

# Современный учебник JavaScript

© Илья Кантор

Сборка от 23 июля 2013 для чтения с устройств

Внимание, эта сборка может быть устаревшей и не соответствовать текущему тексту.

Актуальный онлайн-учебник, с интерактивными примерами, доступен по адресу <http://learn.javascript.ru>.

Вопросы по JavaScript можно задавать в комментариях на сайте или на форуме [javascript.ru/forum](http://javascript.ru/forum).

Вопросы по сборке, предложения по её улучшению – можно писать мне, по адресу [iliakan@javascript.ru](mailto:iliakan@javascript.ru).

## Глава: Документ и объекты страницы

В файле находится только одна глава учебника. Это сделано в целях уменьшения размера файла, для удобного чтения с устройств.

---

## Содержание

Окружение: DOM, BOM и JS

BOM-объекты: navigator, screen, location, frames

navigator: платформа и браузер

screen

location

Методы и свойства Location

Методы объекта Location

frames

history

Итого

DOM-элементы и их свойства

Дерево DOM

Пример DOM

Ещё узлы

Возможности, которые дает DOM

Итого

Работа с DOM из консоли

Доступ к элементу

Elements - Консоль

Ещё методы консоли

Кросс-браузерность

Навигация в DOM, свойства-ссылки

Корень: documentElement и body

Дочерние элементы

childNodes

children

## Ссылки вверх и вниз

firstChild и lastChild

parentNode, previousSibling и nextSibling

## Таблицы

## Формы

## От элементов к форме

## Дополнительные ссылки для элементов (кроме IE<9)

## Итого

Ссылки для таблиц

Для форм

Только элементы

## Свойства узлов: тип, тег, содержимое и другие

Тип: nodeType

Тег: nodeName и tagName

Какая разница между tagName и nodeName ?

innerHTML: содержимое элемента

Тонкости innerHTML

outerHTML: HTML узла целиком

nodeValue/data: содержимое текстового узла

## Другие свойства

## Итого

## Атрибуты и "свои" свойства

«Свои» свойства

## Атрибуты

## Синхронизация свойств и атрибутов

id

href

value

class/className

«Особенности» старых IE

## Итого

## Интерактивное путешествие по DOM

## HTML-разметка документа для путешествия

Так выглядит документ

Здесь стоите Вы

## Поиск: getElement\* и querySelector\*

getElementById

getElementsByTagName

getElementsByName

getElementsByClassName

querySelectorAll

querySelector

matchesSelector

## XPath в современных браузерах

## Внутреннее устройство поисковых методов

document.getElementById(id)

elem.querySelector(query), elem.querySelectorAll(query)

elem.getElementsByTagName(...)

Алгоритмы getElementsBy\*

## Практика

Отобразите число потомков

Дополнительно

## Итого

## Добавление и удаление узлов

Создание элементов: createElement

Добавление элемента: appendChild, insertBefore

Удаление узлов: removeChild

Пример: показ сообщения

- Создание сообщения

- Добавление

Текстовые узлы, тонкости использования

Итого

## Мультивставка: insertAdjacentHTML и DocumentFragment

Оптимизация вставки в документ

Добавление множества узлов

insertAdjacentHTML

- insertAdjacentElement и insertAdjacentText

DocumentFragment

Итого

## Метод document.write

Как работает document.write

Только до конца загрузки

Преимущества перед innerHTML

- Реклама

- Альтернатива: вставка через DOM

Итого

## Внешний вид: стили, прокрутка, координаты

### Стили и классы, getComputedStyle

className

classList

- Функции-заменители classList

style

- style.cssText

- Получение информации о style

getComputedStyle и currentStyle

- currentStyle (IE)

Итого

### Размеры и прокрутка элементов

Образец документа

CSS-метрики width/height

- Полоса прокрутки

JavaScript-метрики

- clientWidth/Height

- scrollWidth/Height

- scrollTop/scrollLeft

- offsetWidth/Height

- clientTop/Left

- offsetParent, offsetLeft/Top

Итого

### Размеры и прокрутка для страницы

Ширина/высота видимой части окна

Ширина/высота всей страницы, с учётом прокрутки

Прокрутка страницы

- Получение текущей прокрутки

- С учётом IE7- и Quirks Mode

- Изменение прокрутки: scrollTo, scrollBy, scrollIntoView

Запрет прокрутки

Итого

## Координаты

Координаты относительно окна и документа

### Получение координат элемента

Координаты в окне: `elem.getBoundingClientRect()`

Координаты в документе

Устаревший метод: `offset*`

Сравнение `offset*` с `getBoundingClientRect`

Комбинированный подход

Получение элемента по координатам: `elementFromPoint(x,y)`

Координаты на экране `screenX/screenY`

Итого

## Проверка вложенности и соседства

Метод `compareDocumentPosition`

Поддержка в IE8-

Итого

## DOM-шпаргалка

Создание

Свойства узлов

Ссылки

Таблицы

Формы

Поиск

Изменение

Классы и стили

Размеры и прокрутка элемента

Размеры и прокрутка страницы

Координаты

## Решения задач

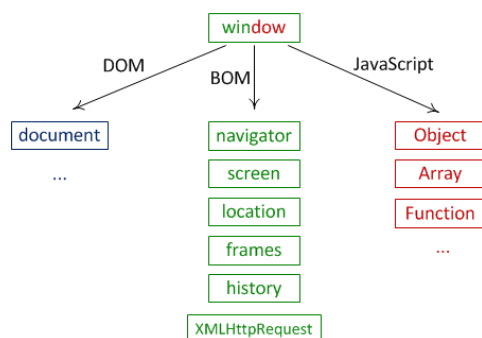
---

# Окружение: DOM, BOM и JS

---

Браузер дает доступ к иерархии объектов, которые мы можем использовать для разработки.

На рисунке схематически отображена структура основных браузерных объектов.



На вершине стоит `window`, который еще называют *глобальным объектом*.

Все остальные объекты делятся на 3 группы.

### Объектная модель документа (DOM)

Доступна через `document`. Дает доступ к содержимому страницы.

На странице **W3C DOM** вы можете найти стандарты DOM, разработанные самим W3C. На данный момент существует 3 уровня DOM. Современные браузеры также поддерживают некоторые возможности, которые называются DOM 0 и которые остались еще с той эпохи, когда не было W3C.

### Объектная модель браузера (BOM)

BOM — это объекты для работы с чем угодно, кроме документа.

Доступ к фреймам, запросы к серверу, функции `alert/confirm/prompt` — все это BOM.

Большинство возможностей BOM стандартизированы в HTML5, но браузеры любят изобрести что-нибудь своё, особенное.

### Объекты и функции JavaScript

JavaScript — связующий все это язык. Встроенные в него объекты и сам язык в идеале должны соответствовать стандарту ECMA-262, но пока что браузеры к этому не пришли. Хотя положительная тенденция есть.

Глобальный объект `window` имеет две роли:

1. Это окно браузера. У него есть методы `window.focus()`, `window.open()` и другие.
2. Это глобальный объект JavaScript.

Вот почему он на рисунке представлен зеленым и красным цветом.

## BOM-объекты: `navigator`, `screen`, `location`, `frames`

В этой главе мы разберём браузерные объекты, которые не относятся к документу, но бывают полезны.

### `navigator`: платформа и браузер

Объект `navigator` содержит общую информацию о браузере и операционной системе. Особенно примечательны два свойства:

- `navigator.userAgent` — содержит информацию о браузере.
- `navigator.platform` — содержит информацию о платформе, позволяет различать Windows/Linux/Mac и т.п..

Для вашего браузера значения:

```
1 alert(navigator.userAgent);  
2 alert(navigator.platform);
```

С другой стороны, зачем нужно определять браузер? Если хочется проверить, поддерживает ли браузер какую-то возможность, то всегда лучше провести проверку так, чтобы не зависеть от названия браузера.

Тогда в будущем, когда браузер добавит новую возможность, ваш скрипт тут же подхватит её и продолжит работать.

Но бывает (редко), что под определенными платформами, в некоторых версиях браузеров есть ошибки (да-да, в самих браузерах), и никак кроме

проверки на название/платформу это не обнаружить. Поэтому такую возможность стоит иметь в виду.

## screen

Объект `screen` содержит общую информацию об экране, включая его разрешение, цветность и т.п. Оно может быть полезно для определения, что код выполняется на мобильном устройстве с маленьким разрешением.

Текущее разрешение экрана посетителя по горизонтали/вертикали находится в `screen.width/screen.height`.

Разрешение у вас сейчас:

1280 x 800

Это свойство можно использовать для сбора статистической информации о посетителях.

JavaScript-код счетчиков считывает эту информацию и отправляет на сервер. Именно поэтому можно просматривать в статистике, сколько посетителей приходило с каким экраном.

## location

Объект `location` предоставляет информацию о текущем URL и позволяет JavaScript перенаправить посетителя на другой URL. Значением этого свойства является объект типа `Location`.

### Методы и свойства Location

Самый главный метод — это, конечно же, `toString`. Он возвращает полный URL.

Следующая кнопка выведет текущий адрес:

```
alert(window.location)
```

Код, которому нужно провести строковую операцию над `location`, должен *сначала привести объект к строке*. Вот так будет ошибка:

```
1 // будет ошибка, т.к. location - не строка
2 alert( window.location.indexOf('/://') );
```

... А так - правильно:

```
1 // привели к строке перед indexOf
2 alert( (window.location + '').indexOf('/://') );
```

Все следующие свойства являются строками.

Колонка «Пример» содержит их значения для тестового URL:

→ <http://www.google.com:80/search?q=javascript#test>

Свойство	Описание	Пример
hash	часть URL, которая идет после символа решетки '#', включая символ '#'	#test
host	хост и порт	www.google.com:80

href	весь URL	http://www.google.com:80/search?q=javascript#test
hostname	хост (без порта)	www.google.com
pathname	строка пути (относительно хоста)	/search
port	номер порта (если порт не указан, то пустая строка)	80
protocol	протокол	http: (двоеточие на конце)
search	часть адреса после символа "?", включая символ "?"	?q=javascript



### Баг с hash в Firefox

В Firefox есть баг: если hash-компонент адреса содержит **URL-кодированные** символы, то свойство hash возвращает раскодированный компонент. Другие браузеры ведут себя корректно и не раскодируют hash.

То есть, для hash вида "%2F" все браузеры вернут его, как есть, а Firefox раскодирует и вернет символ "/".

## Методы объекта Location

### assign(url)

загрузить документ по данному url. Можно и просто приравнять `window.location.href = url`.

### reload([forceget])

перезагрузить документ по текущему URL. Аргумент `forceget` - булево значение, если оно `true`, то документ перезагружается всегда с сервера, если `false` или не указано, то браузер может взять страницу из своего кэша.

### replace(url)

заменить текущий документ на документ по указанному url.

### toString()

Как обсуждалось выше, возвращает строковое представление URL.

При изменении любых свойств `window.location`, кроме `hash`, документ будет перезагружен, как если бы для модифицированного url был вызван метод `window.location.assign()`.

Можно перенаправить и явным присвоением `location`, например:

```
1 // браузер загрузит страницу http://javascript.ru
2 window.location = "http://javascript.ru";
```

Еще пример. Он перезагрузит страницу с новыми параметрами после "?":

```
function refreshSearch(search) {
    window.location.search = search;
}
```

При вызове `refreshSearch('My Data')` на сервер отправится строка с параметрами `?My%20Data`.



### location и история посещения

При смене URL вызовом `location.replace(url)` страница гарантированно не записывается в истории посещений. В частности, это значит, что посетитель не сможет использовать для возврата кнопку браузера «Назад».

При прямом присвоении или вызове `location.assign(url)` — зависит от браузера: некоторые запишут это в историю, некоторые — нет.

## frames

Коллекция, содержащая фреймы и ифреймы. Можно обращаться к ним как по номеру, так и по имени.

**В frames содержатся window-объекты дочерних фреймов.**

Следующий код переводит фрейм на новый URL:

```
1 <iframe name="example" src="http://example.com"
  width="200" height="100"></iframe>
2
3 <script>
4   window.frames.example.location = 'http://example.com';
5 </script>
```



### Фреймы и безопасность

Политика безопасности браузера устроена так, что окно может обращаться к переменным фрейма напрямую лишь в том случае, если находится на одном с ним домене (включая тот же протокол и порт).

То есть, *прочитать* `frames.example.location` и узнать, таким образом, по какому адресу находится посетитель внутри ифрейма, мы бы не смогли. Но, в качестве исключения, *присвоить* это свойство можно.

## history

Объект `history` позволяет менять URL без перезагрузки страницы (в пределах того же домена) при помощи [History API](#), а также перенаправлять посетителя назад-вперед по истории. Эта тема достаточно обширна, поэтому будет раскрыта в отдельной главе.

Объект `history` не предоставляет возможности *читать* историю посещений. Можно отправить посетителя назад вызовом `history.back()` или вперед вызовом `history.forward()`, но сами адреса браузер не дает из соображений безопасности.

## Итого

Браузерные объекты:

### **navigator, screen**

Содержат информацию о браузере и экране.

### **location**

Содержит информацию о текущем URL и позволяет её менять. Любое изменение, кроме `hash`, перегружает страницу. Также можно



перезагрузить страницу с сервера вызовом `location.reload(true)`.

## frames

Содержит коллекцию `window`-объектов для каждого из дочерних фреймов. Каждый фрейм доступен по номеру (с нуля) или по имени, что обычно удобнее.

## history

Позволяет отправить посетителя на предыдущую/последующую страницу по истории, а также изменить URL без перезагрузки страницы с использованием [History API](#).

Существуют и другие браузерные объекты, такие как `XMLHttpRequest`, которые мы более подробно разберем в дальнейшем.

# DOM-элементы и их свойства

## Дерево DOM

Основным инструментом работы и динамических изменений на странице является DOM (Document Object Model) — объектная модель, используемая для XML/HTML-документов.

Согласно DOM-модели, документ является иерархией, деревом.

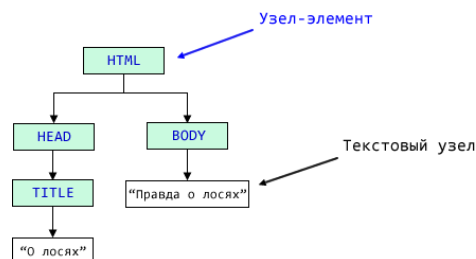
Каждый HTML-тег образует узел дерева с типом «элемент». Вложенные в него теги становятся дочерними узлами. Для представления текста создаются узлы с типом «текст».

Проще говоря, DOM — это представление документа в виде дерева тегов, *доступное для изменения через JavaScript*.

## Пример DOM

Построим, для начала, дерево DOM для следующего документа.

```
1 <html>
2   <head>
3     <title>О лосях</title>
4   </head>
5   <body>
6     Правда о лосях
7   </body>
8 </html>
```



В этом дереве выделено два типа узлов.

1. Теги образуют **узлы-элементы** (element node) DOM-дерева. Естественным образом одни узлы вложены в другие.
2. Текст внутри элементов образует **текстовые узлы**. Текстовый узел содержит исключительно строку текста и не может иметь потомков.

Структура дерева образована за счет синих элементов-узлов — тегов HTML.



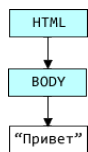
## Браузер автоматически исправляет HTML при создании DOM

В DOM всегда есть тег HTML. Даже если документ некорректен — он там будет, браузер создаст его самостоятельно.

То же самое касается и тега BODY.

Например, документ из одного слова "Привет" эквивалентен такому:

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



Это для общей информации. Конечно, при разработке лучше использовать валидный HTML с корректным DOCTYPE.



Даже при валидном HTML браузер может построить немного другую структуру DOM.

В первую очередь это относится к таблице. Например, если в HTML таблица имеет такой вид:

```
<table>
  <tr><td>1</td></tr>
</table>
```

...То в DOM будет автоматически добавлен промежуточный тег TBODY, как будто документ такой:

```
1 <table>
2   <tbody>
3     <tr><td>1</td></tr>
4   </tbody>
5 </table>
```

Здесь нет никакой ошибки в HTML, но по стандарту TBODY должен быть, поэтому браузер добавляет его.

Впрочем, такой случай является лишь важным исключением. На практике такое происходит очень редко, обычно DOM полностью соответствует структуре HTML.

## Ещё узлы

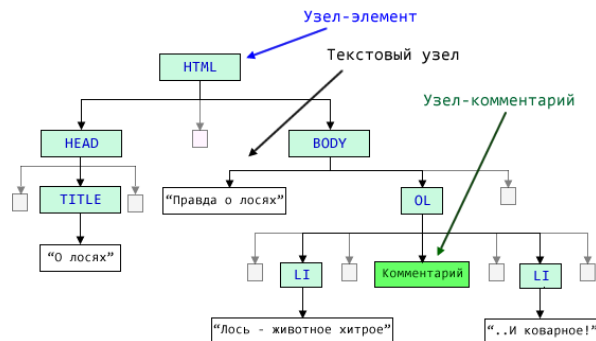
Дополним страницу новыми тегами и комментарием:

```
01 <!DOCTYPE HTML>
02 <html>
03   <head>
04     <title>0 лосях</title>
05   </head>
06   <body>
```

```

07   Правда о лосях
08   <ol>
09     <li>Лось – животное хитрое</li>
10     <!-- комментарий -->
11     <li>..и коварное!</li>
12   </ol>
13 </body>
14 </html>

```



В этом примере тегов уже больше, и даже появился узел нового типа — **узел-комментарий**. Кажется бы, зачем комментарий в DOM? На отображение-то он всё равно не влияет. Но так как он есть в HTML — обязан присутствовать в DOM-дереве.

**Всё, что есть в HTML, находится и в DOM.**

Как вы думаете, пробелы и переводы строки между тегами должны попадать в DOM ?

Элемент с пробелами `<li> </li>` чем-то должен отличаться от `<li></li>` в DOM ?

**Поскольку DOM-модель в точности соответствует документу, пробельные символы так же важны, как и любой другой текст.** Ответ на оба вопроса: «Да».

На картинке выше текстовые узлы, содержащие пробелы, выделены серым. Единственное исключение — пробелы перед и между HEAD/BODY на самом верхнем уровне документа. В DOM их никогда нет, согласно требованиям стандарта. Все остальные пробелы сохраняются в точности.



Пробелы есть в DOM, только если они есть в документе

**Если не будет пробелов между тегами — не будет и пробельных узлов.**

Например: `<li></li>` — узел-элемент без потомков.

Следующий документ вообще не содержит пробельных узлов:

```

<!DOCTYPE
HTML><html><head><title>Title</title></head><body></body></html>

```



**В IE до версии 9 пробельных узлов нет**

IE до версии 9 не генерировал узлы из пробелов. Это очень важно, т.к. приводит к тому, что DOM-дерево в IE6,7,8 не такое как в остальных браузерах.

Точнее говоря, оно такое же, как на картинке выше, но **без пробельных узлов**.

Это различие следует иметь в виду при разработке под IE<9.



## DOCTYPE — тоже узел!

Вообще-то это секрет, но DOCTYPE тоже является DOM-узлом, и находится в дереве DOM слева от HTML (на рисунке выше этот факт скрыт).

P.S. Насчет секрета - конечно, шутка, но об этом и правда далеко не все знают. Сложно придумать, где такое знание может пригодиться...

## Возможности, которые дает DOM

Зачем, кроме красивых рисунков, нужна иерархическая модель DOM?

**DOM нужен для того, чтобы манипулировать страницей — читать информацию из HTML, создавать и изменять элементы.**

Например, можно поменять цвет BODY и вернуть обратно:

```
1 document.body.style.backgroundColor = 'red';
2 alert('Поменяли цвет BODY');
3
4 document.body.style.backgroundColor = '';
5 alert('Сбросили цвет BODY');
```

Фактически, DOM предоставляет возможность делать со страницей всё, что угодно. Далее в этом разделе мы научимся этому.

## Итого

- DOM-модель — это внутреннее представление HTML-страницы в виде дерева.
- Все элементы страницы, включая теги, текст, комментарии, являются узлами DOM.
- У элементов DOM есть свойства и методы, которые позволяют изменять их.
- Кстати, DOM-модель используется не только в JavaScript, это известный способ представления XML-документов.

В следующих главах мы лучше познакомимся с DOM и увидим, какие свойства и методы нужны, чтобы творить красивые штуки со страницей.

## Работа с DOM из консоли

Исследовать и изменять DOM можно с помощью инструментов разработки, встроенных в браузер. Посмотрим средства для этого на примере Google Chrome.

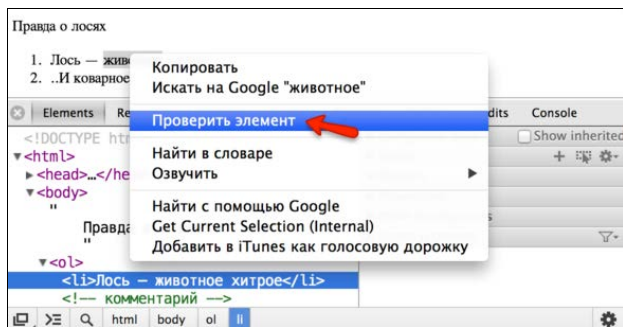
### Доступ к элементу

Чтобы проанализировать любой элемент:

- Выберите его во вкладке Elements.
- ...Либо внизу вкладки Elements есть лупа, при нажатии на которую

можно выбрать элемент кликом.

⇒ ...Либо, что обычно удобнее всего, просто кликните на нужном месте на странице правой кнопкой и выберите в меню «Проверить Элемент».



Справа будет различная информация об элементе:

## Computed Style

Итоговые свойства CSS элемента, которые он приобрёл в результате применения всего каскада стилей, включая внешние CSS-файлы и атрибут style.

## Style

Каскад стилей, применённый к элементу. Каждое стилевое правило отдельно, здесь же можно менять стили кликом.

## Metrics

Размеры элемента.

...

И еще некоторые реже используемые вкладки, которые станут понятны по мере изучения DOM.



**DOM не полностью отображается в инструментах!**

Во вкладке Elements не отображаются пробельные узлы.

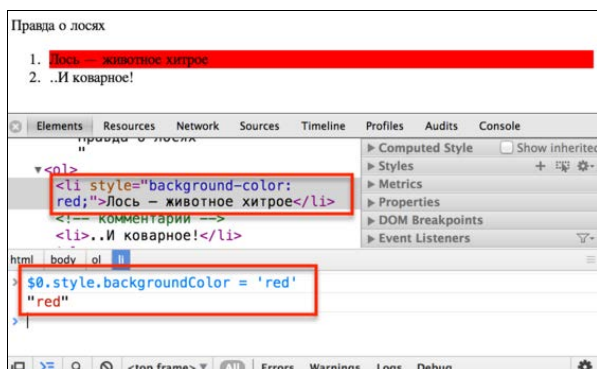
Это сделано для удобства просмотра. На самом-то деле они есть.

## Elements - Консоль

Зачастую бывает нужно выбрать элемент DOM и сделать с ним что-то на JavaScript.

Консоль можно либо открыть тут же, нажатием Esc, либо выбрать на соответствующую вкладку.

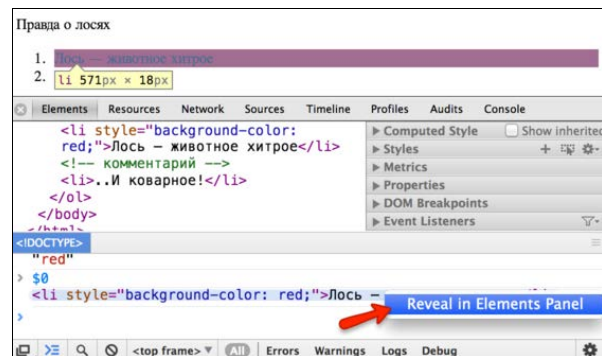
**Последний элемент, выбранный во вкладке Elements, доступен в консоли как \$0, предыдущий — \$1 и так далее.**



Есть и обратный путь.

Чтобы показать элемент из JS-переменной во вкладке Elements:

1. Выведите его в консоли.
2. Кликните на нём правой кнопкой мыши.
3. Выберите соответствующий пункт меню.



Таким образом можно легко перемещаться из Elements в консоль и обратно. Выбрал элемент — попробовал на нём JavaScript — посмотрел что получилось в DOM.

## Ещё методы консоли

Для поиска элементов в консоли есть два специальных метода:

- `$("div.my")` — возвращает элемент по CSS-селектору, только один (первый).
- `$$("div.my")` — возвращает массив элементов по CSS-селектору.

Более полная документация по методам консоли доступна здесь: ([https://getfirebug.com/wiki/index.php/Command\\_Line\\_API](https://getfirebug.com/wiki/index.php/Command_Line_API))[Command Line API], а также на (<http://firebug.ru>)[firebug.ru].

## Кросс-браузерность

Очень похожим образом работают Chrome/Safari и Firefox. Меньше возможностей поддерживает Opera. Замыкает цепочку, пожалуй, IE.

Поэтому обычно начинают разработку в Chrome/Safari/Firefox, а затем, если что-то не работает в других браузерах, то используют их инструменты.

## Навигация в DOM, свойства-ссылки

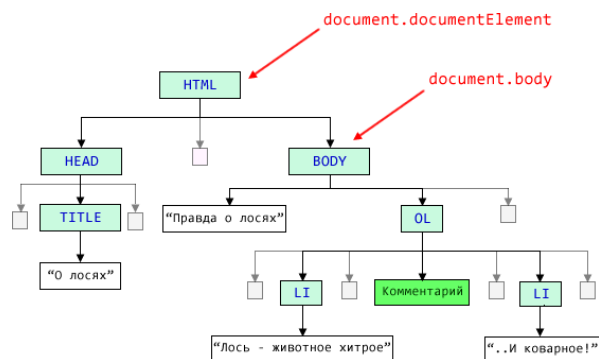
Для того, чтобы изменить узел DOM, например, раскрыть в нем меню, нужно сначала его получить.

Доступ к DOM начинается с `document`. Оттуда можно добраться до любых других узлов.

## Корень: `documentElement` и `body`

Войти в «корень» дерева можно двумя путями.

1. Первая точка входа — `document.documentElement`. Это свойство ссылается на DOM-объект для тега HTML.
2. Вторая точка входа — `document.body`, который соответствует тегу BODY.



Оба варианта отлично работают. Но есть одна тонкость: **document.body** может быть равен **null**.

Например, при доступе к `document.body` в момент обработки тега `HEAD`, то `document.body = null`. Это вполне логично, потому что `BODY` еще не существует.

**Нельзя получить доступ к элементу, которого еще не существует в момент выполнения скрипта.**

В следующем примере, первый `alert` выведет `null`:

```

01 <!DOCTYPE HTML>
02 <html>
03   <head>
04     <script>
05       alert("Из HEAD: " + document.body); // null
06     </script>
07   </head>
08   <body>
09
10     <script>
11       alert("Из BODY: " + document.body);
12     </script>
13
14   </body>
15 </html>

```

В мире DOM для свойств-ссылок на узлы в качестве значения «нет такого элемента» или «узел не найден» используется не `undefined`, а `null`.

## Дочерние элементы

Из узла-родителя можно получить все дочерние элементы. Для этого есть несколько способов.

### childNodes

Псевдо-массив `childNodes` хранит все дочерние элементы, включая текстовые.

Пример ниже последовательно выведет дочерние элементы `document.body`:

```

01 <!DOCTYPE HTML>
02 <html>
03   <head><meta charset="utf-8"></head>
04   <body>
05     <div>Пользователи:</div>
06     <ul>
07       <li>Маша</li>
08       <li>Вовочка</li>
09     </ul>

```

```

10
11     <!-- комментарий -->
12
13     <script>
14         var childNodes = document.body.childNodes;
15
16         for(var i=0; i<childNodes.length; i++) {
17             alert(childNodes[i]);
18         }
19     </script>
20
21 </body>
22 </html>

```

Во всех браузерах, кроме старых IE, `document.body.childNodes[0]` это текстовый узел из пробелов, а `DIV` — второй потомок: `document.body.childNodes[1]`.

В IE8- не создаются пустые текстовые узлы, поэтому там дети начнутся с `DIV`.

Как вы думаете, почему перечисление узлов в примере выше заканчивается на `SCRIPT` ? Неужели под скриптом нет пробельного узла?

Узел будет в итоговом документе, его еще нет на момент выполнения скрипта.

## children

А что если текстовые узлы не нужны? Для этого существует свойство `children`, которое перечисляет только дочерние узлы, соответствующие тегам.

Посмотрим следующий пример. Он идентичен предыдущему, за исключением того, что в нем используется `children` вместо `childNodes`. Поэтому он будет выводить не все узлы, а только узлы-элементы.

```

01 <!DOCTYPE HTML>
02 <html>
03     <head><meta charset="utf-8"></head>
04     <body>
05         <div>Пользователи:</div>
06         <ul>
07             <li>Маша</li>
08             <li>Вовочка</li>
09         </ul>
10
11     <!-- комментарий -->
12
13     <script>
14         var children = document.body.children;
15
16         for(var i=0; i<children.length; i++) {
17             alert(children[i]); // DIV, UL, SCRIPT
18         }
19     </script>
20
21 </body>
22 </html>

```



В IE<9 в `children` присутствуют узлы-комментарии

С точки зрения стандарта это ошибка, но IE<9 также включает в `children` узлы, соответствующие HTML-комментариям.





## Задача: DOM children

Для страницы:

Важность: 5

```
01 <!DOCTYPE HTML>
02 <html>
03   <head><meta charset="utf-8"></head>
04   <body>
05     <div>Пользователи:</div>
06     <ul>
07       <li>Маша</li>
08       <li>Вовочка</li>
09     </ul>
10
11     <!-- комментарий -->
12
13     <script>
14       // ... ваш код
15     </script>
16
17   </body>
18 </html>
```

- Напишите код, который получит элемент HEAD при помощи children.
- Напишите код, который получит UL.
- Напишите код, который получит второй LI. Будет ли ваш код работать кросс-браузерно, если комментарий переместить между элементами LI?

