

Максим Усачёв
Илья Стрельцын

Инлайновый контекст форматирования

Цикл «Тайны CSS 2.1»



Содержание

Введение.....	3
1. Введение в инлайновый контекст форматирования (ИКФ): основные понятия (1-я публикация цикла "Тайны CSS2.1").....	6
2. Введение в инлайновый контекст форматирования (ИКФ): основные понятия (2-я публикация цикла "Тайны CSS2.1").....	11
3. ИКФ: высота строки, часть 1 (3-я публикация цикла "Тайны CSS2.1").....	17
4. ИКФ: высота строки, часть 2 (4-я публикация цикла "Тайны CSS2.1").....	21
5. ИКФ: высота строки, часть 3 (5-я публикация цикла "Тайны CSS2.1").....	25
6. ИКФ: высота строки, часть 4 (6-я публикация цикла "Тайны CSS2.1").....	30
7. ИКФ: высота строки, часть 5 (7-я публикация цикла "Тайны CSS2.1").....	35
8. ИКФ: Вертикальное выравнивание в строке, часть 1 (8-я публикация цикла "Тайны CSS2.1").....	44
9. ИКФ: Вертикальное выравнивание в строке, часть 2 (9-я публикация цикла "Тайны CSS2.1").....	49
10. ИКФ: Вертикальное выравнивание в строке, часть 3 (10-я публикация цикла "Тайны CSS2.1").....	58
11. ИКФ: Горизонтальное выравнивание, часть 1 (11-я публикация цикла "Тайны CSS2.1").....	61
12. ИКФ: Горизонтальное выравнивание, часть 2 (12-я публикация цикла "Тайны CSS2.1").....	61
13. ИКФ: Горизонтальное выравнивание, часть 3 (13-я публикация цикла "Тайны CSS2.1").....	76

Введение

Эта книга, написанная создателями css-live.ru (Максимом Усачёвым aka psywalker и Ильёй Стрельцыным aka SelenIT), посвящена одной из самых важных и вместе с тем самой загадочной, неизученной, неоднозначной и полной неочевидных сюрпризов стороне действующей спецификации CSS — визуальному форматированию текста.

В этой книге вы не увидите рассказов о том, как сверстать тот или иной блок или как сделать красивую тень. Напротив, в них подробно описывается сам процесс работы строчного форматирования в CSS, построения строк, инлайн-элементов, механизм влияния на высоту или ширину строки и т.п.

Эта книга будет полезна прежде всего тем, кому важна не только практическая часть CSS, но и теоретическая составляющая, а также тем, кому мало лишь поверхностного знания и есть желание познать таинственные стороны, которые скрываются за ширмой CSS.

Наш Веб-сайт

На нашем сайте по адресу <http://css-live.ru> вы сможете найти очень много полезного материала, статей и новостей из мира HTML5 и CSS2.1, 3.

Контактная информация

Если вдруг вы обнаружили какие-то ошибки или неточности в книге, или же вам просто хочется поделиться своим материалом и идеями, то пишите нам на почту psywalker@css-live.ru.

Благодарности

Спасибо Александру ([s0rr0w](#)) Шпаку и Рашиду Берёзкину ([Great Rash](#)) за то время, которое они потратили на нас, консультируя в вопросах CSS.

Отдельное спасибо Павлу Тринько ([Polly.Glot](#)), который несмотря на свой молодой возраст, проявил невероятные старания, помогая нам во многих вещах.

Также нельзя не поблагодарить Александра Старцева ([hypnocolor](#)) за дизайн обложки для этой книги, и сказать ему отдельное спасибо за очень забавного хорька :)

Ну и, конечно же, хочется выразить глубокую признательность Екатерине Карнюшиной ([cath_kb](#)) за героический труд, который она проявила, редактируя всю ту писанину, которую наши безумные мозги посмели выплеснуть в этой книге.

Об авторах

Максим Усачёв

Максим Усачёв ([aka psywalker](#)) является автором сайта [css-live.ru](#) и администратором одного из самых известных форумов рунета по веб-технологиям [forum.htmlbook.ru](#). Несколько лет назад влюбился и женился на CSS и теперь не может без него жить. Интересуется всем, что касается мира HTML, CSS, JavaScript и сопутствующих им технологий.

Илья Стрельцын

Трудно найти человека, который был бы похож на Илью Стрельцына ([aka SelenIT](#)). Этот талантливый безумец умудрился совместить в себе искусного теоретика и умелого мастера своего дела. Являясь экспертом по веб-технологиям и соавтором [css-live.ru](#), Илья придумал много полезных решений, которыми сейчас пользуются верстальщики, а также написал кучу интересных и познавательных статей. Генератор идей и мозг, без которого эта книга никогда бы не появилась на свет.

Авторские права

Цикл статей “Инлайновый контекст форматирования (ИКФ)” (далее произведение) распространяется на условиях следующей лицензии Creative Commons:
«Указание авторства — Некоммерческое использование — Без производных» (Attribution-NonCommercial-NoDerivs 2.5 Generic (CC BY-NC-ND 2.5)).

Вы можете свободно



— копировать, распространять и передавать данное произведение.

На следующих условиях



— обязательно указывать авторов произведения (Илью Стрельцына и Максима Усачёва) и ссылку на сайт авторов (css-live.ru).



— нельзя использовать произведение для заработка.



— запрещается изменять произведение или создавать другие с его помощью.

Введение в инлайновый контекст форматирования (ИКФ): основные понятия (1-я публикация цикла “Тайны CSS2.1”)

Добавлено 31 Май 2012 | Раздел: [CSS](#), [Статьи](#), [Тайны CSS2.1](#)

В основе этой статьи лежит перепевка спецификации CSS 2.1 по телефону Рабиновичем SelenIT-ом. Если что-то вышло криво — я не виноват, это всё он. Конструктивная критика и предложения по улучшению материала приветствуются!

Перед тем как погрузиться в мир инлайнового контекста, мне казалось, что это не очень сложная тема, и что понять её будет не трудно. Но, чем больше я углублялся в процесс изучения, тем больше понимал, что это далеко не простая, а, напротив, даже очень обширная и сложная тема для понимания.

Ещё больше я боялся за то, что не смогу передать на бумаге все тонкости и особенности построения этого механизма нашим читателям. Поэтому статьи цикла я буду стараться писать так, чтобы все их детали были написаны на доступном и понятном языке.

Ну, как говорится — с Богом!

Лайн-бокс

Зачем вообще нужен лайн-бокс?

Говоря простым языком, **лайн-бокс** - это удобный виртуальный контейнер, от которого «пляшут» элементы строки (**инлайн-боксы**). Дело в том, что при построении строки рендерер (механизм отрисовки страниц в браузерах и построения DOM-дерева) должен как-то узнать, какую Y-координату дать следующей порции инлайнового контента (тексту, картинкам и т.п.). Можно сказать, что **лайн-бокс** - это способ учёта пространства по вертикали, занимаемого инлайном.

Построение лайн-бокса

В инлайновом контексте форматирования **инлайн-боксы** выкладываются друг за другом горизонтально, начиная от верха контейнера и заполняются слева направо (или справа налево у евреев и арабов). Горизонтальные **margin**, **border** и **padding** между инлайн-боксами учитываются. **Инлайн-боксы** могут выравниваться по вертикали по-разному: по верхней либо нижней границе, или по базовым линиям текста в них. Прямоугольная область, содержащая **инлайн-боксы** из одной строки, называется **лайн-боксом**.

Так, стоп. Понимаю, что многие из вас уже запутались и потеряли нить. Поэтому теперь предлагаю чуть помедленнее, с кодом и картинками.

Вначале рассмотрим простой пример. Код будет таким:

Пример 1.1 Одна строка (лайн-бокс)

HTML

```
<p> один <em> два </em> три </p>
```

Вроде бы всё просто, обычный **блочный элемент** (**<p>**), содержащий в себе одну строку, в которой есть текст и **инлайн-элемент** (****). Но на самом деле в подноготной самой строки произошёл следующий эффект:

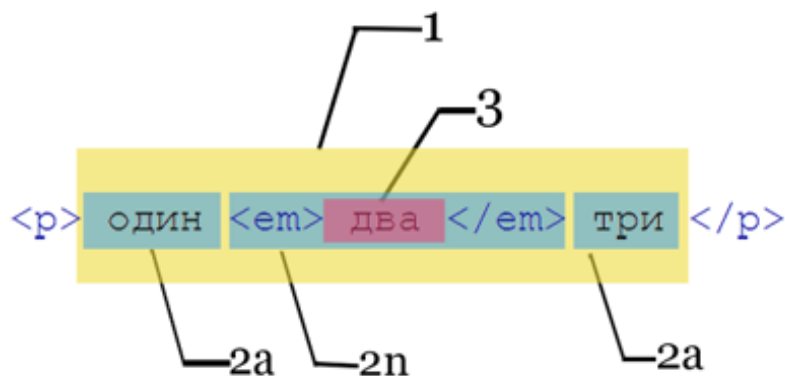


Рис.1.1 Одна строка (лайн-бокс)

На картинке мы можем видеть параграф, генерирующий блочный бокс, в котором и должны находиться **лайн-боксы**. Этот блочный элемент содержит в себе **один лайн-бокс (1)**, **три инлайн-бокса (2a, 2n, 2a)**, в одном из которых (2n) находится просто текст. А теперь разберём более детально.

Под цифрой "1" изображён **лайн-бокс**, сгенерированный сразу же после генерации блочного бокса (параграфа). Так как у нас **одна строка**, то **лайн-бокс** соответственно **тоже один** (о двух-строчных чуть позже).

Под цифрами **2a, 2n, 2a** находятся **инлайн-боксы**. На рисунке их **ровно три**. Они очень похожи друг на друга, но, несмотря на это, между некоторыми из них есть одно важное отличие. Два боковых **инлайн-бокса (2a)** на самом деле являются **анонимными**, в отличие от среднего (2n), "**натурального**". Что это значит?

Анонимный инлайн-бокс

Анонимными инлайн-боксами называются те боксы, которые находятся **непосредственно в самом лайн-боксе**. Те, которые являются его **ближайшими текстовыми потомками**, не обернутыми ни в один инлайн-элемент.

Для чего это нужно? Давайте представим себе ситуацию. Есть такой html:

Пример 1.2 Лайн-бокс с анонимным инлайн-боксом

а)

HTML

```
<p> red <em>green <span>blue</span></em></p>
```

б)

CSS

```
p { color: red }
  em {color: green }
  span { color: blue }
```

В этом примере есть три инлайн-бокса разного цвета. Со вторым и третьим понятно, а что "красит" первый? Лайн-бокс взять на себя работу по его "покраске" не может, он нужен для других вещей, которые я описывал выше. Лайн-бокс — не "физический объект", а скорее геометрическая условность, он является всего лишь "счетчиком", передавая главному "штабу" информацию о своих координатах. В связи с этим нужен тот, кто сможет взять на себя стили, доставшиеся по наследству от блочного родителя, и "распорядиться этим наследством" как следует. И этим "кем-то" является анонимный инлайн-бокс, к которому и применяется CSS.

* Да, стоит уточнить, что в **строчном контексте форматирования CSS применяется именно к инлайн-боксам**, будь то они анонимные или "натуральные". Ну, и конечно же то, что в самих инлайн-боксах не бывает **анонимных инлайн-бокс**ов, они существуют только на верхнем уровне, как я и говорил выше. В самих **инлайн-боксах** текст является просто текстом.

"Натуральный" инлайн-бокс (т.е. не анонимный)

"Натуральным" инлайн-боксом можно назвать тот бокс, который генерируется самим **инлайн-элементом**, в нашем случае это элемент ****. Так же существует ещё ряд "натуральных" инлайн-боксов, таких как картинки, инлайн-блоки и т.д. В общем, все "живые" инлайн-сущности, которые мы можем видеть в структуре, превращаются в **инлайн-боксы**.

Ну, и чтобы совсем уж было понятно, я попытаюсь изобразить дерево данной структуры.

- * Block (p)
 - * line box
 - * anonymous inline box (текст: "один")
 - * inline box (em)
 - * anonymous inline box (текст: "три")

Это был самый лёгкий пример, но, пожалуй, рассмотрим более сложную ситуацию, где структура будет такой:

Пример 1.3 Две строки (два лайн-бокса)

HTML

```
<p> один <em> два <i> три </i> четыре </em> пять  
шесть <span> семь <i> восемь </i> девять </span></p>
```

В 99% случаев текст в документе состоит из двух и более строк. Что же происходит в таком случае с лайн-боксом? Нет, он не разрывается, вместо этого у нас появляется **уже два лайн-бокса**! Да, именно так. В этой ситуации рендерер заканчивает с первой строкой и начинает рендерить новую. Причём рендерер производит абсолютно такую же операцию над следующей строкой, как и с предыдущей, т.е. начинает выкладывать в ней новые кирпичи один за другим, заново.

Структура в таком случае выглядит так:

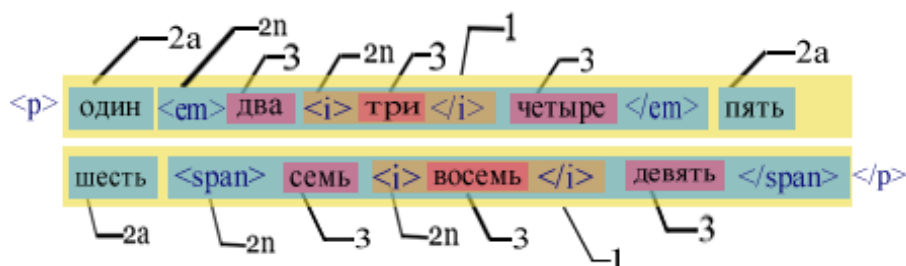


Рис. 1.2 Две строки (два лайн-бокса)

На рисунке я постарался отметить каждую детальку, обозначив их цифрами. Как можно увидеть, строк стало две, а следовательно, и лайн-боксов на рисунке образовалось ровно два! Но всё-таки, давайте по порядку.

Как и на предыдущем рисунке, под цифрами "1" находятся лайн-боксы, на изображении они отмечены **жёлтым** цветом. Самое интересное, что, несмотря на то, что контейнер у нас один (**<p>**), лайн-боксы в нем абсолютно независимы друг от друга. Когда закончилось построение первого, то на "поле" сразу же появился второй **лайн-бокс с новыми координатами по оси Y**. Первый закончил своё построение по той причине, что боксы внутри него попросту заняли всё доступное в нём пространство, а последнему не хватило в лайн-боксе места, и он вынужден был переместиться на новую строку. А раз **новая строка** – значит **новый лайн-бокс**. Это железное правило, и о нем нужно помнить!

Далее идут уже знакомые вам цифры: "2a" и "2n" обозначают анонимные инлайн-боксы (2a) и соответственно "натуральные" (2n). Их отличие я уже объяснил в первом примере, поэтому единственное, что хочу тут отметить, это то, что теперь у нас в структуре появились **составные инлайн-боксы**.

Составной инлайн-бокс

Если в инлайн-боксе находится от двух и более инлайн-боксов или, например, текст и инлайн-боксы, то такой инлайн-бокс считается **составным**, потому что **состоит из нескольких частей**. В нашем случае мы имеем **два составных инлайн-бокса**, один из них — элемент **** в первом лайн-боксе, а другой — **** во втором. Например, **** состоит из трёх частей: текст ("два"), инлайн-бокс (**<i>**) и за ним снова текст ("четыре").

Ну, и последнее, на чём стоит заострить наше внимание, это цифры "3", которые указывают на обычный текст в инлайн-боксах. Стоит заметить, что инлайн-боксом для такого текста является их контейнер, а сами они ничего не генерируют, в отличие, например, от текста в самих лайн-боксах (2a). Дело в том, что внутри инлайн-боксов текст является просто текстом, ему нет смысла генерировать лишние инлайн-боксы, так как CSS стили ему может передать его собственный контейнер.

Дерево такой структуры будет выглядеть так:

- * Block (p)
 - * line box
 - * anonymous inline box (текст: "один")
 - * inline box (em)
 - * текст: "два"
 - * inline box (i)
 - * текст: "четыре"
 - * anonymous inline box (текст: "пять")
 - * line box
 - * anonymous inline box (текст: "шесть")
 - * inline box (span)
 - * текст: "семь"
 - * inline box (i)
 - * текст: "девять"

Кстати, обратите внимание на один интересный момент. А именно, на анонимные инлайн-боксы в конце первого лайн-бокса и в начале второго. Оба они являются анонимными, потому что их родителями, по факту, являются лайн-боксы. А если бы, например структура выглядела бы так:

Пример 1.4 Составной инлайн-бокс

HTML

```
<p> один <em> два <i> три </i> четыре </em><i> пять  
шесть </i><span> семь <i> восемь </i> девять </span></p>
```

То дерево было бы уже таковым:

- * Block (p)
 - * line box
 - * anonymous inline box (текст: "один")
 - * inline box (em)
 - * текст: "два"
 - * inline box (i)
 - * текст: "четыре"
 - * inline box (i)
 - * line box
 - * inline box (i)
 - * inline box (span)
 - * текст: "семь"
 - * inline box (i)
 - * текст: "девять"

В этом примере элемент `<i>` как бы разрывается на две части и образует два инлайн-бокса — в двух строках. Часть содержимого элемента `<i>`, которой хватило места в первом лайн-боксе, образовала первый инлайн-бокс, а сам факт переноса привел к появлению нового лайн-бокса (новой строки), в начале которой разместился второй инлайн-бокс с остатком содержимого элемента.

Пробел - как анонимный инлайн-бокс или текст

Ещё один важный момент в этой части рассказа, который стоило бы разобрать, это пробелы между инлайн-боксами, которые могут превратиться в анонимные инлайн-боксы или в обычный текст. Чтобы было понятно о чём идёт речь, сразу же приведу наглядный пример.

Пример 1.5 Пробелы между инлайн-боксами

HTML

```
<p> один <em> два <i> три </i> <i> четыре </i> пять </em> <span> шесть </span></p>
```

Ну и, конечно же, изображение:

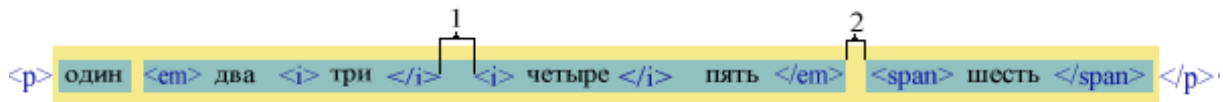


Рис 1.3 Пробелы между инлайн-боксами

Я не стал расписывать эту структуру по всем частям, здесь суть не в этом. Я лишь прошу вас обратить внимание на два выделенных места, которые помечены цифрами "1" и "2". Дело в том, что это один из важных моментов при строчном форматировании, о котором следует знать. Суть в самих пробелах. Пробелах между **инлайн-элементами** или попросту "**натуральными**" **инлайн-боксами**. Выяснилось, что эти, как мне до этого казалось, обычные "сдвиги координат", могут генерировать **анонимные инлайн-боксы**, точнее не все из них, а только пробелы верхнего уровня, находящиеся непосредственно в самом лайн-боксе. В нашем случае такой пробел (сгенерированный анонимный инлайн-бокс) находится между элементами `` и `` и обозначается цифрой "2".

А вот пробелы, которые находятся между инлайн-элементами в самих инлайн-боксах, тоже не пропадают зря, а становятся **обычным текстом**, как и любой другой текст в инлайн-боксах. Такой пробел вы можете наблюдать под цифрой "1" в нашем примере.

Я полагаю, что это логично, так как пробел всё же тоже является символом, а значит, так же, как и другие символы, имеет право на собственный контейнер, не так ли?

Введение в инлайновый контекст форматирования (ИКФ): основные понятия (2-я публикация цикла “Тайны CSS2.1”)

Добавлено 07 Июнь 2012 | Раздел: [CSS](#), [Статьи](#), [Тайны CSS2.1](#)

Поля, отступы и границы в инлайновом форматировании

В блочном контексте форматирования **поля** (**margin**), **отступы** (**padding**) и **границы** (**border**) могут оказывать влияние на **любые координаты** и **расположение элемента**, и так же могут влиять на **окружающий контекст**. Но что происходит в этом плане в инлайновом контексте? Давайте разберёмся.

Спецификация сообщает нам, что в **инлайн-форматировании** горизонтальные **margin**, **border** и **padding** между инлайн-блоками учитываются, а вот **вертикальные не дают никакого эффекта** и не могут влиять на окружающий контекст. Да, сразу стоит уточнить, что речь идет о инлайн-блоках, создаваемых обычными элементами с **display:inline** (читайте [первую часть цикла](#)), а не об инлайн-блоках или замещаемых элементах вроде картинок.

Некоторым может показаться, что здесь всё прозрачно, но на самом деле это не так. Поэтому я решил, что нам следует подробнее разобрать эти моменты, и начнём мы, пожалуй, с **margin**.

Горизонтальные поля

При назначении обычным инлайнам свойства **margin**, последний будет представлять из себя сдвиг от левого края, либо другого инлайн-блока, если это **margin-left**, и отодвигание правого инлайн-блока (если таковой имеется) при **margin-right**. Плюс **правый инлайн-блок** (опять же, если таковой имеется) может перенестись на следующую строку в случае нехватки места в текущей.

