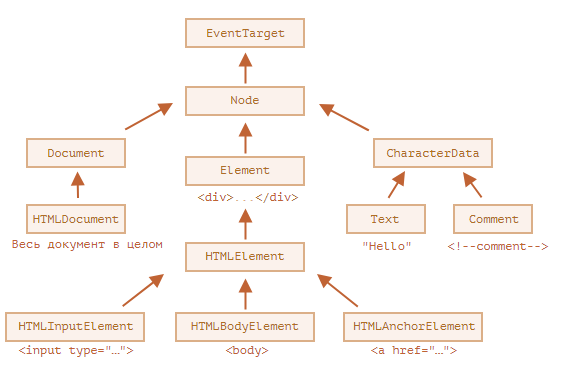
**[Классы DOM-узлов](https://learn.javascript.ru/basic-dom-node-properties" \l "klassy-dom-uzlov)**

Каждый DOM-узел принадлежит определённому классу. Классы формируют иерархию. Весь набор свойств и методов является результатом наследования.

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.

Корнем иерархии является [EventTarget](https://dom.spec.whatwg.org/#eventtarget), от него наследует [Node](https://dom.spec.whatwg.org/#interface-node) и остальные DOM-узлы.



Существуют следующие классы:

* [EventTarget](https://dom.spec.whatwg.org/#eventtarget) – это корневой «абстрактный» класс для всего.

Объекты этого класса никогда не создаются. Он служит основой, благодаря которой все DOM-узлы поддерживают так называемые «события», о которых мы поговорим позже.

* [Node](https://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 определённые классы узлов, которые наследуются от него (и следовательно наследуют функционал Node).

* [Document](https://dom.spec.whatwg.org/#interface-document), по историческим причинам часто наследуется HTMLDocument (хотя последняя спецификация этого не навязывает) – это документ в целом.

Глобальный объект document принадлежит именно к этому классу. Он служит точкой входа в DOM.

* [CharacterData](https://dom.spec.whatwg.org/#interface-characterdata) – «абстрактный» класс. Вот, кем он наследуется:
  + [Text](https://dom.spec.whatwg.org/#interface-text) – класс, соответствующий тексту внутри элементов. Например, Hello в <p>Hello</p>.
  + [Comment](https://dom.spec.whatwg.org/#interface-comment) – класс для комментариев. Они не отображаются, но каждый комментарий становится членом DOM.
* [Element](https://dom.spec.whatwg.org/#interface-element) – это базовый класс для DOM-элементов.

Он обеспечивает навигацию на уровне элементов: nextElementSibling, children. А также и методы поиска элементов: getElementsByTagName, querySelector.

Браузер поддерживает не только HTML, но также XML и SVG. Таким образом, класс Element служит основой для более специфичных классов: SVGElement, XmlElement (они нам здесь не нужны) и HTMLElement.

* И наконец, [HTMLElement](https://html.spec.whatwg.org/multipage/dom.html#htmlelement) является базовым классом для всех остальных HTML-элементов. Мы будем работать с ним большую часть времени.

От него наследуются 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.

Также существует множество других тегов со своими собственными классами, которые могут иметь определенные свойства и методы, в то время как некоторые элементы, такие как <span>, <section> и <article>, не имеют каких-либо определенных свойств, поэтому они являются экземплярами класса HTMLElement.

Таким образом, полный набор свойств и методов данного узла является результатом цепочки наследования.

Рассмотрим DOM-объект для тега <input>. Он принадлежит классу [HTMLInputElement](https://html.spec.whatwg.org/multipage/forms.html#htmlinputelement).

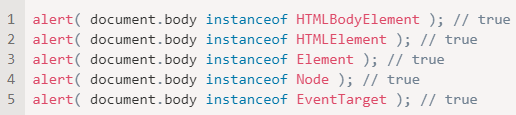
Он получает свойства и методы из (в порядке наследования):

* 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),
* …и, наконец, он наследует от Object, поэтому доступны также 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:



Проверить наследование можно также при помощи instanceof:



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.

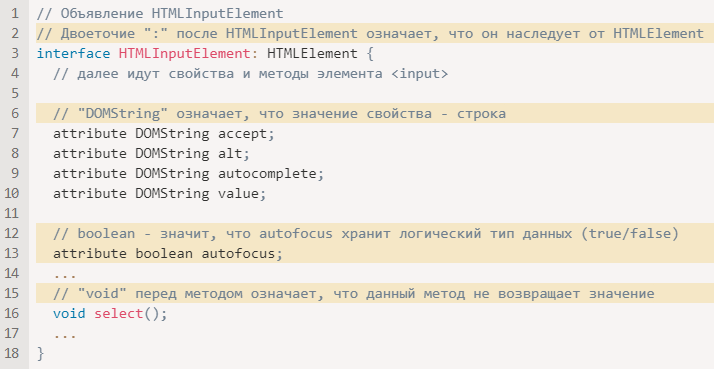
Попробуйте сами на document.body. Вы увидите разницу во всех современных браузерах (кроме Firefox, где console.log(elem) и console.dir(elem) выводят одно и то же – элемент в виде DOM-объекта).

**Спецификация IDL**

В спецификации для описания классов DOM используется не JavaScript, а специальный язык [Interface description language](https://ru.wikipedia.org/wiki/%D0%AF%D0%B7%D1%8B%D0%BA_%D0%BE%D0%BF%D0%B8%D1%81%D0%B0%D0%BD%D0%B8%D1%8F_%D0%B8%D0%BD%D1%82%D0%B5%D1%80%D1%84%D0%B5%D0%B9%D1%81%D0%BE%D0%B2) (IDL), с которым достаточно легко разобраться.

В IDL все свойства представлены с указанием их типов. Например, DOMString, boolean и т.д.