[Решение задачи "DOM children" »»](#)

## Ссылки вверх и вниз

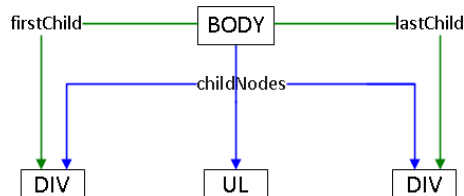
Для комфортного перемещения по узлам существуют дополнительные свойства, указывающие вверх, вниз, на соседей и т.п.

### firstChild и lastChild

Свойства firstChild и lastChild обеспечивают быстрый доступ к первому и последнему потомку.

Например, для документа:

```
<html><body><div>...</div><ul>...</ul><div>...</div></body></html>
```



Почему документ в одну строчку? Да потому что если добавить пробелов, то первым и последним узлами будут не элементы, а пробельные узлы.

Вот в этом документе firstChild и lastChild будут указывать уже не на узлы-элементы, а на текстовые, пробельные узлы:

```
1 <html>
2 <body>
3   <div>...</div>
4   <ul>...</ul>
5   <div>...</div>
```

```
6     </body>
7 </html>
```

В любом случае `firstChild` и `lastChild` — это более быстрый и короткий способ обратиться к первому и последнему элементам `childNodes`. Верны равенства:

```
body.firstChild === body.childNodes[0]
body.lastChild === body.childNodes[body.childNodes.length-1]
```

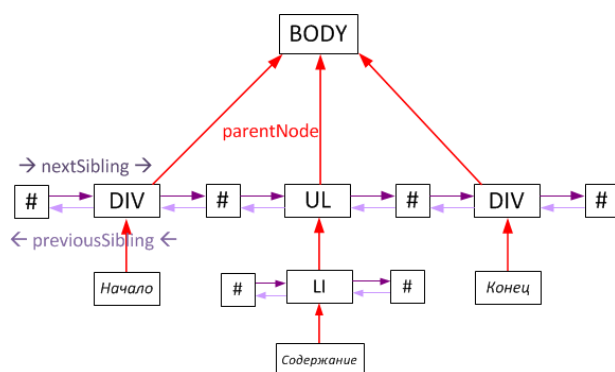
parentNode, previousSibling и nextSibling

- Свойство `parentNode` ссылается на родительский узел.
- Свойства `previousSibling` и `nextSibling` дают доступ к левому и правому соседу.

Ниже изображены ссылки между BODY и его потомками документа:


```
01 <!DOCTYPE HTML>
02 <html>
03 <head><meta charset="utf-8"></head>
04 <body>
05   <div>Начало</div>
06
07   <ul>
08     <li>Содержание</li>
09   </ul>
10
11   <div>Конец</div>
12 </body>
13 </html>
```

Ссылки (пробельные узлы обозначены решеткой #):



Все навигационные ссылки — только для чтения. При изменениях DOM, добавлении или удалении элементов они обновляются автоматически.



 **Задача:** Проверка существования детей

Напишите самый короткий код для проверки, пуст ли элемент `elem`. «Пустой» — значит нет дочерних узлов, даже текстовых.

```
if (/*...ваш код проверки elem... */) { узел elem пуст
}
```

## Решение задачи "Проверка существования детей" »»



 **Задача:** Вопрос по навигационным ссылкам

Если `elem` — это произвольный узел DOM...

Важность: 5

Верно ли, что `elem.lastChild.nextSibling` всегда `null`?

Верно ли, что `elem.children[0].previousSibling` всегда `null` ?

[Решение задачи "Вопрос по навигационным ссылкам" »»](#)

## Таблицы

У таблиц есть дополнительные свойства для более удобной навигации по ним (выделены наиболее полезные):

### TABLE

- `table.rows` — список строк TR таблицы.
- `table.caption/tHead/tFoot` — ссылки на элементы таблицы CAPTION, THEAD, TFOOT.
- `table.tBodies` — список элементов таблицы TBODY, по спецификации их может быть несколько.

### THEAD/TFOOT/TBODY

- `tbody.rows` — список строк TR секции.

### TR

- `tr.cells` — список ячеек TD/TH
- `tr.sectionRowIndex` — номер строки в текущей секции THEAD/TBODY
- `tr.rowIndex` — номер строки в таблице

### TD/TH

- `td.cellIndex` — номер ячейки в строке

Пример использования:

```
01 <table>
02   <tr> <td>один</td> <td>два</td>    </tr>
03   <tr> <td>три</td>  <td>четыре</td> </tr>
04 </table>
05
06 <script>
07   var table = document.body.children[0];
08
09   alert( table.rows[0].cells[0].innerHTML ) // "один"
10 </script>
```

Спецификация: [HTMLTableElement](#) and [HTMLTableRowElement](#) .

Эти свойства бывают удобны при работе с таблицами, т.к. делают код проще и короче.

В IE7 ряд «табличных» свойств не работает, если элемент вне документа.

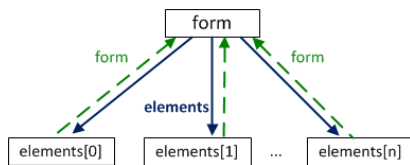
## Формы

Элементы FORM можно получить по имени или номеру, используя свойство `document.forms[name/index]`.

Например:

```
document.forms.my -- форма с именем 'my'  
document.forms[0] -- первая форма в документе
```

Любой элемент формы `form` можно получить аналогичным образом, используя свойство `form.elements`.



Например:

```
01 <body>  
02 <form name="my">  
03   <input name="one" value="1">  
04   <input name="two" value="2">  
05 </form>  
06  
07 <script>  
08 var form = document.forms.my; // можно document.forms[0]  
09  
10 var elem = form.elements.one; // можно form.elements[0]  
11  
12 alert(elem.value); // "один"  
13 </script>  
14 </body>
```

Может быть несколько элементов с *одинаковым именем*. В таком случае `form.elements[name]` вернет коллекцию элементов, например:

```
01 <body>  
02 <form>  
03   <input type="radio" name="age" value="10">  
04   <input type="radio" name="age" value="20">  
05 </form>  
06  
07 <script>  
08 var form = document.forms[0];  
09  
10 var elems = form.elements.age;  
11  
12 alert(elems[0].value); // 10, первый input  
13 </script>  
14 </body>
```

Эти ссылки не зависят от окружающих тегов. Элемент может быть «зарыт» где-то глубоко в форме, но он всё равно доступен через `form.elements`.

Спецификация: [HTMLFormElement](#) .



`form.name` тоже работает, но с ошибками

Получить доступ к элементам формы можно не только через `form.elements[name/index]`, но и проще: `form[index/name]`.

Этот способ — нестандартный. Он работает во всех браузерах, но в некоторых (напр. Firefox) — с ошибками. Когда вы удалите элемент, он все равно будет доступен как `form[name]`:

```
01 <form name="myform"> <input name="text"> </form>  
02  
03 <script>  
04 var form = document.forms.myform;
```

```

05
06 form.removeChild(form.text); // удаляем этот input
07
08 alert(form.elements.text); // undefined (правильно,
   т.к. удалён)
09 alert(form.text); // в Firefox возвращает элемент!
10 </script>

```

В этом примере элемент удален, но `form[name]` все равно ссылается на него. Чтобы не наступать на эти грабли, пользуйтесь синтаксисом `form.elements[name]`.

## От элементов к форме

По элементу можно получить его форму, используя свойство `element.form`.

Пример:

```

01 <body>
02 <form>
03   <input type="text" name="surname">
04 </form>
05
06 <script>
07 var form = document.forms[0];
08
09 var elem = form.elements.surname;
10
11 alert(elem.form == form); // true
12 </script>
13 </body>

```

Посмотрите также спецификацию о [HTMLInputElement](#) и других типах элементов.

## Дополнительные ссылки для элементов (кроме IE<9)

Все современные браузеры, включая IE9+, поддерживают дополнительные ссылки:

- `childElementCount` — число детей-элементов (`=children.length`)
- `firstElementChild` — первый потомок-элемент (`=children[0]`)
- `lastElementChild` — последний потомок-элемент (`=children[children.length-1]`)
- `nextElementSibling` — правый брат-элемент
- `previousElementSibling` — левый брат-элемент

Любые другие узлы, кроме элементов, при этом просто игнорируются. Например:

```

01 <body>
02   firstElementChild: <div>...</div>
03   <!-- комментарий -->
04   lastElementChild: <span>...</span>
05
06 <script>
07   alert(document.body.firstElementChild.nextElementSibling);
   // SPAN
08 </script>
09

```

Современные браузеры также поддерживают дополнительные интерфейсы для обхода DOM с фильтром по узлам: `NodeIterator`, `TreeFilter` и `TreeWalker`. Они были утверждены аж в 2000-м году, однако на практике оказались неудобными, и потому практически не применяются. Вы можете почитать о них в стандарте [DOM 2 Traversal](#) .

## Итого

Сверху в DOM можно войти либо через `document.documentElement` (тег HTML), либо через `document.body` (тег BODY).

По элементу DOM можно получить всех соседей через ссылки:

### **childNodes, children**

Список дочерних узлов.

### **firstChild, lastChild**

Первый и последний потомки

### **parentNode**

Родительский узел

### **previousSibling, nextSibling**

Соседи влево-вправо

Все навигационные ссылки доступны только для чтения и поддерживаются автоматически.

## Ссылки для таблиц

---

### Таблица TABLE

- `table.rows` — список строк TR таблицы.
- `table.caption/tHead/tFoot` — ссылки на элементы таблицы CAPTION, THEAD, TFOOT.
- `table.tBodies` — список элементов таблицы TBODY.

### Секция THEAD/TFOOT/TBODY

- `tbody.rows` — список строк TR секции.

### Строка TR

- `tr.cells` — список ячеек TD/TH
- `tr.sectionRowIndex` — номер строки в текущей секции THEAD/TBODY
- `tr.rowIndex` — номер строки в таблице

### Ячейка TD/TH

- `td.cellIndex` — номер ячейки в строке

## Для форм

---

### FORM

Форму можно получить как `document.forms[name/index]`.

### Элементы

Элементы в форме: `form.elements[name/index]`. Каждый элемент имеет ссылку на форму в свойстве `form`.

## Только элементы

В современных браузерах, включая IE9+, реализованы дополнительные свойства, работающие только для элементов:

- `childElementCount` — число детей-элементов
- `firstElementChild` — первый потомок-элемент
- `lastElementChild` — последний потомок-элемент
- `nextElementSibling` — правый брат-элемент
- `previousElementSibling` — левый брат-элемент

См. также

- [DOM Traversal](#)
- [Element Traversal Specification](#)

## Свойства узлов: тип, тег, содержимое и другие

В этой главе мы рассмотрим три основных свойства DOM-узлов: тип, тег и содержимое.

Мы увидим, какие значения они могут принимать и как с ними работать из JavaScript.

### Тип: `nodeType`

С тремя типами мы уже встречались. Это 1) элемент, 2) текстовый узел и 3) комментарий.

На самом деле типов узлов гораздо больше. Строго говоря, их 12, и они описаны в спецификации [DOM Уровень 1](#) :

```
01 interface Node {
02   // NodeType
03   const unsigned short    ELEMENT_NODE        = 1;
04   const unsigned short    ATTRIBUTE_NODE       = 2;
05   const unsigned short    TEXT_NODE            = 3;
06   const unsigned short    CDATA_SECTION_NODE   = 4;
07   const unsigned short    ENTITY_REFERENCE_NODE = 5;
08   const unsigned short    ENTITY_NODE          = 6;
09   const unsigned short    PROCESSING_INSTRUCTION_NODE =
10     7;
11   const unsigned short    COMMENT_NODE         = 8;
12   const unsigned short    DOCUMENT_NODE        = 9;
13   const unsigned short    DOCUMENT_TYPE_NODE   = 10;
14   const unsigned short    DOCUMENT_FRAGMENT_NODE = 11;
15   const unsigned short    NOTATION_NODE        = 12;
16   ...
}
```

Нам важны номера основных типов.

**Самые важные** — это `ELEMENT_NODE` под номером 1 и `TEXT_NODE` под номером 3.

**Тип узла содержится в его свойстве `nodeType`.**

Например, выведем все узлы-потомки `document.body`, *являющиеся*

элементами:

```
01 <body>
02 <div>Читатели:</div>
03 <ul>
04 <li>Вася</li>
05 <li>Петя</li>
06 </ul>
07
08 <!-- комментарий -->
09
10 <script>
11     var childNodes = document.body.childNodes;
12
13     for (var i=0; i<childNodes.length; i++) {
14
15         // отфильтровать не-элементы
16         if (childNodes[i].nodeType != 1) continue;
17
18         alert(childNodes[i]);
19
20     }
21 </script>
22 </body>
```



**Задача:** В инлайн скрипте lastChild.nodeType

Что выведет скрипт на этой странице?

Важность: 5

```
1 <!DOCTYPE HTML>
2 <html>
3 <body>
4 <script>
5     alert(document.body.lastChild.nodeType);
6 </script>
7 </body>
8 </html>
```

[Решение задачи "В инлайн скрипте lastChild.nodeType" »»](#)

## Тег: nodeName и tagName

Существует целых два свойства: nodeName и tagName, которые содержат название(тег) элемента узла.

**Название HTML-тега всегда находится в верхнем регистре.**

Например, для document.body:

```
1 alert( document.body.nodeName ); // BODY
2 alert( document.body.tagName ); // BODY
```



**Когда nodeName не в верхнем регистре?**

У браузера есть два режима обработки документа: HTML и XML-режим. Обычно используется режим HTML, XML-режим включается, когда браузер получает XML-документ через XMLHttpRequest(технология AJAX) или при наличии заголовка Content-Type: application/xml+xml.

В XML-режиме сохраняется регистр и nodeName может выдать «body» или даже «bOdY» — в точности как указано в документе.

Какая разница между tagName и nodeName ?



Разница отражена в названиях свойств, но неочевидна.

- Свойство `nodeName` определено для многих типов DOM-узлов.
- Свойство `tagName` — есть только у элементов (в IE<9 также у комментариев, это ошибка в браузере).

Иначе говоря, при помощи `tagName` мы можем работать только с элементами, а `nodeName` может что-то сказать и о других типах узлов.

Например:

```
01 <body><!-- комментарий -->
02
03 <script>
04   // для комментария
05   alert(document.body.firstChild.nodeName); // #comment
06   alert(document.body.firstChild.tagName); // undefined (в
    IE<9 воскл. знак "!")
07
08   // для документа
09   alert(document.nodeName); // #document, т.к. корень DOM
    -- не элемент
10   alert(document.tagName); // undefined
11 </script>
12 </body>
```

При работе только с узлами элементов имеет смысл использовать `tagName` — так короче 😊.



#### **Задача:** Найти следующий элемент

Напишите функцию `getNextElement(elem)`, которая **Важность: 5** возвращает следующий за `elem` узел-элемент (игнорирует остальные узлы).

Пример:

```
01 <div>Первый</div>
02 <!-- комментарий... -->
03 <p>Второй</p>
04
05 <script>
06   function getNextElement(elem) { /* ваш код */ }
07
08   alert(getNextElement(document.body.children[0]).tagName);
    // P
09   alert(getNextElement(document.body.lastChild)); //
    null
10 </script>
```

P.S. Функция должна работать максимально эффективно и учитывать возможности современных браузеров.

[Решение задачи "Найти следующий элемент" »»](#)

## innerHTML: содержимое элемента

Свойство `innerHTML` описано в спецификации HTML 5 — [embedded content](#) .

Оно позволяет получить HTML-содержимое узла в виде строки. В `innerHTML` можно и читать и писать.

Пример выведет на экран все содержимое `document.body`, а затем заменит его на другое:

```

01 <body>
02   <p>Параграф</p>
03   <div>Div</div>
04
05   <script>
06     alert( document.body.innerHTML ); // читаем текущее
    содержимое
07     document.body.innerHTML = 'Новый BODY!'; // заменяем
    содержимое
08   </script>
09
10 </body>

```

При чтении `innerHTML` HTML-код всегда будет валиден. При записи — можно записать что угодно, браузер поправит некорректный HTML-код:

```

1 <body>
2
3   <script>
4     document.body.innerHTML = '<b>тест'; // незакрытый тег
5     alert(document.body.innerHTML); // <b>тест</b>
    (исправлено)
6   </script>
7
8 </body>

```

`innerHTML` — очень полезное свойство и одно из самых часто используемых.

## Тонкости `innerHTML`

`innerHTML` не так прост, как может показаться, и таит в себе некоторые тонкости, которые могут сбить с толку новичка, а иногда и опытного программиста.

Ознакомьтесь с ними. Даже если этих сложностей у вас *пока* нет, эта информация отложится где-то в мозге и поможет, когда проблема появится.



Для таблиц в IE9- — `innerHTML` только для чтения

В Internet Explorer версии 9 и ранее, `innerHTML` доступно только для чтения для элементов `COL`, `COLGROUP`, `FRAMESET`, `HEAD`, `HTML`, `STYLE`, `TABLE`, `TBODY`, `TFOOT`, `THEAD`, `TITLE`, `TR`.

В частности, **в IE9- нельзя присвоить `innerHTML` табличным элементам, кроме ячеек (`TD/TH`).**



Добавление `innerHTML+=` осуществляет перезапись

Синтаксически, можно добавить текст к `innerHTML` через `+=`:

```

chatDiv.innerHTML += "<div>Привет<img src='smile.gif' />
!</div>";
chatDiv.innerHTML += "Как дела?";

```

На практике этим следует пользоваться с большой осторожностью, так как фактически происходит не добавление, а перезапись:

1. Удаляется старое содержание
2. На его место становится новое значение `innerHTML`.

Так как новое значение записывается с нуля, то **все изображения и другие ресурсы будут перезагружены**. В примере выше вторая строка перезагрузит `smile.gif`, который был до неё.

Если в `chatDiv` много текста, то эта перезагрузка будет очень заметна.

К счастью, есть и другие способы добавить содержимое, не используя `innerHTML`.



### Скрипты не выполняются

Если в `innerHTML` есть тег `script` — он не будет выполнен.

Пример — ниже:

```
1 <div id="my"></div>
2
3 <script>
4   var elem = document.getElementById('my');
5   elem.innerHTML =
6     'ТЕСТ<script>alert(1);</scr'+ 'ipt>';
7 </script>
```

В примере закрывающий тег `</scr'+ 'ipt>` разбит на две строки, т.к. иначе браузер подумает, что это конец скрипта. Вставленный скрипт не выполнится.

Исключение — IE<10, в нем вставляемый скрипт выполняются, если у него есть атрибут `defer` (это нестандартная возможность):

```
1 <div id="my"></div>
2
3 <script>
4   var elem = document.getElementById('my');
5   elem.innerHTML = 'Для IE<script
6     defer>alert(1);</scr'+ 'ipt>';
7 </script>
```



### IE<9 обрезает `style` и `script` в начале `innerHTML`

Если скрипт или стиль находятся в начале `innerHTML`, то старый IE уберет их. Визуальный эффект очевиден — стили не применяются.

[Открыть пример](#) (в IE9 будет режим IE8).

Смотрите также [innerHTML на MSDN](#) на эту тему.

## outerHTML: HTML узла целиком

Свойство `outerHTML` содержит HTML узла целиком.

Оно поддерживается всеми браузерами давно, кроме в Firefox — в нём начиная с версии 11.

Пример чтения `outerHTML`:

```
1 <div id="hi">Привет <b>Мир</b></div>
2
3 <script>
```

```

4 var div = document.getElementById('hi');
5
6 alert(div.outerHTML); // <div>Привет <b>Мир</b></div>
7 </script>

```

Можно и записать `outerHTML`, однако при этом новый текст будет вставлен *вместо узла*, а сама переменная не изменится.

Это отлично видно на примере:

```

01 <div id="hi">Привет <b>Мир</b></div>
02
03 <script>
04 var div = document.getElementById('hi');
05
06 div.outerHTML = '<p>Новый HTML!</p>';
07
08 alert(div.tagName); // DIV
09 alert(div.innerHTML); // Привет <b>Мир</b>
10 </script>

```

Как видно из примера выше, при замене `div.outerHTML` в переменной `div` остаётся старый узел, который теперь оторван от документа, но полностью «жизнеспособен».

## nodeValue/data: содержимое текстового узла

Свойство `innerHTML` есть только у узлов-элементов.

**Содержимое других узлов, например, текстовых или комментариев, доступно через два свойства: `nodeValue` и `data`.**

Его тоже можно читать и обновлять. Следующий пример демонстрирует это:

```

01 <body>
02   Привет
03   <!-- Комментарий -->
04   <script>
05     for (var i=0; i<document.body.childNodes.length; i++)
06     {
07       alert(document.body.childNodes[i].nodeValue);
08       alert(document.body.childNodes[i].data);
09     }
10     document.body.firstChild.data = "Здравствуйте!";
11   </script>
12 </body>

```

В этом примере выводятся последовательно:

1. Содержимое первого узла (текстового): Привет (2 раза).
2. Содержимое второго узла (комментария): Комментарий (2 раза).
3. Содержимого третьего узла (текста между комментарием и скриптом): (там пробелы, 2 раза)
4. Свойство `nodeValue=null` для узла `SCRIPT`, так как это узел-элемент. А вот `data=undefined`. Это единственное различие в поведении этих свойств.
5. Наконец, последний вызов заменит содержимое `document.body.firstChild`, и это тут же отразится в документе.

Кстати, после `SCRIPT` есть еще один текстовый узел, но на момент работы скрипта браузер дошел в разборе документа только до `SCRIPT`, поэтому

он не будет выведен.

## Другие свойства

У DOM-узлов есть свойства, зависящие от типа, например:

- `value` — значение для `INPUT`, `SELECT` или `TEXTAREA`
- `id` — идентификатор
- `href` — адрес ссылки
- ...многие другие...

Например:

```
1 <input type="text" id="my-input" value="значение">
2
3 <script>
4 var input = document.body.children[0];
5
6 alert(input.type); // "text"
7 alert(input.id); // "my-input"
8 alert(input.value); // значение
9 </script>
```

Полный список свойств можно получить из спецификации. В частности, для `input[type="text"]` она находится <http://www.w3.org/TR/html-markup/input.text.html>. Как правило, основные атрибуты элемента дают свойство с тем же названием.

## Итого

Основные свойства DOM-узлов:

### `nodeType`

Тип узла. Самые популярные типы: `"1"` - для элементов и `"3"` - для текстовых узлов. Только для чтения.

### `nodeName/tagName`

Название тега заглавными буквами. `nodeName` имеет специальные значения для узлов-неэлементов. Только для чтения.

### `innerHTML`

Внутреннее содержимое узла-элемента в виде HTML. Можно изменять.

### `outerHTML`

Полный HTML узла-элемента. При записи в `elem.outerHTML` переменная `elem` сохраняет старый узел.

### `nodeValue/data`

Содержимое текстового узла или комментария. Свойство `nodeValue` также определено и для других типов узлов. Можно изменять.

Узлы DOM также имеют другие свойства, в зависимости от тега.

Например, у `INPUT` есть свойства `value` и `checked`, а у `A` есть `href` и т.д.

Мы рассмотрим их далее.



### Задача: Тег в комментарии

Что выведет этот код?

Важность: 3

```
1 <script>
2 var body = document.body;
3
```

```
4 | body.innerHTML = "<!--" + body.tagName + "-->";
5 |
6 | alert(body.firstChild.data); // что выведет?
7 | </script>
```

[Решение задачи "Тег в комментарии" »»](#)

## Атрибуты и "свои" свойства

У DOM-узлов бывают не только *свойства*, но и *атрибуты*.

Между ними есть кое-что общее, поэтому иногда их путают. Но на самом деле, атрибуты и свойства — совершенно разные вещи.

### «Свои» свойства

**Узел DOM - это объект, поэтому, как и любой объект в JavaScript, он может содержать пользовательские свойства и методы.**

Например, создадим в `document.body` новое свойство и запишем в него объект:

```
1 | document.body.myData = {
2 |   name: 'Петр',
3 |   familyName: 'Петрович'
4 | };
5 |
6 | alert(document.body.myData.name); // Петр
```

Можно добавить и новую функцию:

```
1 | document.body.sayHi = function() {
2 |   alert(this.nodeName);
3 | }
4 |
5 | document.body.sayHi(); // BODY, выполнялась с правильным
   | this
```

Такие свойства и методы видны только в JavaScript и никак не влияют на отображение соответствующего тега.

Эти свойства можно даже перечислить с помощью `for...in` (свойства будут все, и встроенные тоже):

```
1 | document.body.custom = 5;
2 |
3 | var list = [];
4 | for (var key in document.body) {
5 |   list.push(key);
6 | }
7 |
8 | alert( list.join('\n') );
```

Пользовательские DOM-свойства:

- Могут иметь любое значение.
- Названия свойств *чувствительны* к регистру.
- Работают за счет того, что DOM-узлы являются объектами JavaScript

## Атрибуты

Узлы DOM, с другой стороны, являются HTML-элементами, у которых

есть *атрибуты*.

Доступ к атрибутам осуществляется при помощи стандартных методов:

- `elem.hasAttribute(name)` - проверяет наличие атрибута
- `elem.getAttribute(name)` - получает значение атрибута
- `elem.setAttribute(name, value)` - устанавливает атрибут
- `elem.removeAttribute(name)` - удаляет атрибут



Атрибуты неправильно работают в IE<8 и в IE8 в режиме совместимости:

- Существуют только методы `getAttribute` и `setAttribute`.
- Фактически, они изменяют DOM-свойства, а не атрибуты.
- Атрибуты и свойства в IE<8 объединены. Иногда это приводит к странным результатам, но способы работы с атрибутами, о которых мы здесь говорим, работают правильно.

В отличие от свойств, атрибуты:

- Могут быть только строками.
- Их имя *нечувствительно* к регистру(т.к. это HTML)
- Видны в `innerHTML` (за исключением старых IE)
- Все атрибуты элемента можно получить с помощью свойства `attributes`, которое содержит псевдо-массив объектов типа `Attr` .

Например, рассмотрим этот HTML-код:

```
<body>
  <div about="Elephant" class="smiling"></div>
</body>
```

Пример ниже устанавливает атрибуты и демонстрирует их особенности.

```
01 <body>
02   <div about="Elephant" class="smiling"></div>
03
04   <script>
05     var div = document.body.children[0];
06     alert( div.getAttribute('ABOUT') ); // (1)
07
08     div.setAttribute('Test', 123); // (2)
09     alert( document.body.innerHTML ); // (3)
10
11     var attrs = div.attributes; // (4)
12     for (var i=0; i<attrs.length; i++) {
13       alert(attrs[i].name + " = " + attrs[i].value);
14     }
15   </script>
16 </body>
```

При запуске кода выше обратите внимание:

1. `getAttribute('ABOUT')` использует имя атрибута в верхнем регистре, но это не имеет значения, т.к имена нечувствительны к регистру.
2. Вы можете записать в атрибут строку или другое значение, которые будет превращено в строку. Объект, например, будет автоматически сконвертирован, но у IE<9 с этим проблемы, поэтому придерживайтесь примитивов.

3. В innerHTML появился новый атрибут "test".
4. Коллекция attributes содержит все атрибуты в виде объектов класса Attr со свойствами name и value.

## Синхронизация свойств и атрибутов

Каждый тип узлов DOM имеет свои стандартные свойства.

Например, свойства тега 'A' описаны в спецификации DOM:

[HTMLAnchorElement](#) .

Например, у него есть свойство "href". Кроме того, он имеет "id" и другие свойства, общие для всех элементов, которые описаны в спецификации в [HTMLElement](#) .

**Стандартные свойства DOM синхронизируются с атрибутами.**

### id

---

Например, браузер синхронизирует атрибут "id" со свойством id.

Изменим одно - поменяется и другое:

```
1 <script>
2   document.body.setAttribute('id','la-la-la');
3   alert(document.body.id); // la-la-la
4 </script>
```

### href

---

**Синхронизация не гарантирует одинакового значения.** Для примера, посмотрим, что произойдет с атрибутом "href" при изменении свойства:

```
01 <a href="#"></a>
02 <script>
03   var a = document.body.children[0];
04
05   a.href = '/';
06
07   alert( 'атрибут:' + a.getAttribute('href') ); // '/'
08   alert( 'свойство:' + a.href ); // полный URL
09
10 </script>
```

Это происходит потому, что атрибут может быть любым, а свойство href, [в соответствии со спецификацией W3C](#) , должно быть полной ссылкой.

Есть и другие свойства-атрибуты, которые не копируются в точности.

Например, таково свойство input.checked:

```
1 <input type="checkbox" checked>
2
3 <script>
4   var input = document.body.children[0];
5
6   alert( input.checked ); // true <-- может быть только
   true/false
7
8   alert( input.getAttribute('checked') ); // пустая строка
9 </script>
```

### value

---

**А еще есть встроенные свойства, которые синхронизируются в одну сторону.**



Например, `input.value` синхронизируется только из *атрибута* в свойство:

```
01 <body>
02   <input type="text" value="markup">
03   <script>
04     var input = document.body.children[0];
05
06     input.setAttribute('value', 'new');
07
08     alert( input.value ); // 'new', input.value изменено
09   </script>
10 </body>
```

Но атрибут не синхронизируется из свойства:

```
01 <body>
02   <input type="text" value="markup">
03   <script>
04     var input = document.body.children[0];
05
06     input.value = 'new';
07
08     alert(input.getAttribute('value')); // 'markup', не
    изменилось!
09   </script>
10 </body>
```

В результате атрибут `value` хранит оригинальное (исходное) значение даже после того, как пользователь заполнил поле и свойство изменилось. Текущее же значение хранится в *свойстве* `value`.

**Изначальное значение может использоваться для проверки, было ли изменено значение `input` и, при необходимости, для того, чтобы отменить изменения.**

## class/className

**Исключение: атрибут `"class"` соответствует свойству `className`.**

Так как слово `"class"` является зарезервированным словом в Javascript, то атрибуту `"class"` соответствует свойство `className`.

```
1 <body>
2   <script>
3     document.body.setAttribute('class', 'big red bloom');
4
5     alert( document.body.className ); // ^^^
6   </script>
7 </body>
```

Приведенный пример работает во всех современных браузерах, т.к. в них это соответствие выполняется точно.