В общем, начнём с простого примера:

Пример 2.1 Горизонтальные поля

а)

HTML

```
<p>Lorem Ipsum is simply <span>dummy text printing </span> and type setting industry.
Lorem Ipsum has been the industry's standard dummy text ever since the 1500s
PageMaker including versions of Lorem Ipsum.</p>
```

б)

CSS

```
p {font-size: 16px;}
  span {
    background: #FC3;
    margin: 500px 180px 500px 40px;
    border-radius: 5px;
  }
```

И вот что из этого вышло:

Lorem Ipsum is simply **dummy text printing**
 and type setting industry. Lorem Ipsum has been the industry's standard dummy
 text ever since the 1500s PageMaker including versions of Lorem Ipsum.

Рис 2.1 Горизонтальные поля

Итак, я выставил **инлайн-элементу** `` левое поле со значением в **40px**, а правое - в **180px**. Что произошло? А произошло следующее. Наш элемент сместился вправо от левого **инлайн-бокса** (текста) ровно на **40px**, а своим правым полем отодвинул следующий за ним **инлайн-бокс** (текст) на **180px**, но так как последний попросту не поместился на эту же строку, он перескочил на новую, а на предыдущей (т.е. на той где наш **элемент**), в правой части теперь балом правит **правый margin** ``-а.

Да, и как вы могли заметить, я специально назначил верхние и нижние поля нашему элементу в **500px**, для того, чтобы вы увидели, что такие вещи не дают никаких эффектов, и **вертикальные margin**-ы попросту не работают.

Отрицательные поля

Рассмотренный до этого пример касался положительных значений **свойства margin** для **инлайн-элементов**. Но, что, например, произойдёт при отрицательных значениях? Давайте это выясним, сразу же перейдя к наглядному примеру.

Пример 2.2 Отрицательные поля

а)

HTML

```
<p>Lorem Ipsum is simply <span>dummy text printing </span> <em>and type setting  
industry.</em> Lorem Ipsum has been the industry's standard dummy text ever since the  
1500s PageMaker including versions of Lorem Ipsum.</p>
```

б)

CSS

```
p {font-size: 16px;}  
  span {  
    background: #FC3;  
    border-radius: 5px;  
    margin-left: -20px;  
    margin-right: -10px;  
  }  
  em {  
    background: #C30;  
    border-radius: 5px;  
  }
```

И. результат:

Lorem Ipsum is simply dummy text printing and type setting industry. Lorem Ipsum has been the industry's standard dummy text ever since the 1500s PageMaker including versions of Lorem Ipsum.

Рис 2.2 Отрицательные поля

В нашей структуре находятся два **инлайн-элемента**: `` и ``. Я специально поставил их рядом, чтобы вы могли лучше понять суть отрицательных **margin** для "инлайн-блочных инлайнов". Жёлтенькому ``-у я присвоил левый отрицательный **margin-left: -20px**, и сразу же правый отрицательный **margin-right: -10px**.

Смотрите, что произошло. При **margin-left: -20px** `` сместился влево ровно на **20px**, при этом наехав своим блоком на левый текст. Отсюда можно сделать вывод, что при отрицательных левых значениях **margin** инлайн-элементы сдвигаются в левую сторону, наезжая на своих соседей слева. И не важно, будь то какой-нибудь текст или другой элемент.

А вот при правом отрицательном **margin** у нас происходит немного другой эффект. В этом случае сдвигается уже следующий за элементом **инлайн-блок**, причём сдвиг происходит в левую сторону навстречу нашему элементу. В нашем случае справа от `` идёт **инлайн-элемент** ``, и как раз таки он перемещается влево ровно на отрицательный **margin** ``-а, и выходит на **-10px**, наезжая при этом поверх ``-а.

В принципе, ничего сложного здесь нет, отрицательные **margin** просты для понимания. Нам всего лишь нужно запомнить пару особенностей их поведения, вот и всё.

Отступы в инлайновом форматировании

Отступы, а они же **padding**, в инлайновом форматировании, так же как и поля (**margin**), не могут влиять на соседние отступы по вертикали, но при этом у **padding**, в отличие от **margin**, имеется интересный эффект, о котором тоже следовало бы знать. Как обычно, рассмотрим всё на примере.

Пример 2.3 Отступы в инлайновом форматировании

а)

HTML

```
<p>Lorem Ipsum is simply dummy text printing and type settings industry. Lorem Ipsum has been the industry's standard <span>dummy text printing</span> dummy text ever since the 1500s PageMaker including versions of Lorem Ipsum. Text ever since the 1500s PageMaker including versions of Lorem Ipsum</p>
```

б)

CSS

```
p {font-size: 16px;}
  span {
    background: #FC3;
    border-radius: 15px;
    padding: 20px 50px;
  }
```

Ну, и конечно же, [результат](#) и изображение:

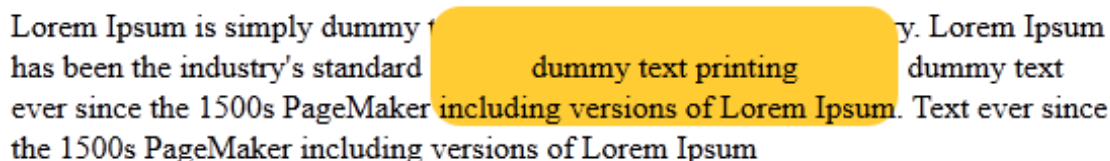


Рис 2.3 Отступы в инлайновом форматировании

В данном примере я решил объединить сразу оба **padding**-а, горизонтальный и вертикальный. Но, говорить мы будем именно о последнем (вертикальном), так как горизонтальный **padding** всего лишь расширяет свой элемент на ширину самого **padding**-а, и, соответственно, такой инлайн-элемент становится просто шире, получая внутренние боковые отступы от своего содержимого. Такое поведение **инлайн-элемента** можно наблюдать на рисунке.

Но вот на вертикальных отступах я бы хотел остановиться подробнее. Дело в том, что вертикальные **padding**-и, с одной стороны, не влияют на поведение своих соседей сверху и снизу, но с другой, могут оказать влияние на высоту и визуальный вид самого элемента. В нашем случае **элемент ** имеет в своих свойствах верхние и нижние **padding**-и в значении **20px**. Но, несмотря на одинаковые значения, поведение этих отступов всё же немного разное. Как можно увидеть на рисунке, за счёт отступов у нашего элемента поменялась не только ширина, но и высота, а точнее у внутреннего содержимого элемента появились верхние и нижние отступы, ровно на значение **padding**, т.е. выходит **20px**. Самое интересное здесь то, что элемент своим верхним отступом залез на своего соседа (текст) **сверху**, а нижним наоборот подлез **под** сам текст снизу. Т.е., по сути, вышло, что в вертикальном контексте сам элемент остался на месте, но вот его внутренние отступы заставили его сменить высоту и налезть **на** и **под** своих соседей.

Такое поведение может стать полезным во многих случаях. Например, если нам необходимо сделать меню из инлайн-пунктов, которым невозможно прописать ширину и высоту, потому что это не блоки, но при этом необходимо как-то выделить их или назначить необходимый фон. Вот тут-то и смогут помочь вертикальные **padding**-и.

Границы в инлайновом форматировании

Границы (**border**) в инлайновом форматировании работают практически так же, как и **padding**. Но, несмотря на их похожесть, я всё-таки решил сделать тестовый примерчик для наглядности.

Пример 2.4 Границы в инлайновом форматировании

а)

HTML

```
<p>Lorem Ipsum is simply dummy text printing and type setting industry. Lorem Ipsum has been the industry's standard <span>dummy text printing</span> dummy text ever since the 1500s PageMaker including versions of Lorem Ipsum. Text ever since the 1500s PageMaker including versions of Lorem Ipsum</p>
```

б)

CSS

```
p {font-size: 16px;}
  span {
    background: #FC3;
    border-radius: 15px;
    border: 20px solid red;
  }
```

И, конечно же, [результат](#):

Lorem Ipsum is simply dummy text printing and type setting industry. Lorem Ipsum has been the industry's standard dummy text printing dummy text ever since the 1500s PageMaker including versions of Lorem Ipsum. Text ever since the 1500s PageMaker including versions of Lorem Ipsum

Рис 2.4 Границы в инлайн-форматировании

Что и следовало доказать. Правые и левые границы отодвинули сам элемент и следующий за ним вправо, а верхние и нижние границы налезли на верхний текст и соответственно под нижний. В общем всё аналогично **padding**.

Разрыв инлайн-бокса

На самом деле на примере границ и фона я хотел бы показать вам ещё одну очень важную особенность в поведении инлайн-боксов. А точнее, в их разрыве в случаях, когда они не умещаются в один **лайн-бокс**. Чтобы было понятно, о чём идёт речь, предоставляю код и изображение.

Пример 2.5 Разрыв инлайн-бокса

а)

HTML

```
<p>Lorem Ipsum is simply dummy text printing and type setting industry. Lorem Ipsum has been the <span>industry's standard dummy text printing</span> dummy text ever since the 1500s PageMaker including versions of Lorem Ipsum. Text ever since the 1500s PageMaker including versions of Lorem Ipsum</p>
```

б)

CSS

```
p {font-size: 16px; margin: 0;}
span {
  background: rgba(160,224,160,1);
  padding: .5em;
  border: 2px dotted #caa;
  border-radius: 1em;}
```

А вот и сам результат и изображение:

Lorem Ipsum is simply dummy text printing and type setting industry. Lorem Ipsum has been the industry's standard dummy text printing dummy text ever since the 1500s PageMaker including versions of Lorem Ipsum. Text ever since the 1500s PageMaker including versions of Lorem Ipsum

Рис 2.5 Разрыв инлайн-бокса

В общем ситуация следующая. В параграфе находится всё тот же ****, но теперь я решил сделать его длиннее, чтобы он не поместился на свою полочку (лайн-бокс). **Каждая строка – это отдельный лайн-бокс**, а наш **инлайн-бокс ()** слишком длинный, и поэтому при столкновении с правой границей лайн-бокса его попросту разорвало на две части, и та часть, которая не поместилась на текущей строке, перескочила на новую.

Но, постойте, выходит, что у нас теперь в двух лайн-боксах один и тот же **инлайн-бокс**? – Нет. Дело в том, что если инлайн-бокс не вмещается в свой контейнер, и при этом перенос строки не запрещен правилами языка, и **white-space: nowrap/pre**, то такой бокс разрывается на две части, и образует уже две половинки

бокса. Но, суть в том, что обе половинки внутри бокса ведут себя по-обычному, и никаких особых правил к ним не применяется. Мало того, обе части встраиваются в свои **лайн-боксы** точно так же, как если бы это были два разных **инлайн-бокса**. Просто заканчивается один лайн-бокс, а за ним начинается другой, первым элементом в котором становится новый инлайн-бокс, содержащий тот текст, что не поместился в первой строке, и имеющий те же CSS-стили.

Да, важная деталь. **padding, margin и border** не применяются к месту разрыва, и поэтому в конце разорвавшегося **инлайн-бокса** и в начале нового можно считать значения этих свойств равными нулю.

Для некоторых такое поведение может показаться странным и даже удивительным, но, на самом деле, здесь все вполне логично. Представьте себе, что у вас в строке находится элемент, и вы задаете ему стили: фон, границы, поля. И вдруг он не помещается в один **лайн-бокс**, рвётся, последняя часть перескакивает на другую строчку, и у вашего элемента становится, например, не две боковых границы, а целых четыре. А вот это уже и правда странно, не правда ли? Поэтому не удивляйтесь, а примите это как правильность.

ИКФ: высота строки, часть 1 (3-я публикация цикла “Тайны CSS2.1”)

Добавлено 14 Июнь 2012 | Раздел: [CSS](#), [Статьи](#), [Тайны CSS2.1](#)

В прошлых статьях мы проделали неплохую работу, разобрав то, как в целом создаются строки и как происходит их построение. Но на таком рубеже не следует останавливаться, потому что это далеко не конец, а лишь начало нашего путешествия в мир под названием "Контекст форматирования".

Продолжая разбирать строчное форматирование, я предлагаю вам коснуться ещё одной познавательной и необходимой для изучения темы. **Высота строки** — одна из важных особенностей при построении строк. Выстраивая нашу строку и заполняя её инлайн-блоками, мы не задумывались о том, **откуда берётся и от чего зависит их высота**.

Высота лайн-бокса

Первым делом давайте всё-таки уточним, из чего же в итоге вычисляется общая высота **лайн-бокса**. Общее правило таково: высота строки берётся с расчётом, чтобы заведомо вместить самый высокий ее элемент. Чем бы этот элемент ни оказался — текстом, картинкой, инлайн-блоком и т.п.

Высота картинок и инлайн-блоков определяется их свойством **height (min-height)** или внутренней высотой (высотой контента для инлайн-блоков). Высота обычных **инлайн-блоков** с текстом определяется значением свойства **line-height** (присвоенного инлайн-элементу или унаследованному им), а если **line-height** не задано — то от метрик шрифта (также заданного самому элементу или унаследованному). **Анонимные инлайн-боксы** получают значения **line-height** и свойства шрифта от своего блочного контейнера.

Высота строки также может зависеть от **вертикального выравнивания ('vertical-align')** инлайн-блоков внутри строки. Возможны варианты, когда итоговая высота строки будет даже больше наибольшей из высот ее элементов. Рассмотрению этих вариантов, и этого свойства в целом, мы посвятим третью часть этого цикла.

Если в строке нет ни одного инлайн-бокса (текстового или "неделимого", типа картинки), то всё вышеперечисленное для её высоты не играет никакой роли, так как в таком случае **лайн-бкс** будет иметь нулевую высоту. В связи с этим, предлагаю начать наши познания с обычного текста. Поехали...

Высота текста

Надо сказать, что высота текстового **лайн-бокса** высчитывается за счёт размера шрифта. Но дело в том, что не все так просто, и здесь есть много нюансов. Поэтому предлагаю начать немного издалека и плавно подойти к сути.

Любой символ шрифта в вебе представлен прямоугольной площадкой определенного размера и пропорций. Основной размер шрифта определяется его **кеглем** (он задается свойством **font-size**), а пропорции — сколько тысячных долеи кегля будут выступать над базовой линией, сколько будет "свисать" под ней, и сколько составит ширина каждого символа — определяется свойствами самого шрифта, его **метриками**. Как правило, ширина площадки у каждого символа своя (только в моноширинных шрифтах площадки разных символов в ширину одинаковы), а вертикальные метрики общие для всех символов и характеризуют весь шрифт целиком. Например, для буквы "А" в шрифте **Ubuntu Titling Bold** площадка будет на 0,75 кегля выступать над базовой линией, на 0,25 свисать вниз под нее и в ширину займет 0,592 кегля — т.е. ее габариты составят 1,000×0,592 кегля. Метрики шрифта хранятся где-то в файле шрифта, детали зависят от его формата и операционной системы (напр., для **шрифтов TrueType** в Windows высота определяется метриками [usWinAscent](#) и [usWinDescent](#)). Но, давайте по порядку.

Площадка

В старинные времена высота самих кегельных площадок каждого символа была абсолютно одинаковой, причём для всех шрифтов данного размера. Не было никаких чётких правил, по которым высота каждой из площадок могла вычисляться по-разному. Выглядело это примерно так:

В эпоху металлического набора было так:

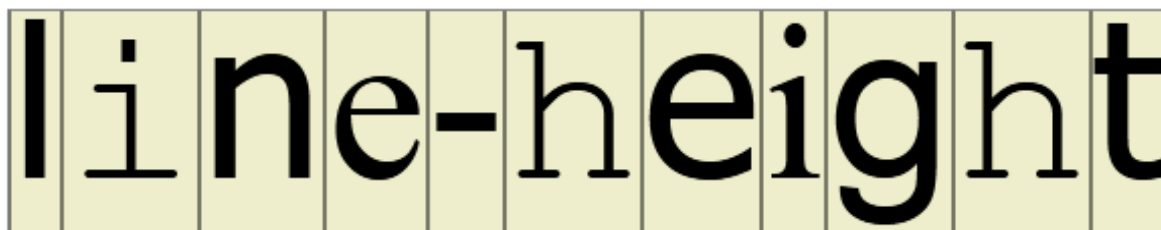


Рис. 3.1 Высота кегельных площадок в металлическом наборе

На изображении видно, что, несмотря на то, что символы разных шрифтов различаются начертанием, насыщенностью, величиной очка буквы и пропорциями, высота площадок при одинаковом кегле (в данном примере он равен 100px) у них одна и та же. Причем даже если бы некоторые символы были верхними индексами — высота их кегельных площадок осталась бы той же самой, равной высоте соседних площадок.

Если бы такое поведение было в компьютерных шрифтах, то эта статья была бы совершенно не актуальна, и все вопросы по поводу высоты строки отпали бы сами по себе. Но, слава Богу, в вебе, в этом плане, происходит всё иначе. Высота площадки каждого символа может быть разной, вычисляться в зависимости от множества факторов, и, соответственно, высота самой строки может легко меняться.

Чтобы была видна разница, приведу наглядный пример:

А с приходом цифры стало так:

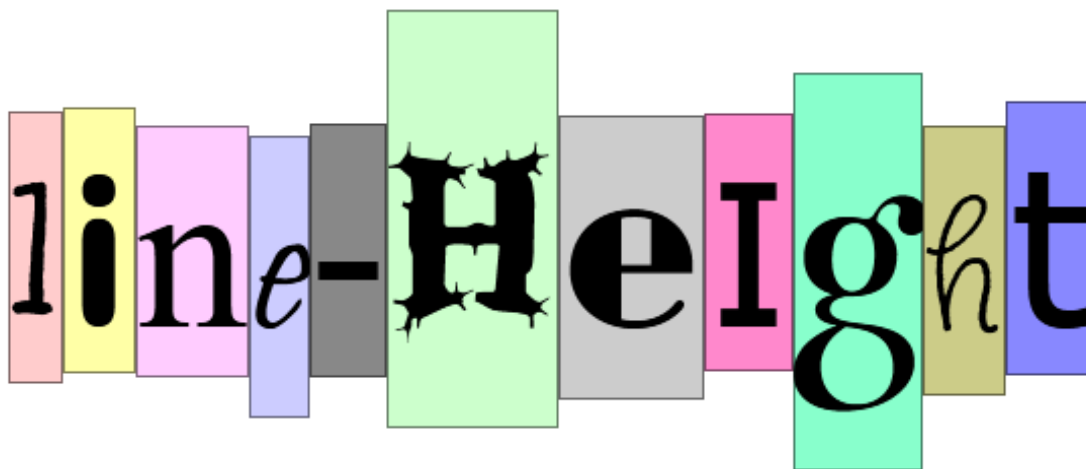


Рис. 3.2 Высота площадок символов в компьютерных шрифтах

Размер шрифта (кегель) по-прежнему один и тот же у всех символов (те же 100px), но площадки символов разных шрифтов, различающихся по начертанию и пропорциям, теперь тоже оказались разными. По сравнению с предыдущим примером разница налицо, не правда ли?

Как определяется высота каждой площадки?

Прежде чем выяснить этот вопрос, я хотел бы познакомить вас с **очень важной деталью** в построении текста, которая имеет название "**Базовая линия**".

Базовая линия

Базовая линия (baseline) для шрифта — что-то вроде ватерлинии для корабля: она делит кегельные площадки на "надводную" и "подводную" части, и служит основой для выравнивания всех инлайн-боксов по умолчанию, независимо от размеров и соотношения этих частей.

Выглядит это примерно так:

А с приходом цифры стало так:

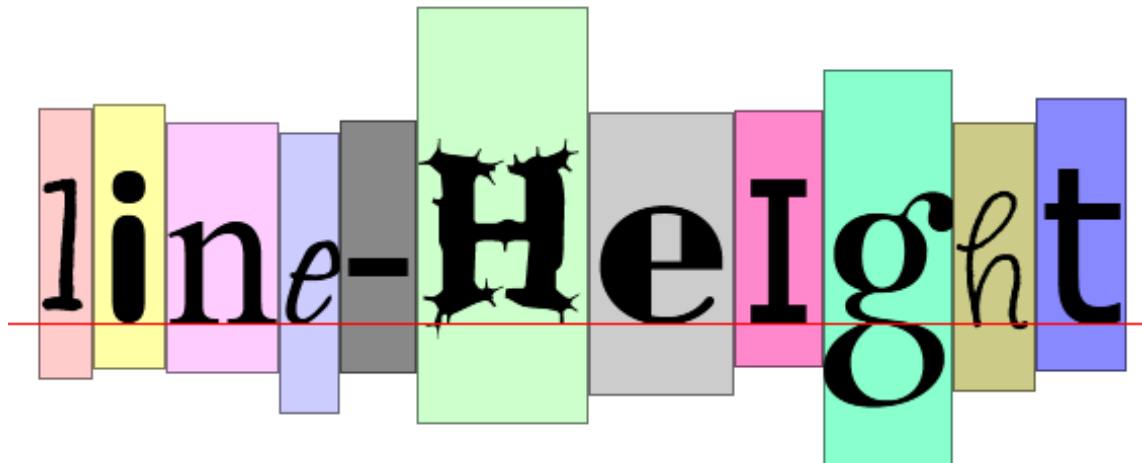


Рис 3.3 Базовая линия шрифта

Красная линия на изображении и есть та самая базовая линия, о которой идёт речь. И, как можно видеть, несмотря на различность в высоте площадок, все символы "сидят" прямо на ней. Обратите внимание на первую букву "g". Её нижняя часть выпирает из базовой линии. Это — нижний **выносной элемент** буквы.

