Уральский федеральный университет
имени первого Президента России Б. Н. Ельцина
Институт естественных наук и математики

С. И. СОЛОДУШКИН
И. Ф. ЮМАНОВА

Web и DHTML

Учебное пособие

МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РОССИЙСКОЙ ФЕДЕРАЦИИ
УРАЛЬСКИЙ ФЕДЕРАЛЬНЫЙ УНИВЕРСИТЕТ ИМЕНИ ПЕРВОГО
ПРЕЗИДЕНТА РОССИИ Б. Н. ЕЛЬЦИНА

С. И. Солодушкин, И. Ф. Юманова

Web и DHTML

Учебное пособие

Рекомендовано
методическим советом Уральского федерального университета
в качестве учебного пособия для студентов вуза, обучающихся
по направлениям подготовки 01.03.01 «Математика»,
01.03.03 «Механика и математическое моделирование»,
02.03.01 «Математика и компьютерные науки»,
02.03.02 «Фундаментальная информатика
и информационные технологии»

Екатеринбург
Издательство Уральского университета
2018

УДК 004.43(075.8)
С604

Рецензенты:
кафедра прикладной математики и технической графики Уральского государственного архитектурно-художественного университета (заведующий кафедрой доктор физико-математических наук, профессор С. С. Титов);
А. В. Созыкин, кандидат технических наук, заведующий отделом вычислительной техники Института математики и механики им. Н. Н. Красовского УрО РАН

Научный редактор
доктор физико-математических наук, профессор В. Г. Пименов (Уральский федеральный университет)

Солодушкин, С. И.

С604 Web и DHTML : учеб. пособие / С. И. Солодушкин, И. Ф. Юманова ; [науч. ред. В. Г. Пименов] ; М-во образования и науки Рос. Федерации, Урал. федер. ун-т. — Екатеринбург : Изд-во Урал. ун-та, 2018. — 128 с.

ISBN 978-5-7996-2410-1

Рассматриваются вопросы использования языка разметки гипертекста HTML и каскадных таблиц стилей CSS для верстки веб-страниц. Основное внимание уделяется подробному разбору сложных теоретических правил верстки. Излагаются формальные правила вычисления размеров и положения элементов на странице. Анализируются методы разработки адаптивных веб-страниц.

Для студентов, специализирующихся в области прикладной информатики, компьютерных наук и занимающихся разработкой программного обеспечения.

УДК 004.43(075.8)

ISBN 978-5-7996-2410-1

ОГЛАВЛЕНИЕ

ПРЕДИСЛОВИЕ

Учебное пособие «Web и DHTML» написано авторами на основе опыта чтения одноименного курса в Уральском федеральном университете. Цель курса — знакомство с основными понятиями верстки веб-страниц, как то: языки разметки, формальный синтаксис CSS, блочная модель, поток верстки, визуализация элементов и вычисление их размеров, схемы позиционирования, адаптивный дизайн.

Учебное пособие призвано помочь студентам в освоении курса «Web и DHTML» и отражает его структуру. Пособие разбито на главы. Каждая глава соответствует рассматриваемой на занятиях теме и содержит необходимые теоретические сведения, примеры, всесторонне иллюстрирующие теорию, и иногда листинги программ. Кроме того, в конце глав приводятся вопросы для самоконтроля.

При подготовке учебного пособия авторы в основном обращались к первоисточникам, т. е. к официальным стандартам и документации.

Глава 1
ОСНОВНЫЕ ПОНЯТИЯ И ИСТОРИЯ ЯЗЫКОВ РАЗМЕТКИ

Чтобы изучать курс «Веб», необходимо знать некоторые базовые определения, составить тезаурус по теме. Этому посвящена первая часть главы. Далее мы покажем, что основной контент, который отдает веб-сервер клиенту, — это html-документы, т. е. документы, написанные на языке гипертекстовой разметки. Соответственно, мы введем понятия разметки и языков разметки. Этому посвящена вторая часть главы.

После краткого исторического обзора и введения базовых понятий мы приступим к непосредственному изучению языка HTML, и начнем со структуры html-документа и режима работы браузера.

1.1. Введение. Определение Интернета и его служб

Следует сразу сказать, что веб и Интернет — это не одно и то же. Соответственно, будем вводить понятия, начиная с самых базовых, а именно с Интернета. При этом мы соблюдем хронологический порядок — Интернет появился до веба.

Чтобы ответить на вопрос «Что такое Интернет?», рассмотрим рис. 1. Это частичная карта Интернета, основанная на данных сайта www.opte.org/maps от 15 января 2005 г. Каждая линия соединяет два хоста, т. е. компьютера с маршрутизируемыми IP-адресами. Длина линии показывает временную задержку (время пинга) между

узлами. Карта представляет менее 30 % сетей класса C, доступных для сбора данных в 2005 г.; сегодня Интернет еще более сложный.

Рассмотрим выносной рисунок, показывающий в увеличенном масштабе скопление компьютеров внизу основного рисунка. Это компьютерная сеть, соединенная с остальным Интернетом внешним линком. Таким образом мы приходим к идее, что Интернет — это объединение компьютерных сетей. Более формально, *Интернет* — всемирная система объединенных компьютерных сетей, построенная на базе стека протоколов TCP/IP и маршрутизации пакетов данных. Таким образом, по сути, Интернет — это сеть сетей.

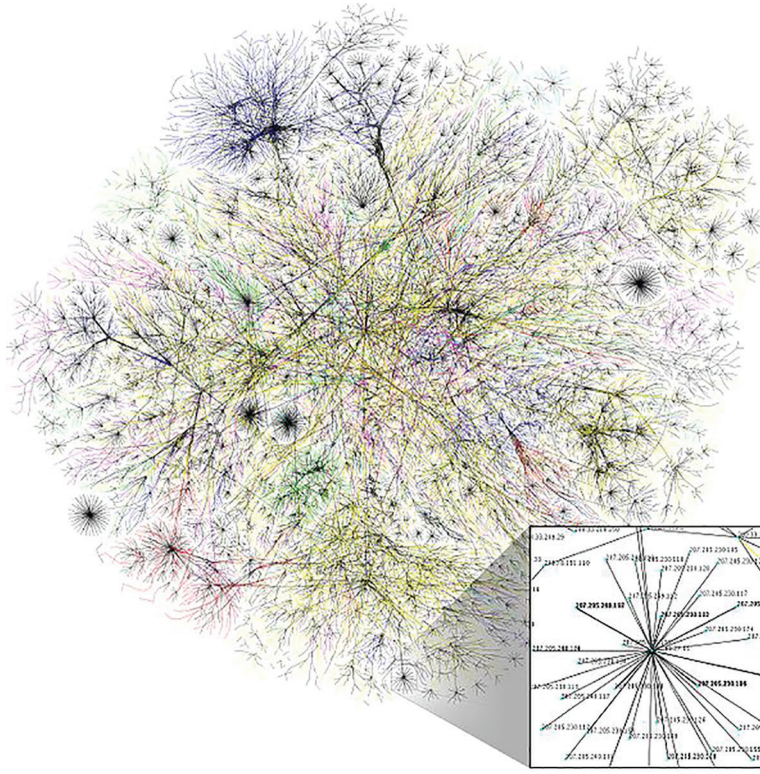


Рис. 1. Частичная карта Интернета
и отдельная сеть в увеличенном масштабе

Компьютерная сеть — это набор связанных между собой автономных компьютеров и/или компьютерного оборудования (серверы, маршрутизаторы и другое оборудование).

Благодаря использованию различных сетевых протоколов Интернет может обеспечить выполнение двух основных функций: 1) быть средством общения между удаленными пользователями и 2) быть средством доступа к общим информационным ресурсам, размещенным в Сети. Очевидно, что каждая из этих функций может быть реализована с помощью различных средств, что обеспечивает многообразие услуг, предоставляемых пользователям Интернета. Средства обеспечения определенных услуг для пользователей Интернета принято называть *службами Интернета*.

В обиходе слово «Интернет» часто употребляют как синоним «Всемирной паутины» и доступной в ней информации (т. е. службы веб и ее ресурсов), специалист, однако, должен понимать принципиальную разницу. Интернет является инфраструктурой для служб Интернета, предоставляя последним средства передачи данных (например, провода и протоколы нижних уровней, ответственных за надежную доставку данных). Следует сразу отметить, что веб не единственная служба Интернета; в Интернете существует большое количество сервисов, обеспечивающих работу со всем спектром ресурсов. Наиболее известными среди служб Интернета являются следующие:

1) World Wide Web (WWW, W3), или служба веб;

2) электронная почта (e-mail);

3) сервис DNS, или система доменных имен;

4) сервис FTP и BitTorrent, система файловых архивов;

5) сервис Telnet, предназначенный для управления удаленными компьютерами.

В настоящем учебном пособии подробно будет рассмотрена служба веб, работа веб-серверов и клиентов, способы и средства создания веб-сайтов.

Служба веб (англ. World Wide Web — всемирная паутина) — работающая по протоколу HTTP распределенная система, предоставляющая доступ к связанным между собой документам, расположенным на различных компьютерах, подключенных к Интернету.

Приведем выдержку из документации (http://www.w3.org/TR/html401/intro/ intro.html):

The World Wide Web is a network of information resources. The Web relies on three mechanisms to make these resources readily available to the widest possible audience:

- A uniform naming scheme for locating resources on the Web.
- Protocols, for access to named resources over the Web.
- Hypertext, for easy navigation among resources.

The ties between the three mechanisms are apparent throughout this specification.

В переводе данное определение звучит следующим образом. Служба веб — это сеть информационных ресурсов. Работа службы веб основана на трех механизмах, которые позволяют сделать эти ресурсы доступными максимально широкой аудитории:

- адресация с использованием URL (англ. Uniform Resource Locator),
- протокол для доступа к именованным ресурсам,
- гипертекст для легкой навигации между ресурсами.

Напомним, что HTTP (англ. HyperText Transfer Protocol — протокол передачи гипертекста) — протокол прикладного уровня передачи данных (изначально — в виде гипертекстовых документов в формате HTML, в настоящий момент используется для передачи произвольных данных). Основой HTTP является технология клиент-сервер, т. е. предполагается существование:

- клиентов, которые инициируют соединение и посылают запрос;
- серверов, которые ожидают соединения для получения запроса, производят необходимые действия и возвращают обратно сообщение с результатом.

Считается, что читатели знакомы с протоколом HTTP. Подробно ознакомиться с протоколом HTTP можно в учебном пособии С. И. Солодушкина, И. Ф. Юмановой «Разработка программных комплексов на базе протокола HTTP».

1.2. Понятие гипертекста

Что такое гипертекст, передачей которого занимается протокол HTTP? Говоря простым языком, гипертекст — это совокупность документов, содержащих текстовую, аудио- и видеоинформацию и связанных между собой взаимными ссылками в единый текст.

HTML (англ. HyperText Markup Language) — это язык, принятый в World Wide Web для создания и публикации веб-страниц. HTML предоставляет авторам средства для:

- форматирования текста, т. е. включения в веб-документы заголовков, текста, таблиц, списков, фотографий и т. п.;
- вставки гиперссылок для перехода к другим веб-страницам посредством щелчка кнопки мыши по гиперссылке;
- создания и заполнения форм для транзакций с удаленными службами, например, для поиска информации, бронирования билетов, оформления заказов на товары и т. п.;
- непосредственного включения в веб-документы видео, звука и других внешних объектов.

Фактически современная веб-страница формируется с помощью трех языковых средств:

- язык HTML используется для задания логической структуры документа (заголовки, абзацы, графические изображения и прочие объекты);
- язык каскадных стилей CSS используется для задания способа отображения документа (цвета текста и фона, шрифты, способы выравнивания и позиционирования отдельных объектов на странице и т. п.);
- языки программирования сценариев (чаще всего JavaScript) используются для написания сценариев, т. е. небольших программ, которые исполняются обозревателем в процессе отображения документа и обеспечивают его динамическое изменение в ответ на различные события.

При этом именно html-документ является той средой, в которой размещаются остальные компоненты веб-страницы.

Приведем пример простейшего html-документа:

```
<! DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0//EN">
<html>
    <head>
        <title>Мой первый документ</title>
    </head>
    <body>
        <p>Это html-документ.</p>
    </body>
</html>
```

Здесь наряду с информацией для пользователя типа «Это html-документ» есть управляющие конструкции, так называемые *теги*, они заключены в угловые скобки.

1.3. Понятие разметки. История языков разметки

Первоначально HTML был разработан приблизительно в 1986–1991 гг. Тимом Бернерсом-Ли (англ. Tim Berners-Lee) и его коллегами из CERN Laboratories (Швейцария) для обмена текстовыми документами и другими данными между учеными. Одновременно с этим Т. Бернерс-Ли разрабатывает протокол HTTP. Для того чтобы стать стандартом, протокол должен иметь не менее двух независимых реализаций, первую написал в 1990 г. сам Т. Бернерс-Ли.

22 апреля 1993 г. вышел браузер Mosaic, авторы которого добавили к нему поддержку графических образов и ряд других полезных качеств; благодаря Mosaic протокол HTTP стал популярным.

Быстрое развитие Сети в 90-е гг. потребовало стандартизации этого языка, и 22 сентября 1995 г. под эгидой IETF был создан HTML 2.0. Официальной спецификации HTML 1.0 не существует. До 1995 г. разработчики ранних браузеров выпускали свои корпоративные стандарты, чтобы официальная версия отличалась от них, ей сразу присвоили второй номер.

Версия 3, предложенная Консорциумом Всемирной паутины (W3C), обеспечивала много новых возможностей, таких как создание таблиц, «обтекание» изображений текстом и отображение сложных математических формул, поддержка gif-формата. Даже при том, что этот стандарт был совместим со второй версией, реализация его была сложна для браузеров того времени. Версия 3.1 официально никогда не предлагалась, и следующей версией стандарта HTML стала 3.2, в которой были опущены многие нововведения версии 3.0, но добавлены нестандартные элементы, поддерживаемые браузерами Netscape Navigator и Mosaic.

Наконец, в декабре 1997 г. появился HTML 4.0, который является в настоящее время действующим стандартом языка (в уточненной редакции HTML 4.01, опубликованной в декабре 1999 г.). В версии HTML 4.0 произошла некоторая «очистка» стандарта. Многие элементы были отмечены как устаревшие и нерекомендуемые (англ. deprecated). В частности, элемент font, используемый для изменения свойств шрифта, был помечен как устаревший (вместо него рекомендуется использовать таблицы стилей CSS).

Мертвой веткой является XHTML, однако сказать о ней стоит. В 1998 г. Консорциум Всемирной паутины начал работу над новым языком разметки, основанным на HTML 4, но соответствующим синтаксису XML. Впоследствии новый язык получил название XHTML. Первая версия XHTML 1.0 одобрена в качестве рекомендации Консорциумом Всемирной паутины 26 января 2000 г.

Планируемая версия XHTML 2.0 должна была разорвать совместимость со старыми версиями HTML и XHTML, но 2 июля 2009 г. W3C объявил, что полномочия рабочей группы XHTML2 истекают в конце 2009 г. Таким образом, была приостановлена вся дальнейшая разработка стандарта XHTML 2.0. Ресурсы рабочей группы были направлены на развитие HTML 5.

Стандарт HTML 5 был принят 28 октября 2014 г.

Итак, мы рассмотрели язык гипертекстовой разметки, обобщенный язык разметки… А что же такое «разметка»?

Язык разметки — набор символов или последовательностей, вставляемых в текст для передачи информации о его выводе или строении. Текстовый документ, написанный с использованием языка

разметки, содержит не только сам текст (как последовательность слов и знаков препинания), но и дополнительную информацию о различных его участках — например, указание на заголовки, выделения, списки и т. д.

Чтобы быть специалистом в сфере разработки программного обеспечения, необходимо иметь представление о разных компьютерных языках, в том числе о языках разметки. К тому же желательно знать историю языков разметки.

Различают логическую и визуальную разметки. В первом случае речь идет только о том, какую роль играет данный участок документа в его общей структуре (например, «данная строка является заголовком»). Во втором — определяется, как именно будет отображаться этот элемент (например, «данную строку следует отображать жирным шрифтом»).

Идея языков разметки состоит в том, что визуальное отображение документа должно автоматически получаться из логической разметки и не зависеть от его непосредственного содержания. Это упрощает автоматическую обработку документа и его отображение в различных условиях (например, один и тот же файл может по-разному отображаться на экране компьютера, мобильного телефона и на печати, поскольку свойства этих устройств существенно различаются). В жизни это правило часто нарушается: например, создавая документ в редакторе наподобие MS Word, пользователь может выделять заголовки жирным шрифтом, но нигде не указывать, что эта строка является заголовком.

Различают командную и теговую разметки. Ярким представителем языков командной разметки является LaTeX — язык, применяемый для набора и верстки сложных научных, в первую очередь математических, текстов. Приведем пример команд LaTeX для создания интеграла от минус бесконечности до нуля от дроби:

```
\int_{–\infty}^0\frac{x^2+1}{\sqrt{x+2+y^2}} dx
```

Идея использовать языки разметки в компьютерной обработке текстов, вероятнее всего, была впервые обнародована Вильямом Танниклиффом (англ. William W. Tunnicliffe) на конференции

в 1967 г. В 1970-е гг. В. Танниклифф руководил разработкой стандарта GenCode для издательской индустрии. Тем не менее в настоящее время «отцом» языков разметки обычно называют Чарльза Голдфарба (англ. Charles Goldfarb).

В 1969 г. компания IBM предложила Ч. Голдфарбу присоединиться к Кембриджскому научному центру и применить компьютеры в юридической практике. Точнее, этот проект требовал интеграции приложения по редактированию текстов с системой информационного поиска и программой по рендерингу (или оформлению) страниц. Документы должны были храниться в репозитории, откуда извлекаться по запросу. Извлеченные документы должны были быть доступными для редактирования в текстовом редакторе и сохранения в базе данных, или рендеринга, и подготовке к печати.

Эта стандартная на сегодняшний день задача было далека от решения в 1969 г. Далека настолько, что приложения, которые надо было интегрировать, не то что не были подготовлены для совместной работы, они не могли быть запущены в одной операционной системе. По ходу выяснилось, что для взаимодействия с каждой из программ требовалась разная процедурная разметка.

В 1969 г. Ч. Голдфарб формулирует основную концепцию будущего языка GML. Руководство IBM решает, что GML может использоваться не только в адвокатских конторах — будущий продукт имеет серьезный коммерческий потенциал, и засекречивает разработки до их окончания.

В 1973 г. Ч. Голдфарб публично представил GML. GML не зависел ни от марки компьютеров, ни от операционной системы, и IBM удалось перевести 90 % своей документации в этот формат.

В 1978 г. Ч. Голдфарб убедил руководство IBM использовать GML в коммерческих целях в составе разработанного компанией «средства формирования документов» (англ. Document Composition Facility), после чего GML несколько лет широко использовался в бизнесе.

GML (Generalized Markup Language — обобщенный язык разметки) — набор макросов, основной целью которых является реализация разметки, использующей теги для оформления текста.

При использовании GML документ помечается тегами, которые определяют характер текста (параграфы, заголовки, списки, таблицы). Такой документ может быть автоматически отформатирован для различных устройств, остается только указать разновидность устройства. Например, можно форматировать документ для лазерного принтера или матричного или просто вывести на экран, всего лишь указав профиль оборудования, без изменения самого документа.

Приведем пример GML-документа:

```
: h1.Chapter 1: Introduction
: p.GML supported hierarchical containers, such as
: ol
: li.Ordered lists (like this one),
: li.Unordered lists, and
: li.Definition lists
: eol.
as well as simple structures.
```

В 70-х параллельно языкам разметки развивалось понятие «гипертекст» (которое возникло раньше языков разметки), и развивалось в некотором смысле независимо от них. Иными словами, для создания гипертекстовых документов применялись не языки разметки, а иные средства.

В 1974 г. Теодор Нельсон предложил идею гипермедиа и начал активно ее разрабатывать. Он предложил организовывать взаимосвязанную сеть спрайтовых[1] картинок и даже создавать фильмы с меняющимся по требованию пользователя сюжетом.

Эту идею воплотила в 1978 г. в системе Aspen Movie Map группа ученых Массачусетского технологического института во главе с Андреем Липпманом. Система предлагала виртуальное путешествие по Аспену (США, штат Колорадо). Машина киногруппы предварительно объехала весь город, засняв всевозможные места

[1] CSS-спрайт — способ объединить много изображений в одно с целью сократить количество обращений к серверу.

под разными углами с помощью четырех камер, а затем фотографии были оцифрованы и заложены в Aspen. Пользователям было доступно не только множество навигационных средств, но и глобальная карта для быстрого переключения к нужной точке города (рис. 2 и 3). Заметим, что проект Google street view («Простор улиц») был запущен в мае 2007 г., спустя 30 лет!

К 1970 г. стало ясно, что для задач из разных предметных областей необходимы разные способы описания документов (говоря современным языком, не удастся обойтись ограниченным набором тегов разметки текста). Иными словами, невозможно создать язык с фиксированным набором тегов разметки, который будет одинаково хорошо справляться с задачами, вновь и вновь возникающими в разных предметных областях. Появилась идея создать метаязык, т. е. язык описания языков разметки. Принцип был такой: возникает задача из новой предметной области, с помощью метаязыка стандартным образом создается новый язык разметки, который будет предназначен для решения данной задачи и целого класса задач, ей эквивалентных.

Чтобы можно было включать одни документы в другие, авторы проекта решили ориентироваться на гипертекстовый способ организации данных. Основа для реализации сформулированных требований в виде GML уже существовала, но трудно было остановиться на конкретном технологическом решении.

К 1978 г. комитет по обработке информации Американского национального института стандартов (ANSI) всерьез заинтересовался языками подготовки гипертекстовых данных. Чарльз Голдфарб возглавил комитет по созданию мощного языка разметки документов, который получил название SGML (Standard Generalized Markup Language). В его основу был положен GML.

SGML создавался под влиянием идеи, что разметка должна быть сфокусирована на структурных аспектах документа и оставить внешнее представление документа интерпретатору. SGML точно определял синтаксис для включения разметки в текст, а также отдельно описывал, какие теги разрешены и где. Это давало возможность авторам создавать и использовать любую разметку, выбирая теги и давая им имена на нормальном языке. Таким образом, SGML

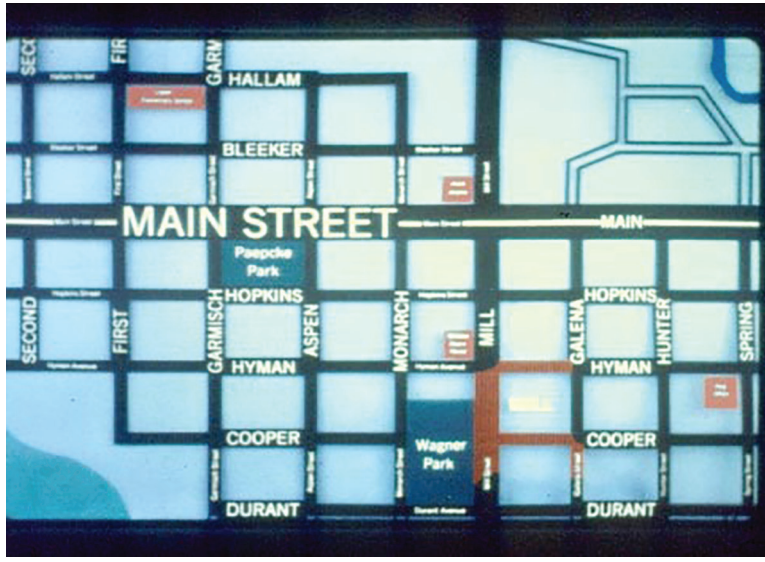


Рис. 2. Интерфейс системы Aspen Movie Map, карта

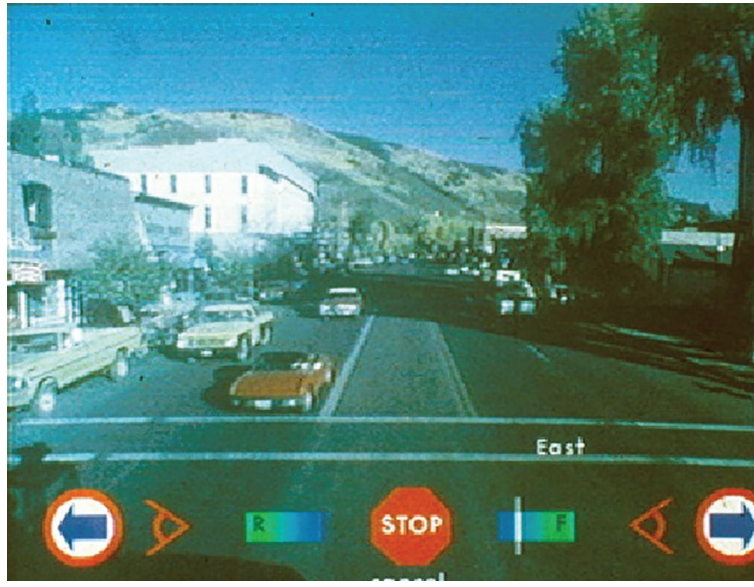


Рис. 3. Интерфейс системы Aspen Movie Map, фотография улицы

следует считать метаязыком; множественные специальные языки разметок произошли от него.