Но он не работает в IE<9, точнее — не работает установка класса.

Поэтому рекомендуется, по возможности, не использовать атрибут `"class"`, а использовать только свойство `className`.

Итак:

- Стандартные свойства синхронизируются с атрибутами.
- ..Но при этом свойство не всегда равно атрибуту, например `href` — по стандарту полная ссылка.

- ..Некоторые свойства меняются независимо от атрибута, например value, т.е. синхронизация односторонняя.
- Атрибуту class соответствует свойство className.

Кстати, атрибуту for (<label for="...">) тоже соответствует свойство с другим названием (htmlFor). Больше несовпадающих имён нет.



**Задача:** Получите пользовательский атрибут

Важность: 5

1. Получите div в переменную.
2. Получите значение атрибута "data-widget-name" в переменную.
3. Выведите его.

Документ:

```
1 <body>
2
3   <div data-widget-name="menu">Выберите
   жанр</div>
4
5   <script>/* ... */</script>
6 </body>
```

Исходный код:

[http://learn.javascript.ru/play/tutorial/browser/dom/custom\\_attribute.html](http://learn.javascript.ru/play/tutorial/browser/dom/custom_attribute.html)

.

[Решение задачи "Получите пользовательский атрибут" »»](#)

## Особенности старых IE

Для полноты картины ознакомимся с проблемами версий IE<9.

1. **Во-первых, версии IE<9 синхронизируют все свойства и атрибуты, а не только стандартные:**

```
1 document.body.setAttribute('my', 123);
2
3 alert( document.body.my ); // 123 в IE<9
```

При этом даже тип данных не меняется. Атрибут не становится строкой, как ему положено.

2. **Во-вторых, в IE8 это исправлено, но в IE<8 (или в IE8 в режиме совместимости с IE7) свойства и атрибуты — одно и то же.**

Поэтому возникают забавные казусы.

Например, названия свойств регистрозависимы, а названия атрибутов - нет. Что будет если два свойства имеют одинаковое имя в разном регистре? Как поведет себя соответствующий атрибут?

```
1 document.body.abba = 1; // задаем свойство
2 document.body.ABBA = 5; // задаем свойство, теперь уже
  прописными буквами
3
4 // запрашиваем атрибут в смешанном регистре
5 alert( document.body.getAttribute('AbBa') ); // что
  должен вернуть браузер?
```

Браузер выходит из ситуации, возвращая первое назначенное свойство(abba). Также, в IE<9 существует второй параметр для `getAttribute`, который делает его чувствительным к регистру. Подробнее тут:[MSDN getAttribute](#) .

3. Также есть еще одно следствие смешения свойств с атрибутами в старых IE: изменение атрибута "class" в IE<9 не меняет класс. Эта проблема нас никогда не коснется, если мы работаем со *свойством* `className`.

Для того, чтобы избежать проблем с IE, используйте атрибуты правильно.

Другими словами, всегда старайтесь использовать свойства, а атрибуты - только там, где это *действительно* нужно.

А *действительно* нужны атрибуты лишь в трёх случаях:

1. Когда нужно получить пользовательский HTML-атрибут, потому что он не синхронизирован со свойством.
2. Когда нужно получить «оригинальное значение» стандартного HTML-атрибута, например, `<INPUT value="...">`. Но в IE<8 это не сработает, т.к. там атрибуты смешаны со свойствами, и значение атрибута и свойства будет одинаковое. **Получить доступ именно к атрибуту в IE6,7 нельзя.**
3. Когда нужно получить список всех атрибутов, включая пользовательские. Для этого используется коллекция `attributes`.

## Итого

Таблица сравнений для атрибутов и свойств:

Свойства	Атрибуты
Любое значение	Строка
Названия регистрозависимы	Не чувствительны к регистру
Не видны в <code>innerHTML</code>	Видны в <code>innerHTML</code>
Стандартные свойства и атрибуты синхронизируются, некоторые - в одну сторону. Пользовательские - не синхронизируются.	
В IE<8 и IE8 в режиме совместимости, свойства и атрибуты смешаны.	

Если вы хотите использовать собственные атрибуты в HTML, то помните, что атрибуты с именем, начинающимся на `data-` валидны в HTML5.

Современные браузеры поддерживают особый доступ к таким атрибутам через DOM: [Свойство `dataset` для `data-\*` атрибутов](#).

В реальной жизни, в 98% случаев используются свойства DOM.

Оставшиеся 2% это:

1. Когда нужно получить пользовательский HTML-атрибут, потому что он не синхронизирован со свойством..
2. Когда нужно получить именно значение атрибута, потому что в свойстве уже не совсем то (`href` для ссылки) или оно изменилось

(value у INPUT).

# Интерактивное путешествие по DOM

Привет, отважные путешественники!

Вам предоставляется возможность побродить в чудесном мире DOM.

Путь начинается с `document.documentElement` (он же - тег `<HTML>`).

Ниже находится документ и кнопки управления. Сверяйте свои движения с картой (документом).

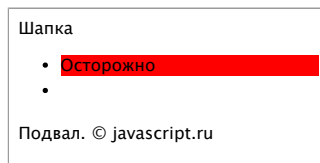
Все может оказаться совсем не так просто. Как правило, уже первые движения вызывают вопрос «где я нахожусь?». Чтобы на него ответить - посмотрите на «Текущий узел» и сверьте его с HTML.

Удачи и новых открытий!

## HTML-разметка документа для путешествия

```
01 <!DOCTYPE HTML>
02 <html>
03   <head>
04     <meta charset="utf-8">
05     <title>Документ</title>
06   </head>
07   <body>
08     <div id="header">Шапка</div>
09
10     <ul>
11       <li style="background-color:red">Осторожно</li>
12       <li class="info"><input type="text"
13         value="Информация"></li>
14     </ul>
15     <div id="footer">Подвал. &copy; javascript.ru</div>
16     <!-- комментарий -->
17   </body>
18 </html>
```

### Так выглядит документ



### Здесь стоите Вы

Текущий элемент

nodeType:

tagName:

nodeValue:

От текущего элемента могу двигаться:

Получить:

Для более удобного перемещения откройте документ в [новом окне](#) и походите по нему в Firebug (вкладка HTML). Посмотрите разницу между показанным там DOM и реальным.

# Поиск: getElement\* и querySelector\*

Прямая навигация от родителя к потомку удобна, если элементы рядом. А если нет?

Для этого в DOM есть дополнительные методы поиска.

## getElementById

У каждого DOM-элемента может быть атрибут `id`. Его значение должно быть уникальным для документа.

**Вызов `document.getElementById(id)` возвращает элемент с данным `id`.**

Если не найден, то `null`.

Например:

```
01 <body>
02
03   <div id="info">Информация</div>
04
05   <script>
06
07       var div = document.getElementById('info');
08
09       alert( div.innerHTML ); // Информация
10
11   </script>
12 </body>
```

Важно, что в документе может быть только один элемент с данным `id`. Конечно, можно нарушить это правило и создать много элементов с одинаковым идентификатором, но в таком случае поведение метода `getElementById` будет непредсказуемым.



### Неявные переменные с именем `id`

Браузер автоматически создаёт переменные для элементов с `id`, через которые к ним можно обратиться.

Для примера, запустите следующий код. Выведется элемент.

```
1 <div id="test">тест</div>
2 <script>
3   alert(test); // DIV
4 </script>
```

Это поведение соответствует [стандарту](#). Оно существует, в первую очередь, для совместимости, как осколок далёкого прошлого.

## getElementsByTagName

Метод `elem.getElementsByTagName(tag)` ищет все элементы с заданным тегом `tag` внутри элемента `elem` и возвращает их в виде списка.

Регистр тега не имеет значения.

Можно искать и в элементе и в документе:

```
// получить все div-элементы
var elements = document.getElementsByTagName('div');
```

Найдём все элементы input внутри таблицы:

```
01 <table id="age-table">
02   <tr>
03     <td>Ваш возраст:</td>
04
05     <td>
06       <label>
07         <input type="radio" name="age" value="young"
checked> младше 18
08       </label>
09       <label>
10         <input type="radio" name="age" value="mature"> от
18 до 50
11       </label>
12       <label>
13         <input type="radio" name="age" value="senior">
старше 60
14       </label>
15     </td>
16   </tr>
17
18 </table>
19
20 <script>
21
22   var tableElem = document.getElementById('age-table');
23   var elements = tableElem.getElementsByTagName('input');
24
25   for (var i=0; i<elements.length; i++) {
26     var input = elements[i];
27     alert(input.value + ': ' + input.checked);
28   }
29
30 </script>
```

Можно получить все элементы, передав звездочку '\*' вместо тега:

```
// получить все элементы документа
var allElems = document.getElementsByTagName('*');
```

Если хочется получить только один элемент — можно указать индекс сразу же:

```
var element = document.getElementsByTagName('input')[0]
```

## getElementsByTagName

Вызов `document.getElementsByTagName(name)` позволяет получить все элементы с данным атрибутом name.

Например, все элементы с именем age:

```
var elems = document.getElementsByTagName('age');
```

До появления стандарта HTML5 этот метод возвращал только те элементы, в которых предусмотрена поддержка атрибута name, в частности: `iframe`, `a`, `input` и другими.

В современных браузерах тег не имеет значения, но старое поведение

можно увидеть, попробовав пример ниже в IE10 и до версии 10:

```
1 <input name="test">
2
3 <div name="test"></div>
4
5 <script>
6 var elems = document.getElementsByName('test');
7
8 alert(elems.length); // 2 в современных браузерах, 1 в
  IE<10
9 </script>
```

В IE9- метод не найдёт элементы, для которых в стандарте нет атрибута name.

## getElementsByClassName

Вызов `elem.getElementsByClassName(className)` возвращает коллекцию элементов с классом `className`. Находит элемент и в том случае, если у него несколько классов, а искомый - один из них.

Поддерживается всеми современными браузерами, кроме IE8-.

Например:

```
1 <div class="article">Статья</div>
2 <div class="long article">Длинная статья</div>
3
4 <script>
5 var articles = document.getElementsByClassName('article');
6 alert( articles.length ); // 2, найдёт оба элемента
7 </script>
```

Как и `getElementsByTagName`, этот метод может быть вызван и в контексте DOM-элемента и в контексте документа.

## querySelectorAll

Вызов `elem.querySelectorAll(cssQuery)` возвращает все элементы внутри `elem`, удовлетворяющие CSS-селектору `cssQuery`.

Он работает во всех современных браузерах, включая IE9+. Также работает и в IE8, но с некоторыми ограничениями:

1. IE8 должен быть именно в режиме IE8, а не в режиме совместимости.
2. В IE8 синтаксис `cssQuery` должен соответствовать не CSS 3, а CSS 2.1. Не так мощно, конечно, но этого хватает для большинства случаев.

Следующий запрос получает все элементы LI, которые являются последними потомками своих UL. Это будет работать и в IE8.

```
01 <ul>
02   <li>Этот</li>
03   <li>тест</li>
04 </ul>
05 <ul>
06   <li>полностью</li>
07   <li>пройден</li>
08 </ul>
09 <script>
```

```

10     var elements = document.querySelectorAll('ul > li:last-child');
11
12     for (var i=0; i<elements.length; i++) {
13         alert(elements[i].innerHTML ); // "тест", "пройден"
14     }
15 </script>

```

## querySelector

То же самое, что `elem.querySelectorAll(cssQuery)`, но возвращает только первый элемент.

Фактически, эквивалентен `elem.querySelectorAll(cssQuery)[0]`, но быстрее, так как ищутся не все элементы, а только первый.

## matchesSelector

Вызов `elem.matchesSelector(css)` проверяет, удовлетворяет ли `elem` селектору `css`.

Он возвращает `true` либо `false`.

**Спецификация** с ним ещё не утверждена, поэтому браузеры используют префиксы для его поддержки.

Например:

```

01 <div id="test">
02   <a href="http://example.com/file.zip" id="link">...</a>
03 </div>
04
05 <script>
06 var testElem = document.documentElement;
07 var matchesSelector = testElem.matchesSelector
08   || testElem.webkitMatchesSelector
09   || testElem.mozMatchesSelector
10   || testElem.msMatchesSelector
11   || testElem.oMatchesSelector;
12
13 var elem = document.getElementById('link');
14
15 alert( matchesSelector.call(elem, '#test a[href$="zip"]')
16 ); // true
16 </script>

```

Поддерживается во всех современных браузерах, не поддерживается в IE8-.

## XPath в современных браузерах

Для поиска в XML-документах существует [язык запросов XPath](#) .

Он очень мощный, во многом мощнее CSS, но сложнее. Например, запрос для поиска элементов H2, содержащих текст "XPath", будет выглядеть так: `//h2[contains(., "XPath")]`.

**Все современные браузеры, кроме IE, поддерживают XPath с синтаксисом, близким к [описанному в MDN](#) .**

Найдем заголовки с текстом XPath в текущем документе:

```

1 var result = document.evaluate("//h2[contains(.,
  'XPath')]", document.documentElement,

```



```

    null,
    XPathResult.ORDERED_NODE_SNAPSHOT_TYPE, null);
2
3 for (var i=0; i<result.snapshotLength; i++) {
4     alert(result.snapshotItem(i).outerHTML);
5 }

```

IE тоже поддерживает XPath, но эта поддержка не соответствует стандарту и работает только для XML-документов, например, полученных с помощью XMLHttpRequest (AJAX). Для обычных же HTML-документов XPath в IE не поддерживается.

## Внутреннее устройство поисковых методов

Несмотря на схожесть в синтаксисе, методы внутри устроены очень по-разному. Эту разницу нужно знать, так как она сильно влияет на скорость работы, а иногда — и на результаты.

### document.getElementById(id)

Браузер поддерживает у себя внутреннее соответствие id -> элемент. Поэтому нужный элемент возвращается сразу. Это очень быстро.

### elem.querySelector(query), elem.querySelectorAll(query)

Чтобы найти элементы, удовлетворяющие поисковому запросу, браузер не использует никаких сложных структур данных.

Он просто перебирает все подэлементы внутри элемента elem(или по всему документу, если вызов в контексте документа) и проверяет каждый элемент на соответствие запросу query.

Вызов `querySelector` прекращает перебор после первого же найденного элемента, а `querySelectorAll` собирает найденные элементы в «псевдомассив»: внутреннюю структуру данных, по сути аналогичную массиву JavaScript.

Этот перебор происходит очень быстро, так как осуществляется непосредственно движком браузера, а не JavaScript-кодом.

Оптимизации:

- В случае поиска по ID: `elem.querySelector('#id')`, большинство браузеров оптимизируют поиск, используя вызов `getElementById`.
- Последние результаты поиска сохраняются в кеше. Но это до тех пор, пока документ как-нибудь не изменится.

### elem.getElementsByTagName(...)

**Результаты поиска `getElementsByTagName` — живые! При изменении документа — изменяется и результат запроса.**

Например, найдём все div при помощи `querySelectorAll` и `getElementsByTagName`, а потом изменим документ:

```

01 <div></div>
02 <script>
03     var resultGet = document.getElementsByTagName('div');
04     var resultQuery = document.querySelectorAll('div');
05
06     alert(resultQuery.length + ', ' + resultGet.length); //

```

```

1, 1
07
08 document.body.innerHTML = ''; // удалить всё содержимое
   BODY
09
10 alert(resultQuery.length + ', ' + resultGet.length); //
   1, 0
11 </script>

```

Как видно, длина коллекции, найденной через `querySelectorAll`, осталась прежней. А длина списка, возвращённого `getElementsByTagName`, изменилась.

Дело в том, что результат запросов `getElementsByTagName*` — это не массив, а специальный объект, имеющий тип `NodeList` или `HTMLCollection`. Он похож на массив, так как имеет нумерованные элементы и длину, но внутри это не готовый список, а «живой поисковой запрос».

Собственно поиск выполняется только при обращении к элементам списка или к его длине.

## Алгоритмы `getElementsByTagName*`

Поиск `getElementsByTagName*` наиболее сложно сделать эффективно, так как его результат — «живая» коллекция, она должна быть всегда актуальной для текущего состояния документа.

```

1 var elems = document.getElementsByTagName('div');
2 alert( elems[0] );
3 // изменили документ
4 alert( elems[0] ); // результат может быть уже другой

```

Можно искать заново при каждой попытке получить элемент из `elems`. Тогда результат будет всегда актуален, но поиск будет работать уж слишком медленно. Да и зачем? Ведь, скорее всего, документ не поменялся.

**Чтобы производительность `getElementsByTagName*` была достаточно хорошей, активно используется кеширование результатов поиска. Для этого есть два основных способа: назовём их условно «Способ Firefox» (Firefox, IE) и «Способ WebKit» (Chrome, Safari, Opera).**

Для примера, рассмотрим поиск в произвольном документе, в котором есть 1000 элементов `div`.

Посмотрим, как будут работать браузеры, если нужно выполнить следующий код:

```

1 // вместо document может быть любой элемент
2 var elems = document.getElementsByTagName('div');
3 alert( elems[0] );
4 alert( elems[995] );
5 alert( elems[500] );
6 alert( elems.length );

```

### Способ Firefox

Перебрать подэлементы `document.body` в порядке их появления в поддереве. Запоминать *все найденные элементы* во внутренней структуре данных, чтобы при повторном обращении обойтись без поиска.

Разбор действий браузера по строкам:

1. Браузер создаёт пустую «живую коллекцию» `elems`. Пока ничего не ищет.
2. Перебирает элементы, пока не найдёт первый `div`. Запоминает его и возвращает.
3. Перебирает элементы дальше, пока не найдёт элемент с индексом 995. Запоминает все найденные.
4. Возвращает ранее запомненный элемент с индексом 500, без дополнительного поиска!
5. Продолжает обход поддерева с элемента, на котором остановился (995) и до конца. Запоминает найденные элементы и возвращает их количество.

### Способ WebKit

Перебирать подэлементы `document.body`. Запоминать только один, *последний найденный*, элемент, а также, по окончании перебора — длину коллекции.

Здесь кеширование используется меньше.

Разбор действий браузера по строкам:

1. Браузер создаёт пустую «живую коллекцию» `elems`. Пока ничего не ищет.
2. Перебирает элементы, пока не найдёт первый `div`. Запоминает его и возвращает.
3. Перебирает элементы дальше, пока не найдёт элемент с индексом 995. Запоминает его и возвращает.
4. Браузер запоминает только последний найденный, поэтому не помнит об элементе 500. Нужно найти его перебором поддерева. Этот перебор можно начать либо с начала — вперед по поддереву, 500й по счету) либо с элемента 995 — назад по поддереву, 495й по счету. Так как назад разница в индексах меньше, то браузер выбирает второй путь. Запоминает теперь уже 500й элемент.
5. Продолжает обход поддерева с 500го элемента и до конца. Запоминает число найденных элементов и возвращает его.

Основное различие — в том, что Firefox запоминает все найденные, а Webkit — только последний. Таким образом, «метод Firefox» требует больше памяти, но гораздо эффективнее при повторном доступе к предыдущим элементам.

А «метод Webkit» ест меньше памяти и при этом работает не хуже в самом важном и частом случае — последовательном переборе коллекции, без возврата к ранее выбранным.

### Запомненные элементы сбрасываются при изменениях DOM.

Документ может меняться. При этом, если изменение может повлиять на результаты поиска, то запомненные элементы необходимо сбросить.

Например, добавление нового узла `div` сбросит запомненные элементы `elem.getElementsByTagName('div')`.

Сбрасывание запомненных элементов при изменении документа выполняется интеллектуально.

1. Во-первых, при добавлении элемента будут сброшены только те

коллекции, которые могли быть затронуты обновлением. Например, если в документе есть два независимых раздела `<section>`, и поисковая коллекция привязана к первому из них, то при добавлении во второй — она сброшена не будет.

Если точнее — будут сброшены все коллекции, привязанные к элементам вверх по иерархии от непосредственного родителя нового `div` и выше. И только они.

- Во-вторых, если добавлен только `div`, то не будут сброшены запомненные элементы для поиска по другим тегам, например `elem.getElementsByTagName('a')`. Если же добавлено целое поддерево разных элементов, то будут сброшены все соответствующие коллекции, но тоже ничего лишнего.
- ...И, конечно же, не любые изменения DOM приводят к сбросу, а только те, которые могут повлиять на список. Если где-то добавлен новый атрибут — с поиском по тегу ничего не произойдёт.

Прочие поисковые методы, такие как `getElementsByClassName` тоже сбрасывают кеш при изменениях интеллектуально.

Разницу в алгоритмах поиска легко увидеть. Посмотрите сами:

```
01 <script>
02   for(var i=0; i<10000;i++) document.write('<span>
    </span>');
03
04   var elements =
    document.getElementsByTagName('span');
05   var len = elements.length, el;
06
07   var d = new Date;
08   for(var i = 0; i<len; i++) elements[i];
09   alert("Последовательно: "+ (new Date - d) + "мс");    //
    (1)
10
11   var d = new Date;
12   for(var i = 0; i<len; i+=2) elements[i], elements[len-i-
    1];
13   alert("Вразнобой: "+ (new Date - d) + "мс");    // (2)
14 </script>
```

В примере выше первый цикл проходит элементы последовательно. А второй — идет по шагам: один с начала, потом один с конца, потом ещё один с начала, ещё один — с конца, и так далее.

Количество обращений к элементам одинаково.

- В браузерах, которые запоминают все найденные (Firefox, IE) — скорость будет одинаковой.
- В браузерах, которые запоминают только последний (Webkit, Opera) — разница будет порядка 100 и более раз, так как браузер вынужден бегать по дереву при каждом запросе заново.

## Практика



**Задача:** Длина коллекции после удаления элементов

Вот небольшой документ:

Важность: 5

```
1 <ul id="menu">
```

```

2   <li>Главная страница</li>
3   <li>Форум</li>
4   <li>Магазин</li>
5 </ul>

```

1. Что выведет следующий код (простой вопрос)?

```

1 var lis = document.getElementsByTagName('li');
2
3 document.body.innerHTML = "";
4
5 alert(lis.length);

```

2. А такой код (вопрос посложнее)?

```

1 var menu = document.getElementById('menu');
2 var lis = menu.getElementsByTagName('li');
3
4 document.body.innerHTML = "";
5
6 alert(lis.length);

```

Решение задачи "Длина коллекции после удаления элементов"

»»

Задачи ниже будут использовать следующий HTML-код:

```

01 <!DOCTYPE HTML>
02 <html>
03   <head>
04     <meta charset="utf-8">
05   </head>
06
07   <body>
08     <form name="search">
09       <label>Поиск по сайту: <input type="text"
name="search"></label>
10       <input type="submit" value="Искать!">
11     </form>
12
13     <hr>
14
15     <form name="search-person">
16       Поиск по посетителям:
17       <table id="age-table">
18         <tr>
19           <td>Возраст:</td>
20           <td id="age-list">
21             <label><input type="radio" name="age"
value="young">до 18</label>
22             <label><input type="radio" name="age"
value="mature">18-50</label>
23             <label><input type="radio" name="age"
value="senior">более 50</label>
24           </td>
25         </tr>
26
27         <tr>
28           <td>Дополнительно:</td>
29           <td>
30             <input type="text" name="info[0]">
31             <input type="text" name="info[1]">
32             <input type="text" name="info[2]">
33           </td>
34         </tr>
35       </table>
36     </form>
37

```

```
38     <input type="submit" value="Искать!">
39   </form>
40 </body>
41
42 </html>
```



### Задача: Поиск элементов

В документе

Важность: 4

<http://learn.javascript.ru/play/tutorial/browser/dom/searchTask.html>..

Найдите (получите в переменную):

1. Все элементы `label` внутри таблицы. Должно быть 3 элемента.
2. Первую ячейку таблицы (со словом "Возраст").
3. Вторую форму в документе.
4. Форму с именем `search`, без использования её позиции в документе.
5. Элемент `input` в форме с именем `search`. Он там один.
6. Элемент с именем `info[0]`, без точного знания его позиции в документе.
7. Элемент с именем `info[0]`, внутри формы с именем `search-person`.

[Решение задачи "Поиск элементов" »»](#)



### Задача: Проверка, находится ли элемент внутри другого

Для документа

Важность: 5

<http://learn.javascript.ru/play/tutorial/browser/dom/searchTask.html>.

Напишите функцию `checkInsideTable(elem)`, которая будет возвращать `true`, если произвольный элемент `elem` находится внутри таблицы `table#age-table`.

Если это не так, или такого элемента нет, то функция должна вернуть `false`.

Например:

```
checkInsideTable(document.getElementById('age-list'));
// true
checkInsideTable(document.forms.search);           //
false
checkInsideTable(document.getElementById('non-existent-id')); // false
```

**Предложите несколько вариантов решения этой задачи.**

[Решение задачи "Проверка, находится ли элемент внутри другого" »»](#)

## Отобразите число потомков



### Задача: Дерево

Есть дерево:

Важность: 5

<http://learn.javascript.ru/play/tutorial/browser/dom/treeSource.html>.

Напишите код, который добавит каждому элементу списка(LI) количество вложенных в него элементов. Узлы нижнего уровня — пропускайте.

Добавьте ваш скрипт в конец BODY.

Результат:

- Животные [9]
  - Млекопитающие [4]
    - Коровы
    - Ослы
    - Собаки
    - Тигры
  - Другие [3]
    - Змеи
    - Птицы
    - Ящерицы
- Рыбы [5]
  - Аквариумные [2]
    - Гуппи
    - Скалярии
  - Морские [1]
    - Морская форель

[Решение задачи "Дерево" »»](#)

## Дополнительно



### Задача: Сравнение количества элементов

Для любого документа сделаем следующее:

Важность: 5

```
var aList1 = document.getElementsByTagName('a'),  
var aList2 = document.querySelectorAll('a');
```

Что произойдёт со значениями `aList1.length`, `aList2.length`, если в документе вдруг появится ещё одна ссылка

```
<a href="#">...</a>?
```

[Решение задачи "Сравнение количества элементов" »»](#)



### Задача: Бенчмаркинг методов поиска в DOM

Какой метод поиска элементов работает быстрее:

Важность: 2

`getElementsByTagName(tag)` или  
`querySelectorAll(tag)`?

Напишите код, который измеряет разницу между ними.

Исходный документ:

<http://learn.javascript.ru/play/tutorial/browser/dom/speed-selector-src/index.html>

*P.S. В задаче есть подвох, все не так просто. Если разница больше 10 раз — вы решили ее неверно. Тогда подумайте, почему такое может быть.*

[Решение задачи "Бенчмаркинг методов поиска в DOM" »»](#)



### Задача: Получить второй LI

Есть длинный список `ul`:

Важность: 5

```
1 <ul>  
2   <li>...</li>  
3   <li>...</li>  
4   <li>...</li>  
5   ...  
6 </ul>
```

Как наиболее эффективно получить второй LI?

[Решение задачи "Получить второй LI" »»](#)

## Итого

Есть 5 основных способов получения элементов DOM:

Метод	Ищет по..	Ищет внутри элемента?	Поддержка
<code>getElementById</code>	<code>id</code>	-	езде
<code>getElementsByName</code>	<code>name</code>	-	езде
<code>getElementsByTagName</code>	тег или <code>'*'</code>		езде
<code>getElementsByClassName</code>	классу		езде, IE9+
<code>querySelector(All)</code>	CSS-селектор		езде, CSS 2.1 в IE8+

- Дополнительно есть метод `elem.matchesSelector(css)`, который проверяет, удовлетворяет ли элемент CSS-селектору. Он поддерживается большинством браузеров в префиксной форме (`ms`, `moz`, `webkit`).
- XPath поддерживается большинством браузеров, кроме IE, даже 9й версии. Кроме того, как правило, `querySelector` удобнее. Поэтому он используется редко.

## Добавление и удаление узлов

Изменение DOM — ключ к созданию «живых» страниц.

В этой главе мы рассмотрим, как создавать новые элементы «на лету» и заполнять их данными.

### Создание элементов: `createElement`

Для создания элементов используются следующие методы документа:

#### `document.createElement(tag)`

Создает новый элемент с указанным тегом:

```
var div = document.createElement('div');
```

#### `document.createTextNode(text)`

Создает новый *текстовый* узел с данным текстом:

```
var textElem = document.createTextNode('Тут был я');
```

Новому элементу тут же можно поставить свойства:

```
1 var newDiv = document.createElement('div');
2 newDiv.className = 'myclass';
3 newDiv.id = 'myid';
4
5 newDiv.innerHTML = 'Привет, мир!';
```



#### Клонирование

Новый элемент можно также клонировать из существующего:



```
newElem = elem.cloneNode(true)
```

Клонирует элемент `elem`, вместе с атрибутами, включая вложенные в него.

```
newElem = elem.cloneNode(false)
```

Клонирует элемент `elem`, вместе с атрибутами, но без подэлементов.

## Добавление элемента: `appendChild`, `insertBefore`

Чтобы DOM-узел был показан на странице, его необходимо вставить в документ.

Для этого у любого элемента есть метод `appendChild`:

**`parentElem.appendChild(elem)`**

Добавляет `elem` в список дочерних элементов `parentElem`. Новый узел добавляется в конец списка.

Следующий пример добавляет новый элемент в уже существующий `div`:

```
01 <div>
02   ...
03 </div>
04 <script>
05   var parentElem = document.body.children[0];
06
07   var newDiv = document.createElement('div');
08   newDiv.innerHTML = 'Привет, мир!';
09
10   parentElem.appendChild(newDiv);
11 </script>
```

**`parentElem.insertBefore(elem, nextSibling)`**

Вставляет `elem` в список дочерних `parentElem`, перед элементом `nextSibling`.

