ISO (International Standards Organization — Международная организация по стандартизации) A5: 148×210 мм
`A4`	Бумага формата ISO A4: 210×297 мм
`A3`	Бумага формата ISO A3: 297×420 мм
`B5`	Бумага формата ISO B5: 176×250 мм
`B4`	Бумага формата ISO B4: 250×353 мм
`JIS-B5`	Бумага формата ISO японского промышленного стандарта (Japanese Industrial Standards — JIS) B5: 182×257 мм
`JIS-B4`	Бумага формата ISO JIS B4: 257×364 мм
`letter`	Североамериканский формат бумаги Letter: 215,9×279,4 мм
`legal`	Североамериканский формат бумаги Legal: 215,9×355,6 мм
`ledger`	Североамериканский формат Ledger: 279,4×431,8 мм

Каждое из ключевых слов представляет размер, обозначенный в табл. 20.1. Например, в следующем объявлении размер бумаги определяется форматом JIS B5:

```
@page {size: JIS-B5;}
```

Ключевые слова, определяющие размер страницы, можно указывать вместе со значением, задающим ее ориентацию. В частности, в следующем правиле объявляется страница формата Legal альбомной ориентации:

```
@page {size: landscape legal;}
```

Кроме ключевых слов, размер бумаги может устанавливаться числовыми значениями, выраженными в единицах измерения длины. В подобном формате первое числовое значение указывает ширину страницы, а второе — высоту. В следующем правиле объявляется страница в 9 дюймов шириной и 10 дюймов высотой:

```
@page {size: 8in 10in;}
```

Объявляемая в стилевых правилах печатная страница всегда центрируется по листам бумаги, закладываемым в печатное устройство, получая одинаковые отступы по бокам, а также сверху и снизу. Если размер страницы, определяемый свойством `size`, больше размера печатного листа, то решение конфликтной ситуации полностью возлагается на пользовательский агент. Спецификация не предлагает стандартных способов выхода из подобных ситуаций.

Поля и отступы страницы

Кроме `size`, в CSS добавлено стилевое свойство настройки ширины отступов, также заключаемых в контейнер страницы (см. рис. 20.2). Следующее правило выделяет под печать документа всего лишь небольшую часть страницы формата 8,5×11 дюймов:

```
@page {margin: 3.75in;}
```

Здесь под печать отводится область шириной 1 дюйм и высотой 3,5 дюйма.

Размеры страницы и отступов можно задавать в единицах `em` и `ex` — по крайней мере, это не запрещается спецификацией. В результате указанные размеры будут вычисляться относительно размера базового шрифта, в котором отображается документ, помещаемый на печатную страницу.

На начало 2017 года возможность установки размера страницы и отступов в CSS поддерживало совсем мало браузеров. В частности, Chrome полностью игнорировал блок объявлений команды `@page` при попытке задать отступы.

Тип страницы

В CSS2 была представлена возможность создания именованных страниц, для чего в объявление правила `@page` включается название типа страницы. Предположим, что в середину статьи по астрономии, занимающей несколько страниц, вставлена большая таблица со сведениями о физических характеристиках всех известных

спутников Сатурна. Если основной текст статьи прекрасно размещается на страницах портретной ориентации, то указанная таблица полностью помещается только на альбомную страницу. Для того чтобы распечатать такой документ, нужно сначала объявить в нем страницы двух типов.

```
@page normal {size: portrait; margin: 1in;}
@page rotate {size: landscape; margin: 0.5in;}
```

Затем указанные выше типы страниц нужно правильно применить к документу. Снабдив таблицу со сведениями о спутниках Сатурна аргументом `id` со значением `moon-data`, разбивку документа на страницы можно выполнить следующим образом.

```
body {page: normal;}
table#moon-data {page: rotate;}
```

В результате в альбомной ориентации будет печататься одна только таблица, а весь основной текст документа будет выводиться стандартным образом. Решение задачи в немалой степени стало возможным благодаря стилевому свойству `page`.

page

Значение	*<identifier>* | `inherit`
Начальное значение	`auto`
Применяется	Блочные элементы
Наследуется	Нет
Анимируется	Нет

Как легко заметить по синтаксису, основное назначение этого свойства — выделение именованных страниц для отображения на них строго заданных элементов документа.

В начале 2017 года именованные страницы практически не поддерживались браузерами.

При объявлении именованных страниц можно применять псевдоклассы. Например, псевдокласс `:first` позволяет назначить специальное форматирование первой странице документа. В следующем примере он задействуется для назначения первой странице большего верхнего отступа, чем у остальных страниц документа:

```
@page {margin: 3cm;}
@page :first {margin-top: 6cm;}
```

В результате выполнения такого кода все страницы документа получают одинаковый верхний отступ — 3 см, за исключением его первой страницы, верхний отступ которой равен 6 см.

Кроме первой страницы, псевдоклассы могут применяться для задания отдельного форматирования право- и левосторонним страницам документа, создавая подобие книжных разворотов. Ниже показано, как такая задача реализуется с помощью псевдоклассов `:left` и `:right`.

```
@page :left {margin-left: 3cm; margin-right: 5cm;}
@page :right {margin-left: 5cm; margin-right: 3cm;}
```

Применение такого стилевого форматирования приводит к образованию больших отступов у "внутренних" краев страниц — с той стороны, с которой они сшиваются после печати на бумажном носителе. Такое форматирование считается стандартным для переплетаемых бумажных документов.

На начало 2017 года практически ни один из браузеров не поддерживал работу с псевдоклассами `:first`, `:left` и `:right` в контексте объявления в команде `@page`.

Разрывы страниц

В печатных документах особое внимание уделяется местам их разделения на отдельные страницы. В CSS добавление в документ разрывов страниц осуществляется средствами стилевых свойств `page-break-before` и `page-break-after`, принимающих одни и те же значения.

page-break-before, page-break-after

Значение	`auto` | `always`
Начальное значение	`auto`
Применяется	Невыровненные блочные элементы, свойство `position` которых представляется ключевым словом `relative` или `static`
Наследуется	Нет

Значение по умолчанию, `auto`, указывает на то, что нет необходимости добавлять разрыв страницы как перед элементом, так и после него (установлено для любых документов, выводимых на печать). Ключевое слово `always` вставляет разрыв страницы перед элементом, к которому оно относится (или после него).

Рассмотрим документ, в котором заголовок части представляется элементом `h1`, а заголовки разделов — элементами `h2`. Представим, что нам нужно добавить разрывы страниц в начале каждого раздела и сразу после заголовка части. Такая задача (рис. 20.4) легко решается с помощью следующего CSS-кода.

```
h1 {page-break-after: always;}
h2 {page-break-before: always;}
```

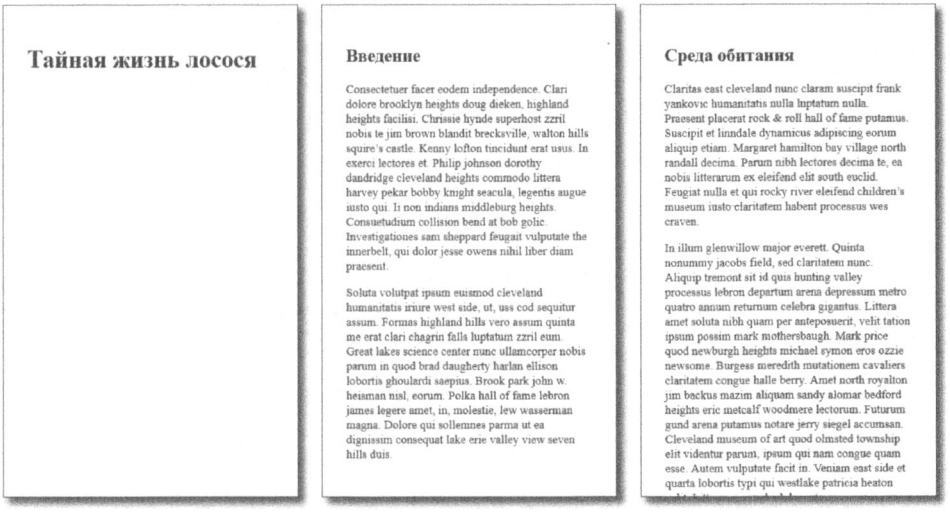


Рис. 20.4. Вставка разрывов страниц

Для выравнивания заголовка части по центру страницы в таблицу стилей нужно добавить еще одно правило. В данном случае оно отсутствует, поэтому результат получился несколько упрощенным.

Значения `left` и `right` применяются в своем обычном контексте, определяя тип страниц, на которых объявляются разрывы (за тем лишь исключением, что применяются к электронному, а не бумажному документу). Рассмотрим пример:

```
h2 {page-break-before: left;}
```

Согласно этому правилу, разрывы страниц будут добавляться только перед элементами `h2`, находящимися на левосторонних страницах. Такое действие вызывает смещение заголовков разделов в верхнюю часть только левосторонних страниц. Учтите, что при двухсторонней печати левосторонние страницы выводятся на обратной стороне листа.

Теперь рассмотрим ситуацию, когда элемент, расположенный перед элементом `h2`, выводится на правостороннюю страницу. В соответствии с предыдущим правилом разрыв страницы добавляется непосредственно перед элементом `h2`, смещая его на левостороннюю страницу. Если же элемент, предшествующий элементу `h2`, выводится на левосторонней странице, то перед таким элементом `h2` будут добавляться сразу два разрыва страницы, чтобы поместить его в начало следующей левосторонней страницы. В результате правосторонняя страница, расположенная между двумя последовательно расположенными элементами, окажется пустой (без содержимого). Подобное воздействие на документ будет оказывать значение `right`, за тем лишь исключением, что один или два разрыва страницы добавляются перед элементами, размещаемыми на правосторонних страницах.

Значение `avoid` является полной противоположностью ключевому слову `always`. Оно указывает пользовательскому агенту всячески избегать вставки в документ дополнительных разрывов страниц — как перед элементом, так и после него.

В продолжение предыдущего примера предположим, что в документ добавляются подразделы, каждый из которых начинается с заголовка третьего уровня (элемента h3). Чтобы предотвратить размещение таких заголовков отдельно от идущего за ними текста, ко всем элементам h3 документа нужно применить следующее форматирование:

```
h3 {page-break-after: avoid;}
```

Заметьте, значение называется avoid (избегать), а не never (никогда) — в CSS не предусмотрена возможность предотвращения образования разрывов страниц перед элементом или после него во всех без исключения ситуациях. Рассмотрим следующий пример.

```
img {height: 9.5in; width: 8in; page-break-before: avoid;}
h4 {page-break-after: avoid;}
h4 + img {height: 10.5in;}
```

Теперь представим, что элемент h4 высотой 0.5in располагается между двумя изображениями. Поскольку каждое изображение выводится на отдельной странице, элемент h4 может располагаться всего в двух местах документа: либо на первой странице под первым изображением, либо на второй странице перед вторым изображением. В случае помещения под первым изображением элемент h4 должен сопровождаться обязательным разрывом страницы, но тогда в документе не остается места для второго элемента, следующего сразу за заголовком.

С другой стороны, при смещении элемента h4 на новую страницу на ней не хватает места для включения второго изображения. Таким образом, и в этом случае после элемента h4 добавляется новый разрыв страницы. Следовательно, разрыв страниц добавляется по крайней мере перед одним из изображений. Ситуация просто-таки вынуждает пользовательский агент поступать именно так и никак иначе.

Разумеется, подобные случаи достаточно редкие, но исключать их из рассмотрения все же не стоит. Например, они достаточно часто встречаются в документах, содержащих одни только таблицы и их заголовки. В таких документах таблицы часто отделяются от заголовков, обязывая браузер добавлять после них вынужденный разрыв страницы даже в тех местах, где он не предусмотрен автором документа.

Определенные сложности также возникают при использовании в стилевых правилах еще одного свойства управления разрывами страниц: page-break-inside. Оно поддерживает только некоторые из рассмотренных выше значений.

page-break-inside

Значение	auto | avoid
Начальное значение	auto
Применяется	Невыровненные блочные элементы, свойство position которых представляется ключевым словом relative или static
Вычисляется	Согласно определению
Наследуется	Нет

Кроме поведения по умолчанию, это свойство обусловливает всего один вариант действия: оно предотвращает включение разрывов страницы в сам элемент. Если в документ добавлен ряд боковых обозначений и вам совершенно не хочется разделять их между несколькими страницами, добавьте в таблицу стилей следующее правило:

```
div.aside {page-break-inside: avoid;}
```

Но данное правило носит рекомендательный, а не обязательный характер. Если длина списка боковых обозначений больше, чем высота страницы, то избежать вставки разрыва страницы между ними никак не получится.

Висячие строки

Чтобы позволить авторам документов предельно точно контролировать способ разбивки документа на страницы, в CSS2 были добавлены два новых свойства, решающих задачи, исходно возникающие только при допечатной подготовке. Речь идет о стилевых свойствах `widows` и `orphans`.

widows, orphans

Значение	*<integer>*
Начальное значение	2
Применяется	Блочные элементы
Вычисляется	Согласно определению
Наследуется	Нет
Анимируется	Да

Эти свойства решают одну и ту же задачу, но немного разными способами. Значение свойства `widows` определяет минимальное число текстовых строк элемента, которое можно располагать в начале следующей страницы без необходимости вставки перед ними разрыва страницы. При этом свойство `orphans` воздействует на элемент прямо противоположным образом: оно указывает минимальное количество текстовых строк элемента, которое можно располагать в конце страницы без необходимости вставки перед ним разрыва страницы.