Если присмотреться повнимательнее, то можно увидеть, что между выносным элементом и краем площадки есть еще некоторое пустое пространство. Аналогичное пространство есть и под верхним краем площадки. Это — **запечики**. Они нужны для того, чтобы при построении лайн-боксов тексты из разных строк не слипались друг с другом.

А теперь вернёмся к нашим баранам.

Итак, узнав ~~одну из главных тайн~~ про базовую линию, разобраться с остальным будет намного легче. Поэтому вернёмся к метрикам, о которых я упомянул в начале рассказа.

Две похожие вертикальные метрики берут на себя вычисление двух частей самой площадки. Первая метрика вычисляет верхнюю часть, до базовой линии, а вторая нижнюю, после неё. Например, если брать такие метрики, как `usWinAscent` и `usWinDescent`, то первая (`usWinAscent`) будет вычислять высоту над базовой линией, а вторая (`usWinDescent`) — под.

Ради любопытства мы решили разобрать шрифт в формате svg (где каждая литера описана, как определённая кривая), чтобы посмотреть лежащие на виду метрики. Мы воспользовались генератором с fontsquirrel.com, создающим для каждого шрифта набор файлов, включая svg, и прогнали через него [вот этот шрифт](#). Для каждой литеры указана авансовая ширина (сдвиг координаты следующего символа), а для всего шрифта указаны `ascent` и `descent` вот в таком виде: `<font-face units-per-em="2048" ascent="1638" descent="-410" />`. Именно эти параметры используются для деления площадок на "над базовой" / "под базовой". Над базовой получается $1638/2048$, а под ней — $410/2048$.

Плюс к этому, можно заметить, что высота над и под базовой линией у многих площадок намного превышает высоту самих символов и оставляет кучу пустого пространства над и под ними. Причина кроется в самих шрифтах, а точнее в их символах с большими верхними или нижними выносными и диакритиками, например, всякие там **À** и т.п. Выходит, что высота площадок строится с запасом, чтобы вместить самый высокий символ (со своим хвостом/диакритиком), если таковой окажется на месте текущего.

Кстати, ширина глифов (очертаний) символов может превышать ширину самой площадки, и даже залезать на соседние. Это считается вполне нормальным поведением. Учитывайте и не удивляйтесь этому, если такое встретите.

В общем мы выяснили, что высота площадки складывается из следующих вещей: **Верхние заплечики + высота части символа с запасом над базовой линией + высота части символа с запасом под базовой линией + нижние заплечики.**

Но всё-таки, какова же высота строки в итоге?

Поняв, что же из себя представляют площадки для символов и как высчитывается их высота, мы уже с лёгкостью можем узнать высоту самой строки (**лайн-бокса**). Чуть ранее я говорил, что высота строки зависит от размера шрифта. Но, если говорить точнее, то от размера шрифта зависит высота площадок, а вот уже от высоты наибольшей площадки и зависит высота самого лайн-бокса. Причём для лайн-бокса важен не только верхний край самой высокой площадки, но и нижний – самой низкой из них.

В нашем случае самая высокая площадка вышла у шестого символа слева – буквы "H", а самая низкая у буквы "g" в слове "height", и в итоге высота нашей строки получилась **228px**. А кегль шрифта, напомним, равен всего 100px!

А с приходом цифры стало так:

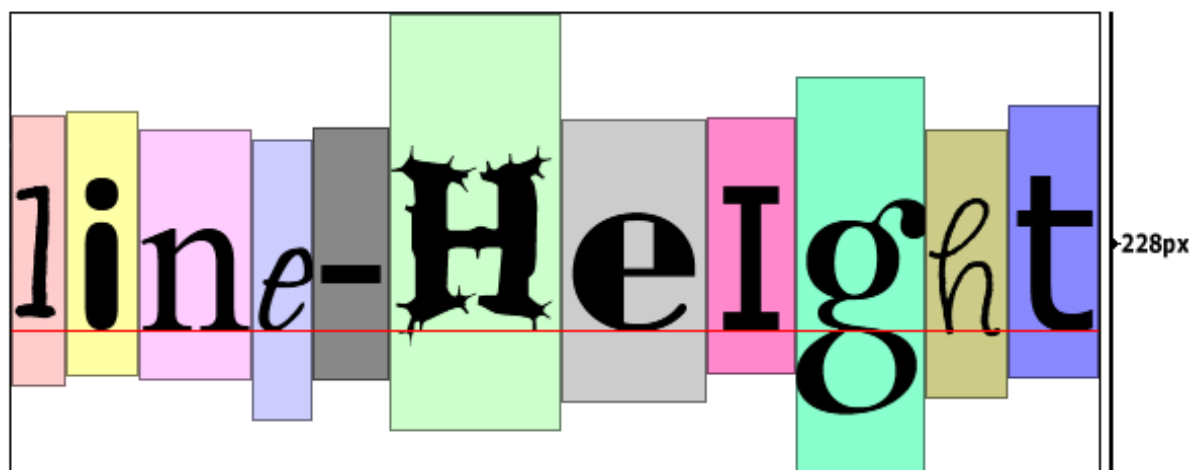


Рис. 3.4 Итоговая высота строки

ИКФ: высота строки, часть 2 (4-я публикация цикла “Тайны CSS2.1”)

Добавлено 22 Июнь 2012 | Раздел: [CSS](#), [Статьи](#), [Тайны CSS2.1](#)

line-height

С помощью текста мы можем регулировать высоту строки только благодаря размерам шрифта, но что делать, если мы хотим задать высоту строки принудительно? Вот тут я должен ввести в игру следующего игрока: “**line-height**”.

Свойство **line-height** устанавливает интерлиньяж строки, т.е. попросту её высоту, а само состоит из высоты кегельной площадки + **half-leading**-ов (о них позже). Рассмотрим по порядку, как это действует.

line-height для текста

Начнем с простейшего случая: пусть наша строка состоит из единственного инлайнового бокса, представляющего собой обычный текст, набранный одним шрифтом одного размера.

Сразу пример:

Пример 4.1 line-height для текста (единственный инлайн-бокс)

а)

HTML

```
<p> Свойство line-height - устанавливает интерлиньяж строки </p>
```

б)

CSS

```
p {  
    border: 1px solid #000;  
    background: #F90;  
    font-size: 30px;  
    line-height: 200px;  
    padding-left: 20px;  
}
```

А вот как это выглядит:



Свойство line-height - устанавливает интерлиньяж строки

Рис. 4.1 line-height для текста (единственный инлайн-бокс)

Как я и говорил, внутри параграфа находится обычный текст. Но, на самом деле, весь оранжевый блок – это и есть **лайн-бокс**, и как можно заметить, его высота прилично превышает высоту самого текста. Размер шрифта я выставил **30px**, а шрифт я даже не стал трогать, и значит, весь текст состоит из одного шрифта, да к тому же простого, не экзотического. Следовательно, можно взять высоту по максимуму из всех площадок, которая может составлять... думаю, не больше **50px** (и то вряд ли). **line-height** я выставил в **200px**, и если отнять от этой цифры размер максимальной площадки (**50px**), то остаётся **150px**, но простите, а куда они тогда делись? Что бы это понять, нужно углубиться и разобрать все составляющие **line-height**.

YUX]b['fi L

В старые времена **leading**-ом считались свинцовые бруски, которые прокладывали между рядами литер металлического набора, т.е. добавочная высота "свинцовых полосок", отсюда **leading**, от lead – свинец. Именно с помощью них в старые времена в типографиях раздвигали строки. Но вот в Вебе **leading**-ом принято считать математическую разность заданного **line-height**-а и высоты кегля. Но, если говорить уже знакомым нам языком, то **leading** - это разность **line-height**-а и самой большой площадки, которые отрендерились **font-size**'ом и метриками шрифта, о которых мы говорили выше.

А теперь вернёмся к нашему примеру. Мы допустили, что максимальная высота самой высокой площадки составляет **50px**, а разность **line-height**-а и высоты этой площадки получилась 150px. Вот последняя цифра и будет тем самым **leading**-ом, добавочным "бруском".

Да, но, вы можете задать вопрос: "А почему же тогда текст находится посередине лайн-бокса?". Всё дело в том, что, на самом деле, **leading** делится на две равные части, а каждая часть называется "**half-leading**", что в переводе означает «половина междустрочия». И вот, как раз таки, к каждой стороне высоких площадок и прибавляется вот по такому **half-leading**-у. В нашем случае вышло, что к каждой стороне прибавилось по **75px**, вот почему текст выравнился посередине.

Я предлагаю рассмотреть эту ситуацию прямо на нашем примере:

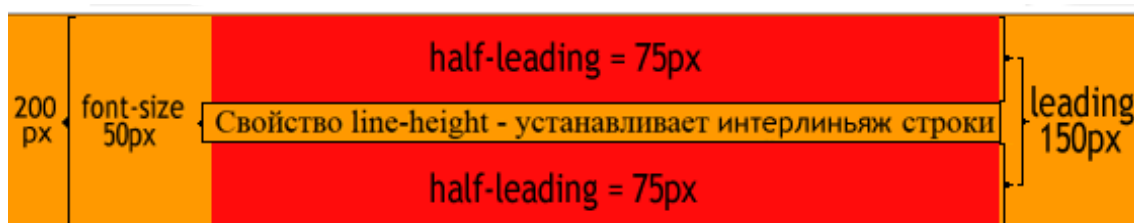


Рис 4.2 Вычисление half-leading'ов

На рисунке можно увидеть, из чего состоит высота строки: **half-leading (75px) + font-size (50px) + half-leading (75px)**. Это и есть наш **line-height**.

Отрицательный leading

До этого момента мы разобрали ситуацию, когда **line-height** больше, чем размер шрифта, но что, например, произойдёт, если **line-height** окажется меньше? Именно на этот вопрос мы и должны ответить прямо сейчас.

Давайте рассмотрим конкретный пример.

Пример 4.2 Отрицательный leading

а)

HTML

```
<p> line-height = font-size + (half-leading * 2) </p>
```

б)

CSS

```
p {  
    border: 1px solid #000;  
    background: #F90;  
    font-size: 30px;  
    line-height: 10px;  
    padding-left: 20px;  
}
```

А вот результат:

$$\text{line-height} = \text{font-size} + (\text{half-leading} * 2)$$

Рис 4.3 Отрицательный leading

Если предположить, что самая большая площадка по высоте составляет **30px**, а **line-height** равен **10px**, то высота строки в нашем случае будет равна именно десяти пикселям. Почему? А дело в том, что в такой ситуации **leading** начинает работать в обратную сторону, т.е. попросту становится отрицательным, а так как он состоит из двух частей (**half-leading*2**), то уменьшаться, по сути, начинают именно последние. Ну, и, конечно же, высота укорачивается с двух сторон в абсолютно одинаковой степени. Например, в нашем случае, **half-leading**-и составляют по **-10px** (+-), так как шрифт установлен в **30px**, а **line-height** в **10px**.

leading можно сравнить с полями (**margin**), только последние делают отступы снаружи элемента, а **leading** снаружи самой высокой площадки в строке. Как и **margin**, **leading** тоже может быть отрицательным, но самое интересное то, что в итоге **line-height** будет всё равно равен конечному результату. Наш пример можно трактовать как: **line-height = 30 + (-10 * 2) = 10**.

Мы выяснили, что делая значение **line-height** меньше размера шрифта, на самом деле мы делаем отрицательным **leading** и уменьшаем именно его, за счёт чего вычисляется итоговая высота строки. Но есть ли какие-нибудь ограничения? Да, есть. На самом деле отрицательный **leading** не может быть больше высоты самой высокой площадки в строке. Т.е, если, например, размер шрифта составляет **30px**, то **half-leading**-и с двух сторон могут быть максимум по **-15px** каждый. Что и логично, ведь посудите сами, мы же не можем вывернуть строку наизнанку и сделать **line-height** равным: -1, -2 и т.д.

Другими словами, **half-leading**-и не могут отобрать от высоты строки больше, чем по половине, и минимальный теоретический предел высоты строки – ноль. На практике он может быть еще больше, стандарт разрешает браузерам иметь "свое мнение" по этому поводу. Например, совсем недавно Webkit-браузеры не позволяли сжать высоту строки меньше высоты строчных букв.

Отрицательный leading и более одной строки

Учитывая, что из-за отрицательного **leading**-а высота строки становится меньше содержимого (в нашем случае текста), нужно учитывать ещё и то, что эта особенность может навредить в тех моментах, когда строк становится две или больше.

По традиции, сразу приведу пример:

Пример 4.3 Отрицательный leading и более одной строки

а)

HTML

```
<p> Свойство line-height - устанавливает интерлиньяж строки, т.е. попросту её  
высоту</p>
```

б)

CSS

```
p {  
    border: 1px solid #000;  
    background: #F90;  
    font-size: 30px;  
    line-height: 10px;  
    padding-left: 20px;  
}
```

Ну, и конечно же, сам результат:

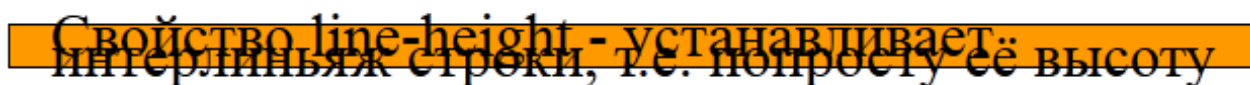


Рис 4.4 Отрицательный leading и более одной строки

По результату можно сделать вывод, что вторая строка началась раньше, чем закончилась первая, а точнее раньше, чем закончилась высота текста в первой строке. Такое поведение легко объясняется тем, что фактическая высота строки составляет всего **10px**, и содержимое строк вынуждено подстраиваться под этот размер. В связи с чем происходит наложение второй строки на первую, третьей на вторую и т.д.

ИКФ: высота строки, часть 3 (5-я публикация цикла “Тайны CSS2.1”)

Добавлено 28 Июнь 2012 | Раздел: [CSS](#), [Статьи](#), [Тайны CSS2.1](#)

line-height для инлайн-элементов

Элементы с **display: inline**, как мы уже знаем, создают инлайновые боксы. В основном они ведут себя так же, как обычный текст, но могут иметь собственный фон, границы и т.п. Как же влияет **line-height** на них? Это мы прямо сейчас и выясним.

Начнём с того, что высота инлайн-бокса определяется либо действующим значением **line-height**, либо характеристиками шрифта (читайте про метрики и площадки шрифта [в третьей части](#)). Это касается случая, когда инлайн-бокс не пуст, т.е. содержит хотя бы один любой символ. В противном (пустом) случае высота инлайна схлопывается в ноль.

Простейший пример:

Пример 5.1 line-height для инлайн-элементов

а)

HTML

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

б)

CSS

```
p {
  border: 1px solid #000;
  background: #F90;
  padding-left: 20px;
  width: 80px;
}

span {
  font-size: 30px;
  background: #FC6;
}
```

И [результат](#):

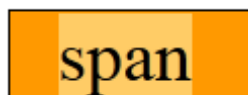


Рис. 5.1 line-height для инлайн-элементов

В данной ситуации высота лайн-бокса, содержащего наш инлайн-элемент (****), равна высоте площадки в данном шрифте, т.е. здесь повторяется та же ситуация с метриками шрифта и площадками, о которых мы говорили в предыдущей части.

line-height меньше или больше размера шрифта

Я решил объединить те два случая, когда **line-height** больше или меньше самого размера шрифта у инлайнов, потому что они схожи с таковыми же для простого текста, а значит, практически ничего нового мы тут для себя не откроем. Но всё же код и результаты следует увидеть.

Пример 5.2 line-height больше и меньше размера шрифта

а)

HTML

```
<p class="minus"><span>span</span></p>
<p class="plus"><span>span</span></p>
```

б)

CSS

```
p {
  border: 1px solid #000;
  background: #F90;
  padding-left: 10px;
  width: 80px;
  margin: 40px auto;
}

.minus span {
  font-size: 30px;
  background: #FC6;
  line-height: 10px;
}

.plus span {
  font-size: 30px;
  background: #FC6;
  line-height: 100px;
}
```

[Оба результата](#) сразу:

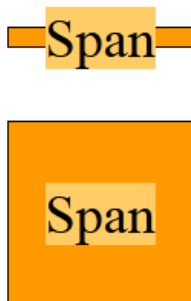


Рис. 5.2 line-height больше и меньше размера шрифта

Когда **line-height** инлайнов больше или меньше размера шрифта (площадок, вычисленных метриками), в дело вступают уже знакомые нам "**leading**" и, следовательно, **half-leading**-и с обеих сторон. В первом случае (отрицательном) они уменьшают высоту строки из-за того, что **half-leading**-и у площадок становятся отрицательными, а во втором (положительном) наоборот – увеличивают строку по высоте, добавив при этом лишние "свинцовые бруски" к нашим площадкам. В общем, всё тоже самое, что и с обычным текстом.

Ну, и конечно же, в случае с маленьким **line-height** не следует забывать о наложении строк.

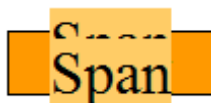


Рис. 5.3 Наложение строк

line-height + границы и отступы

В случае, если мы захотим назначить нашим инлайнам **padding**, **border** или **margin**, то последние никак не повлияют на высоту строки, так как эти свойства по вертикали не оказывают влияния на окружающий контент (об этом мы уже говорили в предыдущей статье).

В последнем коде я сделал некоторые изменения, добавив к нашим инлайн-элементам **padding** и **border**.

Пример 5.3 Стили line-height + границы и отступы

CSS

```
p {
    border: 1px solid #000;
    background: #F90;
    padding-left: 10px;
    width: 100px;
    margin: 40px auto;
}

.minus span {
    font-size: 30px;
    background: #FC6;
    line-height: 10px;
    padding: 100px 0;
    border: 10px solid #000;
}

.plus span {
    font-size: 30px;
    background: #FC6;
    line-height: 100px;
    padding: 100px 0;
    border: 10px solid #000;
}
```

И вот что получилось:

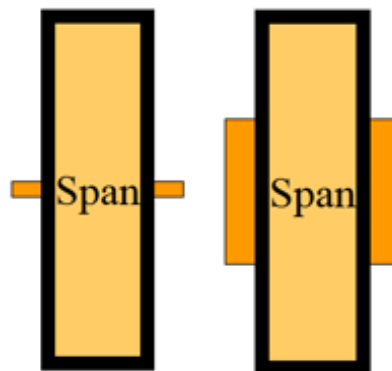


Рис. 5.4 line-height + границы и отступы

Визуально может показаться, что отступы и границы расширили по высоте сами площадки, но это не так. На самом деле площадки остались **той же высоты**, что и были после расчёта метрик шрифта, а **padding** и **border** добавили к ним лишь **визуальные** фон и границы, не влияя при этом на их положение по вертикали.

Грубо говоря, мы имеем следующее: полосу с границей (**border**), далее идёт полоска пространства с фоном (**padding**), потом **сама площадка**, потом опять пространство с фоном (**padding**), ну и завершает всю эту конструкцию снова граница (**border**).

Вследствие чего можно сделать вывод, что **"leading"** и **padding или border** – это совершенно разные вещи, которые никак не пересекаются, не относятся друг к другу и выполняют в строке каждый свои функции.

Например, **padding** визуально добавляет к площадке место с тем же фоном, но на положение по вертикали не влияет, а вот **"leading"**, наоборот, влияет только на положение площадок и на сдвиг их координат.

Высота строки при разных инлайн-боксах и line-height

Мы выяснили, как вычисляется высота строки, если в ней содержится всего лишь один инлайн-элемент, но

какова будет она, если скажем лайн-бокс будет состоять не только из инлайн-элемента, но ещё и какого-нибудь текста или другого инлайна?

Разберём на примере:

Пример 5.4 Высота строки при разных инлайн-боксах и line-height

а)

HTML

```
<p> Before <span> Span Text Span</span> After </p>
```

б)

CSS

```
p {  
    border: 1px solid #000;  
    background: #F90;  
    padding-left: 10px;  
    width: 400px;  
    line-height: 50px;  
    font-size: 40px;  
}  
  
p span {  
    background: #FC6;  
    line-height: 150px;  
    font-size: 12px;  
}
```

Посмотрим результат:



Рис. 5.5 Высота строки при разных инлайн-боксах и line-height

Ситуация следующая. Есть строка, в которой содержатся два **анонимных инлайн-бокса (текста)** и **"натуральный" инлайн-бокс** (инлайн-элемент ``). Последний, надо заметить, находится ровно по середине. Примерные площадки у текста и инлайн-элемента я выделил специальными цветами для того, чтобы вы могли лучше представить происходящее. Несмотря на то, что сам текст и его площадки явно превышают по высоте площадку инлайн-элемента, они вынуждены подстраиваться именно под него, а не наоборот. Почему так происходит? Давайте разберёмся...