Сделать новый `div` первым дочерним можно так:

```
01 <div>
02   ...
03 </div>
04 <script>
05
06   var parentElem = document.body.children[0];
07
08   var newDiv = document.createElement('div');
09   newDiv.innerHTML = 'Привет, мир!';
10
11   parentElem.insertBefore(newDiv,
12     parentElem.firstChild);
12 </script>
```

Вместо `nextSibling` может быть `null`, тогда `insertBefore` работает как `appendChild`.

```
parentElem.insertBefore(elem, null);
// то же, что и:
parentElem.appendChild(elem)
```

Все методы вставки возвращают вставленный узел, например `parent.appendChild(elem)` возвращает `elem`.

## Удаление узлов: `removeChild`

Для удаления узла есть два метода:

`parentElem.removeChild(elem)`

Удаляет `elem` из списка детей `parentElem`.

`parentElem.replaceChild(elem, currentElem)`

Среди детей `parentElem` заменяет `currentElem` на `elem`.

Оба этих метода возвращают удаленный узел. Они просто вынимают его из списка, никто не мешает вставить его обратно в DOM в будущем.



Если вы хотите *переместить* элемент на новое место — не нужно его удалять со старого.

**Все методы вставки автоматически удаляют вставляемый элемент со старого места.**

Конечно же, это очень удобно.

Например, поменяем элементы местами:

```
1 <div>Первый</div>
2 <div>Второй</div>
3 <script>
4   var first = document.body.children[0];
5   var last = document.body.children[1];
6
7   // нет необходимости в предварительном
   removeChild(last)
8   document.body.insertBefore(last, first); //
   поменять местами
9 </script>
```

## Пример: показ сообщения

В качестве реального примера рассмотрим добавление сообщения на страницу. Чтобы показывалось посередине экрана и было красивее, чем обычный `alert`.

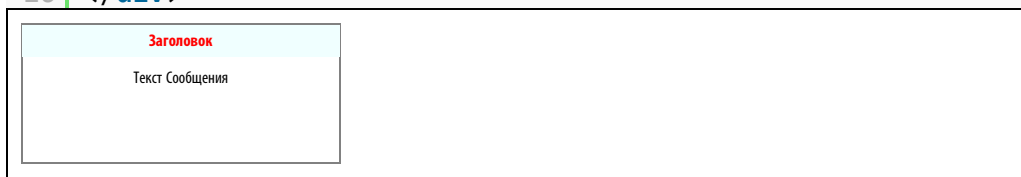
HTML-код для сообщения (без JS):

```
01 <style>
02 .message {
03   width: 300px;
04   height: 130px;
05   border: 1px solid gray;
06   text-align: center;
07 }
08 .message h1 {
09   color: red;
10   background: azure;
11   font-size: 16px;
12   height: 30px;
13   line-height: 30px;
14   margin: 0;
15 }
16 .message .content {
17   height: 50px;
18   padding: 10px;
```

```

19 }
20 </style>
21
22 <div class="message">
23   <h1>Заголовок</h1>
24   <div class="content">Текст Сообщения</div>
25   <input class="ok" type="button" value="OK"/>
26 </div>

```



Как видно - сообщение вложено в div фиксированного размера message и состоит из заголовка h1, тела content и кнопки ОК, которая нужна, чтобы сообщение закрыть.

Кроме того, добавлено немного стилей, чтобы как-то смотрелось.

## Создание сообщения

Для создания сложных структур DOM, как правило, используют либо готовый «шаблонный узел» и метод `cloneNode`, либо свойство `innerHTML`.

Следующая функция создает сообщение с указанным телом и заголовком.

```

01 function createMessage(title, body) {
02   // (1)
03   var container = document.createElement('div');
04
05   // (2)
06   container.innerHTML = '<div class="message"> \
07     <h1>' + title + '</h1> \
08     <div class="content">' + body + '</div> \
09     <input class="ok" type="button" value="OK"> \
10   </div>';
11
12   // (3)
13   return container.firstChild;
14 }

```

Как видно, она поступает довольно хитро. Чтобы создать элемент по текстовому шаблону, она сначала создает временный элемент `container` (1), а потом записывает (2) сообщение как `innerHTML` временного элемента. Теперь готовый элемент можно получить как `container.firstChild` и вернуть в (3).

## Добавление

Полученный элемент можно добавить в DOM:

```

var messageElem = createMessage('Привет, Мир!', 'Я - элемент DOM!')

document.body.appendChild(messageElem);

```

Окончательный результат:

```

01 <!DOCTYPE HTML>
02 <html>
03 <head>
04 <meta charset="utf-8">
05 <link type="text/css" rel="stylesheet" href="alert.css" />

```

```

06 </head>
07 <body>
08
09 Сообщение добавлено из JavaScript.
10
11 <script>
12
13 function createMessage(title, body) {
14     var container = document.createElement('div');
15
16     container.innerHTML = '<div class="message"> \
17         <h1>' + title + '</h1> \
18         <div class="content">' + body + '</div> \
19         <input class="ok" type="button" value="OK"> \
20     </div>';
21
22     return container.firstChild;
23 }
24
25 var messageElem = createMessage('Привет, Мир!', 'Я -
элемент DOM!')
26
27 document.body.appendChild(messageElem);
28
29 </script>
30
31 </body>
32 </html>

```

Конечно, нам хотелось бы пойти дальше. Расположить сообщение не где-нибудь, а посередине окна. Все это вполне возможно, а как — читайте в следующих занятиях.

## Текстовые узлы, тонкости использования

Как правило, при создании узлов и заполнении их используется `innerHTML`. Но текстовые узлы тоже имеют интересную область применения.

У них есть две особенности. Начнем с небольшого вопроса.



**Задача:** `createTextNode` vs `innerHTML`

Есть *пустой* узел DOM `elem`.

Важность: 5

**Одинаковый ли результат дадут эти скрипты?**

Первый:

```
elem.appendChild(document.createTextNode(text));
```

Второй:

```
elem.innerHTML = text;
```

Если нет — дайте пример значения `text`, для которого результат разный.

[Решение задачи "createTextNode vs innerHTML" »»](#)

Ответили на вопрос выше? Даже если нет, то, поглядев в решение, вы легко увидите разницу.

Итак, отличий два:

1. При создании текстового узла `createTextNode(' <b>...</b>')` любые

специальные символы и теги в строке будут интерпретированы как текст. А `innerHTML` вставит их как *HTML*.

- Во всех современных браузерах (кроме IE<8) создание и вставка текстового узла работает гораздо быстрее, чем присвоение `HTML`.

## Итого

Методы для создания узлов:

- `document.createElement(tag)` — создает элемент
- `document.createTextNode(value)` — создает текстовый узел
- `elem.cloneNode(deep)` — копирует элемент, если `deep == true`, то со всеми потомками.

Вставка и удаление узлов:

- `parent.appendChild(elem)`
- `parent.insertBefore(elem, nextSibling)`
- `parent.removeChild(elem)`
- `parent.replaceChild(elem, currentElem)`

Все эти методы возвращают `elem`.

Запомнить порядок аргументов очень просто: **новый(вставляемый) элемент — всегда первый.**

Методы для изменения DOM также описаны в спецификации [DOM Level 1](#).



### Задача: Удаление элементов

Напишите функцию, которая удаляет элемент из DOM.

Важность: 5

Синтаксис должен быть таким: `remove(elem)`, то есть, в отличие от `parentNode.removeChild(elem)` — без родительского элемента.

```
01 <div>Это</div>
02 <div>Все</div>
03 <div>Элементы DOM</div>
04
05 <script>
06   var elem = document.body.children[0];
07
08   function remove(elem) { /* ваш код */ }
09   remove(elem); // <-- функция должна удалить
    элемент
10 </script>
```

[Решение задачи "Удаление элементов" »»](#)



### Задача: insertAfter

Напишите функцию `insertAfter(elem, refElem)`, которая добавит `elem` после узла `refElem`.

Важность: 5

```
01 <div>Это</div>
02 <div>Элементы</div>
03
04 <script>
```

```

05     var elem = document.createElement('div');
06     elem.innerHTML = '<b>Новый элемент</b>';
07
08     function insertAfter(elem, refElem) { /* ваш код
09     */ }
10
11     var body = document.body;
12
13     // вставить elem после первого элемента
14     insertAfter(elem, body.firstChild); // <---
15     должно работать
16
17     // вставить elem за последним элементом
18     insertAfter(elem, body.lastChild); // <---
19     должно работать
20
21 </script>

```

[Решение задачи "insertAfter" »»](#)



#### Задача: removeChildren

Напишите функцию `removeChildren`, которая удаляет всех потомков элемента.

Важность: 5

```

01 <table>
02   <tr>
03     <td>Это</td><td>Все</td><td>Элементы DOM</td>
04   </tr>
05 </table>
06
07 <ol>
08   <li>Вася</li>
09   <li>Петя</li>
10   <li>Маша</li>
11   <li>Даша</li>
12 </ol>
13
14 <script>
15   function removeChildren(elem) { /* ваш код */ }
16
17   removeChildren(document.body.children[0]); //
18   очищает таблицу
19   removeChildren(document.body.children[1]); //
20   очищает список
21 </script>

```

P.S. Проверьте ваше решение в IE8.

[Решение задачи "removeChildren" »»](#)



#### Задача: Почему остаётся "aaa" ?

Запустите этот пример. Почему вызов `removeChild` не удалил текст "aaa"?

Важность: 1

```

01 <table>
02   aaa
03   <tr>
04     <td>Test</td>
05   </tr>
06 </table>
07
08 <script>
09   var table = document.body.children[0];
10
11   alert(table); // таблица, пока всё правильно
12

```

```
13 document.body.removeChild( table );
14 // почему в документе остался текст?
15 </script>
```

Решение задачи "Почему остаётся "aaa" ?" »»



#### Задача: Создать список

Напишите интерфейс для создания списка.

Важность: 4

Для каждого пункта:

1. Запрашивайте содержимое пункта у пользователя с помощью `prompt`.
2. Создавайте пункт и добавляйте его к `UL`.
3. Процесс прерывается, когда пользователь нажимает `ESC`.

**Все элементы должны создаваться динамически.**

Если посетитель вводит теги — в списке они показываются как обычный текст.

Работающий пример тут:

<http://learn.javascript.ru/files/tutorial/browser/dom/createList.html>

P.S. `prompt` возвращает `null`, если пользователь нажал `ESC`.

Решение задачи "Создать список" »»



#### Задача: Создайте дерево из объекта

Напишите функцию, которая создаёт вложенный список `UL/LI` (дерево) из объекта.

Важность: 5

Например:

```
01 var data = {
02   "Рыбы":{
03     "Форель":{},
04     "Щука":{}
05   },
06
07   "Деревья":{
08     "Хвойные":{
09       "Лиственница":{},
10       "Ель":{}
11     },
12     "Цветковые":{
13       "Берёза":{},
14       "Тополь":{}
15     }
16   }
17 };
```

Синтаксис:

```
1 var container =
  document.getElementById('container');
2 createTree(container, data); // создаёт
```

Результат (дерево):

- Рыбы
  - Форель
  - Щука
- Деревья
  - Хвойные
    - Лиственница
    - Ель

- Цветковые
  - Берёза
  - Тополь

Выберите один из двух способов решения этой задачи:

1. Создать строку, а затем присвоить через `container.innerHTML`.
2. Создавать узлы через методы DOM.

Если получится — сделайте оба.

Исходный документ:

<http://learn.javascript.ru/play/tutorial/browser/dom/build-tree-src.html>.

[Решение задачи "Создайте дерево из объекта" »»](#)



**Задача:** Создать календарь в виде таблицы

Напишите функцию, которая умеет генерировать календарь для заданной пары (месяц, год).

Важность: 4

Календарь должен быть таблицей, где каждый день — это TD. У таблицы должен быть заголовок с названиями дней недели, каждый день — TH.

Синтаксис: `createCalendar(id, year, month)`.

Такой вызов должен генерировать текст для календаря месяца `month` в году `year`, а затем помещать его внутрь элемента с указанным `id`.

Например: `createCalendar("cal", 2012, 9)` сгенерирует в `<div id='cal'></div>` следующий календарь:

пн	вт	ср	чт	пт	сб	вс
					1	2
3	4	5	6	7	8	9
10	11	12	13	14	15	16
17	18	19	20	21	22	23
24	25	26	27	28	29	30

Начальный документ со стилями:

[http://learn.javascript.ru/play/tutorial/date/calendar\\_src.html](http://learn.javascript.ru/play/tutorial/date/calendar_src.html)

P.S. Достаточно сгенерировать календарь, кликабельным его делать не нужно.

[Решение задачи "Создать календарь в виде таблицы" »»](#)



**Задача:** Часики с использованием "setInterval"

Создайте цветные часики как в примере ниже, используя `setInterval`:

Важность: 4

hh:mm:ss

Исходный документ:

<http://learn.javascript.ru/play/tutorial/advanced/timing/clock-interval-src/index.html>.

[Решение задачи "Часики с использованием "setInterval"" »»](#)



**Задача:** Часики при помощи "setTimeout"



вместо `setInterval`:

hh:mm:ss

Исходный документ:

<http://learn.javascript.ru/play/tutorial/advanced/timing/clock-timeout-src/index.html>.

Решение задачи "Часики при помощи `"setTimeout"`" »»

## Мультивставка: `insertAdjacentHTML` и `DocumentFragment`

Обычные методы вставки работают с одним узлом. Но есть и способы вставлять множество узлов одновременно.

### Оптимизация вставки в документ

Рассмотрим задачу: сгенерировать список UL/LI.

Есть две возможных последовательности:

1. Сначала вставить UL в документ, а потом добавить к нему LI:

```
var ul = document.createElement('ul');
document.body.appendChild(ul); // сначала в документ
for(...) ul.appendChild(li);   // потом узлы
```

2. Полностью создать список «вне DOM», а потом — вставить в документ:

```
var ul = document.createElement('ul');
for(...) ul.appendChild(li);   // сначала вставить узлы
document.body.appendChild(ul); // затем в документ
```

Как ни странно, между этими последовательностями есть разница. В большинстве браузеров, второй вариант — быстрее.

Почему же? Иногда говорят: «потому что браузер перерисовывает каждый раз при добавлении элемента». Это не так. **Дело вовсе не в перерисовке.**

Браузер достаточно «умён», чтобы ничего не перерисовывать понапрасну. В большинстве случаев процессы перерисовки и сопутствующие вычисления будут отложены до окончания работы скрипта, и на тот момент уже совершенно без разницы, в какой последовательности были изменены узлы.

**Тем не менее, при вставке узла происходят разные внутренние события и обновления внутренних структур данных, скрытые от наших глаз.**

Что именно происходит — зависит от конкретной, внутренней браузерной реализации DOM, но это отнимает время. Конечно, браузеры развиваются и стараются свести лишние действия к минимуму. На момент написания этой статьи, в Chrome разница минимальная.

Чтобы легко проверить текущее состояние дел — вот два бенчмарка.

Оба они создают таблицу 20x20, наполняя `TBODY` элементами `TR/TD`.

При этом первый вставляет все в документ тут же, второй — задерживает вставку TBODY в документ до конца процесса.

Кликните, чтобы запустить.

```
01  /* 1. Вставляет TBODY в документ сразу. а затем элементы
02  */
03  var appendFirst = new function() {
04      var benchTable;
05      this.setup = function() {
06          // ОЧИСТИТЬ ВСЁ
07          benchTable = document.getElementById('bench-table')
08          while(benchTable.firstChild) {
09              benchTable.removeChild(benchTable.firstChild);
10          }
11      }
12
13      this.work = function() {
14          // ВСТАВИТЬ TBODY и элементы
15          var tbody = document.createElement('TBODY');
16          benchTable.appendChild(tbody);
17
18          for(var i=0; i<20; i++) {
19              var tr = document.createElement('TR');
20              tbody.appendChild(tr);
21              for(var j=0; j<20; j++) {
22                  var td = document.createElement('td');
23                  td.appendChild(document.createTextNode(''+i.toString(20)+j.toString(20)));
24                  tr.appendChild(td);
25              }
26          }
27      }
28
29  }
30
31  /* 2. Полностью делает TBODY, а затем вставляет в документ
32  */
33  var appendLast = new function() {
34      var benchTable;
35      this.setup = function() {
36          // ОЧИСТИТЬ ВСЁ
37          benchTable = document.getElementById('bench-table');
38          while(benchTable.firstChild) {
39              benchTable.removeChild(benchTable.firstChild);
40          }
41      }
42
43      this.work = function() {
44          var tbody = document.createElement('TBODY');
45
46          for(var i=0; i<20; i++) {
47              var tr = document.createElement('TR');
48              tbody.appendChild(tr);
49              for(var j=0; j<20; j++) {
50                  var td = document.createElement('td');
51                  tr.appendChild(td);
52                  td.appendChild(document.createTextNode(''+i.toString(20)+j.toString(20)));
53              }
54          }
55
56          benchTable.appendChild(tbody);
57      }
58
59  }
```

# Добавление множества узлов

Продолжим работать со вставкой узлов.

Рассмотрим случай, когда в документе уже *есть* большой список UL. И тут понадобилось срочно добавить еще 20 элементов LI.

Как это сделать?

Если новые элементы пришли в виде строки, то можно попробовать добавить их так:

```
ul.innerHTML += "<li>1</li><li>2</li>...";
```

Но операция += с innerHTML не работает с DOM. Она не прибавляет, а заменяет всё содержимое списка на дополненную строку. Это не только медленно, но все внешние ресурсы (картинки) будут загружены заново! Так лучше не делать.

А если нужно вставить в середину списка? Здесь innerHTML вообще не поможет.

Можно, конечно, вставить строку во временный DOM-элемент и перенести оттуда элементы, но есть и гораздо лучший вариант: метод insertAdjacentHTML!

## insertAdjacentHTML

Метод **insertAdjacentHTML** позволяет вставлять произвольный HTML в любое место документа, в том числе *и между узлами*!

Он поддерживается всеми браузерами, кроме Firefox меньше версии 8, ну а там его можно эмулировать.

Синтаксис:

```
elem.insertAdjacentHTML(where, html);
```

**html**

Строка HTML, которую нужно вставить

**where**

Куда по отношению к elem вставлять строку. Всего четыре варианта:

1. beforeBegin — перед elem.
2. afterBegin — внутрь elem, в самое начало.
3. beforeEnd — внутрь elem, в конец.
4. afterEnd — после elem.

The diagram illustrates the four possible positions for inserting HTML relative to an element 'elem'. It shows a sequence of HTML fragments: <li>предыдущий</li>, <li>, elem, </li>, and <li>следующий</li>. Red arrows point to the gaps between these fragments: 'beforeBegin' points to the gap before the first <li>, 'afterBegin' points to the gap between the first <li> and elem, 'beforeEnd' points to the gap between elem and </li>, and 'afterEnd' points to the gap after </li>.

Например, вставим пропущенные элементы списка *перед* <li>5</li>:

```
01 <ul>
02   <li>1</li>
03   <li>2</li>
04   <li>5</li>
05 </ul>
06
07 <script>
08   var ul = document.body.children[0];
09   var li5 = ul.children[2];
```

```
10  
11 li5.insertAdjacentHTML("beforeBegin",  
    "<li>3</li><li>4</li>");  
12 </script>
```

Единственный недостаток этого метода — он не работает в Firefox до версии 8. Но его можно легко добавить, используя следующий JavaScript:

→ [insertAdjacentFF.js](#)

## insertAdjacentElement и insertAdjacentText

У этого метода есть «близнецы-братья», которые поддерживаются везде, кроме FF, но в него они добавляются этим же скриптом:

- `elem.insertAdjacentElement(where, newElem)` — вставляет в произвольное место не строку HTML, а элемент `newElem`.
- `elem.insertAdjacentText(where, text)` — создаёт текстовый узел из строки `text` и вставляет его в указанное место относительно `elem`.

Синтаксис этих методов, за исключением последнего параметра, полностью совпадает с `insertAdjacentHTML`. Они могут оказаться намного удобнее обычных `insertBefore/appendChild`.

## DocumentFragment



### Важно для старых браузеров

Оптимизация, о которой здесь идёт речь, важна в первую очередь для старых браузеров, включая IE9-. В современных браузерах эффект от нее не превышает 5-10%, а может быть и отрицательным.

До этого мы говорили о вставке строки в DOM. А что делать в случае, когда надо в существующий UL вставить много *DOM-элементов*?

Можно вставлять их один за другим, вызовом `insertBefore/appendChild`, но при этом получится много операций с большим живым документом. Избежать их поможет `DocumentFragment`. Это особенный *кросс-браузерный* DOM-объект, который похож на обычный DOM-узел, но им не является.

Синтаксис для его создания:

```
var fragment = document.createDocumentFragment();
```

В него можно добавлять другие узлы.

```
fragment.appendChild(node);
```

Его можно клонировать:

```
fragment.cloneNode(true); // клонирование с подэлементами
```

...Но у `DocumentFragment` нет обычных свойств DOM-узлов, таких как `innerHTML`, `tagName` и т.п. Это не узел.

«Фишка» заключается в том, что **когда `DocumentFragment` вставляется в DOM — то он исчезает, а вместо него вставляются его дети. Это**

свойство является уникальной особенностью DocumentFragment.

Например, можно добавить в него много LI, и потом appendChild к UL.

При этом фрагмент растворится, и в DOM вставятся именно LI, его прямые потомки, в том же порядке, в котором были во фрагменте.

```
01 // хотим вставить в список UL много LI
02
03 // делаем вспомогательный DocumentFragment
04 var fragment = document.createDocumentFragment();
05
06 for (цикл по li) {
07     fragment.appendChild(list[i]); // вставить каждый LI в
    DocumentFragment
08 }
09
10 ul.appendChild(fragment); // вместо фрагмента вставятся
    элементы списка
```

В современных браузерах, а также при вставке в узлы вне документа эффект от этой оптимизации меньше, вплоть до никакого.

Чтобы понять текущее положение вещей, попробуйте в различных браузерах следующий небольшой бенчмарк.

При нажатии на кнопки ниже в список добавляются 100 элементов.

```
01 var DocumentFragmentTest = new function() {
02     var benchList = document.getElementById('bench-list');
03
04     var items = [];
05     for(var i=0; i<100; i++) {
06         var li = document.createElement('li');
07         li.innerHTML = i;
08         items.push(li);
09     }
10
11     this.insertPlain = new function() {
12
13         this.setup = function() {
14             while(benchList.firstChild) {
15                 benchList.removeChild(benchList.firstChild);
16             }
17         }
18
19         this.work = function() {
20             for(var i=0; i<items.length; i++) {
21                 benchList.appendChild(items[i]);
22             }
23         }
24
25     };
26
27     this.insertDocumentFragment = new function() {
28
29         this.setup = function() {
30             // ОЧИСТИТЬ ВСЁ
31             while(benchList.firstChild) {
32                 benchList.removeChild(benchList.firstChild);
33             }
34         }
35
36         this.work = function() {
37             var docFrag = document.createDocumentFragment();
38             for(var i=0; i<items.length; i++) {
39                 docFrag.appendChild(items[i]);
40             }
41         }
42     };
43 }
```

```
41 | benchList.appendChild(docFrag);  
42 | }  
43 |  
44 | };  
45 | }
```

## Итого

- Манипуляции, меняющие структуру DOM (вставка, удаление элементов), как правило, быстрее с отдельным маленьким узлом, чем с большим DOM, который находится в документе.

Конкретная разница зависит от внутренней реализации DOM в браузере. На момент написания статьи, в Chrome она минимальна.

- Метод `elem.insertAdjacentHTML(where, html)` позволяет вставлять HTML-текст в произвольное место документа.

Совместимость:

- В FF7- его нет.
- IE6-7 поддерживают его лишь для тех элементов, где работает свойство `innerHTML` (т.е. кроме TABLE, TR и т.п.).
- `DocumentFragment` позволяет минимизировать количество вставок в большой живой DOM. Эта оптимизация особо эффективна в старых браузерах, в новых эффект от неё меньше.

Элементы сначала вставляются в него, а потом — он вставляется в DOM. При вставке `DocumentFragment` «растворяется», и вместо него вставляются содержащиеся в нём узлы.

`DocumentFragment` и `insertAdjacentHTML` дополняют друг друга. Один работает с узлами, другой — с текстом. Так что оба подхода могут быть полезны.



### Задача: Вставьте элементы в конец списка

Напишите код для вставки текста `html` в конец списка `ul` с использованием метода `insertAdjacentHTML`. Такая вставка, в отличие от присвоения `innerHTML+=`, не будет перезаписывать текущее содержимое.

Важность: 5

Добавьте к списку ниже элементы

```
<li>3</li><li>4</li><li>5</li>
```

```
1 | <ul>  
2 |   <li>1</li>  
3 |   <li>2</li>  
4 | </ul>
```

[Решение задачи "Вставьте элементы в конец списка" »»](#)



### Задача: Вставка `insertAdjacentHTML/DocumentFragment`

Напишите кроссбраузерную функцию `insertBefore(elem, html)`, которая:

Важность: 4

- Вставляет HTML-строку `html` перед элементом `elem`, используя `insertAdjacentHTML`,
- Если он не поддерживается (старый Firefox) — то через

DocumentFragment.

В обоих случаях должна быть лишь одна операция с DOM документа.

Следующий код должен вставить два пропущенных элемента списка `<li>3</li><li>4</li>`:

```
01 <ul>
02   <li>1</li>
03   <li>2</li>
04   <li>5</li>
05 </ul>
06
07 <script>
08 var ul = document.body.children[0];
09 var li5 = ul.children[2];
10
11 function insertBefore(elem, html) {
12   /* ваш код */
13 }
14
15 insertBefore(li5, "<li>3</li><li>4</li>")
16 </script>
```

[Решение задачи "Вставка insertAdjacentHTML/DocumentFragment"](#)

»»



### Задача: Отсортировать таблицу

Есть таблица:

Важность: 5

Имя	Фамилия	Отчество	Возраст
Вася	Петров	Александрович	10
Петя	Иванов	Петрович	15
Владимир	Ленин	Ильич	9
...	...	...	...

Строк в таблице много: может быть 20, 50, 100.. Есть и другие элементы в документе.

Как бы вы предложили отсортировать содержимое таблицы по полю Возраст? Обдумайте алгоритм, реализуйте его.

Как сделать, чтобы сортировка работала как можно быстрее? А если в таблице 10000 строк (бывает и такое)?

P.S. Может ли здесь помочь DocumentFragment?

P.P.S. Если предположить, что у нас заранее есть массив данных для таблицы в JavaScript — что быстрее: отсортировать эту таблицу или сгенерировать новую?

[Решение задачи "Отсортировать таблицу" »»](#)

См. также

- [John Resig: DOM insertAdjacentHTML](#)
- [MSDN: insertAdjacentHTML](#)
- [JSPerf: documentFragment VS Naive](#)

## Метод document.write

Метод `document.write` — один из наиболее древних методов добавления текста к документу.

У него есть существенные ограничения, поэтому он используется редко, но бывает полезен.

## Как работает `document.write`

Метод `document.write(str)` корректно работает только пока HTML еще не догружен.

Он дописывает текст в текущее место HTML.

HTML-документ ниже будет содержать 1 2 3.

```
1 <body>
2 1
3 <script>
4   document.write(2);
5 </script>
6 3
7 </body>
```

Нет никаких ограничений на содержимое `document.write`.

Строка просто пишется в HTML-документ без проверки структуры тегов, как будто она всегда там была.

Например:

```
01 <script>
02   document.write('<style> td { color: #F40 } </style>');
03 </script>
04 <table>
05   <tr>
06     <script> document.write('<td>') </script>
07     Текст внутри TD.
08     <script> document.write('</td>') </script>
09   </tr>
10 </table>
```

Также существует метод `document.writeln(str)` — не менее древний, который добавляет после `str` символ перевода строки `'\n'`.

## Только до конца загрузки

Во время загрузки браузер читает документ и тут же строит из него DOM, по мере получения информации достраивая новые и новые узлы, и тут же отображая их. Этот процесс идет непрерывным потоком. Вы наверняка видели это, когда заходили на сайты в качестве посетителя — браузер отображает неготовый документ, добавляя его новыми узлами по мере их получения.

**Методы `document.write` и `document.writeln` пишут напрямую в текст документа, до того как браузер построит из него DOM.**

Именно благодаря этому они могут записать в документ все, что угодно, любые стили и незакрытые теги. Браузер учтет их при построении DOM, точно так же, как учитывает очередную порцию HTML-текста.

Когда HTML загрузился, и браузер полностью построил DOM, документ становится «*закрытым*». Попытка дописать что-то в закрытый документ открывает его заново. При этом все текущее содержимое удаляется.

Текущая страница, скорее всего, уже загрузилась, поэтому если вы



нажмёте на эту кнопку — её содержимое удалится:

Из-за этой особенности `document.write` для загруженных документов не используют. Тем более, что есть всё многообразие методов DOM, изменяющих страницу.



### XHTML и `document.write`

В некоторых современных браузерах при получении страницы с заголовком `Content-Type: text/xml` или `Content-Type: text/xhtml+xml` включается «XML-режим» чтения документа. Метод `document.write` при этом не работает. Это одна из причин, по которой XML-режим обычно не используют.

## Преимущества перед `innerHTML`

В большинстве случаев, для модификаций DOM подходит `innerHTML`.

**Но `document.write` работает быстрее, фактически это самый быстрый способ добавить на страницу текст, сгенерированный скриптом.**

Это естественно, ведь он не модифицирует существующий DOM, а пишет в текст страницы до его генерации.

## Реклама

Например, его используют для вставки рекламных скриптов и различных счетчиков:

```
1 <script>
2   var url = 'http://ads.com/buy?rand='+Math.random();
3   document.write('<script src="'+url+'"></scr'+ 'ipt>');
4 </script>
```

- Здесь скрипт генерирует URL динамически, чтобы можно было добавлять к URL какую-либо информацию, доступную из JS. Например, добавляют разрешение экрана посетителя.
- Случайное значение добавляется для того, чтобы предотвратить кэширование.
- Закрывающий тег `</script>` в строке разделен, чтобы браузер не увидел `</script>` и не посчитал его концом скрипта.

Также используют запись:

```
document.write('<script src="'+url+'"><\script>');
```

Здесь `<\script>` вместо `</script>`, символ `'` при экранировании не меняется.