SGML разрабатывался для совместного использования машинно-читаемых документов в больших правительственных и аэрокосмических проектах. Он широко использовался в печатной и издательской сфере.

В 1980 г. появился первый рабочий вариант спецификации. В 1983 г. SGML принят в качестве промышленного стандарта, поддержанного Министерством обороны и Налоговым управлением США. Спустя два года сформировалась международная группа пользователей SGML. Так начался расцвет гипертекстовых технологий. В 1986 г. SGML принят как стандарт ISO 8879.

SGML (Standard Generalized Markup Language — стандартный обобщенный язык разметки) — метаязык для определения языков разметки документов. SGML предоставляет множество вариантов синтаксической разметки для использования различными приложениями.

Приведем пример sgml-документа, описывающего рецепт приготовления десерта на 6 персон:

```
<! DOCTYPE recipe SYSTEM "recipe.dtd">
<recipe type="dessert" ser="6" preptime="15">
<title>Two-Minute Fudge</title>

<ingredient-list>
<ingredient>1 pound of sugar</ingredient>
<ingredient>1/2 cup cocoa</ingredient>
…
</recipe>
```

Язык SGML создавался для решения больших и сложных задач общего вида. Авторы языка стремились сделать его очень гибким и всеобъемлющим, как следствие, язык получился чрезвычайно сложным. В то же время HTML создавался как язык для обмена научной и технической документацией, пригодный для использования людьми, не являющимися специалистами в области верстки.

HTML успешно справлялся с проблемой сложности SGML путем определения небольшого набора структурных и семантических элементов — дескрипторов (тегов). С помощью HTML можно легко создать относительно простой, но красиво оформленный документ. Помимо упрощения структуры документа, в HTML внесена поддержка гипертекста и мультимедийных возможностей. Успех HTML способствовал развитию и распространению языков разметки.

Чтобы пояснить соотношение и связи между языками, рассмотрим диаграмму (рис. 4). Язык SGML произошел от GML в том смысле, что испытал его идейное влияние. Выше мы сказали, что SGML — это метаязык, позволяющий создавать новые языки, одним из них является HTML. Таким образом, по определению HTML — это приложение SGML, реализация, язык, написанный на метаязыке SGML.

Язык HTML был ограничен в своих возможностях только разметкой гипертекста, сообществу потребовался язык, который сочетал бы в себе гибкость и мощь языка SGML (позволял бы описывать произвольные данные иерархической структуры) и простоту HTML. Так в августе 1996 г. началась работа над языком XML. Язык XML — это не приложение, а подмножество SGML, и таким образом сам является метаязыком; благодаря языку XML было создано великое множество новых языков разметки.

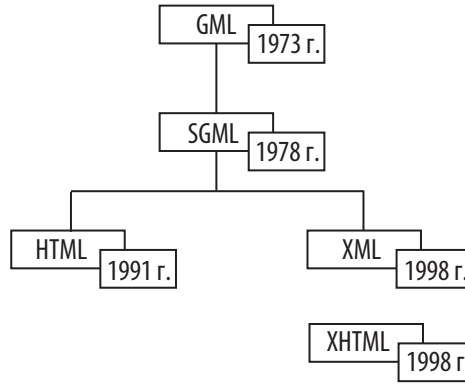


Рис. 4. История развития языков разметки

На конференции SGML-96 14 ноября 1996 г. рабочая группа представила предварительный вариант нового языка на 32 страницах. Уже 10 февраля 1998 г. XML 1.0 стал рекомендацией W3C.

Разработка началась с определения десяти положений, которым должен был соответствовать новый язык. Приведем некоторые из них.

1. XML должен напрямую использоваться в сети Интернет.
2. XML должен поддерживать разнообразные приложения.
3. XML должен быть совместим с SGML.
4. Разработка программ для обработки xml-документов не должна быть сложной задачей.

Спецификация xml описывает xml-документы и частично описывает поведение xml-процессоров (программ, читающих xml-документы и обеспечивающих доступ к их содержимому). XML разрабатывался как язык с простым формальным синтаксисом, удобный для создания и обработки документов программами и одновременно удобный для чтения и создания документов человеком, прежде всего нацеленный на использование в Интернете.

Язык называется расширяемым, поскольку он не фиксирует разметку, используемую в документах: разработчик волен создать разметку в соответствии с потребностями к конкретной области, будучи ограниченным лишь синтаксическими правилами языка.

Сочетание простого формального синтаксиса, удобства для человека, расширяемости, а также базирование на кодировках Юникод для представления содержания документов привело к широкому использованию как собственно XML, так и множества производных специализированных языков на базе XML в самых разнообразных программных средствах.

Рассмотрим пример:

```
<?xml version="1.0" encoding="WINDOWS-1251"?>
<?xml-stylesheet href="my-style.css"?>
<knowledgeDatabase>
<tutorial>
    <title>Учебник по XSL</title>
    <author>Иванов Иван Иванович</author>
```

```
</tutorial>
</knowledgeDatabase>
```

Объявление XML указывает версию языка, на котором написан документ. Поскольку интерпретация содержимого документа, вообще говоря, зависит от версии языка, то спецификация предписывает начинать документ с объявления XML. В первой (1.0) версии языка использование объявления не было обязательным, в последующих версиях оно обязательно. Таким образом, версия языка определяется из объявления, и если объявление отсутствует, то принимается версия 1.0.

Кроме версии XML, объявление может также содержать информацию о кодировке документа и «оставаться ли документу со своим собственным DTD или с подключенным» (по умолчанию "no"):

- Standalone = "yes" — значит документ будет пользоваться своим DTD,
- Standalone = "no" — значит DTD подключается извне.

Инструкции обработки (англ. processing instruction, PI) позволяют размещать в документе инструкции для приложений. В данном примере показана инструкция обработки, передающая xml-stylesheet-приложению (например, браузеру) инструкции в файле mystyle.css посредством атрибута href:

```
<?xml-stylesheet type="text/css" href="mystyle.css"?>
```

Далее следует корневой тег <knowledgeDatabase>, в который вложены все остальные теги по принципу матрешки.

В 1998 г. W3C начал работу над XHTML — новым языком разметки, основанным на HTML 4, но соответствующим синтаксису XML.

26 января 2000 г. W3C опубликовал рекомендацию XHTML 1.0.

В те годы назывались следующие преимущества XHTML в сравнении с HTML4:

1. Xhtml-документы соответствуют стандарту XML, и, следовательно, они могут просматриваться, редактироваться и проверяться на синтаксическую правильность стандартными средствами поддержки языка XML.

2. Xhtml-документы могут отображаться как существующими обозревателями html-документов, так и новыми обозревателями, поддерживающими стандарт XHTML.

3. Xhtml-документы могут обращаться к сценариям и аплетам, основанным на объектной модели документов (DOM).

С практической точки зрения выделялись следующие преимущества XHTML:

1. И разработчики документов, и разработчики обозревателей постоянно ищут новые способы выражения своих идей с помощью новых html-тегов. XML обеспечивает единый и простой способ создания новых элементов языка и их дополнительных атрибутов. XHTML призван унифицировать такие расширения языка HTML посредством xhtml-модулей, которые будут поддерживать комбинации существующих элементов HTML с новыми элементами при разработке и отображении документов.

2. Постоянно возникают все новые способы и средства доступа к Сети: карманные компьютеры и телевизионные приставки, сотовые телефоны и пейджеры. XHTML был разработан с ориентацией на обобщенный обозреватель, который в сочетании с механизмами словарей метаданных должен обеспечить оптимальное преобразование содержимого документа при его отображении, с тем чтобы в конце концов перейти к разработке таких документов, которые будут адекватно отображаться любым обозревателем, поддерживающим стандарт XHTML.

Как показало время, язык XHTML не получил своего развития, так как не был принят сообществом разработчиков.

1.4. Составные элементы html-документа и его структура

HTML — это теговый язык разметки документов, т. е. любой документ на языке HTML представляет собой набор элементов, причем начало и конец каждого элемента обозначаются специальными пометками, называемыми тегами.

Как правило, элементы имеют следующую структуру: открывающий тег, содержимое, закрывающий тег. Однако элементы могут быть пустыми, т. е. не содержащими никакого текста и других данных (например, тег перевода строки
).

Элементы могут иметь атрибуты, определяющие какие-либо их свойства (например, URL-адрес для элемента a). Атрибуты указываются в открывающем теге, как показано в примере:

```
<img src='lion.jpg' alt='лев в клетке' width='100px' height='100px' />
```

Каждый html-документ, отвечающий спецификации HTML какой-либо версии, должен включать:

1) строку, содержащую декларацию типа документа;

2) заголовок документа (заключенный в теги <head>…</head>);

3) тело документа (заключенное в теги <body>…</body> или <frameset>…</frameset>).

```
<! DOCTYPE html public "-//w3c//dtd html 4.0//en">
<html>
    <head>
        <title>мой первый html-документ</title>
    </head>
    <body>
        <p>это html-документ.</p>
    </body>
</html>
```

1.5. Doctype и режимы работы браузера

Существуют разные версии языка HTML — со второй по пятую; формально это не разные версии, а разные языки. Кроме того, существуют разные режимы работы браузеров: режим поддержки стандартов и режим имитации работы старых браузеров. Инструкция <! DOCTYPE> определяет, в соответствии с каким синтаксисом

необходимо разбирать данную веб-страницу, а также задает режим работы браузера.

Непонимание принципов работы <! DOCTYPE> приводит к сложно выявляемым ошибкам. Разберем несколько примеров. Рассмотрим работу браузера в режиме совместимости:

```
<! DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
<html>
<head>
    <title>Quirc mode</title>
    <style>
        ody {font-size: 20;}
        td {font-size: 200%;}
    </style>
</head>
<body>
    <table border='1'>
        <tr><td>
            <img src='lion.jpg' alt='лев' width='300px' height='168px'>
        </td></tr>
    </table>
    <p>Текст в абзаце</p>
    <table border='1'>
        <tr><td>
            Текст в таблице
        </td></tr>
    </table>
</body>
</html>
```

В заголовочной части заданы стили, указан размер шрифта для текста, расположенного в абзацах и в ячейках таблицы. Следует обратить внимание на то, что во втором случае размер шрифта указан в процентах (от размера некоторого базового шрифта). Непосредственной проверкой можно убедиться, что изменение размера базового шрифта (например, body {font-size: 10}) не приво-

дит к соответствующему изменению размеров шрифта в таблице, хотя стандарт предписывает иное. Это известная ошибка старых браузеров, которая перекочевала в новые версии для «сохранения преемственности».

Теперь заменим <! DOCTYPE> на строгий:

```
<! DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0//EN"
"http://www.w3.org/TR/REC-html40/strict.dtd">
```

Во-первых, размер базового шрифта перестанет реагировать на заданные в стилях правила. В режиме совместимости браузер предполагал, что если единицы измерения не указаны, то это пиксели по умолчанию. В строгом режиме таких предположений не делается, браузер игнорирует синтаксически неверную инструкцию.

Если задать единицы измерения (body {font-size: 10px}), то браузер а) примет инструкцию по размеру базового шрифта, б) будет вычислять размер шрифта в таблице относительно базового.

Во-вторых, в строгом режиме под изображением в ячейке таблицы появится отступ, поскольку изображение в строгом режиме обрабатывается как строчный[2] элемент.

Структура элемента <! DOCTYPE> подчинена правилам:

```
<! DOCTYPE [Элемент верхнего уровня] [Публичность] "[Регистрация] //
[Организация]//[Тип] [Имя]//[Язык]" "[URL]">
```

1. Параметр Публичность указывает на то, каким ресурсом является объект — публичным (значение PUBLIC) или системным (значение SYSTEM), например, таким как локальный файл. Для HTML/XHTML указывается значение PUBLIC.

2. Параметр Регистрация сообщает, что разработчик DTD зарегистрирован в международной организации по стандартизации (International Organization for Standardization, ISO). Принимает одно из двух значений: плюс (+) — разработчик зарегистрирован в ISO и минус (–) — разработчик не зарегистрирован. Для W3C ставится значение «–».

[2] О строчных и блочных элементах будет рассказано в следующих лекциях.

3. Параметр Организация — уникальное название организации, разработавшей DTD. Официально HTML/XHTML публикует W3C, это название и пишется в <! DOCTYPE>.

4. Параметр Тип — тип описываемого документа. Для HTML/XHTML значение указывается DTD.

5. Параметр Имя — уникальное имя документа для описания DTD.

6. Параметр Язык — язык, на котором написан текст для описания объекта. Содержит две буквы, пишется в верхнем регистре. Для документа HTML/XHTML указывается английский язык (EN).

7. Параметр URL — адрес документа с DTD.

Приведем примеры использования <! DOCTYPE> для HTML 4.01.

Строгий вариант, или стандартный (англ. Strict), означает, что документ не содержит элементов, помеченных Консорциумом W3C как «устаревшие» или «неодобряемые» (англ. deprecated).

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

Переходный вариант, или совместимости (англ. Transitional), означает, что документ может содержать устаревшие теги для совместимости и упрощения перехода со старых версий HTML.

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

Приведем примеры, показывающие различие режимов (табл. 1).

В HTML 5 используется только один вариант <! DOCTYPE>:

```
<! DOCTYPE html>
```

Синтаксические правила построения документов разметки гласят, что теги должны быть правильно вложены т. е. так, как это иллюстрирует пример:

```
<u> <b> Корректная вложенность тегов </b> </u>
<u> <b> Ошибка </u> </b>
```

Таблица 1

Режимы работы браузеров

Strict	Transitional
Чувствительность к регистру имен классов и идентификаторов	
Чувствительны к регистру	Нечувствительны к регистру
Пиксели по умолчанию	
Числа без указания единиц измерения игнорируются. Если в стилях при указании размеров задано одно лишь число без упоминания единиц (width: 500 вместо width: 500 px), такое значение игнорируется	Если в стилях в качестве единицы размера указано число без единицы измерения (10), считается, что значение задано в пикселях (как 10 px)
Свойство display для изображений	
По умолчанию установлено как inline, при этом внизу картинок добавляется небольшой отступ	По умолчанию установлено как block, при этом внизу картинок отступ отсутствует

Правильная вложенность тегов, вообще говоря, не обеспечивает синтаксическую корректность документа. Пример ниже показывает, что в документе могут существовать неизвестные теги, теги могут иметь недопустимые атрибуты, и вложенность тегов может быть неверной с точки зрения здравого смысла — действительно, строка не должна быть вложена в ячейку таблицы.

```
<body>
    <unknowntag>
        Сомнительный текст
    </unknowntag>
    <table src="figure.jpg">
        <td>
            <tr>Строка в ячейке</tr>
        </td>
    </table>
</body>
```

Описать, какие элементы и где могут присутствовать в документе, какие элементы могут быть вложены в другие элементы и т. д. позво-

ляет язык схем DTD. Язык схем DTD — язык, который используется для записи фактических синтаксических правил языков разметки текста. Приведем фрагмент dtd-документа, который показывает, что элемент TR должен содержать один и более элементов TD или TH:

```
<! ELEMENT TR          – O (TH|TD)+ >
```

Поясним, что дефис означает обязательность открывающего тега TR, а «O» — необязательность закрывающего. Еще один пример показывает, что обязательные атрибуты тега img — это src и alt, в то время как height и width — необязательные:

```
<! ATTLIST IMG
    src      %URI;      #REQUIRED
    alt      %Text;     #REQUIRED
    height   %Length;   #IMPLIED
    width    %Length;   #IMPLIED
```

Используя DTD, браузер, в частности, понимает, что именно ячейка таблицы должна быть вложена в строку, а не наоборот.

1.6. Тег html, атрибут manifest

Тег html является контейнером, который заключает в себе все содержимое веб-страницы, включая теги head и body. Среди атрибутов тега html интересным является manifest, введенный в HTM 5.

Атрибут manifest реализует механизм кэширования, который позволяет создавать оффлайновые приложения, т. е. работающие в автономном режиме без непосредственного подключения к Интернету. При первой загрузке страницы браузер обычно просит сохранить данные для своей работы, а затем уже обращается к ним при необходимости.

В качестве значения атрибута manifest указывается относительный или абсолютный путь к текстовому файлу, он называется «файл манифеста» или просто «манифест». Имя и расположение файла

может быть любым, но он должен отдаваться сервером с заголовком text/cache-manifest. Например, для веб-сервера Apache в файле .htaccess, расположенном в корне сайта, следует прописать строку

```
AddType text/cache-manifest .cache
```

В этом случае файл манифеста имеет расширение cache. Сам манифест информирует браузер о том, какие ресурсы необходимо сохранить в локальном кэше. Этот список может содержать html- и css-файлы, изображения, скрипты. Манифест начинается с обязательной строки CACHE MANIFEST. Имена файлов перечисляются внутри трех секций (табл. 2).

Пример файла манифеста:

```
CACHE MANIFEST
# Версия 1.0

CACHE:
22.css
22.js
images/old.png
```

Таблица 2

Секции манифеста

CACHE	Содержит перечень ресурсов, которые браузер должен кэшировать. Данный раздел используется по умолчанию, если его не указать и нет других разделов, то записи в манифесте причисляются к этому разделу
NETWORK	Список ресурсов, которые доступны только при подключении к Сети. Обычно в этот раздел входят программы, выполняемые на стороне сервера
FALLBACK	Список замещающих файлов, которые будут использоваться при отсутствии подключения к Сети. Допустимо использовать символ * для обозначения всех файлов. Например, *.php /offline.html означает, что вместо любого файла с расширением php будет показана страница offline.html

Внутри тега html находятся теги head и body. Тег head предназначается для размещения в нем некоторой служебной информации, которая не отображается пользователю, в свою очередь, body содержит данные, отображаемые на странице. Тег head содержит:

1) единственный титул документа (title);
2) метаописатели документа (meta);
3) базовый uri внешних ссылок (base);
4) ссылки на другие документы (link);
5) внутренние таблицы стилей (style);
6) сценарии клиента (script).

Структура тега head подробно описана в учебниках и справочниках, куда мы и адресуем читателей.

Контрольные вопросы

1. Рассмотрим html-документ, содержащий, например,

```
<! DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
```

а) Зачем браузеру нужна эта строка кода?

б) Здесь прописан URL-адрес, откуда можно загрузить DTD. Загружает ли браузер DTD из Интернета? А если доступа к Сети нет, откуда браузер берет DTD?

2. Иногда полезно добавлять в html-теги «пользовательские» атрибуты. Это приводит к невалидному html4-коду. Пример технологии, которая приводит к невалидному HTML, но имеет огромное значение, — это ARIA. В чем суть этой технологии, для чего она нужна?

Есть ли возможность определять валидные «пользовательские» атрибуты в HTML5? Если да, то какая?

Указание. См.: http://www.w3.org/WAI/intro/aria

3. Что проверяет валидатор (x)html-кода? Чем руководствуется? Для чего нужна валидация html-кода? Приведите аргументы «за» и «против» валидного кода.

Глава 2
КАСКАДНЫЕ ТАБЛИЦЫ СТИЛЕЙ

Настоящая глава посвящена основным понятиям каскадных таблиц стилей CSS (англ. Cascading Style Sheets). Мы разберем, что такое CSS и почему эта технология появилась в свое время. После этого перейдем к изучению основных понятий.

Вопросы типа «Какой селектор более специфичный?» и «Что такое наследование?» часто встречаются на собеседованиях с претендентами на должность веб-разработчиков (особенно фронтэнда).

2.1. Cascading Style Sheets

В первой главе мы выяснили, что HTML нужен для форматирования гипертекстовых документов, т. е. выделения структурных логических единиц — заголовков, абзацев, списков и т. д. В этой главе рассмотрим оформление документов.

Прежде всего необходимо различать два понятия — форматирование и оформление. Здесь уместно обратиться к примеру текстового редактора MS Word. Он относится к классу WYSIWYG редакторов (англ. what you see is what you get — что видишь, то и получаешь); это означает, что процессы верстки и просмотра совмещены в одно целое. Напротив, верстка в LaTeX отделена от просмотра: сначала автор с помощью команд разметки верстает документ, потом транслирует его и просматривает получившийся результат.

В большом документе мы выделяем отдельные фрагменты и делаем их заголовками, используя стили; после чего а) можно собрать оглавление, б) один раз для стиля «заголовок первого уровня» настроить правила оформления (цвет, размер, выравнивание, подчеркивание) — и все заголовки изменят свой внешний вид.

С другой стороны, когда пользователь создает одностраничный документ, он зачастую просто выделяет фрагмент текста полужирным шрифтом, устанавливает размер шрифта и центрирует, т. е. использует средства оформления.

Надо понимать, в чем различие между первым и вторым вариантом. Заголовок — это структурная единица, а выделенный полужирным шрифтом фрагмент особой логической, структурной единицей не является.

Аналогично структурной единицей является абзац — отдельная законченная мысль. Текст, который отбит от соседнего текста отступами, внешне может выглядеть как абзац, но по сути таковым не являться.

В первом случае мы говорим о форматировании, а во втором — об оформлении. В начале 90-х гг. создатели веба осознали, что необходимо отделять форматирование от оформления. В результате были созданы каскадные таблицы стилей CSS.

CSS — формальный язык описания внешнего вида документа, написанного с использованием языка разметки. Как правило, такими языками являются HTML и XML.

В начале 90-х для оформления текста на веб-страницах использовался тег font, который позволяет менять цвет текста или его начертание. Также использовались теги i и b, которые сегодня не рекомендуются к использованию (англ. deprecated).

Надо понимать, что языки и технологии (в том числе CSS) создаются в контексте некоторых исторических реалий и нужна достаточно сильная мотивация для создания нового языка. Давайте вернемся во времена становления веба (начало 90-х), чтобы понять, насколько плохо раньше обстояли дела со средствами оформления веб-страниц. Разберем это на примере тега font.

```
<font face="verdana" color="green">some text</font>
```

Здесь атрибут face имеет значение verdana, что соответствует шрифтам без засечек, атрибут color имеет значение green, а потому цвет текста будет зеленым.

Надо признать, что такой способ оформления веб-страниц очень плохой: перемешивается логическая разметка и визуальное оформление. Если дизайн сайта потребует многократно применить такой стиль оформления, то тег font с соответствующими атрибутами придется многократно вставлять в текст. Это делает решение немасштабируемым.

Кроме того, если в дальнейшем потребуется изменить цвет текста с зеленого на красный, то потребуется везде сделать изменения. Автозамена в этом случае не поможет, так как строка «зеленый» заменится на «красный» везде, даже там, где не надо. Таким образом, использование тега font делает решение негибким.

2.2. Элемент STYLE

Начнем знакомство с CSS с примера. Заметим, что до тех пор, пока не предусмотрено никакого специального оформления, используются настройки браузера по умолчанию. Сейчас ответим на вопросы «Как стили создавать?» и «Где их создавать?».

Поскольку размещать вместе правила оформления и верстки — плохое решение, возникает вопрос, куда их выносить. Существует несколько вариантов. Можно во внешних файлах описать стили и потом подключить их к html-документу, можно (как в данном примере) — внутри html-документа, внутри тега style в головной части, т. е. внутри тега head.

```
<html>
    <head>
        <title>Ter STYLE</title>
        <style>
            h1 {  font-size: 40px;
                  font-family: Verdana, sans-serif;
                  color: #333366;}
```

```
            p {color: green;}
        </style>
        </head>
        <body>
            <h1>Hello, world!</h1>
            <p>Текст абзаца зеленого цвета</p>
        </body>
    </html>
```

Тег style нужен для того, чтобы определить стили, которые будут использоваться внутри данной страницы.

