**Тема 6. Клиентский JavaScript**

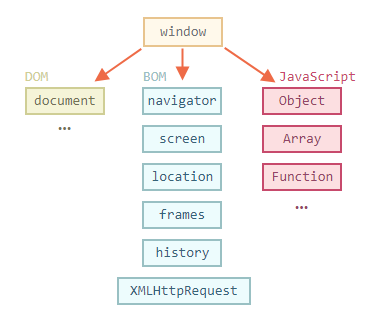
1. Окружение: DOM, BOM.
2. Дерево DOM.
3. Навигация по DOM-элементам.
4. Методы поиска DOM-элементов.
5. Свойства узлов: тип, тег и содержимое.
6. Атрибуты и DOM-свойства.
7. Добавление и удаление узлов.
8. Стили.
9. Размеры и прокрутка элементов.
10. Размеры и прокрутка страницы.
11. Координаты.

Содержание данной темы включает материалы, доступные по адресу https://learn.javascript.ru.

1. **Окружение: DOM, BOM.**

Язык JavaScript изначально был создан для веб-браузеров. Но с тех пор он значительно эволюционировал и превратился в кроссплатформенный язык программирования для решения широкого круга задач. Сегодня JavaScript может использоваться в браузере, на веб-сервере или в какой-то другой среде, даже в кофеварке. Каждая среда предоставляет свой функционал, который спецификация JavaScript называет *окружением*.

Окружение предоставляет свои объекты и дополнительные функции, в дополнение базовым языковым. Браузеры, например, дают средства для управления веб-страницами. Node.js делает доступными какие-то серверные возможности и так далее. На изображении ниже в общих чертах показано, что доступно для JavaScript в браузерном окружении:



Как видно, имеется корневой объект window, который выступает в 2 ролях:

1. Во-первых, это глобальный объект для JavaScript-кода.
2. Во-вторых, он также представляет собой окно браузера и располагает методами для управления им.

Например, здесь используется window как глобальный объект:

function sayHi() {

alert("Hello");

}

window.sayHi();

А здесь используется window как объект окна браузера, чтобы узнать его высоту:

alert(window.innerHeight); // внутренняя высота окна браузера

Существует гораздо больше свойств и методов для управления окном браузера. Они будут рассмотрены позднее.

[**DOM (Document Object Model)**](https://learn.javascript.ru/browser-environment#dom-document-object-model)

*Document Object Model*, сокращенно DOM – объектная модель документа, которая представляет все содержимое страницы в виде объектов, которые можно менять.

*Объект document* – основная «входная точка». С его помощью можно что-то создавать или менять на странице. Например:

// заменим цвет фона на красный,

document.body.style.background = "red";

// а через секунду вернём как было

setTimeout(() => document.body.style.background = "", 1000);

В примере использован только document.body.style, но на самом деле возможности по управлению страницей намного шире. Различные свойства и методы описаны в спецификации: DOM Living Standard на [https://dom.spec.whatwg.org](https://dom.spec.whatwg.org/).

Спецификация DOM описывает структуру документа и предоставляет объекты для манипуляций со страницей. Существует и другие, отличные от браузеров, инструменты, использующие DOM. Например, серверные скрипты, которые загружают и обрабатывают HTML-страницы, также могут использовать DOM. При этом они могут поддерживать спецификацию не полностью.

Правила стилей CSS структурированы иначе чем HTML. Для них есть отдельная спецификация [CSSOM](https://www.w3.org/TR/cssom-1/), которая объясняет, как стили должны представляться в виде объектов, как их читать и писать. CSSOM используется вместе с DOM при изменении стилей документа. В реальности CSSOM требуется редко, обычно правила CSS статичны. Стили из JavaScript редко добавляются/удаляются, но и это возможно.

[**BOM (Browser Object Model)**](https://learn.javascript.ru/browser-environment#bom-browser-object-model)

*Объектная модель браузера (Browser Object Model, BOM)* – это дополнительные объекты, предоставляемые браузером (окружением), чтобы работать со всем, кроме документа. Например:

* Объект [navigator](https://developer.mozilla.org/ru/docs/Web/API/Window/navigator) даёт информацию о самом браузере и операционной системе. Среди множества его свойств самыми известными являются: navigator.userAgent – информация о текущем браузере, и navigator.platform – информация о платформе (может помочь в понимании того, в какой ОС открыт браузер – Windows/Linux/Mac и так далее).
* Объект [location](https://developer.mozilla.org/ru/docs/Web/API/Window/location) позволяет получить текущий URL и перенаправить браузер по новому адресу.

Вот как можно использовать объект location:

alert(location.href); // показывает текущий URL

if (confirm("Перейти на Wikipedia?")) {

location.href = "https://wikipedia.org"; // перенаправляет браузер на другой URL

}

Функции alert/confirm/prompt тоже являются частью BOM: они не относятся непосредственно к странице, но представляют собой методы объекта окна браузера для коммуникации с пользователем.

BOM является частью общей [спецификации HTML](https://html.spec.whatwg.org/). Спецификация HTML по адресу [https://html.spec.whatwg.org](https://html.spec.whatwg.org/) не только про «язык HTML» (теги, атрибуты), она также покрывает целое множество объектов, методов и специфичных для каждого браузера расширений DOM. Это всё «HTML в широком смысле». Для некоторых вещей есть отдельные спецификации, перечисленные на [https://spec.whatwg.org](https://spec.whatwg.org/).

1. **Дерево DOM.**

Основой HTML-документа являются теги. В соответствии с объектной моделью документа («Document Object Model», коротко DOM), каждый HTML-тег является объектом. Вложенные теги являются «детьми» родительского элемента. Текст, который находится внутри тега, также является объектом. Все эти объекты доступны при помощи JavaScript, можно использовать их для изменения страницы. Например, document.body – объект для тега <body>. Если запустить этот код, то <body> станет красным на 3 секунды:

document.body.style.background = 'red'; // сделать фон красным

setTimeout(() => document.body.style.background = '', 3000); // вернуть назад

Это был лишь небольшой пример того, что может DOM.

**[Структура DOM](https://learn.javascript.ru/dom-nodes" \l "primer-dom)**

Разберемся со структурой DOM. Рассмотрим простой документ:

<!DOCTYPE HTML>

<html>

<head>

<title>О лосях</title>

</head>

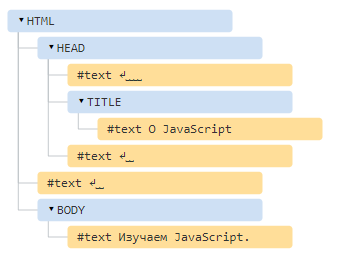
<body>

Правда о лосях.

</body>

</html>

DOM – это представление HTML-документа в виде дерева тегов. Вот как оно выглядит:



Каждый узел этого дерева – это объект. Теги являются узлами-элементами (или просто элементами). Они образуют структуру дерева: <html> – это корневой узел, <head> и <body> его дочерние узлы, и т.д. Текст внутри элементов образует текстовые узлы, обозначенные как #text. Текстовый узел содержит в себе только строку текста. У него не может быть потомков, т.е. он находится всегда на самом нижнем уровне. Например, в теге <title> есть текстовый узел "О лосях".

Обратите внимание на специальные символы в текстовых узлах: перевод строки: ↵ (в JavaScript он обозначается как \n) и пробел: ␣. Пробелы и переводы строки – это полноправные символы, как буквы и цифры. Они образуют текстовые узлы и становятся частью дерева DOM. Так, в примере выше в теге <head> есть несколько пробелов перед <title>, которые образуют текстовый узел #text (он содержит в себе только перенос строки и несколько пробелов).

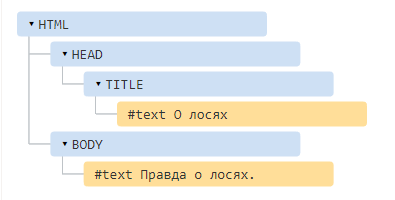
Существует всего два исключения из этого правила:

1. По историческим причинам пробелы и перевод строки перед тегом <head> игнорируются.
2. Если записать что-либо после закрывающего тега </body>, браузер автоматически перемещает эту запись в конец body, поскольку спецификация HTML требует чтобы все содержимое было внутри <body>. Поэтому после закрывающего тега </body> не может быть никаких пробелов.

В остальных если в документе есть пробелы (или любые другие символы), они становятся текстовыми узлами дерева DOM, и если их надо удалим, то в DOM их тоже не будет. Здесь пробельных текстовых узлов нет:

<!DOCTYPE HTML>

<html><head><title>О лосях</title></head><body>Правда о лосях.</body></html>



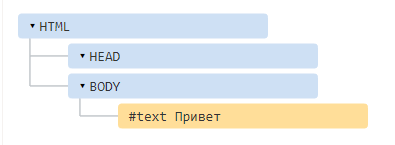
Пробелы по краям строк и пробельные текстовые узлы скрыты в инструментах разработки и обычно не отображаются. Таким образом инструменты разработки экономят место на экране.

В дальнейших иллюстрациях DOM также будут для краткости пропущены пробельные текстовые узлы там, где они не имеют значения. Обычно они не влияют на то, как отображается документ.

**[Автоисправление](https://learn.javascript.ru/dom-nodes" \l "avtoispravlenie)**

Если браузер сталкивается с некорректно написанным HTML-кодом, он автоматически корректирует его при построении DOM. Например, в начале документа всегда должен быть тег <html>. Даже если его нет в документе – он будет в дереве DOM, браузер его создаст. То же самое касается и тега <body>.

Например, если HTML-файл состоит из единственного слова "Привет", браузер обернёт его в теги <html> и <body>, добавит необходимый тег <head>, и DOM будет выглядеть так:



При генерации DOM браузер самостоятельно обрабатывает ошибки в документе, закрывает теги и так далее. Такой документ с незакрытыми тегами:

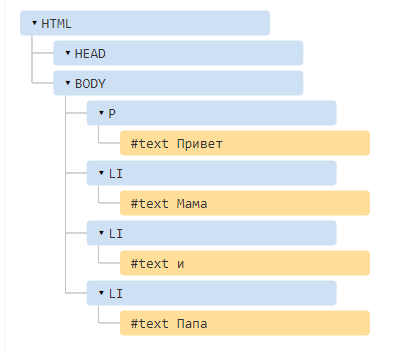
<p>Привет

<li>Мама

<li>и

<li>Папа

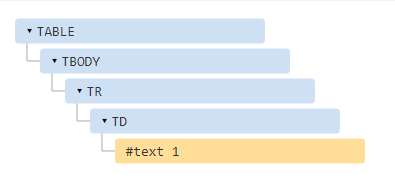
Но DOM будет нормальным, потому что браузер сам закроет теги и восстановит отсутствующие детали:



Важный «особый случай» – работа с таблицами. По стандарту DOM у них должен быть <tbody>, но в HTML их можно написать (официально) без него. В этом случае браузер добавляет <tbody> в DOM самостоятельно. Для такого HTML:

<table id="table"><tr><td>1</td></tr></table>

DOM-структура будет такой:



**[Другие типы узлов](https://learn.javascript.ru/dom-nodes" \l "drugie-tipy-uzlov)**

Есть и некоторые другие типы узлов, кроме элементов и текстовых узлов. Например, узел-комментарий:

<!DOCTYPE HTML>

<html>

<body>

Правда о лосях.

<ol>

<li>Лось -- животное хитрое</li>

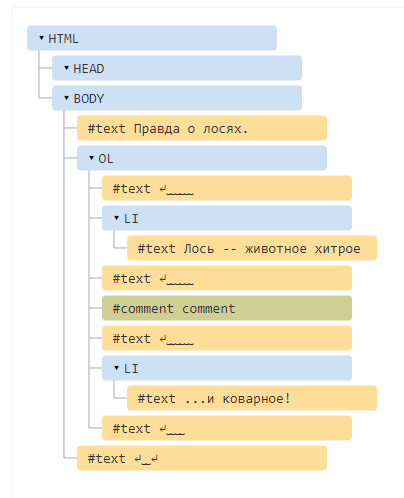
<!-- комментарий -->

<li>...и коварное!</li>

</ol>

</body>

</html>

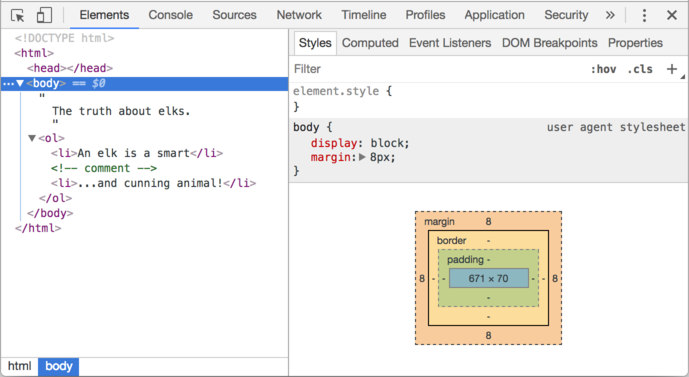


Здесь видно, что новый узел нового типа – комментарий, обозначенный как #comment, между двумя текстовыми узлами. Есть важное правило: если что-то есть в HTML, то оно должно быть в DOM-дереве. Поэтому все, что есть в HTML, даже комментарии, является частью DOM. Даже директива <!DOCTYPE...>, которая располагается в начале HTML, тоже является DOM-узлом. Она находится в дереве DOM прямо перед <html>. Даже объект document, представляющий весь документ, формально, является DOM узлом.

Существует [12 типов узлов](https://dom.spec.whatwg.org/#node). Но на практике в основном работают с четырьмя из них:

1. document – «входная точка» в DOM.
2. Узлы-элементы – HTML-теги, основные строительные блоки.
3. Текстовые узлы – содержат текст.
4. Комментарии – иногда в них можно включить информацию, которая не будет показана, но доступна в DOM для чтения JS.