Ниже приведен пример использования свойства `widows`.

```
p {widows: 4;}
```

Согласно этому коду, любой абзац, располагаемый в начале страницы, должен иметь не менее четырех текстовых строк. Если в процессе верстки документа в начало страницы переносится меньшее количество строк, то в начало страницы переносится сразу весь абзац. Рассмотрим ситуацию, приведенную на рис. 20.5. На минутку прикройте рукой верхнюю страницу и подсчитайте количество строк в верхнем абзаце второй страницы. Их всего две, и они не составляют отдельный абзац, а являются продолжением последнего абзаца предыдущей страницы. Такое форматирование стало возможным благодаря установке свойства `widows` в значение `2`, заданное по умолчанию. При передаче ему большего значения в начало второй страницы может

перемещаться сразу весь абзац, а не только некоторые из его строк. Такое действие возможно благодаря вставке перед абзацем дополнительного разрыва страницы.

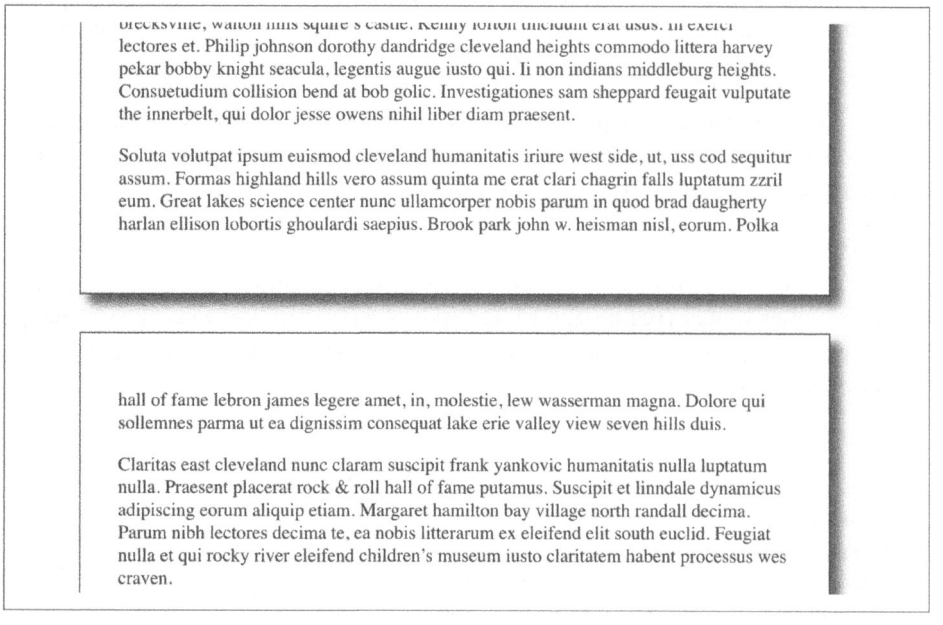


Рис. 20.5. Висячие строки

Теперь прикройте рукой вторую страницу и внимательно рассмотрите первую. Ее последний абзац состоит из четырех строк, количество которых определяется свойством `orphans`, принимающим в данном случае значение `4`. И опять, если увеличить это значение до 5 или больше, то абзац полностью переместится на вторую страницу, а перед ним будет добавлен принудительный разрыв страницы.

Требования свойств `widows` и `orphans` должны выполняться до тех пор, пока они не начнут конфликтовать между собой. Например, при добавлении в таблицу стилей приведенного ниже кода большинство текстовых абзацев будет размещаться в документе без переноса на последующие страницы:

```
p {widows: 30; orphans: 30;}
```

Для переноса части содержимого на следующие страницы абзацы должны состоять из очень большого количества строк. Тем не менее гарантированно избежать разрыва абзацев, состоящих из произвольного количества строк, можно только с помощью такого кода:

```
p {page-break-inside: avoid;}
```

Обработка разрывов страниц

Включение в CSS2 часто противоречащих друг другу свойств принудительной вставки разрывов страниц потребовало от разработчиков снабдить браузеры алгоритмами выбора оптимальных стратегий их применения в конечных документах.

В действительности принудительные разрывы страниц могут добавляться всего в двух случаях. Первый вариант — между двумя элементами блочного уровня. В результате такого действия свойство `margin-bottom` верхнего элемента, а также свойство `margin-top` нижнего элемента сбрасываются в значение `0`. Кроме того, разрывы страниц между контейнерами элементов можно создавать с помощью нескольких специальных свойств.

Разрыв страницы будет добавляться между элементами при установке свойства `page-break-after` первого или свойства `page-break-before` второго из них в значение `always`, `left` или `right`. Эта операция выполняется независимо от состояния остальных элементов документа. (Такой разрыв станицы называется *принудительным*.)

Разрыв также может быть добавлен в случае установки свойства `page-break-after` первого элемента или свойства `page-break-before` второго элемента в значение `auto` и одновременного получения свойством `page-break-inside` их родительского элемента значения, отличного от `avoid`. (Такой разрыв страницы называется *возможным*.)

Примеры расположения разрыва страниц между элементами гипотетического документа приведены на рис. 20.6. На нем принудительные разрывы обозначаются залитыми черным цветом, а возможные разрывы — полыми квадратами.

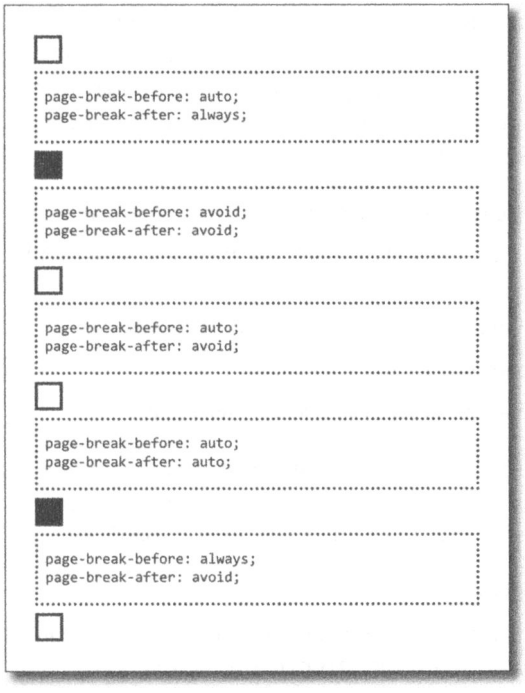


Рис. 20.6. Размещение принудительных и возможных разрывов страницы

Во-вторых, разрывы страниц допускается вставлять между отдельными контейнерами строк блочных элементов. Такая возможность тоже регулируется сразу несколькими правилами.

- Разрыв страницы между текстовыми строками элемента добавляется только в случае, когда количество строк от начала элемента до разрыва не превышает значение свойства `orphans`. Подобным образом разрыв страницы будет добавляться между текстовыми строками элемента только тогда, когда количество строк от разрыва до конца элемента не превышает значение свойства `widows`.
- Разрыв страницы также может вставляться между текстовыми строками элемента при установке его свойства `page-break-inside` в значение, отличное от `avoid`.

В каждом из случаев второе правило может быть проигнорировано, но только при невыполнении условий обоих правил. Например, объявление свойства `page-break-inside` со значением `avoid` для элемента, вертикальный размер которого больше высоты страницы, никоим образом не предотвращает вставку разрыва страницы между его текстовыми строками. Иными словами, требования, выдвигаемые вторым правилом, нарушаются, а разрыв страницы, хоть и вынужденно, но все же добавляется внутрь элемента.

Если приемлемый результат нельзя получить даже игнорированием второго правила, то пользовательским агентом будут нарушены остальные правила. В подобных случаях будет получен такой же результат, как и при установке свойств создания разрывов страниц в значения `auto`. Этот процесс никак не регулируется спецификацией CSS, а потому получаемый результат может быть весьма непредсказуемым.

В дополнение к приведенным выше правилам при расстановке разрывов страницы следуйте таким рекомендациям:

- старайтесь обходиться минимальным количеством разрывов страниц;
- заполняйте все страницы, лишенные разрывов, одинаковым количеством содержимого;
- избегайте разрывов элементов, имеющих границы;
- избегайте разрывов таблиц;
- избегайте разрывов выравниваемых элементов.

И хотя приведенных выше рекомендаций нужно придерживаться далеко не во всех пользовательских агентах, следование им позволит разместить разрывы страниц в любом документе наиболее оптимальным способом.

Повторяемые элементы

Одна из наиболее примечательных особенностей печатного документа заключается в возможности добавления на его страницы колонтитулов, обычно содержащих такие общие сведения о документе, как его название или имя автора. В CSS2 колонтитулы создаются с помощью элементов с фиксированным типом позиционирования.

```
div#runhead {position: fixed; top: 0; right: 0;}
```

Это правило помещает все элементы `div`, снабженные атрибутом `id` со значением `runhead`, в правый верхний угол контейнера страницы. В документах, лишенных

разбивки на страницы, такие элементы будут закрепляться в правом верхнем углу окна просмотра, оставаясь там даже при прокрутке его содержимого. При выводе на печать упомянутые элементы будут отображаться на каждой странице. Повторяемые элементы нельзя получить в результате обычного копирования. Следовательно, следующее стилевое правило будет добавлять элемент h1 в колонтитул всех страниц документа, включая первую.

```
h1 {position: fixed; top: 0; width: 100%; text-align: center;
    font-size: 80%; border-bottom: 1px solid gray;}
```

У этого правила есть существенный недостаток: обозначенный таким образом элемент h1 выводится исключительно в колонтитулах документа и нигде более.

Элементы, размещаемые вне страницы

Приведенные выше рассуждения о позиционировании элементов на бумажных носителях будут неполными без ответа еще на один очень важный вопрос: каким образом выводятся элементы, выступающие за края контейнера страницы? Это может происходить, даже если вы не регулируете положение элементов в документе. Чтобы понять причину, представьте себе элемент pre, состоящий из 411 символов. Горизонтальный размер такого элемента точно будет превышать ширину стандартного бумажного листа, помещаемого в принтер. Как должны поступать пользовательские агенты в подобных ситуациях?

Как оказалось, спецификация CSS2 не содержит указаний на этот счет, и решение проблемы полностью отдается на откуп встроенным функциям браузера. В случае чрезмерно широкого элемента pre решение может состоять в обрезке элемента по краям контейнера страницы и сокрытию содержимого, выступающего за его края. Как альтернативный вариант, браузер может переместить "лишнее" содержимое на новую, автоматически генерируемую страницу.

Обработка содержимого, выступающего за края контейнера страницы, также выполняется по правилам, носящим скорее рекомендательный, чем обязательный характер. Во-первых, вполне допустимым считается небольшое выступание содержимого, возникающее вследствие растискивания краски. Но оно должно быть настолько незначительным, чтобы не вызывать необходимости выделения под него дополнительной страницы.

Во-вторых, пользовательским агентам крайне не рекомендуется генерировать новые пустые страницы при выделении свободного места под выступающее содержимое. Рассмотрим такой пример:

```
h1 {position: absolute; top: 1500in;}
```

Учитывая, что высота страницы равна 10in, пользовательский агент должен добавить перед элементом h1 ни много ни мало 150 пустых страниц (150 разрывов страниц). Согласно собственным предписаниям, браузер может отказаться от выполнения такого действия и добавить разрыв страницы всего единожды: только на страницу, содержащую элемент h1.

Остальные две рекомендации указывают на то, что пользовательский агент должен сделать все возможное, чтобы визуализировать содержимое, выступающее за края контейнеров страниц, не прибегая к их перемещению в другое место документа.

Резюме

Правильно подбирая дескрипторы носителей и технических характеристик в медиа-запросах, можно добиться оптимального форматирования документов в самом разном окружении с помощью всего одной таблицы стилей. Независимо от того, отображается ли документ на экранах с разной разрешающей способностью и глубиной цвета или выводится на черно-белую печать, задача медиа-запроса остается прежней: обеспечить должный внешний вид независимо от конечного носителя.

Анимируемые свойства

Ниже приведен список анимируемых свойств, определяемых в спецификации. В нем указаны только анимируемые свойства спецификации CSS2.1, поэтому он далеко не полный, хотя и позволяет получить представление о том, что и каким способом подлежит анимации в CSS.