Что в этом стиле мы видим? Для заголовков первого уровня определены размер шрифта, начертание и цвет, для абзацев задан только цвет (соответственно, значения иных свойств, в том числе размера шрифта для абзацев, берутся из настроек браузера по умолчанию).

С помощью тега style можно вынести описание стилей, использующихся на веб-странице, в заголовочную часть документа. Это позволяет сделать решение более масштабируемым и гибким, не разбрасывать описание стилей по странице. Однако это решение половинчатое, с оформлением одной страницы мы справились, но что делать, если таких одинаково оформленных страниц сотня? Неужели придется вставлять однообразные описания стилей в каждый html-документ? Ответ — разумеется, нет. Если бы так было, то решение нужно было признать по-прежнему немасштабируемым, смена дизайна сайта потребовала бы внесения изменений во все html-документы.

2.3. Включение таблиц стилей в документ

Для подключения к документу внешней таблицы стилей (т. е. таблицы стилей, хранящейся в отдельном файле) следует поместить в заголовок документа элемент LINK, как показано в примере.

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

Содержимое файла main.css:

```
h1 {  font-size: 40px;
      font-family: Verdana, sans-serif;
      color: #333366;}
p {   font-size: 14px;
      text-align: center;}
```

Во внешних таблицах стилей следует описывать стили, которые применяются ко многим html-документам. Внешние таблицы можно подключать с помощью директивы @import, но подробнее об этом мы поговорим позже.

Если конкретная страница имеет свои индивидуальные правила оформления, их можно описать во внутренних стилях.

Если конкретный элемент страницы требует определения индивидуальных правил, то можно описать его стили непосредственно в открывающимся теге, используя атрибут style. Обратим внимание читателей на то, что тег style и атрибут style — разные вещи по сути, хотя и называются одинаково. Использовать третий способ надо с осторожностью, ибо переплетение стилей и верстки может привести к сложно поддерживаемому коду.

Сразу обратим внимание на то, что для одного и того же элемента веб-страницы стилевые правила могут быть описаны и во внешних таблицах, и во внутренней (в заголовочной части в теге style), и с использованием атрибута style. Таким образом, может возникнуть конфликт определений. В этом нет ничего плохого, это обычная практика написания стилей. Как этот конфликт разрешается, детально разберем далее в этой главе, а сейчас просто отметим, что внешние таблицы стилей имеют самый низкий приоритет, внутренние — выше, а стили элемента — самый высокий.

В элементах link и style можно дополнительно указать типы устройств, на которые распространяется данная таблица стилей, например:

```
<link rel="stylesheet" href="sc.css" media="screen">
<link rel="stylesheet" href="prn.css" media="print">
```

```
<style media="screen">
    h1 {color: #333366;}
</style>
<style media="print">
    h1 {text-align: center;}
</style>
```

В табл. 3 перечислены типы устройств, для которых можно применять стили.

Таблица 3

Значения атрибута media и описание

aural	синтезатор речи
print	принтер
projection	проектор
screen	графический дисплей
tv	телевизор

2.4. Исторический обзор

Теперь, когда составлено общее представление о том, что такое CSS, для чего он нужен и как подключаются к html-документу, сделаем исторический обзор.

Термин «каскадные таблицы стилей» был предложен Хоконом Виумом Ли в 1994 г. Совместно с Бертом Босом он стал развивать CSS. В отличие от многих существовавших на тот момент языков стиля, CSS использует наследование от родителя к потомку, поэтому разработчик может определить разные стили, основываясь на уже определенных ранее стилях.

В середине 1990-х Консорциум Всемирной паутины (W3C) стал проявлять интерес к CSS, и в декабре 1996 г. была издана рекомендация CSS1.

Среди возможностей, предоставляемых рекомендацией CSS1, были:

- параметры шрифтов, позволяющие задавать гарнитуру и размер шрифта, а также его стиль — обычный, курсивный или полужирный;
- цвета, позволяющие определять цвета текста, фона, рамок и других элементов страницы;
- атрибуты текста, позволяющие задавать межсимвольный интервал, расстояние между словами, высоту строки и межстрочные отступы;
- выравнивание для текста, изображений, таблиц и других элементов;
- свойства блоков, позволяющие задавать высоту, ширину, внутренние и внешние отступы и рамки.

Также в спецификацию входили ограниченные средства по позиционированию элементов.

Принятая в мае 1998 г. CSS2 основана на CSS1 с сохранением обратной совместимости, за несколькими исключениями. Перечислим некоторые возможности, предоставляемые рекомендацией CSS2:

- Блочная верстка. Относительное, абсолютное и фиксированное позиционирование дает возможность управлять размещением элементов на странице без табличной верстки.
- Типы носителей. Эта возможность позволяет устанавливать разные стили для разных носителей (например, монитор, принтер, проектор). Страничные носители, например, позволяют установить разные стили для элементов на четных и нечетных страницах при печати.
- Звуковые таблицы стилей. Эта возможность определяет параметры голоса: громкость, тембр и т. п. — для звуковых носителей, что может быть полезно для слепых посетителей сайта.
- Расширенный механизм селекторов. Эта возможность позволяет выбирать элементы на странице, к которым будет применено то или иное свойство CSS.

В настоящее время W3C больше не поддерживает CSS2 и рекомендует использовать CSS2.1, которая была принята в июне 2011 г.

Примерно в то же время, когда была опубликована CSS2, Консорциум W3C начал разработку стандарта CSS3 и продолжает

вести ее до настоящего времени. CSS3 основан на CSS2.1, дополняет существующие свойства и значения и добавляет новые. В отличие от предыдущих версий спецификация разбита на модули, разработка и развитие которых идет независимо: так, в сентябре 2011 г. был принят модуль Selectors Level 3, а в июне 2016 г. — Media Queries.

С 2011 г. Консорциум W3C ведет разработку CSS4. Модули CSS4 построены на основе CSS3 и дополняют их новыми свойствами и значениями. Все они существуют пока в виде черновиков (англ. working draft).

2.5. Синтаксис CSS

Таблица стилей состоит из набора операторов. При этом каждый оператор является либо директивой, либо правилом. Операторы могут разделяться пробельными символами.

Директива (англ. at-rule) начинается с символа «@» и соответствующего ключевого слова. Директивы CSS2 приведены в табл. 4. Правило состоит из селектора и блока деклараций, заключенного в фигурные скобки.

Приведем примеры использования директив и правил:

```
@import "mystyle.css";
h1 {
    font-size: 120%; /*относительный размер*/
    font-family: Verdana;
    color: #333366;
}
```

Таблица 4

Директивы языка CSS

@charset	Задает кодировку символов
@font-face	Задает описания шрифтов
@import	Включает другую таблицу стилей в текущую
@media	Задает имена устройств отображения
@page	Задает свойства страницы для печати

Каждая декларация, как видно из данного примера, состоит из названия свойства (например, font-family), символа «двоеточие» (:) и значения свойства (например, Verdana). Декларации должны разделяться точкой с запятой (;).

Таблицы стилей могут содержать комментарии.

Директивы @import должны располагаться в таблице стилей перед первым правилом и не могут находиться внутри блока; в противном случае они игнорируются обозревателем.

```
<head>
<title>Импортирование</title>
<style>
    @import url(mystyle.css)
    /*далее следуют правила… */
</style>
</head>
```

Директива @import может содержать список названий устройств отображения, к которым должна применяться данная таблица стилей, названия разделяются запятыми, например:

```
@import url("fineprint.css") print;
@import url("bluish.css") projection, tv;
```

Если списка названий устройств нет, то предполагается, что он равен all, т. е. импортируемая таблица стилей применима ко всем устройствам.

Директива @media предназначена для создания таблиц стилей, зависящих от устройства отображения. Она задает список устройств отображения, к которым применимы содержащиеся в ней правила, заключенные в фигурные скобки; элементы списка разделяются запятыми. Если текущего устройства отображения нет в списке, то содержимое данной директивы должно игнорироваться обозревателем.

```
@media print, screen {
```

```
    body {font-size: 10pt}
}
```

Читатели, должно быть, заметили, что разные свойства имеют разные типы значений. Так, мы задаем размеры блоков и шрифтов в пикселях или процентах, определяем цвета с помощью зарезервированных названий (например, red или blue), используем строковые значения для названий шрифтов (например, Verdana или Arial).

Значения свойств CSS могут быть следующих типов:

- целые и действительные числа,
- размеры,
- строки,
- URL-адреса,
- цвета,
- процентные значения,
- угловые величины,
- счетчики,
- времена,
- частоты.

Больше всего нам придется работать с размерами и процентными значениями. И размеры, и проценты основаны на числовом типе данных. Числовые значения могут записываться только в десятичной нотации. Целое число состоит из одной или более десятичных цифр (0–9). Действительное число либо является целым, либо состоит из нуля или более десятичных цифр, за которыми следует точка и одна или более десятичных цифр. И перед целыми, и перед действительными числами может стоять знак плюс (+) или минус (–), например, –21, 1.234567, -.999.

Указываются вертикальный или горизонтальный размеры чего-либо. Размер задается как число, за которым следует единица измерения. Если размер равен 0, то единицу измерения можно не указывать.

Единицы измерения подразделяются на абсолютные и относительные:

- абсолютные единицы измерения задают точный физический размер;

- относительные единицы измерения указывают размер относительно другого размера.

Согласно первоначальной редакции CSS2.1, пиксель считался относительной единицей. Это объяснялось тем, что пиксель — это точка дисплея, и ее размер зависит как от физических размеров экрана, так и от его разрешения: пиксель на экране с разрешением 640 × 480 будет больше, чем на экране с разрешением 1280 × 1024. Большое разнообразие современных аппаратных платформ (телефоны, настольные компьютеры и т. д.) привело к немалым сложностям с определением пикселей, и Консорциум W3C в обновленной редакции CSS2.1 определил пиксель как абсолютную единицу измерения.

Следует различать пиксели CSS и аппаратные пиксели. Задавая значения свойств, мы всегда работаем с пикселями CSS.

2.6. Иерархия элементов в html-документе

Для дальнейшего изучения материала потребуется ввести понятие дерева документа и соответствующую терминологию. Рассмотрим пример:

```
<html>
<head>
    <title>Дерево документа</title>
</head>
<body>
    <h1>Заголовок</h1>
    <p>Текст абзаца</p>
</body>
</html>
```

Корнем этого дерева является элемент html, который имеет двух сыновей: head и body. Элемент head является отцом элемента title, а элемент body — отцом элементов h1 и p (два последних элемента называются братьями, причем h1 является старшим братом,

а p — младшим). Все элементы дерева являются потомками корня, а тот является их предком. Таким образом, можно представить в виде дерева любой документ, к которому применимы правила языка CSS.

2.7. Селекторы CSS

Ранее мы выяснили, что правило в CSS состоит из селектора и блока деклараций, заключенного в фигурные скобки.

Декларации определяют свойства отображения, а селектор указывает, к каким именно элементам документа именно эти декларации должны применяться (такие элементы называются субъектами этого селектора). Приведем формальное определение селекторов, параллельно дадим их классификацию.

Селектор типа. Совпадает с именем элемента в документе и указывает, что его субъектами являются все элементы документа с данным именем. Все примеры, которые мы приводили ранее в этой главе, содержали селекторы типа.

Универсальный селектор. Символ звездочка (*), означающий, что субъектами селектора являются все элементы документа. Если звездочка не единственная составляющая селектора, то она может опускаться.

Базовый селектор. Универсальный селектор, или селектор типа.

Простой селектор. Базовый селектор, за которым следуют нуль или более селекторов атрибутов, селекторов идентификаторов или псевдоклассов (в любом порядке).

Составной селектор. Образуется из простых селекторов соединением их с помощью специальных символов.

Как мы видим, основу этих определений составляют базовые селекторы. Поясним их определения примерами:

```
h1 {font-size: 120%;}
* {font-size: 120%;}
h1, h2, h3 {font-size: 120%;}
```

Как видно из последнего примера, если несколько селекторов имеют одинаковые декларации, то их можно сгруппировать, т. е. объединить в одно правило, перечислив селекторы через запятую.

Селекторы классов

В большом документе разные абзацы могут выполнять разные функции: одни содержат основное повествование, другие — выключные цитаты, третьи — пояснения или предупреждения. Соответственно, эти абзацы должны быть оформлены по-разному. Обойтись селектором типа здесь не удастся, так как во всех случаях мы работаем с элементами одного типа — абзацами; возникает задача как-то эти абзацы различать.

Для решения этой задачи в языке HTML определен атрибут элементов class, который указывает, что элемент является членом определенного класса. В свою очередь, CSS поддерживает селекторы классов, которые основаны на использовании атрибута class и имеют вид element.class.

Рассмотрим пример:

```
<style>
    p.note {font-size: 80%;}
    p.cite {text-align: center;}
</style>

<p class="note">Для HTML документов CSS поддерживает селекторы классов, которые основаны на использовании атрибута class и имеют вид element.class</p>

<p class="cite">If an element has multiple class attributes, their values must be concatenated with spaces between the values before searching for the class</p>

<p>А это текст… Просто текст</p>
```

Здесь таблица стилей содержит два правила. Простой селектор p.note состоит из базового (а точнее, селектора типа) и следующего за ним селектора класса .note. Первый абзац, имеющий класс note, будет отображаться уменьшенным шрифтом, а второй, имеющий

класс cite, — центрироваться. Последний абзац не получит никакого специального оформления.

Селектор класса может не содержать названия элемента, например,

```
.info {font-style: italic}
```

Такое правило относится ко всем элементам, имеющим атрибут class="info". Селектор класса, не содержащий имени элемента, следует понимать как селектор *.class, в котором универсальный селектор * опущен.

Html-элемент может принадлежать к нескольким классам. В этом случае значением атрибута является список имен классов, разделенных пробелами. В свою очередь, CSS позволяет задавать подмножества значений атрибута class. В следующем примере правило применяется к первому абзацу, но не применяется ко второму.

```
p.warning.red {font-style: italic}

<p class="warning red blue"> Применимо </p>
<p class="warning blue"> Неприменимо </p>
```

Селекторы идентификаторов

Документы могут содержать элементы с атрибутами id, задающими уникальные идентификаторы этих элементов. Соответствующий селектор идентификатора в CSS имеет вид element#id или просто #id.

Рассмотрим пример. Здесь указано, что элемент с идентификатором note должен иметь меньший размер шрифта (по сравнению с базовым). Это правило будет применено для абзаца с данным значением атрибута id.

```
<style>
    #note {font-size: 80%;}
</style>

<p id="note">Абзац с мелким текстом</p>
```

Селекторы атрибутов

Многие теги различаются по своему действию в зависимости от того, какие в них используются атрибуты. Например, тег <input> может создавать кнопку, текстовое поле и другие элементы управления за счет изменения значения атрибута type. Если добавить стилевые правила к селектору input, то стиль будет применен одновременно ко всем элементам, созданным с помощью тега input.

Чтобы гибко управлять стилем подобных элементов, в CSS введены селекторы атрибутов. Они позволяют установить стиль по присутствию определенного атрибута тега или его значения. Селекторы атрибутов обладают мощным и гибким синтаксисом. Поясним синтаксис на примерах:

```
<style>
h1[title] {color: blue;}
p[text-align="right"] {color: blue;}
a[href^="http"]: after {content: attr(href)}
a[href$=".pdf"]: after{content: url(pdf-con.png)}
a[href*="w3c"] {text-decoration: none;}
p[class~="info"] {color: red;}
*[lang|="en"] {color: blue;}
</style>
```

Селектор h1[title] относится к заголовкам, имеющим атрибут title.

Селектор p[text-align="right"] относится к абзацам, чей атрибут text-align имеет значение right.

Селектор a[href^="http"] относится к гиперссылкам, чей атрибут href начинается с подстроки «http». Как правило, у внешних ссылок URL-адрес начинается со схемы http, а у внутренних — с пути к ресурсу сервера, следовательно, таким образом можно по-разному оформлять внешние по отношению к сайту и внутренние гиперссылки.

Селектор a[href$=".pdf"] относится к гиперссылкам, чей атрибут href заканчивается подстрокой «.pdf».

Возможны ситуации, когда стиль следует применить к тегу с определенным атрибутом, при этом частью значения этого атрибута является некоторый текст, однако точно не известно, в каком месте значения включен данный текст — в начале, середине или конце. В подобном случае следует использовать такой синтаксис: [attr*="value"].

Селектор [attr~=value] относится ко всем элементам, чей атрибут attr состоит из списка значений, разделенных пробелами, и одно из этих значений равно value.

[attr|=value] применяется ко всем элементам, чей атрибут attr имеет значение, состоящее из нескольких слов, разделенных дефисом, причем первое из этих слов равно value (первоначально предназначалось для выделения кода основного языка из полного кода языка).

Рассмотрим пример:

```
<style>
    a[target="_blank"]{
    background: url(ico.png) 0 6px no-repeat;
    padding-left: 15px;}
</style>

<a href="1.html">Обычная ссылка</a>
<a href="2.html" target="_blank">Ссылка в новом окне</a>
```

Здесь стиль ссылки изменяется в том случае, когда тег <a> содержит атрибут target со значением _blank. При этом ссылка будет открываться в новом окне, и чтобы показать это, с помощью стилей добавляется небольшой рисунок перед текстом ссылки.

Составные селекторы

Перейдем теперь к рассмотрению составных селекторов, которые образуются из простых селекторов путем их соединения в новый селектор. Важнейшими из составных селекторов являются селекторы потомков, т. е. селекторы тех элементов, которые являются потомками другого элемента в дереве документа. В следующем

примере определен стиль отображения всех элементов span, которые находятся внутри элементов p.

```
p span {color: blue;}
```

CSS также поддерживает использование селекторов детей, т. е. селекторов тех элементов, которые являются детьми другого элемента в дереве документа. Селекторы детей образуются путем соединения нескольких селекторов символом «>». Следующее правило будет применяться ко всем элементам p, которые являются детьми элемента body (иными словами, оно применимо к тем и только тем элементам p, которые находятся внутри элемента body, и никакого промежуточного элемента между body и p нет).

```
body > p {text-indent: 3em;}
```

CSS поддерживает использование селекторов соседей, т. е. селекторов тех элементов, которые являются братьями в дереве документа и расположены в документе непосредственно друг за другом. Селекторы соседей образуются путем соединения нескольких селекторов символом «+». Следующее правило уменьшает расстояние между элементами h1 и h2, если h2 непосредственно следует в документе за h1.

```
h1 + h2 {margin-top: –5mm;}
```

Селекторы могут комбинироваться в весьма длинные цепочки, но злоупотреблять этим не стоит, так как это усложняет стили.

```
div.wrap ol>li p {line-height: 1.5;}
```

Псевдоклассы

Наряду с селекторами классов существуют селекторы псевдоклассов. Среди них есть связанные с динамическим состоянием объекта: active, link, focus, hover, visited. Они применяются, прежде всего, к гиперссылкам.

Следующая группа — псевдоклассы, имеющие отношение к дереву документа. Например, :first-child применяется к первому дочернему элементу селектора, который расположен в дереве элементов документа.

Рассмотрим пример:

```
<style>
    div > p: first-child {text-indent: 3em;}
</style>

<div>
<p>Этот абзац является первым ребенком блока. Он начинается с красной
строки.</p>
<p>Второй абзац с красной строки не начинается.</p>
</div>
```

2.8. Наследование в CSS

Некоторые свойства наследуются детьми элемента в дереве документа.

Рассмотрим пример:

```
<style>
    h1 {color: red;}
</style>
<h1>
    Этот заголовок <em>очень важен</em>!
</h1>
```

Здесь для элемента em не определены стили; элемент em унаследует цвет своего отца, т. е. элемента h1.

По этой причине для задания стиля отображения элементов по умолчанию достаточно задать стиль элемента html или body. Все остальные элементы являются потомками этих элементов, поэтому

они будут наследовать их свойства. При этом важно помнить, что значения, заданные в виде процентных величин, не наследуются.

Многие свойства допускают значение inherit, из этого следует, что в качестве значения свойства должно использоваться вычисленное значение данного свойства отцовского элемента.

Наследование — тема отнюдь не простая; некоторые свойства являются наследуемыми, некоторые — нет. Для проверки своих догадок нужно обратиться к справочнику.

Разберем более сложный пример:

```
<style>
    table {color: red;
        background: #333;
        border: 2px solid red;}
</style>
    <table cellpadding="4" cellspacing="0">
        <tr>
            <td>Ячейка 1</td>
            <td>Ячейка 2</td>
        </tr>
        <tr>
            <td>Ячейка 3</td>
            <td>Ячейка 4</td>
        </tr>
    </table>
```

Особенностью таблиц можно считать строгую иерархическую структуру тегов. Вначале следует контейнер <table>, внутри которого добавляются теги <tr>, а затем теги <td>. Если в стилях для селектора table задать цвет текста, то он автоматически устанавливается для содержимого ячеек.

В данном примере для всей таблицы установлен красный цвет текста, поэтому в ячейках он также применяется, поскольку тег <td> наследует свойства тега <table>. При этом следует понимать, что не все стилевые свойства наследуются. Так, border задает рамку вокруг таблицы в целом, но никак не вокруг ячеек. Аналогично не

наследуется значение свойства background. Возникает вопрос: а почему же тогда цвет фона у ячеек в данном примере темный, если он не наследуется? Дело в том, что у свойства background в качестве значения по умолчанию выступает transparent, т. е. прозрачность. Таким образом цвет фона родительского элемента «проглядывает» сквозь дочерний элемент.

2.9. Каскадирование

Таблицы стилей могут иметь три источника происхождения: автор, пользователь и обозреватель.

Каскадность языка CSS состоит в том, что каждому правилу приписан определенный вес; если к конкретному элементу применимы несколько правил, то используется то, которое имеет наибольший вес. В результате происходит кумулятивное накопление свойств элементов в соответствии с правилами наследования и тем самым образуется каскад свойств, распространяющийся от предков к потомкам.

По умолчанию вес правил таблицы автора больше, чем вес правил пользователя (за исключением правил с атрибутом !important, для которых это соотношение является обратным), а вес правил пользователя больше, чем вес правил обозревателя. Общий порядок определения правила и свойства, которые будут применены к элементу, таков:

1. Выбираются все декларации, которые соответствуют данному устройству отображения; из них выбираются все правила, чьи селекторы соответствуют данному элементу.

2. Декларации сортируются по весу их источника происхождения, как описано выше.

3. Производится вторичная сортировка по специфичности селектора: более специфичные селекторы сильнее, чем более общие.

4. И, наконец, последняя сортировка: если два правила имеют одинаковые вес и специфичность, то применяется последнее из них. При этом правила импортированных таблиц располагаются до всех правил импортирующей таблицы.

Для того чтобы правила пользовательской таблицы стилей могли перекрывать авторскую, CSS содержит атрибут !important. Правило пользовательской таблицы стилей с таким атрибутом имеет больший вес, чем соответствующее правило авторской таблицы стилей.

Разберем пример:

```
/* из таблицы стилей пользователя */
p {text-indent: 1em!important}
p {font-style: italic!important}
p {font-size: 18pt}

/* из таблицы стилей автора */
p {text-indent: 1.5em!important}
p {font: 12pt sans-serif!important}
p {font-size: 24pt}
```

Первое правило таблицы стилей пользователя содержит атрибут !important, поэтому оно весомей, чем первое правило таблицы стилей автора. Второе правило пользователя также более весомо по той же самой причине. Однако третье правило пользователя менее весомо, чем второе правило автора. Точно также третье правило автора менее весомо, чем его второе правило.

Для справки отметим, что свойство font является составным:

```
font: font-style | font-size | font-family и другие
```

2.10. Специфичность

Для элемента может подходить несколько свойств. Какое выбрать? Стандарт гласит, что выбираются наиболее специфичные. Специфичность селектора вычисляется следующим образом:

1) подсчитать количество атрибутов id в селекторе;

2) подсчитать количество атрибутов class в селекторе;

3) подсчитать количество названий тегов html в селекторе (все псевдоэлементы игнорируются).

Теперь запишем эти три числа подряд, чтобы получить число из трех цифр (нам, возможно, придется привести числа *a*, *b* и *c* к системе счисления с большим основанием, чтобы каждое из них выражалось одной цифрой). Результатом и будет специфичность селектора (чем она выше, тем селектор специфичней). Приведем список примеров селекторов, отсортированных по их специфичности:

```
#info          {…} /* a=1 b=0 c=0 */
p ul li.red    {…} /* a=0 b=1 c=3*/
li.red         {…} /* a=0 b=1 c=1*/
li             {…} /* a=0 b=0 c=1*/
```

В первом примере специфичность равна 100, во втором — 13, в третьем — 11, в последнем — 1.