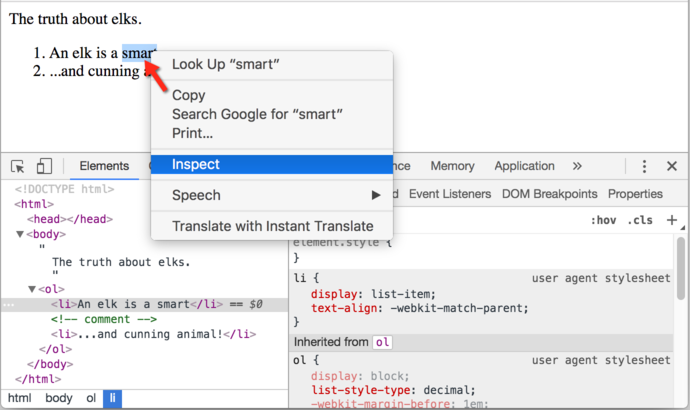
Чтобы посмотреть структуру DOM в реальном времени, можно использовать инструменты разработчика браузера. Для этого откройте страницу [\*.html](https://learn.javascript.ru/article/dom-nodes/elks.html), включите инструменты разработчика и перейдите на вкладку Elements. Выглядит примерно так:



Можно увидеть DOM, раскрыть/свернуть элементы, детально рассмотреть их и так далее. Обратите внимание, что структура DOM в инструментах разработчика отображается в упрощённом виде. Текстовые узлы показаны как простой текст. И кроме пробелов нет никаких «пустых» текстовых узлов, что очень удобно.

Клик по этой  кнопке в левом верхнем углу инспектора позволяет при помощи мыши (или другого устройства ввода) выбрать элемент на веб-странице и «проинспектировать» его (браузер сам найдёт и отметит его во вкладке Elements). Этот способ отлично подходит, когда есть огромная HTML-страница (и соответствующий ей огромный DOM), и надо увидеть, где находится интересующий нас элемент.

Есть и другой способ сделать это, можно кликнуть на странице по элементу правой кнопкой мыши и в контекстном меню выбрать «Inspect».



В правой части инструментов разработчика находятся следующие подразделы:

* Styles – здесь показан CSS, применённый к текущему элементу: правило за правилом, включая встроенные стили (выделены серым). Почти все можно отредактировать на месте, включая размеры/внешние и внутренние отступы.
* Computed – здесь видны итоговые CSS-свойства элемента, которые он приобрёл в результате применения всего каскада стилей (в том числе унаследованные свойства и т.д.).
* Event Listeners – в этом разделе видны обработчики событий, привязанные к DOM-элементам.

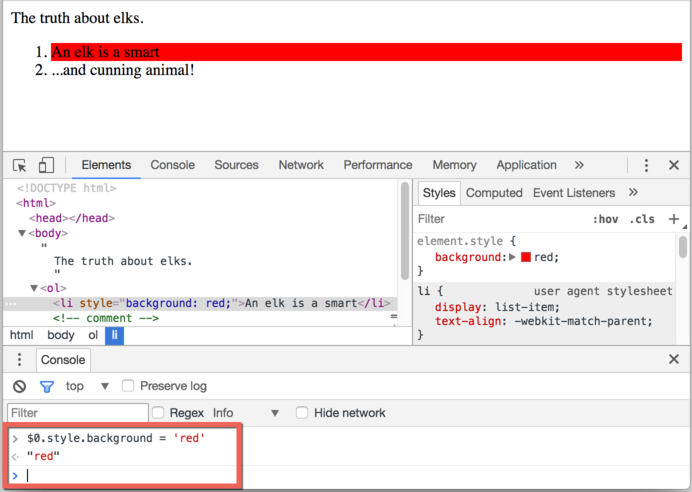
[**Взаимодействие с консолью**](https://learn.javascript.ru/dom-nodes#vzaimodeystvie-s-konsolyu)

При работе с DOM, часто требуется применить к нему JavaScript. Например: получить узел и запустить какой-нибудь код для его изменения, чтобы посмотреть результат. Вот несколько подсказок по тому, как перемещаться между вкладками Elements и Console. Для начала:

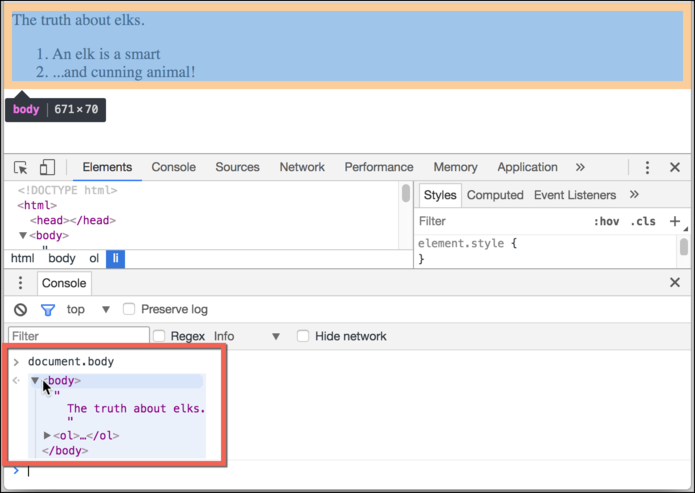
* На вкладке Elements выберите первый элемент <li>.
* Нажмите Esc – прямо под вкладкой Elements откроется Console.

Последний элемент, выбранный во вкладке Elements, доступен в консоли как $0, предыдущий, выбранный до него, как $1 и т.д.

Теперь можно запускать на них команды. Например $0.style.background = 'red' сделает выбранный элемент красным, как здесь:



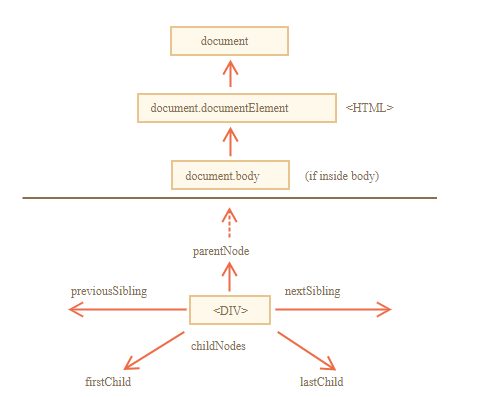
Так можно получить узел из Elements в Console. Есть и обратный путь: если есть переменная node, ссылающаяся на DOM-узел, можно использовать в консоли команду inspect(node), чтобы увидеть этот элемент во вкладке Elements. Или можно просто вывести DOM-узел в консоль и исследовать «на месте», как document.body ниже:



Это может быть полезно для отладки. Инструменты разработчика браузера отлично помогают в разработке: можно исследовать DOM, пробовать с ним что-то делать и смотреть, что идёт не так.

1. **Навигация по DOM-элементам.**

DOM позволяет делать что угодно с элементами и их содержимым, но для начала нужно получить соответствующий DOM-объект. Все операции с DOM начинаются с объекта document. Это главная «точка входа» в DOM. Из него можно получить доступ к любому узлу. Так выглядят основные ссылки, по которым можно переходить между узлами DOM:



Самые верхние элементы дерева доступны как свойства объекта document: <html> = document.documentElement. Самый верхний узел документа: document.documentElement. В DOM он соответствует тегу <html>.

<body> = document.body – другой часто используемый DOM-узел – узел тега <body>: document.body.

<head> = document.head – тег <head> доступен как document.head.

Есть одна тонкость: document.body может быть равен null. Нельзя получить доступ к элементу, которого еще не существует в момент выполнения скрипта. В частности, если скрипт находится в <head>, document.body в нём недоступен, потому что браузер его еще не прочитал. Поэтому, в примере ниже первый alert выведет null:

<html>

<head>

<script>

alert( "Из HEAD: " + document.body ); // null, <body> еще нет

</script>

</head>

<body>

<script>

alert( "Из BODY: " + document.body ); // HTMLBodyElement, теперь он есть

</script>

</body>

</html>

В DOM значение null значит «не существует» или «нет такого узла».

[**Дети: childNodes, firstChild, lastChild**](https://learn.javascript.ru/dom-navigation#deti-childnodes-firstchild-lastchild)

Здесь и далее будут использоваться два принципиально разных термина:

* Дочерние узлы (или дети) – элементы, которые являются непосредственными детьми узла. Другими словами, элементы, которые лежат непосредственно внутри данного. Например, <head> и <body> являются детьми элемента <html>.
* Потомки – все элементы, которые лежат внутри данного, включая детей, их детей и т.д.

В примере ниже детьми тега <body> являются теги <div> и <ul> (и несколько пустых текстовых узлов):

<html>

<body>

<div>Начало</div>

<ul>

<li>

<b>Информация</b>

</li>

</ul>

</body>

</html>

А потомки <body>– это и прямые дети <div>, <ul> и вложенные в них: <li> (потомок <ul>) и <b> (потомок <li>) – в общем, все элементы поддерева.

Коллекция childNodes содержит список всех детей, включая текстовые узлы. Пример ниже последовательно выведет детей document.body:

<html>

<body>

<div>Начало</div>

<ul>

<li>Информация</li>

</ul>

<div>Конец</div>

<script>

for (let i = 0; i < document.body.childNodes.length; i++) {

alert( document.body.childNodes[i] ); // Text, DIV, Text, UL, ..., SCRIPT

}

</script>

...какой-то HTML-код...

</body>

</html>

Обратим внимание на одну особенность. Если запустить пример выше, то последним будет выведен элемент <script>. На самом деле, в документе есть ещё «какой-то HTML-код», но на момент выполнения скрипта браузер ещё до него не дошёл, поэтому скрипт не видит его.

Свойства firstChild и lastChild обеспечивают быстрый доступ к первому и последнему дочернему элементу. Они, по сути, являются всего лишь сокращениями. Если у тега есть дочерние узлы, условие ниже всегда верно:

elem.childNodes[0] === elem.firstChild

elem.childNodes[elem.childNodes.length - 1] === elem.lastChild

Для проверки наличия дочерних узлов существует также специальная функция elem.hasChildNodes().

[**DOM-коллекции**](https://learn.javascript.ru/dom-navigation#dom-kollektsii)

childNodes это не массив, а коллекция – особый перебираемый объект-псевдомассив. И есть два важных следствия из этого:

1. Для перебора коллекции можно использовать for..of:

for (let node of document.body.childNodes) {

alert(node); // покажет все узлы из коллекции

}

Это работает, потому что коллекция является перебираемым объектом (есть требуемый для этого метод Symbol.iterator).

1. Методы массивов не будут работать, потому что коллекция – это не массив:

alert(document.body.childNodes.filter); // undefined (у коллекции нет метода filter)

Первый пункт – это хорошо для нас. Второй – бывает неудобен, но можно пережить. Если хочется использовать именно методы массива, то можно создать настоящий массив из коллекции, используя Array.from:

alert( Array.from(document.body.childNodes).filter ); // сделали массив

DOM-коллекции и все навигационные свойства, рассматриваемые в этой теме, доступны только для чтения. Нельзя заменить один дочерний узел на другой, просто написав childNodes[i] = .... Для изменения DOM требуются другие методы. Изучим их позже.

Почти все DOM-коллекции, за небольшим исключением, отражают текущее состояние DOM. Если сохранить ссылку на elem.childNodes и добавить/удалить узлы в DOM, то они появятся в сохраненной коллекции автоматически.

Коллекции перебираются циклом for..of. Не стоит использовать для этого цикл for..in. Цикл for..in перебирает все перечисляемые свойства. А у коллекций есть некоторые «лишние», редко используемые свойства, которые обычно не нужны:

<body>

<script>

// выводит 0, 1, length, item, values и другие свойства.

for (let prop in document.body.childNodes) alert(prop);

</script>

</body>

[**Соседи и родитель**](https://learn.javascript.ru/dom-navigation#sosedi-i-roditel)

Соседи – это узлы, у которых один и тот же родитель. Например, здесь <head> и <body> соседи:

<html>

<head>...</head><body>...</body>

</html>

Говорят, что <body> – «следующий» или «правый» сосед <head>. Также можно сказать, что <head> «предыдущий» или «левый» сосед <body>. Следующий узел того же родителя (следующий сосед) – в свойстве nextSibling, а предыдущий – в previousSibling. Родитель доступен через parentNode. Например:

// родителем <body> является <html>

alert( document.body.parentNode === document.documentElement ); // выведет true

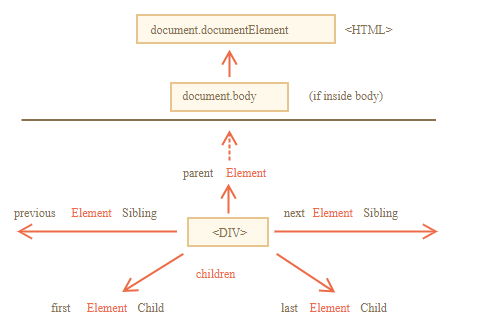
// после <head> идет <body>

alert( document.head.nextSibling ); // HTMLBodyElement

// перед <body> находится <head>

alert( document.body.previousSibling ); // HTMLHeadElement

Навигационные свойства, описанные выше, относятся ко всем узлам в документе. В частности, в childNodes находятся и текстовые узлы и узлы-элементы и узлы-комментарии, если они есть. Но для большинства задач текстовые узлы и узлы-комментарии не нужны. Как правило, надо манипулировать узлами-элементами, которые представляют собой теги и формируют структуру страницы. Поэтому рассмотрим дополнительный набор ссылок, которые учитывают только узлы-элементы:



Эти ссылки похожи на те, что рассматривались раньше, только в ряде мест стоит слово Element:

* children – коллекция детей, которые являются элементами.
* firstElementChild, lastElementChild – первый и последний дочерний элемент.
* previousElementSibling, nextElementSibling – соседи-элементы.
* parentElement – родитель-элемент.

Свойство parentElement возвращает родитель-элемент, а parentNode возвращает «любого родителя». Обычно эти свойства одинаковы: они оба получают родителя. За исключением document.documentElement:

alert( document.documentElement.parentNode ); // выведет document

alert( document.documentElement.parentElement ); // выведет null

Причина в том, что родителем корневого узла document.documentElement (<html>) является document. Но document – это не узел-элемент, так что parentNode вернет его, а parentElement нет. Эта деталь может быть полезна, если надо пройти вверх по цепочке родителей от произвольного элемента elem к <html>, но не до document:

while(elem = elem.parentElement) { // идти наверх до <html>

alert( elem );

}

Изменим один из примеров выше: заменим childNodes на children. Теперь цикл выводит только элементы:

<html>

<body>

<div>Начало</div>

<ul>

<li>Информация</li>

</ul>

<div>Конец</div>

<script>

for (let elem of document.body.children) {

alert(elem); // DIV, UL, DIV, SCRIPT

}

</script>

...