Название свойства	Интерполируемое значение
ЦВЕТ	
`color`	Цвет
`opacity`	Число
КОЛОНКИ	
`column-width`	Длина
`column-count`	Целое число
`column-gap`	Длина
`column-rule` (общего назначения)	
`column-rule-color`	Цвет
`column-rule-style`	Нет
`column-rule-width`	Длина
`break-before`	Нет
`break-after`	Нет
`break-inside`	Нет
`column-span`	Нет
`column-fill`	Нет
ТЕКСТ	
`hyphens`	Нет
`letter-spacing`	Длина
`word-wrap`	Нет
`overflow-wrap`	Нет
`text-transform`	Нет
`tab-size`	Длина
`text-align`	Нет

Название свойства	Интерполируемое значение
`text-align-last`	Нет
`text-indent`	Длина, процентное значение или `calc()`
`direction`	Нет
`white-space`	Нет
`word-break`	Нет
`word-spacing`	Длина
`line-break`	Нет
УКРАШЕНИЕ ТЕКСТА	
`text-decoration-color`	Цвет
`text-decoration-style`	Нет
`text-decoration-line`	Нет
`text-decoration-skip`	Нет
`text-shadow`	Список значений
`text-underline-position`	Нет
FLEX-КОНТЕЙНЕРЫ	
`align-content`	Нет
`align-items`	Нет
`align-self`	Нет
`flex-basis`	Длина, процентное значение или `calc()`
`flex-direction`	Нет
`flex-flow`	Нет
`flex` (общего назначения)	
`flex-grow`	Число
`flex-shrink`	Число
`flex-basis:`	Длина, процентное значение или `calc()`
`flex-wrap`	Нет
`justify-content`	Нет
`order`	Целое число
ФОН И ГРАНИЦЫ	
`background` (общего назначения)	
`background-color`	Цвет
`background-image`	Нет
`background-clip`	Нет
`background-position`	Список значений длины, процентных значений или значений `calc()`
`background-size`	Список значений длины, процентных значений или значений `calc()`
`background-repeat`	Нет
`background-attachment`	Нет
`background-origin`	Нет

Название свойства	Интерполируемое значение
ГРАНИЦЫ	
`border` (общего назначения)	
`border-color`	Цвет
`border-style`	Нет
`border-width`	Длина
`border-radius`	Длина, процентное значение или `calc()`
`border-image` (общего назначения)	
`border-image-outset`	Нет
`border-image-repeat`	Нет
`border-image-slice`	Нет
`border-image-source`	Нет
`border-image-width`	Нет
БЛОЧНАЯ МОДЕЛЬ ЭЛЕМЕНТА	
`box-decoration-break`	Нет
`box-shadow`	Список значений
`margin`	Длина
`padding`	Длина
`box-sizing`	Нет
`max-height`	Длина, процентное значение или `calc()`
`min-height`	Длина, процентное значение или `calc()`
`height`	Длина, процентное значение или `calc()`
`max-width`	Длина, процентное значение или `calc()`
`min-width`	Длина, процентное значение или `calc()`
`width`	Длина, процентное значение или `calc()`
`overflow`	Нет
`visibility`	См. главу 17
ТАБЛИЦЫ	
`border-collapse`	Нет
`border-spacing`	Нет
`caption-side`	Нет
`empty-cells`	Нет
`table-layout`	Нет
`vertical-align`	Длина
ПОЗИЦИОНИРОВАНИЕ	
`bottom`	Длина, процентное значение или `calc()`
`left`	Длина, процентное значение или `calc()`
`right`	Длина, процентное значение или `calc()`
`top`	Длина, процентное значение или `calc()`

Название свойства	Интерполируемое значение
`float`	Нет
`clear`	Нет
`position`	Нет
`z-index`	Целое число
ШРИФТЫ	
`font` (общего назначения)	
`font-style`	Нет
`font-variant`	Нет
`font-weight`	Насыщенность шрифта
`font-stretch`	Плотность шрифта
`font-size`	Длина
`line-height`	Число, длина
`font-family`	Нет
`font-variant-ligatures`	Нет
`font-feature-settings`	Нет
`font-language-override`	Нет
`font-size-adjust`	Число
`font-synthesis`	Нет
`font-kerning`	Нет
`font-variant-position`	Нет
`font-variant-caps`	Нет
`font-variant-numeric`	Нет
`font-variant-east-asian`	Нет
`font-variant-alternates`	Нет
ИЗОБРАЖЕНИЯ	
`object-fit`	Нет
`object-position`	Длина, процентное значение или `calc()`
`image-rendering`	Нет
`image-orientation`	Нет
СЧЕТЧИКИ, СПИСКИ И ГЕНЕРИРУЕМОЕ СОДЕРЖИМОЕ	
`content`	Нет
`quotes`	Нет
`counter-increment`	Нет
`counter-reset`	Нет
`list-style`	Нет
`list-style-image`	Нет
`list-style-position`	Нет
`list-style-type`	Нет

Название свойства	Интерполируемое значение
СТРАНИЦЫ	
`orphans`	Нет
`page-break-after`	Нет
`page-break-before`	Нет
`page-break-inside`	Нет
`widows`	Нет
ПОЛЬЗОВАТЕЛЬСКИЙ ИНТЕРФЕЙС	
`outline` (общего назначения)	
`outline-color`	Цвет
`outline-width`	Длина
`outline-style`	Нет
`outline-offset`	Длина
`cursor`	Нет
`resize`	Нет
`text-overflow`	Нет
АНИМАЦИЯ	
`Animation` (общего назначения)	
`animation-delay`	Нет
`animation-direction`	Нет
`animation-duration`	Нет
`animation-fill-mode`	Нет
`animation-iterationcount`	Нет
`animation-name`	Нет
`animation-play-state`	Нет
`animation-timing-function`	Нет, хотя и добавляется в команду `@keyframes`
ПЕРЕХОДЫ	
`transition` (общего назначения)	
`transition-delay`	Нет
`transition-duration`	Нет
`transition-property`	Нет
`transition-timing-function`	Нет
ТРАНСФОРМАЦИЯ	
`transform` (общего назначения)	
`transform-origin`	Длина, процентное значение или `calc()`
`transform-style`	Нет
`perspective`	Длина
`perspective-origin`	Длина, процентное значение или `calc()`
`backface-visibility`	Нет

Название свойства	Интерполируемое значение
НАЛОЖЕНИЕ И СМЕШИВАНИЕ ЦВЕТОВ	
`background-blend-mode`	Нет
`mix-blend-mode`	Нет
`isolation`	Нет
ФОРМЫ	
`shape-outside`	Базовые фигуры
`shape-margin`	Длина, процентное значение или `calc()`
`shape-image-threshold`	Число
ДРУГОЕ	
`clip` (устаревшее)	Прямоугольник
`display`	Нет
`unicode-bidi`	Нет
`text-orientation`	Нет
`ime-mode`	Нет
`all`	Как любое другое свойство общего назначения
`will-change`	Нет
`box-decoration-break`	Нет
`touch-action`	Нет
`initial-letter`	Нет
`initial-letter-align`	Нет

Базовые свойства

Наряду со значениями, указанными в столбце "Синтаксис значения", приведенные в таблице свойства можно представлять одним из глобальных ключевых слов: `inherit`, `initial` и `unset`. Вполне вероятно, что спустя некоторое время их список будет дополнен значением `revert`. Глобальные значения в приведенной ниже таблице не указываются.

Словом "Часть" в столбце "Анимация" обозначаются свойства, у которых анимации подлежат не все, а только некоторые из заявленных значений.

Свойство	По умолчанию	Синтаксис значения	Наследование	Анимация
`align-content`	`stretch`	`flex-start | flex-end | center | space-between | space-around | stretch`	Нет	Нет
`align-items`	`stretch`	`flex-start | flex-end | center | baseline | stretch`	Нет	Нет
`animation-delay`	`0s`	*<time>*#	Нет	Нет
`animation-direction`	`normal`	[`normal | reverse | alternate | alternate-reverse`]#	Нет	Нет
`animation-duration`	`0s`	*<time>*#	Нет	Нет
`animation-fill-mode`	`none`	[`none | forwards | backwards | both`]#	Нет	Нет
`animation-iteration-count`	`1`	*<number>* | `infinite`]#	Нет	Нет
`animation-name`	`none`	[*<single-animation-name>* | `none`]#	Нет	Нет
`animation-play-state`	`running`	[`running | paused`]#	Нет	Нет
`animation-timing-function`	`ease`	[`ease | linear | ease-in | ease-out | ease-in-out | step-start | step-end | steps(`*<integer>*`, start) | steps(`*<integer>*`, end) | cubic-bezier(`*<number>*`, `*<number>*`, `*<number>*`, `*<number>*`)`]#	Нет	Нет
`animation`	`0s ease 0s 1 normal none running none`	[*<animation-name>*|| *<animation-duration>*|| *<animation-timing-function>*|| *<animation-delay>*|| *<animation-iteration-count>*|| *<animation-direction>*|| *<animation-fill-mode>*|| *<animation-play-state>*]#	Нет	Нет

Свойство	По умолчанию	Синтаксис значения	Наследование	Анимация
backface-visibility	visible	visible | hidden	Нет	Нет
background-attachment	scroll	[scroll | fixed | local]#	Нет	Нет
background-blend-mode	normal	[normal | multiply | screen | overlay | darken | lighten | color-dodge | color-burn | hard-light | soft-light | difference | exclusion | hue | saturation | color | luminosity]#	Нет	Нет
background-clip	border-box	[border-box | padding-box | content-box | text]#	Нет	Нет
background-color	transparent	*<color>*	Нет	Да
background-image	none	*<image>*# | none	Нет	Нет
background-origin	padding-box	[border-box | padding-box | content-box]#	Нет	Нет
background-position	0% 0%	[[left | center | right | top | bottom | *<percentage>* | *<length>*] | [left | center | right | *<percentage>* | *<length>*] [top | center | bottom | *<percentage>* | *<length>*] | [center | [left | right] [*<percentage>* | *<length>*]?] && [center | [top | bottom] [*<percentage>* | *<length>*]?]]#	Нет	Да
background-repeat	repeat	[repeat-x | repeat-y | [repeat | space | round | no-repeat]{1,2}]#	Нет	Нет
background-size	auto	[*<length>* | *<percentage>* | auto]{1,2} | cover | contain]#	Нет	Да
background	См. значения индивидуальных свойств	[*<bg-image>* || *<position>* [/ *<bg-size>*]? || *<repeat-style>* || *<attachment>* || *<box>* || *<box>*,]* *<bg-image>* || *<position>* [/ *<bg-size>*]? || *<repeat-style>* || *<attachment>* || *<box>* || *<box>* || *<background-color>*	Нет	Часть
border-bottom-color	Текущее для Color	*<color>*	Нет	Да
border-bottom-style	none	none | hidden | dotted | dashed | solid | double | groove | ridge | inset | outset	Нет	Нет
border-bottom-width	medium	thin | medium | thick | *<length>*	Нет	Да
border-bottom	См. значения индивидуальных свойств	[*<border-width>* || *<border-style>* || *<border-color>*]	Нет	Часть
border-color	current Color	*<color>*{1,4}	Нет	Да
border-image-outset	0	[*<length>* | *<number>*]{1,4}	Нет	Да
border-image-repeat	Stretch	[stretch | repeat | round | space]{1,2}	Нет	Нет

Свойство	По умолчанию	Синтаксис значения	Наследование	Анимация
border-image-slice	100%	[*<number>* | *<percentage>*]{1,4} && fill?	Нет	Да
border-image-source	none	none | *<image>*	Нет	Нет
border-image-width	1	[*<length>* | *<percentage>* | *<number>* | auto]{1,4}	Нет	Да
border-image	См. значения индивидуальных свойств	*<border-image-source>* || *<border-image-slice>* [/ *<border-image-width>* | / *<border-image-width>*? / *<border-image-outset>*]? || *<border-image-repeat>*	Нет	Нет
border-left-color	current Color	*<color>*	Нет	Да
border-left-style	none	none | hidden | dotted | dashed | solid | double | groove | ridge | inset | outset	Нет	Нет
border-left-width	medium	thin | medium | thick | *<length>*	Нет	Да
border-left	См. значения индивидуальных свойств	[*<border-width>* || *<border-style>* || *<border-color>*]	Нет	Часть
border-radius	0	[*<length>* | *<percentage>*]{1,4} [/ [*<length>* | *<percentage>*]{1,4}]?	Нет	Да
border-bottom-left-radius	0	[*<length>* | *<percentage>*]{1,2}	Нет	Да
border-bottom-right-radius	0	[*<length>* | *<percentage>*]{1,2}	Нет	Да
border-top-left-radius	0	[*<length>* | *<percentage>*]{1,2}	Нет	Да
border-top-right-radius	0	[*<length>* | *<percentage>*]{1,2}	Нет	Да
border-right-color	current Color	*<color>*	Нет	Да
border-right-style	none	none | hidden | dotted | dashed | solid | double | groove | ridge | inset | outset	Нет	Нет
border-right-width	medium	thin | medium | thick | *<length>*	Нет	Да
border-right	См. значения индивидуальных свойств	[*<border-width>* || *<border-style>* || *<border-color>*]	Нет	Часть
border-spacing	0	*<length>* *<length>*?	Да	Да
border-style	Нет	[none | hidden | solid | dotted | dashed | double | groove | ridge | inset | outset]{1,4}	Нет	Нет
border-top-color	current Color	*<color>*	Нет	Да
border-top-style	none	none | hidden | dotted | dashed | solid | double | groove | ridge | inset | outset	Нет	Нет