2.11. Вычисление значения свойств

После того как обозреватель проанализировал документ и построил дерево документа, для каждого элемента дерева вычисляется значение каждого из его свойств, применимых к текущему устройству отображения. Окончательное значение свойства вычисляется в три этапа. Сначала значение определяется из спецификации (специфицированное значение), затем при необходимости преобразуется к абсолютному значению (вычисленное значение) и, наконец, преобразуется с учетом ограничений контекста (фактическое значение). Поясним каждый из этих этапов подробнее.

Специфицированное значение определяется с помощью следующих механизмов, перечисленных в порядке предпочтения.

1. Если каскад возвращает значение, то используется это значение.

2. В противном случае, если свойство является наследуемым, используется соответствующее значение отцовского элемента.

3. В противном случае используется начальное значение свойства (оно указано ниже в определении каждого из свойств).

Специфицированные значения могут быть как абсолютными (например, red или 2 mm), так и относительными (например, auto, или 2 em, или 10 %). Для абсолютных значений вычисленное значение совпадает со специфицированным. С другой стороны, относительные значения должны быть преобразованы к абсолютным. Так, процентные величины преобразуются в числа путем умножения на соответствующее значение; размеры, заданные в em или пикселях, умножаются на размер шрифта или пикселя; значение auto заменяется на величину, вычисляемую по формуле, указанной в определении соответствующего свойства, и т. п.

Наконец, обозреватель проверяет, допустимо ли вычисленное значение в контексте данного свойства, и если нет, соответственно преобразует его. Например, размер в пикселях может быть только целым, поэтому потребуется округлить полученное действительное число до целого. Результатом таких преобразований и является фактическое значение свойства, используемое при отображении элемента.

Контрольные вопросы

1. CSS имеет разные типы селекторов, эффективность обработки которых существенно отличается, а значит, способ записи селекторов влияет на скорость отрисовки веб-страниц. Для разработки высокопроизводительных проектов необходимо знать, как обрабатываются селекторы. Как браузеры «читают» CSS правила типа

```
div.info #first li>a: hover?
```

1) справа налево;

2) слева направо;

3) сначала обрабатываются селекторы элементов, потом классов, потом идентификаторов, потом все остальные;

4) варианты 1–3 неверные, написать свой вариант.

2. Какой селектор более эффективно будет обработан браузером?

1) .menu#firstItem {…}
2) li#firstItem {…}
3) #firstItem {…}

Глава 3
БЛОЧНАЯ ВЕРСТКА: БЛОЧНЫЕ И СТРОЧНЫЕ ЭЛЕМЕНТЫ, ПОЗИЦИОНИРОВАНИЕ

Когда браузер просматривает исходный текст html-документа, он создает объектное дерево документа, где html-элементам соответствуют объекты. Кроме того, браузер анализирует CSS-файл, после чего стилевые свойства объектов в объектном дереве документа получают свои значения. Среди значений есть весьма понятные, такие как цвет или тип шрифта. А есть набор свойств, которые определяют размеры элемента, его положение на странице и взаиморасположение по отношению к другим элементам. Почему мы говорим именно «набор свойств»? Дело в том, что изучать эти свойства (отвечающие за визуализацию и раскладку элементов) невозможно по отдельности друг от друга, они очень тесно связаны, влияют друг на друга. Например, значение высоты родительского блока зависит от способа позиционирования дочерних элементов.

С одной стороны, многообразие свойств и их значений дает очень мощный инструмент в руки верстальщика — можно реализовать в точности тот дизайн, который был задуман. С другой стороны, изучение этой темы похоже на решение большой системы нелинейных уравнений. Чтобы разобраться с первым уравнением, необходимо решить второе и десятое, а чтобы упростить десятое, надо решить первое… Вот и получается замкнутый круг.

Как мы подойдем к этой задаче? Мы будем изучать свойства и их значения итерационно. Начнем 1) с изучения структуры обрамляющего прямоугольника в самом простом случае на примере

одного-двух абзацев, когда нет никаких побочных влияний других свойств. Потом 2) введем понятие блочных и строчных элементов, опишем их основные характеристики. Далее 3) сообщим самые основные сведения о схемах позиционирования и потоке верстки. После этого возвратимся к блочным и строчным элементам и сделаем их подробнейший разбор. Потом подробно разберем схемы позиционирования и формально определим поток верстки. При этом в деталях мы не завязнем: перечни значений изучаемых свойств не будут исчерпывающими, мы ограничимся лишь самым необходимым.

Далее 4) подробно изучим плавающие элементы, при этом каждый раз будем возвращаться к предыдущим разделам и уточнять содержащиеся в них сведения.

В итоге должно сформироваться понимание блочной модели CSS, и можно переходить к самостоятельному изучению документации, содержащей исчерпывающее описание правил визуализации и раскладки элементов на визуальном носителе. Из учебников, пожалуй, лишь в книге Эрика Майера «CSS — каскадные таблицы стилей» эта тема изложена удовлетворительно. Точные сведения содержатся только в документации Cascading Style Sheets Level 2 Revision 1 (CSS2.1) Specification, W3C Recommendation.

3.1. Объемлющий прямоугольник

Каждый элемент порождает объемлющий прямоугольник, строение которого показано на рис. 5.

Объемлющий прямоугольник состоит из области содержимого (content), из необязательных обрамляющих полей (англ. padding), границы (англ. border) и отступов (англ. margin). В свою очередь, поля, границы и отступы распадаются на четыре части: левую, правую, верхнюю и нижнюю (англ. left, right, top и bottom соответственно). На рисунке эти части обозначены соответствующими сокращениями: «LM» — левый отступ (left margin), «TB» — верхняя граница (top border) и т. п.

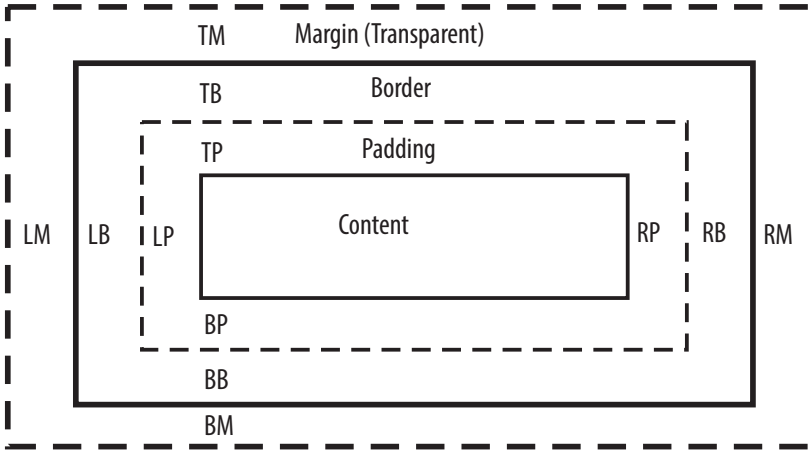


Рис. 5. Структура объемлющего прямоугольника

Периметр каждой из четырех частей прямоугольника называется ее краем, так что каждый объемлющий прямоугольник содержит четыре края.

- Край содержимого, или внутренний край, ограничивает отображаемое содержимое элемента.
- Край полей ограничивает поля прямоугольника. Если ширина полей равна 0, то край полей совпадает с внутренним краем элемента. Прямоугольник, ограниченный краем полей, иначе называется *вмещающим блоком*[3] элемента.

Аналогично определяются край границы и край отступа (он же внешний край).

Обратим внимание на то, что, говоря о краях полей, не надо уточнять, какой это край — «внешний» или «внутренний». То, что можно было бы назвать «внутренним краем полей», уже имеет свое название — «край содержимого»; то же для границ и отступов.

Размеры области содержимого в прямоугольнике, или ширина и высота содержимого, зависят от нескольких факторов: что именно является содержимым элемента (текст, таблица или другие элемен-

[3] О вмещающих блоках мы будем говорить при изучении позиционирования.

ты), заданы ли свойства элементов width и height и т. д. Подробно эти вопросы обсуждаются далее в этой главе.

Проиллюстрируем сказанное на примере:

```
<! DOCTYPE html>
<html>
    <head>
        <title>position absolute</title>
        <style>
            div{
                margin: 40px;
                border: solid blue 2px;
                padding: 10px;
                background: yellow;
            }
            p{
                margin: 20px;
                border: solid green 2px;
                padding: 15px;
                background: #ffccaa;
            }
        </style>
    </head>
    <body>
        <div>
            <p> Текст 1-го внутреннего абзаца.</p>
            <!--p> Текст 2-го внутреннего абзаца.</p-->
        </div>
    </body>
</html>
```

В примере показано два вложенных блочных элемента — блок div и абзац p. Для того чтобы было проще видеть края областей, мы используем свойство background для задания цвета фона (рис. 6).

Если открыть средства разработчика (кнопка F12 в браузере), мы увидим, где находятся отступы, далее — граница, поля, область

контента для div'a. Разберем, чем заполнена область контента div'a. Она заполнена абзацем p, который, в свою очередь, имеет отступы, границу, поля и контент. Обратим внимание на то, что желтый «фон» разбивается на две части: поля внешнего блока и отступы внутреннего абзаца, отступы всегда прозрачные. То есть сквозь прозрачные отступы внутреннего абзаца просвечивает желтый фон области контента родительского div'a. Неверно считать, что отступы абзаца унаследовали желтый цвет, отступы — прозрачные.

Высветим в средствах разработчика отступы внешнего div'a. Видны белые полосы по бокам div'a, т. е. отступы div'a не упираются в границы окна браузера (рис. 7). Возникает вопрос: откуда эти боковые полосы берутся? Касалось бы, div самый внешний, и его отступы должны отсчитываться от границ окна браузера. На самом деле он тоже вложен — в body. Белые полоски — это отступы body; чтобы убрать их, задаем стилевые свойства:

```
body {margin: 0;}
```

Следующий вопрос: а почему в средствах разработчика только с боков видны белые полоски — отступы body? Почему сверху

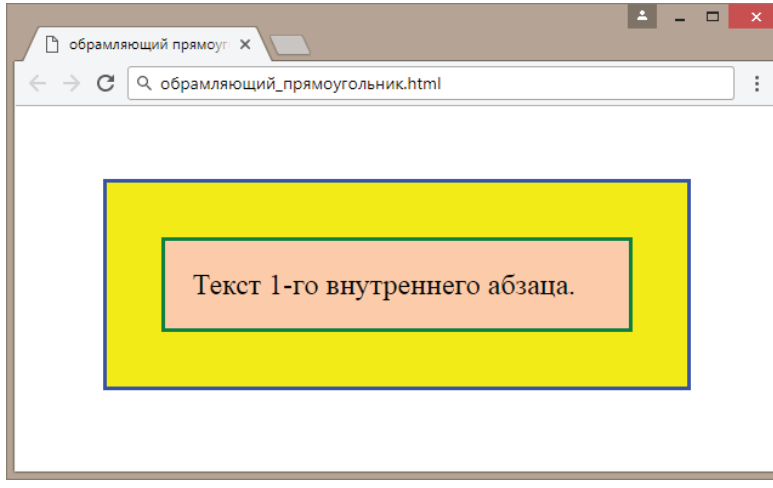


Рис. 6. Структура объемлющего прямоугольника

не было белой полосы (см. рис. 6) между границей окна браузера и отступом div'а? Ниже мы дадим ответ.

Вставим в div еще один абзац, для этого в рассматриваемом примере достаточно снять комментарий. Обратим внимание на то, что отступ между первым и вторым абзацами равен не 40 px, а 20 px, т. е. отступ равен максимуму из них, а не их сумме. Этот эффект называется слияние (вертикальных) отступов. Слияние вертикальных отступов полезно для экономии места на странице. Уберем второй абзац.

Слияние отступов возможно не только между соседними («братскими») элементами, но и между отцовским и дочерним элементом. Если между верхним отступом родительского элемента и верхним отступом дочернего элемента нет границ и полей, то произойдет слияние верхних границ.

Уберем границы и поля у div'а, посмотрим на результаты в средствах разработчика. Теперь из 40 px, занимаемых отступами div'а, нижние 20 также занимает отступ абзаца. Ранее абзац был окружен желтым «ореолом» с четырех сторон. Теперь желтые полоски только по бокам абзаца. Откуда они? Горизонтальные отступы никогда не сливаются — сквозь боковые отступы абзаца просвечивает цвет фона, вмещающего div'а.

Вернемся к вопросу «А почему только с боков видны белые полоски?». Теперь понятно, что имел место эффект слияния отсту-

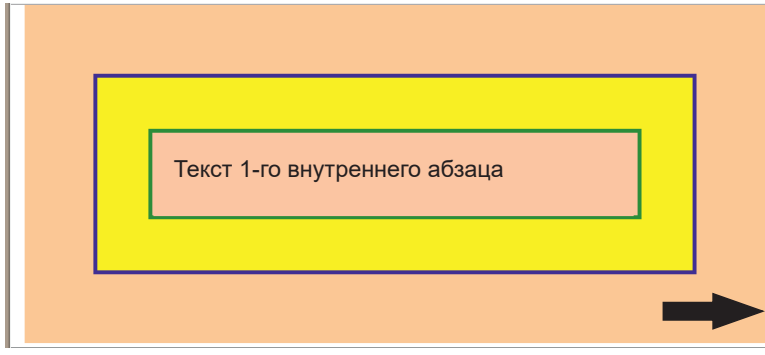


Рис. 7. В средствах разработчика подсвечены розовые отступы div'а, видны белые горизонтальные отступы у body

пов у body и div'a. По умолчанию отступы body составляют 8 px. Таким образом, из 40 px, занимаемых отступами div'a, нижние 8 приходились на отступы body. Можно образно сказать, что отступы div'a поглотили отступы body. Если определить размеры отступов body так:

```
body {margin: 50;},
```

то вертикальные отступы body будут видны в средствах разработчика.

Заметим, что отступы могут быть отрицательными. Например, отрицательный верхний отступ подтягивает элемент вверх.

Обратим внимание на то, что текста в абзаце мало, но абзац все равно занимает всю ширину родительского div'a. Почему абзацы так себя ведут, обсудим в следующем параграфе.

3.2. Блочные и строчные элементы

Когда мы просматриваем веб-страницу, нам кажется естественным, что заголовки и абзацы начинаются с новой строки, отделены от предыдущего контента пустым вертикальным отступом, а гиперссылки встраиваются в текст, новую строку не начинают.

Абзацы так же, как и заголовки и блоки div, визуализируясь, располагаются на новой строке и занимают всю доступную ширину, т. е. ширину родительского блока. В предыдущем примере это ширина body для div'a и ширина div'a для абзаца. Последовательные блочные элементы располагаются один под другим.

Строчные элементы, такие как span, a, b, не начинают новую строку, располагаются последовательно один за другим в строке, и лишь если в строке не хватает места, то «хвост» переносится на новую строку.

Большинство элементов HTML, используемых в теле документа, подразделяются на блочные (англ. block-level) и строчные (англ. inline-level) элементы. А к какому типу относятся таблицы? А картинки?

Сразу стоит сказать, что существуют элементы, не относящиеся ни к блочным, ни к строчным, например, ячейка или строка таблицы.

То, как обрабатывается элемент (как блочный или строчный), определяется его свойством display. Таблица стилей по умолчанию, которая есть у браузера, задает для элементов p, div, h1 и им подобных значение block у свойства display, а для span, a, em и т. д. — inline.

Сразу отметим, что различных значений у свойства display более десятка; есть, например, еще строчно-блочные элементы. Совсем иначе обрабатываются строки и ячейки таблиц, соответствующие значения свойства display — table-row и table-cell.

Свойство display не является read-only. Так, например, для элемента li можно определить display: inline и все элементы списка вытянутся в строку. Так иногда делают для превращения вертикальных меню в горизонтальные.

3.3. Три схемы позиционирования

В HTML формирование элементов на странице происходит сверху вниз согласно схеме документа. Элементы, размещенные в самом верху кода, отобразятся раньше тех, которые расположены в коде ниже. Такая логика позволяет легко прогнозировать результат вывода элементов и управлять им. Потоком называется порядок вывода объектов на странице. При этом существует несколько возможностей «вырвать» элемент из потока и позиционировать его в соответствии с потребностями дизайнера. Поскольку элемент не существует в потоке, то в html-коде его можно описать где угодно, а также вывести в заданное место окна.

Согласно стандарту CSS2.1, элементы могут раскладываться в соответствии с тремя схемами позиционирования:

1) нормальный поток, который включает в себя блочный и строчный контекст форматирования, а также относительное позиционирование;

2) плавающие элементы;

3) абсолютное позиционирование, которое включает в себя абсолютное позиционирование в классическом понимании и фиксированное позиционирование.

3.4. Блочные и строчные элементы и боксы

Как уже было сказано, элементы делятся на блочные, строчные и иные (про иные будем лишь упоминать по ходу изложения, не отвлекаясь на детали). Терминологически более точно называть их block-level elements и inline-level elements.

Блочные элементы (англ. block-level elements) — это те элементы исходного кода, которые визуализируются как блоки (типа абзацев или заголовков). Следующие значения свойства display превращают элемент в блочный: block, table, list-item. А что значит «визуализируется как блок»? Фактически это означает, что создается блочный объемлющий прямоугольник (англ. block-level box), который мы будем называть *блочным боксом*.

Итак, последовательность следующая. Есть элемент исходного кода, как то заголовок h1 или div, чье свойство display имеет значение block. Или в исходном коде встретилась таблица, чье свойство display имеет значение table. Такие элементы является блочными элементами, соответственно, визуализируются как блоки, т. е. порождают блочные боксы, которые принимают участие в блочном контексте форматирования. По сути, блочный контекст форматирования как раз и означает, что объемлющие прямоугольники раскладываются друг под другом по вертикали, а не вбок.

Каждый блочный элемент порождает *главный блочный бокс* (англ. principal block-level box). Возникает вопрос: «Раз есть главный, значит, есть и не главный?» Да, и такие могут быть, например, элементы списка li могут порождать такие «дополнительные» блоки для маркеров списка, но разбирать их мы не будем. Достаточно знать, что главный блочный бокс содержит боксы дочерних элементов и является тем боксом, который вовлечен во все схемы позиционирования. Например, при относительном позиционировании блока

div смещается главный блочный бокс этого div'а вместе с соответствующим контентом.

Строчные элементы (англ. inline-level elements) — это те элементы исходного кода, которые, визуализируясь, не образуют блоки, контент распределяется построчно. Следующие значения свойства display превращают элемент в строчный: inline, inline-table, inline-block.

Строчные элементы порождают строчные объемлющие прямоугольники (англ. inline-level box), которые мы будем называть *строчными боксами*; это такие боксы, которые принимают участие в строчном контексте форматирования.

Что же такое «контекст форматирования», который мы упомянули выше? Это способ раскладки боксов и контента. Бывает строчный и блочный контент (другого нет).

В блочном контексте форматирования боксы раскладываются один за другим вертикально, начиная с верха вмещающего блока. Вертикальные расстояния между блоками определяются свойством margin, при этом может иметь место эффект слияния отступов.

При строчном контексте форматирования боксы раскладываются один за другим горизонтально, начиная с верха вмещающего блока. Горизонтальные отступы, границы и поля принимаются во внимание. Когда строчному боксу не хватает места в строке, он или целиком переносятся на новую строку, или разрезается на части (например, по границам слов) и невместившаяся часть переносится. Здесь можно было бы ввести понятие лайн-бокса[4] (англ. line box) и показать, как строчные боксы заполняют лайн-боксы, но это понятие становится важным лишь при изучении плавающих элементов (англ. float). Пока того, что сказано о строчном контексте форматирования, достаточно.

Приведем пример, показывающий, как строчный бокс разрезается на части.

---

[4] Лайн-бокс (англ. line box) — это прямоугольная область, содержащая боксы, которые формируют одну строку. Отметим, что line box и inline-level box — это не синонимы, это разные, по сути, вещи. Лайн-бокс может содержать несколько строчных боксов, и один строчный бокс может не вместиться в лайн-бокс, разорваться и занять два или несколько лайн-боксов.

```
<! DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN">
<html>
    <head>
        <title>Inline flow on several lines</title>
        <style>
        em{  padding: 8px;
            margin: 1em;
            border-width: medium;
            border-style: dashed;
            line-height: 2.4em;}
        </style>
    </head>
    <body>
        <p>Several <em>emphasized words</em> appear here</p>
    </body>
</html>
```

Если развернуть окно браузера на весь экран, то тексту хватит места расположиться в одну строку. Словосочетание «emphasized words» будет обведено пунктирной рамкой, между рамкой и текстом будут восьмипиксельные поля. Если уменьшить ширину окна, как показано на рис. 8, то строчный бокс, соответствующий строчному элементу em, будет разрезан на две части — первая часть помещается в первом лайн-боксе, а вторая часть переносится во второй лайн-бокс. Обратим внимание, что в месте разрыва строчного бокса не отображается граница и не учитываются поля.

Зададимся теперь вопросом «Зачем нужны разнообразные значения свойства display?». Действительно, если и таблицы, чье свойство display имело значение table, и абзацы, чье свойство display имело значение block, порождают блочные боксы и визуализируются как блоки, то зачем это разнообразие нужно? Ответ такой: свойство display определяет не только способ визуализации элементов, но и способ организации их контента. Это две разные характеристики:

- первая характеристика — способ визуализации бокса; здесь определяется, как раскладываются боксы — по вертикали (как div, p или h1) или по горизонтали (как span, em или a);

- вторая — способ организации контента; здесь определяется способ размещения дочерних элементов внутри данного блочного бокса.

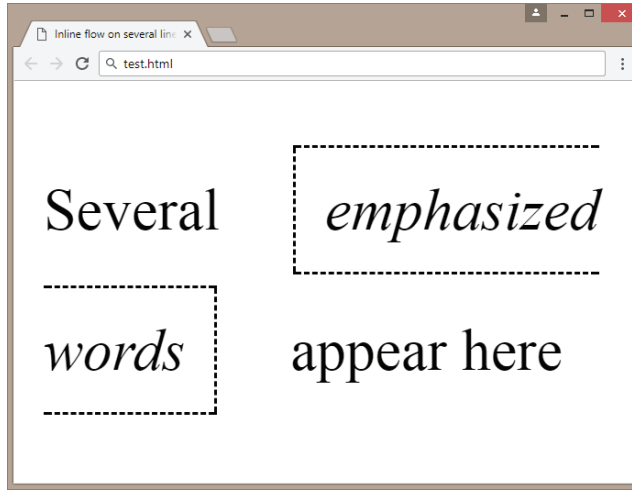


Рис. 8. Строчный бокс, соответствующий строчному элементу em, был разрезан на границе лайн-бокса. Границы и поля в месте разрыва не учитываются

Элемент (например, div), у которого свойство display имеет значение block, визуализируется как блок, и его контент организуется соответствующим образом — так, как это предписано блочным контекстом форматирования. Таблица сама визуализируется как блок, в частности, начинается с новой строки, но ее контент организован табличным образом, т. е. по строкам и по ячейкам.

Теперь мы готовы дать иную классификацию элементов, в основу которой положен способ организации контента. Блочные элементы бывают трех типов:

1. Таблицы.
2. Заменяемые элементы, например, блочные картинки.
3. Блок-контейнерные боксы.

Кроме таблиц и заменяемых элементов все блочные боксы являются также блок-контейнерными боксами, т. е. содержат:

а) либо только блочные боксы,

б) либо только строчные боксы и устанавливают строчный контекст форматирования.

Приведем примеры:

```
<div>
    <p>Первый абзац</p>
    <p>Абзац <em>со строчным элементом</em></p>
</div>
```

Здесь блочный бокс, соответствующий div'у, является блок-контейнерным боксом и содержит только блочные боксы — два абзаца (случай *а*). Абзацы тоже является блок-контейнерным боксом, но для них установлен строчный контекст форматирования, они содержат (если вообще содержат) только строчные боксы (случай *б*).

```
<div>
    Некоторый текст
    <p>Абзац с текстом</p>
</div>
```

Здесь в блочном элементе div одновременно встречаются срочный и блочный контенты. Чтобы упростить раскладку элементов и форматирование, вводятся в рассмотрение так называемые *анонимные блоки*. В данном примере такой анонимный блок окружает строчный контент «некоторый текст». Иными словами, если блок-контейнерный бокс (как div в данном примере) содержит блочные боксы (как p в данном примере), то имеющиеся строчные куски насильственно заключаются в анонимные блоки таким образом, чтобы внутри блок-контейнерного бокса были только блочные боксы.