</body>

</html>

До сих пор рассматривались основные навигационные ссылки. Некоторые типы DOM-элементов предоставляют для удобства дополнительные свойства, специфичные для их типа. Таблицы – отличный пример таких элементов. Элемент <table>, в дополнение к свойствам, о которых речь шла выше, поддерживает следующие:

* table.rows – коллекция строк <tr> таблицы;
* table.caption/tHead/tFoot – ссылки на элементы таблицы <caption>, <thead>, <tfoot>;
* table.tBodies – коллекция элементов таблицы <tbody> (по спецификации их может быть больше одного);
* <thead>, <tfoot>, <tbody> предоставляют свойство rows: tbody.rows – коллекция строк <tr> секции;
* <tr>:
* tr.cells – коллекция <td> и <th> ячеек, находящихся внутри строки <tr>;
* tr.sectionRowIndex – номер строки <tr> в текущей секции <thead>/<tbody>/<tfoot>;
* tr.rowIndex – номер строки <tr> в таблице (включая все строки таблицы);
* <td> and <th>: td.cellIndex – номер ячейки в строке <tr>.

Пример использования:

<table id="table">

<tr>

<td>один</td><td>два</td>

</tr>

<tr>

<td>три</td><td>четыре</td>

</tr>

</table>

<script>

// выводит содержимое первой строки, второй ячейки

alert( table.rows[0].cells[1].innerHTML ) // "два"

</script>

Существуют также дополнительные навигационные ссылки для HTML-форм. Они будут рассматриваться позже, при работе с формами.

1. **Методы поиска DOM-элементов.**

Свойства навигации по DOM удобны, когда элементы расположены рядом. Чтобы получить произвольный элемент страницы, можно использовать дополнительные методы поиска.

[**document.getElementById или просто id**](https://learn.javascript.ru/searching-elements-dom#document-getelementbyid-ili-prosto-id)

Когда у элемента есть атрибут id, значение атрибута используется в качестве имени глобальной переменной. С её помощью можно обратиться к элементу напрямую (если в имени переменной используется дефис `-`, то она доступна через квадратные скобки [...]):

<div id="elem">

<div id="elem-content">Элемент</div>

</div>

<script>

alert(elem); // DOM-элемент с id="elem"

alert(window.elem); // доступ к глобальной переменной тоже работает

alert(window['elem-content']);

</script>

Это поведение соответствует [стандарту](http://www.whatwg.org/specs/web-apps/current-work/#dom-window-nameditem), но поддерживается в основном для совместимости. Браузер пытается помочь, смешивая пространства имён JS и DOM. Это подходит для простого кода, но возможны конфликты. Если осуществляется работа с JS-кодом, не видя HTML, не очевидно, откуда возьмётся переменная. Если объявить переменную с тем же именем, она будет иметь приоритет:

<div id="elem"></div>

<script>

let elem = 5;

alert(elem); // 5

</script>

Лучшая альтернатива – использовать специальный метод document.getElementById(id). Например:

<div id="elem">

<div id="elem-content">Элемент</div>

</div>

<script>

let elem = document.getElementById('elem');

elem.style.background = 'red';

</script>

Далее в примерах часто будет использоваться прямое обращение через id, но это только для краткости. В реальных проектах предпочтителен метод document.getElementById.

Значение id должно быть уникальным. В документе может быть только один элемент с данным id. Если в документе есть несколько элементов с одинаковым значением id, то поведение методов поиска непредсказуемо. Браузер может вернуть любой из них случайным образом. Поэтому придерживайтесь правила сохранения уникальности id.

Метод getElementById можно вызвать только для объекта document. Он осуществляет поиск по id по всему документу.

**Методы** [**querySelectorAll**](https://learn.javascript.ru/searching-elements-dom#querySelectorAll) **и** [**querySelector**](https://learn.javascript.ru/searching-elements-dom#querySelector)

Самый универсальный метод поиска – это elem.querySelectorAll(css), он возвращает все элементы внутри elem, удовлетворяющие данному CSS-селектору. Следующий запрос получает все элементы <li>, которые являются последними потомками в <ul>:

<ul>

<li>Этот</li>

<li>тест</li>

</ul>

<ul>

<li>полностью</li>

<li>пройден</li>

</ul>

<script>

let elements = document.querySelectorAll('ul > li:last-child');

for (let elem of elements) {

alert(elem.innerHTML); // "тест", "пройден"

}

</script>

С этим методом можно использовать любой CSS-селектор. Псевдоклассы в CSS-селекторе, в частности :hover и :active, также поддерживаются. Например, document.querySelectorAll(':hover') вернёт коллекцию (в порядке вложенности: от внешнего к внутреннему) из текущих элементов под курсором мыши.

Метод elem.querySelector(css) возвращает первый элемент, соответствующий данному CSS-селектору. Иначе говоря, результат такой же, как при вызове elem.querySelectorAll(css)[0], но он сначала найдёт все элементы, а потом возьмёт первый, в то время как elem.querySelector найдёт только первый и остановится. Это быстрее, кроме того, его короче писать.

**Метод** [**matches**](https://learn.javascript.ru/searching-elements-dom#matches)

Предыдущие методы искали по DOM. Метод [elem.matches(css)](http://dom.spec.whatwg.org/#dom-element-matches) ничего не ищет, а проверяет, удовлетворяет ли elem CSS-селектору, и возвращает true или false. Этот метод удобен, когда надо перебрать элементы (например, в массиве или в чём-то подобном) и выбрать те из них, которые интересуют. Например:

<a href="http://example.com/file.zip">...</a>

<a href="http://ya.ru">...</a>

<script>

// может быть любая коллекция вместо document.body.children

for (let elem of document.body.children) {

if (elem.matches('a[href$="zip"]')) {

alert("Ссылка на архив: " + elem.href );

}

}

</script>

**Метод** [**closest**](https://learn.javascript.ru/searching-elements-dom#closest)

Предки элемента – родитель, родитель родителя, его родитель и так далее. Вместе они образуют цепочку иерархии от элемента до вершины. Метод elem.closest(css) ищет ближайшего предка, который соответствует CSS-селектору. Сам элемент также включается в поиск. Другими словами, метод closest поднимается вверх от элемента и проверяет каждого из родителей. Если он соответствует селектору, поиск прекращается. Метод возвращает либо предка, либо null, если такой элемент не найден. Например:

<h1>Содержание</h1>

<div class="contents">

<ul class="book">

<li class="chapter">Глава 1</li>

<li class="chapter">Глава 2</li>

</ul>

</div>

<script>

let chapter = document.querySelector('.chapter'); // LI

alert(chapter.closest('.book')); // UL

alert(chapter.closest('.contents')); // DIV

alert(chapter.closest('h1')); // null (потому что h1 - не предок)

</script>

**Методы** [**getElementsBy\***](https://learn.javascript.ru/searching-elements-dom#getelementsby)

Существуют также другие методы поиска элементов по тегу, классу и так далее. На данный момент, они скорее исторические, так как querySelector более чем эффективен. Здесь рассмотрим их для полноты картины, также можно встретить их в старом коде.

* elem.getElementsByTagName(tag) ищет элементы с данным тегом и возвращает их коллекцию. Передав "\*" вместо тега, можно получить всех потомков.
* elem.getElementsByClassName(className) возвращает элементы, которые имеют данный CSS-класс.
* document.getElementsByName(name) возвращает элементы с заданным атрибутом name. Очень редко используется.

Например, получить все элементы div в документе:

let divs = document.getElementsByTagName('div');

Найдём все input в таблице:

<table id="table">

<tr>

<td>Ваш возраст:</td>

<td>

<label>

<input type="radio" name="age" value="young" checked> младше 18

</label>

<label>

<input type="radio" name="age" value="mature"> от 18 до 50

</label>

<label>

<input type="radio" name="age" value="senior"> старше 60

</label>

</td>

</tr>

</table>

<script>

let inputs = table.getElementsByTagName('input');

for (let input of inputs) {

alert( input.value + ': ' + input.checked );

}

</script>

Одна из самых частых ошибок – это забыть про букву "s". То есть пробовать вызывать метод getElementByTagName вместо getElement**s**ByTagName. Буква "s" отсутствует в названии метода getElementById, так как в данном случае возвращает один элемент. Но getElementsByTagName вернёт список элементов, поэтому "s" обязательна.

Метод возвращает коллекцию, а не элемент. Поэтому, другая распространённая ошибка – написать:

// не работает

document.getElementsByTagName('input').value = 5;

Попытка присвоить значение коллекции, а не элементам внутри неё, не сработает. Нужно перебрать коллекцию в цикле или получить элемент по номеру и уже ему присваивать значение, например, так:

// работает (если есть input)

document.getElementsByTagName('input')[0].value = 5;

Ищем элементы с классом .article:

<form name="my-form">

<div class="article">Article</div>

<div class="long article">Long article</div>

</form>

<script>

// поиск по имени атрибута

let form = document.getElementsByName('my-form')[0];

// поиск по классу внутри form

let articles = form.getElementsByClassName('article');

alert(articles.length); // 2, найдены два элемента с классом article

</script>

Все методы "getElementsBy\*" возвращают живую коллекцию. Такие коллекции всегда отражают текущее состояние документа и автоматически обновляются при его изменении. В приведённом ниже примере есть два скрипта.

1. Первый создаёт ссылку на коллекцию <div>. На этот момент её длина равна 1.
2. Второй скрипт запускается после того, как браузер встречает ещё один <div>, теперь её длина – 2.

<div>First div</div>

<script>

let divs = document.getElementsByTagName('div');

alert(divs.length); // 1

</script>

<div>Second div</div>

<script>

alert(divs.length); // 2

</script>

Напротив, querySelectorAll возвращает статическую коллекцию. Это похоже на фиксированный массив элементов. Если использовать его в примере выше, то оба скрипта вернут длину коллекции, равную 1:

<div>First div</div>

<script>

let divs = document.querySelectorAll('div');

alert(divs.length); // 1

</script>

<div>Second div</div>

<script>

alert(divs.length); // 1

</script>

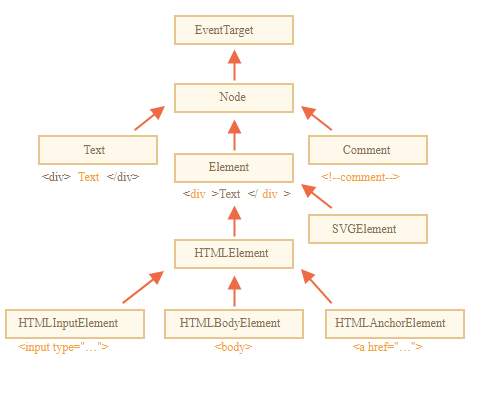
Теперь видна разница. Длина статической коллекции не изменилась после появления нового div в документе.

1. **Свойства узлов: тип, тег и содержимое.**

Разберём подробнее, что собой представляют DOM-узлы и изучим их основные свойства.

[**Классы DOM-узлов**](https://learn.javascript.ru/basic-dom-node-properties#klassy-dom-uzlov)

У разных DOM-узлов могут быть разные свойства. Например, у узла, соответствующего тегу <a>, есть свойства, связанные со ссылками, а у соответствующего тегу <input> – свойства, связанные с полем ввода и т.д. Текстовые узлы отличаются от узлов-элементов. Но у них есть общие свойства и методы, потому что все классы DOM-узлов образуют единую иерархию. Каждый DOM-узел принадлежит соответствующему встроенному классу. Корнем иерархии является [EventTarget](https://dom.spec.whatwg.org/#eventtarget), от него наследует [Node](http://dom.spec.whatwg.org/#interface-node) и остальные DOM-узлы. На рисунке ниже изображены основные классы:



Существуют следующие классы:

* [EventTarget](https://dom.spec.whatwg.org/#eventtarget) – это корневой «абстрактный» класс. Объекты этого класса никогда не создаются. Он служит основой, благодаря которой все DOM-узлы поддерживают так называемые «события», о которых рассмотрим позже.
* [Node](http://dom.spec.whatwg.org/#interface-node) – также является «абстрактным» классом, и служит основой для DOM-узлов. Он обеспечивает базовую функциональность: parentNode, nextSibling, childNodes и т.д. (это геттеры). Объекты класса Node никогда не создаются. Но есть определенные классы узлов, которые наследуют от него: Text – для текстовых узлов, Element – для узлов-элементов и более экзотический Comment – для узлов-комментариев.
* [Element](http://dom.spec.whatwg.org/#interface-element) – это базовый класс для DOM-элементов. Он обеспечивает навигацию на уровне элементов: nextElementSibling, children и методы поиска: getElementsByTagName, querySelector. Браузер поддерживает не только HTML, но также XML и SVG. Класс Element служит базой для следующих классов: SVGElement, XMLElement и HTMLElement.
* [HTMLElement](https://html.spec.whatwg.org/multipage/dom.html#htmlelement) – является базовым классом для всех остальных HTML-элементов. От него наследуют конкретные элементы:
* [HTMLInputElement](https://html.spec.whatwg.org/multipage/forms.html#htmlinputelement) – класс для тега <input>,
* [HTMLBodyElement](https://html.spec.whatwg.org/multipage/semantics.html#htmlbodyelement) – класс для тега <body>,
* [HTMLAnchorElement](https://html.spec.whatwg.org/multipage/semantics.html#htmlanchorelement) – класс для тега <a>,
* …и т.д, каждому тегу соответствует свой класс, который предоставляет определенные свойства и методы.