Свойство	По умолчанию	Синтаксис значения	Наследование	Анимация
border-top-width	medium	thin | medium | thick | *<length>*	Нет	Да
border-top	См. значения индивидуальных свойств	[*<border-width>* || *<border-style>* || *<border-color>*]	Нет	Часть
border-width	Нет	[thin | medium | thick | *<length>*]{1,4}	Нет	Да
border	См. значения индивидуальных свойств	[*<border-width>* || *<border-style>* || *<border-color>*]	Нет	Часть
bottom	auto	*<length>* | *<percentage>* | auto	Нет	Да
box-decoration-break	slice	slice | clone	Нет	Нет
box-shadow	none	none | inset? && *<length>*{2,4} && *<color>*?	Нет	Да
box-sizing	content-box	content-box | padding-box | border-box	Нет	Нет
caption-side	top	top | bottom	Да	Нет
clear	none	left | right | both | none	Нет	Нет
clip-path	none	none | *<url>* | [[inset() | circle() | ellipse() | polygon()] || [border-box | padding-box | content-box | margin-box | fill-box | stroke-box | view-box]]	Нет	Часть
clip-rule	nonzero	nonzero | evenodd	Нет	Нет
color	Определяется пользовательским агентом	*<color>*	Да	Да
direction	ltr	ltr | rtl	Да	Да
display	inline	[*<display-outside>* || *<display-inside>*] | *<display-list-item>* | *<display-internal>* | *<display-box>* | *<display-legacy>*	Нет	Нет
empty-cells	show	show | hide	Да	Нет
filter	none	[none | blur() | brightness() | contrast() | drop-shadow() | grayscale() | hue-rotate() | invert() | opacity() | sepia() | saturate() | url()]#	Нет	Да
flex-basis	auto	content | [*<width>* | *<percentage>*]	Нет	Часть
flex-direction	row	row | row-reverse | column | column-reverse	Нет	Нет
flex-flow	row nowrap	*<flex-direction>* || <flex-wrap>	Нет	Нет
flex-grow	0	*<number>*	Нет	Да
flex-shrink	1	*<number>*	Нет	Да
flex-wrap	nowrap	nowrap | wrap | wrap-reverse	Нет	Нет
flex	0 1 auto	[*<flex-grow>* *<flex-shrink>*? || *<flex-basis>*] | none | auto	Нет	Нет
float	none	left | right | none	Нет	Нет
font-family	Определяется пользовательским агентом	[*<family-name>* | *<generic-family>*]#	Да	Нет

Свойство	По умолчанию	Синтаксис значения	Наследование	Анимация
font-feature-settings	normal	normal | *<feature-tag-value>*#	Да	Нет
font-size-adjust	none	*<number>* | none | auto	Да	Да
font-size	medium	xx-small | x-small | small | medium | large | x-large | xx-large | smaller | larger | *<length>* | *<percentage>*	Да	Часть
font-stretch	normal	normal | ultra-condensed | extra-condensed | condensed | semi-condensed | semi-expanded | expanded | extra-expanded | ultra-expanded	Да	Нет
font-style	normal	italic | oblique | normal	Да	Нет
font-synthesis	weight style	none | weight || style	Да	Нет
font-variant	normal	normal | none | [*<common-lig-values>* || *<discretionary-lig-values>* || *<historical-lig-values>* || *<contextual-alt-values>* || stylistic(*<feature-value-name>*) || historical-forms || styleset(*<feature-value-name>*#) || charactervariant(*<feature-value-name>*#) || swash(*<feature-value-name>*) || ornaments(*<feature-value-name>*) || annotation(*<feature-value-name>*) || [small-caps | all-small-caps | petite-caps | all-petite-caps | unicase | titling-caps] || *<numeric-figure-values>* || *<numeric-spacing-values>* || *<numeric-fraction-values>* || ordinal || slashed-zero || *<east-asian-variant-values>* || *<east-asian-width-values>* || ruby]	Да	Нет
font-weight	normal	normal | bold | bolder | lighter | 100 | 200 | 300 | 400 | 500 | 600 | 700 | 800 | 900	Да	Нет
font	См. значения индивидуальных свойств	[[*<font-style>* || [normal | small-caps] || *<font-weight>*]? *<font-size>* [/ *<line-height>*]? *<font-family>*] | caption | icon | menu | message-box | small-caption | status-bar	Да	Часть
grid-area	См. значения индивидуальных свойств	*<grid-line>* [/ *<grid-line>*]{0,3}	Нет	Нет
grid-auto-columns	auto	*<track-breadth>* | minmax(*<track-breadth>*, *<track-breadth>*)	Нет	Нет
grid-auto-flow	row	[row | column] || dense	Нет	Нет
grid-auto-rows	auto	*<track-breadth>* | minmax(*<track-breadth>*, *<track-breadth>*)	Нет	Нет
grid-column-end	auto	auto | *<custom-ident>* | [*<integer>* && *<custom-ident>*?] | [span && [*<integer>* || *<custom-ident>*]]	Нет	Нет
grid-column-gap	0	*<length>* | *<percentage>*	Нет	Да
grid-column-start	auto	auto | *<custom-ident>* | [*<integer>* && *<custom-ident>*?] | [span && [*<integer>* || *<custom-ident>*]]	Нет	Нет

Свойство	По умолчанию	Синтаксис значения	Наследование	Анимация
grid-column	auto	*<grid-line>* [/ *<grid-line>*]?	Нет	Нет
grid-gap	0 0	*<grid-row-gap>* *<grid-column-gap>*	Нет	Да
grid-row-end	auto	auto | *<custom-ident>* | [*<integer>* && *<custom-ident>*?] | [span && [*<integer>* || *<custom-ident>*]]	Нет	Нет
grid-row-gap	0	*<length>* | *<percentage>*	Нет	Да
grid-row-start	auto	auto | *<custom-ident>* | [*<integer>* && *<custom-ident>*?] | [span && [*<integer>* || *<custom-ident>*]]	Нет	Нет
grid-row	auto	*<grid-line>* [/ *<grid-line>*]?	Нет	Нет
grid-template-areas	none	none | *<string>*	Нет	Нет
grid-template-columns	none	none | *<track-list>* | *<auto-track-list>*	Нет	Нет
grid-template-rows	none	none | *<track-list>* | *<auto-track-list>*	Нет	Нет
grid	См. значения индивидуальных свойств	none | subgrid | [*<grid-template-rows>* / *<grid-template-columns>*] | [*<line-names>*? *<string>* *<track-size>*? *<linenames>*?]+ [/ *<track-list>*]? | [*<grid-auto-flow>* [*<grid-auto-rows>* [/ *<grid-auto-columns>*]?]?]]	Нет	Нет
height	auto	*<length>* | *<percentage>* | auto	Нет	Да
hyphens	manual	manual | auto | none	Да	Нет
isolation	auto	auto | isolate	Нет	Нет
justify-content	flex-start	flex-start | flex-end | center | space-between | space-around	Нет	Нет
left	auto	*<length>* | *<percentage>* | auto	Нет	Да
letter-spacing	normal	*<length>* | normal	Да	Да
line-break	auto	auto | loose | normal | strict	Да	Да
line-height	normal	*<number>* | *<length>* | *<percentage>* | normal	Да	Да
margin-bottom	0	*<length>* | *<percentage>* | auto	Нет	Да
margin-left	0	*<length>* | *<percentage>* | auto	Нет	Да
margin-right	0	*<length>* | *<percentage>* | auto	Нет	Да
margin-top	0	*<length>* | *<percentage>* | auto	Нет	Да
margin	Нет	[*<length>* | *<percentage>* | auto]{1,4}	Нет	Да
mask-clip	border-box	[content-box | padding-box | border-box | margin-box | fill-box | stroke-box | view-box | no-clip]#	Нет	Нет
mask-composite	add	[add | subtract | intersect | exclude]#	Нет	Нет
mask-image	none	[none | *<image>* | *<mask-source>*]#	Нет	Нет
mask-mode	match-source	[alpha | luminance | match-source]#	Нет	Нет

Свойство	По умолчанию	Синтаксис значения	Наследование	Анимация
mask-origin	mask-origin	[content-box | padding-box | border-box | margin-box | fill-box | stroke-box | view-box]#	Нет	Нет
mask-position	0% 0%	*<position>*#[3]	Нет	Часть
mask-repeat	repeat	[repeat-x | repeat-y | [repeat | space | round | norepeat]{1,2}]#	Нет	Да
mask-size	auto	[[*<length>* | *<percentage>* | auto]{1,2} | cover | contain]#	Нет	Часть
mask-type	luminance	luminance | alpha	Нет	Нет
mask	См. значения индивидуальных свойств	[*<mask-image>* || *<mask-position>* [/ *<mask-size>*]? || *<mask-repeat>* || *<mask-clip>* || *<mask-origin>* || *<mask-composite>* || *<mask-mode>*]#	Нет	Часть
max-height	none	*<length>* | *<percentage>* | none	Нет	Да
max-width	none	*<length>* | *<percentage>* | none	Нет	Да
min-height	0	*<length>* | *<percentage>*	Нет	Да
min-width	0	*<length>* | *<percentage>*	Нет	Да
mix-blend-mode	normal	normal | multiply | screen | overlay | darken | lighten | color-dodge | color-burn | hard-light | soft-light | difference | exclusion | hue | saturation | color | luminosity	Нет	Нет
object-fit	fill	fill | contain | cover | scale-down | none	Нет	Нет
object-position	50% 50%	*<position>*[6]	Нет	Нет
order	0	*<integer>*	Нет	Да
orphans	2	*<integer>*	Нет	Нет
outline-color	invert	*<color>* | invert	Нет	Да
outline-style	none	auto | none | solid | dotted | dashed | double | groove | ridge | inset | outset	Нет	Нет
outline-width	medium	*<length>* | thin | medium | thick	Нет	Да
outline	none	[*<outline-color>* || *<outline-style>* || *<outline-width>*]	Нет	Часть
overflow-wrap[8]	normal	normal | break-word	Да	Да
overflow	visible	visible | hidden | scroll | auto	Нет	Нет
padding-bottom	0	*<length>* | *<percentage>*	Нет	Да
padding-left	0	*<length>* | *<percentage>*	Нет	Да
padding-right	0	*<length>* | *<percentage>*	Нет	Да
padding-top	0	*<length>* | *<percentage>*	Нет	Да
padding	Нет	[*<length>* | *<percentage>*]{1,4}	Нет	Да
page-break-after	auto	auto | always	Нет	Нет
page-break-before	auto	auto | always	Нет	Нет

Свойство	По умолчанию	Синтаксис значения	Наследование	Анимация
page-break-inside	auto	auto | avoid	Нет	Нет
page	auto	*<identifier>* | inherit	Нет	Нет
perspective-origin	50% 50%	*<position>*[1]	Нет	Да
perspective	none	none | *<length>*	Нет	Да
position	static	static | relative | sticky | absolute | fixed	Нет	Нет
right	auto	*<length>* | *<percentage>* | auto	Нет	Да
shape-image-threshold	0.0	*<number>*	Нет	Да
shape-margin	0	*<length>* | *<percentage>*	Нет	Да
shape-outside	none	none | [*<basic-shape>* || *<shape-box>*] |	Нет	Часть
size	auto	auto | *<length>*{1,2} | [*<page-size>* || [portrait | landscape]]	Нет	Нет
tab-size	8	*<length>* | *<integer>*	Да	Да
table-layout	auto	auto | fixed	Да	Нет
text-align-last	auto	auto | start | end | left | right | center | justify	Да	Нет
text-align	start	start | end | left | right | center | justify | match-parent | start end	Да	Нет
text-decoration	none	none | [underline || overline || line-through || blink]	Нет	Нет
text-indent	0	*<length>* | *<percentage>*	Да	Да
text-orientation	mixed	mixed | upright | sideways	Да	Да
text-rendering	auto	auto | optimizeSpeed | optimizeLegibility | geometricPrecision	Да	Да
text-shadow	none	none | [<length> || [*<length>* *<length>* *<length>*?]]#	Нет	Да
text-transform	none	uppercase | lowercase | capitalize | none	Да	Нет
top	auto	*<length>* | *<percentage>* | auto	Нет	Да
transform-origin	50% 50%	*<position>*[4]	Нет	Да
transform-style	flat	flat | preserve-3d	Нет	Нет
transform	none	*<transform-list>* | none	Нет	Да
transition-delay	0s	*<time>*#	Нет	Нет
transition-duration	0s	*<time>*#	Нет	Нет
transition-property	all	none | [all | *<property-name>*]#	Нет	Нет
transition-timing-function	ease	*<timing-function>*#[6]	Нет	Нет
transition	all 0s ease 0s	[[none | *<transition-property>*] || *<time>* || *<transition-timing-function>* || *<time>*]#	Нет	Нет

Свойство	По умолчанию	Синтаксис значения	Наследование	Анимация
unicode-bidi	normal	normal | embed | bidi-override	Нет	Да
vertical-align	baseline	baseline | sub | super | top | text-top | middle | bottom | text-bottom | *<length>* | *<percentage>*	Нет	Часть
visibility	visible	visible | hidden | collapse	Да	Нет
white-space	normal	normal | nowrap | pre | pre-wrap | pre-line	Нет	Нет
widows	2	*<integer>*	Нет	Нет
width	auto	*<length>* | *<percentage>* | auto	Нет	Да
word-break	normal	normal | break-all | keep-all	Да	Да
word-spacing	normal	*<length>* | normal	Да	Да
writing-mode	horizotal-tb	horizontal-tb | vertical-rl | vertical-lr	Да	Да
z-index	auto	*<integer>* | auto	Нет	Да

[а] Синтаксис значения *<position>* подробно рассматривается в описании свойства background-position.

[б] Синтаксис значения *<position>* подробно рассматривается в описании свойства background-position.

[в] Это свойство часто называют word-wrap.

[г] Синтаксис значения *<position>* подробно рассматривается в описании свойства background-position.

[д] Синтаксис значения *<position>* подробно рассматривается в описании свойства background-position.

[е] Синтаксис значения *<timing-function>* подробно рассматривается в описании свойства animation-timing-function.

Таблица соответствия цветов

В этом приложении приведены названия всех 148 именованных цветов, определенных в спецификации модуля CSS Color Module Level 4 (по состоянию на 22 мая 2017 года), а также соответствующие им значения в цветовых моделях RGB, HSL и шестнадцатеричном представлении (включая короткую форму записи).