Дело в том, что дефолтное поведение **лайн-боксов** в блоке определяется размером шрифта (**font-size**) и **высотой строки (line-height)**, указанным для самого блока (или теми значениями **font-size** и **line-height**, которые выходят для блока по итогу всех каскадов и наследований), а всё, что внутри – переопределяется. Исключения вынуждены приспосабливаться под общие правила. В нашем случае исключением является инлайн-элемент, потому что его размеры шрифта и высота переопределены и явно отличаются от шрифта и высоты строки самого блока. Несмотря на свои малые размеры, он выстроился по базовой линии, которая берётся по основному тексту. Именно такое поведение "исключений" и является подстраиванием под общие правила.

font-size нашего блока равен **40px**, **line-height 50px**, а это означает, что высота текста в блоке будет **50px**, включая площадки и "**leading**". Но почему же тогда общая высота строки у нас получилась **150px**? А всё дело в том, что в таком поведении **лайн-бокса** виновато именно наше "исключение" (инлайн-элемент). У последнего выставлен **line-height** в значении **150px**, что заставляет его "**leading**" распирать строку. Следовательно, высота инлайн-элемента теперь становится больше родительского **лайн-бокса**, и он в итоге и определяет фактическую высоту общей строки, центрируется в ней и подтягивает за собой базовую линию всей строки.

Базовая линия — это своего рода проволока, на которой сидят все «буквы-птицы», независимо от их размера. И только в самом высоком инлайн-боксе буквы не могут двигаться, т.к. жёстко зафиксированы **half-leading**-ами (чтобы обеспечить требуемый **line-height**) и поэтому двигаться приходится всем остальным буквам. За счёт чего и происходит "подтягивание" общей базовой линии.

Отсюда можно сделать вывод, что сколько бы инлайн-боксов не находилось в строке, её высота будет определяться самым высоким из них.

ИКФ: высота строки, часть 4 (6-я публикация цикла “Тайны CSS2.1”)

Добавлено 05 Июль 2012 | Раздел: [CSS](#), [Статьи](#), [Тайны CSS2.1](#)

line-height для инлайн-блоков и инлайн-таблиц

Несмотря на то, что **инлайн-блоки** (**inline-block**), так же, как и инлайн-таблицы (**inline-table**) входят в инлайновое форматирование строк, они имеют свои особенности, которые отличаются от особенностей обычных инлайн-элементов или текста. Это довольно-таки интересно и, нам следует о них поговорить.

Эти боксы — особенные: снаружи они ведут себя как неразрывные инлайн-боксы (наподобие отдельных слов текста), но внутри у них — свои собственные, обособленные блочные «миры», блочные контексты форматирования (о них мы расскажем в скором будущем). У инлайн-блока всё его содержимое представляет собой такой «блочный мир» (поэтому он и инлайн-блок), а у инлайн-таблицы отдельный блочный контекст создается каждой ячейкой. И в каждом из этих «маленьких блочных миров» могут находиться собственные лайн-боксы, а в них — инлайн-боксы, как обычно.

Высота **инлайн-блоков** и **инлайн-таблицы** определяется по их контенту или по жёстко заданной высоте, как и у обычных блоков. А вот с базовой линией у них всё же есть некие отличия.

Когда **инлайн-блок** выступает в своей инлайновой роли, у него, как и у площадки шрифта, есть своя базовая линия, по которой он выравнивается относительно окружающего текста. Этой линией становится базовая линия последней строки текста инлайн-блока.

Чтобы не нагонять путаницу, сразу покажу пример:

Пример 6.1 line-height для инлайн-блоков

а)

HTML

```
<p> Внешний текст <span> inline-block inline-block inline-block </span> внешний текст</p>
```

б)

CSS

```
p {
    border: 1px solid #000;
    background: #F90;
    width: 300px;
    line-height: 150px;
}

span {
    line-height: normal;
    display: inline-block;
    border: 1px solid #000;
    width: 70px;}
```

И вот что вышло:

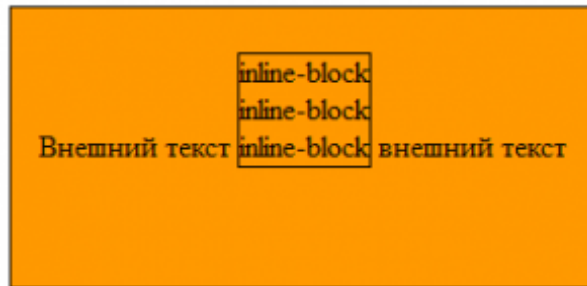


Рис. 6.1 line-height для инлайн-блоков

Как можно понять из рисунка, инлайн-блок (****) встал на общую базовую линию своим самым нижним текстом (своей базовой линией).

Так ведет себя инлайн-блок во всех случаях, кроме двух исключений: если он пуст (не содержит ни символа текста) или у него изменено значение overflow (на любое, кроме visible). В этих случаях по внешней базовой линии равняется нижняя граница самого инлайн-блока или окончательная координата его нижнего поля (margin).

Последний случай очень редкий, но, несмотря на это, знать об этом тоже желательно.

У инлайн-таблиц тоже есть базовая линия, по которой они выравниваются внутри лайн-бокса. В качестве этой линии берется нижний край ячеек первой строки инлайн-таблицы.

Пример 6.2 line-height для инлайн-таблиц

а)

HTML

```
<div>
  Внешний текст
  <table>
    <tr>
      <td>1</td>
      <td>2</td>
    </tr>
    <tr>
      <td>3</td>
      <td>4</td>
    </tr>
  </table>
  внешний текст
</div>
```

б)

CSS

```
div {  
  border: 1px solid #000;  
  background: #F90;  
  width: 300px;  
  padding-left: 20px;  
  line-height: 100px;  
}  
  
table {  
  line-height: normal;  
  display: inline-table;  
  border: 1px solid #000;  
  width: 70px;  
  border-spacing: 4px;  
}  
  
td {padding: 0; border: 1px solid #00f;}
```

И тут уже результат будет следующим:

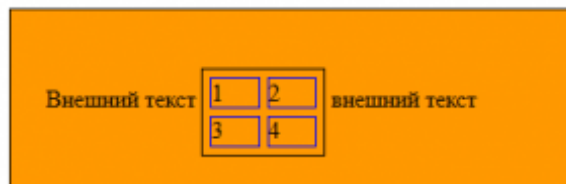


Рис. 6.2 line-height для инлайн-таблиц

В нашей инлайн-таблице я сделал пару строк, чтобы было понятно, что вторая и последующая строки не имеют никакого смысла для её базовой линии. Низ первой строки играет решающую роль в этом деле. Именно низ первой строки в инлайн-таблице и является её базовой линией, он же и выстраивается по внешней.

Система координат для инлайн-блоков и инлайн-таблиц

Если **line-height**, и следовательно, **leading** может добавить к площадкам шрифта дополнительные "свинцовые" полосы и отодвинуть координату площадки, то с инлайн-блоками и инлайн-таблицами такой номер не пройдет. Дело в том, что **line-height**, так же, как и **leading**, актуальны только для шрифтов (**текста** и **инлайн-элементов**). Поэтому вертикальная координата **инлайн-блока** и **инлайн-таблицы** зависит от их высоты.

Можно немного доработать наш пример с текстом и инлайн-блоком.

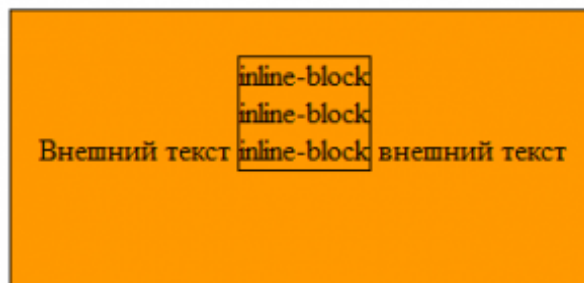


Рис. 6.3 line-height для инлайн-блоков

Текст, который находится непосредственно в самом абзаце, имеет достаточно большой **line-height** (и соответственно **leading**), чтобы контролировать высоту строки. Сам инлайн-блок состоит из нескольких строк текста, но **line-height** не действует на них по той причине, что я намеренно отменил **line-height** именно для инлайн-блока, вернув его в нормальное состояние. Это можно увидеть в коде (**line-height: normal**). В противном случае строки внутри инлайн-блока стали бы огромными и растянули бы его самого на соответствующую высоту.

line-height инлайн-блока или инлайн-таблицы применяется к их текстовому содержимому. Если в результате этого часть **инлайн-блока (инлайн-таблицы)**, выступающая над базовой линией или под ней, оказывается больше расстояния между базовой линией и соответствующей границей лайн-бокса, растягиваться начинает уже сам **лайн-бокс**.

Рассмотрим небольшой пример:

Пример 6.3 line-height для инлайн-блоков

а)

HTML

```
<p> Внешний текст <span> Инлайн-блок  Инлайн-блок </span>  внешний текст </p>
```

б)

CSS

```
p {
    border: 1px solid #000;
    background: #F90;
    width: 320px;
    padding-left: 20px;
    line-height: 30px;
}

span {
    line-height: normal;
    display: inline-block;
    border: 1px solid #000;
    width: 90px;
    margin: 25px 0;
    padding: 25px 0;
}
```

А вот и результат:

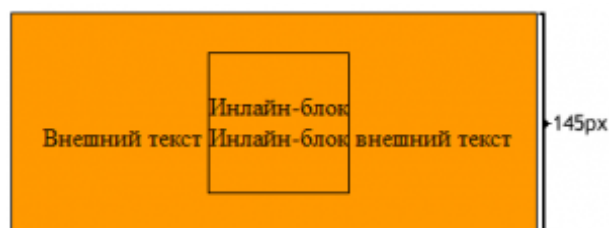


Рис. 6.4 line-height для инлайн-блоков

Несмотря на то, что высота основных текстовых площадок в абзаце составляет **30px**, общая высота строки получается **145px**. Происходит это, как вы уже поняли, благодаря нашему инлайн-блоку, который по своей высоте явно превосходит высоту соседних площадок, упираясь краями в верх и низ лайн-бокса и тем самым растягивая его. **Высота инлайн-блока состоит из верхнего и нижнего полей (margin), границ (border), внутреннего отступа (padding) и общей высоты всех строк его содержимого.**

Заметьте, что общая базовая линия тоже меняет свои координаты, подтягиваясь к базовой линии нашего инлайн-блока. Здесь ситуация напоминает вариант, когда базовая линия выравнивается по самому высокому инлайн-блоку в строке. В нашем случае самым высоким в строке является инлайн-блок, поэтому весь остальной "инлайновый мир" по бокам вынужден подстраиваться именно под него.

Кстати, с точки зрения размещения на странице, логичнее считать неподвижной именно верхнюю границу, потому что она упирается в предыдущий лайн-бокс. А базовая линия и нижняя граница могут гулять от нее вниз. Даже если инлайн-блок или инлайн-таблица растягивает верх лайн-бокса, на самом деле это базовая линия удаляется от верха.

Заключение

В этой статье мы описали то, как инлайн-блоки и инлайн-таблицы должны вести себя по спецификации CSS. К сожалению, браузеры "любят" ее нарушать (даже такие, как Хром и Сафари). О паре таких исключений мы расскажем в следующей главе.

ИКФ: высота строки, часть 5 (7-я публикация цикла “Тайны CSS2.1”)

Добавлено 12 Июль 2012 | Раздел: [CSS](#), [Статьи](#), [Тайны CSS2.1](#)

В комментариях ([1](#), [2](#)) к предыдущей публикации нашего цикла внимательные читатели указали, что не все браузеры одинаково (по спецификации) определяют базовую линию для инлайн-блоков и инлайн-таблиц. Сюрприз: две наиболее заметные ошибки — в браузерах на движке webkit, которые всегда славятся хорошей поддержкой модных CSS-новинок. Вот эти ошибки:

1. Overflow инлайн-блоков не влияет на положение их базовой линии ([ссылка на баг](#)).
2. Инлайн-таблицы выравниваются не по базовой линии первой строки, а по нижней границе всей таблицы. [Этот баг](#) уже исправлен и ближайшие релизы вот-вот начнут вести себя по спецификации.

line-height для замещаемых элементов

Поведение замещаемых элементов отчасти напоминает поведение инлайн-блоков по отношению к высоте строки, но в остальном они всё же отличаются. Но, давайте по порядку...

Что есть "Замещаемый элемент"

Для начала давайте выясним, о каких таких "Замещаемых элементах" идёт речь.

Замещаемый элемент — это элемент, который до загрузки документа имеет лишь пустой контейнер, а в дальнейшем подгружает своё содержимое из внешних источников. Т.е. его содержимое **замещается** чем-то, что **не находится** непосредственно в самом документе. Самый простой пример, это элемент ``, который **замещается** файлом изображения. Так же существует ещё ряд замещаемых элементов, таких как: `<video>`, `<object>`, `<input>` (много разных типов) и т.д.

Высота замещаемых элементов

Если содержимое замещаемого элемента неизвестно заранее, то как же тогда узнать его высоту? В этом плане у замещаемых элементов есть несколько критериев, по которым можно определить их точную высоту. Предлагаю ознакомиться со всеми ними:

- Как и у **инлайн-блоков**, высота замещаемых элементов так же может зависеть от вертикальных полей (**margin**) и прибавлять их значения к общей высоте элемента. Но, если нижние и **верхние поля (margin)** у замещаемых элементов имеют значение **"auto"**, то итоговое значение их полей будет равно нулю.
- Высоту замещаемым элементам можно задать непосредственно в стилях либо в атрибуте, прописав в свойстве **"height"** нужное значение. Это хорошо в том случае, если мы хотим, чтобы высота замещаемого элемента была принципиального размера. В противном случае, если у замещаемого элемента высота и ширина имеют **значение "auto"**, то его высота рассчитывается по контенту, т.е. попросту по загруженному содержимому. Например, высота картинки может зависеть от ширины и высоты внешнего файла, который в неё загрузится.
- В случае, если высота замещаемого элемента **"auto"**, и у элемента есть внутренние пропорции, то итоговая высота элемента будет равна: (итоговая ширина) / (внутренние пропорции (Ш:В)). Например, почти у всех фотографий с "зеркалок" пропорции **3:2**, а с компактных фотоаппаратов **4:3**. В случае **4:3** при высоте **"auto"** и чётко заданной ширине, допустим **100px**, высота получится $100 / (4/3) = (100*3) / 4 = 75px$.

Надо понимать, что не все замещаемые элементы имеют внутренние пропорции, это в основном касалось таких элементов, как: ``, `<object>` (если в нём есть картинка или что-то типа того), `<input type="image">` и т.д. Все остальные элементы скорее попадают под следующий критерий.

- Если вычисленное значение **'height'** равно **'auto'** и у элемента есть внутренняя высота, то эта внутренняя высота становится итоговым значением **'height'**. Т.е. у многих элементов есть высота по умолчанию. Она зависит от предпочтений браузеров и в каждом из них может быть разной. Например, у того же `<input type="file">` нет никаких внутренних пропорций, но зато есть своя, собственная высота.

- И последний случай — это когда вычисленное значение **'height'** замещаемого элемента равно **'auto'**, но ни одно из условий выше не выполняется. В таком случае итоговое значение **'height'** должно устанавливаться как высота максимально возможного прямоугольника с пропорцией **2:1**, который не выше **150px** и не шире чем ширина устройства. Т.е. если у нас, к примеру, есть картинка без "родных" пропорций, то она считается как имеющая пропорции **2:1** и пытается втиснуться в ширину экрана и в **150px** по высоте. Например, на большом экране размер у такой картинки будет **300x150**, на маленьком её ширина будет равна ширине экрана, а высота — половине этой ширины.

Базовая линия замещаемых элементов

Относительно базовой линии окружающего текста замещаемые элементы ведут себя по-разному. Некоторые становятся на нее своим нижним краем (как пустой инлайн-блок), некоторые — нижним краем последней строки своего текста. В общем, давайте смотреть на примерах.

Начнём с тех элементов, которые встают на базовую линию своим нижним краем или полем (margin), если таковое имеется. Таковыми элементами являются, например, ****, **<object>**, **<video>** и т.д.

Рассмотрим маленький пример.

Пример 7.1 Базовая линия замещаемых элементов (изображения)

а)

HTML

```
<p> Внешний текст  внешний текст </p>
```

б)

CSS

```
p {
  border: 1px solid #000;
  height: 200px;
  padding: 0 10px;
  font-size: 20px;
  background: #F90;
}

img {background: #000;}
```

И вот такой вот результат:



Рис. 7.1 Базовая линия замещаемых элементов

Для примера я выбрал элемент ``. Как видно из скриншота, его базовая линия совпадает с его нижним краем. В случае с нижним полем (например, `margin-bottom: 20px`) отсчёт базовой был бы уже от низа самого поля и выглядело бы это примерно так:



Рис. 7.2 Базовая линия замещаемых элементов с нижним полем

А вот с другими замещаемыми элементами, такими как: `<select>`, `<input>` (разными типами) ситуация немного поинтереснее. Она даже чем-то напоминает ситуацию с инлайн-блоками. Из следующего примера вы поймёте, почему я так говорю.

Пример 7.2 Базовая линия замещаемых элементов (input-ы)

а)

HTML

```
<div>
  Текст
  <input type="text" value="123" size="3">
  <input type="button" value="456">
  <input type="checkbox">
  <input type="radio">
  <button>789<br>zzz</button>
  Текст
</div>
```

б)

CSS

```
div {
  border: 1px solid #000;
  padding: 10px;
  font-size: 20px;
  background: #F90;
}
```

А результат будет такой:



Рис. 7.3 Базовая линия замещаемых элементов (input-ы)

Обратите внимание, где находится базовая линия элементов, в которых есть текст. Правильно! Именно там, где заканчивается последняя строка в тексте. И лучше всего это можно заметить по элементу **<button>**, содержимое которого состоит из двух строк. Тоже самое у нас происходило с инлайн-блоками и отчасти с инлайн-таблицами, только в последних за основу для базовой линии брались ячейки первого ряда.

Надо заметить, что такое поведение нигде особо не документировано и поэтому это можно смело назвать доброй волей и здравым смыслом разработчиков браузеров.

Система координат для замещаемых элементов

Поведение замещаемых элементов относительно высоты строки схоже с поведением инлайн-блоков или **инлайн-таблиц**. Следовательно, у замещаемых элементов нет никакого **leading**-а и их высота определяется по выше описанному маршруту (см. выше **Высота замещаемых элементов**). И, если, например, их верхний край будет выше верхнего края родительского лайн-бокса, то высота последнего будет такой, чтобы уместить самый высокий замещаемый элемент. Такую ситуацию мы уже наблюдали чуть ранее, на одном из рисунков:



Рис. 7.4 Базовая линия замещаемых элементов с нижним полем

Здесь, в общем-то, ничего нового. Намного интереснее было бы поговорить о влиянии самого **line-height**'а на замещаемые элементы, потому что в этой области есть много интересных нюансов.

line-height не оказывает влияния на элементы ****, **<video>** или **<object>**, но зато с такими элементами, как **<select>**, **<button>** или **<input>** (разными типами), происходят удивительные вещи.

Рассмотрим ситуацию на конкретном примере.

Пример 7.3 Система координат для замещаемых элементов

а)

HTML

```
<div>
  Текст
  <input type="text" value="123" size="3">
  <input type="button" value="456">
  <input type="checkbox">
  <input type="radio">
  <button>789<br>zzz</button>
  Текст
</div>
```

б)

CSS

```
div {  
  border: 1px solid #000;  
  padding: 10px;  
  font-size: 20px;  
  background: #F90;  
}  
  
div * {line-height: 50px;}
```

К сожалению ситуация настолько отличается от браузера к браузеру, что мне пришлось сделать огромный скриншот. Вы просто обязаны это увидеть!



Рис. 7.5 Система координат для замещаемых элементов

Почти во всех браузерах ситуация абсолютно разная. Единственные, кто солидарен между собой – это **Firefox** и **Opera**, именно их поведение я считаю правильным. **Chrome** и **Safari** позволили себе лишнего, заставив **line-height** влиять на `<input type="text">` и на `<input type="button">`. А браузеры **Internet Explorer** вдобавок вообще сотворили нечто ужасное с `<input type="text">`. В итоге, предсказуемо ведёт себя только лишь элемент `<button>` (поведение которого напоминает поведение обычного инлайн-блока), а остальные — как повезёт.

Почему я считаю, что поведение Firefox и Opera самое правильное? Дело в том, что текст, который мы привыкли видеть в элементах `<input>`, на самом деле **не является** контентом. Атрибут **value**, в котором содержится этот текст, не отвечает ни за какой контент, а вместо этого является лишь свойством, которое, строго говоря, вообще не обязано отображаться. Вспомните тот же чекбокс (**type="checkbox"**), разве вы когда-нибудь видели, чтобы его **"value"** где-нибудь отображалось? Визуально мы, конечно же, можем видеть текст в самих `<input>`, но на самом деле это абсолютно пустые элементы.