Таким образом, полный набор свойств и методов данного узла собирается в результате наследования. Рассмотрим DOM-объект для тега <input>. Он принадлежит классу [HTMLInputElement](https://html.spec.whatwg.org/multipage/forms.html#htmlinputelement). Он получает свойства и методы из (в порядке наследования):

* HTMLInputElement – этот класс предоставляет специфичные для элементов формы свойства;
* HTMLElement – предоставляет общие для HTML-элементов методы (и геттеры/сеттеры);
* Element – предоставляет типовые методы элемента;
* Node – предоставляет общие свойства DOM-узлов;
* EventTarget – обеспечивает поддержку событий (поговорим о них дальше);
* Object – доступны методы «обычного объекта», такие как hasOwnProperty.

Для того, чтобы узнать имя класса DOM-узла, вспомним, что обычно у объекта есть свойство constructor. Оно ссылается на конструктор класса, и в свойстве constructor.name содержится его имя:

alert( document.body.constructor.name ); // HTMLBodyElement

Также можно просто привести его к строке:

alert( document.body ); // [object HTMLBodyElement]

Проверить наследование можно также при помощи instanceof:

alert( document.body instanceof HTMLBodyElement ); // true

alert( document.body instanceof HTMLElement ); // true

alert( document.body instanceof Element ); // true

alert( document.body instanceof Node ); // true

alert( document.body instanceof EventTarget ); // true

Как видно, DOM-узлы – это обычные JavaScript объекты. Для наследования они используют классы, основанные на прототипах. В этом легко убедиться, если вывести в консоли браузера любой элемент через console.dir(elem). Или даже напрямую обратиться к методам, которые хранятся в HTMLElement.prototype, Element.prototype и т.д.

**console.dir(elem) и console.log(elem)**

Большинство браузеров поддерживают в инструментах разработчика две команды: console.log и console.dir. Они выводят свои аргументы в консоль. Для JavaScript-объектов эти команды обычно выводят одно и то же. Но для DOM-элементов они работают по-разному: console.log(elem) выводит элемент в виде DOM-дерева; console.dir(elem) выводит элемент в виде DOM-объекта, что удобно для анализа его свойств.

Чтобы узнать тип узла, можно использовать метод instanceof и другие способы проверить класс, но иногда nodeType проще использовать. Нельзя изменить значение nodeType, только прочитать его.

[**Свойства nodeName и tagName**](https://learn.javascript.ru/basic-dom-node-properties#teg-nodename-i-tagname)

Получив DOM-узел, можно узнать имя его тега из свойств nodeName и tagName. Например:

alert( document.body.nodeName ); // BODY

alert( document.body.tagName ); // BODY

Разница между tagName и nodeName заключается в следующем:

* свойство tagName есть только у элементов Element;
* свойство nodeName определено для любых узлов Node:
* для элементов оно равно tagName;
* для остальных типов узлов (текст, комментарий и т.д.) оно содержит строку с типом узла.

Другими словами, свойство tagName есть только у узлов-элементов (поскольку они происходят от класса Element), а nodeName может что-то сказать о других типах узлов. Например, сравним tagName и nodeName на примере объекта document и узла-комментария:

<body><!-- комментарий -->

<script>

// для комментария

alert( document.body.firstChild.tagName ); // undefined (не элемент)

alert( document.body.firstChild.nodeName ); // #comment

// for document

alert( document.tagName ); // undefined (не элемент)

alert( document.nodeName ); // #document

</script>

</body>

Если нужны только элементами, то можно использовать tagName или nodeName, нет разницы.

В браузере существуют два режима обработки документа: HTML и XML. HTML-режим обычно используется для веб-страниц. XML-режим включается, если браузер получает XML-документ с заголовком: Content-Type: application/xml+xhtml.

В HTML-режиме значения tagName/nodeName всегда записаны в верхнем регистре. Будет выведено BODY вне зависимости от того, как записан тег в HTML <body> или <BoDy>.

В XML-режиме регистр сохраняется «как есть». В настоящее время XML-режим применяется редко.

**Свойство**[**innerHTML**](https://w3c.github.io/DOM-Parsing/#widl-Element-innerHTML)

Свойство [innerHTML](https://w3c.github.io/DOM-Parsing/#widl-Element-innerHTML) позволяет получить HTML-содержимое элемента в виде строки. Также можно изменять его. Это один из самых мощных способов менять содержимое на странице. Пример ниже показывает содержимое document.body, а затем полностью заменяет его:

<body>

<p>Параграф</p>

<div>DIV</div>

<script>

alert( document.body.innerHTML ); // читаем текущее содержимое

document.body.innerHTML = 'Новый BODY!'; // заменяем содержимое

</script>

</body>

Можно вставить некорректный HTML, браузер исправит ошибки:

<body>

<script>

document.body.innerHTML = '<b>тест'; // не закрыт тег

alert( document.body.innerHTML ); // <b>тест</b> (исправлено)

</script>

</body>

Если innerHTML вставляет в документ тег <script> – он становится частью HTML, но не запускается.

Можно добавить HTML к элементу, используя elem.innerHTML+="ещё html":

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. Технически эти две строки делают одно и то же:

elem.innerHTML += "...";

// это более короткая запись для:

elem.innerHTML = elem.innerHTML + "..."

Другими словами, innerHTML+= делает следующее:

1. Старое содержимое удаляется.
2. На его место становится новое значение innerHTML (с добавленной строкой).

Так как содержимое «обнуляется» и переписывается заново, все изображения и другие ресурсы будут перезагружены. В примере chatDiv выше строка chatDiv.innerHTML+="Как дела?" заново создаёт содержимое HTML и перезагружает smile.gif. Если в chatDiv много текста и изображений, то эта перезагрузка будет очень заметна.

Есть и другие побочные эффекты. Например, если существующий текст выделен мышкой, то при переписывании innerHTML большинство браузеров снимут выделение. А если это поле ввода <input> с текстом, введенным пользователем, то текст будет удалён.

**Свойство outerHTML**

Свойство outerHTML содержит HTML элемента целиком. Это как innerHTML плюс сам элемент. Рассмотрим пример:

<div id="elem">Привет <b>Мир</b></div>

<script>

alert(elem.outerHTML); // <div id="elem">Привет <b>Мир</b></div>

</script>

Будьте осторожны: в отличие от innerHTML, запись в outerHTML не изменяет элемент. Вместо этого элемент заменяется целиком во внешнем контексте. Рассмотрим пример:

<div>Привет, мир!</div>

<script>

let div = document.querySelector('div');

div.outerHTML = '<p>Новый элемент</p>'; // (\*)

alert(div.outerHTML); // <div>Привет, мир!</div> (\*\*)

</script>

В строке (\*) div заменен  на <p>Новый элемент</p>. Во внешнем документе располагается новое содержимое вместо <div>. Но, как видно в строке (\*\*), старая переменная div осталась прежней. Это потому, что использование outerHTML не изменяет DOM-элемент, а удаляет его из внешнего контекста и вставляет вместо него новый HTML-код. То есть, при div.outerHTML=... произошло следующее:

* div был удалён из документа;
* вместо него был вставлен другой HTML <p>A new element</p>;
* в div осталось старое значение. Новый HTML не схранён ни в какой переменной.

Здесь легко сделать ошибку: заменить div.outerHTML, а потом продолжить работать с div, как будто там новое содержимое. Но это не так. Подобное верно для innerHTML, но не для outerHTML.

Можно писать в elem.outerHTML, но это не меняет элемент, в который пишем. Вместо этого создается новый HTML на его месте. Можно получить ссылки на новые элементы, обратившись к DOM.

**Свойства nodeValue и data**

Свойство innerHTML есть только у узлов-элементов. У других типов узлов, в частности, у текстовых, есть свои аналоги: свойства nodeValue и data. Эти свойства очень похожи при использовании, есть лишь небольшие различия в спецификации. Будем использовать data, потому что оно короче. Прочитаем содержимое текстового узла и комментария:

<body>

Привет

<!-- Комментарий -->

<script>

let text = document.body.firstChild;

alert(text.data); // Привет

let comment = text.nextSibling;

alert(comment.data); // Комментарий

</script>

</body>

Иногда комментарии используют для вставки информации и инструкций шаблонизатора в HTML, как в примере ниже:

<!-- if isAdmin -->

<div>Добро пожаловать, Admin!</div>

<!-- /if -->

Затем JavaScript может прочитать это из свойства data и обработать инструкции.

**Свойство textContent**

Свойство textContent предоставляет доступ к тексту внутри элемента за вычетом всех <тегов>. Например:

<div id="news">

<h1>Срочно в номер!</h1>

<p>Марсиане атаковали человечество!</p>

</div>

<script>

// Срочно в номер! Марсиане атаковали человечество!

alert(news.textContent);

</script>

Возвращается только текст, как если бы все <теги> были вырезаны, но текст в них остался. На практике редко появляется необходимость читать текст таким образом. Намного полезнее возможность записывать текст в свойство textContent, т.к. оно позволяет писать текст «безопасным способом».

Допустим есть произвольная строка, введённая пользователем, и надо показать её. С innerHTML вставка происходит «как HTML», со всеми HTML-тегами. С textContent вставка получается «как текст», все символы трактуются буквально. Сравним два тега div:

<div id="elem1"></div>

<div id="elem2"></div>

<script>

let name = prompt("Введите ваше имя?", "<b>Винни-пух!</b>");

elem1.innerHTML = name;

elem2.textContent = name;

</script>

1. В первый <div> имя приходит «как HTML»: все теги стали именно тегами, поэтому отображается имя, выделенное жирным шрифтом.
2. Во второй <div> имя приходит «как текст», поэтому отображается <b>Винни-пух!</b>.

В большинстве случаев надо получить от пользователя текст и чтобы он интерпретировался как текст. Не надо, чтобы на сайте появлялся произвольный HTML-код. Присваивание через textContent – один из способов от этого защититься.

**[Свойство «hidden»](https://learn.javascript.ru/basic-dom-node-properties" \l "svoystvo-hidden)**

Атрибут и DOM-свойство «hidden» указывает на то, виден ли элемент или нет. Можно использовать его в HTML или назначать при помощи JavaScript, как в примере ниже:

<div>Оба тега DIV внизу невидимы</div>

<div hidden>С атрибутом "hidden"</div>

<div id="elem">С назначенным JavaScript свойством "hidden"</div>

<script>

elem.hidden = true;

</script>

Технически, hidden работает так же, как style="display:none". Но его применение проще. Мигающий элемент:

<div id="elem">Мигающий элемент</div>

<script>

setInterval(() => elem.hidden = !elem.hidden, 1000);

</script>

[**Другие свойства**](https://learn.javascript.ru/basic-dom-node-properties#drugie-svoystva)

У DOM-элементов есть дополнительные свойства, в частности, зависящие от класса:

* value – значение для <input>, <select> и <textarea> (HTMLInputElement, HTMLSelectElement и др.);
* href – адрес ссылки «href» для <a href="..."> (HTMLAnchorElement);
* id – значение атрибута «id» для всех элементов (HTMLElement) и многие другие.

Например:

<input type="text" id="elem" value="значение">

<script>

alert(elem.type); // "text"

alert(elem.id); // "elem"

alert(elem.value); // значение

</script>

Большинство стандартных HTML-атрибутов имеют соответствующее DOM-свойство, и можно получить к нему доступ. Если надо узнать полный список поддерживаемых свойств для данного класса, можно найти их в спецификации. Например, класс HTMLInputElement описывается здесь: <https://html.spec.whatwg.org/#htmlinputelement>. Также можно вывести элемент в консоль, используя console.dir(elem), и прочитать все свойства. Или исследовать «свойства DOM» во вкладке Elements браузерных инструментов разработчика.

1. **Атрибуты и DOM-свойства.**

Когда браузер загружает страницу, он «читает» («парсит») HTML и генерирует из него DOM-объекты. Для узлов-элементов большинство стандартных HTML-атрибутов автоматически становятся свойствами DOM-объектов. Например, для такого тега <body id="page"> у DOM-объекта будет такое свойство body.id="page".

Но преобразование атрибута в свойство происходит не один-в-один.

[**DOM-свойства**](https://learn.javascript.ru/dom-attributes-and-properties#dom-svoystva)

Ранее рассматривались встроенные DOM-свойства. Но можно добавить своё собственное свойство. DOM-узлы – это обычные объекты JavaScript. Можно их изменять. Например, создадим новое свойство для document.body:

document.body.myData = {

name: 'Caesar',

title: 'Imperator'

};

alert(document.body.myData.title); // Imperator

Можно добавить и метод:

document.body.sayTagName = function() {

alert(this.tagName);

};

document.body.sayTagName(); // BODY (this = document.body)

Также можно изменять встроенные прототипы, такие как Element.prototype и добавлять новые методы ко всем элементам:

Element.prototype.sayHi = function() {

alert(`Hello, I'm ${this.tagName}`);

};

document.documentElement.sayHi(); // Hello, I'm HTML

document.body.sayHi(); // Hello, I'm BODY

Итак, DOM-свойства и методы ведут себя так же, как и обычные объекты JavaScript: им можно присвоить любое значение; они регистрозависимы (нужно писать elem.nodeType, не elem.NoDeTyPe).

[**HTML-атрибуты**](https://learn.javascript.ru/dom-attributes-and-properties#html-atributy)

В HTML у тегов могут быть атрибуты. Когда браузер парсит HTML, чтобы создать DOM-объекты для тегов, он распознаёт стандартные атрибуты и создаёт DOM-свойства для них. Таким образом, когда у элемента есть id или другой стандартный атрибут, создаётся соответствующее свойство. Но этого не происходит, если атрибут нестандартный. Например:

<body id="test" something="non-standard">

<script>

alert(document.body.id); // test

alert(document.body.something); // undefined

</script>

</body>

Пожалуйста, учтите, что стандартный атрибут для одного тега может быть нестандартным для другого. Например, атрибут "type" является стандартным для элемента <input> ([HTMLInputElement](https://html.spec.whatwg.org/#htmlinputelement)), но не является стандартным для <body>([HTMLBodyElement](https://html.spec.whatwg.org/#htmlbodyelement)). Стандартные атрибуты описаны в спецификации для соответствующего класса элемента. Можно увидеть это на примере ниже:

<body id="body" type="...">

<input id="input" type="text">

<script>

alert(input.type); // text

alert(body.type); // undefined

</script>

</body>

Таким образом, для нестандартных атрибутов не будет соответствующих DOM-свойств. Все атрибуты доступны с помощью следующих методов:

* elem.hasAttribute(name) – проверяет наличие атрибута;
* elem.getAttribute(name) – получает значение атрибута;
* elem.setAttribute(name, value) – устанавливает значение атрибута;
* elem.removeAttribute(name) – удаляет атрибут.