При создании html-таблиц возникают и иные типы анонимных блоков, однако их изучение выходит за рамки настоящего пособия.

С точки зрения организации контента строчные боксы бывают следующих видов:

1) собственно строчные боксы (англ. inline box),

2) атомарные строчные боксы (англ. atomic inline-level box), которые, в свою очередь, делятся на:

а) незаменяемые строчные элементы,

б) заменяемые строчные элементы и строчные таблицы.

Собственно строчные элементы — это строчные элементы (т. е. чье свойство display имеет значение inline), контент которых принимает участие в строчном контексте форматирования родителя. Например:

```
<a>Абзац <em>со строчным элементом</em></a>,
```

здесь контент строчного элемента em принимает участие в строчном контексте форматирования родителя — тега a.

Атомарные строчные боксы — это агрегированное название для блочно-строчных элементов (display: inline-block), заменяемых строчных элементов (типа строчной картинки) и строчных таблиц (display: inline-table). Они называются атомарными, поскольку принимают участие в строчном контексте форматирования родителя как единое непрозрачное целое.

По большому счету, эта классификация основана на том, может ли строчный бокс поделиться на части (например, на границе слов) и начаться в одной строке (в одном лайн-боксе), а закончиться в следующей строке (в новом лайн-боксе). Например, a, em и span могут, а строчная картинка или строчная таблица — нет.

Чтобы обратить внимание на то, что способ визуализации и способ организации внутреннего контента — разные понятия, рассмотрим незаменяемые элементы, чье свойство display имеет значение inline-block. По способу визуализации они являются строчными, а по способу организации контента — блок-контейнерными боксами. Или снова вернемся к таблицам: по способу визуализации они блочные, а по организации контента — табличные. А строчные таблицы по способу визуализации строчные, а по организации контента — табличные. Все это многообразие определяется свойством display.

3.5. Позиционирование и поток

Как было сказано ранее, в HTML формирование элементов на странице происходит сверху вниз согласно исходным кодам разметки. Элементы, размещенные в самом верху кода, отобразятся раньше тех, которые расположены в коде ниже. Такая логика позволяет легко прогнозировать результат вывода элементов и управлять им.

Далее мы неформально описали поток как порядок вывода объектов на странице и сказали, что существует несколько возможностей «вырвать» элемент из потока и позиционировать его в соответствии с потребностями дизайнера. Теперь 1) дадим строгое определение потока (точнее, потока элемента), 2) укажем, как и относительно чего вычисляются смещения при позиционировании, 3) разберем, как влияет позиционирование на раскладку братских и дочерних элементов.

Элемент называется *вырванным из потока* (англ. out of flow), если он плавающий, абсолютно позиционированный или корневой. Элемент *принадлежит потоку* (англ. in flow), если он не вырван из него. *Потоком элемента X* называется множество, состоящее из элемента *X* и всех принадлежащих потоку элементов, чей ближайший вырванный из потока предок — это *X*. Поясним последний пункт определения на примере:

```
div  {position: absolute;}
     <body>
     <div>
          <p id='first'>Первый абзац</p>
          <p id='second'>Второй <b>абзац</b></p>
     </div>
</body>
```

Здесь элементом *X* является абсолютно позиционированный div, он вырван из потока и устанавливает свой собственный поток. Абзацы, которые являются потомками div'а, принадлежат его потоку. Элемент *b* также принадлежит потоку блока div.

Если же вырвать из потока один из абзацев, например, следующим образом:

```
#first {position: absolute;},
```

то только второй абзац со своим потомком — элементом *b* будет принадлежать потоку блока div.

Чтобы изучать позиционирование, необходимо понимать, относительно чего рассчитываются смещения, задаваемые свойствами top и left, т. е. необходимо знать систему отсчета. Такой системой отсчета является вмещающий блок (англ. containing block), дадим его определение.

1. Вмещающий блок для корневого элемента — это так называемый начальный вмещающий блок (англ. initial containing block). В случае отображения на экране начальный вмещающий блок имеет размеры вьюпорта[5], т. е. в простейшем случае задается границами окна браузера. В случае отображения на печатной странице — размерами страницы.

2. Для элементов, позиционированных как relative или static, вмещающий блок определяется краем контента ближайшего предка, который является блок-контейнерным боксом.

3. Для элементов, позиционированных как fixed, вмещающий блок определяется вьюпортом.

4. Для элементов, чье свойство position имеет значение absolute, вмещающий блок задается ближайшим предком, чье свойство position имеет значение absolute, relative или fixed; если же такого предка нет, то вмещающий блок формально определяется как начальный вмещающий блок. При этом если этот предок — блочный элемент, то вмещающий блок определяется краем полей этого позиционированного предка. Если же этот предок является строчным элементом, то стандарт не устанавливает, как именно определять края вмещающего блока.

---

[5] Вьюпорт (англ. viewport) — область просмотра. Подробно концепция вьюпорта разбирается в следующих главах.

Мы рекомендуем никогда не позиционировать строчные элементы, так как это может сделать верстку непредсказуемой.

Разберем примеры, поясняющие определение вмещающего блока.

```
<! DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN">
<html>
    <head>
        <title>illustration of containing blocks</title>
    </head>
    <body id="body">
        <div id="div1">
        <p id="p1">The first paragraph</p>
        <p id="p2">This is text <em id="em1"> in the
        <strong id="strong1">second</strong> paragraph</em></p>
        </div>
    </body>
</html>
```

Здесь не задано позиционирование, соответственно, раскладка элементов починяется обычному потоку (normal flow), а свойство position получает значение по умолчанию, т. е. static.

- Для элемента html вмещающим блоком будет начальный вмещающий блок, что соответствует п. 1 определения, так как html является корневым элементом.
- Для всех остальных боксов, сгенерированных элементами из первого столбца табл. 5, вмещающий блок определяется соответствующим элементом, указанным во втором столбце. Например, body вложен в html, который является блок-контейнерным боксом, а потому вмещающий блок для body определяется элементом html; это соответствует п. 2. определения.
- Рассмотрим элемент strong, который вложен в элемент em. Поскольку родителем элемента strong (элемент em) является обычный строчный элемент, то ближайшим блок-контейнерным предком для strong является элемент p с идентификатором id = p2, и именно он определяет вмещающий блок для strong.

Таблица 5

Соответствие элементов и их вмещающих блоков

Кто сгенерировал бокс	Чем определяется вмещающий блок
html	Начальный вмещающий блок
body	html
div1	body
p1	div1
p2	div1
em1	p2
strong1	p2

Добавим позиционирование двух элементов:

```
#div1 {position: absolute; left: 50px; top: 50px}
#em1 {position: absolute; left: 10px; top: 10px}
```

Разберем имевшие место изменения (табл. 6):

- Для элемента div1 вмещающим блоком будет начальный вмещающий блок, что соответствует п. 4 определения.
- Для элемента em1 вмещающим блоком будет div1, что соответствует п. 4 определения.
- Согласно таблице стилей браузера, свойство display у элемента em имело значение inline. Касалось бы, строчный элемент em не должен образовывать блок-контейнерный бокс, а потому не может являться вмещающим элементом для strong. Однако, согласно спецификации CSS2.1, раздел 9.7 «Relationships between display, position, and float», для абсолютно позиционированных строчных[6] элементов вычисленное значение свойства display равно block. Таким образом, абсолютно позиционированный элемент em является блочным и образует блок-контейнерный бокс, который является вмещающим элементом для strong.

[6] А также для строчно-блочных элементов.

Таблица 6

Соответствие элементов и их вмещающих блоков

Кто сгенерировал бокс	Чем определяется вмещающий блок
html	Начальный вмещающий блок
body	html
div1	Начальный вмещающий блок
p1	div1
p2	div1
em1	div1
strong1	em1

Теперь, когда читатели разобрались с понятием вмещающего блока, который служит системой отсчета при позиционировании, перейдем к анализу серии примеров. Рассмотрим верстку:

```
<body>
    <div>Begin of div
        <p>The paragraph</p>
    </div>
    The end of div content
</body>
```

Зададим стили

```
body    {   margin: 10px;
            border: solid green 20px;
            padding: 10px}

div     {   border: solid blue 2px;
            padding: 20px;
            background: yellow;
            }

p       {   border: solid black 2px;}
```

Когда не задано позиционирование элементов, они раскладываются согласно нормальному потоку:

1. Добавим позиционирование блоку div. При этом не будем задавать значения свойствам left и top, они получат значения по умолчанию.

```
div  {position: absolute}
```

Положение блока div не изменится. Если свойство left (для top то же самое) не задано, то оно получает значения auto; значение свойства auto вычисляется таким образом, чтобы элемент остался в той позиции, где он бы находился, если бы его свойство position имело значение static[7].

Поскольку блок div абсолютно позиционирован, то он (согласно определению потока) вырван из потока своего родительского элемента body. Можно сказать, что div не является контентом для body и не определяет его высоту; иными словами, body является пустым. В средствах разработчика (кнопка F12 в браузере) можно убедиться, что область контента body имеет высоту, равную 0.

2. Зададим значения свойствам left и top.

```
div  {    top: 0px;
          left: 0px}
```

Согласно п. 4 определения вмещающего блока, в данном примере отсчет смещений будет производиться от границ окна браузера, поскольку вмещающим блоком для div'а является начальной вмещающий блок.

3. Изменим значения свойств left и top.

```
div  {    top: 100px;
          left: 100px}
```

[7] Детали вычисления значений свойств top и left следует изучать по документации.

Точка отсчета для смещений для блока div не изменилась — это по-прежнему граница окна браузера. Отметим другое, div задает свой поток, все элементы, которые в него вложены, позиционируются вместе с ним.

4. Зададим позиционирование для body. Изменим значения свойств left и top для блока div.

```
body    {   position: absolute;
            width: 400px; /*зададим ширину явно*/}
div     {   top: 0px;
            left: 0px}
```

Согласно п. 4 определения вмещающего блока, теперь отсчет смещений блока div будет производиться от края полей body. В этом легко убедиться: а) изменяя значения свойств top и left у блока div и б) изменяя размеры полей у body.

```
body    {   padding: 30px}
div     {   top: 10px;
            left: 10px}
```

Изменение размеров полей у body не приводит к изменению положения той точки, откуда эти поля начинаются, а потому не влияет на положение блока div. А вот если изменить размеры границ у body:

```
body    {   border: solid green 1px;
            padding: 10px},
```

то изменится положение края вмещающего блока для блока div. Именно изменяющиеся границы body «таскают» за собой позиционированный блок div.

Отметим, что в данном тесте можно было задать относительное позиционирование для body, это бы не повлияло на логику рассуждений.

Контрольные вопросы

1. Правда ли, что вычисленное и специфицированное значения свойства display всегда совпадают?
    1. Да, это верно. (Зачем введено такое ограничение?)
    2. Нет, это неверно. (Приведите контрпример.)
2. Правда ли, что начальным значением (initial value) свойства display у всех элементов является значение inline?
    1. Если это верно, то почему тогда без написания автором каких бы то ни было стилей абзацы и заголовки обрабатываются как блочные элементы?
    2. Если это неверно, то какое начальное значение свойства display у элементов?
3. Дайте определение вмещающего блока (англ. containing block). Приведите примеры, когда вмещающим блоком является родительский элемент и когда это не так.

Глава 4
БЛОЧНАЯ ВЕРСТКА: ПЛАВАЮЩИЕ ЭЛЕМЕНТЫ, ВИЗУАЛИЗАЦИЯ

Для того чтобы лучше управлять раскладкой элементов на странице, введены так называемые *плавающие элементы* (англ. float elements). Плавающим называется бокс, который сдвигается в текущей строке влево или вправо (в зависимости от значения, которое принимает свойство float — left или right) до конца. В дальнейшем мы будем рассматривать все правила для случая, когда свойство float имеет значение left[8]; для случая right все будет симметрично, соответствующие формулировки оставим читателю в качестве несложного упражнения.

4.1. Плавающие элементы: основные свойства

Кратко перечислим некоторые важные особенности раскладки плавающих элементов и их соседей.

1. Наиболее интересным свойством плавающих элементов является то, что соседний контент может обтекать их по краю (прекратить такое обтекание можно с помощью свойства clear).

2. Ранее мы сказали, что плавающий элемент сдвигается в бок «до конца» или «до упора». А что значит «до упора»? Плавающий

[8] При этом мы будем предполагать, что свойство direction имеет значение ltr (от англ. left to right), т. е. выбрано направление слева направо.

элемент сдвигается влево до тех пор, пока левый внешний край (т. е. край отступов) не упрется в край контента родительского элемента. Либо, если несколько плавающих элементов выстроилось в ряд, до тех пор, пока левый внешний край не упрется в правый внешний край другого плавающего элемента, расположенного в текущей строке левее данного. При этом если очередному плавающему элементу не хватило в текущей строке места (потому что это место заняли другие плавающие элементы), то он смещается вниз до тех пор, пока либо ему не хватит места в новой (нижележащей) строке, либо в новой строке уже не останется других плавающих элементов.

3. Поскольку плавающие элементы не принадлежат потоку, непозиционированные блоки, созданные до и после плавающего элемента, раскладываются вертикально так, как если бы плавающего элемента не было бы вовсе. Однако текущие и последующие лайн-боксы, созданные браузером после (читай «левее на странице») плавающего элемента, укорачиваются, и место занимает плавающий элемент со своими отступами (англ. margin box of the float).

Проиллюстрируем поведение плавающих элементов на примерах:

```
<html>
    <head>
        <title>Float elements</title>
        <style>
        img {    float: left;
                 border: solid 1px;
                 margin: 10px;}
        </style>
    </head>
    <body>
        <p>Наиболее интересным (X)
        <img src='car.jpg' alt='car'>свойством плавающих элементов является
        то…</p>
    </body>
</html>
```

Плавающая картинка вставлена в исходный текст html-документа в месте, которое обозначено крестом (X). Поскольку свойство float имеет значение left, картинка, в соответствии со вторым свойством плавающих элементов, сдвинулась влево до упора — правый край отступов картинки уперся в край контента родительского элемента, т. е. body (рис. 9). Текст обтекает картинку справа в соответствии с первым свойством плавающих элементов.

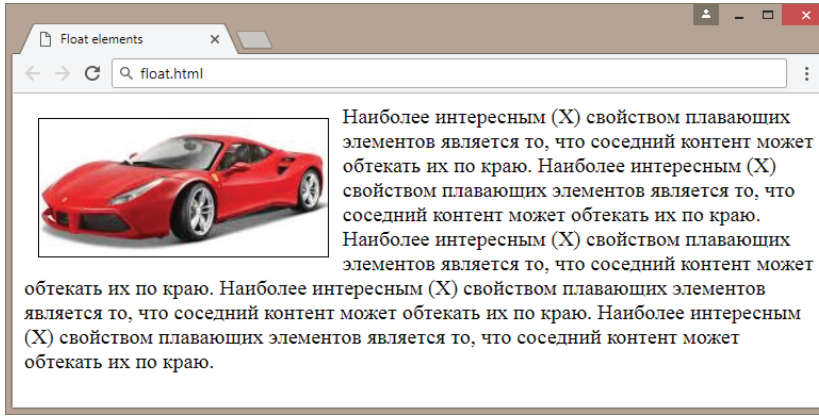


Рис. 9. Обтекание плавающего изображения. Положение изображения в исходном тексте html-документа обозначено (X)

Теперь добавим на веб-страницу еще одну плавающую картинку, между картинками вставим текст.

```
<body>
    <p>Наиболее интересным (X)
    <img src='car.jpg' alt='car'> свойством
    <img src='tractor.jpg' alt='tractor'>
    плавающих элементов является то…</p>
</body>
```

В соответствии со вторым свойством плавающих элементов обе картинки уплыли влево до упора, при этом левая граница картинки с трактором упрется в правый край картинки с машиной.

При уменьшении ширины окна браузера картинке с трактором перестанет хватать места в текущей строке, и она сместится вниз, где ей хватит места.

Разберем третью особенность плавающих элементов. На рис. 10 показано расположение двух блочных элементов, абзацев, в обычном потоке и одного плавающего элемента. В html-документе плавающий элемент описан там, где нарисован крест (X).

```
img{ float: left;
     border: solid black 1px;
     margin: 10px;}
p{   padding: 3px;
     border: solid pink 1px;
     margin: margin: 20px 10px;
     /* top & bottom, right & left*/}
```

Мы видим, что абзацы располагаются так, как если бы плавающей картинки не было бы вовсе. Вертикальные отступы между абзацами схлапываются, картинка этому не мешает. Кроме того, мы

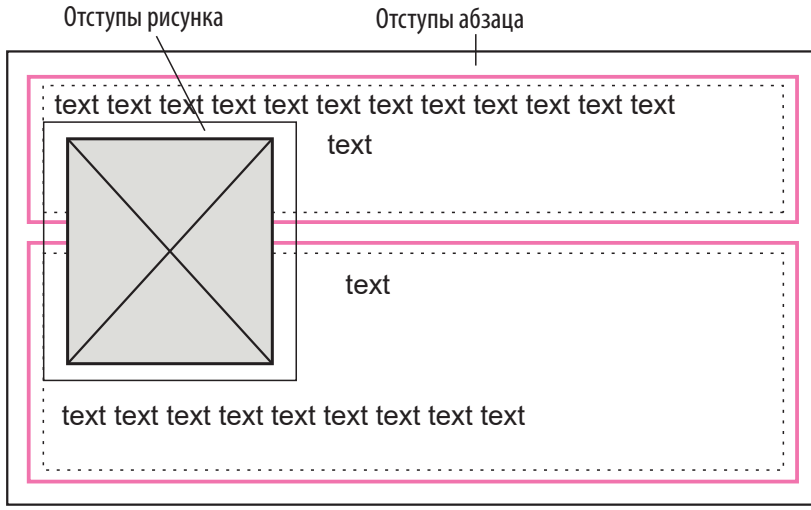


Рис. 10. Раскладка блочных элементов

видим, что сами абзацы не стали у́же (см. розовую границу абзацев); уже стали лайн-боксы, которые раскладываются в абзацах. Не только в первом абзаце, где плавающий элемент был описан, но и во втором абзаце, куда частично попал плавающий элемент, лайн-боксы становились у́же на ширину плавающего элемента с его отступами.

Теперь отметим, что отступы плавающих элементов не сливаются. Рассмотрим пример:

```
<html>
    <head>
        <title>float example</title>
        <style type="text/css">
            img {float: left}
            body, p, img {margin: 2em}
        </style>
    </head>
    <body>
        <p><img src="img.png" alt="float">
            some sample text that has …
    </body>
</html>
```

Вертикальные отступы у body и абзаца p схлопнулись, а отступы у картинки не схлапываются с соседними отступами абзаца или тела (рис. 11).

Рассмотрим еще один пример. Внутри абзаца вложен плавающий span с явно заданными размерами, далее следует очень длинное слово[9]:

```
p {     width: 10em;
        border: solid 1 px aqua;}
```

[9] Supercalifragilisticexpialidocious — название песни из фильма «Мэри Поппинс» (1964). Смысл этого 34-буквенного слова в фильме объясняется как «слово, которое говорят, когда не знают, что сказать».

```
span {    float: left;
          width: 5em;
          height: 5em;
          border: solid 1px blue;}

<p>
    <span> </span>
    Supercalifragilisticexpialidocious
</p>
```

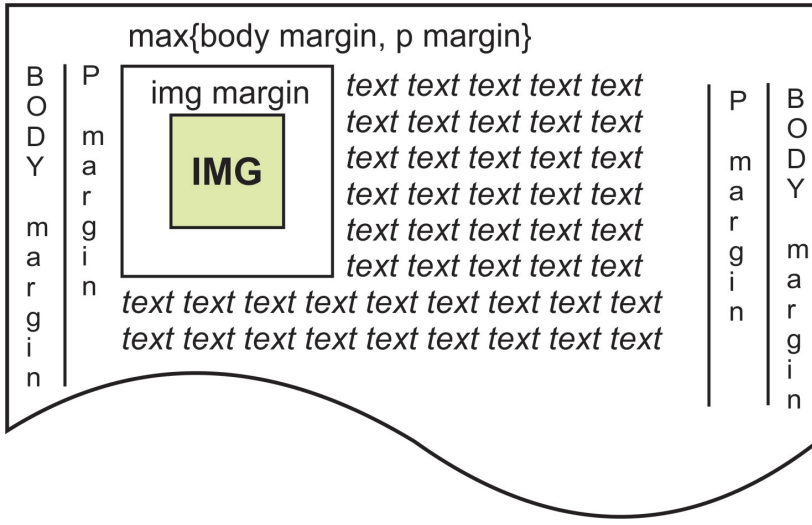


Рис. 11. Отступы плавающих элементов не схлапываются

Длинное слово не вместилось справа от плавающего элемента и сместилось вниз. Под плавающим элементом ширина лайн-боксов стала больше, но все равно была недостаточна для того, чтобы слово поместилось в абзац. Дальнейшее смещение слова вниз бессмысленно, так как лайн-боксы от того шире не станут. Слово «отрисовывается» так, как показано на рис. 12, невместившийся «хвост» виден за правой границей абзаца. Отметим, что видимостью невместившегося контента можно управлять с помощью свойства visibility.

Рис. 12. Текст сместился вниз, под плавающий элемент span, но не вместился в абзац

4.2. Свойство clear

Плавающие элементы имеют замечательное свойство: контент следующих (по исходникам) элементов может обтекать их по краю. Иногда, однако, обтекание требуется прекратить; данная возможность предусмотрена в CSS, соответствующее свойство — clear. Свойство clear может принимать значения left, right, none и both.

Со значением none все ясно, обтекание происходит по обычным правилам. Разбираться с эффектом, который дают другие значения, будем обстоятельно, т. е. не просто скажем, что «что-то куда-то уплывает, а что-то прекращает обтекать соседей», а формализуем эффект свойства clear в терминах раскладки элементов на веб-странице.

Значения, отличные от none, приводят к появлению клиренса (англ. clearance). Клиренс отменяет схлопывание границ и действует как пространство над верхним отступом элемента. Клиренс служит для того, чтобы сдвинуть элемент вниз и разместить его после float-элемента.

Выше мы отметили, что непозиционированные блоки, созданные до и после плавающего элемента, раскладываются вертикально так, как если бы плавающего элемента не было бы вовсе, и привели соответствующий пример (рис. 10). При этом текст во втором абзаце обтекает картинку. Что нужно сделать в терминах раскладки элементов на странице, чтобы такое обтекание убрать? По сути, необходимо сдвинуть второй абзац вниз так, как показано на рис. 13, т. е. добавить дополнительное вертикальное расстояние, которое

называется клиренс; при этом перестают схлапываться отступы первого и второго абзацев.

Прежде чем приступать к вычислению клиренса для элемента, у которого установлено свойство clear, отличное от none, определяется гипотетическая позиция верхнего края его границы (англ. top border edge). Что такое гипотетическая позиция? Это позиция, где верхний край границы данного элемента должен был бы быть, если бы у элемента свойство clear имело значение none. Если эта гипотетическая позиция находится не ниже (т. е. не после) соответствующего плавающего элемента, то вводится клиренс.

Величина клиренса устанавливается как максимум из двух величин:

1) величина, необходимая для того, чтобы поместить верхний край границы элемента после внешнего края самого нижнего float'a, для которого свойство clear имеет значение не none;

2) величина, необходимая для того, чтобы оставить верхний край границы элемента (блока) на его гипотетической позиции.