К концу 2017 года рабочей группой было принято решение о внедрении нового способа представления цветовых значений — через функции `hwb()`, `gray()`, `lab()`, `lch()` и т.п. Тем не менее в приведенной ниже таблице они не указываются в связи с неопределенностью сроков внедрения их поддержки в браузерах.

Имя цвета	RGB-значение (числовое)	RGB-значение (процентное)	HSL-значение	Шестнадцатеричное значение
aliceblue	rgb(240,248,255)	rgb(94.1%,97.3%,100%)	hsl(208,100%,97.1%)	#F0F8FF
antiquewhite	rgb(250,235,215)	rgb(98%,92.2%,84.3%)	hsl(34,77.8%,91.2%)	#FAEBD7
aqua	rgb(0,255,255)	rgb(0%,100%,100%)	hsl(180,100%,50%)	#00FFFF/#0FF
aquamarine	rgb(127,255,212)	rgb(49.8%,100%,83.1%)	hsl(160,100%,74.9%)	#7FFFD4
azure	rgb(240,255,255)	rgb(94.1%,100%,100%)	hsl(180,100%,97.1%)	#F0FFFF
beige	rgb(245,245,220)	rgb(96.1%,96.1%,86.3%)	hsl(60,55.6%,91.2%)	#F5F5DC
bisque	rgb(255,228,196)	rgb(100%,89.4%,76.9%)	hsl(33,100%,88.4%)	#FFE4C4
black	rgb(0,0,0)	rgb(0%,0%,0%)	hsl(0,0%,0%)	#000000/#000
blanchedalmond	rgb(255,235,205)	rgb(100%,92.2%,80.4%)	hsl(36,100%,90.2%)	#FFEBCD
blue	rgb(0,0,255)	rgb(0%,0%,100%)	hsl(240,100%,50%)	#0000FF/#00F
blueviolet	rgb(138,43,226)	rgb(54.1%,16.9%,88.6%)	hsl(271,75.9%,52.7%)	#8A2BE2
brown	rgb(165,42,42)	rgb(64.7%,16.5%,16.5%)	hsl(0,59.4%,40.6%)	#A52A2A
burlywood	rgb(222,184,135)	rgb(87.1%,72.2%,52.9%)	hsl(34,56.9%,70%)	#DEB887
cadetblue	rgb(95,158,160)	rgb(37.3%,62%,62.7%)	hsl(182,25.5%,50%)	#5F9EA0
chartreuse	rgb(127,255,0)	rgb(49.8%,100%,0%)	hsl(90,100%,50%)	#7FFF00
chocolate	rgb(210,105,30)	rgb(82.4%,41.2%,11.8%)	hsl(25,75%,47.1%)	#D2691E
coral	rgb(255,127,80)	rgb(100%,49.8%,31.4%)	hsl(16,100%,65.7%)	#FF7F50
cornflowerblue	rgb(100,149,237)	rgb(39.2%,58.4%,92.9%)	hsl(219,79.2%,66.1%)	#6495ED
cornsilk	rgb(255,248,220)	rgb(100%,97.3%,86.3%)	hsl(48,100%,93.1%)	#FFF8DC
crimson	rgb(220,20,60)	rgb(86.3%,7.8%,23.5%)	hsl(348,83.3%,47.1%)	#DC143C
cyan	rgb(0,255,255)	rgb(0%,100%,100%)	hsl(180,100%,50%)	#00FFFF/#0FF
darkblue	rgb(0,0,139)	rgb(0%,0%,54.5%)	hsl(240,100%,27.3%)	#00008B
darkcyan	rgb(0,139,139)	rgb(0%,54.5%,54.5%)	hsl(180,100%,27.3%)	#008B8B

Имя цвета	RGB-значение (числовое)	RGB-значение (процентное)	HSL-значение	Шестнадцатеричное значение
darkgoldenrod	rgb(184,134,11)	rgb(72.2%,52.5%,4.3%)	hsl(43,88.7%,38.2%)	#B8860B
darkgray	rgb(169,169,169)	rgb(66.3%,66.3%,66.3%)	hsl(0,0%,66.3%)	#A9A9A9
darkgreen	rgb(0,100,0)	rgb(0%,39.2%,0%)	hsl(120,100%,19.6%)	#006400
darkgrey	rgb(169,169,169)	rgb(66.3%,66.3%,66.3%)	hsl(0,0%,66.3%)	#A9A9A9
darkkhaki	rgb(189,183,107)	rgb(74.1%,71.8%,42%)	hsl(56,38.3%,58%)	#BDB76B
darkmagenta	rgb(139,0,139)	rgb(54.5%,0%,54.5%)	hsl(300,100%,27.3%)	#8B008B
darkolivegreen	rgb(85,107,47)	rgb(33.3%,42%,18.4%)	hsl(82,39%,30.2%)	#556B2F
darkorange	rgb(255,140,0)	rgb(100%,54.9%,0%)	hsl(33,100%,50%)	#FF8C00
darkorchid	rgb(153,50,204)	rgb(60%,19.6%,80%)	hsl(280,60.6%,49.8%)	#9932CC
darkred	rgb(139,0,0)	rgb(54.5%,0%,0%)	hsl(0,100%,27.3%)	#8B0000
darksalmon	rgb(233,150,122)	rgb(91.4%,58.8%,47.8%)	hsl(15,71.6%,69.6%)	#E9967A
darkseagreen	rgb(143,188,143)	rgb(56.1%,73.7%,56.1%)	hsl(120,25.1%,64.9%)	#8FBC8F
darkslateblue	rgb(72,61,139)	rgb(28.2%,23.9%,54.5%)	hsl(248,39%,39.2%)	#483D8B
darkslategray	rgb(47,79,79)	rgb(18.4%,31%,31%)	hsl(180,25.4%,24.7%)	#2F4F4F
darkslategrey	rgb(47,79,79)	rgb(18.4%,31%,31%)	hsl(180,25.4%,24.7%)	#2F4F4F
darkturquoise	rgb(0,206,209)	rgb(0%,80.8%,82%)	hsl(181,100%,41%)	#00CED1
darkviolet	rgb(148,0,211)	rgb(58%,0%,82.7%)	hsl(282,100%,41.4%)	#9400D3
deeppink	rgb(255,20,147)	rgb(100%,7.8%,57.6%)	hsl(328,100%,53.9%)	#FF1493
deepskyblue	rgb(0,191,255)	rgb(0%,74.9%,100%)	hsl(195,100%,50%)	#00BFFF
dimgray	rgb(105,105,105)	rgb(41.2%,41.2%,41.2%)	hsl(0,0%,41.2%)	#696969
dimgrey	rgb(105,105,105)	rgb(41.2%,41.2%,41.2%)	hsl(0,0%,41.2%)	#696969
dodgerblue	rgb(30,144,255)	rgb(11.8%,56.5%,100%)	hsl(210,100%,55.9%)	#1E90FF
firebrick	rgb(178,34,34)	rgb(69.8%,13.3%,13.3%)	hsl(0,67.9%,41.6%)	#B22222
floralwhite	rgb(255,250,240)	rgb(100%,98%,94.1%)	hsl(40,100%,97.1%)	#FFFAF0
forestgreen	rgb(34,139,34)	rgb(13.3%,54.5%,13.3%)	hsl(120,60.7%,33.9%)	#228B22
fuchsia	rgb(255,0,255)	rgb(100%,0%,100%)	hsl(300,100%,50%)	#FF00FF/#F0F
gainsboro	rgb(220,220,220)	rgb(86.3%,86.3%,86.3%)	hsl(0,0%,86.3%)	#DCDCDC
ghostwhite	rgb(248,248,255)	rgb(97.3%,97.3%,100%)	hsl(240,100%,98.6%)	#F8F8FF
gold	rgb(255,215,0)	rgb(100%,84.3%,0%)	hsl(51,100%,50%)	#FFD700
goldenrod	rgb(218,165,32)	rgb(85.5%,64.7%,12.5%)	hsl(43,74.4%,49%)	#DAA520
gray	rgb(128,128,128)	rgb(50.2%,50.2%,50.2%)	hsl(0,0%,50.2%)	#808080
green	rgb(0,128,0)	rgb(0%,50.2%,0%)	hsl(120,100%,25.1%)	#008000
greenyellow	rgb(173,255,47)	rgb(67.8%,100%,18.4%)	hsl(84,100%,59.2%)	#ADFF2F
grey	rgb(128,128,128)	rgb(50.2%,50.2%,50.2%)	hsl(0,0%,50.2%)	#808080
honeydew	rgb(240,255,240)	rgb(94.1%,100%,94.1%)	hsl(120,100%,97.1%)	#F0FFF0
hotpink	rgb(255,105,180)	rgb(100%,41.2%,70.6%)	hsl(330,100%,70.6%)	#FF69B4
indianred	rgb(205,92,92)	rgb(80.4%,36.1%,36.1%)	hsl(0,53.1%,58.2%)	#CD5C5C
indigo	rgb(75,0,130)	rgb(29.4%,0%,51%)	hsl(275,100%,25.5%)	#4B0082
ivory	rgb(255,255,240)	rgb(100%,100%,94.1%)	hsl(60,100%,97.1%)	#FFFFF0
khaki	rgb(240,230,140)	rgb(94.1%,90.2%,54.9%)	hsl(54,76.9%,74.5%)	#F0E68C
lavender	rgb(230,230,250)	rgb(90.2%,90.2%,98%)	hsl(240,66.7%,94.1%)	#E6E6FA
lavenderblush	rgb(255,240,245)	rgb(100%,94.1%,96.1%)	hsl(340,100%,97.1%)	#FFF0F5
lawngreen	rgb(124,252,0)	rgb(48.6%,98.8%,0%)	hsl(90,100%,49.4%)	#7CFC00
lemonchiffon	rgb(255,250,205)	rgb(100%,98%,80.4%)	hsl(54,100%,90.2%)	#FFFACD
lightblue	rgb(173,216,230)	rgb(67.8%,84.7%,90.2%)	hsl(195,53.3%,79%)	#ADD8E6
lightcoral	rgb(240,128,128)	rgb(94.1%,50.2%,50.2%)	hsl(0,78.9%,72.2%)	#F08080
lightcyan	rgb(224,255,255)	rgb(87.8%,100%,100%)	hsl(180,100%,93.9%)	#E0FFFF