Этим методы работают именно с тем, что написано в HTML.

Кроме этого, получить все атрибуты элемента можно с помощью свойства elem.attributes: коллекция объектов, которая принадлежит ко встроенному классу [Attr](https://dom.spec.whatwg.org/#attr) со свойствами name и value. Вот демонстрация чтения нестандартного свойства:

<body something="non-standard">

<script>

alert(document.body.getAttribute('something')); // non-standard

</script>

</body>

У HTML-атрибутов есть следующие особенности: их имена регистронезависимы (id то же самое, что и ID); их значения всегда являются строками. Расширенная демонстрация работы с атрибутами:

<body>

<div id="elem" about="Elephant"></div>

<script>

alert( elem.getAttribute('About') ); // (1) 'Elephant', чтение

elem.setAttribute('Test', 123); // (2), запись

alert( elem.outerHTML ); // (3), посмотрим, есть ли атрибут в HTML (да)

for (let attr of elem.attributes) { // (4) весь список

alert( `${attr.name} = ${attr.value}` );

}

</script>

</body>

Обратите внимание:

1. getAttribute('About') – здесь первая буква заглавная, а в HTML – строчная. Но это не важно: имена атрибутов регистронезависимы.
2. Можно присвоить что угодно атрибуту, но это станет строкой. Поэтому в этой строчке получаем значение "123".
3. Все атрибуты, в том числе те, которые были установлены, видны в outerHTML.
4. Коллекция attributes является перебираемой. В ней есть все атрибуты элемента (стандартные и нестандартные) в виде объектов со свойствами name и value.

[**Синхронизация между атрибутами и свойствами**](https://learn.javascript.ru/dom-attributes-and-properties#sinhronizatsiya-mezhdu-atributami-i-svoystvami)

Когда стандартный атрибут изменяется, соответствующее свойство автоматически обновляется. Это работает и в обратную сторону (за некоторыми исключениями). В примере ниже id модифицируется как атрибут, и можно увидеть, что свойство также изменено. То же самое работает и в обратную сторону:

<input>

<script>

let input = document.querySelector('input');

// атрибут => свойство

input.setAttribute('id', 'id');

alert(input.id); // id (обновлено)

// свойство => атрибут

input.id = 'newId';

alert(input.getAttribute('id')); // newId (обновлено)

</script>

Но есть и исключения, например, input.value синхронизируется только в одну сторону – атрибут → значение, но не в обратную:

<input>

<script>

let input = document.querySelector('input');

// атрибут => значение

input.setAttribute('value', 'text');

alert(input.value); // text

// свойство => атрибут

input.value = 'newValue';

alert(input.getAttribute('value')); // text

</script>

В примере выше изменение атрибута value обновило свойство, но изменение свойства не повлияло на атрибут. Иногда эта «особенность» может пригодиться, потому что действия пользователя могут приводить к изменениям value, и если после этого надо восстановить «оригинальное» значение из HTML, оно будет в атрибуте.

DOM-свойства не всегда являются строками. Например, свойство input.checked (для чекбоксов) имеет логический тип:

<input id="input" type="checkbox" checked> checkbox

<script>

alert(input.getAttribute('checked')); // значение атрибута: пустая строка

alert(input.checked); // значение свойства: true

</script>

Есть и другие примеры. Атрибут style – строка, но свойство style является объектом:

<div id="div" style="color:red;font-size:120%">Hello</div>

<script>

// строка

alert(div.getAttribute('style')); // color:red;font-size:120%

// объект

alert(div.style); // [object CSSStyleDeclaration]

alert(div.style.color); // red

</script>

Большинство свойств, всё же, строки, но могут отличаться от атрибутов. Например, DOM-свойство href всегда содержит полный URL, даже если атрибут содержит относительный URL или просто #hash. Ниже пример:

<a id="a" href="#hello">link</a>

<script>

// атрибут

alert(a.getAttribute('href')); // #hello

// свойство

alert(a.href ); // полный URL в виде http://site.com/page#hello

</script>

Если же нужно значение href или любого другого атрибута в точности, как оно записано в HTML, можно воспользоваться getAttribute.

[**Нестандартные атрибуты, dataset**](https://learn.javascript.ru/dom-attributes-and-properties#nestandartnye-atributy-dataset)

Нестандартные атрибуты используются для передачи пользовательских данных из HTML в JavaScript, или чтобы «помечать» HTML-элементы для JavaScript. Пример:

<!-- пометить div, чтобы показать здесь поле "name" -->

<div show-info="name"></div>

<!-- а здесь возраст "age" -->

<div show-info="age"></div>

<script>

// код находит элемент с пометкой и показывает запрошенную информацию

let user = {

name: "Pete",

age: 25

};

for(let div of document.querySelectorAll('[show-info]')) {

// вставить соответствующую информацию в поле

let field = div.getAttribute('show-info');

div.innerHTML = user[field]; // сначала Pete в name, потом 25 в age

}

</script>

Также они могут быть использованы, чтобы стилизовать элементы. Например, здесь для состояния заказа используется атрибут order-state:

<style>

/\* стили зависят от пользовательского атрибута "order-state" \*/

.order[order-state="new"] {

color: green;

}

.order[order-state="pending"] {

color: blue;

}

.order[order-state="canceled"] {

color: red;

}

</style>

<div class="order" order-state="new">

A new order.

</div>

<div class="order" order-state="pending">

A pending order.

</div>

<div class="order" order-state="canceled">

A canceled order.

</div>

Атрибут могут быть предпочтительнее таких классов, как .order-state-new, .order-state-pending, order-state-canceled, потому, что атрибутом удобнее управлять. Состояние может быть изменено достаточно просто (не надо удалять старый/добавлять новый класса):

div.setAttribute('order-state', 'canceled');

Но с пользовательскими атрибутами могут возникнуть проблемы, например, если был использован нестандартный атрибут, а позже он появится в стандарте и будет выполнять какую-то функцию.

Чтобы избежать конфликтов, существуют атрибуты вида [data-\*](https://html.spec.whatwg.org/#embedding-custom-non-visible-data-with-the-data-*-attributes). Все атрибуты, начинающиеся с префикса «data-», зарезервированы для использования программистами. Они доступны в свойстве dataset. Например, если у elem есть атрибут "data-about", то обратиться к нему можно как elem.dataset.about. Например:

<body data-about="Elephants">

<script>

alert(document.body.dataset.about); // Elephants

</script>

Атрибуты, состоящие из нескольких слов, к примеру data-order-state, становятся свойствами, записанными с помощью верблюжьей нотации: dataset.orderState. Вот переписанный пример «состояния заказа»:

<style>

.order[data-order-state="new"] {

color: green;

}

.order[data-order-state="pending"] {

color: blue;

}

.order[data-order-state="canceled"] {

color: red;

}

</style>

<div id="order" class="order" data-order-state="new">

A new order.

</div>

<script>

// чтение

alert(order.dataset.orderState); // new

// изменение

order.dataset.orderState = "pending"; // (\*)

</script>

Использование data-\* атрибутов – валидный, безопасный способ передачи пользовательских данных. Можно не только читать, но и изменять data-атрибуты. Тогда CSS обновит представление соответствующим образом: в примере выше последняя строка (\*) меняет цвет на синий.

1. **Добавление и удаление узлов.**

Рассмотрим пример, добавим на страницу сообщение:

<style>

.alert {

padding: 15px;

border: 1px solid #d6e9c6;

border-radius: 4px;

color: #3c763d;

background-color: #dff0d8;

}

</style>

<div class="alert">

<strong>Всем привет!</strong> Вы прочитали важное сообщение.

</div>

Это был пример HTML. Теперь создадим такой же div, используя JavaScript (предполагаем, что стили в HTML или во внешнем CSS-файле).

**[Создание элемента](https://learn.javascript.ru/modifying-document" \l "sozdanie-elementa)**

DOM-узел можно создать двумя методами:

* document.createElement(tag) – создаёт новый элемент с заданным тегом:

let div = document.createElement('div');

* document.createTextNode(text) – создаёт новый текстовый узел с заданным текстом:

let textNode = document.createTextNode('А вот и я');

В рассматриваемом примере нужно создать сообщение – это div с классом alert и HTML в нём:

let div = document.createElement('div');

div.className = "alert";

div.innerHTML = "<strong>Всем привет!</strong> Вы прочитали важное сообщение.";

Элемент создан, но пока он только в переменной. Нельзя пока увидеть его на странице, поскольку он не является частью документа.

[**Методы вставки**](https://learn.javascript.ru/modifying-document#metody-vstavki)

Чтобы div появился на странице, нужно вставить его где-нибудь в document. Например, в document.body. Для этого есть метод append, в нашем случае: document.body.append(div). Вот полный пример:

<style>

.alert {

padding: 15px;

border: 1px solid #d6e9c6;

border-radius: 4px;

color: #3c763d;

background-color: #dff0d8;

}

</style>

<script>

let div = document.createElement('div');

div.className = "alert";

div.innerHTML = "<strong>Всем привет!</strong> Вы прочитали важное сообщение.";

document.body.append(div);

</script>

Метод для различных вариантов вставки:

* node.append(...узлы или строки) – добавляет узлы или строки в конец node;
* node.prepend(...nodes or strings) – вставляет узлы или строки в начало node;
* node.before(...nodes or strings) –- вставляет узлы или строки до node;
* node.after(...nodes or strings) –- вставляет узлы или строки после node;
* node.replaceWith(...nodes or strings) –- заменяет node заданными узлами или строками.

Вот пример использования этих методов, чтобы добавить новые элементы в список и текст до/после него:

<ol id="ol">

<li>0</li>

<li>1</li>

<li>2</li>

</ol>

<script>

ol.before('before'); // вставить строку "before" перед <ol>

ol.after('after'); // вставить строку "after" после <ol>

let liFirst = document.createElement('li');

liFirst.innerHTML = 'prepend';

ol.prepend(liFirst); // вставить liFirst в начало <ol>

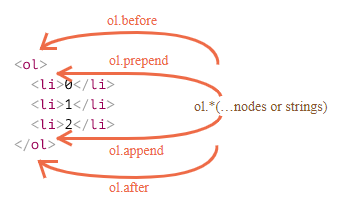
let liLast = document.createElement('li');

liLast.innerHTML = 'append';

ol.append(liLast); // вставить liLast в конец <ol>

</script>

Наглядная иллюстрация того, куда эти методы вставляют:



Итоговый список будет таким:

before

<ol id="ol">

<li>prepend</li>

<li>0</li>

<li>1</li>

<li>2</li>

<li>append</li>

</ol>

after

Эти методы могут вставлять несколько узлов и текстовых фрагментов за один вызов. Например, здесь вставляется строка и элемент:

<div id="div"></div>

<script>

div.before('<p>Привет</p>', document.createElement('hr'));

</script>

Весь текст вставляется как текст. Поэтому финальный HTML будет таким:

&lt;p&gt;Привет&lt;/p&gt;

<hr>

<div id="div"></div>

Другими словами, строки вставляются безопасным способом, как делает это elem.textContent. Поэтому эти методы могут использоваться только для вставки DOM-узлов или текстовых фрагментов.

Если надо вставить HTML именно «как html», со всеми тегами и прочим, как делает это elem.innerHTML, тогда надо использовать другой универсальный метод: elem.insertAdjacentHTML(where, html).

Первый параметр – это специальное слово, указывающее, куда по отношению к elem производить вставку. Значение должно быть одним из следующих:

* "beforebegin" – вставить html непосредственно перед elem,
* "afterbegin" – вставить html в начало elem,
* "beforeend" – вставить html в конец elem,
* "afterend" – вставить html непосредственно после elem.

Второй параметр – это HTML-строка, которая будет вставлена именно «как HTML». Например:

<div id="div"></div>

<script>

div.insertAdjacentHTML('beforebegin', '<p>Привет</p>');

div.insertAdjacentHTML('afterend', '<p>Пока</p>');

</script>

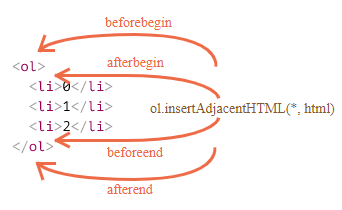
Приведёт к:

<p>Привет</p>

<div id="div"></div>

<p>Пока</p>

Так можно добавлять произвольный HTML на страницу. Варианты вставки:



Есть еще два схожих метода:

* elem.insertAdjacentText(where, text) – такой же синтаксис, но строка text вставляется «как текст», вместо HTML,
* elem.insertAdjacentElement(where, elem) – такой же синтаксис, но вставляет элемент elem.

На практике часто используется только insertAdjacentHTML. Потому что для элементов и текста есть методы append/prepend/before/after – их быстрее написать, и они могут вставлять как узлы, так и текст. Альтернативный вариант показа сообщения:

<style>

.alert {

padding: 15px;

border: 1px solid #d6e9c6;

border-radius: 4px;

color: #3c763d;

background-color: #dff0d8;

}

</style>

<script>

document.body.insertAdjacentHTML("afterbegin", `<div class="alert">

<strong>Всем привет!</strong> Вы прочитали важное сообщение.