Рассмотрим пример: для простоты изложения предположим, что существует ровно три бокса, которые в html-файле описаны

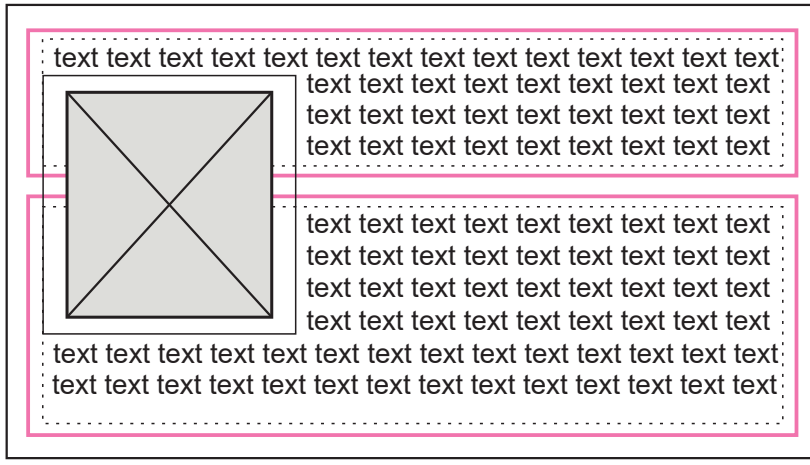


Рис. 13. Иллюстрация клиренса
на примере двух абзацев и плавающей картинки

в следующем порядке: блок B1 с нижним отступом M1 (B1 не имеет потомков, границ и полей), плавающий элемент F высоты H, блок B2 с верхним отступом M2 (нет детей, границ, полей). Для B2 установлено clear: left.

Существует два и только два принципиально различных варианта расположения элементов. В методических целях первый вариант разобьем на два случая и опишем их формально:

1) блок B2 не ниже плавающего элемента:

а) отступ M2 меньше отступа M1, $M2 < M1$,

б) отступ M2 больше отступа M1, $M1 < M2 < M1 + H$;

2) блок B2 ниже плавающего элемента, $M1 + H < M2$.

На рис. 14 и 15 приведено схематичное расположение блоков относительно друг друга. Ось ординат направим вниз, за начало отчета выберем нижний край блока B1.

Первым рассмотрим случай 1,*б* как самый показательный. Согласно определению клиренса, необходимо вычислить две величины (обозначим их C1 и C2) и взять максимум из них.

1. Поместим верхний край границы блока B2 под F, т. е. на позицию $y = M1 + H$. Поскольку отступы более не схлопываются (их разделяет клиренс), имеем:

нижний край F = верхний край границы B2,

$M1 + H = M1 + C1 + M2$,

$C1 = H — M2$.

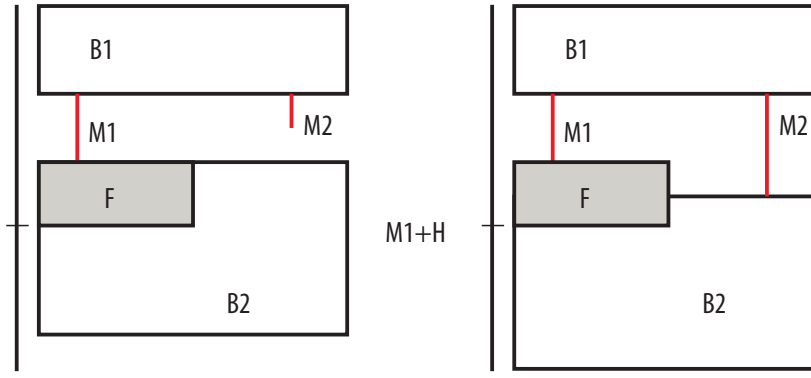


Рис. 14. Верхний край границы блока B2 не ниже плавающего элемента

2. Оставим верхний край границы блока B2 там, где он должен был бы быть, если бы свойство clear имело значение none, т. е. на позиции y = max{M1, M2}.

max{M1, M2} = M1 + C2 + M2,

M2 = M1 + C2 + M2,

C2 = –M1.

Поскольку, согласно предположению 1,*б*, M2 < M1 + H, то –M1 < –M2 + H и max{C1, C2} = H — M2. Соответствующий сдвиг блока B2 показан на рис. 16. Случай 1,*а* полностью аналогичен, рекомендуем читателю самостоятельно провести те же выкладки.

Теперь рассмотрим случай 2: здесь отступ M2 настолько большой, что блок B2 уже находится под плавающим элементом. Также необходимо вычислить C1 и C2, а потом взять максимум из них. На этот раз, в силу предположения 2, имеем max{C1, C2} = –M1. Соответствующий сдвиг блока B2 показан на рис. 14.

Суть этого клиренса в следующем. Блок B2 должен остаться на том месте, где был; однако так как отступы M1 и M2 перестали схлапываться, теперь отступ M1 тоже пытается сдвинуть блок B2 вниз. Чтобы блок остался там, где был, клиренс должен «противодействовать» дополнительному отступу M1. На рис. 17 клиренс обозначен стрелкой, направленной вверх.

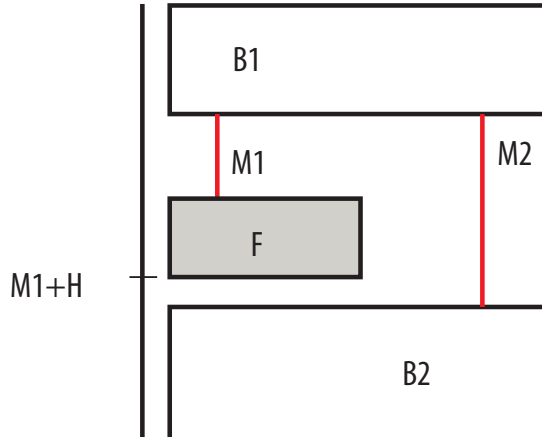


Рис. 15. Верхний край границы блока B2 ниже плавающего элемента

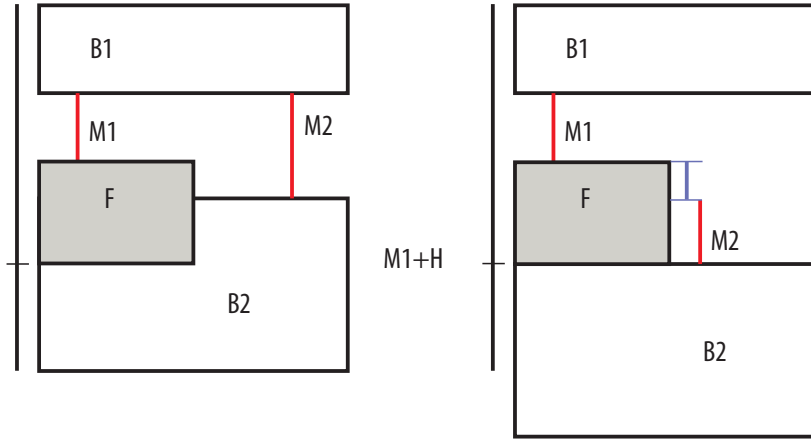


Рис. 16. Сдвиг блока B2, добавление клиренса C = H — M2

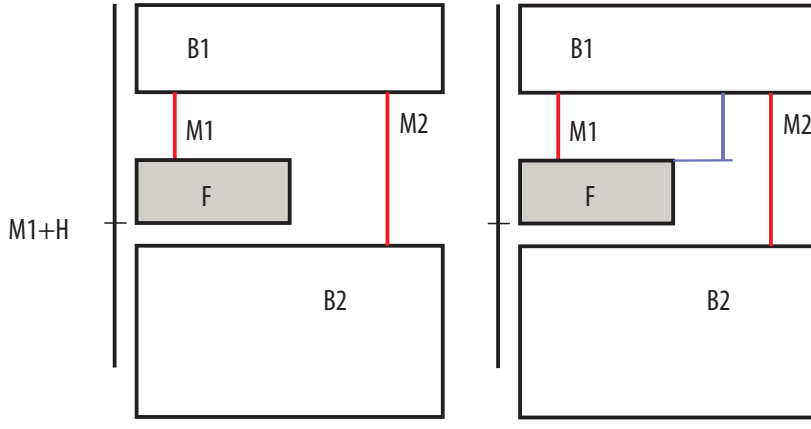


Рис. 17. Блок B2 остался на своей первоначальной позиции,
был добавлен клиренс C = –M1

4.3. О связи между positioning, float и display

Как было отмечено во второй главе, специфицированное значение свойства может отличаться от вычисленного. Три свойства — display, position и float взаимодействуют весьма причудливым

образом, специфицированные значения одних могут повлиять на то, как будут вычислены значения других.

В данной главе мы уже столкнулись с тем, что абсолютно позиционированный строчный элемент em стал определять вмещающий блок для своих потомков, а такое поведение характерно для блочных элементов. Разберемся, почему так произошло, какие CSS-правила диктуют такое поведение браузеров.

- Если свойство display имеет значение none, то свойства position и float не применяются. В этом случае элемент не порождает бокс, он изымается из потока, а его место занимают другие элементы.
- Иначе, если свойство position имеет значение absolute или fixed, вычисленное значение свойства float равно none; это означает, что абсолютное позиционирование имеет приоритет над свойством float. Действительно, если «плавание» диктует сдвиг влево, а позиционирование — сдвиг в другое место, то возникает конфликт. Стандарт предписывает разрешать этот конфликт в пользу свойства position. При этом вычисление значения свойства display происходит согласно табл. 7.
- Иначе, если свойство float имеет значение, отличное от none, а вычисление значения свойства display происходит согласно табл. 7.

Дадим комментарии к табл. 7. В главе 3 был разобран пример, иллюстрирующий понятие вмещающего блока (см. главу 3, табл. 6).

Таблица 7

Соответствие специфицированных и вычисленных значений свойства display для абсолютно позиционированных и плавающих элементов

Специфицированное значение	Вычисленное значение
inline-table	table
inline, inline-block, table-row-group, table-column, table-column-group, table-header-group, table-footer-group, table-row, table-cell, table-caption	block
Иначе	Без изменений

Для абсолютно позиционированного элемента em (строчного) вычисленное значение свойства display равно block. По этой причине em становится ближайшим блочным предком для элемента strong, и, как следствие, em определяет вмещающий блок для strong.

Еще раз отметим, что не рекомендуется делать строчные элементы абсолютно позиционированными или плавающими, так как возникают проблемы с определением краев порождаемого ими вмещающего блока (рис. 18). Действительно, как определить края для элемента, который начался у правой границы окна браузера, потом разорвался, был перенесен на следующую строку и там закончился у левой границы окна браузера? Получается, что левый край правее правого…

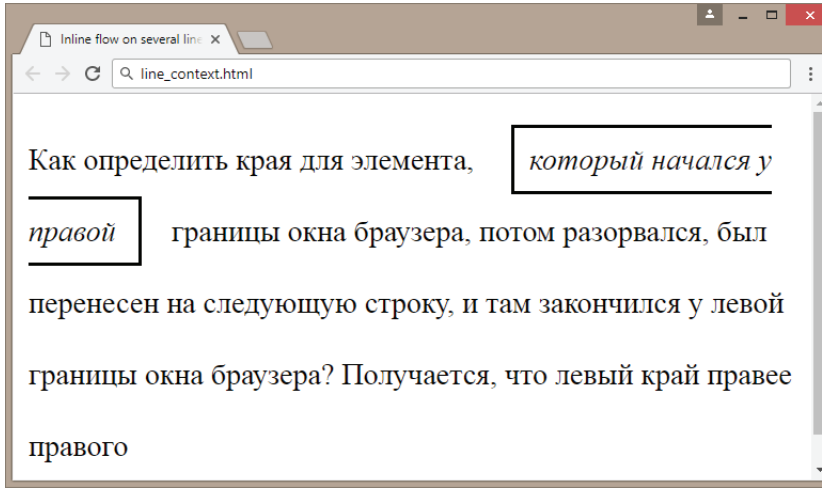


Рис. 18. Проблемы, вызванные позиционированием строчных элементов

4.4. Замечание о лайн-боксах

Прямоугольник, который содержит боксы, формирующие одну строку, называется лайн-боксом. Ширина лайн-бокса определяется шириной вмещающего бокса и наличием плавающих элементов. Ширина лайн-боксов, находящихся в одном строчном контексте

форматирования, обычно постоянная; однако если появляется флоат, то ширина лайн-боксов уменьшается.

Строчные боксы, образующие один лайн-бокс, могут иметь разную высоту, их выравнивание в строке определяется свойством vertical-align.

Если строчный бокс разбивается на части и части размещаются в различных лайн-боксах, то отступы, границы и поля не «отрисовываются» в местах разрывов.

4.5. Вычисление ширины элемента: свойство width

Ширина блочных элементов определяется свойством width (пер. с англ. *ширина*) и является одним из самых сложных свойств. В Стандарте CCS2.1, в гл. 10 «Детали визуального форматирования», ему посвящено 10 параграфов.

Сразу отметим, что для строчных элементов свойство width неприменимо. Ширина незамещаемого строчного элемента (например, span или em) определяется его контентом; ширина замещаемых строчных элементов (например, картинок), как правило, определяется внешними метаданными[10].

Разберем, как вычисляется ширина блочного незамещаемого элемента в нормальном потоке; все остальные правила, аналогичные разбираемому ниже, читатели смогут изучить по документации.

Согласно формуле (1), сумма ширин отступов, границ, полей и самого элемента должна быть равна ширине вмещающего блока.

margin-left + border-left-width + padding-left + **width** + padding-right + border-right-width + **margin-right** = width of containing block (1)

В формуле (1) полужирным шрифтом выделены те слагаемые, которые могут принимать значения auto. По умолчанию значением свойств border-left-width, padding-left, padding-right и border-right-width является 0, т. е. не заданные явно поля и границы имеют нулевые размеры. А вот для ширины отступов и контента, т. е. свойств

[10] Для графических файлов высота и ширина в пикселях известны.

margin-left, margin-right и width, значением по умолчанию является auto; и это значение должно быть вычислено на этапе рендеринга страницы. Основные проблемы, связанные со свойством width, как раз и состоят в том, чтобы понять, как определяются конкретные значения, выраженные в абсолютных единицах, когда width имеет значение auto.

Разберем сначала простой случай (I), когда ширина вмещающего блока достаточна для размещения его потомка (см. рис. 16, верхняя часть).

Определившись с тем, чему равна ширина блока, необходимо различать два принципиальных случая:

1) ширина не определена в CSS-файле, т. е. width: auto;

2) ширина определена в CSS-файле, например, width: 100 px.

Разберем случай 1. Как это принято у блочных элементов, если ширина явно не задана, то блочный элемент занимает всю доступную ему ширину. При этом auto-отступы полагаются равными нулю. Систематизируя, получаем:

а) если отступы[11] явно заданы, например, margin: 10 px, то в формуле (1) имеется только одно неизвестное, которое можно найти, решая уравнение (1);

б) если отступы (один или оба) определены как auto, то они полагаются равными нулю, т. е. их фактическое значение 0 px, а ширина находится из уравнения (1).

Разберем случай 2. Необходимо выделить три подслучая; эта необходимость напрямую следует из вида формулы (1). Для того чтобы выполнялось равенство (1), необходимо распорядиться теми степенями свободы, которые у нас есть. Такими степенями свободы могут быть либо левый, либо правый отступ (возможно, оба сразу). Напомним, что границы и поля не образуют степени свободы, так как, будучи незаданными, они принимают значение 0, а не auto.

а) Если оба отступа заданы как auto, то в уравнении (1) два неизвестных. Чтобы его решить, задают дополнительное ограничение: margin-left = margin-right, после чего (1) допускает однозначную разрешимость. Побочным хорошим визуальным эффектом явля-

[11] В этом параграфе будем говорить только о горизонтальных отступах.

ется то, что блочный элемент центрируется в своем родительском контейнере.

Следует отметить, что для центрирования блочного элемента не удастся использовать конструкцию вида text-align: center, так как данное свойство применяется для строчного контента внутри блока. То есть центрироваться будет не сам блок, а его строчный контент.

б) В случае, когда только один отступ имеет значение auto, он определяется из (1).

в) Оба отступа явно заданы, следовательно, нет ни одной степени свободны в уравнении (1). Если равенство (1) окажется верным, то нам крупно повезло. К сожалению, может так получиться, что равенство (1) окажется неверным — сумма ширин в левой части (1) может оказаться больше или меньше ширины вмещающего блока. Выше мы договорились рассмотреть простой случай (I), когда в родительском блоке места достаточно; таким образом, если (1) окажется неверным, то сумма ширин в левой части (1) будет меньше ширины вмещающего блока. Как это исправить?

Браузер насильственно переопределяет явно заданный margin-right[12] как auto, т. е. сводит задачу к пункту б). Далее фактическое значение margin-right определяется так, чтобы (1) было верным.

Теперь разберем более сложный случай (II), когда ширина вмещающего блока недостаточна для того, чтобы вместить своего потомка с полями, границами и отступами (см. рис. 19, нижняя часть). В основе лежит тот же принцип — надо различать ширину auto и конкретно заданную, однако в дереве разбора появляется еще одна ветвь, которая на рис. 19 показана в правом углу.

Разберем случай 1, width: auto. Данный случай разбирается аналогично, но есть одна проблема. Может оказаться, что сумма полей, границ и отступов, чье значение отлично от auto, больше ширины вмещающего блока. В этом случае вычисленное значение ширины будет отрицательным, таким, чтобы (1) было верным. После этого вычисленное значение ширины будет заменено на фактическое, равное нулю, равенство (1) превратится в неверное. Как браузер будет

[12] Переопределяется margin-right в том случае, когда направление текста задано слева направо, т. е. свойство direction имеет значение ltr.

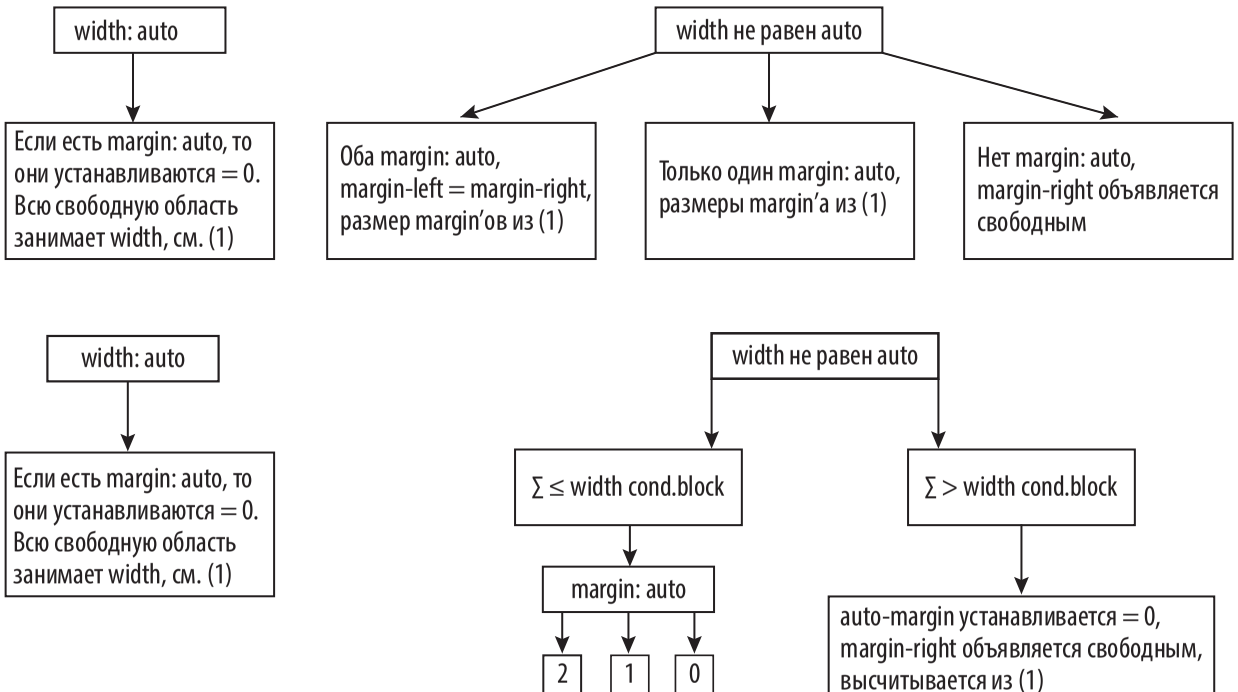


Рис. 19. Правила вычисления ширины блочного элемента

действовать, чтобы (1) снова стало верным, разберем в дальнейшем. Пока считаем, что вычисленное значение width больше нуля.

Разберем случай 2, width не auto. Ситуация, когда сумма ширин в левой части (1) меньше ширины вмещающего блока (на рис. 19 обозначено как ∑ ≤ width cont. block), была разобрана ранее. На рис. 19 цифрами 2, 1 и 0 обозначено количество auto-отступов. Разберем ситуацию, когда сумма ширин в левой части (1) больше ширины вмещающего блока (на рис. 19 это правая нижняя ветка).

Во-первых, auto-отступы (если такие были) устанавливаются равными нулю. Далее margin-right объявляется равным auto и определяется из (1); разумеется, при этом он получится отрицательным. Фактически в окне браузера отрицательный отступ не будет никак «отрисован», здесь важно понимать принцип работы и определения вычисленных значений свойств.

Теперь вернемся к случаю 1. Мы отметили, что возможна ситуация, когда вычисленное значение свойства width окажется отрицательным. Тогда полагают ширину равной нулю; далее браузер насильственно объявляет правый отступ свободным и вычисляет его, исходя из (1).

4.6. Вычисление ширины элемента: примеры

Рассмотрим простой пример. Внешний блок div класса outer служит вмещающим блоком для внутреннего блока div класса inner и нужен лишь для того, чтобы зафиксировать ширину вмещающего блока, т. е. правую часть в формуле (1). Отступы внешнего блока нужны лишь для улучшения наглядности, они не повлияют на раскладку внутреннего блока. Для того чтобы были видны границы блоков, им заданы свойства border.

```
<html>
<head>
    <title>width</title>
    <style>
    .outer {
```

```
            margin: 30px;
            width: 300px;
            border: solid black 1px;}
        .inner {
            margin: 40px;
            width: auto;
            border: solid red 1px;}
        </style>
    </head>
    <body>
    <div class='outer'>
        <div class='inner'>Внутренний текст</div>
    </div>
    </body>
    </html>
```

Пример показывает, что текстовый контент внутреннего блока div не влияет на его ширину, она равна 300 – 2 × 1 – 2 × 40 = 212 px (рис. 20).

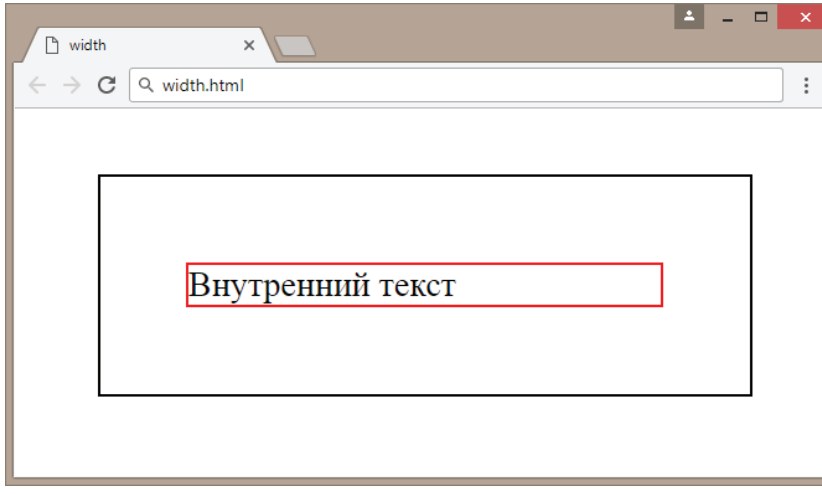


Рис. 20. Вычисление ширины. Блочный элемент в обычном потоке

Далее изменим свойство margin у внутреннего блока:

```
.inner {
        margin: auto;
        width: auto;
        border: solid red 1px;}
```

Теперь отступы внутреннего блока станут равны нулю, и внутренний блок захватит всю доступную ширину, она будет равна 298 px.

Контрольные вопросы

1. Изучите свойства min-width и max-width. Как наличие этих свойств отражается на алгоритме вычисления фактического значения ширины блочных элементов?

2. В главе 4 был разобран алгоритм вычисления ширины блочного незамещаемого элемента в нормальном потоке. Приведите аналогичный алгоритм вычисления ширины абсолютно позиционированного блочного незамещаемого элемента. Как наличие свойств left и right отражается на алгоритме вычисления ширины блочного элемента?

3. Если ширина элемента оказалась больше ширины вмещающего блока, будет ли невошедшая часть отображаться или нет? А может быть появится горизонтальная полоса прокрутки? Чем регламентируется такое поведение браузера? (Указание: изучите свойство overflow и его значения.)

4. Есть два способа не отображать блочный элемент на странице: display: none и visibility: hidden. В чем отличие этих двух вариантов?

