**[innerHTML: содержимое элемента](https://learn.javascript.ru/basic-dom-node-properties" \l "innerhtml-soderzhimoe-elementa)**

Свойство [innerHTML](https://w3c.github.io/DOM-Parsing/#widl-Element-innerHTML) позволяет получить HTML-содержимое элемента в виде строки.

Мы также можем 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>

Мы можем попробовать вставить некорректный HTML, браузер исправит наши ошибки:

<body>

<script>

document.body.innerHTML = '<b>тест'; // забыли закрыть тег

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

</script>

</body>

Скрипты не выполнятся

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

**[Будьте внимательны: «innerHTML+=» осуществляет перезапись](https://learn.javascript.ru/basic-dom-node-properties" \l "budte-vnimatelny-innerhtml-osuschestvlyaet-perezapis)**

Мы можем добавить HTML к элементу, используя elem.innerHTML+="ещё html".

Вот так:

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 (надеемся, картинка закеширована). 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, введённым пользователем, то текст будет удалён. И т.д.

К счастью, есть и другие способы добавить содержимое, не использующие innerHTML, которые мы изучим позже.

**[outerHTML: HTML элемента целиком](https://learn.javascript.ru/basic-dom-node-properties" \l "outerhtml-html-elementa-tselikom)**

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

Посмотрим на 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.**

Да, звучит странно, и это действительно необычно, поэтому здесь мы и отмечаем это особо.

Рассмотрим пример:

<div>Привет, мир!</div>

<script>

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

// заменяем div.outerHTML на <p>...</p>

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

// Содержимое div осталось тем же!

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

</script>

Какая-то магия, да?

В строке (\*) мы заменили div на <p>Новый элемент</p>. Во внешнем документе мы видим 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>Новый элемент</p>.
* В div осталось старое значение. Новый HTML не сохранён 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.

Мы можем писать в elem.outerHTML, но надо иметь в виду, что это не меняет элемент, в который мы пишем. Вместо этого создаётся новый HTML на его месте. Мы можем получить ссылки на новые элементы, обратившись к DOM.

**[textContent: просто текст](https://learn.javascript.ru/basic-dom-node-properties" \l "textcontent-prosto-tekst)**

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>

Как мы видим, 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 textContent, т.к. позволяет писать текст «безопасным способом».**

Представим, что у нас есть произвольная строка, введённая пользователем, и мы хотим 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 мы видим имя, выделенное жирным шрифтом.
2. Во второй <div> имя приходит «как текст», поэтому мы видим <b>Винни-пух!</b>.

В большинстве случаев мы рассчитываем получить от пользователя текст и хотим, чтобы он интерпретировался как текст. Мы не хотим, 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, видим ли мы элемент или нет.

Мы можем 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" \l "drugie-svoystva)**

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

* value – значение для <input>, <select> и <textarea> (HTMLInputElement, HTMLSelectElement…).
* 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 мы можем получить к нему доступ.

Если мы хотим узнать полный список поддерживаемых свойств для данного класса, можно найти их в спецификации. Например, класс HTMLInputElement описывается здесь: <https://html.spec.whatwg.org/#htmlinputelement>.

Если же нам нужно быстро что-либо узнать или нас интересует специфика определённого браузера – мы всегда можем вывести элемент в консоль, используя console.dir(elem), и прочитать все свойства. Или исследовать «свойства DOM» во вкладке Elements браузерных инструментов разработчика.

**Атрибуты и свойства**

Когда браузер загружает страницу, он «читает» (также говорят: «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! В этой главе мы уделим внимание различию этих двух понятий, чтобы посмотреть, как работать с ними, когда они одинаковые и когда разные.

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

Ранее мы уже видели встроенные DOM-свойства. Их много. Но технически нас никто не ограничивает, и если этого мало – мы можем добавить своё собственное свойство.

DOM-узлы – это обычные объекты JavaScript. Мы можем их изменять.