Именно поэтому результат в браузерах настолько отличается. Браузеры как бы **выкручиваются** из сложившейся ситуации, применяя **line-height** к замещаемым элементам так, как им вздумается. И результат при этом может быть абсолютно непредсказуемым.

Можно лишь предположить, что, например, **Internet Explorer**, скорее всего отображает текст в `<input>` по **аналогии** с отображением настоящего контента, но и другие браузеры делают что-то подобное.

Единицы измерения в line-height

Значения **line-height** делятся на следующие четыре группы: **<число>** (просто безразмерное число), **'normal'**, **<длина>** (в любых **CSS-единицах длины**— **em**, **px**, **cm** и т.п.) и **проценты**.

Число

Число (или безразмерное число) в качестве значения **line-height** умножается на фактическое значение **font-size**, и тем самым получается нужная нам длина. Вот маленький пример:

Пример 7.4 Единицы измерения в line-height

CSS

```
h1 {font-size: 20px; line-height: 2}
```

Результат будет означать то же самое, что и **font-size: 20px; line-height: 40px**. Т.е. размер шрифта увеличился ровно вдвое. Соответственно, единица в **line-height: 1** будет идентична установленному размеру шрифта.

Так же числа могут быть дробными и положительными. Отрицательные значения не допустимы.

normal

Если вы не установите никакого значения в **line-height**, то по умолчанию оно установится в значение **"normal"**. Поэтому **"normal"** считается самым простым, но, в то же время, коварным значением.

В чём же его коварность? Дело в том, что это значение зависит от самого шрифта и от ОС/браузера. Поэтому может получиться, что в одном браузере это значение превратится в 1.1 (от размера шрифта), а в другом 1.2 или 1.3 с хвостиком. Следовательно, результат может быть совершенно непредсказуемым.

Какой же выход? Выход прост — явно задавать значения **line-height**, удобные для чтения. Например, **1.3** или **1.4**.

Длина

Длина указывается в любых единицах измерения, а затем пересчитывается в пиксели. Например, **1in = 2.54cm = 25.4mm = 72pt = 96px**. Разница с нормальной высотой делится пополам в виде **half-leading**ов. Это как раз тот случай, который мы уже разобрали в прошлых статьях.

Единицы **em** для **line-height** писать бессмысленно, потому что **line-height: 1.5em** и **line-height: 1.5** — одно и то же. Т.к. и там, и там **1.5** — множитель при **font-size**.

Проценты

Проценты — фактически тот же множитель к **font-size**, но умноженный на 100%. Т.е. **1.5em**, просто **1.5** и **150%** для **line-height** — одно и то же. А в остальном проценты точно так же пересчитываются в пиксели, как и обычная длина.

Родительский инлайновый бокс с дочерними инлайнами и разным line-height

Вложенный инлайн с line-height больше родительского

В ситуации, когда в инлайне находятся другие инлайны с **line-height**, больше, чем у родительского **инлайн-элемента**, родительский инлайн остается своей высоты, а вложенные выпирают из него, но если переносятся целиком – высота предыдущей строки определяется **line-height** родителя.

Рассмотрим простой пример:

Пример 7.5 Вложенный инлайн с line-height больше родительского

а)

HTML

```
<p>
  Просто текст
  <span>ВЫСОКИЙ ТЕКСТ<br><i>с маленьким вложением</i></span>
  и опять просто текст
</p>
```

б)

CSS

```
p {
  font-size: 14px;
  line-height: 20px;
  border: 1px solid #000;
  padding: 0 10px;
}

span {
  font-size: 20px;
  line-height: 20px;
  background: #f00;
}

i {
  font-size: 10px;
  line-height: 50px;
  padding: 19px 0;
  background: rgba(0, 255, 128, .5);
}
```

И результат:

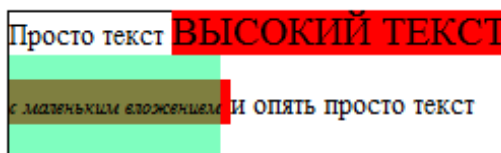


Рис. 7.6 Вложенный инлайн с line-height больше родительского

Перед нами абзац с текстом и инлайн-элементом **** (выделен красным), внутри которого содержится текст, элемент **
** (для разрыва строки) и инлайн-элемент **<i>**. Сам элемент **** вынужден разорваться на две части, последняя из которых включает в себя только лишь элемент **<i>**.

line-height span'a составляет **20px**, а **line-height** его дочернего инлайна **<i>** — **50px**, но при этом мы можем наблюдать следующую картину. Во второй части **span'a**, которая перенеслась на следующую строку, инлайн-элемент **<i>** своими **half-leading**'ами выпирает из своего родительского элемента (для наглядности я выставил вертикальные **padding'i**), но при этом **высота самого родительского** элемента не изменилась.

Суть в том, что инлайн сам по себе вообще никогда не растягивается по вертикали, потому что не умеет этого делать. У него нет ничего, что могло бы растягиваться от содержимого. Всё, что есть у инлайна –

это площадки и (возможно) **half-leading**'и. Ни то, ни другое — не контейнер, который может растягиваться содержимым.

В нашем случае получается следующее. Всё инлайновое содержимое **span**'а сидит на одной базовой проволоке, в смысле базовой линии. Но слева у нас сидят большие площадки с такими же **half-leading**'ми, а справа — маленькие с большими **half-leading**'ми.

Если наш **span** перенесется перед или после слова "ТЕКСТ" и займет два **лайн-бокса**, то высота первого будет **20px**, а второго — **50px** — по высоте наибольших инлайновых элементов. И то, что **<i>** является потомком того же инлайна, на высоту первого лайн-бокса не повлияет. А вот если хоть одна буква из слов "с маленьким вложением" останется на первой строке, то высота первого лайн-бокса окажется **50px**, потому что самой высокой там будет уже именно эта площадка со своими **half-leading**'ми.

Вывод таков. Контейнером для инлайнов выступает **лайн-бокс**, и глубина вложенности инлайнов не имеет никакого значения. Вложенные инлайны не влияют на высоту внешних инлайнов, и каждый лайн-бокс работает только со своими площадками, и неважно, кто их родитель и где он начинается. У инлайнов строго говоря вообще нет высоты как таковой, высота есть только у **лайн-бокса**, а у инлайнов — только **площадки**.

Вложенный инлайн с **line-height** меньше родительского

Но в ситуации, когда у вложенного инлайна **line-height** меньше, чем у родительского, и при этом он переносится на новую строку, то высота этой строки определяется **line-height**'ом родительского элемента, а не вложенного.

Снова рассмотрим на примере:

Пример 7.6 Вложенный инлайн с **line-height** меньше родительского

а)

HTML

```
<p>
  Просто текст
  <span>ВЫСОКИЙ ТЕКСТ<br><i>с маленьким вложением</i></span>
  и опять просто текст
</p>
```

б)

CSS

```
p {
  font-size: 14px;
  line-height: 20px;
  border: 1px solid #000;
}

span {
  font-size: 20px;
  line-height: 50px;
  padding: 15px 0;
  background: #f00;
}

i {
  font-size: 10px;
  line-height: 20px;
  padding: 5px 0;
  background: rgba(0,255,128,.5);
}
```

И результат:

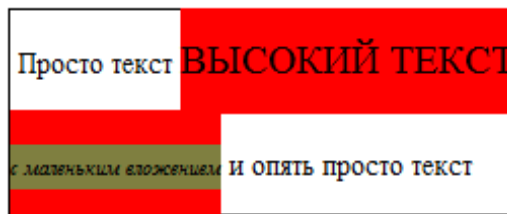


Рис. 7.7 Вложенный инлайн с line-height меньше родительского

Саму структуру описывать не имеет смысла. Она ничем не отличается от предыдущего примера. Единственное, что поменялось – это некоторые стили, а точнее **line-height span'a** и вложенного в него инлайна **<i>**. Их **line-height** поменялся местами и теперь **line-height span'a** стал равен **50px**, а вот у **<i>** — **20px**.

Здесь стоит обратить внимание на поведение вложенного инлайна **<i>**, который из-за маленького **line-height'a** болтается внутри своего родителя. Несмотря на то, что **<i>**, по сути, единственный в **span'e**, кто перенёсся вместе с ним на следующую строку, высота последней строки не изменилась, а осталась родительской. Спецификация объясняет это поведение так:

*Высота инлайн-бокса включает в себя все глифы и их **half-leading**'и сверху и снизу, и тем самым является в точности **line-height**'ом. Боксы дочерних элементов не влияют на эту высоту.*

Другими словами, если высота потомка выше, она выпирает и влияет на высоту **лайн-бокса**, но не на высоту инлайна, а если она меньше, то она просто болтается внутри и не влияет вообще ни на что.

Что ещё следует знать?

Если в элементе есть текст разными шрифтами, **line-height** высчитывается по наибольшему из них. Это в основном бывает, когда нужного символа не оказывается в заданном шрифте, в этом случае браузер по спецификации имеет право поискать его в дефолтном, и так может получиться, что часть надписи – модным шрифтом, а часть – обычным, хотя это одна и та же текстовая нода.

ИКФ: Вертикальное выравнивание в строке, часть 1 (8-я публикация цикла “Тайны CSS2.1”)

Добавлено 19 Июль 2012 | Раздел: [CSS](#), [Статьи](#), [Тайны CSS2.1](#)

В прошлых статьях мы узнали, что такое базовая линия и высота строки (**line-height**), но на этом история далеко не закончена. Помимо базовой линии есть и другие части лайн-бокса, по которым может происходить выравнивание инлайн-боксов или обычного текста, а, следовательно, от этого может меняться и сама высота строки.

В следующем цикле статей мы постараемся досконально углубиться в работу **vertical-align**, разобрать поведение этого свойства и способы его влияния на высоту строки.

Страт (Strut)

Прежде чем переходить к вертикальному выравниванию (**vertical-align**), нам необходимо познакомиться с ещё одной важной штукой под названием "[Страт](#)". Несмотря на то, что в прошлых статьях мы, хоть и косвенно, но уже касались этого понятия (чуть позже вы поймёте почему), я посчитал, что полное знакомство с ним будет своевременным именно с изучением вертикального выравнивания.

В блоке всегда имеется "дефолтный" текстовый потомок, задающий шрифтовые параметры по умолчанию (в т.ч. **line-height**) для инлайнового содержимого этого блока. Если в блоке есть обычный текст, то в роли "дефолтного" текстового потомка выступает порождаемый этим текстом анонимный инлайновый бокс. Если же текста нет, то форматирование идет так, как если бы он был, как будто такой инлайновый бокс (с теми же свойствами) есть. Вот этот воображаемый инлайн и называется стратом – "**Strut**" (название навеяно редактором [TeX](#)).

Чтобы не запутать вас окончательно, приведу, пожалуй один наглядный пример:

Пример 8.1 Strut

а)

HTML

```
<div class="withoutText">
  <span></span>
</div>
```

б)

CSS

```
div {
  border: 1px solid #000;
  padding: 0 10px;
  font-size: 20px;
  background: #F90;
  width: 150px;
}

span {
  display: inline-block;
  width: 70px;
  height: 70px;
  background: #900;
}
```

И сразу [результат](#):

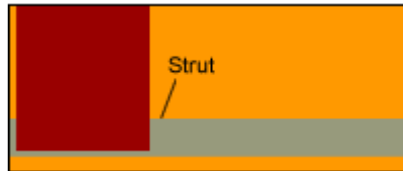


Рис 8.1 Strut

Итак, есть блок, в котором находится элемент **** (красный квадрат). Натурального текста в блоке нет, **но** вместо него там находится "**воображаемый бокс**", который и является нашим **Strut'ом** (на картинке я специально сделал его видимым). Т.е. текста в блоке нет, но он всё равно условно как бы есть. Так же, как и у обычного текста, у страта есть базовая линия, и его высота высчитывается метриками шрифта, о которых мы говорили в прошлых статьях.

Высота ****'а (инлайн-блока) заставляет лайн-бокс растягиваться, но при этом высота **Strut'a** не меняется. Вместо того, чтобы растянуться, он поднимается или опускается в зависимости от базовой линии самого блока, т.к. его базовая линия привязана к общей базовой линии строки.

Vertical-align

line-height оказывает влияние на высоту строки, но, надо признать, что это свойство скудно по своим возможностям. В **CSS** есть другое средство, которое намного интереснее. Представляю вам главного гостя на сегодняшнем вечере — **vertical-align**.

vertical-align — свойство, способное влиять на высоту лайн-бокса путём перемещения элементов строки по вертикали. Это свойство может передвигать элементы, заставляя одни влиять на координаты других, а в целом — на высоту строки.

Но всё же, давайте по порядку.

baseline

Вся сила **vertical-align** заключается в его значениях. Именно благодаря разнообразию последних **vertical-align** и считается мощным оружием. По умолчанию **vertical-align** имеет значение **baseline** (базовая линия), но особого интереса оно собой не представляет. Дело в том, что изначально все элементы в строке и так выравниваются по базовой линии (см. предыдущие статьи), поэтому **baseline** не сможет как-то изменить их координаты. Просто держите это значение в уме, как по умолчанию.

middle

Вот здесь уже интереснее. Значение **middle** выравнивает середину бокса по середине строчных букв текста. Таким образом, "родительская" базовая линия оказывается на половину высоты строчных букв (так называемой "x-высоты", ей в CSS соответствует единица `ex`) ниже середины нашего бокса.

Сразу перейдём к примеру:

Пример 8.2 vertical-align: middle

а)

HTML

```
<p>xxx <span>100x100</span> xxx</p>
```

б)

CSS

```
p {  
  border: 1px solid #0a0;  
}  
  
span {  
  display: inline-block;  
  width: 100px;  
  height: 100px;  
  background: #aa0;  
  vertical-align: middle;  
}
```

И результат:

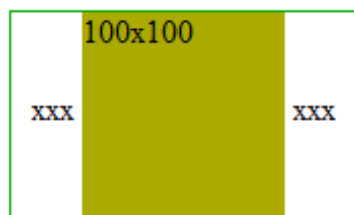


Рис. 8.2 vertical-align: middle

Нам уже известно, что высоту строки определяют самые высокие боксы в ней, поэтому наш инлайн-блок растянул строку ровно на **100px**. У инлайн-блока стоит **vertical-align** в значении **middle**. Но, т.к. общая базовая линия идёт по нижнему краю букв "x", а положение середины элемента зафиксировано относительно положения базовой линии текста (на половину высоты строчных букв выше нее), то этой базовой линии приходится сместиться, подстраиваясь под самый высокий бокс.

Если говорить точнее, высота строки сейчас задается инлайн-блоком, но базовая линия абзаца привязана к отметке **50px + 0.5ex** от верха (или **50px – 0.5ex** от низа). Инлайн-блок ("высокая птица") немного подтянул базовую линию ("проволоку") как ему удобнее, но при этом остался к ней привязанным.

sub

Значение **sub** опускает базовую линию бокса на уровень, отведенный для нижних индексов родительского бокса (не влияя на размер шрифта элемента).

Пример:

Пример 8.3 vertical-align: sub

а)

HTML

```
<p><span>xxx_</span><i>_100x100_</i><span>_xxx</span></p>
```

б)

CSS

```
p {  
    border: 1px solid #0a0;  
    padding: 0 10px;  
}  
  
span {  
    background: #88f;  
}  
  
i {  
    display: inline-block;  
    background: #aa0;  
    vertical-align: sub;  
}
```

Результат:

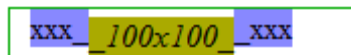


Рис. 8.3 vertical-align: sub

Я поставил подчёркивания, чтобы вы могли видеть базовую линию. Она осталась на своём месте, а вот элемент `<i>`, у которого стоит **vertical-align: sub** – опустился вниз на отведённое место для нижних индексов.

Что есть "**Место нижних индексов**"? Это уровень, отведённый для подстрочных индексов (как "двойка" в H_2O), которые в **HTML** оформляются тегом `<sub>`. Точное значение их местоположения спецификация не определяет, так что браузеры и ОС имеют право на "немного самостоятельности" (а иногда и метрики шрифта могут на него повлиять).

super

Значение **super** аналогично **sub**, только на место для верхних индексов (как в $E=mc^2$, в **HTML** — `<sup>`).

text-top

Значение **text-top** выравнивает верхний край инлайн-бокса по верхнему краю **области контента** предка.

Пример 8.4 vertical-align: text-top

а)

HTML

```
<p><span>xxx_</span><i>_100x100_</i><span>_xxx</span></p>
```

б)

CSS

```
p {  
  border: 1px solid #0a0;  
  font-size: 32px;  
  padding: 0 10px;  
}  
  
  span {  
    background: #88f;  
  }  
  i {  
    display: inline-block;  
    background: #aa0;  
    font-size: 16px;  
    vertical-align: text-top;  
  }
```

И результат:

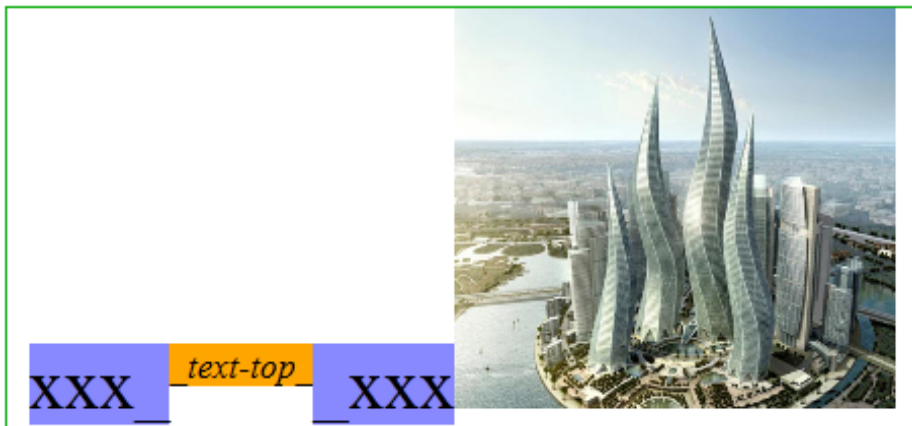


Рис. 8.4 vertical-align: text-top

Я нарочно поместил в лайн-бокс картинку, чтобы вы могли увидеть ту самую **контентную область** (которая не зависит от высоты строки), по верху которой и равняется элемент с **text-top** (оранжевый бокс в нашем случае). Это та самая **контентная часть**, которая определяет **высоту строки по умолчанию**. Браузер может, например, брать размер кегельной площадки или координаты максимальных верхних и нижних выносных элементов символов шрифта (см. предыдущие статьи).

Картинка лишь определила высоту строки, но высота **контентной части** при этом не изменилась. Картинка — это атомарный (неразрывный) инлайновый бокс со своими внутренними размерами и размер текста ей не указ, т.к. она не является буквой шрифта. Метрики шрифта родительского элемента влияют лишь на положение ее нижнего края, "стоящего" на базовой линии.

Вывод таков, что элементов с **vertical-align: text-top** не интересует верхний край строки, им важен **верх области текста**, и выравниваются они именно по нему.

text-bottom

Значение **text-bottom** аналогично **text-top**, только в отличие от последнего, выравнивает нижний край инлайн-бокса по нижнему краю **области контента** предка.

Важное примечание

Выше мы описали, как должно происходить форматирование строки согласно [спецификации CSS 2.1](#). Нужно ли говорить, что в реальности всё не совсем так, и браузеры (даже с виду приличные) часто имеют своё "особое мнение". Но об этих выкрутасах отдельных браузеров мы расскажем уже в следующей части нашего цикла.

ИКФ: Вертикальное выравнивание в строке, часть 2 (9-я публикация цикла “Тайны CSS2.1”)

Добавлено 27 Июль 2012 | Раздел: [CSS](#), [Статьи](#), [Тайны CSS2.1](#)

top и bottom

Если с вышеупомянутыми значениями было всё достаточно прозрачно, то с **top** и **bottom** не всё так просто. У этих значений есть особая черта, которой нет ни у одного другого значения **vertical-align**. Инлайн-боксы с **vertical-align** и значениями **top** или **bottom** способны отвязывать себя от базовой линии.

Но всё же, давайте по порядку.

В спецификации сказано:

*Следующие значения (**top** и **bottom**) выравнивают элемент по отношению к лайн-боксу. Поскольку у элемента могут быть потомки, выравненные относительно него самого (а у них — в свою очередь, тоже такие потомки), эти значения применяются к границам всей ветки DOM-дерева, к которой применяется выравнивание (**aligned subtree**, букв. "выравниваемое поддерево"). Это выравниваемое поддерево инлайнового элемента содержит сам элемент и выравниваемые поддерева всех его потомков с действующим значением '**vertical-align**' не '**top**' и не '**bottom**'. Верх выравниваемого поддерева — самый верхний из верхних краев боксов в поддереве, низ — аналогично (т.е. самый нижний из нижних краев)*

Проще говоря, при **top** и **bottom** положение таких элементов, которые могут иметь свой лайн-бокс (например, инлайн-блок), привязано только к границам внешнего лайн-бокса и они игнорируют общую базовую линию. Внутри такого элемента действуют те же законы инлайн-форматирования, и абсолютно не важно, какое в нём содержимое, нам важна только **его итоговая высота**.