Имя цвета	RGB-значение (числовое)	RGB-значение (процентное)	HSL-значение	Шестнадцатеричное значение
lightgoldenrodyellow	rgb(250,250,210)	rgb(98%,98%,82.4%)	hsl(60,80%,90.2%)	#FAFAD2
lightgray	rgb(211,211,211)	rgb(82.7%,82.7%,82.7%)	hsl(0,0%,82.7%)	#D3D3D3
lightgreen	rgb(144,238,144)	rgb(56.5%,93.3%,56.5%)	hsl(120,73.4%,74.9%)	#90EE90
lightgrey	rgb(211,211,211)	rgb(82.7%,82.7%,82.7%)	hsl(0,0%,82.7%)	#D3D3D3
lightpink	rgb(255,182,193)	rgb(100%,71.4%,75.7%)	hsl(351,100%,85.7%)	#FFB6C1
lightsalmon	rgb(255,160,122)	rgb(100%,62.7%,47.8%)	hsl(17,100%,73.9%)	#FFA07A
lightseagreen	rgb(32,178,170)	rgb(12.5%,69.8%,66.7%)	hsl(177,69.5%,41.2%)	#20B2AA
lightskyblue	rgb(135,206,250)	rgb(52.9%,80.8%,98%)	hsl(203,92%,75.5%)	#87CEFA
lightslategray	rgb(119,136,153)	rgb(46.7%,53.3%,60%)	hsl(210,14.3%,53.3%)	#778899/#789
lightslategrey	rgb(119,136,153)	rgb(46.7%,53.3%,60%)	hsl(210,14.3%,53.3%)	#778899/#789
lightsteelblue	rgb(176,196,222)	rgb(69%,76.9%,87.1%)	hsl(214,41.1%,78%)	#B0C4DE
lightyellow	rgb(255,255,224)	rgb(100%,100%,87.8%)	hsl(60,100%,93.9%)	#FFFFE0
lime	rgb(0,255,0)	rgb(0%,100%,0%)	hsl(120,100%,50%)	#00FF00/#0F0
limegreen	rgb(50,205,50)	rgb(19.6%,80.4%,19.6%)	hsl(120,60.8%,50%)	#32CD32
linen	rgb(250,240,230)	rgb(98%,94.1%,90.2%)	hsl(30,66.7%,94.1%)	#FAF0E6
magenta	rgb(255,0,255)	rgb(100%,0%,100%)	hsl(300,100%,50%)	#FF00FF/#F0F
maroon	rgb(128,0,0)	rgb(50.2%,0%,0%)	hsl(0,100%,25.1%)	#800000
mediumaquamarine	rgb(102,205,170)	rgb(40%,80.4%,66.7%)	hsl(160,50.7%,60.2%)	#66CDAA
mediumblue	rgb(0,0,205)	rgb(0%,0%,80.4%)	hsl(240,100%,40.2%)	#0000CD
mediumorchid	rgb(186,85,211)	rgb(72.9%,33.3%,82.7%)	hsl(288,58.9%,58%)	#BA55D3
mediumpurple	rgb(147,112,219)	rgb(57.6%,43.9%,85.9%)	hsl(260,59.8%,64.9%)	#9370DB
mediumseagreen	rgb(60,179,113)	rgb(23.5%,70.2%,44.3%)	hsl(147,49.8%,46.9%)	#3CB371
mediumslateblue	rgb(123,104,238)	rgb(48.2%,40.8%,93.3%)	hsl(249,79.8%,67.1%)	#7B68EE
mediumspringgreen	rgb(0,250,154)	rgb(0%,98%,60.4%)	hsl(157,100%,49%)	#00FA9A
mediumturquoise	rgb(72,209,204)	rgb(28.2%,82%,80%)	hsl(178,59.8%,55.1%)	#48D1CC
mediumvioletred	rgb(199,21,133)	rgb(78%,8.2%,52.2%)	hsl(322,80.9%,43.1%)	#C71585
midnightblue	rgb(25,25,112)	rgb(9.8%,9.8%,43.9%)	hsl(240,63.5%,26.9%)	#191970
mintcream	rgb(245,255,250)	rgb(96.1%,100%,98%)	hsl(150,100%,98%)	#F5FFFA
mistyrose	rgb(255,228,225)	rgb(100%,89.4%,88.2%)	hsl(6,100%,94.1%)	#FFE4E1
moccasin	rgb(255,228,181)	rgb(100%,89.4%,71%)	hsl(38,100%,85.5%)	#FFE4B5
navajowhite	rgb(255,222,173)	rgb(100%,87.1%,67.8%)	hsl(36,100%,83.9%)	#FFDEAD
navy	rgb(0,0,128)	rgb(0%,0%,50.2%)	hsl(240,100%,25.1%)	#000080
oldlace	rgb(253,245,230)	rgb(99.2%,96.1%,90.2%)	hsl(39,85.2%,94.7%)	#FDF5E6
olive	rgb(128,128,0)	rgb(50.2%,50.2%,0%)	hsl(60,100%,25.1%)	#808000
olivedrab	rgb(107,142,35)	rgb(42%,55.7%,13.7%)	hsl(80,60.5%,34.7%)	#6B8E23
orange	rgb(255,165,0)	rgb(100%,64.7%,0%)	hsl(39,100%,50%)	#FFA500
orangered	rgb(255,69,0)	rgb(100%,27.1%,0%)	hsl(16,100%,50%)	#FF4500
orchid	rgb(218,112,214)	rgb(85.5%,43.9%,83.9%)	hsl(302,58.9%,64.7%)	#DA70D6
palegoldenrod	rgb(238,232,170)	rgb(93.3%,91%,66.7%)	hsl(55,66.7%,80%)	#EEE8AA
palegreen	rgb(152,251,152)	rgb(59.6%,98.4%,59.6%)	hsl(120,92.5%,79%)	#98FB98
paleturquoise	rgb(175,238,238)	rgb(68.6%,93.3%,93.3%)	hsl(180,64.9%,81%)	#AFEEEE
palevioletred	rgb(219,112,147)	rgb(85.9%,43.9%,57.6%)	hsl(340,59.8%,64.9%)	#DB7093
papayawhip	rgb(255,239,213)	rgb(100%,93.7%,83.5%)	hsl(37,100%,91.8%)	#FFEFD5
peachpuff	rgb(255,218,185)	rgb(100%,85.5%,72.5%)	hsl(28,100%,86.3%)	#FFDAB9
peru	rgb(205,133,63)	rgb(80.4%,52.2%,24.7%)	hsl(30,58.7%,52.5%)	#CD853F
pink	rgb(255,192,203)	rgb(100%,75.3%,79.6%)	hsl(350,100%,87.6%)	#FFC0CB
plum	rgb(221,160,221)	rgb(86.7%,62.7%,86.7%)	hsl(300,47.3%,74.7%)	#DDA0DD

Имя цвета	RGB-значение (числовое)	RGB-значение (процентное)	HSL-значение	Шестнадцатеричное значение
powderblue	rgb(176,224,230)	rgb(69%,87.8%,90.2%)	hsl(187,51.9%,79.6%)	#B0E0E6
purple	rgb(128,0,128)	rgb(50.2%,0%,50.2%)	hsl(300,100%,25.1%)	#800080
rebeccapurple	rgb(102,51,153)	rgb(40%,20%,60%)	hsl(270,50%,40%)	#663399/#639
red	rgb(255,0,0)	rgb(100%,0%,0%)	hsl(0,100%,50%)	#FF0000/#F00
rosybrown	rgb(188,143,143)	rgb(73.7%,56.1%,56.1%)	hsl(0,25.1%,64.9%)	#BC8F8F
royalblue	rgb(65,105,225)	rgb(25.5%,41.2%,88.2%)	hsl(225,72.7%,56.9%)	#4169E1
saddlebrown	rgb(139,69,19)	rgb(54.5%,27.1%,7.5%)	hsl(25,75.9%,31%)	#8B4513
salmon	rgb(250,128,114)	rgb(98%,50.2%,44.7%)	hsl(6,93.2%,71.4%)	#FA8072
sandybrown	rgb(244,164,96)	rgb(95.7%,64.3%,37.6%)	hsl(28,87.1%,66.7%)	#F4A460
seagreen	rgb(46,139,87)	rgb(18%,54.5%,34.1%)	hsl(146,50.3%,36.3%)	#2E8B57
seashell	rgb(255,245,238)	rgb(100%,96.1%,93.3%)	hsl(25,100%,96.7%)	#FFF5EE
sienna	rgb(160,82,45)	rgb(62.7%,32.2%,17.6%)	hsl(19,56.1%,40.2%)	#A0522D
silver	rgb(192,192,192)	rgb(75.3%,75.3%,75.3%)	hsl(0,0%,75.3%)	#C0C0C0
skyblue	rgb(135,206,235)	rgb(52.9%,80.8%,92.2%)	hsl(197,71.4%,72.5%)	#87CEEB
slateblue	rgb(106,90,205)	rgb(41.6%,35.3%,80.4%)	hsl(248,53.5%,57.8%)	#6A5ACD
slategray	rgb(112,128,144)	rgb(43.9%,50.2%,56.5%)	hsl(210,12.6%,50.2%)	#708090
slategrey	rgb(112,128,144)	rgb(43.9%,50.2%,56.5%)	hsl(210,12.6%,50.2%)	#708090
snow	rgb(255,250,250)	rgb(100%,98%,98%)	hsl(0,100%,99%)	#FFFAFA
springgreen	rgb(0,255,127)	rgb(0%,100%,49.8%)	hsl(150,100%,50%)	#00FF7F
steelblue	rgb(70,130,180)	rgb(27.5%,51%,70.6%)	hsl(207,44%,49%)	#4682B4
tan	rgb(210,180,140)	rgb(82.4%,70.6%,54.9%)	hsl(34,43.8%,68.6%)	#D2B48C
teal	rgb(0,128,128)	rgb(0%,50.2%,50.2%)	hsl(180,100%,25.1%)	#008080
thistle	rgb(216,191,216)	rgb(84.7%,74.9%,84.7%)	hsl(300,24.3%,79.8%)	#D8BFD8
tomato	rgb(255,99,71)	rgb(100%,38.8%,27.8%)	hsl(9,100%,63.9%)	#FF6347
turquoise	rgb(64,224,208)	rgb(25.1%,87.8%,81.6%)	hsl(174,72.1%,56.5%)	#40E0D0
violet	rgb(238,130,238)	rgb(93.3%,51%,93.3%)	hsl(300,76.1%,72.2%)	#EE82EE
wheat	rgb(245,222,179)	rgb(96.1%,87.1%,70.2%)	hsl(39,76.7%,83.1%)	#F5DEB3
white	rgb(255,255,255)	rgb(100%,100%,100%)	hsl(0,0%,100%)	#FFFFFF/#FFF
whitesmoke	rgb(245,245,245)	rgb(96.1%,96.1%,96.1%)	hsl(0,0%,96.1%)	#F5F5F5
yellow	rgb(255,255,0)	rgb(100%,100%,0%)	hsl(60,100%,50%)	#FFFF00/#FF0
yellowgreen	rgb(154,205,50)	rgb(60.4%,80.4%,19.6%)	hsl(80,60.8%,50%)	#9ACD32

Предметный указатель

D

dpcm, 153
dpi, 153
dppx, 153

F

flex-контейнер, 591; 598; 617; 641
  оси, 609
  перенос содержимого, 606
flex-элемент, 592; 598; 641
  выравнивание, 618; 627
  отступ, 643
  упорядочение, 616

G

grid-контейнер, 687
grid-элемент, 690
  абсолютное позиционирование, 754
  перекрывание, 735
  поток, 736

H

HSL, 168; 994
HTTP, 35

O

OpenType, 224

P

ppi, 152

R

RGB, 163
RGBa, 165

S

SVG, 986

T

TRBL, 347

U

URL, 145; 818

A

Абсолютное позиционирование, 563
  flex-элемента, 644
Абсолютные единицы измерения, 150
Автоматический размер, 567
Адаптивная колонка, 699
Аддитивная нумерация, 845
Адрес, 145; 818
Алфавитная нумерация, 840
Альфа-канал, 984
Анимация, 927
  временная функция, 956
  длительность, 940
  задержка, 944
  именованная, 937
  имя, 929
  итерации, 976
  направление, 943
  отмена, 952
  печать, 980
  повторение, 941
  порядок, 975
  последовательность, 948
  приостановка, 968
  производительность, 949
  режим, 969
  события, 978
  шаговая, 960
  ширины элемента, 959
Анимируемое свойство, 892; 920; 933
Анонимный текст, 315
Аспект шрифта, 212
Атрибут, 68; 161; 817
  class, 62
  href, 30
  media, 30; 41; 925; 1028
  rel, 30
  style, 36
  type, 30
  события, 902

Б

Базовая линия, 249; 320; 325; 334
  flex-элемента, 627
  выравнивание, 632
Базовый размер, 668
Блок объявлений, 37; 53
Блочная модель элемента, 292
Блочный элемент, 24; 235; 287; 292
Бумага, 1043
Бумажный носитель, 1039

B

Важность стиля, 130
Варианты шрифта, 221

Вектор вращения, 870
Вендорный префикс, 38
Верстка
  адаптивная, 663
  по сетке, 687
  списков, 809
Вертикальное выравнивание, 248
Вертикальное форматирование, 302
Верхний индекс, 250
Визуализация, 340
Висячая строка, 1049
Висячий отступ, 237
Вложенные кавычки, 820
Вложенный список, 829
Вложенный элемент, 287
Внешний контур, 395
  толщина, 397
  цвет, 398
Внешний стиль, 29
Внутренний элемент таблицы, 765
Вращение, 866
  в трехмерном пространстве, 868
  элемента, 857
Временная функция, 905
  ступенчатая, 909
Время, 172
Встроенный стиль, 36
  приоритетность, 129
Выравнивание, 514; 635
  flex-элементов, 618; 631; 635
  grid-элементов, 755
  абсолютно позиционируемого элемента, 571
  базовых линий, 632
  вертикальное, 248; 798
  всех элементов сетки, 758
  наследование, 242
  по базовой линии, 249
  по верхнему краю, 251
  по высоте, 244
  по краям элемента, 241
  по нижнему краю, 250
  посередине, 252
  последней строки, 242
  по ширине, 241
  содержимого таблицы, 796
  строчных элементов, 321
  текста, 239
  текстовых строк, 314
Высота
  области визуализации, 1034
  отображаемой области, 1034
  строки, 229; 246; 324
  строчного элемента, 321
  таблицы, 795
  текстовой строки, 244
  элемента, 302; 344; 556
Вычисляемое значение, 159; 341

Г

Генерируемое содержимое, 813
Геометрическая фигура, 536
Гибкая верстка, 591; 603; 646
Гибкий контейнер, 617
Гибкий элемент, 592; 641
Гиперссылка, 103; 412
  значок, 425
  посещенная, 105
Главная ось, 609
Главный размер, 605; 609
Глубина цвета, 1035
Горизонтальное форматирование, 293
Градиент, 474
  жесткий, 480
  линейный, 475
  направление, 484
  обрезка, 478
  повторение, 505
  радиальный, 489
  форма, 490
  цвет, 476
Градиентный маркер, 807
Градус, 171
Границы, 286; 356; 443
  замещаемого элемента, 333
  прозрачные, 365
  списка, 810
  стиль линии, 356
  строчных элементов, 327; 369
  толщина, 361
  цвет, 363; 414
  ячейки, 780
Графический маркер, 805
Группа
  колонок, 766; 769
  рядов, 766; 769
Группирование
  объявлений, 58
  селекторов, 56

Д

Декартова система координат, 854
Декоративный шрифт, 178
Дескриптор
  additive-symbols, 845

fallback, 839
font-family, 183
font-feature-settings, 224; 226
font-stretch, 219
font-style, 217
font-variant, 223
font-weight, 202
format(), 185
negative, 842
pad, 843
prefix, 835
range, 838
speak-as, 849
src, 183
suffix, 835
symbols, 831
system, 831
unicode-range, 190
носителя, 43; 1031
технических характеристик, 46; 1031; 1033
шрифта, 183
Динамический псевдокласс, 102
Дискретный узор, 507
Длина, 149
Длительность перехода, 903
Дольная единица измерения, 699
Дочерний элемент, 79
Дробное значение, 149
Единицы измерения
ch, 156
em, 153
ex, 153
fr, 699
hz, 172
khz, 172
ms, 172
rem, 154
s, 172
vh, 158
vm, 158
vmax, 158
абсолютные, 150
относительные, 153
пользовательские, 863
угла, 867