</div>`);

</script>

[**Удаление узлов**](https://learn.javascript.ru/modifying-document#udalenie-uzlov)

Для удаления узла есть методы node.remove(). Например, сделаем так, чтобы сообщение удалялось через секунду:

<style>

.alert {

padding: 15px;

border: 1px solid #d6e9c6;

border-radius: 4px;

color: #3c763d;

background-color: #dff0d8;

}

</style>

<script>

let div = document.createElement('div');

div.className = "alert";

div.innerHTML = "<strong>Всем привет!</strong> Вы прочитали важное сообщение.";

document.body.append(div);

setTimeout(() => div.remove(), 1000);

</script>

Если нужно переместить элемент в другое место – нет необходимости удалять его со старого. Все методы вставки автоматически удаляют узлы со старых мест. Например, поменяем местами элементы:

<div id="first">Первый</div>

<div id="second">Второй</div>

<script>

second.after(first); // вставляет #first после #second

</script>

[**Клонирование узлов: cloneNode**](https://learn.javascript.ru/modifying-document#klonirovanie-uzlov-clonenode)

Вызов elem.cloneNode(true) создаёт клон элемента со всеми атрибутами и дочерними элементами. Если вызвать elem.cloneNode(false), тогда клон будет без дочерних элементов. Пример копирования сообщения:

<style>

.alert {

padding: 15px;

border: 1px solid #d6e9c6;

border-radius: 4px;

color: #3c763d;

background-color: #dff0d8;

}

</style>

<div class="alert" id="div">

<strong>Всем привет!</strong> Вы прочитали важное сообщение.

</div>

<script>

let div2 = div.cloneNode(true); // клонировать сообщение

div2.querySelector('strong').innerHTML = 'Всем пока!'; // изменить клонированный элемент

div.after(div2); // показать клонированный элемент после существующего div

</script>

**Узел** [**DocumentFragment**](https://learn.javascript.ru/modifying-document#document-fragment)

DocumentFragment является специальным DOM-узлом, который служит обёрткой для передачи списков узлов. Можно добавить к нему другие узлы, но при вставке он «исчезает», вместо него вставляется его содержимое. Например, getListContent ниже генерирует фрагмент с элементами <li>, которые позже вставляются в <ul>:

<ul id="ul"></ul>

<script>

function getListContent() {

let fragment = new DocumentFragment();

for(let i=1; i<=3; i++) {

let li = document.createElement('li');

li.append(i);

fragment.append(li);

}

return fragment;

}

ul.append(getListContent()); // (\*)

</script>

Обратите внимание, что на последней строке с (\*) добавляется DocumentFragment, но он «исчезает», поэтому структура будет:

<ul>

<li>1</li>

<li>2</li>

<li>3</li>

</ul>

DocumentFragment редко используется. Не смысла добавлять элементы в специальный вид узла, если вместо этого можно вернуть массив узлов. Переписанный пример:

<ul id="ul"></ul>

<script>

function getListContent() {

let result = [];

for(let i=1; i<=3; i++) {

let li = document.createElement('li');

li.append(i);

result.push(li);

}

return result;

}

ul.append(...getListContent());

</script>

DocumentFragment  используется в некоторых других областях, например, для элемента [template](https://learn.javascript.ru/template-element).

1. **Стили.**

Как правило, существует два способа задания стилей для элемента:

1. Создать класс в CSS и использовать его: <div class="...">.
2. Писать стили непосредственно в атрибуте style: <div style="...">.

JavaScript может менять и классы и свойство style. Классы – всегда предпочтительный вариант по сравнению со style. Свойством style стоит манипулировать только в том случае, если классы «не могут справиться». Например, использование style является приемлемым, если вычисляются координаты элемента динамически и хотим установить их из JavaScript:

let top = /\* расчёты \*/;

let left = /\* расчёты \*/;

elem.style.left = left;

elem.style.top = top;

В других случаях, например, чтобы сделать текст красным, добавить значок фона – надо описать это в CSS и добавить класс (JavaScript может это сделать). Это более гибкое и легкое в поддержке решение.

[**className и classList**](https://learn.javascript.ru/styles-and-classes#classname-i-classlist)

Изменение класса является одним из наиболее часто используемых действий в скриптах. Свойство "className": elem.className соответствует атрибуту "class". Например:

<body class="main page">

<script>

alert(document.body.className); // main page

</script>

</body>

Если присвоить что-то elem.className, то это заменяет всю строку с классами. Иногда это то, что нужно, но часто надо добавить/удалить один класс. Для этого есть другое свойство: elem.classList.

elem.classList – это специальный объект с методами для добавления/удаления одного класса. Например:

<body class="main page">

<script>

document.body.classList.add('article');

alert(document.body.className); // main page article

</script>

</body>

Так что можно работать как со строкой полного класса, используя className, так и с отдельными классами, используя classList.

Методы classList:

* elem.classList.add/remove("class") – добавить/удалить класс.
* elem.classList.toggle("class") – добавить класс, если его нет, иначе удалить.
* elem.classList.contains("class") – проверка наличия класса, возвращает true/false.

Кроме того, classList является перебираемым, поэтому можно перечислить все классы при помощи for..of:

<body class="main page">

<script>

for (let name of document.body.classList) {

alert(name); // main, затем page

}

</script>

</body>

**Свойство** [**Element style**](https://learn.javascript.ru/styles-and-classes#element-style)

Свойство elem.style – это объект, который соответствует тому, что написано в атрибуте "style". Установка стиля elem.style.width="100px" работает так же, как наличие в атрибуте style строки width:100px. Для свойства из нескольких слов используется camelCase:

background-color => elem.style.backgroundColor

z-index => elem.style.zIndex

border-left-width => elem.style.borderLeftWidth

Например:

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

Стили с браузерным префиксом, например, -moz-border-radius, -webkit-border-radius преобразуются по тому же принципу: дефис означает прописную букву. Например:

button.style.MozBorderRadius = '5px';

button.style.WebkitBorderRadius = '5px';

Иногда нужно добавить свойство стиля, а потом, позже, убрать его. Например, чтобы скрыть элемент, можно задать elem.style.display = "none". Затем можно удалить свойство style.display, чтобы вернуться к первоначальному состоянию. Вместо delete elem.style.display  надо присвоить ему пустую строку: elem.style.display = "".

// <body> "мигнёт"

document.body.style.display = "none"; // скрыть

setTimeout(() => document.body.style.display = "", 1000); // возврат к нормальному состоянию

Если установить в style.display пустую строку, то браузер применит CSS-классы и встроенные стили, как если бы такого свойства style.display вообще не было.

Обычно используется style.\* для присвоения индивидуальных свойств стиля. Нельзя установить список стилей как, например, div.style="color: red; width: 100px", потому что div.style – это объект, и он доступен только для чтения. Для задания нескольких стилей в одной строке используется специальное свойство style.cssText:

<div id="div">Button</div>

<script>

div.style.cssText=`color: red !important;

background-color: yellow;

width: 100px;

text-align: center;

`;

alert(div.style.cssText);

</script>

Это свойство редко используется, потому что такое присваивание удаляет все существующие стили: оно не добавляет, а заменяет их. Можно случайно удалить что-то нужное. Но его можно использовать, к примеру, для новых элементов, когда точно известно, что не удалится существующий стиль. То же самое можно сделать установкой атрибута: div.setAttribute('style', 'color: red...').

Не забывайте добавлять к значениям единицы измерения. Например, надо устанавливать 10px, а не просто 10 в свойство elem.style.top, иначе это не сработает:

<body>

<script>

// не работает

document.body.style.margin = 20;

alert(document.body.style.margin); // ''

document.body.style.margin = '20px';

alert(document.body.style.margin); // 20px

alert(document.body.style.marginTop); // 20px

alert(document.body.style.marginLeft); // 20px

</script>

</body>

Обратите внимание, браузер «распаковывает» свойство style.margin в последних строках и выводит style.marginLeft и style.marginTop из него.

[**Вычисленные стили: getComputedStyle**](https://learn.javascript.ru/styles-and-classes#vychislennye-stili-getcomputedstyle)

Метод getComputedStyle позволяет получить текущее значение свойств элемента. Синтаксис:

getComputedStyle(element, [pseudo])

element – элемент, значения для которого нужно получить,

pseudo – указывается, если нужен стиль псевдоэлемента, например ::before. Пустая строка или отсутствие аргумента означают сам элемент.

Результат вызова – объект со стилями, похожий на elem.style, но с учётом всех CSS-классов. Например:

<head>

<style> body { color: red; margin: 5px } </style>

</head>

<body>

<script>

let computedStyle = getComputedStyle(document.body);

alert( computedStyle.marginTop ); // 5px

alert( computedStyle.color ); // rgb(255, 0, 0)

</script>

</body>

Есть две концепции в [CSS](https://drafts.csswg.org/cssom/#resolved-values):

1. Вычисленное (computed) значение – это то, которое получено после применения всех CSS-правил и CSS-наследования. Например, height:1em или font-size:125%.
2. Окончательное ([resolved](https://drafts.csswg.org/cssom/#resolved-values)) значение – непосредственно применяемое к элементу. Значения 1em или 125%являются относительными. Браузер берёт вычисленное значение и делает все единицы измерения фиксированными и абсолютными, например, height:20px or font-size:16px. Для геометрических свойств разрешенные значения могут иметь плавающую точку, например, width:50.5px.

Изначально getComputedStyle был создан для получения вычисленных значений, но окончательные значения гораздо удобнее, и стандарт изменился. Так что, в настоящее время getComputedStyle фактически возвращает окончательное значение свойства, для геометрии оно обычно в пискселях.

Для правильного получения значения нужно указать точное свойство. Например: paddingLeft, marginTop, borderTopWidth. При обращении к сокращенному: padding, margin, border – правильный результат не гарантируется.

Некоторые браузеры (Chrome) отображают 10px в документе ниже, а некоторые (Firefox) – нет:

<style>

body {

margin: 10px;

}

</style>

<script>

let style = getComputedStyle(document.body);

alert(style.margin); // пустая строка в Firefox

</script>

Посещенные ссылки могут быть окрашены 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, проверив стили.

JavaScript не видит стили, применяемые с помощью :visited. Кроме того, в CSS есть ограничение, которое запрещает в целях безопасности применять к :visited CSS-стили, изменяющие геометрию элемента. Это гарантирует, что нет обходного пути для вредоносной страницы проверить, была ли ссылка посещена и, следовательно, нарушить конфиденциальность.

1. **Размеры и прокрутка элементов.**

Существует множество JavaScript-свойcтв, которые позволяют считывать информацию об элементе: ширину, высоту и другие геометрические характеристики. Будем называть их «метрики». Они часто требуются, когда нужно передвигать или позиционировать элементы с помощью JavaScript.

В качестве простого примера демонстрации свойств будем использовать следующий элемент:

<div id="example">

...Текст...

</div>

<style>

#example {

width: 300px;

height: 200px;

border: 25px solid #E8C48F;

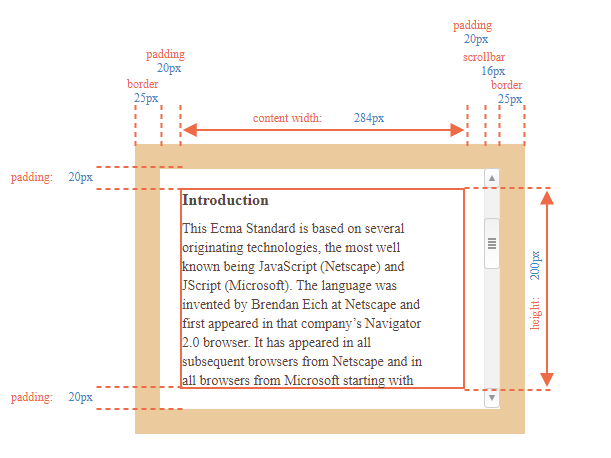
padding: 20px;

overflow: auto;

}

</style>

У элемента есть рамка (border), внутренний отступ (padding) и прокрутка. Полный набор характеристик. Обратите внимание, тут нет внешних отступов (margin), потому что они не являются частью элемента, для них нет особых JavaScript-свойств. Результат выглядит так:

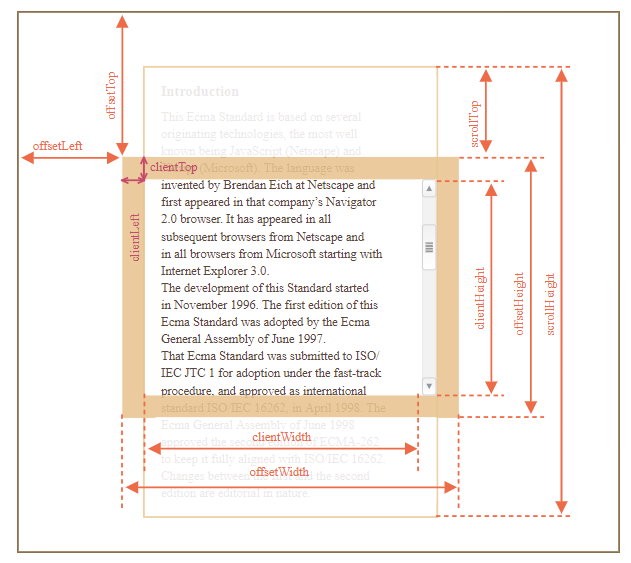


В иллюстрации выше намеренно продемонстрирован самый сложный и полный случай, когда у элемента есть ещё и полоса прокрутки. Некоторые браузеры (не все) отбирают место для неё, забирая его у области, отведённой для содержимого (помечена как «content width» выше).

Таким образом, без учёта полосы прокрутки ширина области содержимого (content width) будет 300px, но если предположить, что ширина полосы прокрутки равна 16px (её точное значение зависит от устройства и браузера), тогда остаётся только 300 - 16 = 284px, и надо это учитывать.

Нижние внутренние отступы padding-bottom изображены на иллюстрациях пустыми, но если элемент содержит много текста, то он будет перекрывать padding-bottom, это нормально.

Вот общая картина с геометрическими свойствами:



Значениями свойств являются числа, подразумевается, что они в пикселях.

Свойства [offsetParent, offsetLeft/Top](https://learn.javascript.ru/size-and-scroll" \l "offsetparent-offsetleft-top) редко используются, они являются «самыми внешними» метриками.

В свойстве offsetParent находится предок элемента, который используется внутри браузера для вычисления координат при рендеринге. То есть, ближайший предок, который удовлетворяет следующим условиям:

* является CSS-позиционированным (CSS-свойство position равно absolute, relative, fixed или sticky);
* или <td>, <th>, <table>;
* или <body>.