Чтобы окончательно разобраться в этой ситуации, предлагаю посмотреть один примерчик:

Пример 9.1 vertical-align top и bottom

а)

HTML

```
<p>
  Внешний текст:
  <span class="one">
    strut 1
    <i>Раз</i>
    <i>Два</i>
  </span>
  <span class="two">
    strut 2
    <i>Раз</i>
    <i>Два</i>
    <i>Три</i>
  </span>
</p>
```

б)

CSS

```
p {border: 1px solid blue;}
  span, i {
    display: inline-block;
    border: 1px solid red;
    vertical-align: bottom;
  }

  .one > i {
    height: 60px;
    border: 1px solid blue;
    vertical-align: top;
  }
  .one > i + i {
    vertical-align: bottom;
    height: 100px;
  }
  .two > i {
    height: 100px;
    border: 1px solid green;
    vertical-align: baseline;
  }
  .two > i + i {
    vertical-align: text-top;
  }
  .two > i + i + i {
    vertical-align: middle;
  }
```

И результат:

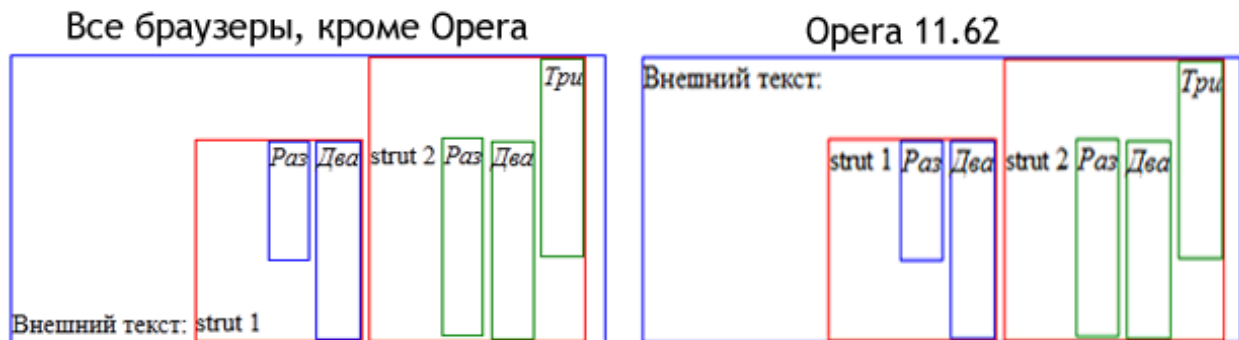


Рис. 9.1 vertical-align top и bottom

Я немного усложнил пример, но это было необходимо для того, чтобы ничего не упустить. Поэтому будем разбираться очень подробно.

Два изображения. На первом показано поведение всех браузеров, кроме **Opera**, а на втором уже самой **Opera** (В **Chrome** и **Safari** есть небольшое отличие из-за интересного бага, но оно не относится к нашей теме, поэтому об этом позже). Я специально сделал скриншот из **Opera**, чтобы можно было увидеть отличие в отображении внешнего текста в блоках, в которых содержатся элементы с **vertical-align: top** или **bottom**. В большинстве браузеров внешний текст прижат к низу своих контейнеров, но в **Opera** кверху.

А теперь по порядку.

У нас есть блочный контейнер (**<p>**). Внутри него я поместил "Внешний текст" и два инлайн-блока (****), каждый из которых выровнен по низу лайн-бокса при помощи **vertical-align: bottom**. Эти элементы растягивают контейнер по высоте самого высокого из них, в данном случае — по высоте второго элемента. Аналогичное поведение происходит и в первом инлайн-блоке, с единственным отличием, что элементы

внутри него прижаты к верху и низу соответственно.

На данном этапе стоит обратить внимание на то, как в хаотичном порядке выравнивается сам текст в параграфе и в его первом элементе. В **Opera** он прижат к верху, а в остальных браузерах, напротив, к низу. На самом деле здесь нет ни какой ошибки, просто в этих контейнерах "гуляет" сама базовая линия, так как она ни к чему не привязана.

Вот что по этому поводу примерно написано в спецификации:

Когда боксы, "прибитые" к "потолку" или к "полу", выше, чем чисто текстовые, последние браузер может "прибивать" туда, куда ему удобнее.

Выходит, что **Opera** "прибивает" к "потолку", остальные к "полу". И никто не нарушает спецификацию. Ведь если самые высокие боксы в строке сорваны с базовой линии (у них **vertical-align top** или **bottom**), то сама эта базовая линия ни к чему не привязана и может находиться где угодно, главное, чтобы из лайн-бокса ничего не торчало наружу.

Кстати, обратите внимание на основной текст в инлайн-блоках, который я специально обозначил как "**strut 1**" и "**strut 2**". Этим самым я хотел показать, что если бы в элементах не было бы текста, то в нём обязательно находился бы воображаемый бокс (**strut**), который выравнивался бы по основной базовой линии в боксах.

Ну и последний инлайн-блок в параграфе содержит элементы, **vertical-align** которых не имеет значений **top** или **bottom** и, соответственно, никто из них не сорван с базовой линии и выравнивается уже относительно неё. Этот случай не стоит отдельного разбирания, т.к. всё из чего он состоит, мы уже обсуждали в этой статье. Попробуйте пока сами проанализировать ситуацию и разобраться, что происходит в блоке. А позже мы обязательно разберём сложные и, возможно, не совсем очевидные случаи.

Занятный баг в Webkit

Ну и напоследок, в этой части рассказа, я бы хотел рассмотреть один занятный баг в браузерах на базе движка **Webkit** (Chrome и Safari). Он не относится напрямую к **vertical-align top** или **bottom**, а, скорее, к наследованию **vertical-align**.

Чтобы было понятно, о чём идёт речь, сразу начну с примера, код которого один в один совпадает с кодом вышеразобранной ситуации, но немного отличается своим результатом. Выше я приводил два скриншота, но нарочно не стал показывать скриншот из **Chrome** или **Safari**, потому что это не относилось к теме. А вот теперь другое дело, смотрим:

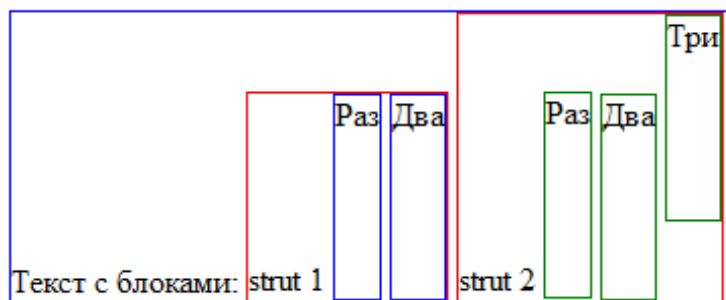


Рис. 9.2 Занятный баг в Webkit

Здесь стоит обратить внимание на последний блок, текст ("**strut**") которого выровнен по нижнему краю. Это поведение является неверным по той простой причине, что **vertical-align не наследуемое свойство**, но вот **Webkit** с этим не согласен, и вместо того, чтобы выравнивать сам инлайн-блок (с красной рамкой) в его родительском лайн-боксе, он вторгается и во внутренний лайн-бокс потомка. Интересно, что баг действует только на анонимный инлайн-бокс (текст "**strut 2**"), остальные потомки этого инлайн-блока — вложенные инлайн-блоки с зеленой рамкой — выравниваются по вертикали как обычно.

Вот такие вот весёлые баги встречаются порой в разных браузерах. Так что учитывайте это.

Абсолютные единицы

В отличие от остальных значений, с абсолютными единицами в **vertical-align** всё намного проще. В зависимости от числа и единицы измерения бокс поднимается (положительное значение) или опускается (отрицательное) на выставленное расстояние.

Рассмотрим простой пример:

Пример 9.2 Абсолютные единицы в vertical-align

а)

HTML

```
<p>
  <span>baseline</span>
  <em>16px</em>
  <i>1em</i>
</p>
```

б)

CSS

```
p {
  border: 1px solid #0a0;
  font-size: 32px;
}
span {
  vertical-align: baseline;
  background: #cc8;
}
em {
  vertical-align: 16px;
  background: #F93;
}
i {
  vertical-align: 1em;
  background: #FC0;
}
```

И результат:

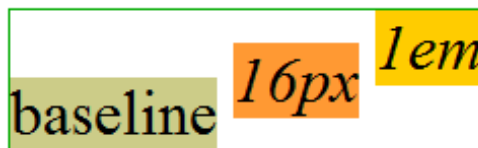


Рис. 9.3 Абсолютные единицы в vertical-align

В блоке находятся три элемента. Первый из них (``) стоит на общей базовой линии, как по умолчанию. Второй элемент (``) визуально сместился* ровно на **16px** (именно такое значение у него и выставлено). Ну, а у третьего элемента (`<i>`) выставлено значение **1em**, и, так как единицы измерения **em** высчитываются относительно размера родительского шрифта (а последний у нас равен **32px**), то **1em** можно пересчитать в **32px**. Выходит, что `<i>` *сместился на **32px**.

Точно так же происходит и с **отрицательными абсолютными единицами**, только в этом случае элементы уже **опускаются вниз** на выставленное значение. Рассматривать поведение отрицательных значений не имеет смысла из-за его явной очевидности.

Да, и не забываем, что боксы раскладываются сверху вниз, поэтому поднятие элементов при положительных значениях происходит **только визуально, и что на самом деле опускается именно **общая базовая линия**.*

Проценты

Проценты поднимают (положительное значение) или опускают (отрицательное) бокс на расстояние в процентах от значения **line-height**. А если нет явно заданного значения, то последнее берётся от наших старых знакомых – метрик шрифта. **0%** - то же самое, что **baseline**.

Положительные значения

Надо заметить, что в браузерах есть отличия в поведении боксов, как с положительными, так и с отрицательными значениями, а в **Chrome** и в **Safari** даже существует баг, который мы оставим на закуску.

Первый пример (общая высота шрифта):

Пример 9.3 Проценты в vertical-align (положительные значения, общая высота шрифта)

а)

HTML

```
<p>
  <span>xxxL</span>
  <i>xx 100% xx</i>
  <em>xx 33% xx</em>
  <span>Jxxx</span>
</p>
```

б)

CSS

```
p {
  border: 1px solid #0a0;
  font-size: 32px;
}
span {
  background: #88f;
}
i {
  background: #aa0;
  vertical-align: 100%;
}
em {
  background: #aa0;
  vertical-align: 33%;
}
```

Результат таков:

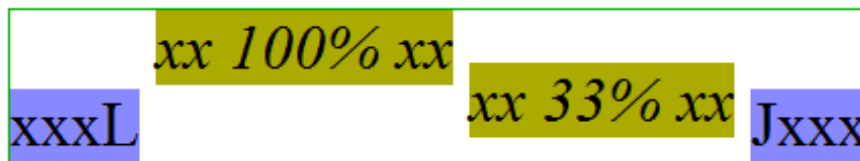


Рис. 9.4 Проценты в vertical-align (положительные значения, общая высота шрифта)

В данном случае во всех браузерах будет идентичная картинка. Здесь высота самих инлайн-боксов отсчитывается от общей высоты шрифта и составляет порядка **38px**. Второму элементу (`<i>xx 100% xx</i>`) мы выставили положительное значение в 100%, что автоматически подняло его наверх ровно на свою высоту (**38px**). А вот третий инлайн-бокс (`xx 33% xx`) мы подняли на **33%**, что составило примерно **12px**.

Сам лайн-бокс растянулся до суммарной высоты всех передвижений внутри себя, и визуально может показаться, что элемент `<i>` встал своим **нижним краем** на бывший **верхний край**, но на самом деле правильно было бы сказать, что **базовая линия** элемента `<i>` в итоге оказалась выше **базовой линии** окружающего нормального текста на 100% процентов от своей высоты. А поскольку боксы раскладываются сверху вниз, то, по факту, **у-координата** увеличилась как раз у боксов с основным текстом (``), а остальные элементы остались на своих местах.

Второй пример (своя высота шрифта):

Если результат с общей высотой шрифта идентичен во всех браузерах, то с конкретно заданным шрифтом элементу всё обстоит немного иначе. Смотрим пример:

Пример 9.4 Проценты в vertical-align (положительные значения, своя высота шрифта)
а)

HTML

```
<p>
  <span>xxxL</span>
  <i>xx 100% xx</i>
  <em>xx 33% xx</em>
  <span>Jxxx</span>
</p>
```

б)

CSS

```
p {
  border: 1px solid #0a0;
  font-size: 32px;
}
span {
  background: #88f;
}
i {
  background: #aa0;
  vertical-align: 100%;
  font-size: 50px;
}
em {
  background: #aa0;
  vertical-align: 33%;
}
```

И результат:

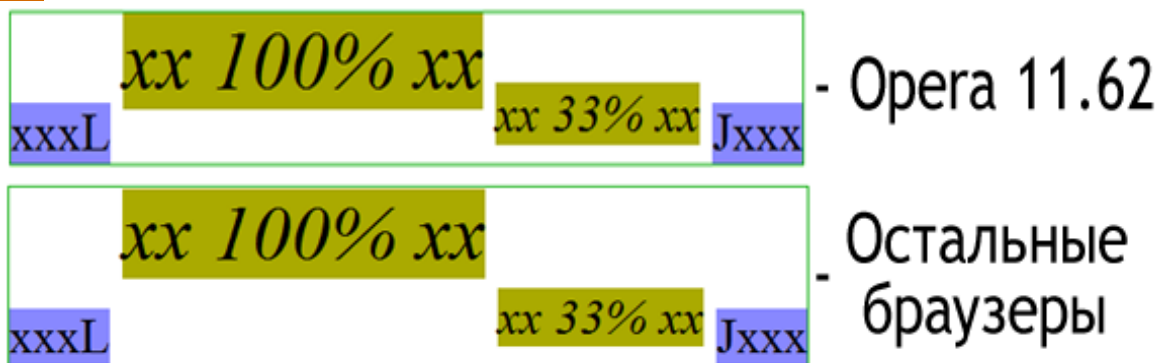


Рис. 9.5 Проценты в vertical-align (положительные значения, своя высота шрифта)

Единственное, что я сделал с кодом – это установил у второго элемента (`<i>xx 100% xx</i>`) размер шрифта (**50px**). Этого вполне было достаточно, чтобы заставить браузер **Opera** вести себя немного иначе, в отличие от своих "собратьев". Вместо того, чтобы сместиться на итоговую высоту элемента `<i>`, нижние элементы изменили свою вертикальную координату лишь на половину того, что от них требовалось. Скорее всего, это поведение можно отнести к **bug** **Opera**, потому что, во-первых, **Opera** - это единственный браузер с таким результатом, а во-вторых, в таком поведении не видно логики.

Вывод таков, что в зависимости от задания шрифта и его размера конкретным элементам, поведение браузеров может отличаться и, несмотря на то, что в данном случае так себя проявила только **Opera**, не факт, что при иных раскладах точно так же себя не проявит и какой-нибудь другой браузер.

Отрицательные значения

С отрицательными процентными значениями у **vertical-align** ситуация зеркально отличается от положительных процентных значений. Только в данном случае боксы смещаются **вниз** на расстояние в процентах от значения **line-height**, если таковой имеется, а если же нет, то, как обычно – от метрик шрифта, опять же – общего или заданного напрямую самому элементу.

Предлагаю просто посмотреть пример и двинуться дальше.

Пример 9.5 Проценты в vertical-align (отрицательные значения)

а)

HTML

```
<p>
  <span>xxxL</span>
  <i>xx -100% xx</i>
  <em>xx -33% xx</em>
  <span>Jxxx</span>
</p>
```

б)

CSS

```
p {
  border: 1px solid #0a0;
  font-size: 32px;
}
span {
  background: #88f;
}
i {
  background: #aa0;
  vertical-align: -100%;
}
em {
  background: #aa0;
  vertical-align: -33%;
}
```

И результат:

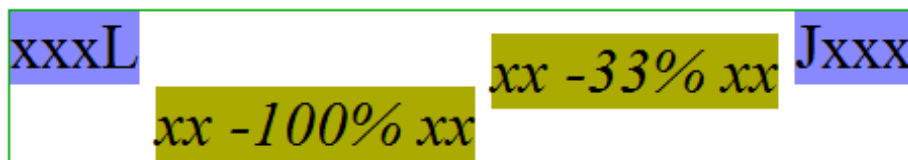


Рис. 9.6 Проценты в vertical-align (отрицательные значения)

За основу я взял первый случай с общей высотой шрифта, и заменил положительные процентные значения **vertical-align** на отрицательные. Ситуация зеркально изменилась, и теперь второй и третий боксы уже сместились **вниз**, а не заставили смещаться окружающие элементы, как в случае с положительными значениями.

Баг Webkit

В целом процентные значения с **vertical-align** в браузерах на базе движка **Webkit (Chrome и Safari)** работают нормально, но это до тех пор, пока дело не касается **замещаемых элементов**, таких как ****, **<input>** и т.д. Применение к ним **vertical-align** в процентах делает поведение этих элементов довольно таки занятным и отличным от остальных браузеров.

Сразу приведу пример:

Пример 9.6 Баг Webkit с процентными значениями vertical-align

а)

HTML

```
<p>
  <span>xxxL</span>
  <i>xx 100% xx</i>
  <em>xx 33% xx</em>
  <span>Jxxx</span>
  
</p>
```

б)

CSS

```
p {
  border: 1px solid #0a0;
  font-size: 32px;
}

span {
  background: #88f;
}

i {
  background: #aa0;
  vertical-align: 100%;
}

em {
  background: #aa0;
  vertical-align: 33%;
}

img {
  width: 100px;
  height: 100px;
  vertical-align: 100%;
}
```


Результат:

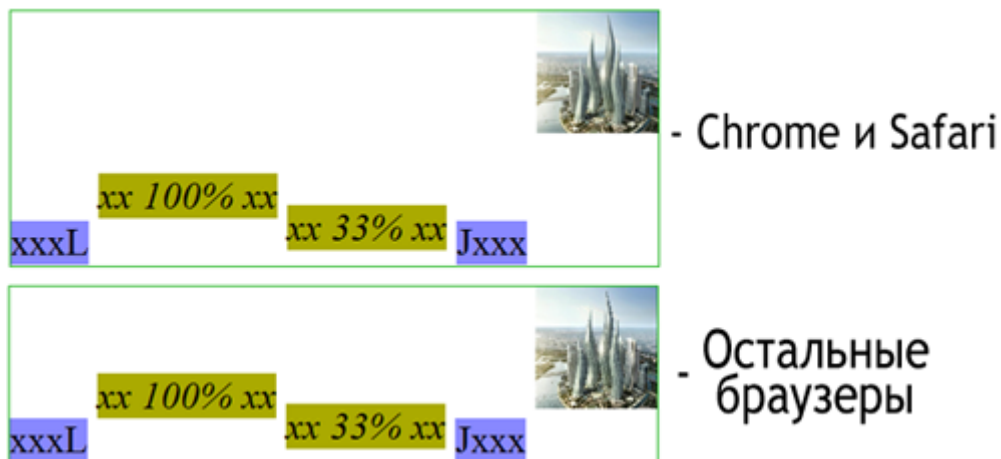


Рис. 9.7 Бар Webkit с процентными значениями `vertical-align`

В качестве испытуемого я подобрал картинку с габаритами **100x100px** (но не забываем, что это касается любых замещаемых элементов). Назначив ей **`vertical-align: 100%`** мы заставили остальные элементы сместиться вниз на определённое расстояние. Но на какое именно? Вопрос именно в этом, потому что в большинстве браузеров элементы сместились на высоту метрик основного шрифта, а вот в **Chrome** и **Safari** на высоту самой картинки.

Кто прав, а кто не прав? Давайте разбираться. Как мы уже поняли, проценты в **`vertical-align`** высчитываются от значения **`line-height`** либо от метрик самого шрифта, но никак не от высоты лайн-бокса. Именно это мы и видим в большинстве браузеров. **WebKit** же включил в `100%` высоту самой картинки. Пожалуй, в этом есть определенная логика (ведь именно от высоты картинки зависит фактическая высота лайн-бокса), но, к сожалению, спецификации эта логика не соответствует.

ИКФ: Вертикальное выравнивание в строке, часть 3 (10-я публикация цикла “Тайны CSS2.1”)

Добавлено 03 Авг 2012 | Раздел: [CSS](#), [Статьи](#), [Тайны CSS2.1](#)

vertical-align + line-height

Если вы внимательно читали предыдущие статьи, то уже наверняка поняли, как по отдельности работают **vertical-align** и **line-height**. Но как эти свойства взаимодействуют между собой? Краткий ответ – практически никак: их работа напрямую не пересекается. Задача **vertical-align** - сдвигать базовую линию, на сколько велено, а задача **line-height** (благодаря **half-leading**'ам) — отталкивать чужие лайн-боксы от этой базовой линии сверх метрик шрифта.