**Загрузка такого скрипта блокирует отрисовку всей страницы.**

Браузер не имеет права отобразить часть страницы после скрипта, до того как загрузит и выполнит скрипт. Если сайт рекламы по каким-то причинам работает медленно или вообще завис — зависнет и такая страница.

## Альтернатива: вставка через DOM

Есть более гибкий способ, который не блокирует страницу. Используйте DOM: создайте элемент SCRIPT и добавьте его в HEAD.

```
1 var script = document.createElement('script');
2 script.src = 'http://ads.com/buy?rand='+Math.random();
3
4 // теперь добавляем скрипт в HEAD, он будет загружен и
  выполнен
5 document.documentElement.children[0].appendChild(script);
```

**Добавление скриптов через DOM делает страницу независимой от возможных тормозов чужого сервера.**

Ситуация немного сложнее, если в скрипте может выполняться ещё один `document.write`, но это можно обойти. Например, заменить `document.write` своей функцией, которая пишет текст в строку, а строку затем добавляет в нужное место через методы DOM.



**Задача:** Загрузите скрипт с передачей разрешения экрана

Напишите код, который подключает внешний скрипт рекламы, передавая ему разрешение экрана, взятое из `window.screen` .

Важность: 3

К серверу должен идти запрос вида

"http://ads.com/load.js?x=1024&y=768&r=12345", где x, y — разрешение экрана, а r=12345 — некоторый случайный параметр, чтобы браузер не взял скрипт из кеша.

**Сделайте два варианта. Один с `document.write`, другой — с созданием элемента через DOM.**

Решение задачи "Загрузите скрипт с передачей разрешения экрана" »»

## Итого

Метод `document.write` (или `writeln`) пишет текст прямо в HTML, как будто он там всегда был.

→ Этот метод редко используется, так как работает только из скриптов, выполняемых в процессе загрузки страницы.

Запуск после загрузки приведёт к очистке документа.

→ Метод `document.write` очень быстр.

В отличие от установки `innerHTML` и DOM-методов, он не изменяет существующий документ, а работает на стадии текста, до того как DOM-структура сформирована.

→ Иногда `document.write` используют для добавления скриптов.

При этом браузер остановит отображение и будет ждать загрузки скрипта. Это оправдано лишь в том случае, если загружаемый скрипт сам вызывает `document.write`, в остальных случаях ведёт к излишним задержкам.

Поэтому желательно подключать внешние скрипты, используя вставку скрипта через DOM.

# Внешний вид: стили, прокрутка, координаты

---

## Стили и классы, getComputedStyle

---

Эта глава — о свойствах стиля, получении о них информации и изменении при помощи JavaScript.

Перед прочтением убедитесь, что хорошо знакомы с [блочной моделью CSS](#) и понимаете, что такое padding, margin, border.

### className

Свойство className соответствует HTML-атрибуту class.

Пример его использования (добавление класса):

```
1 <body class="class1 class2">
2 <script>
3   alert(document.body.className);
4   document.body.className += ' class3';
5 </script>
6 </body>
```

### classList

**Свойство classList предоставляет удобный интерфейс для работы с отдельными классами.**

Оно поддерживается во всех современных браузерах, в IE начиная с IE10, но его можно эмулировать в IE8+.

- `elem.classList.contains(cls)` — возвращает true/false, в зависимости от того, есть ли у элемента класс cls.
- `elem.classList.add/remove(cls)` — добавляет/удаляет класс cls
- `elem.classList.toggle(cls)` — если класса cls нет, добавляет его, если есть — удаляет.

Браузеры IE8,9 не поддерживают это свойство, зато дают возможность расширять встроенный тип HTMLElement, поэтому его поддержку можно эмулировать.

Код, реализующий это, вы можете найти в мини-библиотеке [classList.js](#).

Итого: classList можно использовать:

- Без эмуляции, если не нужны IE<10
- С эмуляцией, если не нужна поддержка IE<8

### Функции-заменители classList

---

Для поддержки старых IE нужна работа с className.

Так как классы перечислены через пробел, то, чтобы менять классы, понадобятся строковые операции с className. Их можно производить с

использованием регулярных выражений, а можно и через разбивку в массив.

Кросс-браузерные функции для работы с классами могут выглядеть так:

```
01 function addClass(el, cls) {
02     var c = el.className ? el.className.split(' ') : [];
03     for (var i=0; i<c.length; i++) {
04         if (c[i] == cls) return;
05     }
06     c.push(cls);
07     el.className = c.join(' ');
08 }
09
10 function removeClass(el, cls) {
11     var c = el.className.split(' ');
12     for (var i=0; i<c.length; i++) {
13         if (c[i] == cls) c.splice(i--, 1);
14     }
15
16     el.className = c.join(' ');
17 }
18
19 function hasClass(el, cls) {
20     for (var c = el.className.split(' '), i=c.length-1; i>=0; i--) {
21         if (c[i] == cls) return true;
22     }
23     return false;
24 }
```

Конечно же, все JavaScript-фреймворки имеют встроенные функции для таких задач.



#### Задача: Поставьте класс ссылкам

Сделайте желтыми внешние ссылки. Все

Важность: 3

относительные ссылки и абсолютные с доменом

javascript.ru считаются внутренними. Желтый цвет должен

обеспечиваться установкой класса external.

```
01 <style>
02 .external { background-color: yellow }
03 </style>
04 <ul>
05     <li><a
06         href="http://google.com">http://google.com</a></li>
07     <li><a href="/tutorial">/tutorial.html</a></li>
08     <li><a
09         href="ftp://ftp.com/my.zip">ftp://ftp.com/my.zip</a></li>
10     <li><a
11         href="http://nodejs.org">http://nodejs.org</a></li>
12     <li><a
13         href="http://javascript.ru/test">http://javascript.ru/test</a></li>
14     <li><a
15         href="ftp://javascript.ru/file">ftp://javascript.ru/file</a></li>
16 </ul>
```

Результат:

- <http://google.com>
- </tutorial.html>
- <ftp://example.com/my.zip>
- <http://nodejs.org>
- <http://javascript.ru>
- <ftp://javascript.ru/file>

Решение задачи "Поставьте класс ссылкам" »»

# style

Свойство `style` дает доступ к стилю элемента. Это свойство можно как читать, так и править.

С помощью него можно изменять большинство CSS-свойств, например `element.style.width = '100px'` работает так, как будто у элемента есть атрибут `style="width:100px"`. Также, как и в CSS, вам нужно указывать единицы измерения, например `px`.

Для свойств, названия которых состоят из нескольких слов, используется вотТакаяЗапись:

```
background-color => backgroundColor
z-index          => zIndex
border-left-width => borderLeftWidth
```

Например:

```
element.zIndex = 10000;
```



## style.cssFloat вместо style.float

Исключением является свойство `float`. В старом стандарте JavaScript слово `"float"` было зарезервировано и недоступно для использования в качестве свойства объекта. Поэтому используется `element.style.cssFloat`.

Впрочем, обычно `float` ставится непосредственно в CSS, а в JavaScript добавляется нужный класс.



## Свойства с префиксами

Специфические свойства браузеров, типа `-moz-border-radius`, `-webkit-border-radius`, записываются следующим способом:

```
button.style.MozBorderRadius = '5px';
button.style.WebkitBorderRadius = '5px';
```

То есть, каждый дефис дает большую букву. В этом смысле преобразование — такое же, как для обычных свойств.

Пример использования `style`:

```
1 | document.body.style.backgroundColor = prompt('background
   | color?', 'green');
```

**Чтобы сбросить поставленный стиль, присваивают в `style` пустую строку: `elem.style.width=""`.**

Например, для того, чтобы спрятать элемент, можно присвоить:

`elem.style.display = "none"`. Чтобы показать его обратно — не стоит явно указывать другой `elem.style.display = "block"`. Можно просто снять поставленный стиль: `elem.style.display = ""`.



## Когда стоит использовать style?

По возможности, для стилизации следует применять не JavaScript, а CSS. Это гораздо гибче и красивее. Такой код проще

расширять.

Свойство `style` можно использовать там, где CSS почему-то не подходит. Например, значение ширины `div.style.width` вычисляется в JavaScript в пикселях, и ему не соответствует какой-то определенный класс.

## `style.cssText`

**Свойство `style` является специальным объектом, ему нельзя присваивать строку.**

Запись `div.style="color:blue"` работать не будет. Но как же, всё-таки, поставить свойство стиль, если хочется задать его строкой?

Можно попробовать использовать атрибут:

`elem.setAttribute("style", ...)`, но самым правильным и кросс-браузерным решением такой задачи будет использование свойства `style.cssText`.

**Свойство `style.cssText` позволяет поставить стиль целиком в виде строки.**

Например:

```
01 <div>Button</div>
02
03 <script>
04   var div = document.body.children[0];
05
06   div.style.cssText="color: red !important; \
07     background-color: yellow; \
08     width: 100px; \
09     text-align: center; \
10     blabla: 5; \
11   ";
12
13   alert(div.style.cssText);
14 </script>
```

Браузер разбирает строку `style.cssText` и применяет известные ему свойства. Нет никаких ограничений на запись несуществующих свойств, но если указать свойство `blabla` — большинство браузеров его просто проигнорируют.

**При установке `style.cssText` все существующие свойства `style` перезаписываются.**

Поэтому, по возможности, во избежание конфликта, присваивают более конкретные подсвойства `style`: `style.color`, `style.width` и т.п, а `style.cssText` используют для более короткой записи, когда это заведомо безопасно.

## Получение информации о `style`

Зачастую перед нами встают задачи, для решения которых нужно узнать размер элемента, отступ и т.п.

**Свойство `style` позволяет читать эту информацию, но лишь ту, которая доступна напрямую из свойства/атрибута `"style"`:**

```
1 <body style="color:red">
2   Красный текст
```

```

3   </script>
4   alert(document.body.style.color); // red
5   </script>
6 </body>

```

Если же стиль получен из CSS, особенно когда задано составное свойство (margin вместо margin-top), то style «не знает» об этом.

Попытка получить marginTop в этом примере, будет неудачной:

```

1 <style>
2   body { margin: 10px }
3 </style>
4 <body>
5   <script>
6     alert(document.body.style.marginTop); // в большинстве
      браузеров ничего не выведет
7   </script>
8 </body>

```

...А вот если сначала присвоить свойство через style, а затем прочесть, то все работает отлично:

```

1 <body>
2   <script>
3     document.body.style.margin = '20px';
4     alert(document.body.style.marginTop); // 20px!
5   </script>
6 </body>

```

Обратите внимание на то, как браузер «распаковал» свойство style.margin, предоставив для чтения style.marginTop. То же самое произойдет и для border, background и т.д.

Свойство в стиле может быть показано не в точности, как его назначали, а как его воспринял браузер.

В примере ниже, color будет показан в виде rgb(...) в Firefox:

```

1 <body style="color:#abc">
2   <script>
3     alert(document.body.style.color); // rgb(170, 187,
      204)
4   </script>
5 </body>

```

В следующей задаче охватываются наиболее используемые свойства.



#### **Задача:** Скругленная кнопка со стилями из JavaScript

Возьмите документ, расположенный по адресу

Важность: 3

<http://learn.javascript.ru/play/tutorial/browser/dom/roundedButton/source.html>

и создайте ссылку <A> с заданным стилем, используя JavaScript.

В примере ниже кнопка создана при помощи HTML/CSS. В решении кнопка должна создаваться, настраиваться и добавляться в документ при помощи *только JavaScript*, без тегов <style> и <a>.

```

01 <!DOCTYPE HTML>
02 <html>
03 <head>
04   <meta charset="utf-8">
05   <style>
06     .button {

```

```

07     -moz-border-radius: 8px;
08     -webkit-border-radius: 8px;
09     border-radius: 8px;
10     border: 2px groove green;
11     display: block;
12     height: 30px;
13     line-height: 30px;
14     width: 100px;
15     text-decoration: none;
16     text-align: center;
17     color: red;
18     font-weight: bold;
19 }
20 </style>
21 </head>
22 <body>
23
24 <a class="button" href="/">Нажми меня</a>
25
26 </body>
27 </html>

```

Нажми меня

**Проверьте себя: вспомните, что означает каждое свойство.**

**В чём состоит эффект его появления здесь?**

[Решение задачи "Скругленная кнопка со стилями из JavaScript" »»](#)

Более полная информация о `style`, включающая другие, реже используемые методы работы с ним, доступна здесь:

[CSSStyleDeclaration](#) .

## getComputedStyle и currentStyle

Итак, свойство `style` дает доступ только к той информации, которая хранится в `elem.style`.

Он не скажет ничего об отступе, если он появился в результате наложения CSS или встроенных стилей браузера:

А если мы хотим, например, сделать анимацию и плавно увеличивать `marginTop` от текущего значения? Как нам сделать это? Ведь для начала нам надо это текущее значение получить.

**Для того, чтобы получить текущее используемое значение (used value) свойства, используется метод `window.getComputedStyle`, описанный в стандарте [DOM Level 2](#) .**

Его синтаксис таков:

```
getComputedStyle(element, pseudo)
```

**element**

Элемент, значения для которого нужно получить

**pseudo**

Указывается, если нужен стиль псевдо-элемента, например `":before"`. Пустая строка означает сам элемент.

Поддерживается всеми браузерами, кроме IE<9. Следующий код будет работать во всех не-IE браузерах и в IE9+:

```
01 <style>
```



```

02     body { margin: 10px }
03 </style>
04 <body>
05
06     <script>
07         var computedStyle = getComputedStyle(document.body,
08         '');
09         alert(computedStyle.marginTop); // выведет отступ в
        пикселях
10         alert(computedStyle.color); // выведет цвет
11     </script>
12 </body>

```



## Вычисленное (computed) и окончательное (resolved) значения

В CSS есть две концепции:

1. *Вычисленное* (computed) значение — это то, которое получено после применения всех правил CSS и CSS-наследования. Например, `width: auto` или `font-size: 125%`.
2. *Окончательное* (resolved) значение — непосредственно применяемое к элементу. При этом все размеры приводятся к пикселям, например `width: 212px` или `font-size: 16px`.

Когда-то `getComputedStyle` задумывалось для возврата вычисленного значения, но со временем оказалось, что окончательное гораздо удобнее. Поэтому сейчас в целом все значения возвращаются именно такие, кроме некоторых небольших глюков в браузерах, которые постепенно вычищаются.



## `getComputedStyle` требует полное свойство!

Для правильного получения значения нужно указать точное свойство. Например: `paddingLeft`, `marginTop`, `borderLeftWidth`.

**При обращении к сокращенному: `padding`, `margin`, `border` — правильный результат не гарантируется.**

Действительно, допустим свойства `paddingLeft/paddingTop` взяты из разных классов CSS. Браузер не обязан объединять их в одно свойство `padding`. Иногда, в простейших случаях, когда свойство задано сразу целиком, `getComputedStyle` сработает для сокращенного свойства, но не во всех браузерах.

Например, некоторые браузеры (Chrome) выведут 10px в документе ниже, а некоторые (Firefox) — нет:

```

1 <style>
2   body {
3     margin: 10px;
4   }
5 </style>
6 <script>
7   var style = getComputedStyle(document.body, '');
8   alert( style.margin ); // в Firefox пустая строка
9 </script>

```



## Стили посещенных ссылок — тайна!

У посещенных ссылок может быть другой цвет, фон, чем у обычных. Это можно поставить в CSS с помощью псевдокласса `:visited`.

Но `getComputedStyle` не дает доступ к этой информации, чтобы произвольная страница не могла определить, посещал ли пользователь ту или иную ссылку.

Кроме того, большинство браузеров запрещают применять к `:visited` CSS-стили, которые могут изменить геометрию элемента, чтобы даже окольным путем нельзя было это понять. В целях безопасности.

## currentStyle (IE)

В IE до 9 версии есть свое свойство `currentStyle`, что *почти* то же самое.

Фишка в том, что `currentStyle` возвращает вычисленное (computed) значение. В тех единицах измерения, в которых оно задано в CSS, а не в пикселях.

Давайте посмотрим:

```
01 <style>
02   body { margin: 10% }
03 </style>
04 <body>
05   <script>
06     var elem = document.body;
07
08     if (window.getComputedStyle) {
09       var computedStyle = getComputedStyle(elem, null);
10     } else {
11       computedStyle = elem.currentStyle;
12     }
13
14     var marginTop = computedStyle.marginTop;
15     alert(marginTop);
16   </script>
17 </body>
```

Можно написать кросс-браузерный код и короче:

```
var computedStyle = window.getComputedStyle ?
getComputedStyle(elem, '') : elem.currentStyle;
```

Некоторые особенности этого метода:

1. Размеры `getComputedStyle` выдает в пикселях.
2. В некоторых браузерах (Firefox, Chrome) пиксели могут быть дробными.
3. В IE свойство `currentStyle` сохраняет единицы измерения в размерах, этим оно отличается от `getComputedStyle`. Предыдущий пример выдаст проценты. Единицы измерения также сохраняются для ряда свойств в Safari (это `bar`).



IE<9: перевод pt, em, % в пиксели

**Этот материал — дополнительный. Он не обязателен для освоения.**

В IE для того, чтобы получить из процентов реальное значение в пикселях существует метод «runtimeStyle+pixel», [описанный Дином Эдвардсом](#).

Он основан на свойствах runtimeStyle и pixelLeft, работающих только в IE.

В следующем примере функция getIEComputedStyle(elem, prop) получает значение в пикселях для свойства prop, используя elem.currentStyle и метод Дина Эдвардса.

Если вам интересно, как он работает, ознакомьтесь со свойствами с [runtimeStyle](#) и [pixelLeft](#) в MSDN и раскройте код.

```
01 function getIEComputedStyle(elem, prop) {
02     var value = elem.currentStyle[prop] || 0
03
04     // we use 'left' property as a place holder so
    backup values
05     var leftCopy = elem.style.left
06     var runtimeLeftCopy = elem.runtimeStyle.left
07
08     // assign to runtimeStyle and get pixel value
09     elem.runtimeStyle.left = elem.currentStyle.left
10     elem.style.left = (prop === "fontSize") ? "1em" :
    value
11     value = elem.style.pixelLeft + "px";
12
13     // restore values for left
14     elem.style.left = leftCopy
15     elem.runtimeStyle.left = runtimeLeftCopy
16
17     return value
18 }
```

Рабочий пример (только IE):

```
01 <style> #margin-test { margin: 1%; border: 1px
    solid black; } </style>
02 <div id="margin-test">Тестовый элемент с margin
    1%</div>
03
04 <script>
05     var elem = document.getElementById('margin-
    test');
06     if (!elem.getComputedStyle) // старые IE
07         document.write(getIEComputedStyle(elem,
    'marginTop'));
08     else
09         document.write('Пример работает только в
    IE<9');
10 </script>
```

Тестовый элемент с margin 1%

*Пример работает только в IE<9*

Современные Javascript-фреймворки используют этот прием для эмуляции getComputedStyle в старых IE.

## Итого

Все DOM-элементы предоставляют следующие свойства.

→ Свойство className всегда синхронизируется с атрибутом class.

- Свойство `style` - это объект, в котором CSS-свойства пишутся вот так. Он хорошо работает для записи свойств. И позволяет читать свойства заданные им или атрибутом «`style`», но не описанные в CSS.
- `cssText` позволяет получать/записывать полный набор свойств для `style` в текстовом виде. С помощью `style` вы не можете добавить `!important`, но можете сделать это с `cssText`.
- Свойство `currentStyle`(IE) и метод `getComputedStyle` (стандарт) позволяют получить живое значение свойств стилей после того, как эти стили были применены.

При этом `currentStyle` возвращает значение из CSS, до окончательных вычислений, а `getComputedStyle` — непосредственно применяемое к элементу (как правило).



#### Задача: Создать уведомление

Напишите функцию `showNotification(options)`, Важность: 5  
которая показывает уведомление.

Описание функции:

```
01  /**
02   * Показывает уведомление, пропадающее через 1.5
    сек
03   *
04   * @param options.top {number} вертикальный отступ,
    в px
05   * @param options.right {number} правый отступ, в
    px
06   * @param options.cssText {string} строка стиля
07   * @param options.className {string} CSS-класс
08   * @param options.html {string} HTML-текст для
    показа
09   */
10  function showNotification(options) {
11      // ваш код
12  }
```

Демо: <http://learn.javascript.ru/files/tutorial/browser/dom/notification-style/index.html> (уведомления будут справа).

Исходный код для демо, кроме самой функции:

<http://learn.javascript.ru/files/tutorial/browser/dom/notification-style-src/index.html>.

[Решение задачи "Создать уведомление" >>>](#)

См. также

- [CSSStyleDeclaration](#)

## Размеры и прокрутка элементов

В этом разделе мы поговорим о размерах элементов DOM, способах их вычисления и *метриках* — различных свойствах, которые содержат эту информацию.

### Образец документа

Образец документа:

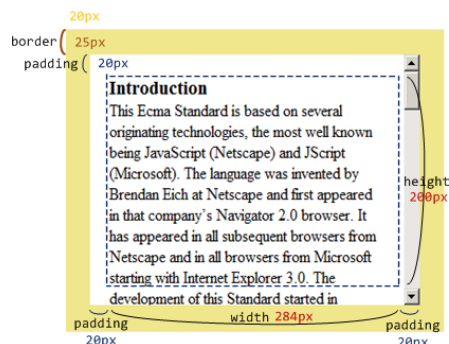
<http://learn.javascript.ru/files/tutorial/browser/dom/metric/metric.html>.

Перед тем, как продолжать чтение, ознакомьтесь с ним.

В примерах мы будем использовать блок с рамкой (border), полями (padding), отступами (margin) и прокруткой:

```
01 <div id="example">
02   ...Текст...
03 </div>
04 <style>
05 #example {
06   width: 300px;
07   height: 200px;
08
09   border: 25px solid #F0E68C; /* рамка 25px */
10
11   padding: 20px;             /* поля 20px */
12   margin: 20px;             /* отступы 20px */
13
14   overflow: auto;           /* прокрутка */
15 }
16 </style>
```

Результат выглядит так:



Вы можете открыть его [по этой ссылке](#).

## CSS-метрики width/height

CSS-высоту и ширину width/height можно установить с помощью elem.style и извлечь, используя getComputedStyle()/currentStyle:

```
style = window.getComputedStyle ? getComputedStyle(elem, '') :
elem.currentStyle;
alert(style.width); // вывести CSS-ширину
```

Они, при стандартном значении CSS-свойства box-sizing, относятся к размеру внутренней части элемента, которая лежит внутри padding. На рисунке выше нижнее поле padding заполнено текстом, но оно всё равно подразумевается и не входит в высоту height.

Подробнее о getComputedStyle/currentStyle обсуждаются в главе [Стили и классы, getComputedStyle](#).

Заметим, что эти метрики нельзя использовать для надежного получения ширины и высоты произвольного элемента!

Например:

```
01 <span>Привет!</span>
02 <div>Мир!</div>
03
04 <script>
05   var span = document.body.children[0];
06   var div = document.body.children[1];
07
08   alert( getComputedStyle(span, '').width ); // auto
```

```
09 | alert( getComputedStyle(div, "").width ); // ..px
10 | </script>
```

Документ выше выводит `width: auto` для `span`. С одной стороны, это правильно, ведь `span` имеет `display: inline`, его размер рассчитывается по содержимому. С другой стороны, это всё — тонкости CSS, а нам-то нужен конкретный размер в пикселях, который получен в результате всех этих вычислений. Увы, как раз его мы не получили.

## Полоса прокрутки

Полоса прокрутки — причина многих проблем и недопониманий. Поэтому мы будем разбирать её влияние на метрики самым внимательным образом.

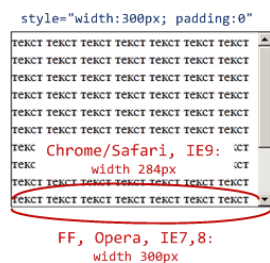
При наличии вертикальной полосы прокрутки — она забирает себе часть ширины элемента.

Ширина полосы прокрутки обычно составляет 16px, 18px, в зависимости от браузера и операционной системы. Бывает и 0 для полупрозрачной прокрутки, не отъедающей место (Chrome под MacOS). В примере подразумевается система Windows, поэтому внутренняя область будет уже не 300px, а 284px.

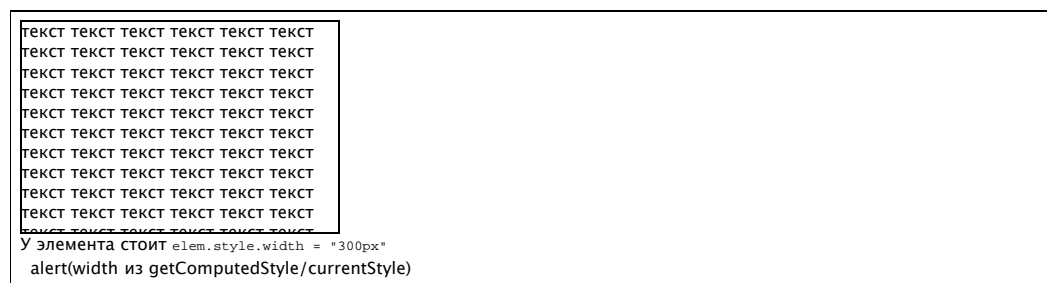
Несмотря на то, что на рисунке полоса прокрутки находится визуально в правом поле — отнимает место она не у `padding`, а у внутренней ширины.

...Но при этом некоторые браузеры отражают это уменьшение ширины в результате `getComputedStyle(...).width`, а некоторые — нет:

На рисунке ниже в CSS указано `width:300px`. А вот `getComputedStyle` возвращает `300px/284px`, в зависимости от браузера:



Вы можете увидеть значение `getComputedStyle(...).width`, нажав на кнопку в ифрейме:



Еще раз заметим, что различия касаются только чтения свойства `getComputedStyle(...).width` из JavaScript, CSS тут не при чем.

Отображение будет одинаковым. Ширина текста при наличии прокрутки уменьшается до 284px.

Здесь и далее, мы будем понимать под `width` именно реальную ширину внутренней области (284px), а не результат чтения CSS-свойства `width`, который может быть разным.

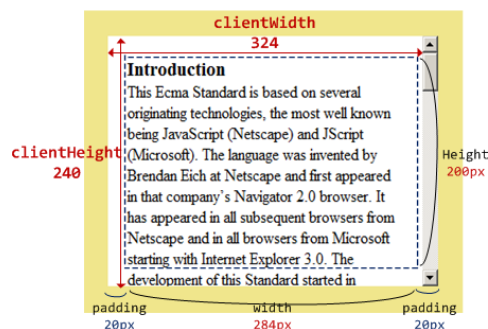
# JavaScript-метрики

В JavaScript существует ряд дополнительных свойств, содержащих размеры элементов. Мы будем называть их «метриками».

**Метрики JavaScript, в отличие от свойств CSS, содержат числа, всегда в пикселях и без единиц измерения на конце.**

## clientWidth/Height

Размер *клиентской зоны*, а именно: внутренняя область плюс padding.



Общая ширина внутри рамки — это

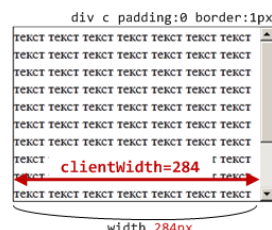
$284 \text{ (width)} + 20 \text{ (padding left)} + 20 \text{ (padding right)} = 324$ .

Получаем:

```
clientWidth = 284(width) + 2*20(padding) = 324
clientHeight = 200(height) + 2*20(padding) = 240
```

Обратите внимание, в clientHeight входят и верхнее и нижнее поля, несмотря на то, что нижнее поле заполнено текстом.

**Если padding нет, то clientWidth/Height покажет реальный размер области данных, внутри рамок и полосы прокрутки.**



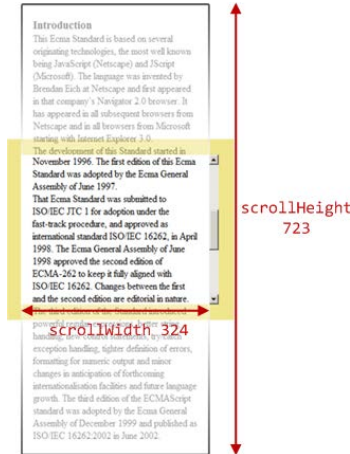
## scrollWidth/Height

Ширина и высота контента с учетом прокручиваемой области.

- `scrollHeight` = 723 — полная высота, включая прокрученную область
- `scrollWidth` = 324 — полная ширина, включая прокрученную область

`scrollWidth/Height` то же самое, что и `clientWidth/Height`, но включает в себя прокручиваемую область.





Эти свойства можно использовать, чтобы «распахнуть» элемент на всю ширину/высоту:

```
element.style.height = element.scrollHeight + 'px';
```

Нажмите на кнопку, чтобы распахнуть элемент:

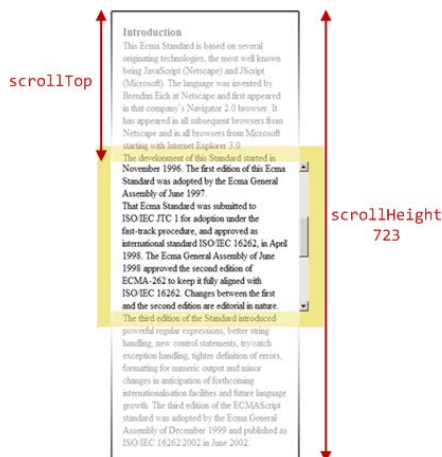
текст текст текст текст  
 текст текст текст текст  
 текст текст текст текст  
 текст текст текст текст  
 текст текст текст текст

```
element.style.height = element.scrollHeight + 'px'
```

## scrollTop/scrollLeft

Размеры текущей прокрученной части элемента — вертикальной и горизонтальной.

Следующее изображение иллюстрирует scrollTop и scrollLeft для блока с вертикальной прокруткой.



scrollTop/scrollLeft можно изменять

В отличие от большинства свойств, которые доступны только для чтения, значения scrollTop/scrollLeft можно изменить, и браузер выполнит прокрутку элемента.