Глава 5
РАЗРАБОТКА ПРОГРАММНЫХ КОМПЛЕКСОВ ДЛЯ МОБИЛЬНЫХ УСТРОЙСТВ. АДАПТИВНЫЙ ВЕБ-ДИЗАЙН

В связи с ростом популярности мобильных устройств с выходом в Интернет и увеличением мобильного интернет-трафика веб-разработчики стали уделять большое внимание отображению сайтов на экранах телефонов и планшетов. В связи с этим в 2010 г. сформировалось понятие адаптивного веб-дизайна. Адаптивный веб-дизайн (англ. Adaptive Web Design) — дизайн веб-страниц, обеспечивающий правильное отображение сайта на различных устройствах, подключенных к Интернету (смартфон, планшет, ноутбук, телевизор с выходом в Интернет и т. д.), и динамически подстраивающийся под заданные размеры окна браузера.

В данной главе мы разберем некоторые фундаментальные понятия, лежащие в основе адаптивной верстки, а в конце приведем несколько полноценных примеров.

5.1. Пиксели устройств, референсные пиксели и CSS-пиксели

Следует различать 3 типа пикселей:

1) аппаратный пиксель, т. е. физический пиксель на экране;
2) аппаратно-независимый пиксель;
3) пиксель CSS.

Разберемся с этими понятиями по порядку.

Аппаратный пиксель — это физический пиксель на экране. Например, у iPhone 5 ширина экрана составляет 640 аппаратных пикселей.

Большинство разработчиков знакомы с аппаратным пикселем. Это наименьшая физическая точка на экране, которая содержит в себе красный, зеленый и синий субпиксели. Цвет этих субпикселей смешивается, и мы видим картинку, выстроенную из пикселей. Так как аппаратный пиксель является физическим элементом экрана, то мы не можем его ни растянуть, ни разделить. Эти свойства делают пиксель чем-то вроде атома — единицей дизайна, на которой все построено.

В конце 90-х — начале 2000-х стандартным считалось разрешение мониторов 96 аппаратных пикселей на дюйм. Но техника не стоит на месте: разрешение мониторов увеличивалось, тем самым достигалось лучшее качество изображения. К 2008 г. телефоны с 150 dpi стали новой нормой. Тенденция к увеличению разрешения дисплея продолжалась, и теперь новые телефоны имеют 300 dpi (например, фирменная Retina от Apple).

Не вводя понятия CSS-пикселя и масштабирования, разберем пример. Представим, что задана ширина блока 100 px.

```
div  {width: 100px}
```

Если бы между пикселями в CSS-файле и аппаратными пикселями было соответствие один к одному, то при улучшении качества экрана (при повышении разрешающей способности) в 2 раза абсолютные размеры блока (т. е. размеры блока в дюймах) уменьшились бы в 2 раза. Такое положение дел не может устраивать ни разработчиков браузеров, ни веб-дизайнеров. Соответственно, W3C понадобилось некоторая универсальная единица измерения.

Производители откликнулись на это аппаратно-независимыми пикселями (англ. device-independent pixel или density-independent pixel). Аппаратно-независимый пиксель — единица измерения размеров и положений на экране, представляет собой абстракцию или обертку над аппаратными пикселями. Аппаратно-независимые пиксели используются приложениями, а соответствующая графи-

ческая подсистема в компьютере (например, графический драйвер операционной системы) переводит аппаратно-независимые пикселы в аппаратные.

Типичный пример использования аппаратно-независимых пикселей — масштабирование соответствующим образом элементов графического интерфейса, чтобы они на экранах с разным разрешением отображались по возможности одинаково.

Эта абстракция, или обертка, позволяет удобно работать с аппаратно-независимыми пикселями как с универсальной единицей измерения, а соответствующая графическая подсистема конвертирует их в аппаратные пиксели. Например, в операционной системе Android принято по определению, что один аппаратно-независимый пиксель равен одному физическому пикселю при разрешении 160 пикселей на дюйм; иными словами, один аппаратно-независимый пиксель равен 1/160 доли дюйма. В Windows Presentation Foundation[13] определено, что один аппаратно-независимый пиксель равен 1/96 доли дюйма.

Ширина экрана iPhone 5 составляет 320 аппаратно-независимых пикселей, таким образом, по ширине один аппаратно-независимый пиксель в два раза больше аппаратного пикселя.

Итак, на разных платформах определения аппаратно-независимого пикселя различаются. В Стандарте CSS2.1 (§ 4.3.2) указано, что пиксель — это абсолютная величина, равная 1/96 части дюйма. Заметим, что в предыдущих версиях стандарта пиксель определялся как относительная величина, которая, вообще говоря, может зависеть от разрешения монитора.

Попытаемся разобраться в возможных мотивах разработчиков, которые определили, что в Android пиксель определен не по стандарту. Кроме существующего понятия аппаратно-независимого пикселя, в документации «Cascading Style Sheets Level 2 Revision 1 (CSS2.1) Specification» вводится еще *референсный* пиксель. Прежде чем говорить о референсных пикселях, в стандарте рассматрива-

[13] Windows Presentation Foundation — система для построения клиентских приложений Windows с визуально привлекательными возможностями взаимодействия с пользователем, графическая подсистема в составе .NET Framework, использующая язык XAML.

ется задача, как должны выглядеть пиксели при рассмотрении их с разного нестандартного расстояния[14].

На рис. 21 изображены глаз пользователя, экран, удаленный на стандартное расстояние 71 см, и пиксель, размеры которого составляют 1/96 долю дюйма, т. е. 0.26 мм. Пусть теперь экран удален от пользователя на 3.5 м; если на этом экране отображать пиксель размером 0.26 мм, то пиксель покажется слишком маленьким (скорее всего, человек этот пиксель не заметит). Для того чтобы удаленный пиксель казался таким же, как стандартный, необходимо сделать масштабирование. И это правильно: для смотрящего с большего расстояния картинка должна быть крупнее, чтобы ее удобно было видеть, а повышенная «зернистость» сглаживается глазом. Читатели могут видеть на рис. 21 два подобных треугольника, размер смасштабированного пикселя равен 1.3 мм. Можно сказать, что референсный пиксель — это оптическая единица. Если референсый пиксель удален от пользователя на стандартное расстояние в 28 дюймов, то и размер его стандартный — 1/96 часть дюйма.

При приближении экрана к глазу пользователя размер референсного пикселя должен уменьшаться. Возможно, поэтому в Android принято, что размер аппаратно-независимого пикселя составляет всего 1/160 часть дюйма.

Разберем теперь, как соотносятся аппаратные и аппаратно-независимые пиксели в зависимости от разрешающей способности устройства (рис. 22). Лазерный принтер имеет очень высокую разрешающую способность; одному аппаратно-независимому пикселю может соответствовать 16 аппаратных пикселей. В то же время на мониторе с низкой разрешающей способностью это соотношение может составлять один к одному.

Эта классификация приведена в нашей лекции, чтобы в дальнейшем, когда читатели начнут знакомиться с серьезной литературой и документацией, они не запутались в терминологии. Мы же, как отмечено ранее, будем работать только с аппаратно-независимыми и CSS-пикселями.

---

[14] Стандартным расстоянием считается расстояние вытянутой руки, точнее, расстояние 71 см или, что то же, 28 дюймов.

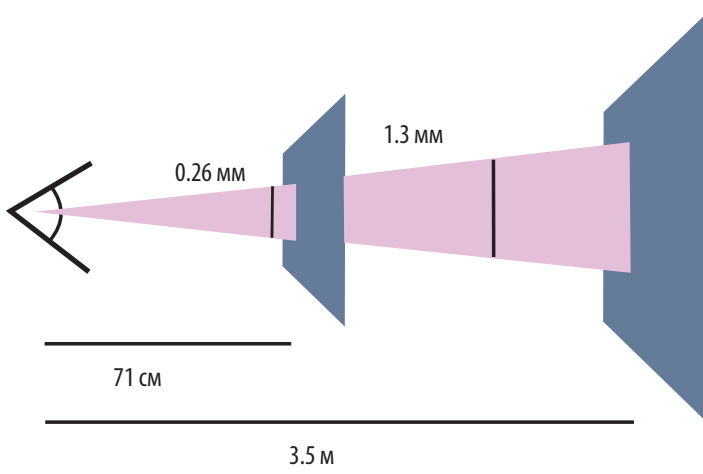
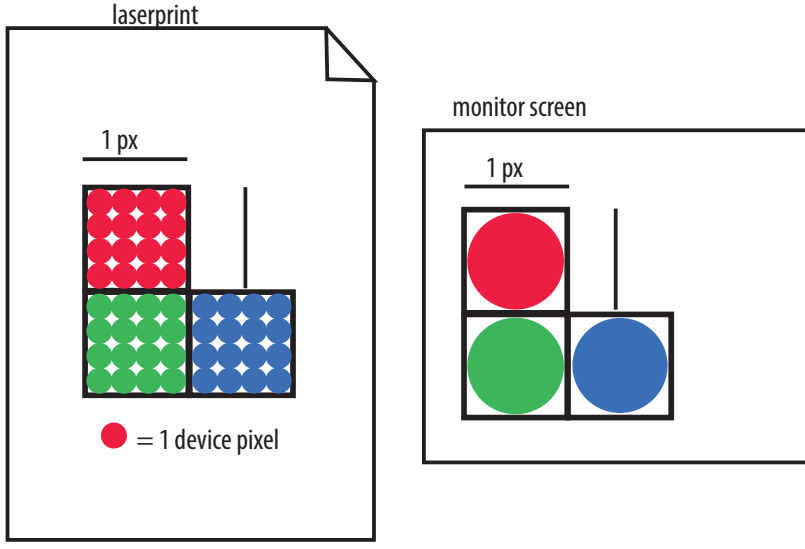


Рис. 21. Зависимость размеров референсного пикселя
от расстояния между зрителем и экраном



а *б*

Рис. 22. Соотношение между аппаратными и аппаратно-независимыми
пикселями в зависимости от разрешающей способности устройства:
а — лазерный принтер; *б* — монитор

Пиксель CSS — единица, используемая для отображения страницы в области просмотра окна браузера. При указании размеров в стилях (например, width: 100 px) используются пиксели CSS. Отношение пикселей CSS к аппаратно-независимым пикселям является коэффициентом масштабирования страницы.

Поскольку веб-разработчик никогда не работает с аппаратными пикселями (он не может получить к ним доступ), мы будем в дальнейшем говорить об экранных и CSS-пикселях, понимая под экранными аппаратно-независимые. Экранные пиксели — вид пикселей, которые мы считаем «реальными». Эти пиксели означают реальное или номинальное разрешение любого устройства и могут быть считаны (в большинстве случаев) с помощью свойства screen.width/height.

Если выставить элементу ширину в 128 px:

```
width: 128px,
```

и развернуть окно браузера на весь экран монитора с разрешением в 1024 px по ширине, то элемент поместится в окне браузера восемь раз[15]. Однако если пользователь увеличит масштаб в браузере, эти расчеты изменятся. Например, если пользователь увеличит масштаб до 200 %, то элемент с шириной 128 px уместится на том же мониторе только четыре раза (см. предыдущую сноску).

Масштабирование в современных браузерах есть не что иное, как простое «растягивание» пикселей. Это означает, что ширина элемента не изменяется от 128 до 256 px, вместо этого пиксели элемента удваиваются в размере. Формально ширина элемента все еще составляет 128 CSS-пикселей, пусть даже кажется, что элемент занимает 256 px устройства; в этом легко убедиться, выведя значение свойства width в консоли браузера.

Другими словами, увеличение масштаба до 200 % делает размер CSS-пикселей в четыре раза больше, чем размер пикселей устройства (увеличение в два раза в ширину и два раза в высоту и дает увеличение в четыре раза).

---

[15] На самом деле чуть меньше, так как нужно еще учесть размеры окна браузера и полосы вертикальной прокрутки.

5.2. Размеры экрана и окна

Выясним, с помощью каких свойств можно узнать размеры экрана, выраженные в аппаратно-независимых пикселях, и размеры окна браузера, выраженные в пикселях CSS. Свойства screen.width/height содержат ширину/высоту экрана пользователя[16]. Заметим, что веб-разработчикам эти свойства практически никогда не нужны; действительно, что толку в том, что ширина экрана составляет 1280 px, если пользователь развернул окно лишь на 960 px? Для раскладки элементов важна ширина окна браузера.

Свойства window.innerWidth/Height содержат ширину/высоту окна браузера, включая полосы прокрутки. Измерения проводятся в CSS-пикселях, а потому при изменении масштаба размеры окна изменяются. Так, при увеличении масштаба до 200 % ширина окна уменьшается в два раза, так как один CSS-пиксель становится в два раза шире и этих пикселей в окне умещается в два раза меньше.

5.3. Ширина в процентах и viewport

Рассмотрим задачу. Предположим, что на странице имеется боковая панель шириной 10 %. При изменении окна браузера эта панель соответствующим образом уменьшается или увеличивается по ширине. Как именно это работает? То есть относительно чего берутся проценты?

Если ширина задана в процентах, то проценты вычисляются относительно ширины вмещающего блока (определение вмещающего блока см. в предыдущей главе и Стандарте CSS2.1, § 10.2).

```
<html>
<head>
    <title>Ширина в процентах</title>
    <style>
```

[16] В браузере Internet Explorer 8 ширина и высота экрана (значение свойств screen.width/height) ошибочно выражаются в пикселях CSS.

```
        body{
            margin: 0px;}
        .sidebar {
            width: 10%;
            background: yellow;}
        </style>
</head>
<body>
    <div class='sidebar'>Колонка</div>
</body>
</html>
```

Родителем боковой панели является body, ширина которого явно не задана; соответственно, встает вопрос: какая ширина у body? Опять же согласно стандарту ширина body равна ширине его родителя — элемента html; а ширина html равна ширине окна браузера. Поэтому в данном примере колонка шириной в 10 % будет занимать 10 % ширины окна браузера. Все веб-разработчики интуитивно это понимают и используют. Наша задача разобраться в теоретической основе такой работы.

Ширина элемента html ограничена шириной области просмотра браузера. Формально, чтобы определить видимую часть страницы, вводится такое понятие, как *вьюпорт*. Вьюпорт — это видимая пользователем часть веб-страницы. Таким образом, функция вьюпорта — ограничить html-элемент, самый внешний блок сайта.

Вьюпорт не является html-конструкцией, поэтому нельзя влиять на него CSS-правилами. Он просто имеет ширину и высоту окна браузера (в десктопных браузерах); в мобильных браузерах с этим немного сложнее.

Такое положение дел приводит к возникновению любопытных последствий. Рассмотрим пример, где ширина горизонтальной панели задана в процентах, а ширина ее контента — в пикселях. После увеличения масштаба на два-три шага некоторая часть контента скроется за пределами окна браузера. При этом синяя панель больше не прилегает должным образом к правой границе сайта.

```
<html>
<head>
    <title>viewport</title>
    <style>
    body{
        margin: 0px;}
    .header {
        width: 100%;
        background-color: blue;}
    .inner {
        width: 800px;
        border: solid red 1px;}
    </style>
</head>
<body>
<div class='header'>
    <div class='inner'>Внутренний текст</div>
</div>
</body>
</html>
```

Разберем теперь это объяснение на языке CSS-пикселей и пикселей устройств (табл. 8).

- При увеличении масштаба ширина html-элемента (как мы помним, она выражается в CSS-пикселях) уменьшается. Действительно, CSS-пиксели становятся шире (т. е. занимают больше экранных пикселей), меньше CSS-пикселей входит в область просмотра, значит, по определению ширина html-элемента уменьшается. Вместе с ней уменьшается ширина синей фоновой панели.
- Отметим, что чисто визуально при зуммировании панель своей ширины не меняет. В CSS-пикселях она становится меньше, но зато каждый CSS-пиксель становится шире, а потому сама фоновая панель как занимала всю область просмотра (в ширину), так и продолжает ее занимать.

- С другой стороны, элементы, ширина которых установлена в пикселях, а не в процентах, при масштабировании сохраняют свою ширину в CSS-пикселях. Соответственно, настает момент, когда сумма ширин элементов превосходит ширину фоновой панели. Визуально это проявляется в том, что контент выходит за границы фоновой панели.

Таблица 8

Влияние масштабирования на ширину элементов

Элемент	CSS-пиксели	Аппаратно-независимые пиксели
Фоновая синяя панель, %	Уменьшается	Без изменений
Контент, px	Без изменений	Увеличивается

Попробуем узнать, чему равны размеры области просмотра и размеры html-элемента. Обычно они совпадают, однако можно явно задать ширину элемента html, и тогда они будут разные (мы не рекомендуем задавать размеры элементу html).

Размеры области просмотра (без полос прокрутки) можно получить, используя свойства

```
document.documentElement.clientWidth/Height
```

Читатель, знакомый с DOM, знает, что documentElement — это ссылка на объект, соответствующий элементу html. Однако при обращении к свойствам clientWidth/Height возвращается ширина именно вьюпорта (это исключение из правил).

Размеры элемента html можно получить, используя свойства

```
document.documentElement.offsetWidth/Height
```

Рекомендуем читателю провести тесты в разных браузерах.

```
<html>
<head>
    <title>viewport and html width</title>
```

```
    <style>
    html{
        width: 400px;
        border: 1px solid;}
    </style>
    <script>
    function f(){
    alert(document.documentElement.clientWidth + ' '
+ document.documentElement.offsetWidth + ' '
+ screen.width) ;}
    </script>
</head>
<body onload='f()'>
    Внутренний текст
</body>
</html>
```

5.4. Два вьюпорта: истоки проблемы

Возможно, читателям понятие вьюпорта показалось ненужным нагромождением. Действительно, зачем вводить лишнее понятие, если и так есть окно браузера, которое задает ширину области просмотра? По-настоящему концепция вьюпорта раскрывается, когда мы начинаем работать с мобильными браузерами.

Когда браузер раскладывает элементы на странице, ему необходимо установить ширину вьюпорта. Для настольных браузеров с широкими экранами можно было ширину вьюпорта задать равной ширине окна просмотра. Рассмотрим теперь телефон с экраном в 320 px (в портретной ориентации). Есть два варианта.

1. Если ширину вьюпорта установить в 320 px, то в большинстве сайтов, которые были сверстаны для широких экранов, верстка «поплывет» или станет непригодной для использования.

Действительно, пусть на сайте есть навигационная боковая панель шириной в 10 %. Если бы мобильные браузеры действовали

так же, как и настольные, они бы сделали этот элемент шириной в 32 px — это крайне мало. Такая резиновая раскладка выглядела бы чересчур сжатой. Пусть на странице вставлена картинка шириной 400 px; ясно, что она выйдет за край экрана, при этом появится горизонтальная полоса прокрутки. Если веб-дизайнер расположил в ряд горизонтального элементы меню, использую float: left, то на узком экране эти плавающие элементы могут не выстроиться в ряд, а начать один за другим перепрыгивать на следующие строки.

2. Чтобы верстка не поплыла, мобильный браузер предполагает, что сайт сверстан на ширину примерно 980 px; и это в большинстве случаев верное предположение. Далее возникает вопрос: как отобразить сайт шириной 980 px на экране в 320 px? Есть два варианта:

а) смасштабировать в 980/320 = 3.06 раза;

б) отобразить левую верхнюю часть, а (980–320) px часть уйдет за правую границу экрана; появится полоса горизонтальной прокрутки.

Вариант 2,*б* всегда плох, разработчики стараются недопустить появление горизонтальной полосы прокрутки. Зачастую в технических заданиях на верстку есть пункт «отсутствие горизонтального скрола». Примеры, показывающие, что вариант 1 может оказаться очень плохим, приведены выше. Браузеры по умолчанию выбирают вариант 2,*а*, как самый безопасный и надежный.

В принципе вариант 1 может сработать нормально. Представьте сайт с маленькими иконками и преимущественно текстовым контентом. Можно не сжимать ни текст, ни картинки: картинки отображать в их истинном размере, а текст переносить по словам на новые строки (рис. 23).

Если вариант 1 такой хороший, то почему браузер по умолчанию выбирает вариант 2, *а*? Дело в том, что если бы верстальщик готовил сайт, одинаково хорошо отображающийся на любых экранах, то он (верстальщик) оставил бы браузеру соответствующие указания в виде метаданных в части head:

```
<meta name="viewport" content="width=device-width, initial-scale=1.0">
```

Если таких указаний нет, то значит (рассуждает браузер) сайт сверстан на широкий экран и надо масштабировать.

Визуальный вьюпорт и вьюпорт страницы

Для дальнейшего изложения нам потребуется ввести понятия визуального вьюпорта и вьюпорта страницы. Лучше всего базовую концепцию объясняет Джордж Камминс (George Cummins): «Представьте вьюпорт страницы как большое изображение с неизменными размерами и формой. Теперь представьте меньшее по размеру окошко, через которое вы смотрите на это большое изображение. Ваше окошко окружено непроницаемым материалом, который не дает вам увидеть большое изображение полностью, за исключением отдельных его частей. Часть большого изображения, которая видна через окошко, и есть визуальный вьюпорт. Вы можете отойти с окошком от большого изображения (уменьшение масштаба), чтобы увидеть сразу все изображение, или вы можете подойти поближе (увеличение масштаба), чтобы рассмотреть только часть изображения. Вы также можете менять ориентацию окошка (на пор-

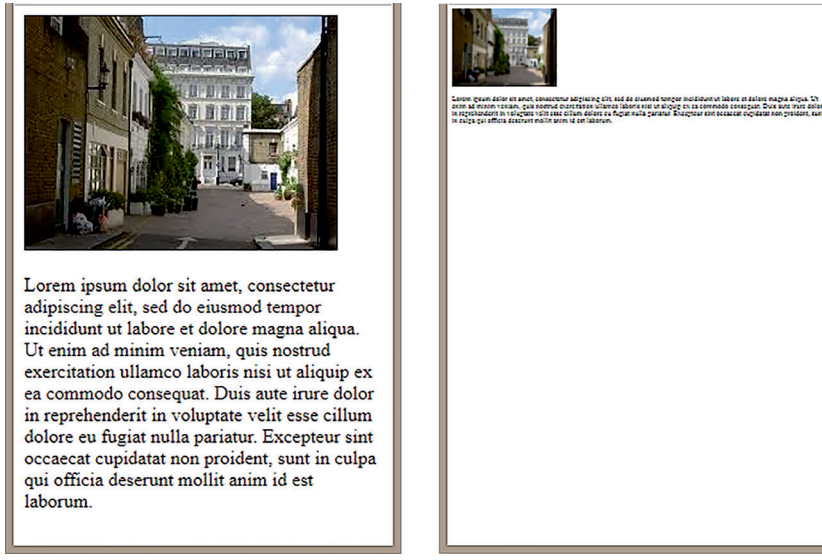


Рис. 23. Два способа отображения контента на узком экране телефона

третную или ландшафтную), но размер и форма большого изображения (вьюпорта страницы) всегда остается неизменной» [https://stackoverflow.com/questions/6333927/difference-between-visual-viewport-and-layout-viewport].

Рассмотрим пример (рис. 24). Здесь ширина вьюпорта страницы равна 980 px, ширина экрана в аппаратно-независимых пикселях — 480 px. На рис. 24, *а* показано, как эта страница выглядит при 100 % зумме на HTC Desire: лишь часть страницы влезла на экран, т. е. ширина визуального вьюпорта равна 480 px. На рис. 24, *б* видно, что страница вошла на 480 px за счет уменьшения масштаба, значит, ширина визуального вьюпорта равна ширине вьюпорта страницы, т. е. 980 px.

Визуальный вьюпорт — часть страницы, которая видна в данный момент на экране. Пользователь может применить прокрутку, чтобы изменить видимую часть страницы, или масштабирование, чтобы изменить размеры визуального вьюпорта.

а *б*

Рис. 24. Визуальный вьюпорт (*а*) и вьюпорт страницы (*б*)

Оба вьюпорта измеряются в CSS-пикселях, но во время масштабирования размеры визуального вьюпорта меняются (если вы увеличиваете масштаб, на экране умещается меньшее количество CSS-пикселей), в то время как размеры вьюпорта страницы остаются неизменными. Если бы это было не так, браузер постоянно совершал бы «перераскладку» (дословная калька с англ. reflow) страницы из-за пересчета процентной ширины, что является очень тяжеловесной операцией.

5.5. Метатег viewport

Теперь, когда все базовые понятия введены и проблемы, связанные с концепцией вьюпорта, разобраны, создадим алгоритм работы мобильного браузера.