Детальную картину предлагаю разобрать на следующем примере:

Пример 10.1 vertical-align + line-height

а)

HTML

```
<p>
  <span>Раз</span>
  <span>два</span>
  <span>три</span>
  <span class="four">четыре</span>
  <span>пять</span>
</p>
```

б)

CSS

```
p {
  border: 1px solid #0a0;
  font-size: 32px;
}

span {
  vertical-align: baseline;
  background: #F90;
}

.four {
  line-height: 200px;
  vertical-align: 50px;
}
```

И результат:

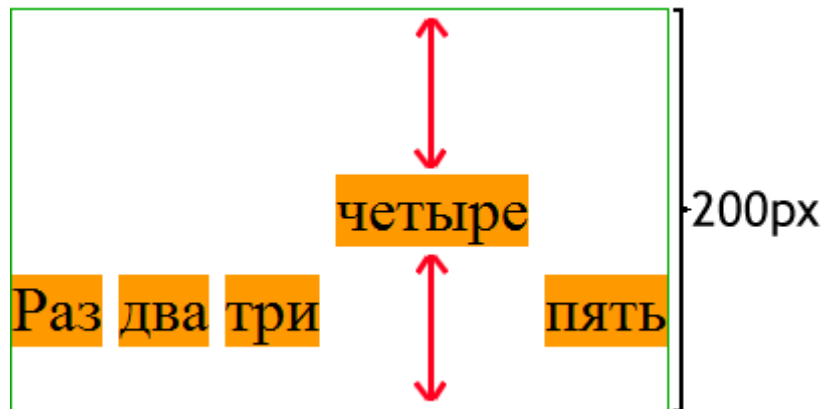


Рис. 10.1 vertical-align + line-height

Перед нами абзац с пятью инлайн-элементами, каждый из которых, по умолчанию, выстроен по общей базовой линии. Но, так как четвёртый инлайн-блок, благодаря большому **line-height** (а соответственно **half-leading**'ам), самый высокий элемент в строке, остальные элементы вынуждены подстраиваться именно под него. Выходит, что именно четвёртый элемент рассказывает другим, где будет находиться общая базовая линия. На данный момент всё просто и уже знакомо, поэтому перейдём к главному.

Здесь стоит обратить внимание на следующие вещи. На смещение общей базовой линии (и, конечно же, вместе с ней всех элементов, кроме четвёртого) после того, как мы добавили к четвёртому элементу (с **line-height**) **vertical-align** в значении **50px** и на то, как после этих действий совершенно не изменилась высота строки. И, главное, на сам четвёртый элемент, который в итоге этих взаимодействий остался на своём месте и его **half-leading**'и (помеченные красными стрелками) не изменились.

Суть в том, что **vertical-align** ничего не знает о **line-height**, а тот об **vertical-align**. Их вклад в итоговую высоту строки складывается уже по итогу, если необходимо. **vertical-align** отвечает за смещение инлайн-боксов, которые могут ездить вверх-вниз со всем, что у них есть - **leading**'и, и т.п. Если край их **leading**'ов при этом упрётся в край лайн-бокса – лайн-бокс начнет растягиваться, а если не упрётся – ничего не произойдет, просто боксы сместятся друг относительно дружки.

Разные значения vertical-align + line-height

Главными вещами при взаимодействии **vertical-align** и **line-height** являются итоговая вертикальная координата элементов и конечная высота строки. Как в первом, так и во втором случаях результат может зависеть от разных значений **vertical-align**, из-за которых может поменяться начальная координата элементов и точка отсчёта самого **line-height**. И, если с абсолютными единицами в этом плане всё в принципе понятно, то результат от применения некоторых других значений может оказаться уже не таким очевидным. Предлагаю посмотреть наглядный пример.

text-top + line-height

Пример 10.2 text-top + line-height

а)

HTML

```
<p>
  <span>xxx_</span>
  <i>_100x100_</i>
  <span>_xxx</span>
  
</p>
```

б)

CSS

```
p {  
  border: 1px solid #0a0;  
  font-size: 32px;  
  padding: 0 10px;  
}  
  
span {  
  background: #88f;  
}  
  
i {  
  background: #aa0;  
  font-size: 16px;  
  vertical-align: text-top;  
  line-height: 100px;  
}
```

И результат:

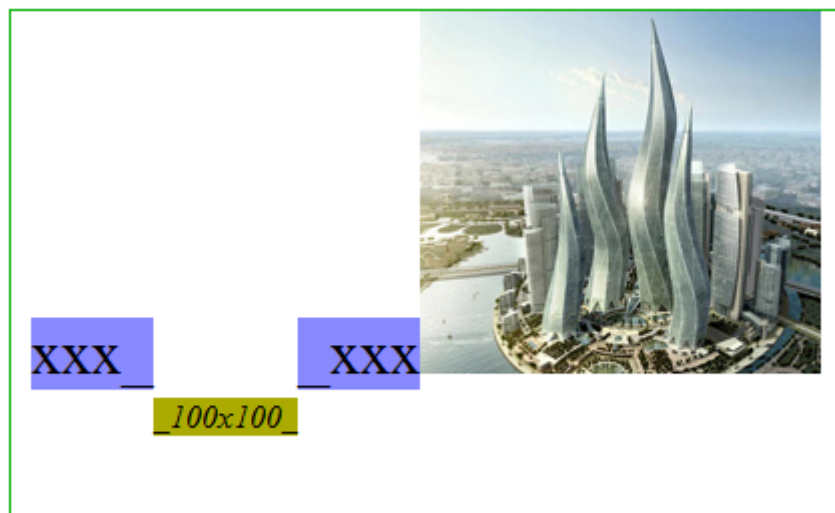


Рис. 10.2 text-top + line-height

Уже знакомое нам значение **vertical-align: text-top**, выставленное элементу `<i>`, заставило его верхний край примыкать к верхней области самого контента. Эту ситуацию мы уже разбирали, поэтому ничего нового тут нет. Я прошу лишь обратить внимание на то, откуда начал свой отсчёт **leading** элемента `<i>`, после того, как мы добавили последнему свойство **line-height** в значении **100px**. Начало его **leading**'а приходится на верхний край контентной части, которое совпадает с начальной координатой значения **text-top**.

Это типичная ситуация того, как после взаимодействия **vertical-align** и **line-height** вертикальная координата элементов и результирующая высота строки зависит от их значений. После всех манипуляций элемент `<i>` оторвался от верхнего края контентной части из-за верхнего **half-leading**'а, а уже нижним **half-leading**'ом растянул строку.

Чтобы увидеть другие варианты, я сделал специальную страницу-пример: [vertical-align + line-height](#)

ИКФ: Горизонтальное выравнивание, часть 1 (11-я публикация цикла “Тайны CSS2.1”)

Добавлено 14 Авг 2012 | Раздел: [CSS](#), [Статьи](#), [Тайны CSS2.1](#)

Итак, мы наконец-то подобрались к завершающей части рассказа про строчное форматирование. Мы уже отлично понимаем, как работает вертикальное выравнивание, и от чего зависит высота строки, но для полного понимания форматирования строк этого недостаточно. Ведь, если есть вертикальное выравнивание, то должно быть и горизонтальное.

В следующих статьях мы постараемся разобрать, что из себя представляют: ширина кегельных площадок, расстояния между ними, а так же разные способы и алгоритмы влияния на эти межсловные и межсимвольные пробелы.

Ширина литеры

Как мы уже выяснили, высота литер определяется вертикальными метриками шрифта. А как насчет ширины? Для этого у шрифта есть и горизонтальная метрика — авансовая ширина. Почему авансовая? Потому что при размещении каждой буквы она заранее (авансом) сообщает рендереру, где ему нужно будет разместить следующую.

В каждую ширину литеры входят всего две вещи – сам символ (ширина его глифа) и полуапроши.

Следующий пример хорошо иллюстрирует эту картину.

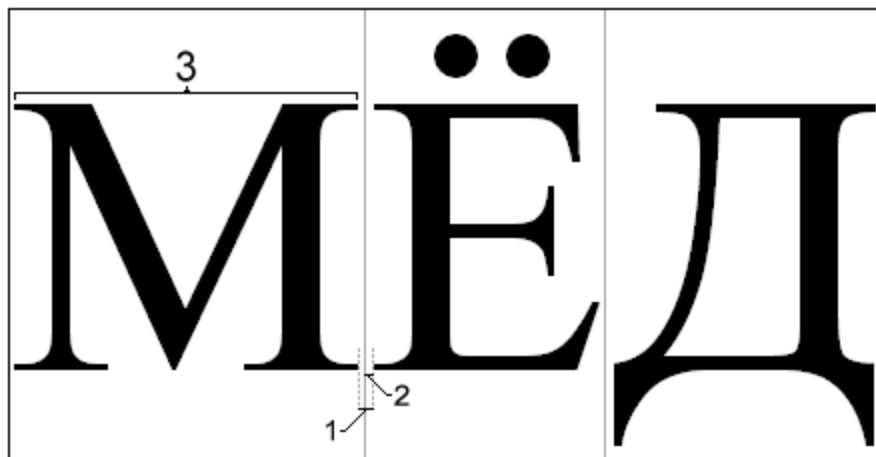


Рис. 11.1 Ширина литеры

Перед нами три площадки, в первой из которых я сделал спец. обозначения нужных нам частей. Цифрой "1" обозначается **апрош** - расстояние между контурами символов (в нашем случае "М" и "Ё") соседних площадок. Соответственно часть под цифрой "2" называется **полуапрош** - половина расстояния между глифами символов (или **половина апроша**). Ну и под цифрой "3" показана **ширина глифа** (ширина контура самого символа).

Выпирающие глифы

Несмотря на то, что горизонтальная метрика литеры определяет её ширину, это не означает, что сами контуры символов не могут быть шире своей площадки. На самом деле горизонтальные контуры символов могут вылезать из своих площадок, при этом фактическая ширина самих площадок не изменится.

Рассмотрим на примере:



Рис. 11.2 Выпирающие глифы

На рисунке хорошо видно, как контуры символов налезают на соседние литеры, а некоторые и вовсе нагло выпирают аж на пол-слова. Как правило, такое может происходить в дизайнерских или рукописных шрифтах. В основном, именно там глифы самих символов могут быть намного шире, чем их площадки.

Да, вы не ослышались! **Авансовая ширина может быть меньше ширины фактического контура символа**, поэтому возможна ситуация, когда хвостик от одной буквы еще тянется, а уже началась другая.

Лигатуры

Ширина литер, о которых мы говорили чуть ранее, не всегда может включать в себя принципиально один символ. Бывают ситуации, когда в зависимости от шрифта площадки могут содержать несколько букв.

Представляю вашему вниманию **лигатуры** — сочетание символов, написанных в виде глифа как единое целое. Зачастую лигатуры состоят из двух символов, но бывает и больше. Эти символьные сочетания находятся в одной литере и служат для красивого написания текста (или для ускорения письма и экономии места в Средние века).

Небольшой пример:



Рис. 11.3 Лигатуры

Для демонстрации я выбрал самый популярный из кириллических шрифтов "**Lobster**", который я скачал на [Google Web Fonts](#). Этот шрифт включает в себя множество разных лигатур.

На рисунке можно увидеть, как в надписи "fish" в одной из литер находятся буквы "f" и "i" (площадка голубого цвета). Эти два символа в этом шрифте сочетаются друг с другом и поэтому не могут находиться на отдельных площадках. Такие сочетания букв с особым начертанием, которые выводятся одной площадкой, и есть наши лигатуры. И если в том или ином шрифте такое предусмотрено, то имейте это ввиду.

Лигатуры, как правило, используются в дизайнерских и старинных шрифтах, таких как **Baskerville, Bembo, Garamond, Caslon**, и т.д.

Сдвиг горизонтальных координат

В горизонтальном форматировании строк литеры выстраиваются одна за другой вплотную, и промежутки между буквами, которые мы видим — исключительно наши знакомые апроши. Но в **CSS** есть свойства, которые способны влиять не только на расстояние между символами, но и на расстояние между словами. Давайте по порядку

letter-spacing

В типографии, **letter-spacing** (так же называемый [трекингом](#)) относится к расстоянию между группой символов, влияющий на плотность и фактуру строки или блока текста.

Порой **letter-spacing** путают с [кернингом](#), но на самом деле это абсолютно разные вещи. **letter-spacing** относится к общему интервалу слов или блоку с текстом, а **кернинг** — это термин, применяемый специально для регулировки расстояния конкретных символов, для коррекции визуально неравномерных расстояний.

letter-spacing, как правило, используется там, где нужно настроить интервалы между символами в словах или общего текста в предложении. С помощью отрицательных или положительных значений можно повысить удобочитаемость текста или его превликательность.

Значения letter-spacing

По сути, **letter-spacing** имеет всего два значения — **normal** и **размерность длины**. При **normal** все литеры остаются на своих местах (каждая сдвигает координату следующей на свою авансовую ширину), с **размерностью длины** эта длина прибавляется к сдвигу координаты (для всех литер).

Рассмотрим на примере значения **normal**:



Рис. 11.4 letter-spacing: normal

При значении **normal** положение каждой буквы определяется исключительно шрифтом — для каждого символа указано, насколько он сдвигает начало следующего символа.

Для примера я решил взять шрифт **Ubuntu Titling**, в котором авансовая ширина заглавной буквы "А" составляет 600/1000 кегля. Выставив размер шрифта **100px (font-size : 100px)**, мы можем видеть, что **х-координата** первой буквы будет **0**, а **х-координата** второй буквы — **60px** (или **0.6em**).

Но, если для этого же текста задать явную длину и **положительное значение**, допустим **letter-spacing: 10px**, то результат будет уже немного иной:



Рис. 11.5 Положительное значение letter-spacing

При **letter-spacing: 10px** расстояние между площадками стало **10px** и, соответственно, координата второй буквы уже стала равна **70px** ($100 \cdot 0.6 + 10$). Отсюда можно сделать вывод, что, чем больше **положительное значение letter-spacing**, тем больше становится расстояние между литерами.

А вот с **отрицательными значениями** дела обстоят немного поинтереснее. Задав **letter-spacing'y -10px** мы можем наблюдать следующую картину:



Рис. 11.6 Отрицательное значение letter-spacing

При **letter-spacing: -10px** координата второй буквы будет равна уже **50px**, т.к. площадки при **отрицательных значениях letter-spacing** налезает друг на друга. На рисунке я попытался это отобразить, отчертив границы первой литеры, чтобы было понятно, о чём идёт речь. И, соответственно, можно сделать вывод, что чем больше отрицательное значение **letter-spacing**, тем больше соседние площадки будут налезать друг на друга.

word-spacing

Логично предположить, что если существует свойство для раздвигания/сдвигания отдельных символов, то, значит, должно быть свойство и для контролирования расстояний между словами.

Представляю вашему вниманию **word-spacing** — свойство, определяющее интервал между *словами*. По умолчанию **word-spacing**, так же, как и **letter-spacing**, задаёт интервал в зависимости от размера и типа шрифта и имеет всего два значения — **normal** и **размерность длины**.

Так как **word-spacing** отличается от **letter-spacing** только тем, что работает с расстояниями между словами, то обсуждать его подробно уже не следует. Поэтому я продемонстрирую пару примеров и двинемся далее.

Положительное значение word-spacing

Для начала давайте посмотрим, что произойдёт, если мы выставим значение **word-spacing** в **20px**:



Рис. 11.7 Положительное значение word-spacing

При **word-spacing: 20px** промежутки между площадками в словах остались на своих местах и каждая литера плотно прижата к соседней, а вот координаты расстояний между словами изменились. Цифрой "1" я отметил сам пробел, который, по сути является тоже символом и его ширина определяется характеристиками шрифта и размером последнего. А вот под цифрой "2" уже находится то расстояние, которое включает в себя **20px**, выставленные для самого **word-spacing**.

Вывод такой. Символ пробела и его размеры в конце каждого слова рассчитываются метриками шрифта и **word-spacing** не может на него повлиять, но зато при **положительном значении** само значение прибавляется к этому пробелу (что и произошло в нашем случае), и таким образом расстояние между слов увеличивается.

Отрицательное значение word-spacing

При **отрицательном значении word-spacing**, как и в случае с **letter-spacing**, картина становится немного интереснее. Снова возьмём значение **20px**, только теперь сделаем его отрицательным:



Рис. 11.8 Отрицательное значение word-spacing

При **word-spacing: -20px** любое слово, которое следует за предыдущим, сдвинулось в левую сторону на **20px** и налезло на своего соседа. При этом стоит заметить одну вещь — пробел, который идёт после каждого слова, остался на своём месте, а сдвиг координат соседских слов (справа) начинается сразу же после него. Красными стрелочками показано, откуда начинается сдвиг и его расстояние. Например, слово "and" переместилось влево ровно на **20px**, наехав своей первой литерой (с буквой "a") на пробел и на последнюю площадку (букву "n") первого слова.

Соответственно, можно смело сделать вывод, что, чем больше отрицательное значение **word-spacing**, тем больше слова справа будут налезать на те слова, которые находятся левее.

Очень большие отрицательные значения letter-spacing и word-spacing

Когда отрицательные значения **letter-spacing** и **word-spacing** достигают определенного порога (скажем, - **0.3em**), то от браузера к браузеру с символами начинают твориться удивительные вещи. Мало того, что не существует даже двух браузеров, в которых большие отрицательные значения **letter-spacing** и **word-spacing** давали бы одинаковое поведение, так ещё и в некоторых есть лимит величины, а, например в **Opera** вообще, на каком-то пороговом значении у **word-spacing**, неожиданно происходит перескок и меньшие значения вообще игнорируются.

Показать всё одним примером и описать поведение всех браузеров просто невозможно, уж очень оно разное, поэтому я решил сделать [интерактивный пример](#), где вы сможете сами поэкспериментировать в разных браузерах, двигая ползунок и меняя значения у элементов.

ИКФ: Горизонтальное выравнивание, часть 2 (12-я публикация цикла “Тайны CSS2.1”)

Добавлено 20 Авг 2012 | Раздел: [CSS](#), [Статьи](#), [Тайны CSS2.1](#)

Горизонтальное выравнивание по ширине строки

Сдвиги междусимвольных или межсловных горизонтальных координат полезны в основном там, где необходимо поработать над красотой текста или отдельных символов. Но бывают задачи более глобального масштаба, когда нужно передвинуть не просто маленькую букву, а, например, **выровнять весь текст** по середине строки или **равномерно растянуть слова** по ширине последней и т.д. Для таких задач в **CSS** тоже имеется конкретный арсенал, с которым нам и предстоит познакомиться в этой части рассказа.

text-align

Свойство **text-align** описывает, как будет вести себя инлайновое содержимое в строке, а точнее способ его выравнивания по ширине строки. Рассмотрим возможные значения этого свойства по порядку.

text-align: center

Значение **center** выравнивает инлайновое содержимое в блоке по **центру строки**. Обычно применяется в заголовках, подписях и т.д.

Его алгоритм мы будем рассматривать на примере **inline-block**'ов. Благодаря их особенностям, о которых мы [писали раньше](#), они могут "чувствовать" ширину, высоту, верхний и нижний **margin**, и это визуально поможет нам лучше увидеть и понять работу **text-align: center**.

И, соответственно, сам пример:

Пример 12.1 text-align: center

а)

HTML

```
<div>
  <span>Слово1</span>
  <span>Слово2</span>
  <span>Слово3</span>
  <span>Слово4</span>
  <span>Слово5</span>
</div>
```

б)

CSS

```
div {
  width: 380px;
  border: 1px solid #633;
  /* выравнивание по центру */
  text-align: center;
}

span {
  display : inline-block;
  width : 80px;
  height: 80px;
  margin: 0 17px 10px;
  background: #E76D13;
  vertical-align: top;
  text-align: center;
  /* эмуляция inline-block для IE6-7*/
  //display : inline;
  //zoom : 1;
}
```

И результат:

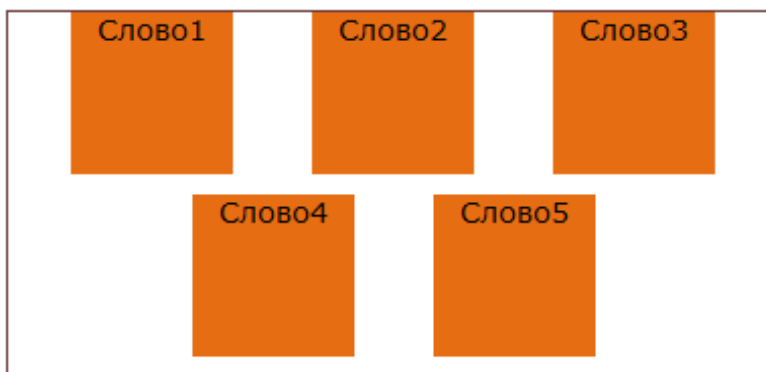


Рис. 12.1 Выравнивание инлайновых элементов по центру

А теперь перейдём к алгоритму **text-align: center**. В нем можно выделить три основных этапа.

Первый этап

В начале берётся строка. Высчитывается общая ширина слов или “монолитного” инлайнового содержимого (**inline-block**, **img**, и т.д) в строке. Причём, если между этим контентом есть фактические пробелы или же отступы, которые реализованы при помощи таких средств, как **word-spacing** и прочих, то эти расстояния так же добавляются к общей сумме ширины контента.

Второй этап

На этом этапе всё ещё проще. Вычисляется оставшаяся ширина строки, т.е. всё свободное пространство, которое не вошло в сумму общей ширины слов с их межсловным расстоянием.