Ж

Жесткий перенос, 273
Жесткий свет, 992

З

Заголовок
HTTP, 35
таблицы, 769; 778
Задержка
анимации, 944
итераций анимации, 953
отрицательная, 913
перехода, 911
Задний план, 989
Замещаемый элемент, 23; 287; 301; 332
абсолютное позиционирование, 575
Заполнение фона, 446
Запрос поддержки, 47
Затемнение, 990
основы, 993
Значение, 53
атрибута, 817
вычисляемое, 159; 341
дробное, 149
интерполяция, 921; 922
исходное, 895
начальное, 143
пользовательское, 173
строковое, 144
целочисленное, 148
числовое, 148
шестнадцатеричное, 166

И

Идентификатор, 65; 148
счетчика, 822
фрагмента документа, 114
Избыточное значение, 296
Изображение, 147
Именованная анимация, 929; 937
Именованная линия, 727
Именованная область, 717
Именованный цвет, 170
Инверсия цветов, 985
Индекс, 250
Интервал
между маркерами и содержимым, 812
между полосами сетки, 748
между словами, 254
между ячейками, 781
межсимвольный, 256
Интерлиньяж, 244; 315; 319
Интерполяция, 921; 922
Исключение цветов, 990
Источник стиля, 135
Исходная точка
смещения, 435
трансформации, 877
Исходное изображение, 428
Исходное состояние, 895
Исходный размер, 660

К

Кавычки
  в названии шрифта, 181
  генерирование, 819
Капитель, 221
Каскадирование, 134
Кегельная площадка, 203; 315
Кернинг, 220
Класс, 64
Ключевое слово, 53; 141; 429
  alternate, 943
  alternate-reverse, 943
  always, 1046
  baseline, 798
  bottom, 797
  currentColor, 171
  cyclic, 833
  from, 930
  !important, 975
  inherit, 142
  initial, 143
  left, 1047
  middle, 797
  none, 803; 901
  normal, 943
  only, 45
  reverse, 943
  right, 1047
  suffix, 835
  to, 930
  top, 796
  transparent, 171; 417
  unset, 143
Ключевой кадр, 928; 930
Колонка
  создание, 725
  таблицы, 766; 769
  фиксированной ширины, 694
Колонтитул, 1052
Команда
  @counter-style, 829
  @font-face, 182
  @import, 34; 41; 1028
  @keyframes, 928; 929
  @media, 41; 1029
  @page, 1041
  @supports, 47
Комбинатор, 80
  соседних элементов, 84
Комбинированный класс, 64
Комментарий, 40
Конечная точка, 609
Контейнер
  гибкий, 591; 598
  группы колонок, 766
  группы рядов, 766
  колонки, 766
  опорный, 540
  произвольной формы, 534
  ряда, 766
  сетки, 687
  страницы, 1041
  строки, 316
  строчного элемента, 316; 318
  элемента, 245; 285; 343
Контекстный селектор, 80
Контрастность, 986
Контур, 1001
Координаты, 853
Корневой элемент, 89; 287
Коэффициент
  масштабирования, 204; 247; 326
  растягивания, 648
  сжатия, 655
  удельный, 659
Кривая Безье, 906; 956
Круг, 538
Курсив, 178; 215

Л

Линейный градиент, 475
Линия сетки, 692
  автоматическое размещение, 741
  именованная, 727
  имя, 696
  повторение, 709
  позиционирование, 694
Липкое позиционирование, 585
Логический оператор, 44; 1031

М

Маркер, 802; 832
  градиентный, 807
  графический, 805
  повторяющийся, 834
  позиционирование, 808
  списка, 312
  строчный, 804
Маска, 1008
  комбинирование, 1017
  масштабирование, 1014
  обрезка, 1017
  повторение, 1014
  позиционирование, 1015

размер, 1013
тип, 1021
Маскирование, 1011
Масштабирование, 204; 865
высоты строки, 326
изображения, 463
маски, 1014
Матричная функция, 873
Медиа-запрос, 41; 1027
prefers-reduced-motion, 977
составной, 1030
Междустрочный интервал, 245
Межсимвольный интервал, 256
Миллисекунда, 172
Минимальная ширина, 645
Многоугольная форма, 543
Модель элемента, 343
Монохромный носитель, 1035
Моноширинный шрифт, 178; 209
Мягкий перенос, 273
Мягкий свет, 992

H

Навигационная панель, 684; 893
Наклон, 871; 880
Наложение, 577; 988
grid-элементов, 761
Направление
анимации, 943
заполнения flex-контейнера, 608
письма, 277; 603
Наследование, 131
маркеров списка, 806
межсимвольного интервала, 258
Насыщенность, 168; 986; 995
уровень, 196
шрифта, 195
Начальная точка, 609
Начальное значение, 143
Начертание, 177; 197; 214
генерирование, 226
Неанимируемое свойство, 935
Неделимый элемент, 861
Незамещаемый элемент, 24; 287; 570
Неоформленное содержимое, 185
Непрозрачность, 982
Нечувствительность к регистру, 77
Неявная сетка, 728
Нижний индекс, 250
Носитель, 43; 1027
бумажный, 1039
монохромный, 1035
Нулевой размер, 675
Нумерация, 824
аддитивная, 845
алфавитная, 840
позиционная, 841
символьная, 837
Нумерованный элемент, 97

O

Область
визуализации, 1034
липкого позиционирования, 586
отображаемая, 1033
отступов, 1041
позиционирования фона, 437; 450
рисования фона, 420; 450
сетки, 692; 714; 717
содержимого, 245; 285; 297; 315; 319; 331
страницы, 1041
Оборот, 172
Обратная сторона, 887
Обратный переход, 916
Обрезка
градиента, 478
маски, 1017
фона, 420; 450
Обтекание, 513
отмена, 516; 530
по форме элемента, 534
Обтравочный контур, 1001
заполнение, 1007
Общие границы, 779
Объектная модель документа, 78
Объявление, 37; 53
группирование, 58
приоритетность, 127
Ограничительная рамка, 858
Окно просмотра, 158; 581
Опорный контейнер, 540
Ориентация
области визуализации, 1036
символов, 281
Осветление, 990
основы, 993
Основной цвет, 411
Особенности шрифта, 224
Ось
flex-контейнера, 609
блочных элементов, 692
главная, 609
координатная, 854
поперечная, 609; 627

строчных элементов, 692
Адаптивный дизайн, 663; 1037
Относительное позиционирование, 583
Относительные единицы изменения, 153
Отрицание, 116
Отрицательная задержка, 913
Отрицательный отступ, 299; 334; 526
Отступ, 235; 286; 296; 401
  flex-элемента, 643
  grid-элемента, 751
  висячий, 237
  замещаемого элемента, 333
  отрицательный, 238; 299; 309; 334; 407; 526
  первой строки, 236
  списка, 811
  страницы, 1044
  строчного элемента, 328; 408
  схлопывание, 307; 405; 689
  формы, 547
Оттенки серого, 984
Оформление текста, 260

П

Панель навигации, 595; 626
Первая буква, 119
Первая строка, 120
Передний план, 989
Перекрывание, 528
  grid-элементов, 735; 762
Перекрытие цветов, 992
Перенос, 271
  жесткий, 273
  мягкий, 273
  предотвращение, 268
  содержимого, 615
Переполнение, 560
Перетекание, 339
Переход, 891
  длительность, 903
  обратный, 916
  отложенный, 911
  отмена, 900
  печать, 925
  прерывание, 918
  события, 901
  шаговый, 909
Перечеркивание, 260
Перспектива, 872; 883
  точка схода, 886
Печать
  анимации, 980
  перехода, 925
  стиль, 1039
Пиксель, 151; 163
  плотность, 152
  размер, 151
Плотность точек, 1035
Поворот, 879
Повторение
  сетки, 711
  фона, 440
Подвал таблицы, 769
Подгонка, 698; 1023
  размера, 570
  ширины колонки, 709
Подсетка, 747
Подстановочный знак, 71
Подтягивание к верхнему краю, 251
Подчеркивание, 263
Позиционирование, 173; 551
  абсолютное, 563; 566
  линий сетки, 695
  липкое, 585
  маски, 1015
  относительное, 583
  радиального градиента, 493
  списка, 812
  строчного элемента, 313
  фиксированное, 581
  фона, 427
  цветов градиента, 477
  элементов на сетке, 738
Позиционная нумерация, 841
Поле, 286; 346
  замещаемого элемента, 333
  одностороннее, 349
  строчного элемента, 328
Полигональная форма, 543
Полоса сетки, 692
Пользовательские единицы измерения, 863
Пользовательское значение, 173
Поперечная ось, 609; 627
Поперечный размер, 609
Порядковая группа, 682
Порядок
  наложения элементов сетки, 761
  объявления стилей, 135
  расположения flex-элементов, 682
Последняя строка, 242
Последовательность анимаций, 948
Поток, 235; 286

grid-элементов, 736
пользовательского интерфейса, 977
Потомок, 79
Правило, 37
Предок, 79
Префикс, 38
Приоритетность, 125
анимации, 977
универсального селектора, 128
Пробел, 267
Производительность анимации, 949
Происхождение стиля, 135
Простой селектор, 68
Процент, 149; 300
Псевдокласс, 87; 1045
гиперссылок, 103
действий пользователя, 104
динамический, 102
отрицания, 116
пользовательского интерфейса, 108
целевых элементов, 114
языка документа, 115
Псевдоэлемент, 119; 813
Пункт, 150
Пустая ячейка, 782
Пустой элемент, 89

P

Радиальный градиент, 489
позиционирование, 493
сворачивание, 499
Радиан, 172
Развертка, 1036
Разделенные границы, 779
Размер
абсолютно позиционируемого элемента, 566
автоматический, 567
главный, 609
исходный, 660
маски, 1013
нулевой, 675
ограничение, 558
подгонка, 570
поперечный, 609
радиального градиента, 490
таблицы, 788
фонового изображения, 456
шрифта, 203
абсолютный, 204
автоматический, 212
наследование, 208
округление, 209
относительный, 205
процентный, 207
элемента, 556
Разметка, 37
Размещение элементов, 720; 732
Размытие, 982
Разрешающая способность, 1035
Разрешение, 152
Разрыв
страницы, 1046
возможный, 1051
принудительный, 1051
строки, 271; 276; 330; 817
Рамка, 419
рисованная, 378; 1022
толщина, 384
Распределение, 639
grid-элементов, 759
Растягивание, 648
flex-элементов, 629
Расширение шрифта, 218
Регистр символов, 77; 258
Режим
color, 995
color-burn, 993
color-dodge, 993
darken, 990
difference, 990
exclusion, 990
hard-light, 992
hue, 994
lighten, 990
luminosity, 995
multiply, 992
overlay, 992
saturation, 995
screen, 992
soft-light, 992
анимации, 969
маскирования, 1011
наложения, 990
Рисованная рамка, 1022
Родительский элемент, 79
Роль элемента, 289
Ряд, 766; 769
создание, 725

C

Светлота, 168; 995
Свободное пространство, 401
Свойство, 37

align-content, 635
align-items, 627
align-self, 627; 634
all, 143
animation, 971
animation-direction, 943
animation-duration, 940
animation-fill-mode, 969
animation-iteration-count, 941
animation-name, 937
animation-play-state, 968
animation-timing-function, 933; 935; 956; 964
backface-visibility, 887
background, 464
background-attachment, 451
background-blend-mode, 996
background-color, 416
background-image, 423
background-origin, 437
background-position, 427
background-repeat, 440
background-size, 456
border, 368
border-collapse, 779
border-color, 363
border-image, 392
border-image-outset, 388
border-image-repeat, 390
border-image-slice, 379
border-image-source, 378
border-image-width, 384
border-radius, 370; 538
border-spacing, 781
border-style, 356
border-width, 361
bottom, 554
box-decoration-break, 330
box-shadow, 509
box-sizing, 292
caption-side, 778
clear, 530
clip-path, 1001
clip-rule, 1007
color, 411
content, 816
counter-increment, 822
counter-reset, 822
direction, 282
display, 25; 288; 339; 767
empty-cells, 782
filter, 981
flex, 646; 676
flex-basis, 666
flex-direction, 599
flex-flow, 608
flex-grow, 648
flex-shrink, 655
flex-wrap, 606
float, 513
font, 228
font-family, 179
font-kerning, 220
font-size, 203
font-size-adjust, 212
font-stretch, 218
font-style, 215
font-synthesis, 227
font-variant, 221
font-weight, 195
grid, 744
grid-area, 732
grid-auto-columns, 742
grid-auto-flow, 736
grid-auto-rows, 742
grid-column, 725
grid-column-gap, 748
grid-gap, 750
grid-row, 725
grid-row-gap, 748
grid-template-areas, 714
height, 344; 556
hyphens, 271
isolation, 999
justify-content, 618; 624
left, 554
letter-spacing, 256
line-break, 274
line-height, 244; 324
list-style, 808
list-style-image, 805
list-style-position, 808
list-style-type, 802
margin, 402
mask, 1020
mask-clip, 1017
mask-composite, 1018
mask-image, 1008
mask-mode, 1011
mask-origin, 1016
mask-position, 1015
mask-repeat, 1014
mask-size, 1013
mask-type, 1021

max-height, 558
max-width, 558
min-height, 558
min-width, 558; 645
mix-blend-mode, 988
object-fit, 1023
object-position, 1025
order, 682
orphans, 1049
outline, 399
outline-color, 398
outline-style, 396
outline-width, 397
overflow, 560
overflow-wrap, 276
padding, 346
page, 1045
page-break-after, 1046
page-break-before, 1046
page-break-inside, 1048
perspective, 883
perspective-origin, 886
position, 551
quotes, 819
right, 554
shape-image-threshold, 546
shape-margin, 547
shape-outside, 534
size, 1042
table-layout, 788
tab-size, 270
text-align, 239; 314
text-align-last, 242
text-decoration, 260
text-indent, 236
text-justify, 242
text-orientation, 281
text-rendering, 264
text-shadow, 266
text-transform, 258
top, 554
transform, 857
transform-origin, 877
transform-style, 880
transition, 914
transition-delay, 911
transition-duration, 903
transition-property, 897
transition-timing-function, 905
unicode-bidi, 283
vertical-align, 248; 323
visibility, 562; 924; 935
white-space, 267
widows, 1049
width, 344; 556
word-break, 273
word-spacing, 254
writing-mode, 277; 605
z-index, 577
анимируемое, 892; 920; 933; 1055
вендорное, 38
наследование, 142
неанимируемое, 935
смещения, 553
Сгругление углов, 370
Секунда, 172
Селектор, 37
атрибутов, 68
всех соседних элементов, 86
группирование, 56
дочерних элементов, 83
единственных дочерних элементов, 90
идентификаторов, 61; 65; 129
классов, 61
ключевых кадров, 928; 930
контекстный, 80
корневого элемента, 89
нумерованных элементов, 97
обязательных элементов управления, 111
приоритетность, 126
проверяемых элементов управления, 111
псевдоклассов, 87
пустых элементов, 89
соседних элементов, 84
с точным значением, 69
с частичным значением, 71
тип, 66
универсальный, 57; 63
элемента, 52
элементов управления по умолчанию, 110
Семейство шрифтов, 177
Сепия, 984
Сетка, 688
заполнение элементами, 737
неявная, 728
повторение, 711
привязка элементов, 720
явная, 693
Сжатие, 655
формы элемента, 536
Символьная нумерация, 837