При клике на следующий элемент будет выполняться код `elem.scrollTop += 10`. Поэтому он будет прокручиваться на 10px вниз:

Кликни  
 Меня





**Задача:** Найти размер прокрутки снизу

Свойство `elem.scrollTop` содержит размер прокрученной области при отсчете сверху. А как подсчитать его снизу?

Важность: 5

Напишите соответствующее выражение для произвольного элемента `elem`.

Проверьте: если прокрутки нет или элемент полностью прокручен — оно должно давать ноль.

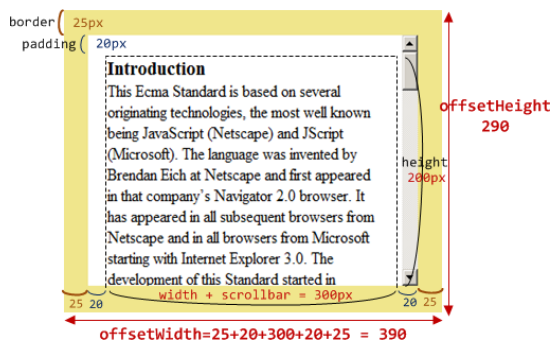
[Решение задачи "Найти размер прокрутки снизу" >>](#)

## offsetWidth/Height

**Внешняя ширина/высота блока, полный размер, включая рамки, исключая внешние отступы margin.**

→ `offsetWidth` = 390 — внешняя ширина блока

→ `offsetHeight` = 290 — внешняя высота блока



Эти свойства показывают *внешние* ширину и высоту блока, то как блок выглядит снаружи.

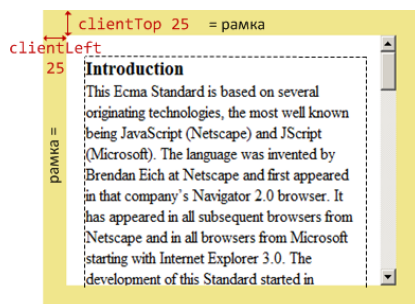
## clientTop/Left

**Отступ клиентской области от внешнего угла блока.**

Другими словами, это ширина верхней/левой рамки(border) в пикселях.

→ `clientLeft` = 25 — ширина левой рамки

→ `clientTop` = 25 — ширина верхней рамки



Казалось бы, зачем еще какие-то свойства, если ширину рамки можно получить напрямую из CSS? Обычно они действительно не нужны.

Но есть две ситуации, когда эти свойства бывают полезны:

1. В случае, когда документ располагается *справа налево* (арабский язык, иврит), свойство `clientLeft` включает в себя еще и ширину *правой*

полосы прокрутки.

- В IE<8 документ, а точнее — элемент `document.documentElement` немного смещен относительно верхнего левого угла документа. Несмотря на то, что рамки там нет, сдвиг существует и хранится в `document.body.clientLeft/clientTop` (обычно это 2 пикселя).

## offsetParent, offsetLeft/Top



### Используются редко...

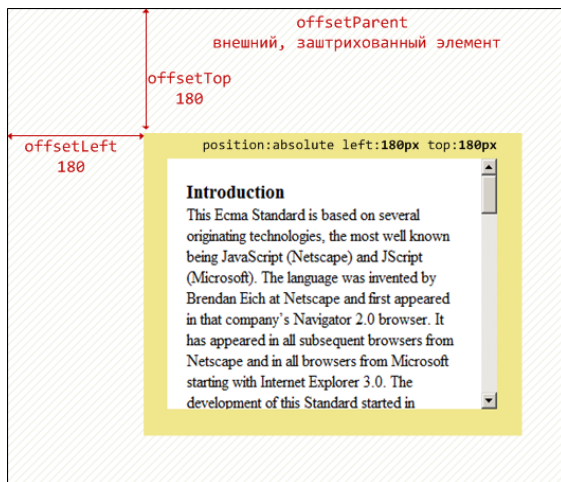
Ситуации, когда эти свойства нужны, можно перечислить по пальцам. Они возникают действительно редко. Как правило, эти свойства используются не по назначению. Возможно, лучше их даже и не знать...

**offsetParent** — это родительский элемент в смысле отображения на странице.

Например, если элемент позиционирован абсолютно, то таким родителем **offsetParent** является ближайший **позиционированный элемент** (т.е. свойство `position` не равно `static`), или `BODY`, если таковой отсутствует.

**Свойства offsetLeft/Top задают смещение относительно offsetParent.**

```
<div style="position: relative">
  <div style="position: absolute; left: 180px; top: 180px">...</div>
</div>
```



### Метрики для невидимых элементов равны нулю.

Координаты и размеры в JavaScript устанавливаются только для **видимых** элементов.

Они равны 0 для элементов с `display:none` или тех, которые вне DOM. Кстати, **offsetParent** для таких элементов тоже `null`.

Это дает нам **замечательный способ для проверки, виден ли элемент**:

```
function isHidden(elem)
  return !elem.offsetWidth && !elem.offsetHeight;
}
```

⇒ Работает, даже если родителю элемента установлено

свойство `display:none`.

- Работает для всех элементов, кроме TR, с которым возникают некоторые проблемы в разных браузерах. Обычно, проверяются не TR, поэтому всё ок 😊.
- Считает элемент видимым, даже если позиционирован за пределами экрана или имеет свойство `visibility:hidden`.
- «Схлопнутый» элемент, например пустой `div` без высоты и ширины, будет считаться невидимым.

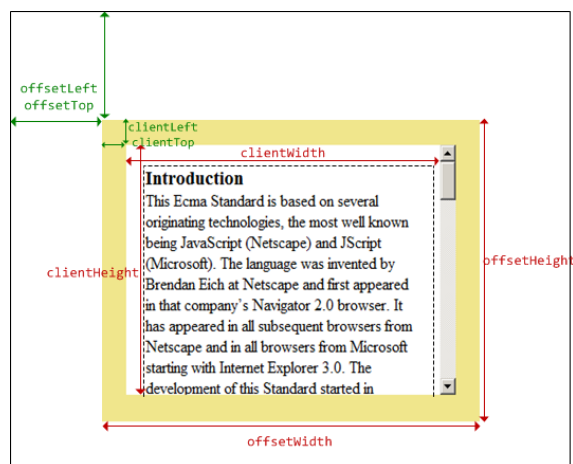
## Итого

У элементов есть следующие метрики:

- `clientWidth/clientHeight` — ширина/высота видимой области, включая поля, но не полосы прокрутки.
- `clientLeft/clientTop` — ширина левой/верхней рамки или, точнее, сдвиг клиентской области, относительно верхнего левого угла блока. Используется, преимущественно, в IE<8 для вычисления сдвига `document.body`.
- `scrollWidth/scrollHeight` — ширина/высота прокручиваемой области. Включат в себя `padding` и не включает полосы прокрутки.
- `scrollLeft/scrollTop` — ширина/высота прокрученной части документа, считается от верхнего левого угла.
- `offsetWidth/offsetHeight` — «внешняя» ширина/высота блока, не считая отступов.
- `offsetParent` — ближайшая ячейка таблицы, `body` для статического позиционирования или ближайший позиционированный элемент для других типов позиционирования.
- `offsetLeft/offsetTop` — позиция в пикселях левого верхнего угла блока, относительно его `offsetParent`.

Все свойства, кроме `scrollLeft/scrollTop` доступны только для чтения. Изменение этих свойств заставляет браузер прокручивать элемент.

Краткая схема:



**Прокрутку элемента можно прочитать или изменить через свойства `scrollLeft/Top`.**

В этой главе мы считали, что страница находится в режиме соответствия стандартам. В режиме совместимости — всё так же, но некоторые старые браузеры требуют `document.body` вместо `documentElement`.



### Задача: Узнать ширину полосы прокрутки

Важность: 3

Напишите код, который возвращает ширину стандартной полосы прокрутки. Именно самой полосы, где ползунок. Обычно она равна 16px, в редких и мобильных браузерах может колебаться от 14px до 18px, а кое-где даже равна 0px.

P.S. Ваш код должен работать на любом HTML-документе, независимо от его содержимого.

[Решение задачи "Узнать ширину полосы прокрутки" »»](#)



### Задача: Подменить div на другой с таким же размером

Важность: 3

Посмотрим следующий случай из жизни. Был текст, который, в частности, содержал div с зелеными границами:

Before Before Before

Text Text Text  
Text Text Text

After After After

Программист Валера из вашей команды написал код, который позиционирует его абсолютно и смещает в правый верхний угол. Вот этот код:

```
var div = document.getElementById('moving-div');  
div.style.position = 'absolute';  
div.style.right = div.style.top = 0;
```

Побочным результатом явилось смещение текста, который раньше шел после DIV. Теперь он поднялся вверх:

Before Before Before  
After After After

Text Text Text  
Text Text Text

**Допишите код Валеры, сделав так, чтобы текст оставался на своем месте после того, как DIV будет смещен.**

Сделайте это путем создания вспомогательного DIV с теми же размерами (width, height, border, margin, padding), что и у желтого DIV. Используйте только JavaScript, без CSS.

Должно быть так (новому блоку задан фоновый цвет для демонстрации):

Before Before Before

Text Text Text  
Text Text Text

After After After

Исходный документ:

<http://learn.javascript.ru/play/tutorial/browser/dom/replaceDiv/2.html>.

[Решение задачи "Подменить div на другой с таким же размером" »»](#)

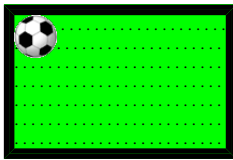


### Задача: Поместите мяч в центр поля

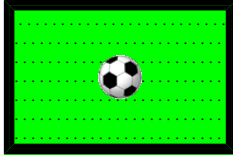
Важность: 5

Поместите мяч в центр поля.

Исходный документ выглядит так:



Используйте JavaScript, чтобы поместить мяч в центр:



- Менять CSS нельзя, мяч должен переносить в центр ваш скрипт, через установку нужных стилей элемента.
- Код не должен быть привязан к конкретному размеру мяча.
- Обратите внимание: мяч должен быть строго по центру! Независимо от местоположения поля и ширины его рамки.

Исходный документ:

<http://learn.javascript.ru/play/tutorial/browser/dom/ball-source/index.html>.

[Решение задачи "Поместите мяч в центр поля" »»](#)



**Задача:** Расширить элемент

В BODY есть элемент DIV с заданной шириной `width`. Важность: 3

Задача — написать код, который «распахнет» DIV по ширине на всю страницу.

Исходный документ (DIV — красный):



Код документа:

<http://learn.javascript.ru/play/tutorial/browser/dom/metric/bodywidth-src/index.html>.

Расширить нужно точно по ширине, чтобы красный DIV не вылез за границы BODY.

P.S. Пользоваться следует исключительно средствами JS, при этом не подглядывая в стили. То есть, код должен быть универсален и не ломаться, если цифры в CSS станут другими.

P.P.S. После того, как решите... Будет ли ваше решение работать, если у красного DIV стоит `position: absolute`? Если нет, то почему и как его поправить?

[Решение задачи "Расширить элемент" »»](#)



**Задача:** В чём отличие `"width"` и `"clientWidth"` ?

В чём отличия между `getComputedStyle(elem, "").width` и

Важность: 5

`elem.clientWidth?`

Приведите несколько, чем больше — тем лучше.

[Решение задачи "В чём отличие "width" и "clientWidth" ?" »»](#)

## Размеры и прокрутка для страницы

Многие метрики для страницы работают совсем не так, как для элементов. Поэтому рассмотрим решения типичных задач для страницы отдельно.

### Ширина/высота видимой части окна

Свойства `clientWidth/Height` для элемента `document.documentElement` позволяют получить ширину/высоту видимой области окна.

Например, кнопка ниже выведет размер такой области для этой страницы:

```
alert(document.documentElement.clientHeight)
```

Этот способ — кросс-браузерный.

### Ширина/высота всей страницы, с учётом прокрутки

Если прокрутка на странице присутствует, то полные размеры страницы можно взять в `document.documentElement.scrollHeight/scrollHeight`.

Проблемы с этими свойствами возникают, когда *прокрутка то есть, то нет*. В этом случае они работают некорректно.

В браузерах Chrome/Safari и Opera при отсутствии прокрутки значение `document.documentElement.scrollHeight` в этом случае может быть даже меньше, чем `document.documentElement.clientHeight` (нонсенс!). Эта проблема — именно для `document.documentElement`, то есть для всей страницы. С обычными элементами здесь всё в порядке.

Надёжно определить размер с учетом прокрутки можно, взяв максимум из двух свойств:

```
1 var scrollHeight = document.documentElement.scrollHeight;
2 var clientHeight = document.documentElement.clientHeight;
3
4 scrollHeight = Math.max(scrollHeight, clientHeight);
5
6 alert('Высота с учетом прокрутки: ' + scrollHeight);
```

## Прокрутка страницы

### Получение текущей прокрутки

Значение текущей прокрутки страницы хранится в свойствах `window.pageXOffset/pageYOffset`.

Но эти свойства:

→ Не поддерживаются IE<9

→

Их можно только читать, а менять нельзя.

Поэтому для кросс-браузерности рассмотрим другой способ — свойство `document.documentElement.scrollTop`.

- `document.documentElement` содержит значение прокрутки, если стоит правильный DOCTYPE. Это работает во всех браузерах, кроме Safari/Chrome.
- Safari/Chrome используют вместо этого `document.body` (это баг в Webkit).
- В режиме совместимости (если некорректный DOCTYPE) некоторые браузеры также используют `document.body`.

Таким образом, для IE8+ и других браузеров, работающих в режиме соответствия стандартам, получить значение прокрутки можно так:

```
1 var scrollTop = window.pageYOffset ||  
  document.documentElement.scrollTop;  
2  
3 alert("Текущая прокрутка: " + scrollTop);
```

## С учётом IE7- и Quirks Mode

Если дополнительно нужна поддержка IE<8, то там тоже есть важная тонкость. Документ может быть смещен относительно начальной позиции (0,0). Это смещение хранится в `document.documentElement.clientLeft/clientTop`, и мы должны вычесть его.

Если дополнительно добавить возможность работы браузера в Quicks Mode, то надёжный способ будет таким:

```
1 var html = document.documentElement;  
2 var body = document.body;  
3  
4 var scrollTop = html.scrollTop || body && body.scrollTop  
  || 0;  
5 scrollTop -= html.clientTop;  
6 alert("Текущая прокрутка: " + scrollTop);
```

Итого, можно создать кросс-браузерную функцию, которая возвращает значения прокрутки:

```
01 var getPageScroll = (window.pageXOffset != undefined) ?  
02   function() {  
03     return {  
04       left: pageXOffset,  
05       top: pageYOffset  
06     };  
07   } :  
08   function() {  
09     var html = document.documentElement;  
10     var body = document.body;  
11  
12     var top = html.scrollTop || body && body.scrollTop ||  
13     0;  
14     top -= html.clientTop;  
15  
16     var left = html.scrollLeft || body && body.scrollLeft  
    || 0;  
    left -= html.clientLeft;
```



```
17  
18     return { top: top, left: left };  
19 }
```

## Изменение прокрутки: `scrollTo`, `scrollBy`, `scrollIntoView`



Чтобы прокрутить страницу при помощи JavaScript, её DOM должен быть полностью загружен.

На обычных элементах свойства `scrollTop/scrollLeft` можно изменять, и при этом элемент будет прокручиваться.

Никто не мешает точно так же поступать и со страницей. Во всех браузерах, кроме Chrome/Safari можно осуществить прокрутку установкой `document.documentElement.scrollTop`, а в Chrome/Safari — использовать для этого `document.body.scrollTop`. И будет работать.

Но есть и другое, полностью кросс-браузерное решение — специальные методы прокрутки страницы `window.scrollBy(x,y)` и `window.scrollTo(pageX,pageY)`.

→ Метод `scrollBy(x,y)` прокручивает страницу относительно текущих координат.

Например, кнопка ниже прокрутит страницу на 10px вниз:

```
window.scrollBy(0,10)
```

→ Метод `scrollTo(pageX,pageY)` прокручивает страницу к указанным координатам относительно документа. Он эквивалентен установке свойств `scrollLeft/scrollTop`.

Чтобы прокрутить в начало документа, достаточно указать координаты (0,0):

```
window.scrollTo(0,0)
```

Для полноты картины рассмотрим также метод `elem.scrollIntoView(top)`.

Метод `elem.scrollIntoView(top)` вызывается на элементе и прокручивает страницу так, чтобы элемент оказался вверху, если параметр `top` равен `true`, и внизу, если `top` равен `false`. Причем, если параметр `top` не указан, то он считается равным `true`.

Кнопка ниже прокрутит страницу так, чтобы кнопка оказалась вверху:

```
this.scrollIntoView()
```

А следующая кнопка прокрутит страницу так, чтобы кнопка оказалась внизу:

```
this.scrollIntoView(false)
```

## Запрет прокрутки

Иногда бывает нужно временно сделать документ «непрокручиваемым». Например, при показе большого диалогового окна над документом — чтобы посетитель мог прокручивать это окно, но не документ.

Чтобы запретить прокрутку страницы, достаточно поставить



```
document.body.style.overflow = "hidden".
```

При этом страница замрёт в текущем положении. Попробуйте сами:

```
document.body.style.overflow = 'hidden'
```

```
document.body.style.overflow = ''
```

При нажатии на верхнюю кнопку страница замрёт на текущем положении прокрутки. После нажатия на нижнюю — прокрутка возобновится.

**Вместо `document.body` может быть любой элемент, прокрутку которого необходимо запретить.**

Недостатком этого способа является то, что сама полоса прокрутки исчезает. Если она занимала некоторую ширину, то теперь эта ширина освободится, и содержимое страницы расширится, заняв её место. Такая перерисовка иногда выглядит как «прыжок» страницы. Это может быть не очень красиво, но обходится, если вычислить размер прокрутки и добавить `padding-right`.

## Итого

### Размеры:

→ Для получения размеров видимой части окна:

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

→ Для получения размеров страницы с учётом прокрутки:

```
1 var scrollHeight =  
  document.documentElement.scrollHeight;  
2 var clientHeight =  
  document.documentElement.clientHeight;  
3  
4 scrollHeight = Math.max(scrollHeight, clientHeight);
```

### Прокрутка окна:

→ Прокрутку окна можно *получить* как `window.pageYOffset` (для горизонтальной — `window.pageXOffset`) везде, кроме IE<9.

Для кросс-браузерности используется другой способ:

```
1 var html = document.documentElement;  
2 var body = document.body;  
3  
4 var scrollTop = html.scrollTop || body && body.scrollTop  
  || 0;  
5 scrollTop -= html.clientTop; // IE<8  
6 alert("Текущая прокрутка: " + scrollTop);
```

→ Установить прокрутку можно при помощи специальных методов:

→ `window.scrollTo(pageX,pageY)` — абсолютные координаты,

→ `window.scrollBy(x,y)` — прокрутить относительно текущего места.

→ `elem.scrollIntoView(top)` — прокрутить, чтобы элемент `elem` стал виден.



**Задача:** Получить прокрутки документа

Напишите функцию `getDocumentScroll()`, которая возвращает объект с координатами области видимости относительно документа.

Важность: 5

Свойства объекта результата:

- `top` — координата верхней границы видимой части (относительно документа).
- `bottom` — координата нижней границы видимой части (относительно документа).
- `height` — полная высота документа, включая прокрутку.

В задаче можно учитывать только вертикальную прокрутку (горизонтальную отдельно нет смысла разбирать, она делается аналогично, а нужна сильно реже).

[Решение задачи "Получить прокрутки документа" » »](#)

## Координаты

В браузере есть три координатные системы.

### Относительно документа `pageX/pageY`

Точка начала координат лежит в левом верхнем углу страницы.

### Относительно окна `clientX/clientY`

Начало координат лежит в левом верхнем углу текущей видимой области.

### Относительно экрана `screenX/screenY`

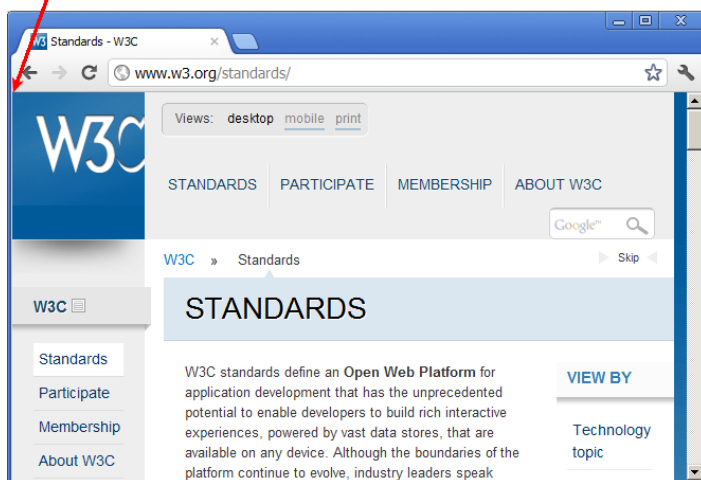
Начало координат — левый-верхний угол экрана.

*Координатами элемента, для краткости, называются координаты его левого верхнего угла.*

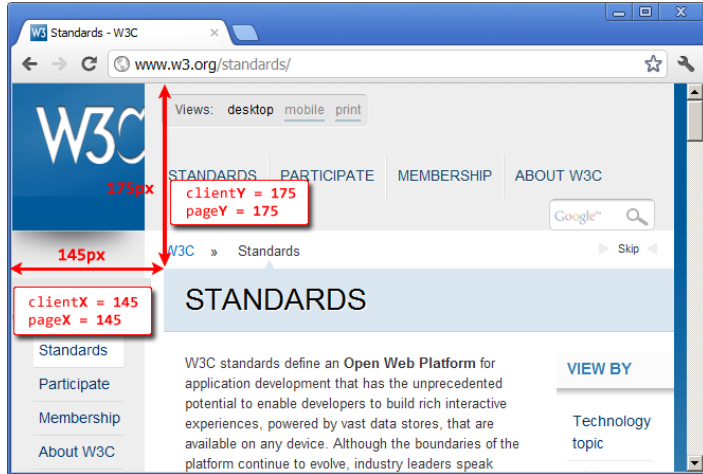
## Координаты относительно окна и документа

Когда страница не прокручена, точки начала координат относительно окна (`clientX,clientY`) и документа (`pageX,pageY`) совпадают:

```
(clientX,clientY) = (0,0)
(pageX, pageY)    = (0,0)
```



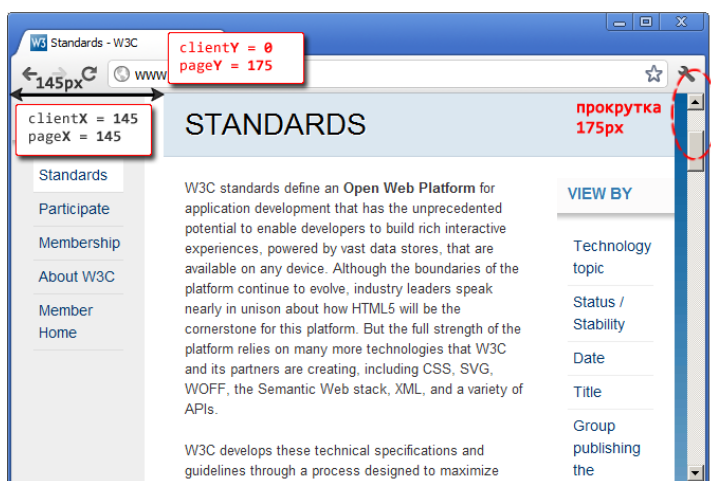
Например, координаты элемента с надписью «STANDARDS» равны расстоянию от верхней/левой границы окна:



**Прокрутим страницу, чтобы элемент был на самом верху:**

Посмотрите на рисунок ниже.

- Координата `clientY` изменилась. Она теперь равна 0, т.к. элемент на самом верху окна.
- Координата `pageY` отсчитывается от левого-верхнего угла документа, поэтому осталась такой же. Она не зависит от прокрутки.



**Итак, координаты `pageX`, `pageY` не меняются при прокрутке, в отличие от `clientX`, `clientY`.**

Разность этих двух координат `pageY - clientY` — в точности размер текущей прокрученной области.

## Получение координат элемента

### Координаты в окне: `elem.getBoundingClientRect()`

Этот метод описан в стандарте W3C и поддерживается всеми современными браузерами(и даже IE6+).

Он возвращает размеры прямоугольника, который охватывает элемент, в виде объекта со свойствами: `top`, `left`, `right`, `bottom`.

Эти 4 числа — координаты верхнего левого и нижнего правого углов элемента *относительно окна*. Например, кликните на кнопку, чтобы увидеть координаты ее углов:

```
1 <input id="brTest" type="button" value="Show
  button.getBoundingClientRect()" onclick='showRect(this)'/>
2
```

```

3 </script>
4 function showRect(elem) {
5     var r = elem.getBoundingClientRect()
6     alert("Top/Left: "+r.top+" / "+r.left)
7     alert("Right/Bottom: "+r.right+" / "+r.bottom)
8 }
9 </script>

```

Современные браузеры также возвращают ширину и высоту элемента в свойствах `width/height`.

**Координаты `elem.getBoundingClientRect()` — относительно окна, а не документа.**

Например, если вы прокрутите эту страницу так, чтобы верх кнопки совпал с верхней границей окна, то её `top`-координата станет равной нулю (или примерно нулю), потому что она считается относительно окна.

Чтобы посчитать ее относительно документа, нужно добавить к ней величину прокрутки.



### Как устроен `elem.getBoundingClientRect()`?

Браузер отображает любое содержимое, используя прямоугольники.

В случае с блочным элементом, таким как `DIV` - элемент сам по себе образует прямоугольник.

Но если элемент строчный и содержит в себе длинный текст, то каждая строка будет отдельным прямоугольником, с одинаковой высотой но разной длиной (у каждой строки — своя длина).

Более подробно это описано в: [спецификации](#) .

Если обобщить, содержимое элемента может отображаться в одном прямоугольнике или в нескольких.

Эти прямоугольники можно получить с помощью `elem.getClientRects()` . Но обычно они нам не нужны. А метод `elem.getBoundingClientRect()` возвращает один охватывающий прямоугольник для всех `getClientRects()`.

## Координаты в документе

Координаты относительно окна иногда бывают полезны. Например, для работы с `position: fixed`. А также чтобы получить текущее расстояние между двумя элементами — можно использовать их координаты относительно окна.

С другой стороны, координаты относительно документа более надёжны, так как они не меняются при прокрутке. Кроме того, именно они нужны для позиционирования на странице при помощи `position: absolute`. Поэтому они используются намного чаще.

К сожалению, готовой функции для получения таких координат нет. Но её можно легко написать самим.

Наша функция `getCoords(elem)` будет брать результат `elem.getBoundingClientRect()` и прибавлять текущую прокрутку документа.

Результат: объект с координатами {left: .., top: ..}

```
01 function getCoords(elem) {
02     // (1)
03     var box = elem.getBoundingClientRect();
04
05     var body = document.body;
06     var docEl = document.documentElement;
07
08     // (2)
09     var scrollTop = window.pageYOffset || docEl.scrollTop
    || body.scrollTop;
10     var scrollLeft = window.pageXOffset ||
    docEl.scrollLeft || body.scrollLeft;
11
12     // (3)
13     var clientTop = docEl.clientTop || body.clientTop || 0;
14     var clientLeft = docEl.clientLeft || body.clientLeft ||
    0;
15
16     // (4)
17     var top = box.top + scrollTop - clientTop;
18     var left = box.left + scrollLeft - clientLeft;
19
20     // (5)
21     return { top: Math.round(top), left: Math.round(left)
    };
22 }
```

Разберем что и зачем, по шагам:

1. Получаем прямоугольник.
2. Считаем прокрутку страницы. Все браузеры, кроме IE<9 поддерживают свойство `pageXOffset/pageYOffset`. В более старых IE, когда установлен DOCTYPE, прокрутку можно получить из `documentElement`, ну и наконец если DOCTYPE некорректен — использовать `body`.
3. В IE документ может быть смещен относительно левого верхнего угла. Получим это смещение.
4. Добавим прокрутку к координатам окна и вычтем смещение `html/body`, чтобы получить координаты всего документа.
5. Координаты округляются вызовом `Math.round()` т.к. в Firefox бывают дробные пиксели.

## Устаревший метод: `offset*`

Есть альтернативный способ нахождения координат — это пройти всю цепочку `offsetParent` от элемента вверх и сложить отступы `offsetLeft/offsetTop`.

Мы разбираем его здесь с учебной целью, т.к. он используется лишь в старых браузерах.

Вот функция, реализующая такой подход.

```
01 function getOffsetSum(elem) {
02     var top = 0, left = 0;
03
04     while(elem) {
05         top = top + parseInt(elem.offsetTop);
06         left = left + parseInt(elem.offsetLeft);
07         elem = elem.offsetParent;
08     }
09 }
```

```
10 | return {top: top, left: left};  
11 | }
```

Она работает, но у этого подхода есть как минимум два недостатка.

1. Он ненадежный. Разные браузеры преподносят «сюрпризы», включая или исключая размер рамок и прокруток из `offsetTop/Left`.
2. Он медленный, ведь нам каждый раз приходится проходить всю цепочку `offsetParent`.

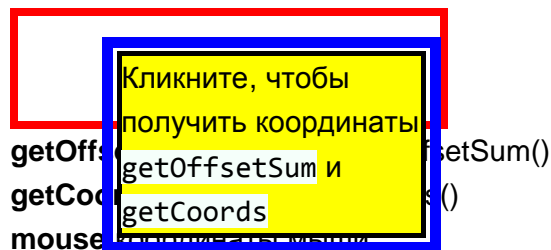
Можно, конечно, попытаться их обойти, но зачем? Ведь есть `getBoundingClientRect`.

## Сравнение `offset*` с `getBoundingClientRect`

Сравним методы для вычисления координат, описанные выше, на примере.

В прямоугольнике ниже есть 3 вложенных DIV. Все они имеют `border`, кое-кто из них имеет `position/margin/padding`.