Третий этап

Ну и на завершающем этапе происходит следующее. Самая первая в строке площадка с буквой сдвигается вправо ровно на половину результата, полученного после этапа номер два. Что даёт равные отступы справа и слева самой строки.

Чтобы лучше понять, как всё происходит, я сделал специальный рисунок.

Алгоритм работы: text-align: center



Рис. 12.2 Алгоритм работы text-align: center

Перед нами контейнер с двумя строками, ширина которых составляет **500px**. Так же мы можем видеть, что сумма всех инлайн-блоков в **первой** строке с их интервалами равна **370px**. Значит, на третьем этапе наш алгоритм вычел из первого второе (**500-370**), получив в результате **130**. Далее, как я уже говорил, поделил эту сумму пополам (**130/2**) и отодвинул самый первый инлайн-блок вправо на полученный результат (**65px**). Таким образом, наши инлайн-блоки оказались точно по середине, а отступы по бокам стали абсолютно одинаковыми. Если бы в первой строке не хватило места, то самый крайний справа инлайн-блок перешёл бы на вторую строку, и алгоритм снова включился бы в дело.

То же самое касается и второй строки. В ней алгоритм работает точно так же. Мало того, можно увидеть, что боковые отступы в ней составляют дробное число (**132.5px**), так как **text-align: center** делит общую ширину инлайн-блоков (включая интервалы) ровно на **2**.

text-align: left или right

Алгоритм расчета значений **right** или **left** еще проще, чем для **center**. Контент выстраивается по правому или левому краю строки соответственно, а оставшееся место целиком высвобождается у противоположного края строки.

Немного изменим CSS нашего предыдущего примера:

Пример 12.2 text-align: right

CSS

```
div {  
    width: 380px;  
    border: 1px solid #633;  
    /* выравнивание по правому краю */  
    text-align: right;  
}  
  
span {  
    display : inline-block;  
    width : 80px;  
    height: 80px;  
    margin: 0 10px 10px;  
    background: #E76D13;  
    vertical-align: top;  
    text-align: center;  
    /* эмуляция inline-block для IE6-7*/  
    //display : inline;  
    //zoom : 1;  
}
```

И результат будет уже таким:

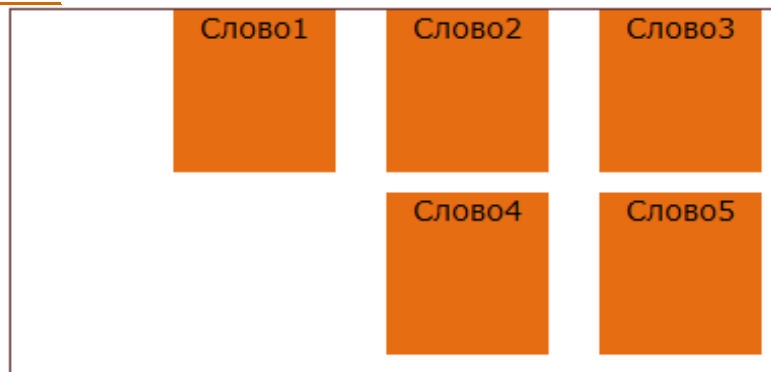


Рис. 12.3 Выравнивание инлайновых элементов по правому краю

Как вы уже поняли, для примера я выбрал **text-align: right**, благодаря которому всё инлайновое содержимое (наши строчно-блочные элементы) в строках переместилось в правую часть, прижавшись к правой границе блока и оставляя слева пустое место.

text-align: left работает по точно такому же принципу, но выравнивает строчное содержимое уже по левой стороне. В отличие от других значений, **left** является значением **text-align** по умолчанию для западных языков, где текст читается слева направо, т.е. можно сказать именно так, как мы с вами и привыкли.

text-align: justify

А вот на значении **justify**, в отличие от предыдущих, стоит остановиться подробнее, разобрать его полностью, включая и алгоритм.

Значение **justify** служит для растягивания текста по всей ширине строки, т.е. выравнивает текст сразу по обеим сторонам, делая расстояния между словами одинаковыми.

В этот раз я не буду предоставлять код, а перейду сразу к алгоритму и рисункам, из которых всё станет ясно.

В общем, углубившись в процесс работы **text-align: justify**, мы можем наблюдать следующий алгоритм:

Этап первый

Сначала в строке текста ищутся минимальные, неразрывные "кирпичики". Это могут быть отдельные слова в тексте, картинки, инлайн-блоки, инлайн-таблицы и т.д. В общем, всё то, что в случае необходимости перенесется на новую строку как единое целое.



Рис. 12.4 Алгоритм работы text-align: justify

Цифрой **1** на картинке отмечены обычные инлайн-боксы, т.е. попросту текст или инлайн элементы, такие, например, как `` или ``.

Под цифрой **2** у нас находится элемент строчно-блочного уровня, т.е. **inline-block**. Как можно заметить, алгоритм отступов внутри него рассчитывается заново. Причина в том, что внутри себя **inline-block** генерирует свой собственный контекст форматирования. Чего нельзя сказать об обычном **inline** элементе, внутри которого межсловное расстояние распределялось бы по общему, внешнему алгоритму.

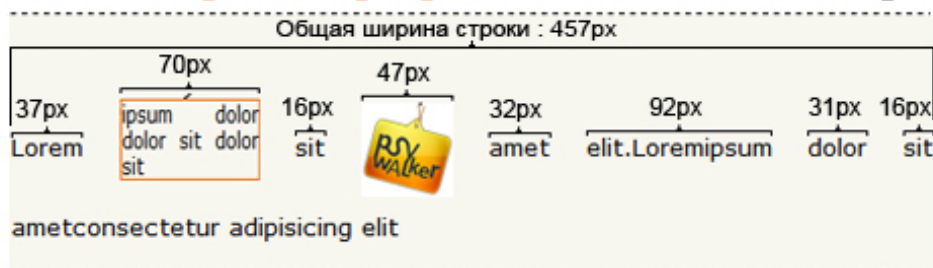
Цифрой **3** отмечена обычная картинка. Так же, как и остальные, она является строчным, целым элементом. Для нашей строки все эти вещи представляют из себя отдельные сущности, неразделимые слова, единые целые. А расстояния между ними как раз и регулируются нашим механизмом под названием **text-align: justify**.

*Последняя же строка не попадает в поле зрения **justify**, так как он работает только для целиком заполненных строк, а в последней строке пробелы всегда остаются своего обычного размера.

Этап второй

Вторым этапом алгоритм высчитывает ширины всех наших "кирпичей" в строке, складывает их, а полученный результат отнимает от общей ширины самой строки.

Общая ширина неразрывных элементов: 341px



Результат: 457 (ширина строки) - 341 (сумма всех ширин элементов) = 116px

Рис. 12.5 Общая ширина неразрывных элементов

Отсюда можно сделать вывод, что сейчас мы имеем общую сумму всех пробельных зон в строке, которая равна **116px**.

Этап третий – завершающий

Третьим и последним этапом алгоритма будет деление полученного числа (в данном случае **116**) на количество пробелов в строке (в нашей строке их **7**). Из полученного результата (**16.571**) вычитается ширина одного пробела, и уже это значение добавляется к каждому из них. Что в итоге даёт равномерное распределение пустого пространства по всей строке.

Выключка по ширине (так **justify** называется по-научному) редко используется в вебе, она более характерна для печатных изданий (газет, журналов). Без хорошей реализации переносов слов она приводит к чрезмерному растягиванию пробелов (особенно в узких колонках), что выглядит ужасно. Однако в особых случаях (напр. стилизация под книгу или газетную полосу) она может оказаться полезным приемом, главное — не злоупотреблять им. А сочетание **justify** с инлайн-блоками иногда и вовсе позволяет решать задачи, казавшиеся нерешаемыми ([1](#), [2](#)).

ИКФ: Горизонтальное выравнивание, часть 3 (13-я публикация цикла “Тайны CSS2.1”)

Добавлено 31 Авг 2012 | Раздел: [CSS](#), [Статьи](#), [Тайны CSS2.1](#)

text-align и новые значения из CSS3

До этого момента мы обсуждали значения **text-align**, которые, во-первых, принадлежат спецификации **CSS2.1**, а во-вторых, уже давно существуют и работают во всех известных нам браузерах. Но прогресс не стоит на месте, и на пороге уже стремительно вырастает **CSS3**, который дарит нам новые модули и значения для известных свойств. И здесь он не остался к нам равнодушен, предложив ещё несколько значений для **text-align**, которые уже есть в **CSS3**, но [содержатся в черновике](#).

Несмотря на то, что новые значения ещё черновики, я посчитал необходимым их обсудить, поэтому продолжим нашу беседу.

text-align: start или end

Значения **start** и **end** не требуют особого разбора, так как оба сводятся к **left** и **right**, но в отличие от последних, они имеют одну небольшую, но интересную особенность. **start** и **end** могут подстраиваться под направление текста.

Сразу перейдём к примеру:

Пример 13.1 text-align: start или end

а)

HTML

```
<div class="ltr">
  <span>Слово1</span>
  <span>Слово2</span>
  <span>Слово3</span>
  <span>Слово4</span>
  <span>Слово5</span>
</div>
<div class="rtl">
  <span>Слово1</span>
  <span>Слово2</span>
  <span>Слово3</span>
  <span>Слово4</span>
  <span>Слово5</span>
</div>
```

б)

CSS

```
div {
  width: 360px;
  border: 1px solid #633;
  margin-bottom: 20px;
}

span {
  display : inline-block;
  width : 60px;
  height: 60px;
  margin: 0 5px 10px;
  background: #E76D13;
  vertical-align: top;
  text-align: center;
  /* эмуляция inline-block для IE6-7*/
  //display : inline;
  //zoom : 1;
}

div.ltr {
  direction: ltr;
  text-align: end;
}

div.rtl {
  direction: rtl;
  text-align: end;
}
```

И результат будет таким:

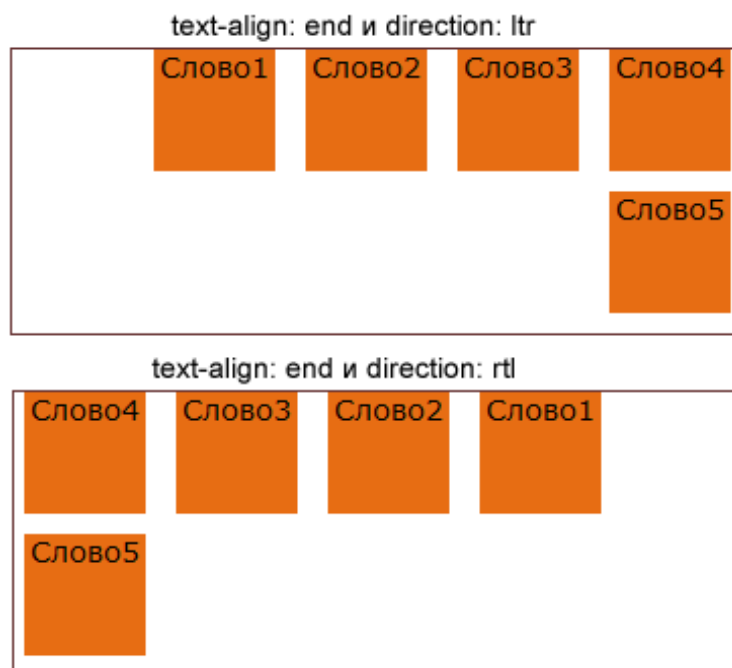


Рис. 13.1 text-align: end

Представим ситуацию: есть два одинаковых абзаца, содержащих по пять слов (наши знакомые строчно-блочные квадратики), и каждому из абзацев выставлено **text-align** в значении **end**. Единственное их отличие заключается в направлении текста. У верхнего абзаца оно работает слева направо (**direction: ltr**), как в английском тексте, а у нижнего абзаца наоборот – справа налево (**direction: rtl**), как на иврите или арабском. Обратите внимание, как в зависимости от направления текста сменилось выравнивание в блоке. В чём же тут дело? А дело всё в том, что значения **start** и **end** подстраиваются под направление текста и действуют

по его правилам, и, т.к. текст идет слева направо, то **start** приравнивается к **left**, а **end** к **right**, и **start** будет выравнивать текст по левому краю. И наоборот, когда направление текста справа налево, то **start**ом будет уже правая сторона, а **end**ом — левая.

В нашем случае значение **end** (считайте **right**) в первом примере прижало блоки вправо, т.к. направление текста в контейнере берёт начало от левого края (**direction: ltr**) и для него **right** — это **end**. Во втором примере элементы прижались к левой границе, потому что направление текста в контейнере пляшет уже от правой стороны (**direction: rtl**) и для него **end** будет уже слева.

Это очень полезно в тех случаях, когда нам нужно поменять язык, например, с западного на арабский, и не нужно беспокоиться о том, что текст не "отзеркалится" в нужном направлении.

text-align: <string>

Ещё одно значение, которое, возможно, будет рекомендовано в **CSS3** — это **<string>**, которое выравнивает **содержимое ячеек таблиц** по **произвольному символу**, заданному в кавычках перед или после других, уже знакомых нам значений.

К сожалению, это значение ещё не поддерживается ни в одном браузере, поэтому предлагаю рассмотреть пример, [который описан в спецификации](#).

Пример 13.2 text-align: <string>

а)

HTML

```
<table>
  <col width="40">
  <tr> <th>Long distance calls
  <tr> <td> $1.30
  <tr> <td> $2.50
  <tr> <td> $10.80
  <tr> <td> $111.01
  <tr> <td> $85.
  <tr> <td> N/A
  <tr> <td> $.05
  <tr> <td> $.06
</table>
```

б)

CSS

```
td { text-align: "." center }
```


А вот и результат:

Long distance calls	
\$11.30	
\$22.50	
\$0.80	
\$200567.01	
\$85.	
N/A	
\$.05	
\$.06	

Рис. 13.2 text-align: <string>

Здесь стоит обратить внимание на две вещи. На само значение, которое состоит из кавычек с точкой (".") и значения **center** после него, и на то, как выравниваются данные в ячейках таблицы. Заметьте, что наша "точка" (".") находится всегда по середине, и выравнивание по **center** происходит **именно относительно неё**.

На месте точки мог бы оказаться любой другой символ (но не слово, иначе строка игнорируется!), а на месте **center** любое другое значение, да и само оно могло стоять перед кавычками или после, т.е. **порядок не имеет значения**.

text-align: match-parent

В спецификации [сказано](#):

Это значение ведет себя так же, как 'inherit', за исключением того, что унаследованные 'start' и 'end' рассчитываются относительно родительского значения 'direction', и в итоге дает рассчитанное значение 'left' или 'right'

Рассмотрим это значение на примере.

Например, на каком-то двуязычном сайте есть комментарии на английском и арабском языках, а в самих комментариях бывают цитаты, в т.ч. иноязычных комментариев. С помощью **match-parent** можно сделать так, чтоб английские комментарии равнялись по левому краю, а арабские – по правому, но цитаты в них равнялись так же, для того чтобы комментарий с цитатой выглядел монолитным блоком.

На деле это будет выглядеть примерно так:

Пример 13.3 text-align: match-parent

а)

HTML

```
<div class="comment">
  <p>This website is for those who are studying English as a foreign language (ESL /
  EFL). The aim of this site is to help you build a firm foundation for your English
  to which you will be able to add new knowledge yourself. This basis consists of four
  essential components that you should master and use actively:</p>
  <blockquote>
    <p>بسم الله الرحمن الرحيم بسم الله الرحمن الرحيم بسم الله الرحمن الرحيم
    الرحيم بسم الله الرحمن الرحيم </p>
  </blockquote>
</div>
```

б)

CSS

```
div.comment {
  background: none repeat scroll 0 0 #F8F7EF;
  border: 1px dashed #000000;
  padding: 0 10px 20px;
  width: 400px;

  direction: ltr;
  text-align: start;
}

blockquote {
  background: #FFC;
  border-radius: 10px;
  padding: 3px 10px;

  direction: rtl;
}
```

И результат будет таким:

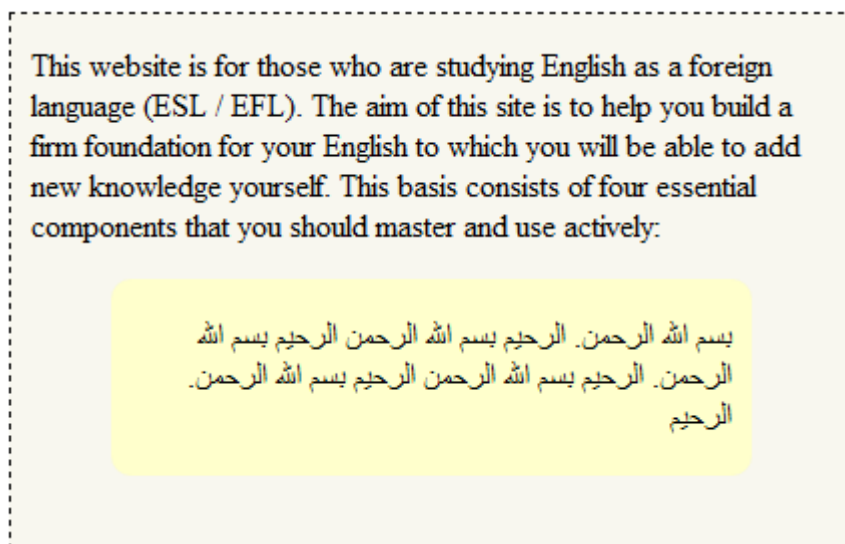


Рис. 13.3 Без указания text-align: match-parent

В таком случае ситуация вышла не очень приятная, так как получается, что текст в блоке выравнивается по разным сторонам, в зависимости от направлений. Наша же задача сделать так, чтобы в независимости от языка и направления текст выравнивался именно по левой части, как и основной текст в блоке.

Вот для этого дела у нас и существует **match-parent**, который мы добавляем потомкам, в нашем случае цитате:

В)

CSS

```
blockquote {  
    background: #FFC;  
    border-radius: 10px;  
    padding: 3px 10px;  
  
    direction: rtl;  
  
    /* Решение */  
    text-align: match-parent;  
}
```

И результат уже будет таким, каким нам и нужно:

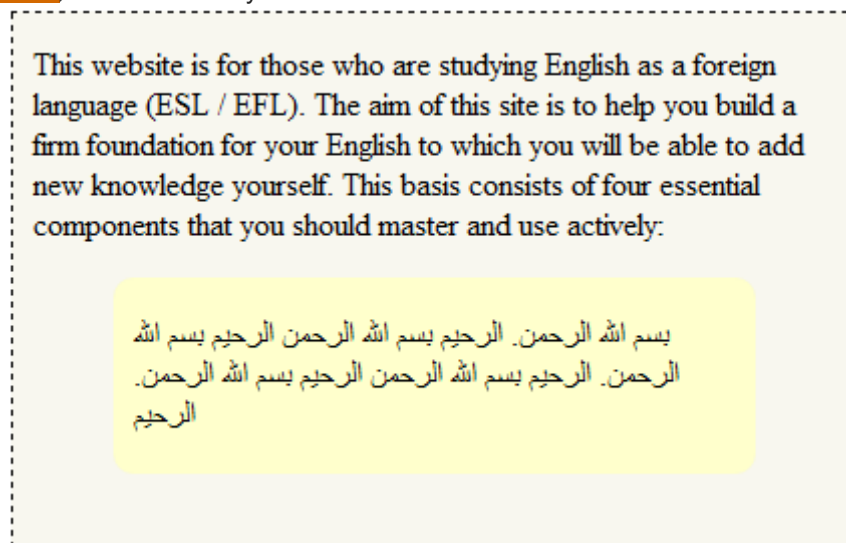


Рис. 13.4 text-align: match-parent

Вот теперь другое дело! Выравнивание внутри блока с цитатой теперь рассчитывается по другому, а именно по направлению слева направо, как у своего предка, у которого стоит **direction: ltr** и его **start** теперь и будет **left**’ом. И цитате теперь абсолютно всё равно, какой **direction** прописан у неё.

К сожалению, это значение в настоящий момент тоже не поддерживается браузерами.

start end

И последнее, немного странное значение, которое осталось разобрать – это **start end**, о котором в спецификации сказано следующее:

Первая строка и каждая последующая строка непосредственно после принудительного разрыва строки выравнивается по 'start', все остальные строки (кроме последней, если на нее действует text-align-last) — по 'end'

По описанию это значение является очень простым и не требует каких-то особых примеров. Поэтому просто разберём на словах.

Допустим, у нас есть пять строк текста. Первая строка разорвана на две части с помощью **
, и так как у блока прописано **text-align: start end, то вот эти две строки (первая и та, которая сразу после **
) будут выравниваться по левому краю (по **start’у), а все остальные три – по правому (по **end**’у).

Честно говоря, я не понял, для чего это нужно, но вроде в каких-то дизайнерских демках видел подобное.

** Стоит заметить, что такие значения, как `<string>` и `'start end'`, помечены в спецификации, как "под риском удаления", поэтому далеко не факт, что эти вещи приживутся и станут кандидатами в рекомендации.*

inko