Случай 1. Веб-сайт сверстан для широкоэкранных настольных мониторов, адаптивный дизайн не применялся и, разумеется, в заголовочной части отсутствует метатег viewport. Браузер должен отобразить веб-страницу на экране шириной в 320 px.

Раскладка элементов и вычисление ширин, заданных в процентах, ведется относительно вьюпорта страницы. Не встретив конструкцию <meta name="viewport" …>, браузер задает большую ширину вьюпорта страницы (Safari iPhone использует 980 px, Opera — 850 px, Android WebKit — 800 px, IE — 974 px). То есть CSS интерпретируется так, как если бы экран телефона был намного шире. Это гарантирует, что раскладка сайта будет вести себя так, как и в настольном браузере. Потом все эти правильно разложенные html-элементы масштабируются (уменьшаются в 2–3 раза) таким образом, чтобы целиком отобразить страницу[17] на экране. Для взаимодействия с ней пользователям приходится увеличивать масштаб. Результат виден на рис. 24, *а*.

Случай 2. Адаптивный веб-сайт верстался таким образом, чтобы он хорошо смотрелся на экранах телефонов. В заголовочной части содержится метатег:

[17] По ширине, а не по высоте.

```
<meta name="viewport" content="width=device-width, initial-scale=1.0">
```

Браузер задает ширину вьюпорта страницы, она равна ширине экрана (а не стандартной ширине в 980 px, как он может предполагать по умолчанию). О том, чтобы верстка не поплыла, позаботился веб-дизайнер! Результат виден на рис. 24, *а*.

Метатег viewport сообщает браузеру, как обрабатывать размеры страницы и изменять ее масштаб. Разберем его параметры.

Значением параметра width может быть либо ширина вьюпорта страницы в пикселях, либо специальная строковая константа «device-width»:

```
<meta name="viewport" content="width=320px">
<meta name="viewport" content="width=device-width">
```

Если установить значение width=device-width, ширина вьюпорта страницы выбирается в соответствии с размером экрана устройства. Это позволяет браузеру корректно осуществлять раскладку элементов для показа на разных экранах. У разных смартфонов ширина экрана может быть разной, поэтому самый оптимальный вариант — использовать device-width. Параметр initial-scale задает начальный масштаб.

В некоторых операционных системах, например в iOS и Windows Phone, ширина страницы при повороте в горизонтальный режим остается неизменной, и вместо перераспределения контента выполняется масштабирование. Атрибут initial-scale = 1 заставит браузер установить соответствие 1 : 1 для пикселей CSS и аппаратно-независимых пикселей без учета ориентации устройства. Это позволит использовать всю ширину экрана в горизонтальном состоянии. Остальные параметры метатега viewport перечислены в табл. 9.

Отметим, что использование метатега viewport не рекомендовано Консорциумом W3C, вместо него рекомендовано использовать директиву @viewport. Метатег viewport был предложен компанией Apple в iPhone, а после того как он приобрел популярность, был поддержан другими производителями браузеров. Так как метатег viewport предназначен исключительно для настройки разметки,

можно сказать, что он по праву относится к CSS. Именно потому W3C стремится стандартизовать новый метод адаптации, при котором управление окном переносится из HTML в CSS.

Вместо параметра initial-scale используется свойство, а вместо

Таблица 9

Параметры метатега viewport

Параметр	Значения	Функция
user-scalable	no или yes	Определяет, может ли пользователь изменять масштаб в окне
minimum-scale	Число (от 0.1 до 10). 1.0 — не масштабировать	Определяет минимальный масштаб viewport
maximum-scale	Число (от 0.1 до 10). 1.0 — не масштабировать	Определяет максимальный масштаб viewport

user-scalable — user-zoom, как показано в примерах:

```
@viewport {
    width: device-width;
}

@viewport {
    width: device-width;
    zoom: 2;
    user-zoom: fixed;
}
```

К сожалению, не все браузеры поддерживают стандарт, поэтому необходимо использовать вендорные префиксы. Например, для браузера Opera правило может выглядеть так:

```
@-o-viewport {
    width: device-width;
}
```

5.6. Адаптивный веб-дизайн

Адаптивный веб-дизайн — дизайн веб-страниц, обеспечивающий правильное отображение сайта на различных устройствах, подключенных к Интернету, и динамически подстраивающийся под заданные размеры окна браузера.

Основные принципы адаптивного веб-дизайна:

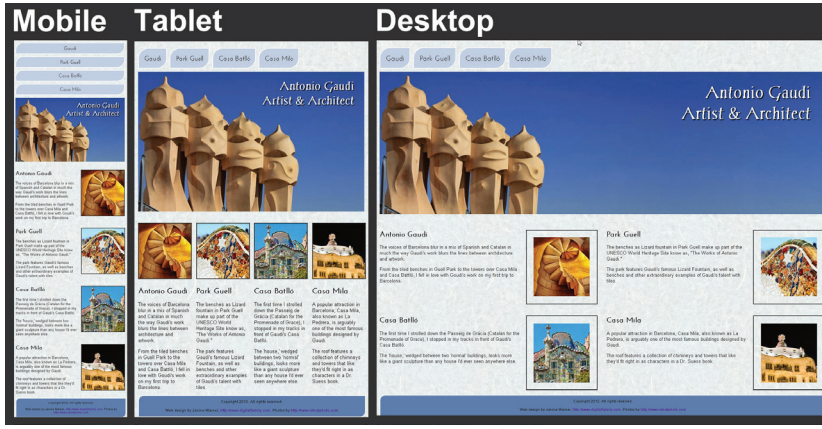
1) применение гибкого макета на основе сетки,

2) использование гибких изображений,

3) работа с медиазапросами.

На рис. 25 представлены три скриншота, показывающих, как мог бы выглядеть сайт на широком экране ноутбука (*а*), планшета (*б*) и телефона (*в*) при использовании адаптивного веб-дизайна. Основная идея адаптивного веб-дизайна состоит в том, что от размеров области просмотра зависит способ отображения: раскладка элементов, их размеры, видимость и т. д.

Для широкого экрана раскладка может быть следующей: под большой картинкой контент располагается в четыре колонки — две для текста и две для изображений. Располагать таким же образом контент на экране планшета не получится — тексту и картинкам



а *б* *в*

Рис. 25. Пример адаптивного веб-сайта. Отображение на экранах ноутбука (*а*), планшета (*б*) и телефона (*в*)

не хватит места. Раскладка перестраивается: в каждой колонке текст располагается не сбоку от рисунка, а под ним. Далее, на экране телефона все выстраивается в одну колонку, при этом меню становится вертикальным. Одновременно с этим сжимается главная картинка.

Возникает вопрос: как узнать ширину области просмотра в CSS-пикселях? Делается это с помощью медиазапросов, о которых рассказано ниже.

Медиазапросы — это модуль CSS3, который позволяет проводить рендеринг контента в зависимости от условий. Такими условиями могут быть ширина и высота экрана в аппаратно-независимых пикселях, ширина и высота области просмотра в CSS-пикселях, ориентация (портретная или ландшафтная), разрешение экрана, глубина цветовой палитры и т. д.

Введенная в CSS2 директива @media давала возможность определять различные стили для различных типов устройств. Медиазапросы в CSS3 расширили идею медиатипов[18] CSS2: вместо привязки к конкретному типу устройства (монитор, принтер, проектор и т. д.) стали исходить из возможностей этого устройства (разрешение, размеры и т. д.).

В несколько упрощенном виде медиазапросы имеют следующий вид:

```
@media [not|only] mediatype and (expressions) {
    CSS-Code;
}
```

После директивы @media указывается медиатип (название устройства), далее логические операторы и медиафункции, затем в фигурных скобках CSS-правила. Приведем примеры:

```
@media screen and (max-width: 480px) {
    body {font-sixe: 80%;}
}
```

[18] Обратите внимание на разницу названий: «медиатипы» и «медиазапросы».

Если страница отображается на экране и ширина области просмотра не более 480 px, то размер шрифта устанавливается 80 %.

Мы не будем подробно останавливаться на синтаксисе медиазапросов и перечислять все возможные медиафункции — эта информация есть в справочниках. Перейдем к разбору примеров адаптивного веб-дизайна.

На странице требуется разместить навигационное меню и основной контент (рис. 26). Если страница отображается на широком экране, то меню должно находиться слева от контента, если же на узком экране (например, на смартфоне), то меню должно располагаться сверху, над контентом.

```
<! DOCTYPE html>
<html>
<head>
<meta name="viewport" content="width=device-width,
    initial-scale=1.0">
<title>Адаптивная страница</title>
<style>
.wrapper {overflow: auto;}

#main {margin-left: 4px;}

#leftsidebar {
    float: none;
    width: auto;
}

#menulist {
    margin: 0;
    padding: 0;
}

.menuitem {
    background: #CDF0F6;
    border: 1px solid #d4d4d4;
```

```
        border-radius: 4px;
        list-style-type: none;
        margin: 4px;
        padding: 2px;
}

@media screen and (min-width: 480px) {
        #leftsidebar {width: 200px; float: left;}
        #main {margin-left: 216px;}
}
</style>
</head>
<body>

<div class="wrapper">
        <div id="leftsidebar">
                <ul id="menulist">
                        <li class="menuitem"> Элемент 1</li>
                        <li class="menuitem"> Элемент 2</li>
                        <li class="menuitem"> Элемент 3</li>
                        <li class="menuitem"> Элемент 4</li>
                        <li class="menuitem"> Элемент 5</li>
                </ul>
        </div>
        <div id="main">
                <h1>Измените размеры окна браузера!</h1>
                <p>Если ширина вьюпорта не меньше 480 пикселей, то меню
прижато к левому краю. Если ширина вьюпорта меньше 480 пикселей, то меню
расположено сверху, над контентом.</p>
        </div>
</div>

</body>
</html>
```

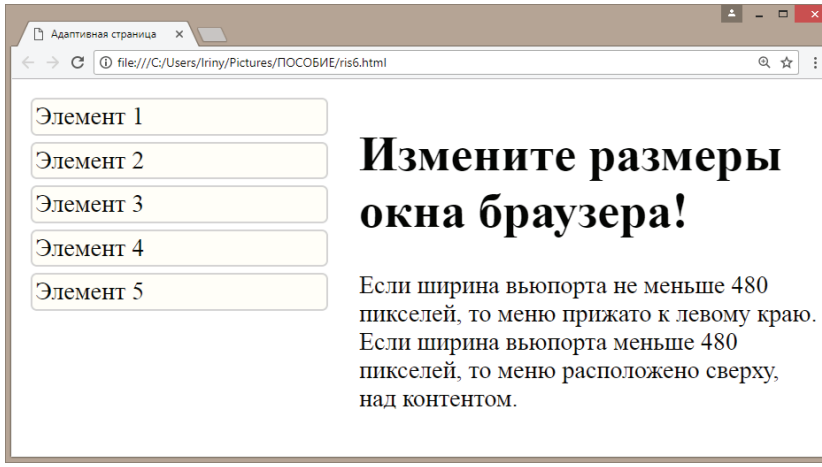


Рис. 26. Пример отображения адаптивной страницы на экране. Меню расположено слева от контента

Во-первых, разберем верстку. Можно выделить два блока div: боковая панель задана блоком leftsidebar, содержащим список, а основной контент задан блоком main, где расположен заголовок и абзац.

Во-вторых, разберем стили. В строках с 7 по 28 описаны стили для узкого экрана, а далее, после директивы @media, описаны стили для экрана шириной не менее 480 px.

На широком экране меню плавающее, оно прижато к левому краю вмещающего блока wrapper и имеет ширину 200 px:

```
#leftsidebar {width: 200px; float: left;}
```

Поскольку плавающий элемент вырван из потока и не влияет на раскладку следующих ниже (по тексту исходных кодов) элементов, то основной контент начинается от верхнего края вмещающего блока wrapper. Чтобы основной контент не накрывал меню, сделан большой правый отступ, размер которого складывается из ширины меню (200 px) и визуального отступа (еще 16 px):

```
#main {margin-left: 216px;}
```

На узком экране меню перестает быть плавающим, занимает всю доступную ширину и, как следствие, располагается над основным контентом, т. е. блоки раскладываются на странице вертикально:

```
#leftsidebar {
    float: none;
    width: 100px;
}
```

Выше среди основных принципов адаптивного веб-дизайна был указан принцип использования гибких изображений. Разберем это на примерах.

Пример 1. Сжатие картинок.

Рассмотрим модельную веб-страницу с большим изображением. Если пользователь начнет сжимать окно браузера, то настанет момент, когда картинка станет шире окна, появится горизонтальная полоса прокрутки. Для решения проблемы можно было бы отобразить картинку в уменьшенном виде, т. е. смасштабировать.

1. У body заданы нулевые отступы, следовательно, ширина body совпадает с шириной области просмотра.
2. Стили блока wrapper:
   а) у блока wrapper задана ширина 96 % от ширины вмещающего блока, которым является body;
   б) для центрирования блока wrapper устанавливаются отступы margin: auto;
   в) поскольку на очень широких экранах текст читается тяжело (взгляду трудно переходить с конца текущей строки на начало следующей), необходимо ограничить ширину блока wrapper. С этой целью устанавливается максимально допустимая ширина max-width: 960 px.
3. Стили изображения:
   а) согласно таблице стилей, рекомендованной Консорциумом W3C, картинки являются строчными элементами. Мы определяем, что картинка является блочной, а значит, для нее будет доступна блочная модель форматирования;

б) для центрирования картинки внутри блока wrapper ей задаются отступы auto;

в) чтобы во время уменьшения размеров окна браузера картинка не оказалась шире своего родителя, задается максимально допустимая ширина картинки, max-width: 100 %.

Используя свойство max-width и задавая ширину в процентах, можно добиться того, что размеры картинки будут адаптироваться к размерам области просмотра.

```
<! DOCTYPE html>
<html>
<head>
<meta name="viewport" content="width=device-width,
    initial-scale=1.0">
<title>Адаптивная картинка 1</title>
<style>
body{margin:0;}
.wrapper{
    width: 96%;
    max-width: 960px;
    padding: 2%;
    margin: auto;
    background: #ddd;
}
img{
    display: block;
    margin: auto;
    max-width: 100%;
}
</style>
</head>
<body>
<div class='wrapper'>
<img src='bear.png' alt='bear'>
<p>Картинка масштабируется в соответствии с размерами окна браузера</p>
</div>
```

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

Пример 2. Работа с плавающей картинкой.

Рассмотрим модельную веб-страницу. Здесь расположено плавающее изображение, текст обтекает картинку по правому краю. Картинка занимает 40 % от ширины родителя — блока wrapper. Если пользователь начнет сжимать окно браузера, то настанет момент, когда картинка станет очень маленькой и плохо различимой. Для решения проблемы можно, начиная с определенного размера окна, запретить уменьшать картинку:

```
min-width: 300px;
```

Это, однако, породит другую проблему: текст справа от картинки станет «рваным», возникнут некрасивые переносы слов, а справа от картинки появится пустое место.

Для решения этой новой проблемы воспользуемся медиазапросами. На экране шириной не более 460 px картинка должна занять всю доступную ширину блока родителя (но не более 100 %).

```
<! DOCTYPE html>
<html>
<head>
<meta name="viewport" content="width=device-width,
    initial-scale=1.0">
<title>Адаптивная картинка 2</title>
<style>
body{margin:0;}
.wrapper{
    width: 96%;
    max-width: 960px;
    padding: 2%;
    margin: auto;
    background: #ddd;
}
```

```
img{
    float: left;
    width: 40%;
    min-width: 300px;
    margin: 0 20px 10px 0;
}
@media screen and (max-width: 460px){
    img{
        width: auto;
        max-width: 100%;
        height: auto;
        display: block;
        margin: auto;
    }
}
</style>
</head>
<body>
<div class='wrapper'>
<p>При уменьшении размеров окна браузера картинка сначала уменьшается, занимая 40% ширины окна. Затем "замораживается" и занимает 300px. Затем условие медиа запроса становится истинным, и картинка превращается в блочный элемент, занимающий всю ширину окна.</p>
<img src='bear.png' alt='bear'>
</div>
</body>
</html >
```

Пример 3. Раскладка картинок в ряд.

Задача состоит в том, чтобы расположить в ряд по горизонтали две или три картинки. Картинки должны быть адаптивными, т. е. менять свою ширину в зависимости от ширины экрана. Это в дальнейшем понадобится для того, чтобы можно было по 4 картинки в ряд раскладывать на широких экранах ноутбуков, по две — на планшетах и только по одной (иными словами, выстраивать картинки вертикально) на телефонах.

Разберем стили. Для класса layout определено, что строчный контент выравнивается по центру:

```
text-align: center;
```

Картинки определяются как строчно-блочные элементы (display: inline-block), а значит, принимают участие в строчном контексте форматирования своего родителя. Ширина картинок задана в процентах от ширины родителя, что делает картинки адаптивными (иногда их называют «резиновыми»). Ширина выбрана такой, чтобы для картинок, стоящих в одной строке, сумма ширин была менее 100 %; это обеспечивает отступы между картинками. Например, 48 % + 48 % = 96 %.

```
<! DOCTYPE html>
<html>
<head>
<meta name="viewport" content="width=device-width,
    initial-scale=1.0">
<title>Адаптивные картинки 1</title>
<style>
.layout{
    width: 100%;
    text-align: center;
    margin: auto;
    background: #ddd;
    border: 1px solid red
}
.two-col{
    display: inline-block;
    width: 48%;
    height: auto;
}
.three-col{
    display: inline-block;
    width: 32%;
```

```
        height: auto;
}
</style>
</head>
<body>
<p>Сказка про двух медведей</p>
<div class='layout'>
        <img class='two-col' src=' brown.png'>
        <img class='two-col' src=' polar.png'>
</div>
<p>Сказка про трех медведей</p>
<div class='layout'>
        <img class='three-col' src='brown.png'>
        <img class='three-col' src=' polar.png'>
        <img class='three-col' src='coala.png'>
</div>
</body>
</html>
```

Пример 4. Раскладка картинок в ряд.

Задача состоит в том, чтобы на широких экранах ноутбуков разложить 4 картинки в ряд по горизонтали, на более узких экранах планшетов — по две картинки в ряд, а на совсем узких экранах телефонов — выстроить картинки вертикально. Картинки должны быть адаптивными, т. е. менять свою ширину в зависимости от ширины экрана.

Изначально картинкам задана ширина 24 % от ширины вмещающего блока. Далее, когда ширина области просмотра станет не более 768 px, ширина картинок будет изменена на 48 %; соответственно, только две картинки разместятся в строке. Наконец, когда ширина области просмотра стане не более 460 px, ширина картинок будет изменена на 100 %; таким образом, картинки выстроятся вертикально.

```
<! DOCTYPE html>
<html>
<head>
```

```
<meta name="viewport" content="width=device-width,
    initial-scale=1.0">
<title>Адаптивные картинки 2</title>
<style>
.four-col{
    display: inline-block;
    width: 24%;
    height: auto;
}
@media screen and (max-width: 768px){
    .four-col{width: 48%;}
}
@media screen and (max-width: 460px){
    .four-col{width: 100%;}
}
</style>
</head>
<body>
<p>Сказка медведей</p>
<img class='four-col' src='brown.png'>
<img class='four-col' src='polar.png'>
<img class='four-col' src='grizzly.png'>
<img class='four-col' src='coala.png'>
</body>
</html>
```

5.7. Некоторые преимущества, которые дает адаптивный веб-дизайн

Используя методы адаптивного веб-дизайна, можно не разрабатывать несколько версий сайта и не размещать их на разных URL-адресах типа site.com и site_mobile.com. Вместо этого можно разработать лишь одну версию дизайна, которая будет автоматиче-

ски адаптироваться под размер экрана устройства. У такого подхода есть сразу несколько преимуществ.

Благодаря адаптивному дизайну упрощается поисковая оптимизация сайта (англ. Search Engine Optimization), поскольку у всех версий сайта (мобильной и настольной) будет один и тот же URL. Не нужно будет беспокоиться о том, что одни ссылки будут вести на мобильную версию сайта, а другие — на широкоэкранную.

В отчетах Google Analytics показатели сайта будут выше, поскольку результаты запросов с мобильных телефонов и с ПК будут синхронизированы. То же самое касается статистики распространения в соцсетях (лайки в Фейсбуке, твиты и т. п.), так как у мобильной и настольной версий будет одинаковый URL.

Контрольные вопросы

1. В примере 4, разобранном в главе 5, ширина для класса .four-col определялась (точнее, переопределялась) в нескольких правилах. Влияет ли порядок описания этих правил на то, как браузер будет вычислять ширину картинок? Если в исходном тексте поменять местами директивы

```
@media screen and (max-width: 768px){…}
@media screen and (max-width: 460px){…},
```

как это повлияет на отображение страницы?

2. Некоторые авторы различают понятия «адаптивный дизайн» и «отзывчивый дизайн». В чем отличие этих понятий?

3. Изображения, тег img, являются одним из самых сложных элементов адаптивного веб-дизайна. В HTML 5 введен элемент picture, который дает возможность устанавливать разные изображения в зависимости от условий (например, ширина и ориентация экрана). Изучите возможности элемента picture.

4. Не мешает ли адаптивный дизайн отображению рекламных блоков, например, в Google AdSense?

СПИСОК РЕКОМЕНДУЕМОЙ ЛИТЕРАТУРЫ

Веб-технологии для разработчиков: CSS (англ. Web technology for developers: CSS) [Электронный ресурс]. URL: https://developer.mozilla.org/en-US/docs/Web/CSS (дата обращения: 27.09.2018).

Каскадные таблицы стилей, версия 2, редакция 1 (CSS2.1) : стандарт (англ. Cascading Style Sheets Level 2 Revision 1 (CSS2.1): Specification) // w3.org : [сайт]. URL: https://www.w3.org/TR/2011/REC–CSS2–20110607/ (дата обращения: 27.09.2018).

Медиазапросы : стандарт (англ. Media Queries) // w3.org : [сайт]. URL: https://www.w3.org/TR/css3-mediaqueries/ (дата обращения: 27.09.2018).

Фрейн Б. HTML5 и CSS3: Разработка сайтов для любых браузеров и устройств / Б. Фрейн. СПб. : Питер, 2014. 304 с.

Meyer E. CSS: The Definitive Guide / E. Meyer, E. Weyl. Sebastopol : O'Reilly Media, 2017. 1123 p.

Учебное издание

Солодушкин Святослав Игоревич
Юманова Ирина Фарисовна

Web и DHTML

Учебное пособие

Заведующий редакцией	М. А. Овечкина
Редактор	Н. В. Чапаева
Корректор	Н. В. Чапаева
Компьютерная верстка	В. К. Матвеев

Подписано в печать 19.06.2018 г. Формат 60×841/16.
Бумага офсетная. Цифровая печать. Усл. печ. л. 7,44.
Уч.-изд. л. 5,8. Тираж 50 экз. Заказ 163.

Издательство Уральского университета
Редакционно-издательский отдел ИПЦ УрФУ
620083, Екатеринбург, ул. Тургенева, 4
Тел.: +7 (343) 389-94-79, 350-43-28
E-mail: rio.marina.ovechkina@mail.ru

Отпечатано в Издательско-полиграфическом центре УрФУ
620083, Екатеринбург, ул. Тургенева, 4
Тел.: +7 (343) 358-93-06, 350-58-20, 350-90-13
Факс: +7 (343) 358-93-06
http://print.urfu.ru

9 785799 624101

СОЛОДУШКИН СВЯТОСЛАВ ИГОРЕВИЧ

Кандидат физико-математических наук, доцент кафедры вычислительной математики и компьютерных наук Уральского федерального университета. Окончил математико-механический факультет Уральского государственного университета по специальности «Математика и компьютерные науки» (2004). Читает лекции по курсам «Протоколы Интернета», «Web и DHTML», «Web-программирование на PHP», «Скрипты», «Параллельные численные методы». Автор более 60 научных публикаций. Область научных интересов — программирование для Интернета, веб-разработка.

ЮМАНОВА ИРИНА ФАРИСОВНА

Кандидат физико-математических наук, ассистент кафедры вычислительной математики и компьютерных наук Уральского федерального университета Окончила факультет прикладной математики Ижевского государственного технического университета по специальности «Прикладная математика» (2010). Читает лекции по курсам «Протоколы Интернета», «Web и DHTML», «Скрипты». Автор более 40 научных публикаций. Область научных интересов — математическое моделирование, численные методы и комплексы программ.