Кликните по внутреннему (жёлтому) элементу, чтобы увидеть результаты обоих методов: `getOffsetSum` и `getCoords`, а также реальные координаты курсора — `event.pageX/pageY` (мы обсудим их позже в статье [Устранение IE-несовместимостей: «fixEvent»](#)).



При клике на любом месте желтого блока вы увидите результаты для `getOffsetSum(elem)` и `getCoords(elem)` ниже. Обратите внимание, что они обычно не совпадают!

Для того, чтобы узнать, какой же результат верный, кликните в левом-верхнем углу желтого блока. Он расположен в левом верхнем углу черной границы. Будут видны точные координаты мыши, так что вы можете сравнить их с `getOffsetSum/getCoords`.

Именно `getCoords` всегда возвращает верное значение 😊.

## Комбинированный подход

Фреймворки, которые хотят быть совместимыми со старыми браузерами, используют комбинированный подход:

```
1 | function getOffset(elem) {  
2 |   if (elem.getBoundingClientRect) {  
3 |     return getCoords(elem);  
4 |   } else { // старый браузер  
5 |     return getOffsetSum(elem);  
6 |   }  
7 | }  
  
01 | function getOffsetSum(elem) {  
02 |   var top=0, left=0  
03 |   while(elem) {  
04 |     top = top + parseInt(elem.offsetTop)  
05 |     left = left + parseInt(elem.offsetLeft)
```

```

06     elem = elem.offsetParent
07   }
08
09   return {top: top, left: left}
10 }
11
12
13 function getCoords(elem) {
14   var box = elem.getBoundingClientRect()
15
16   var body = document.body;
17   var docEl = document.documentElement;
18
19   var scrollTop = window.pageYOffset || docEl.scrollTop
  || body.scrollTop;
20   var scrollLeft = window.pageXOffset ||
docEl.scrollLeft || body.scrollLeft;
21
22   var clientTop = docEl.clientTop || body.clientTop || 0;
23   var clientLeft = docEl.clientLeft || body.clientLeft ||
0;
24
25   var top = box.top + scrollTop - clientTop;
26   var left = box.left + scrollLeft - clientLeft;
27
28   return { top: Math.round(top), left: Math.round(left)
  };
29 }
30
31
32 function getOffset(elem) {
33   if (elem.getBoundingClientRect) {
34     return getCoords(elem)
35   } else {
36     return getOffsetSum(elem)
37   }
38 }

```

## Получение элемента по координатам: elementFromPoint(x,y)

Обратная задача — «получить элемент по координатам в документе» может возникнуть при работе с событиями мыши. Например, при реализации Drag'n'Drop — когда один элемент проносится над другим, может понадобиться узнать, что именно сейчас находится под курсором мыши.

Для её решения есть метод `document.elementFromPoint(clientX, clientY)` .

Параметры `clientX/clientY` — координаты относительно окна.

Например: введите координаты в форме ниже. Соответствующий элемент на этой странице будет подсвечен стилем `background-color:red`. При повторном нажатии подсветка снимается.

clientX=

clientY=

Код функции подсветки:

```

1  /* x,y - координаты относительно окна */
2  function highlightRedXY(x,y) {
3    var el = document.elementFromPoint(x, y);
4    el.style.backgroundColor = (el.style.backgroundColor ==
'red') ? '' : 'red';
5  }

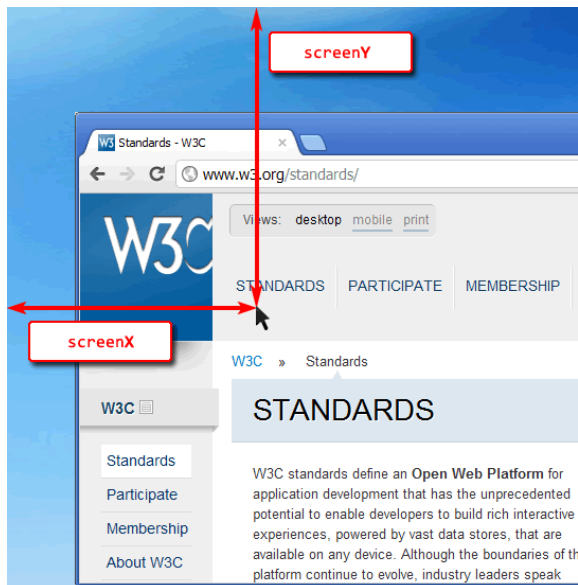
```



Обратите внимание — при прокрутке страницы на тех же координатах будут новые элементы. Это как раз потому, что координаты в этой функции — относительно окна, а не документа.

## Координаты на экране `screenX/screenY`

Координаты относительно экрана `screenX/screenY` отсчитываются от его левого-верхнего угла. Имеется в виду именно *весь экран*, а не окно браузера.



Такие координаты редко используются, так как обычно нас, всё же, интересует происходящее внутри браузера. Но они могут быть полезны, например, при работе с мобильными устройствами или для открытия нового окна посередине экрана вызовом `window.open` .

**Общая информация об экране хранится в глобальной переменной `screen` :**

```
1 // общая ширина/высота
2 alert( screen.width + ' x ' + screen.height );
3
4 // доступная ширина/высота (за вычетом taskбара и т.п.)
5 alert( screen.availWidth + ' x ' + screen.availHeight);
6
7 // есть и ряд других свойств screen (см. документацию)
```

**Координаты левого-верхнего угла браузера на экране хранятся в `window.screenX`, `window.screenY` (не поддерживаются IE<9):**

```
1 alert("Браузер находится на " + window.screenX + ", " +
  window.screenY);
```

Они могут быть и меньше нуля, если окно частично вне экрана.

**Координаты *DOM-элемента* на экране получить нельзя.** Браузер не предоставляет свойств и методов для этого.

## Итого

У любой точки в браузере есть координаты:

1. Относительно окна `window`.
2. Относительно документа `document` — не меняются при прокрутке.
3. Относительно экрана `screen`.



Начало координат — в левом-верхнем углу окна/документа/экрана.

### Координаты элемента относительно окна:

`elem.getBoundingClientRect()`, для получения относительно документа — добавляем прокрутку.

В старых Firefox, Chrome/Safari и Opera нескольких лет давности метода `getBoundingClientRect` нет, для них — можно использовать суммирование `offsetTop/Left`.

Координаты будут нужны нам далее, при работе с событиями мыши (координаты клика) и элементами (перемещение).



#### Задача: Найдите координаты

В ифрейме ниже вы видите документ с зеленым «полем».

Важность: 5

При помощи JavaScript найдите координаты указанных стрелками углов относительно окна браузера.

Для тестирования в документ добавлено удобство: клик в любом месте отображает координаты мыши относительно окна.

Ваш код должен при помощи DOM получить четыре пары координат:

1. Левый-верхний угол снаружи.
2. Левый-верхний угол внутри.
3. Правый-нижний угол внутри.
4. Правый-нижний угол снаружи.

Они должны совпадать с координатами, которые вы получите кликом по полю.

P.S. Код не должен быть как-то привязан к конкретным размерам элемента, стилям, наличию или отсутствию рамки.

**Когда сделаете — ответьте на вопрос: что нужно изменить, чтобы искать координаты относительно документа?**

Исходный документ:

<http://learn.javascript.ru/play/tutorial/browser/dom/field-coords-src/index.html>.

[Решение задачи "Найдите координаты" »»](#)



#### Задача: Разместить заметку рядом с элементом

Создайте функцию

Важность: 5

`positionAt(elem, anchor, position)`, которая позиционирует элемент `elem`, в зависимости от `position`, сверху ("top"), справа ("right") или снизу ("bottom") от элемента `anchor`.

Используйте её, чтобы вывести заметки Top, Right и Bottom рядом с цитатой, как здесь:

Исходный документ:

<http://learn.javascript.ru/play/tutorial/browser/dom/position-at-src/index.html>.

**Задача:** Разместить заметку внутри элемента

На основе решения задачи [Разместить заметку рядом с элементом](#) создайте функцию `positionAt(elem, anchor, position, inside)`, которая позиционирует элемент `elem`, в зависимости от `position`, сверху, справа или снизу от элемента `anchor`, а при указании аргумента `inside` — помещает его внутри `anchor`.

Важность: 5

Пример результата:

В качестве исходного документа возьмите решение задачи [Разместить заметку рядом с элементом](#).

[Решение задачи "Разместить заметку внутри элемента" »»](#)

## Проверка вложенности и соседства

Для проверки взаимного расположения узлов есть стандартный метод `compareDocumentPosition`.

Он не очень известен, но поддерживается везде, кроме IE8-, где способ проверки немного другой, но можно сделать кросс-браузерное решение.

Зачастую, использование этого метода гораздо эффективнее и проще, чем альтернативные методы проверки, в частности, чем «бегать по DOM» JS-кодом.

### Метод `compareDocumentPosition`

Синтаксис:

```
var result = nodeA.compareDocumentPosition(nodeB);
```

Возвращаемое значение — битовая маска (см. [Побитовые операторы](#)), биты в которой означают следующее:

**Биты**   **Число**   **Значение**

000000	0	nodeA и nodeB — один и тот же узел
000001	1	Узлы в разных документах (или один из них не в документе)
000010	2	nodeB предшествует nodeA (в порядке обхода документа)
000100	4	nodeA предшествует nodeB
001000	8	nodeB содержит nodeA
010000	16	nodeA содержит nodeB
100000	32	Зарезервировано для браузера

Понятие «предшествует» — имеется в виду не «предыдущий сосед при общем родителе», а «раньше встречается в порядке [прямого обхода](#) дерева документа.

Могут быть и сочетания битов. Примеры реальных значений:

```

02 <ul>
03   <li>1.1</li>
04 </ul>
05 <p>...</p>
06
07 <script>
08 var ul = document.getElementsByTagName('ul')[0];
09
10 // 1. соседи
11 alert( ul.compareDocumentPosition( ul.previousSibling ) );
12 // 2 = 10
13 alert( ul.compareDocumentPosition( ul.nextSibling ) ); //
14 // 4 = 100
15
16 // 2. родитель/потомок
17 alert( ul.compareDocumentPosition( ul.firstChild ) ); //
18 // 20 = 10100
19 alert( ul.compareDocumentPosition( ul.parentNode ) ); //
20 // 10 = 1010
21
22 // 3. вообще разные узлы
23 var li = ul.children[0];
24 alert( li.compareDocumentPosition( document.body.lastChild
25 ) ); // 4 = 100
26 </script>

```

Комментарии:

1. Узлы являются соседями, поэтому стоит только бит «предшествования»: какой перед каким.
2. Здесь стоят сразу два бита: 10100 означает, что ul одновременно содержит ul.firstChild и является его предшественником, то есть при прямом обходе дерева документа сначала встречается nodeA, а потом nodeB.  
Аналогично, 1010 означает, что ul.parentNode содержит ul и предшествует ему.
3. Так как ни один из узлов не является предком друг друга, то стоит только бит предшествования: li предшествует последнему узлу документа, никакого сюрприза здесь нет.



### Перевод в двоичную систему

Самый простой способ самостоятельно посмотреть, как число выглядит в 2-ной системе — вызвать для него toString(2), например:

```

1 | var x = 20;
2 | alert( x.toString(2) ); // "10100"

```

Или так:

```

1 | alert( 20..toString(2) );

```

Здесь после 20 две точки, так как если одна, то JS подумает, что после неё десятичная часть — будет ошибка.

Проверка условия «nodeA содержит nodeB» с использованием битовых операций: nodeA.compareDocumentPosition(nodeB) & 16, например:

```

01 <ul>
02   <li>1</li>
03 </ul>

```

```

04
05 <script>
06 var nodeA = document.body;
07 var nodeB = document.body.children[0].children[0];
08
09 if( nodeA.compareDocumentPosition(nodeB) & 16 ) {
10     alert( nodeA + ' содержит ' + nodeB );
11 }
12 </script>

```

Более подробно о битовых масках: [Побитовые операторы](#).

## Поддержка в IE8-

В IE<9 поддерживаются свои, нестандартные, метод и свойство:

### nodeA.contains(nodeB)

Результат: true, если nodeA содержит nodeB, а также в том случае, если nodeA == nodeB.

### node.sourceIndex

Номер элемента node в порядке прямого обхода дерева. Только для узлов-элементов.

Это позволяет написать кросс-браузерную реализацию compareDocumentPosition:

```

01 // Адаптировано с http://ejohn.org/blog/comparing-
    document-position/
02 function compareDocumentPosition(a, b) {
03     return a.compareDocumentPosition ?
04         a.compareDocumentPosition(b) :
05         (a != b && a.contains(b) && 16) +
06         (a != b && b.contains(a) && 8) +
07         (a.sourceIndex >= 0 && b.sourceIndex >= 0 ?
08             (a.sourceIndex < b.sourceIndex && 4) +
09             (a.sourceIndex > b.sourceIndex && 2) :
10             1);
11 }

```

Для узлов, которые не являются элементами, например, текстовых эта функция не покажет предшествование в IE<9, так как для таких узлов нет sourceIndex. А для узлов-элементов функция сработает кросс-браузерно.

## Итого

Для проверки, лежит ли один узел внутри другого или перед ним, вовсе не обязательно бегать по DOM при помощи JS. Есть встроенная функция, кросс-браузерный вариант которой приведён выше.

Пример её использования:

```

01 <ul>
02   <li>1</li>
03   <li>2</li>
04 </ul>
05
06 <script>
07 var body = document.body;
08 var li1 = document.body.children[0].children[0];
09 var li2 = document.body.children[0].children[1];
10
11 if( compareDocumentPosition(body, li1) & 16 ) {
12     alert( 'BODY содержит LI-1' );
13 }

```

```

14
15 if( compareDocumentPosition(li1, li2) & 4 ) {
16     alert( 'LI-1 предшествует LI-2' );
17 }
18
19 function compareDocumentPosition(a, b) {
20     return a.compareDocumentPosition ?
21     a.compareDocumentPosition(b) : (a !== b && a.contains(b) &&
16) + (a !== b && b.contains(a) && 8) + (a.sourceIndex >= 0
&& b.sourceIndex >= 0 ? (a.sourceIndex < b.sourceIndex &&
4) + (a.sourceIndex > b.sourceIndex && 2) : 1);
21 }
22 </script>

```

Список битовых масок для проверки:

**Биты Число Значение**

000000	0	nodeA и nodeB — один и тот же узел
000001	1	Узлы в разных документах (или один из них не в документе)
000010	2	nodeB предшествует nodeA (в порядке обхода документа)
000100	4	nodeA предшествует nodeB
001000	8	nodeB содержит nodeA
010000	16	nodeA содержит nodeB

## DOM-шпаргалка

В этой статье перечислены основные свойства и методы DOM, которые мы изучили.

Используйте её, чтобы быстро подглядеть то, что изучали ранее.

### Создание

**document.createElement(tag)**

создать элемент с тегом tag

**document.createTextNode(txt)**

создать текстовый узел с текстом txt

**node.cloneNode(deep)**

клонировать существующий узел, если deep=true то с подузлами.

### Свойства узлов

**node.nodeType**

тип узла: 1(элемент) / 3(текст) / другие.

**elem.tagName**

тег элемента.

**elem.innerHTML**

HTML внутри элемента.

**node.data**

содержимое любого узла любого типа, кроме элемента.

### Ссылки

**document.documentElement**

элемент <HTML>

**document.body**

элемент <BODY>

По всем узлам:

- parentNode
- nextSibling previousSibling
- childNodes firstChild lastChild

Только по элементам:

- Дети: children (В IE 8- также содержит комментарии)
- Соседи, кроме IE8-: nextElementSibling previousElementSibling
- Дети, кроме IE8-: firstElementChild lastElementChild

## Таблицы

---

**table.rows[N]**

строка TR номер N.

**tr.cells[N]**

ячейка TH/TD номер N.

**tr.sectionRowIndex**

номер строки в таблице в секции THEAD/TBODY.

**td.cellIndex**

номер ячейки в строке.

## Формы

---

**document.forms[N/name]**

форма по номеру/имени.

**form.elements[N/name]**

элемент формы по номеру/имени

**element.form**

форма для элемента.

## Поиск

**document.getElementById(id)**

По уникальному id

**document.getElementsByName(name)**

По атрибуту name, в IE<10 работает только для элементов, где name предусмотрен стандартом.

**(elem/doc).getElementsByTagName(tag)**

По тегу tag

**(elem/doc).getElementsByClassName(class)**

По классу, IE9+, корректно работает с элементами, у которых несколько классов.

**(elem/doc).querySelectorAll(css)**

По селектору CSS3, в IE8 по CSS 2.1

**(elem/doc).querySelector(css)**

По селектору, только первый элемент

## Изменение

- `parent.appendChild(newChild)`
- `parent.removeChild(child)`
- `parent.insertBefore(newChild, refNode)`
- `parent.insertAdjacentHTML("beforeBegin|afterBegin|beforeEnd|afterEnd", html)`

## Классы и стили

`elem.className`

Атрибут `class`

`elem.classList.add(class)` `remove(class)` `toggle(class)`

Управление классами в HTML5, для IE8+ есть [эмуляция](#).

`elem.style.display`

Стиль в атрибуте `style` элемента

`getComputedStyle(elem, '').display`

Стиль с учётом CSS и `style` на элементе

## Размеры и прокрутка элемента

`clientLeft/Top`

Ширина левой/верхней рамки `border`

`clientWidth/Height`

Ширина/высота внутренней части элемента, включая содержимое и `padding`, не включая полосу прокрутки (если есть).

`scrollWidth/Height`

Ширина/высота внутренней части элемента, с учетом прокрутки.

`scrollLeft/Top`

Ширина/высота прокрученной области.

`offsetWidth/Height`

Полный размер элемента: ширина/высота, включая `border`.

## Размеры и прокрутка страницы

- ширина/высота видимой области:  
`document.documentElement.clientHeight`
- прокрутка(чтение):  
`window.pageYOffset || document.documentElement.scrollTop`
- прокрутка(изменение):
  - `window.scrollBy(x,y)`: на `x,y` относительно текущей позиции.
  - `window.scrollTo(pageX, pageY)`: на координаты в документе.
  - `elem.scrollIntoView(true/false)`: прокрутить, чтобы `elem` стал видимым и оказался вверху окна(`true`) или внизу(`false`)

## Координаты

- относительно окна: `elem.getBoundingClientRect()`
- относительно документа: `elem.getBoundingClientRect()` + прокрутка страницы
- получить элемент по координатам:  
`document.elementFromPoint(clientX, clientY)`

## Решения задач



### Решение задачи: DOM children

Код:

```
document.documentElement.children[0]; // HEAD
document.body.children[1];           // UL
document.body.children[1].children[1]; // LI
```

Если комментарий переместить между элементами списка, то в IE<9 он станет одним из `children`, в результате последний код станет работать некорректно.

Чтобы это обойти, нужно либо не ставить комментарии в те места HTML, где планируются такие выборки, либо использовать другие методы поиска в HTML, которые мы рассмотрим [далее](#).



### Решение задачи: Проверка существования детей

Вначале нерабочие способы, которые могут прийти на ум:

```
if (!elem) { .. }
```

Это не работает, так как `elem` всегда есть, и является объектом. Так что проверка `if (elem)` всегда верна, вне зависимости от того, есть ли у `elem` потомки.

```
if (!elem.childNodes) { ... }
```

Тоже не работает, так как псевдо-массив `childNodes` всегда существует. Он может быть пуст или непуст, но он всегда является объектом, так что проверка `if (elem.childNodes)` всегда верна.

Несколько рабочих способов:

```
1 if (!elem.childNodes.length) { ... }
2
3 if (!elem.firstChild) { ... }
4
5 if (!elem.lastChild) { ... }
```

Последний - самый короткий.



### Решение задачи: Вопрос по навигационным ссылкам

1. Да, верно, с оговоркой. Элемент `elem.lastChild` последний, у него нет правого соседа.

**Оговорка:** `elem.lastChild.nextSibling` выдаст ошибку если `elem` не имеет детей.

2. Нет, неверно, это может быть текстовый узел. Значением `elem.children[0]` является первый узел-элемент, перед ним может быть текст.



Аналогично предыдущему случаю, если у `elem` нет детей-элементов — будет ошибка.



### Решение задачи: В инлайн скрипте `lastChild.nodeType`

Небольшой подвох — в том, что во время выполнения скрипта последним тегом является `SCRIPT`. Браузер не может обработать страницу дальше, пока не выполнит скрипт.

Так что результат будет 1 (узел-элемент).

```
1 <!DOCTYPE HTML>
2 <html>
3 <body>
4   <script>
5     alert(document.body.lastChild.nodeType);
6   </script>
7 </body>
8 </html>
```



### Решение задачи: Найти следующий элемент

#### Менее эффективный вариант

Цикл по правым соседям в поисках узла-элемента:

```
01 <div>Первый</div>
02 <!-- комментарий... -->
03 <p>Второй</p>
04
05 <script>
06 function getNextElement(elem) {
07   var current = elem.nextSibling;
08
09   while(current && current.nodeType != 1) {
10     current = current.nextSibling;
11   }
12
13   // два варианта окончания цикла:
14   // current == null (нет следующего узла-элемента)
15   // current.nodeType == 1 (нашли)
16   return current;
17 }
18
19 alert(getNextElement(document.body.children[0]).tagName);
20 // "P"
21 alert(getNextElement(document.body.lastChild)); //
22 null
23 </script>
```

#### Более эффективный вариант

Все браузеры, кроме IE<9, поддерживают свойство `nextElementSibling`. Воспользуемся этим.

```
01 <div>Первый</div>
02 <!-- комментарий... -->
03 <p>Второй</p>
04
05 <script>
06 function getNextElement(elem) {
07   if (elem.nextElementSibling !== undefined) {
08     return elem.nextElementSibling;
09   }
10   var current = elem.nextSibling;
```

```

11
12 while(current && current.nodeType != 1) {
13     current = current.nextSibling;
14 }
15
16 // два варианта окончания цикла:
17 // current == null (нет следующего узла-элемента)
18 // current.nodeType == 1 (нашли)
19 return current;
20 }
21
22 alert(getNextElement(document.body.children[0]).tagName);
  // "P"
23 alert(getNextElement(document.body.lastChild)); //
  null
24 </script>

```

В выделенном фрагменте мы проверяем поддержку этого свойства. Если оно поддерживается, то равно либо элементу-соседу, либо `null`, если такого соседа нет. В любом случае это не `undefined`.

Если же оно не поддерживается, то производим те же вычисления, что в предыдущем решении.

### Ещё более эффективный вариант

Поддержка свойства `nextElementSibling` в браузере либо есть, либо её нет. Зачем проверять её для каждого элемента? Можно сделать это один раз и запомнить результат. А можно поступить ещё лучше — определить функцию по-разному, в зависимости от того, поддерживается это свойство или нет.

```

01 <div>Первый</div>
02 <!-- комментарий... -->
03 <p>Второй</p>
04
05 <script>
06 var getNextElement =
07   document.documentElement.nextElementSibling !==
  undefined ?
08   function(elem) {
09       return elem.nextElementSibling;
10   }
11   :
12   function(elem) {
13       var current = elem.nextSibling;
14       while(current && current.nodeType != 1) {
15           current = current.nextSibling;
16       }
17       return current;
18   };
19
20 alert(getNextElement(document.body.children[0]).tagName);
  // "P"
21 alert(getNextElement(document.body.lastChild)); //
  null
22 </script>

```



### Решение задачи: Тег в комментарии

Ответ: BODY.

```

1 <script>
2 var body = document.body;
3

```

```

4 | body.innerHTML = "<!--" + body.tagName + "-->";
5 |
6 | alert(body.firstChild.data); // BODY
7 | </script>

```

Происходящее по шагам:

1. Заменяем содержимое `<body>` на комментарий. Он будет иметь вид `<!--BODY-->`, так как `body.tagName == "BODY"`. Как мы помним, свойство `tagName` в HTML всегда находится в верхнем регистре.
2. Этот комментарий теперь является первым и единственным потомком `body.firstChild`.
3. Получим значение `data` для комментария `body.firstChild`. Оно равно содержимому узла для всех узлов, кроме элементов. Содержимое комментария: `"BODY"`.



#### Решение задачи: Получите пользовательский атрибут

1. Для начала, нам нужно получить `div`. Здесь нет комментариев, так что сработает `document.body.children[0]` даже для старых IE.

```
var div = document.body.children[0];
```

2. Пользовательский атрибут не попадает в свойства DOM-объекта, поэтому используем метод `getAttribute()`.

```

01 | <body>
02 |
03 |   <div data-widget-name="menu">Выберите
    |   жанр</div>
04 |
05 |   <script>
06 |     var div = document.body.children[0];
07 |
08 |     var widgetName = div.getAttribute('data-
    |     widget-name');
09 |
10 |     alert( widgetName ); // "menu"
11 |   </script>
12 | </body>

```



#### Решение задачи: Длина коллекции после удаления элементов

1. Ответ на первый вопрос - 0, пустая коллекция.

```

01 | <ul id="menu">
02 |   <li>Главная страница</li>
03 |   <li>Форум</li>
04 |   <li>Магазин</li>
05 | </ul>
06 | <script>
07 |   var lis =
    |   document.getElementsByTagName('li');
08 |
09 |   document.body.innerHTML = "";
10 |
11 |   alert(lis.length);
12 | </script>

```

Это потому, что все элементы из BODY удаляются, а коллекция - живая.

2. Ответ на второй вопрос зависит от браузера. В большинстве браузеров будет 3, коллекция не изменилась, так как она теперь привязана не к BODY, а к элементу, на котором идёт поиск, т.е. к menu.

Но элемент menu находится в переменной, и поэтому должен быть жив, а значит и его дети тоже. Но некоторые браузеры (IE10) используют агрессивный подход при работе с памятью и очищают все элементы, кроме тех, которые непосредственно хранятся в переменных.

Поэтому результат кода ниже в большинстве браузеров: 3, а в IE10: 0.

```
01 <ul id="menu">
02   <li>Главная страница</li>
03   <li>Форум</li>
04   <li>Магазин</li>
05 </ul>
06 <script>
07
08   var menu = document.getElementById('menu');
09   var lis = menu.getElementsByTagName('li');
10
11   document.body.innerHTML = "";
12
13   alert(lis.length);
14 </script>
```



### Решение задачи: Поиск элементов

Решение:

1.

```
var table = document.getElementById('age-table');
var result = table.getElementsByTagName('label');
```

2.

```
1 var table = document.getElementById('age-table');
2 var result = table.getElementsByTagName('td')[0];
3 // в современных браузерах можно одним запросом:
4 var result = document.querySelector('#age-table td');
```

3.

```
var form = document.getElementsByTagName('form')[1];
```

4.

```
var searchForm = document.forms.search;
// в современных браузерах также:
var searchForm =
document.querySelector('form[name="search"]');
```

5.

```
var searchForm = document.forms.search;
```

```
var result =  
searchForm.getElementsByTagName('input')[0];
```

6.

```
var result =  
document.getElementsByName('info[0]')[0];
```

7.

```
var result = document.forms['search-  
person'].elements['info[0]'];  
// HE elements.info[0], так как имени info вообще нет
```



### Решение задачи: Проверка, находится ли элемент внутри другого

Для начала, нам нужно получить таблицу по id:

```
var table = document.getElementById('age-table');
```

Затем, мы должны проверить, является ли table предком elem. Это можно сделать, пройдя по цепочке родительских элементов: elem.parentNode, elem.parentNode.parentNode..., пока не дойдем либо до таблицы либо до null.

Код рабочей функции:

```
01 function checkInsideTable(elem) {  
02   var table = document.getElementById('age-table');  
03  
04   while (elem != table && elem) { // вверх по  
    родителям  
05     elem = elem.parentNode;  
06   }  
07  
08   // сейчас верно одно из двух:  
09   // 1) либо elem == table (внутри!)  
10   // 2) либо elem == null  
11  
12   return !!elem; // приведение логическому типу  
    даст: 1) true 2) false  
13 }
```

Альтернативные варианты:

1. Перебрать все элементы внутри таблицы и сравнить с данным:

```
1 var elems = table.getElementsByTagName('*');  
2 for(var i=0; i<elems.length; i++) {  
3   if (elems[i] == elem) return true;  
4 }  
5 return false;
```

2. Использовать запрос table.querySelector(elemId), где elemId — это id элемента elem. В случае, если у элемента нет id — временно сгенерировать и присвоить случайный:

```
01 var tmpId = false;  
02 if (!elem.id) {  
03   tmpId = Math.random(); // сгенерировать  
    временный id  
04   elem.id = tmpId;  
05 }
```

```

06
07 var result = table.querySelector('#'+elem.id);
08 if (tmpId) delete elem.id; // удалить временный
   id, если был присвоен
09
10 return !!result;

```

Это более быстро, но забавно то, что внутренняя реализация `querySelector` в большинстве браузеров как раз и заключается в том, чтобы перебрать все элементы.

3. И еще один, самый короткий, вариант — через рекурсию:

```

1 var table = document.getElementById('age-table');
2
3 function checkInsideTable(elem) {
4   if (!elem) return false;
5   return elem.id == 'age-table' ||
      checkInsideTable(elem.parentNode);
6 }

```

По сути — это то же самое, что первый вариант.



### Решение задачи: Дерево

#### Алгоритм решения

Сперва, опишем алгоритм решения.

Он может быть таким:

1. Найти все элементы списка.
2. Для каждого элемента списка:
  1. Подсчитать количество потомков LI.
  2. Если количество равно 0, пропускаем этот элемент, иначе, изменяем DOM, добавляя эту информацию.

Реализуйте его.

#### Полное решение

В алгоритме не хватает всего нескольких деталей.

- Количество потомков — его можно получить как `UL.getElementsByTagName('li').length`.
- Добавление текста с количеством потомков к `li`. Это можно сделать, дописав прямо в текстовый узел `li.firstChild`, с помощью свойства `data`.

Окончательное решение:

<http://learn.javascript.ru/play/tutorial/browser/dom/tree.html>.



### Решение задачи: Сравнение количества элементов

Значение `aList1` изменится, потому что `getElementsByTagName` - живая коллекция. Она автоматически дополнится новым элементом `a` и ее длина увеличится на 1.