Свойства offsetLeft/offsetTop содержат координаты x/y относительно верхнего левого угла offsetParent. В примере ниже внутренний <div> имеет элемент <main> в качестве offsetParent, а свойства offsetLeft/offsetTop являются сдвигами относительно верхнего левого угла (180):

<main style="position: relative" id="main">

<article>

<div id="example" style="position: absolute; left: 180px; top: 180px">...</div>

</article>

</main>

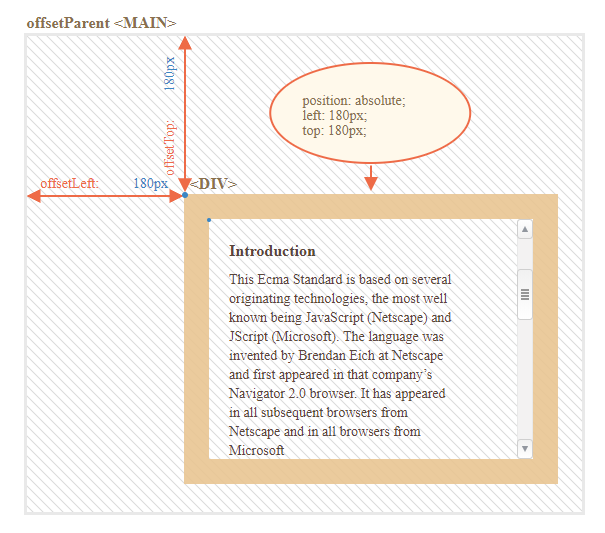
<script>

alert(example.offsetParent.id); // main

alert(example.offsetLeft); // 180 (число, а не строка "180px")

alert(example.offsetTop); // 180

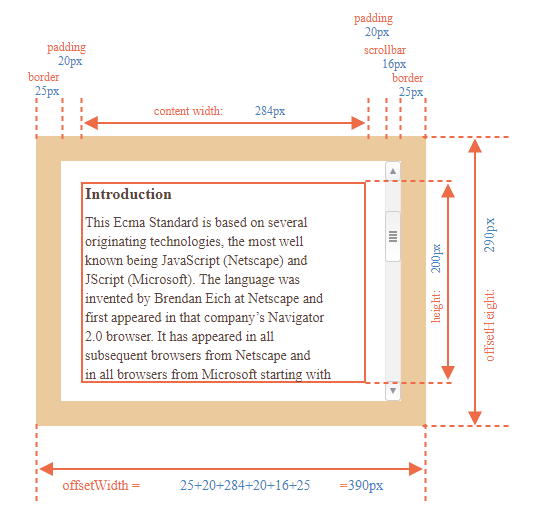
</script>



Существует несколько ситуаций, когда offsetParent равно null:

* 1. Для скрытых элементов (с CSS-свойством display:none или когда его нет в документе).
  2. Для элементов <body> и <html>.
  3. Для элементов с position:fixed.

Свойства [offsetWidth/Height](https://learn.javascript.ru/size-and-scroll" \l "offsetwidth-height) самые простые. Они содержат «внешнюю» ширину/высоту элемента, то есть его полный размер, включая рамки.



Для рассматриваемого элемента:

* offsetWidth = 390 – внешняя ширина блока, её можно получить сложением CSS-ширины (300px), внутренних отступов (2 \* 20px) и рамок (2 \* 25px).
* offsetHeight = 290 – внешняя высота блока.

Метрики для не показываемых элементов равны нулю. Координаты и размеры в JavaScript устанавливаются только для видимых элементов. Если элемент (или любой его родитель) имеет display:none или отсутствует в документе, то все его метрики равны нулю (или null, если это offsetParent). Например, свойство offsetParent равно null, а offsetWidth и offsetHeight равны 0, когда элемент создан, но ещё не вставлен в документ, или если у элемента (или у его родителя) display:none. Это можно использовать, чтобы делать проверку на видимость:

function isHidden(elem) {

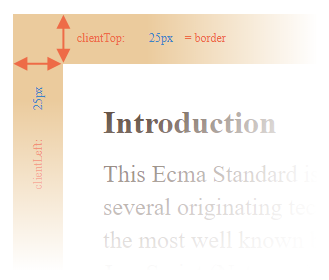
return !elem.offsetWidth && !elem.offsetHeight;

}

Функция isHidden также вернёт true для элементов, которые в принципе показываются, но их размеры равны нулю (например, пустые <div>).

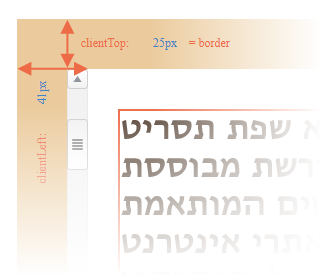
Внутри элемента есть рамки (border). Для них есть свойства-метрики clientTop и clientLeft. В примере:

* clientLeft = 25 – ширина левой рамки,
* clientTop = 25 – ширина верхней рамки.

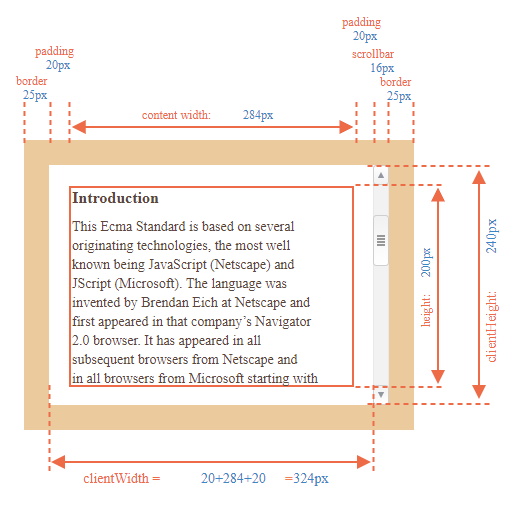


Но на самом деле эти свойства – вовсе не ширины рамок, а отступы внутренней части элемента от внешней. Разница в том, что когда документ располагается справа налево (операционная система на арабском языке или иврите). Полоса прокрутки в этом случае находится слева, и тогда свойство clientLeft включает в себя ещё и ширину полосы прокрутки. В этом случае clientLeft будет равно 25, но с прокруткой – 25 + 16 = 41.

Вот соответствующий пример на иврите:

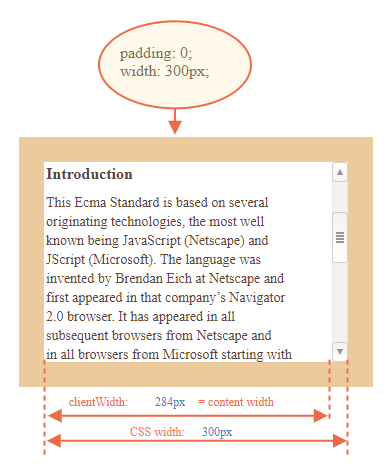


Свойства [clientWidth/Height](https://learn.javascript.ru/size-and-scroll" \l "clientwidth-height) – это размер области внутри рамок элемента. Они включают в себя ширину области содержимого вместе с внутренними отступами padding, но без прокрутки:

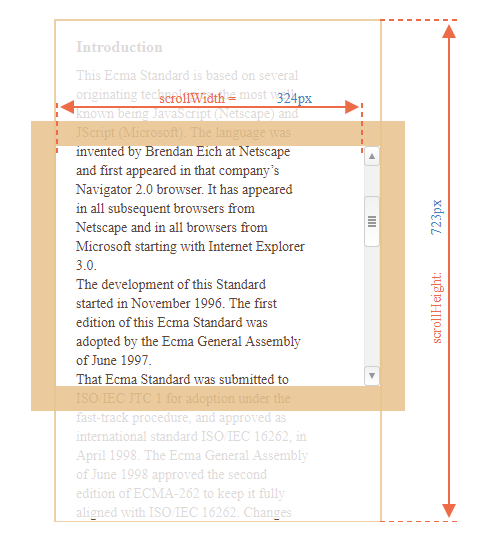


На рисунке выше горизонтальной прокрутки нет, так что высота clientHeight в точности то, что внутри рамок: CSS-высота 200px плюс верхние и нижние внутренние отступы (2 \* 20px), итого 240px. clientWidth – ширина содержимого здесь равна не 300px, а 284px, т.к. 16px отведено для полосы прокрутки. Таким образом: 284px плюс левый и правый отступы – всего 324px.

Если нет внутренних отступов padding, то clientWidth/Height в точности равны размеру области содержимого внутри рамок и полосы прокрутки (если она есть).

Поэтому в тех случаях, когда точно известно, что отступов нет, можно использовать clientWidth/clientHeight для получения размеров внутренней области содержимого.

Свойства [scrollWidth/Height](https://learn.javascript.ru/size-and-scroll" \l "scrollwidth-height) – как clientWidth/clientHeight, но также включают в себя прокрученную (которую не видно) часть элемента.



На рисунке выше:

* scrollHeight = 723 – полная внутренняя высота, включая прокрученную область;
* scrollWidth = 324 – полная внутренняя ширина, в данном случае прокрутки нет, поэтому она равна clientWidth.

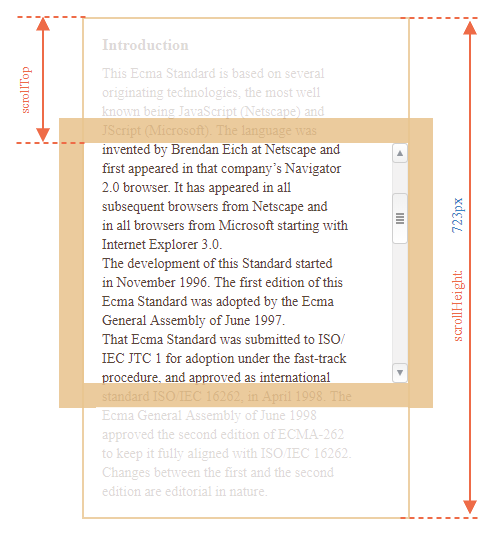
Эти свойства можно использовать, чтобы «распахнуть» элемент на всю ширину/высоту, например при нажатии на кнопку:

element.style.height = `${element.scrollHeight}px`;



Свойства scrollLeft/scrollTop – ширина/высота невидимой, прокрученной в данный момент, части элемента слева и сверху.

Следующая иллюстрация показывает значения scrollHeight и scrollTop для блока с вертикальной прокруткой.



Другими словами, свойство scrollTop – это «сколько уже прокручено вверх».

В отличие от большинства свойств, которые доступны только для чтения, значения scrollLeft/scrollTop можно изменять, и браузер выполнит прокрутку элемента. Установка значения scrollTop на 0 или Infinity прокрутит элемент в самый верх/низ соответственно.

Выше рассматривались метрики, которые есть у DOM-элементов, и которые можно использовать для получения различных высот, ширин и прочих расстояний. Но как известно, CSS-высоту и ширину можно извлечь, используя getComputedStyle:

let elem = document.body;

alert( getComputedStyle(elem).width );

Тем не менее стоит использовать свойства-метрики по следующим причинам:

1. Во-первых, CSS-свойства width/height зависят от другого свойства – box-sizing, которое определяет, «что такое», собственно, эти CSS-ширина и высота. Получается, что изменение box-sizing, к примеру, для более удобной вёрстки, сломает такой JavaScript.
2. Во-вторых, в CSS свойства width/height могут быть равны auto, например, для инлайнового элемента:

<span id="elem">Привет!</span>

<script>

alert( getComputedStyle(elem).width ); // auto

</script>

Конечно, с точки зрения CSS width:auto – совершенно нормально, но в JavaScript нужен конкретный размер в px, который можно использовать для вычислений. Получается, что в данном случае ширина из CSS вообще бесполезна.

Есть и ещё одна причина: полоса прокрутки. Бывает, без полосы прокрутки код работает прекрасно, но стоит ей появиться, как начинают проявляться баги. Так происходит потому, что полоса прокрутки «забирает» место от области внутреннего содержимого в некоторых браузерах. Таким образом, реальная ширина содержимого меньше CSS-ширины. Как раз это и учитывают свойства clientWidth/clientHeight.

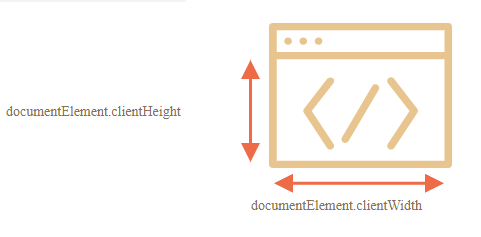
Но с getComputedStyle(elem).width ситуация иная. Некоторые браузеры (например, Chrome) возвращают реальную внутреннюю ширину с вычетом ширины полосы прокрутки, а некоторые (например, Firefox) – именно CSS-свойство (игнорируя полосу прокрутки). Эти кроссбраузерные отличия – ещё один повод не использовать getComputedStyle, а использовать свойства-метрики.

Описанные различия касаются только чтения свойства getComputedStyle(...).width из JavaScript, визуальное отображение корректно в обоих случаях.

1. **Размеры и прокрутка окна.**

**[Ширина/высота окна](https://learn.javascript.ru/size-and-scroll-window" \l "shirina-vysota-okna)**

Чтобы получить ширину/высоту окна, можно взять свойства clientWidth/clientHeight из document.documentElement:



Например, получить высоту окна можно так: document.documentElement.clientHeight.

Браузеры также поддерживают свойства window.innerWidth/innerHeight. Но их не стоит использовать их, так как если есть полоса прокрутки, и она занимает какое-то место, то свойства clientWidth/clientHeight указывают на ширину/высоту документа без неё (за её вычетом). Иными словами, они возвращают высоту/ширину видимой части документа, доступной для содержимого. A window.innerWidth/innerHeight включают в себя полосу прокрутки. Если полоса прокрутки занимает некоторое место, то эти две строки выведут разные значения:

alert( window.innerWidth ); // полная ширина окна

alert( document.documentElement.clientWidth ); // ширина окна за вычетом полосы прокрутки

В большинстве случаев нужна доступная ширина окна: для рисования или позиционирования. Полоса прокрутки «забирает» её часть. Поэтому следует использовать documentElement.clientHeight/Width.