Например, создадим новое свойство для document.body:

document.body.myData = {

name: 'Caesar',

title: 'Imperator'

};

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

Мы можем добавить и метод:

document.body.sayTagName = function() {

alert(this.tagName);

};

document.body.sayTagName(); // BODY (значением "this" в этом методе будет 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" \l "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

// нестандартный атрибут не преобразуется в свойство

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/" \l "htmlinputelement)), но не является стандартным для <body> ([HTMLBodyElement](https://html.spec.whatwg.org/" \l "htmlbodyelement)). Стандартные атрибуты описаны в спецификации для соответствующего класса элемента.

Мы можем увидеть это на примере ниже:

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

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

<script>

alert(input.type); // text

alert(body.type); // undefined: DOM-свойство не создалось, потому что оно нестандартное

</script>

</body>

Таким образом, для нестандартных атрибутов не будет соответствующих DOM-свойств. Есть ли способ получить такие атрибуты?

Конечно. 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.

Эти 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>

Пожалуйста, обратите внимание:

1. getAttribute('About') – здесь первая буква заглавная, а в HTML – строчная. Но это не важно: имена атрибутов регистронезависимы.
2. Мы можем присвоить что угодно атрибуту, но это станет строкой. Поэтому в этой строчке мы получаем значение "123".
3. Все атрибуты, в том числе те, которые мы установили, 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" \l "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 (не обновилось!)

</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 мы захотим восстановить «оригинальное» значение из HTML, оно будет в атрибуте.

# Изменение документа

Модификации DOM – это ключ к созданию «живых» страниц.

Здесь мы увидим, как создавать новые элементы «на лету» и изменять уже существующие.

## [Пример: показать сообщение](https://learn.javascript.ru/modifying-document" \l "primer-pokazat-soobschenie)

Рассмотрим методы на примере – а именно, добавим на страницу сообщение, которое будет выглядеть получше, чем alert.

Вот такое:

<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 давайте 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');

### [Создание сообщения](https://learn.javascript.ru/modifying-document" \l "sozdanie-soobscheniya)

В нашем случае 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.";

Мы создали элемент, но пока он только в переменной. Мы не можем видеть его на странице, поскольку он не является частью документа.

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

Чтобы наш div появился, нам 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>

Вот методы для различных вариантов вставки:

* node.append(...nodes or strings) – 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:

&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.

А что, если мы хотим вставить HTML именно «как html», со всеми тегами и прочим, как делает это elem.innerHTML?

## [insertAdjacentHTML/Text/Element](https://learn.javascript.ru/modifying-document" \l "insertadjacenthtml-text-element)

С этим может помочь другой, довольно 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 мы можем добавлять произвольный HTML на страницу.

Варианты вставки:

Мы можем легко заметить сходство между этой и предыдущей картинкой. Точки вставки фактически одинаковые, но этот метод вставляет HTML.

У метода есть два брата:

* 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.

Они существуют, в основном, чтобы унифицировать синтаксис. На практике часто используется только insertAdjacentHTML. Потому что для элементов и текста у нас есть методы append/prepend/before/after – их быстрее написать, и они могут вставлять как узлы, так и текст.

Так что, вот 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" \l "udalenie-uzlov)

Для удаления узла есть методы node.remove().

Например, сделаем так, чтобы наше 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 нам нужно переместить элемент в другое место – нет необходимости удалять его со старого.

**Все методы вставки автоматически удаляют узлы со старых мест.**

Например, давайте поменяем местами элементы:

<div id="first">Первый</div>

<div id="second">Второй</div>

<script>

// нет необходимости вызывать метод remove

second.after(first); // берёт #second и после него вставляет #first

</script>

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

Как вставить ещё одно подобное сообщение?

Мы могли бы создать функцию и поместить код туда. Альтернатива – клонировать существующий div и изменить текст внутри него (при необходимости).

Иногда, когда у нас есть большой элемент, это может быть быстрее и проще.

* Вызов elem.cloneNode(true) создаёт «глубокий» клон элемента – со всеми атрибутами и дочерними элементами. Если мы вызовем 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>