А вот `querySelector`, наоборот, возвращает статичный список узлов. Он ссылается на те же самые элементы, что бы не происходило с документом. Поэтому длина `aList2.length`

останется неизменной.



### Решение задачи: Бенчмаркинг методов поиска в DOM

Для бенчмаркинга будем использовать функцию `bench(f, times)`, которая запускает функцию `f` `times` раз и возвращает разницу во времени:

```
1 function bench(f, times) {  
2   var d = new Date();  
3   for(var i=0; i<times; i++) f();  
4   return new Date() - d;  
5 }
```

Первый вариант (неверный) — замерять разницу между функциями `runGet/runQuery`, вот так:

```
1 function runGet() {  
2   var results = document.getElementsByTagName('p');  
3 }  
4  
5 function runQuery() {  
6   var results = document.querySelectorAll('p');  
7 }  
8  
9 alert( bench(runGet, 10000) ); // вывести время  
1000*runGet
```

Он даст неверные результаты, т.к. `getElementsByTagName` является «живым поисковым запросом». Если не обратиться к его результатам, то поиска не произойдет вообще, т.е. `runGet` ничего по сути не ищет.

...А `querySelectorAll` всегда производит поиск и формирует список элементов.

Более правильный тест — это не только запустить поиск, но и получить все элементы, как это делается в реальной жизни.

Полное решение:

<http://learn.javascript.ru/play/tutorial/browser/dom/speed-selector/index.html>



### Решение задачи: Получить второй LI

Можно так:

```
var li = ul.getElementsByTagName('li')[1];
```

Или так:

```
var li = ul.querySelector('li:nth-child(2)');
```

Оба этих вызова будут перебирать детей UL и остановят перебор на найденном элементе.

А вот так — браузер найдет все элементы, а затем выберет второй. Это дольше:

```
var li = ul.querySelectorAll('li')[1];
```

На практике разница в производительности будет видна только

для действительно больших списков, либо при частом выполнении запроса. Браузер перебирает элементы весьма шустро.



### Решение задачи: createTextNode vs innerHTML

Результат выполнения может быть разный.

Запустите следующие примеры, чтобы увидеть разницу:

- createTextNode создает текст '**текст**':

```
1 <div></div>
2 <script>
3   var text = '<b>текст</b>';
4
5   var elem = document.body.children[0];
6   elem.appendChild(document.createTextNode(text));
7 </script>
```

- innerHTML присваивает HTML **текст**:

```
1 <div></div>
2 <script>
3   var text = '<b>текст</b>';
4
5   var elem = document.body.children[0];
6   elem.innerHTML = text;
7 </script>
```



### Решение задачи: Удаление элементов

Родителя parentNode можно получить из elem.

Нужно учесть два момента.

1. Родителя может не быть (элемент уже удален или еще не вставлен).
2. Для совместимости со стандартным методом нужно вернуть удаленный элемент.

Вот так выглядит решение:

```
function remove(elem) {
  return elem.parentNode ?
    elem.parentNode.removeChild(elem) : elem;
}
```



### Решение задачи: insertAfter

Для того, чтобы добавить элемент *после* refElem, мы можем вставить его *перед* refElem.nextSibling.

Но что если nextSibling нет? Это означает, что refElem является последним потомком своего родителя и можем использовать appendChild.

Код:

```
1 function insertAfter(elem, refElem) {
2   var parent = refElem.parentNode;
3   var next = refElem.nextSibling;
```



```

4   if (next) {
5       return parent.insertBefore(elem, next);
6   } else {
7       return parent.appendChild(elem);
8   }
9 }

```

Но код может быть гораздо короче, если использовать фишку со вторым аргументом `null` метода `insertBefore`:

```

function insertAfter(elem, refElem) {
    return refElem.parentNode.insertBefore(elem,
    refElem.nextSibling);
}

```

Если нет `nextSibling`, то второй аргумент `insertBefore` становится `null` и тогда `insertBefore(elem, null)` работает как `appendChild`.

В решении нет проверки на существование `refElem.parentNode`, поскольку вставка после элемента без родителя — уже ошибка, пусть она возникнет в функции, это нормально.



## Решение задачи: `removeChildren`

### Неправильное решение

Для начала рассмотрим забавный пример того, как делать *не надо*:

```

1 function removeChildren(elem) {
2     for(var k=0; k<elem.childNodes.length;k++) {
3         elem.removeChild(elem.childNodes[k]);
4     }
5 }

```

Если вы попробуете это на практике, то увидите, то это не сработает.

Не сработает потому, что `childNodes` всегда начинается 0 и автоматически смещается, когда первый потомок удален(т.е. тот, что был вторым, станет первым), поэтому такой цикл по `k` пропустит половину узлов.

### Решение через DOM

Правильное решение:

```

1 function removeChildren(elem) {
2     while(elem.lastChild) {
3         elem.removeChild(elem.lastChild);
4     }
5 }

```

### Неправильное решение (`innerHTML`)

Прямая попытка использовать `innerHTML` была бы неправильной:

```

function removeChildren(elem) {
    elem.innerHTML = '';
}

```

Дело в том, что в IE<9 свойство `innerHTML` на большинстве табличных элементов (кроме ячеек `TH`/`TD`) не работает. Будет

ошибка.

## Верное решение (innerHTML)

Можно завернуть innerHTML в try/catch:

```
1 function removeChildren(elem) {  
2   try {  
3     elem.innerHTML = '';  
4   } catch(e) {  
5     while(elem.firstChild) {  
6       elem.removeChild(elem.firstChild);  
7     }  
8   }  
9 }
```



### Решение задачи: Почему остаётся "aaa" ?

HTML в задаче некорректен. В этом всё дело. И вопрос легко решится, если открыть отладчик.

В нём видно, что браузер поместил текст *aaa* *перед* таблицей. Поэтому он и остался в документе.

Вообще, в стандарте HTML5 описано, как браузеру обрабатывать некорректный HTML, так что такое действие браузера является правильным.



### Решение задачи: Создать список

Решение говорит само за себя:

```
01 <!DOCTYPE HTML>  
02 <html>  
03 <body>  
04 <h1>Создание списка</h1>  
05  
06 <script>  
07   var ul = document.createElement('ul');  
08   document.body.appendChild(ul);  
09  
10   while (true) {  
11     var data = prompt("Введите текст для пункта  
списка", "");  
12  
13     if (data === null) {  
14       break;  
15     }  
16  
17     var li = document.createElement('li');  
18     li.appendChild(document.createTextNode(data));  
19     ul.appendChild(li);  
20   }  
21 </script>  
22  
23 </body>  
24 </html>
```

Делайте проверку на null в цикле. prompt возвращает это значение только если был нажат ESC.

Контент в LI добавляйте с помощью document.createTextNode, чтобы правильно работали <, > и т.д.



### Решение задачи: Создайте дерево из объекта

Решения через рекурсию.

1. <http://learn.javascript.ru/play/tutorial/browser/dom/build-tree.html>.
2. <http://learn.javascript.ru/play/tutorial/browser/dom/build-tree-dom.html>.



### Решение задачи: Создать календарь в виде таблицы

Для решения задачи сгенерируем таблицу в виде строки: "<table>...</table>", а затем присвоим в innerHTML.

Алгоритм:

1. Создать объект даты `d = new Date(year, month-1)`. Это первый день месяца month (с учетом того, что месяцы в JS начинаются от 0, а не от 1).
2. Ячейки первого ряда пустые от начала и до дня недели `d.getDay()`, с которого начинается месяц. Создадим их.
3. Увеличиваем день в `d` на единицу: `d.setDate(d.getDate()+1)`, и добавляем в календарь очередную ячейку, пока не достигли следующего месяца. При этом последний день недели означает вставку перевода строки "</tr><tr>".
4. При необходимости, если календарь окончился не на воскресенье - добавить пустые TD в таблицу, чтобы было все ровно.

Код решения находится здесь:

[http://learn.javascript.ru/play/tutorial/date/calendar\\_rus.html](http://learn.javascript.ru/play/tutorial/date/calendar_rus.html)



### Решение задачи: Часики с использованием "setInterval"

Для начала, придумаем подходящую HTML/CSS-структуру.

Здесь каждый компонент времени удобно поместить в соответствующий SPAN:

```
<div id="clock">
  <span class="hour">hh</span>:<span
class="min">mm</span>:<span class="sec">ss</span>
</div>
```

Каждый SPAN раскрашивается при помощи CSS.

Жизнь часам будет обеспечивать функция `update`, вызываемая каждую секунду: `setInterval(update, 1000)`.

```
01 var timerId; // таймер, если часы запущены
02
03 function clockStart() { // запустить часы
04   if (timerId) return;
05
06   timerId = setInterval(update, 1000);
07   update(); // (*)
08 }
09
10 function clockStop() {
11   clearInterval(timerId);
12   timerId = null;
```

13 | }

Обратите внимание, что вызов `update` не только запланирован, но и производится тут же в строке `(*)`. Иначе посетителю пришлось бы ждать до первого выполнения `setInterval`.

Функция обновления часов:

```
01 function update() {  
02     var clock = document.getElementById('clock');  
03     var date = new Date(); // (*)  
04     var hours = date.getHours();  
05     if (hours < 10) hours = '0'+hours;  
06     clock.children[0].innerHTML = hours;  
07  
08     var minutes = date.getMinutes();  
09     if (minutes < 10) minutes = '0'+minutes;  
10     clock.children[1].innerHTML = minutes;  
11  
12     var seconds = date.getSeconds();  
13     if (seconds < 10) seconds = '0'+seconds;  
14     clock.children[2].innerHTML = seconds;  
15 }
```

В строке `(*)` каждый раз мы получаем текущую дату. Мы должны это сделать, несмотря на то, что, казалось бы, могли бы просто увеличивать счетчик каждую секунду.

На самом деле мы не можем опираться на счетчик для вычисления даты, т.к. `setInterval` не гарантирует точную задержку. Если в другом участке кода будет вызван `alert`, то часы остановятся, как и любые счетчики.

Полный код решения:

<http://learn.javascript.ru/play/tutorial/advanced/timing/clock-interval/index.html>.



#### Решение задачи: Часы при помощи "setTimeout"

Общее решение описано в [аналогичной задаче с setInterval](#).

Способ через `setTimeout` — по сути, такой же, только функция `update` каждый раз ставит себя в очередь заново.

Код решения:

<http://learn.javascript.ru/play/tutorial/advanced/timing/clock-timeout/index.html>.

Заметим, что в данном случае целесообразнее использовать `setInterval`, т.к. нужна не задержка между запусками, а просто запуск каждую секунду.



#### Решение задачи: Вставьте элементы в конец списка

Решение:

```
var ul = document.body.children[0];  
ul.insertAdjacentHTML("beforeEnd",  
    "<li>3</li><li>4</li><li>5</li>");
```



#### Решение задачи: Вставка

## insertAdjacentHTML/DocumentFragment

### Подсказки

- Проверить поддержку insertAdjacentHTML можно так:

```
if (elem.insertAdjacentHTML) { ... }
```

- Если этот метод не поддерживается, то сделайте временный элемент, через innerHTML поставьте туда html, а затем переместите содержимое в DocumentFragment. Последнее действие — вставка в документ.

### Решение

```
01 <ul>
02   <li>1</li>
03   <li>2</li>
04   <li>5</li>
05 </ul>
06
07 <script>
08 var ul = document.body.children[0];
09 var li5 = ul.children[2];
10
11 function insertBefore(elem, html) {
12   if (elem.insertAdjacentHTML) {
13     elem.insertAdjacentHTML("beforeBegin", html);
14   } else {
15     var fragment =
16       document.createDocumentFragment();
17
18     var tmp = document.createElement('DIV');
19     tmp.innerHTML = html;
20
21     while(tmp.firstChild) {
22       // перенести все узлы во fragment
23       fragment.appendChild(tmp.firstChild);
24     }
25     elem.parentNode.insertBefore(fragment, elem);
26   }
27 }
28
29 insertBefore(li5, "<li>3</li><li>4</li>")
30 </script>
```



### Решение задачи: Отсортировать таблицу

Для сортировки нам поможет функция `sort` массива.

Общая идея лежит на поверхности: сделать массив из строк и отсортировать его. Тонкости кроются в деталях.

В ифрейме ниже загружен документ, описывающий и реализующий разные алгоритмы. Обратите внимание: разница в производительности может достигать нескольких раз!

#### Алгоритм 1.

1. Все `tr` удалить из таблицы, при этом собрав их в JavaScript-массив.
2. Отсортировать этот массив, используя свою функцию в `sort(...)` для сравнения `tr`.
3. Добавить `tr` из массива в таблицу в нужном порядке.

Померять время

#### Алгоритм 2.

1. Скопировать `tr` в JavaScript-массив.

2. Отсортировать этот массив, используя свою функцию в `sort(...)` для сравнения `TR`.
3. Добавить `TR` из массива в таблицу в нужном порядке. При добавлении каждый `TR` сам удалится с предыдущего места.

Померять время

Алгоритм 3.

1. Создать массив из объектов вида `{elem: ссылка на TR, value: содержимое TR}`.
2. Отсортировать массив по `value`. Функция сравнения во время сортировки теперь будет обращаться не к `innerHTML`, а к свойству объекта, это быстрее. Сортировка может потребовать многократных сравнений одного и того же элемента, отсюда выигрыш.
3. Добавить `TR` в таблицу в нужном порядке (автоудалятся с предыдущего места).

Померять время

Алгоритм 4.

1. Выполнить алгоритм 3, но перед этим удалить таблицу из документа, а после - вставить обратно.

Померять время

Алгоритм 5.

1. Замерить время генерации таблицы (создаётся строка и пишется в `innerHTML`).

Померять время

Содержимое документа для придания "реалистичности"

P.S. Создавать `DocumentFragment` здесь ни к чему. Можно вытащить из документа `TBODY` и иметь дело с ним в отрыве от DOM (алгоритм 4).

P.P.S. Если нужно сделать много узлов, то обычно `innerHTML` работает быстрее, чем генерация элементов через DOM-вызовы. Но в данном случае мы не создаём элементы, а сортируем и перевставляем готовые, так что результаты могут отличаться.



## Решение задачи: Загрузите скрипт с передачей разрешения экрана

1. Вариант с `document.write`:

```
1 var x = screen.width;
2 var y = screen.height;
3 var src = 'http://ads.com/load.js?
  x='+x+'&y='+y+'&r='+Math.random();
4 document.write('<script
  src="'+src+'"></script>');
```

Обратите внимание: **закрывающий `</script>` нужно разбить на две части**, иначе браузер подумает, что это конец скрипта, и выдаст ошибку.

2. Вариант с DOM:

```
1 var x = screen.width;
2 var y = screen.height;
3 var src = 'http://ads.com/load.js?
  x='+x+'&y='+y+'&r='+Math.random();
4 var script = document.createElement('script');
5 script.src = src;
6 document.documentElement.children[0].appendChild(script);
```



## Решение задачи: Поставьте класс ссылкам

### Алгоритм

Сначала можно найти ссылки, например, при помощи `document.getElementsByTagName('a')`.

Как вы думаете, для проверки адреса нужно использовать свойство `href` или атрибут `getAttribute('href')` ? В чем будет различие?

Свойство будет содержать полный путь ссылки (во всех браузерах, кроме IE<9), а атрибут — значение, указанное в HTML. В данном случае подойдет и то и другое.

Правила определения:

- Ссылки без протокола `://` являются заведомо внутренними.
- Там, где протокол есть — нужно создать новую строку из того, что идет после `://` (`indexOf + slice`) и проверить начало на совпадение с `javascript.ru`.

## Решение

Код решения:

<http://learn.javascript.ru/play/tutorial/browser/dom/markLinks.html>



## Решение задачи: Скругленная кнопка со стилями из JavaScript

Есть два варианта.

1. Можно использовать свойство `elem.style.cssText` и присвоить стиль в текстовом виде. При этом все присвоенные ранее свойства `elem.style` будут удалены.
2. Можно назначить подсвойства `elem.style` одно за другим. Это более безопасно, т.к. меняет только явно присваиваемые свойства.

Мы выберем второй путь в решении:

<http://learn.javascript.ru/play/tutorial/browser/dom/roundedButton/solution.html>

### Описание CSS-свойств:

```
01 .button {
02   -moz-border-radius: 8px;
03   -webkit-border-radius: 8px;
04   border-radius: 8px;
05   border: 2px groove green;
06   display: block;
07   height: 30px;
08   line-height: 30px;
09   width: 100px;
10   text-decoration: none;
11   text-align: center;
12   color: red;
13   font-weight: bold;
14 }
```

#### **\*-border-radius**

Добавляет скругленные углы. Свойство присваивается в вариантах для Firefox `-moz-...`, Chrome/Safari `-webkit-...` и стандартное CSS3-свойство для тех, кто его поддерживает (Opera).

#### **display**

По умолчанию, у А это свойство имеет значение

`display: inline.`

### **height, line-height**

Устанавливает высоту и делает текст вертикально центрированным путем установки `line-height` в значение, равное высоте. Такой способ центрирования текста работает, если он состоит из одной строки.

### **text-align**

Центрирует текст горизонтально.

### **color, font-weight**

Делает текст красным и жирным.



Лучше не присваивать свойства в JavaScript, если оформление можно вынести в CSS.

В отличие от разобранных примера, как правило, JavaScript лишь добавляет или убирает класс, а стили настраиваются уже в CSS.



### **Решение задачи: Создать уведомление**

Решение:

<http://learn.javascript.ru/play/tutorial/browser/dom/notification-style/index.html>.



### **Решение задачи: Найти размер прокрутки снизу**

Решение:

`elem.scrollHeight - elem.scrollTop - elem.clientHeight`.



### **Решение задачи: Узнать ширину полосы прокрутки**

Создадим элемент с прокруткой, но без padding. Тогда разница между его полной шириной `offsetWidth` и внутренней `clientWidth` будет равна как раз прокрутке:

```
01 // создадим элемент с прокруткой
02 var div = document.createElement('div');
03
04 div.style.overflowY = 'scroll';
05 div.style.width = '50px';
06 div.style.height = '50px';
07
08 // при display:none размеры нельзя узнать
09 // нужно, чтобы элемент был видим,
10 // visibility:hidden - можно, т.к. сохраняет
   геометрию
11 div.style.visibility = 'hidden';
12
13 document.body.appendChild(div);
14 var scrollWidth = div.offsetWidth -
   div.clientWidth;
15 document.body.removeChild(div);
16
17 alert( scrollWidth );
```



### **Решение задачи: Подменить div на другой с таким же**



## размером

Нам нужно создать `div` с такими же размерами и вставить его на место «переезжающего».

Один из вариантов — это просто клонировать элемент.

Если делать это при помощи `div.cloneNode(true)`, то скопируется все содержимое, которого может быть много.

Обычно нам это не нужно, поэтому можно использовать `div.cloneNode(false)` для клонирования элемента со стилями, и потом поправить его `width/height`.

Можно и просто создать новый `div` и поставить ему нужные размеры.

**Всё, кроме `margin`, можно получить из свойств DOM-элемента, а `margin` — только через `getComputedStyle`.**

Причём `margin` мы обязаны поставить, так как иначе элемент не будет отодвинут от внешних.

Код:

```
01 var div = document.getElementById('moving-div');
02
03 var placeholder = document.createElement('div');
04 placeholder.style.height = div.offsetHeight + 'px';
05 // можно и width, но в этом примере это не обязательно
06
07 // IE || другой браузер
08 var computedStyle = div.currentStyle ||
    getComputedStyle(div, '');
09
10 placeholder.style.marginTop =
    computedStyle.marginTop; // (1)
11 placeholder.style.marginBottom =
    computedStyle.marginBottom;
```

В строке (1) использование полного название свойства `"marginTop"` гарантирует, что полученное значение будет корректным.

Конечный результат (смотри SCRIPT):

```
01 <!DOCTYPE HTML>
02 <html>
03 <head>
04 <style>
05   #moving-div {
06     border: 5px groove green;
07     padding: 5px;
08     margin: 10px;
09     background-color: yellow;
10   }
11 </style>
12 </head>
13 <body>
14
15 Before Before Before
16
17 <div id="moving-div">
18 Text Text Text<br>
19 Text Text Text<br>
20 </div>
21
22 After After After
```

```

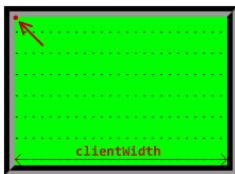
23
24 <script>
25 var div = document.getElementById('moving-div')
26
27 var placeholder = document.createElement('div')
28 placeholder.style.height = div.offsetHeight + 'px'
29
30 var computedStyle = div.currentStyle ||
  getComputedStyle(div, null)
31
32 placeholder.style.marginTop =
  computedStyle.marginTop // full prop name
33 placeholder.style.marginBottom =
  computedStyle.marginBottom
34
35 // highlight it for demo purposes
36 placeholder.style.backgroundColor = '#C0C0C0'
37
38 document.body.insertBefore(placeholder, div)
39
40 div.style.position = 'absolute'
41 div.style.right = div.style.top = 0
42
43 </script>
44
45
46 </body>
47 </html>

```



### Решение задачи: Поместите мяч в центр поля

При абсолютном позиционировании мяча внутри поля его координаты left/top отсчитываются от **внутреннего** угла поля, например верхнего-левого:



Метрики для внутренней зоны поля — это `clientWidth/Height`.

Центр - это  $(clientWidth/2, clientHeight/2)$ .

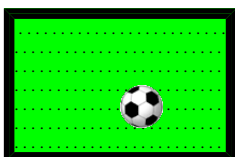
Но если мы установим мячу такие значения

`ball.style.left/top`, то в центре будет не сам мяч, а его левый верхний угол:

```

1 var ball = document.getElementById('ball');
2 var field = document.getElementById('field');
3
4 ball.style.left = Math.round(field.clientWidth /
  2)+'px';
5 ball.style.top = Math.round(field.clientHeight /
  2)+'px';

```



Для того, чтобы центр мяча находился в центре поля, нам нужно сместить мяч на половину его ширины влево и на половину его высоты вверх.

```

1 var ball = document.getElementById('ball');
2 var field = document.getElementById('field');
3
4 ball.style.left = Math.round(field.clientWidth/2 -
  ball.offsetWidth/2)+'px';
5 ball.style.top = Math.round(field.clientHeight/2 -
  ball.offsetHeight/2)+'px';

```

### Внимание, подводный камень!

Код выше стабильно работать не будет, потому что IMG идет без ширины/высоты:

```

```

**Высота и ширина изображения неизвестны браузеру до тех пор, пока оно не загрузится, если размер не указан явно.**

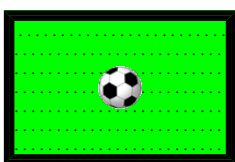
После первой загрузки изображение уже будет в кеше браузера, и его размеры будут известны. Но когда браузер впервые видит документ — он ничего не знает о картинке, поэтому значение `ball.offsetWidth` равно 0. Вычислить координаты невозможно.

Чтобы это исправить, добавим `width/height` к картинке:

```

```

Теперь браузер всегда знает ширину и высоту, так что все работает. Тот же эффект дало бы указание размеров в CSS.



Полный код решения:

<http://learn.javascript.ru/play/tutorial/browser/dom/ball/index.html>



### Решение задачи: Расширить элемент

**Вначале рассмотрим неверный вариант.**

Он выглядит так:

```
elem.style.width = '100%';
```

Если вы его попробуете, то увидите, что элемент начинает вылезать за рамки родителя.

Так происходит потому, что ширина — это то, что *внутри padding*. То есть, ставя ширину в 100%, вы говорите: «внутренняя область должна занимать 100% доступной ширины». А на padding остаётся 0%. В результате поля вылезают наружу.

**Правильное решение через `clientWidth`.**

Доступную внутреннюю ширину родителя можно получить, вычитая `padding` из `clientWidth`, и присвоить элементу:

```

1 var bodyClientWidth = document.body.clientWidth;
2
3 var style = window.getComputedStyle ?

```

```

4   getComputedStyle(elem, '') : elem.currentStyle;
5   var bodyInnerWidth = bodyClientWidth -
   parseInt(style.paddingLeft) -
   parseInt(style.paddingRight);
6
7   elem.style.width = bodyInnerWidth + 'px';

```

Этот вариант сломается, если в IE<9 значение padding указано не в пикселях. Получение пикселей из процентов и других единиц измерения рассмотрено в главе [Стили и классы](#), [getComputedStyle](#).

#### Правильный вариант с CSS.

**Самое лучшее решение получится, если вспомнить, что элемент и сам рад растянуться по всей доступной ширине, и делает это по умолчанию.**

Достаточно вернуть ему стандартный алгоритм вычисления ширины, установив `width: auto`:

```
elem.style.width = 'auto';
```

Но.. Это не будет работать для элементов, которые сами по себе не растягиваются, например в случае `position: absolute` или `float`.

Такой элемент можно расширить, используя предыдущее решение.

Документ с обоими решениями:

<http://learn.javascript.ru/play/tutorial/browser/dom/metric/bodywidth/index.html>



#### Решение задачи: В чём отличие "width" и "clientWidth" ?

Отличия:

1. `getComputedStyle` не работает в IE<9.
2. `clientWidth` соответствует внутренней видимой области элемента, *включая padding*.  
..А свойство `width`, при стандартном значении [box-sizing](#), соответствует зоне *внутри padding*.
3. Если есть полоса прокрутки, то некоторые браузеры включают её ширину в `width`, а некоторые — нет.

Свойство `clientWidth`, с другой стороны, полностью кросс-браузерно. Оно всегда обозначает размер *за вычетом прокрутки*, т.е. реально доступный для содержимого.



#### Решение задачи: Получить прокрутки документа

`top` — можно кроссбраузерно получить, как указано в главе [Размеры и прокрутка для страницы](#):

```

1   function getDocumentScrollTop() {
2       var html = document.documentElement;
3       var body = document.body;

```

```

4   |
5   |     var scrollTop = html.scrollTop || body &&
    |     body.scrollTop || 0;
6   |     scrollTop -= html.clientTop; // IE<8
7   |
8   |     return scrollTop;
9   | }

```

`bottom` — это `top` плюс высота видимой части:

```

function getDocumentScrollBottom() {
    return getDocumentScrollTop() +
    document.documentElement.clientHeight;
}

```

Полная высота — максимум двух значений, детали см. в [Размеры и прокрутка для страницы](#):

```

1  | function getDocumentScrollHeight() {
2  |     var scrollHeight =
    |     document.documentElement.scrollHeight;
3  |     var clientHeight =
    |     document.documentElement.clientHeight;
4  |
5  |     scrollHeight = Math.max(scrollHeight,
    |     clientHeight);
6  |
7  |     return scrollHeight;
8  | }

```

Итого, ответ, использующий описанные выше функции:

```

1  | function getDocumentScroll() {
2  |     return {
3  |         top: getDocumentScrollTop(),
4  |         bottom: getDocumentScrollBottom(),
5  |         height: getDocumentScrollHeight()
6  |     };
7  | }

```



## Решение задачи: Найдите координаты

### Координаты внешних углов

Координаты элемента возвращаются функцией `elem.getBoundingClientRect`. Она возвращает все координаты относительно окна в виде объекта со свойствами `left`, `top`, `right`, `bottom`. Некоторые браузеры также добавляют `width`, `height`.

Так что координаты верхнего-левого `coords1` и правого-нижнего `coords4` внешних углов:

```

1  | var coords = elem.getBoundingClientRect;
2  |
3  | var coords1 = [coords.left, coords.top];
4  | var coords4 = [coords.right, coords.bottom];

```

### Левый-верхний угол внутри

Этот угол отстоит от наружных границ на размер рамки, который доступен через `clientLeft/clientTop`:

```

var coords2 = [coords.left + field.clientLeft,
coords.top + field.clientTop];

```

## Правый-нижний угол внутри

Этот угол отстоит от правой-нижней наружной границы на размер рамки. Так как нужная рамка находится справа-внизу, то специальных свойств для нее нет, но мы можем получить этот размер из CSS:

```
1 var coords3 = [  
2   coords.right -  
3   parseInt(getComputedStyle(field).borderRightWidth) ,  
4   coords.bottom -  
   parseInt(getComputedStyle(field).borderBottomWidth)  
]
```

Можно получить их альтернативным путем, прибавив `clientWidth/clientHeight` к координатам `coords2` левого-верхнего внутреннего угла. Получится то же самое, пожалуй даже быстрее и изящнее.

Полный код решения:

<http://learn.javascript.ru/play/tutorial/browser/dom/field-coords/index.html>.

## Координаты относительно страницы

Чтобы перейти к координатам относительно страницы, нужно добавить к результату `elem.getBoundingClientRect` текущую прокрутку документа, как описано в функции `getCoords`.



### Решение задачи: Разместить заметку рядом с элементом

Решение: <http://learn.javascript.ru/play/tutorial/browser/dom/position-at/index.html>



### Решение задачи: Разместить заметку внутри элемента

Решение: <http://learn.javascript.ru/play/tutorial/browser/dom/position-at-2/index.html>

Кликните на любое место, чтобы получить координаты относительно окна.  
Это для удобства тестирования, чтобы проверить результат, который вы получите из DOM.  
(координаты появятся тут)



Lorem ipsum dolor sit amet, consectetur adipisicing elit. Reprehenderit sint atque dolorum fuga ad incidunt voluptatum error fugiat animi amet! Odio temporibus nulla id unde quaerat dignissimos enim nisi rem provident molestias sit tempore omnis recusandae esse sequi officia sapiente.

- Что на завтрак, Бэрримор?
- Овсянка, сэр.
- А на обед?

Lorem ipsum dolor sit amet, consectetur adipisicing elit. Reprehenderit sint atque dolorum fuga ad incidunt voluptatum error fugiat animi amet! Odio temporibus nulla id unde quaerat dignissimos enim nisi rem provident molestias sit tempore omnis recusandae esse sequi officia sapiente.

- Что на завтрак, Бэрримор?
- Овсянка, сэр.
- А на обед?