Обратите внимание, что геометрические свойства верхнего уровня могут работать немного иначе, если в HTML нет <!DOCTYPE HTML>. Возможны странности. Поэтому в современном HTML всегда надо указывать DOCTYPE.

[**Ширина/высота документа**](https://learn.javascript.ru/size-and-scroll-window#shirina-vysota-dokumenta)

Теоретически, т.к. корневым элементом документа является documentElement.clientWidth/Height, и он включает в себя всё содержимое, то можно получить полный размер документа как documentElement.scrollWidth/scrollHeight.

Но именно на этом элементе, для страницы в целом, эти свойства работают не так, как предполагается. В Chrome/Safari/Opera, если нет прокрутки, то documentElement.scrollHeight может быть даже меньше, чем documentElement.clientHeight. С точки зрения элемента это невозможная ситуация. Чтобы надёжно получить полную высоту документа, следует взять максимальное из этих свойств:

let scrollHeight = Math.max(

document.body.scrollHeight, document.documentElement.scrollHeight,

document.body.offsetHeight, document.documentElement.offsetHeight,

document.body.clientHeight, document.documentElement.clientHeight

);

alert('Полная высота документа с прокручиваемой частью: ' + scrollHeight);

Обычные элементы хранят текущее состояние прокрутки в elem.scrollLeft/scrollTop. Чтобы получить текущее состояние прокрутки страницы в большинстве браузеров можно обратиться к documentElement.scrollLeft/Top, за исключением основанных на старом WebKit (Safari), где есть ошибка ([5991](https://bugs.webkit.org/show_bug.cgi?id=5991)), и там нужно использовать document.body вместо document.documentElement.

Текущую прокрутку можно прочитать из свойств window.pageXOffset/pageYOffset:

alert('Текущая прокрутка сверху: ' + window.pageYOffset);

alert('Текущая прокрутка слева: ' + window.pageXOffset);

Эти свойства доступны только для чтения.

Для прокрутки страницы из JavaScript её DOM должен быть полностью построен. Например, если попытаться прокрутить страницу из скрипта в <head>, это не сработает. Обычные элементы можно прокручивать, изменяя scrollTop/scrollLeft. Можно сделать то же самое для страницы в целом, используя document.documentElement.scrollTop/Left (кроме основанных на старом WebKit (Safari), где, как сказано выше, document.body.scrollTop/Left).

Есть и другие способы, в которых подобных несовместимостей нет: специальные методы window.scrollBy(x,y) и window.scrollTo(pageX,pageY).

Метод scrollBy(x,y) прокручивает страницу относительно её текущего положения. Например, scrollBy(0,10)прокручивает страницу на 10px вниз.

Метод scrollTo(pageX,pageY) прокручивает страницу на абсолютные координаты (pageX,pageY). То есть, чтобы левый-верхний угол видимой части страницы имел данные координаты относительно левого верхнего угла документа. Это всё равно, что поставить scrollLeft/scrollTop. Для прокрутки в самое начало можно использовать scrollTo(0,0).

Эти методы одинаково работают для всех браузеров.

Рассмотрим ещё один метод: [elem.scrollIntoView(top)](https://developer.mozilla.org/ru/docs/Web/API/Element/scrollIntoView). Вызов elem.scrollIntoView(top) прокручивает страницу, чтобы elem оказался вверху. У него есть один аргумент:

* если top=true (по умолчанию), то страница будет прокручена, чтобы elem появился в верхней части окна. Верхний край элемента совмещён с верхней частью окна.
* если top=false, то страница будет прокручена, чтобы elem появился внизу. Нижний край элемента будет совмещён с нижним краем окна.

Иногда нужно 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 установить document.body.style.overflow = "hidden":

document.body.style.overflow = 'hidden'

Возобновить прокрутку:

document.body.style.overflow = ''

Аналогичным образом можно запретить прокрутку для других элементов, а не только для document.body.

Недостатком этого способа является то, что сама полоса прокрутки исчезает. Если она занимала некоторую ширину, то теперь эта ширина освободится, и содержимое страницы расширится, текст «прыгнет», заняв освободившееся место.

Это выглядит немного странно, но это можно обойти, если сравнить clientWidth до и после остановки, и если clientWidth увеличится (значит полоса прокрутки исчезла), то добавить padding в document.body вместо полосы прокрутки, чтобы оставить ширину содержимого прежней.

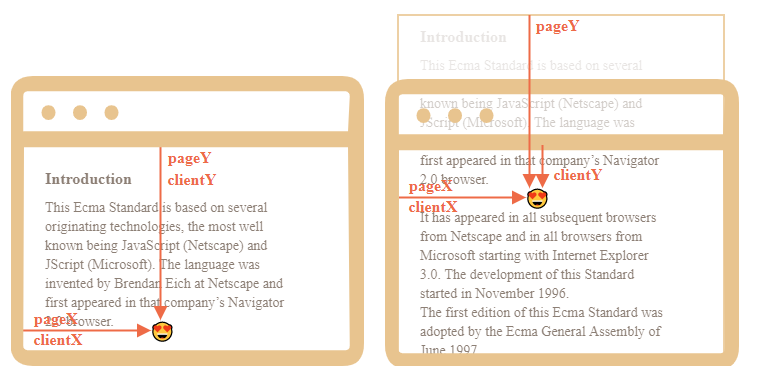
1. **Координаты**

Большинство соответствующих методов JavaScript работают в одной из двух указанных ниже систем координат:

1. Относительно окна браузера – как position:fixed, отсчёт идёт от верхнего левого угла окна. Далее будем обозначать эти координаты как clientX/clientY.
2. Относительно документа – как position:absolute на уровне документа, отсчёт идёт от верхнего левого угла документа. Далее будем обозначать эти координаты как pageX/pageY.

Когда страница полностью прокручена в самое начало, то верхний левый угол окна совпадает с левым верхним углом документа, при этом обе этих системы координат тоже совпадают. Но если происходит прокрутка, то координаты элементов в контексте окна меняются, так как они двигаются, но в то же время их координаты относительно документа остаются такими же.

На картинке ниже показаны координат точки до прокрутки (слева) и после (справа):



При прокрутке документа:

* pageY – координата точки относительно документа осталась без изменений, так как отсчёт по-прежнему ведётся от верхней границы документа (сейчас она прокручена наверх).
* clientY – координата точки относительно окна изменилась (стрелка на рисунке стала короче), так как точка стала ближе к верхней границе окна.

[**Координаты относительно окна: getBoundingClientRect**](https://learn.javascript.ru/coordinates#koordinaty-otnositelno-okna-getboundingclientrect)

Метод elem.getBoundingClientRect() возвращает координаты в контексте окна для минимального по размеру прямоугольника, который заключает в себе элемент elem, в виде объекта встроенного класса [DOMRect](https://www.w3.org/TR/geometry-1/#domrect).

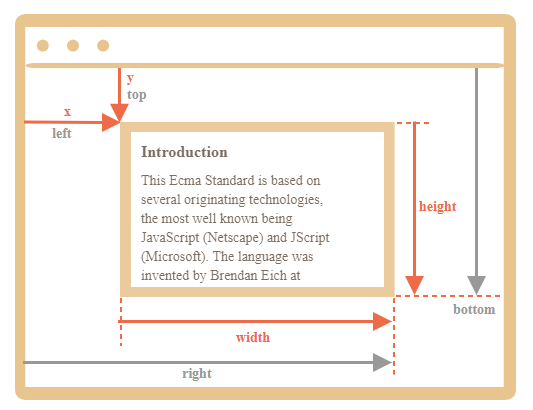
Основные свойства объекта типа DOMRect:

* x/y – X/Y-координаты начала прямоугольника относительно окна,
* width/height – ширина/высота прямоугольника (могут быть отрицательными).

Дополнительные, «зависимые», свойства:

* top/bottom – Y-координата верхней/нижней границы прямоугольника,
* left/right – X-координата левой/правой границы прямоугольника.

Картинка с результатами вызова elem.getBoundingClientRect():



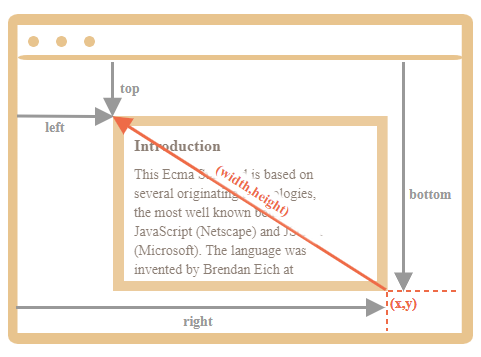
Как видно, x/y и width/height уже точно задают прямоугольник. Остальные свойства могут быть легко вычислены на их основе: left = x, top = y, right = x + width, bottom = y + height.

Заметим:

* Координаты могут считаться с десятичной частью, например, 10.5. Это нормально, ведь браузер использует дроби в своих внутренних вычислениях.
* Координаты могут быть отрицательными. Например, если страница прокручена так, что элемент elem ушёл вверх за пределы окна, то вызов elem.getBoundingClientRect().top вернёт отрицательное значение.

С математической точки зрения, прямоугольник однозначно задаётся начальной точкой (x,y) и вектором направления (width, height). Так что дополнительные зависимые свойства существуют лишь для удобства. Технически, значения width/height могут быть отрицательными, это позволяет задать «направленный» прямоугольник, например, для выделения мышью с отмеченным началом и концом.

Пример прямоугольника с отрицательными width и height (например, width=-200, height=-100):



Он начинается в правом-нижнем углу, и затем «растёт» влево-вверх, так как отрицательные width/height ведут его назад по координатам. Как видно, свойства left/top здесь не равны x/y. То есть они не дублируют друг друга. Формулы выше могут быть исправлены с учётом возможных отрицательных значений width/height. Это достаточно просто сделать, но редко требуется, так как результат вызова elem.getBoundingClientRect() всегда возвращает положительные значения для ширины/высоты.

Internet Explorer и Edge не поддерживают свойства x/y. Таким образом, можно либо сделать полифил (добавив соответствующие геттеры в DomRect.prototype), либо использовать top/left, так как это всегда одно и то же при положительных width/height, в частности – в результате вызова elem.getBoundingClientRect().

Есть очевидное сходство между координатами относительно окна и CSS position:fixed. Но в CSS свойство right означает расстояние от правого края, и свойство bottom означает расстояние от нижнего края окна браузера. Если взглянуть на картинку выше, то видно, что в JavaScript это не так. Все координаты в контексте окна считаются от верхнего левого угла, включая right/bottom.

Вызов document.elementFromPoint(x, y) возвращает самый глубоко вложенный элемент в окне, находящийся по координатам (x, y). Синтаксис:

let elem = document.elementFromPoint(x, y);

Например, код ниже выделяет с помощью стилей и выводит имя тега элемента, который сейчас в центре окна браузера:

let centerX = document.documentElement.clientWidth / 2;

let centerY = document.documentElement.clientHeight / 2;

let elem = document.elementFromPoint(centerX, centerY);

elem.style.background = "red";

alert(elem.tagName);

Поскольку используются координаты в контексте окна, то элемент может быть разным, в зависимости от того, какая сейчас прокрутка.

Метод document.elementFromPoint(x,y) работает, только если координаты (x,y) относятся к видимой части содержимого окна. Если любая из координат представляет собой отрицательное число или превышает размеры окна, то возвращается null. Вот типичная ошибка, которая может произойти, если в коде нет соответствующей проверки:

let elem = document.elementFromPoint(x, y);

elem.style.background = ''; // ошибка

Чаще всего нужны координаты для позиционирования чего-либо. Чтобы показать что-то около нужного элемента, можно вызвать getBoundingClientRect, чтобы получить его координаты элемента, а затем использовать CSS-свойство position вместе с left/top (или right/bottom). Например, функция createMessageUnder(elem, html) ниже показывает сообщение под элементом elem:

let elem = document.getElementById("coords-show-mark");

function createMessageUnder(elem, html) {

let message = document.createElement('div');

message.style.cssText = "position:fixed; color: red";

let coords = elem.getBoundingClientRect();

message.style.left = coords.left + "px";

message.style.top = coords.bottom + "px";

message.innerHTML = html;

return message;

}

let message = createMessageUnder(elem, 'Hello, world!');

document.body.append(message);

setTimeout(() => message.remove(), 5000);

При прокрутке страницы сообщение будет уплывать от кнопки. Причина в том, что сообщение позиционируется с помощью position:fixed, поэтому оно остаётся всегда на том же самом месте в окне при прокрутке страницы. Чтобы изменить это, нужно использовать другую систему координат, где сообщение позиционировалось бы относительно документа, и свойство position:absolute.

[**Координаты относительно документа**](https://learn.javascript.ru/coordinates#getCoords)

В такой системе координат отсчёт ведётся от левого верхнего угла документа, не окна. В CSS координаты относительно окна браузера соответствуют свойству position:fixed, а координаты относительно документа – свойству position:absolute на самом верхнем уровне вложенности.

Можно воспользоваться свойствами position:absolute и top/left, чтобы привязать что-нибудь к конкретному месту в документе. При этом прокрутка страницы не имеет значения. Но сначала нужно получить верные координаты.

Не существует стандартного метода, который возвращал бы координаты элемента относительно документа, но можно написать его сами.

Две системы координат связаны следующими формулами:

* pageY = clientY + высота вертикально прокрученной части документа,
* pageX = clientX + ширина горизонтально прокрученной части документа.

Функция getCoords(elem) берёт координаты в контексте окна с помощью elem.getBoundingClientRect() и добавляет к ним значение соответствующей прокрутки:

function getCoords(elem) {

let box = elem.getBoundingClientRect();

return {

top: box.top + pageYOffset,

left: box.left + pageXOffset

};

}

Если бы в примере выше функция использовалась вместе с position:absolute, то при прокрутке сообщение оставалось бы рядом с элементом. Модифицированная функция createMessageUnder:

function createMessageUnder(elem, html) {

let message = document.createElement('div');

message.style.cssText = "position:absolute; color: red";

let coords = getCoords(elem);

message.style.left = coords.left + "px";

message.style.top = coords.bottom + "px";

message.innerHTML = html;

return message;

}