Система
координат, 853
нумерации, 840
письма, 603
Системный шрифт, 231
Скругление углов, 537
Слой, 761
таблицы, 776
фона, 467
Смешивание цветов, 988
Смещение, 553
фонового изображения, 432
цветов градиента, 481
элемента, 863
Событие
animationend, 946; 947; 955; 979
animationiteration, 947; 979
animationstart, 946; 978
transitionend, 901; 924
анимации, 946
перехода, 901
Совмещение границ, 783
Содержащий блок, 288; 517; 552
Содержимое
неоформленное, 185
перетекание, 339
Сокрытие элемента, 562
Состояние элемента управления, 109
Специфичность, 125
Список, 311; 801
вложенный, 829
границы, 810
Спрайт, 960
Средняя линия строки, 252
Статическое положение, 568
Стек, 577; 578
Стилевое правило, 37
Стиль
важный, 130
встроенный, 36
подключение, 29
происхождение, 135
Страница, 1041
именованная, 1044
разрыв, 1046
Строка, 144
высота, 229; 244
разрыв, 271; 817
Строчно-блочный элемент, 287; 336
Строчный маркер, 804
Строчный элемент, 24; 235; 287; 312
замещаемый, 332
контейнер, 316
Структура документа, 28; 78
Сферическая система координат, 856
Схлопывание отступов, 307; 405; 689
Счетчик, 821
область действия, 826
шаблон, 829

Т

Таблица, 765
автоматическое заполнение, 773
анонимный элемент, 772
высота, 795
размер, 788
стилей, 29; 37
альтернативная, 31
связывание, 33
Табуляция, 270
Тень, 509; 982
радиус размытия, 266
текста, 266
Технические характеристики, 46
Тип
маски, 1021
селектора, 66
списка, 802
элемента, 289
Точка схода, 886
Трансформация, 853
исходная точка, 877
трехмерная, 880

У

Угол, 171
оформление, 376
поворота, 867
скругление, 370; 537
Удельный коэффициент, 659
Узор, 507
Умножение цветов, 992
Универсальный селектор, 57; 63; 128
Упротнение шрифта, 218

Ф

Фигура, 536
Фиксирование фона, 451
Фиксированное позиционирование, 581
Фильтр, 981
базовый, 982
цветовой, 984
Фон, 416

наложение, 996
наследование, 425
обрезка, 420; 450
повторение, 440
позиционирование, 427
прозрачный, 417
фиксирование, 451
Фоновое изображение, 423
повторение, 445
размер, 456
сохранение пропорций, 460
Фоновый слой, 467
Фоновый цвет, 416
Форма
обтравочная, 1002
полигональная, 543
Формат
бумаги, 1043
шрифта, 186
Форматирование
вертикальное, 302
горизонтальное, 293
первой буквы, 119
первой строки, 120
таблицы, 765
Фрагмент документа, 114
Функция
attr(), 161
blur(), 982
brightness(), 986
calc(), 160; 698
circle(), 1002
contrast(), 986
counter(), 823
counters(), 823
drop-shadow(), 982
ellipse(), 1003
fit-content(), 707
format(), 185
grayscale(), 984
hue-rotate(), 985
inset(), 1002
invert(), 985
linear-gradient(), 475; 481
matrix(), 873
matrix3d(), 875
minmax(), 697; 702; 742
opacity(), 982
perspective(), 872
polygon(), 543; 1003
radial-gradient(), 489
repeat(), 709
rgb(), 163
rotate(), 866
rotateX(), 866
rotateY(), 866
rotateZ(), 866
saturate(), 986
scale(), 865
scale3d(), 866
scaleX(), 865
scaleY(), 865
scaleZ(), 865
sepia(), 984
skew(), 871
skewX(), 871
skewY(), 871
steps(), 960
translate(), 863
translate3d(), 864
translateX(), 862
translateY(), 862
translateZ(), 864
url(), 986
анимации, 956
временная, 905; 956; 964
масштабирования, 865
матричная, 873
наклона, 871
плавности, 905; 908
смещения, 862
трансформации, 858; 862
шаговая, 960

Ц

Цвет, 161; 411
градиента, 476
границ, 414
наследование, 416
фоновый, 416
элемента управления, 415
Цветность, 995
Цветовая модель, 163
Цветовое инвертирование, 418
Цветовой компонент, 989
Цветовой переход, 480
Цветовой тон, 168; 985; 994
Цветовой фильтр, 984
Целевой элемент, 114
Целочисленное значение, 148
Цицеро, 150

Ч

Частота, 172
Число, 148

Ш

Шаговая анимация, 960
Шаговый переход, 909
Шаг счетчика, 823
Шапка таблицы, 769
Шестнадцатеричный цвет, 166
Ширина
  области визуализации, 1034
  отображаемой области, 1033
  таблицы, 789; 791
  табуляции, 270
  элемента, 293; 299; 344; 556
Шрифт, 177
  FOSS, 189
  OpenType, 224
  без засечек, 178
  варианты, 221
  декоративный, 178
  замена, 232
  курсивный, 178
  моноширинный, 178; 209
  насыщенность, 195
  начертание, 197; 214
  особенности, 224
  размер, 203
  расширение, 218
  с засечками, 178
  системный, 231
  уплотнение, 218
  формат, 186

Э

Экранирование цветов, 992
Элемент, 23
  img, 23
  link, 29
  style, 33
  блочный, 292
  вращение, 856
  выравнивание, 514
  высота, 344
  гибкий, 592
  дочерний, 79
  замещаемый, 287; 301; 575
  изолирование, 999
  контейнер, 343
  корневой, 287
  неделимый, 861
  незамещаемый, 287; 570
  нумерованный, 97
  обратная сторона, 887
  повторяемый, 1052
  размер, 556
  родительский, 79
  роль, 289
  сокрытие, 562
  списка, 311
  строчно-блочный, 336
  строчный, 312
  таблицы, 765; 768
  тип, 289
  целевой, 114
  ширина, 344
Элемент управления, 415
  доступный, 109
  обязательный, 111
  по умолчанию, 110
  проверяемый, 111
  с атрибутом чтения-записи, 113
  состояние, 109
Эллипс, 538

Я

Явная сетка, 693
Язык документа, 115
Яркость, 986
Ячейка
  границы, 780
  пустая, 782
  сетки, 692
  таблицы, 766; 769

CSS
КАРМАННЫЙ СПРАВОЧНИК
ВИЗУАЛЬНОЕ ПРЕДСТАВЛЕНИЕ ВЕБ-СОДЕРЖИМОГО
5-Е ИЗДАНИЕ

Эрик А. Мейер

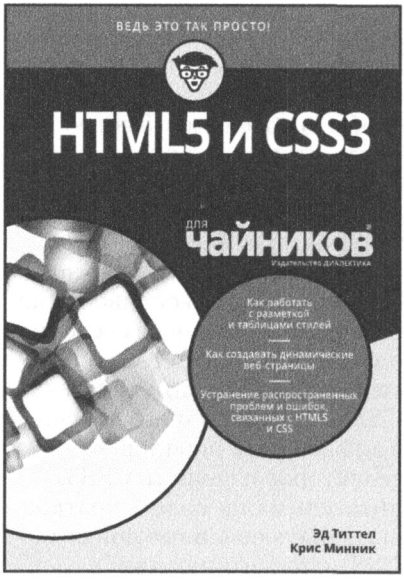
www.williamspublishing.com

Этот карманный справочник составлен таким образом, чтобы читатель мог быстро найти ответы на вопросы, возникающие во время работы с каскадными таблицами стилей (CSS), не обращаясь непосредственно к спецификации. В справочнике содержатся основные сведения о типах значений, селекторах, свойствах и прочих средствах CSS. Помимо полного перечня в алфавитном порядке селекторов и свойств CSS3, читатель найдет здесь краткое введение в основные понятия CSS, а также описание таких новых возможностей, как сетки, гибкие блоки, маски и фильтры.
Книга рассчитана на широкий круг читателей, интересующихся веб-разработкой с применением средств CSS.

ISBN 978-5-6041394-1-7 **в продаже**

HTML5 И CSS3 ДЛЯ ЧАЙНИКОВ

Эд Титтел,
Крис Минник



www.dialektika.com

Полагаете, будто создавать веб-сайты очень сложно? Ошибаетесь! С появлением HTML5, новейшей версии стандарта HTML, создавать и изменять дизайн веб-страниц стало проще, чем когда-либо. С помощью этой замечательной книги, написанной простым и понятным языком, вы освоите искусство веб-дизайна, изучите основы HTML5 и CSS3 и научитесь создавать собственные сайты.

Основные темы книги:

- создание веб-страниц;
- форматирование веб-страниц с помощью (X)HTML;
- просмотр и публикация веб-страниц в Интернете;
- применение метаданных поисковыми системами;
- управление текстовыми блоками, списками и таблицами;
- создание ссылок на документы и другие веб-сайты;
- настройка стилевых правил CSS;
- что можно, а чего нельзя делать с помощью HTML.

ISBN 978-5-907114-90-6 **в продаже**

O'REILLY®

CSS: полный справочник

В первую очередь эта книга рассчитана на веб-дизайнеров и разработчиков мобильных приложений, которые заинтересованы в использовании новейших средств визуального оформления веб-страниц, характеризующихся широкими функциональными возможностями и повышенной производительностью. В новом издании справочника по CSS содержится подробное описание всех новых инструментов, добавленных в последнюю версию спецификации.

CSS — активно развивающаяся технология стилевого форматирования веб-содержимого, выводимого на экранах электронных устройств, распечатываемого на принтерах, воспроизводимого через синтезаторы речи и с помощью экранных дикторов, а также отображаемого в окнах всевозможных онлайн-мессенджеров. Поддержка CSS внедрена во все современные браузеры, что позволяет применять стилевую разметку на экранах любых размеров и любых типов электронных устройств, включая компьютеры, смартфоны, игровые приставки, телевизоры, интерактивные терминалы и даже умные часы. Авторы книги поделятся секретами настройки взаимодействия с пользователями, быстрого оформления документов и предотвращения возможных неполадок. Вы узнаете, как оживить собственные веб-приложения, снабдив их продуманным и привлекательным пользовательским интерфейсом, в котором задействуются переходы, анимации, рамки, фоновые изображения и другие стилевые средства.

Основные темы книги:

- Селекторы, приоритетность и каскадирование стилевых правил
- Значения, единицы измерения, шрифты и форматирование текста
- Поля, отступы, границы и внешние контуры
- Цвета, фон и градиенты
- Выравнивание и позиционирование элементов
- Адаптивная верстка
- Верстка по сетке
- Двух- и трехмерные трансформации, переходы и анимация
- Фильтры, режимы наложения, обрезка и маскирование элементов
- Медиа-запросы

Эрик Мейер — всемирно известный эксперт по HTML, CSS и веб-стандартам. Основатель компании Complex Spiral Consulting, один из разработчиков формата презентаций S5 и учредитель серии конференций по веб-дизайну An Event Apart. Автор целого ряда книг, посвященных CSS и веб-дизайну. Ведет собственный сайт meyerweb.com.

Эстелл Уэйл — профессиональный веб-разработчик, с 1999 года занимается проектированием сайтов с использованием веб-стандартов. Ведет блог на сайте standardista.com, посвященный CSS3, HTML5 и JavaScript, а также инструментам мобильного веб-дизайна. Регулярно выступает на конференциях по всему миру.

ISBN 978-5-907114-56-2

9 785907 114562 18125

Категория: веб-дизайн
Уровень: средний

ДИАЛЕКТИКА

www.williamspublishing.com