Небольшой отрывок IDL с комментариями:



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

Свойство nodeType предоставляет ещё один, «старомодный» способ узнать «тип» DOM-узла.

Его значением является цифра:

* elem.nodeType == 1 для узлов-элементов,
* elem.nodeType == 3 для текстовых узлов,
* elem.nodeType == 9 для объектов документа,
* [В спецификации](https://dom.spec.whatwg.org/#node) можно посмотреть остальные значения.

В современных скриптах, чтобы узнать тип узла, мы можем использовать метод instanceof и другие способы проверить класс, но иногда nodeType проще использовать. Мы не можем изменить значение nodeType, только прочитать его.

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

Получив DOM-узел, мы можем узнать имя его тега из свойств nodeName и tagName:



Есть ли какая-то разница между tagName и nodeName?

Да, она отражена в названиях свойств, но не очевидна.

* 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:



**Имена тегов (кроме XHTML) всегда пишутся в верхнем регистре**

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.

## [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.

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



**Скрипты не выполнятся**

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

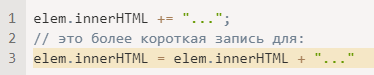
### [Будьте внимательны: «innerHTML+=» осуществляет перезапись](https://learn.javascript.ru/basic-dom-node-properties" \l "budte-vnimatelny-innerhtml-osuschestvlyaet-perezapis)

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



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:



Другими словами, innerHTML+= делает следующее:

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

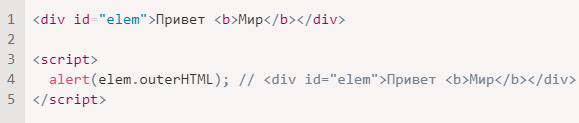
**Так как содержимое «обнуляется» и переписывается заново, все изображения и другие ресурсы будут перезагружены**.

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.

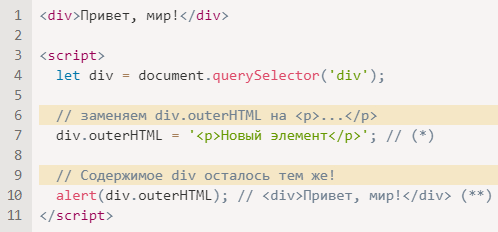
Есть и другие побочные эффекты. Например, если существующий текст выделен мышкой, то при переписывании innerHTML большинство браузеров снимут выделение. А если это поле ввода <input> с текстом, введённым пользователем, то текст будет удалён. И т.д.

## [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.



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



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.

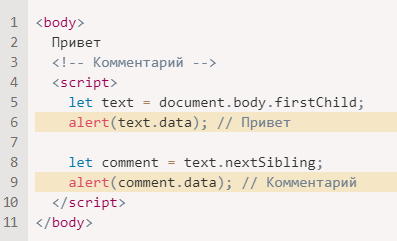
Мы можем писать в elem.outerHTML, но надо иметь в виду, что это не меняет элемент, в который мы пишем. Вместо этого создаётся новый HTML на его месте. Мы можем получить ссылки на новые элементы, обратившись к DOM.

## [nodeValue/data: содержимое текстового узла](https://learn.javascript.ru/basic-dom-node-properties" \l "nodevalue-data-soderzhimoe-tekstovogo-uzla)

Свойство innerHTML есть только у узлов-элементов.

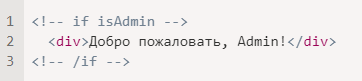
У других типов узлов, в частности, у текстовых, есть свои аналоги: свойства nodeValue и data. Эти свойства очень похожи при использовании, есть лишь небольшие различия в спецификации. Мы будем использовать data, потому что оно короче.

Прочитаем содержимое текстового узла и комментария:



Мы можем представить, для чего нам может понадобиться читать или изменять текстовый узел, но комментарии?

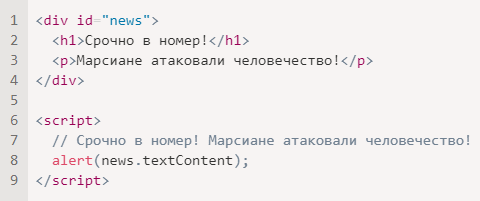
Иногда их используют для вставки информации и инструкций шаблонизатора в HTML, как в примере ниже:



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

## [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>.



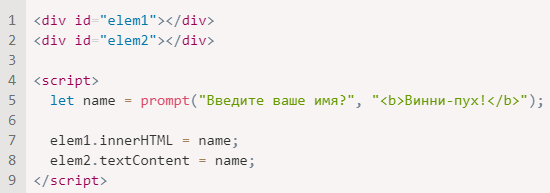
Как мы видим, возвращается только текст, как если бы все <теги> были вырезаны, но текст в них остался.

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.



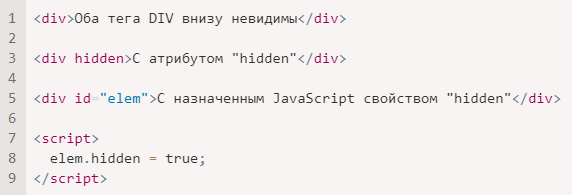
1. В первый <div> имя приходит «как HTML»: все теги стали именно тегами, поэтому мы видим имя, выделенное жирным шрифтом.
2. Во второй <div> имя приходит «как текст», поэтому мы видим <b>Винни-пух!</b>.

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)

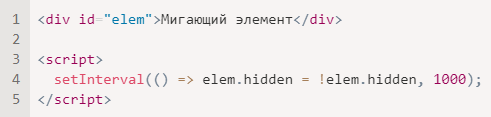
Атрибут и DOM-свойство «hidden» указывает на то, видим ли мы элемент или нет.

Мы можем использовать его в HTML или назначать при помощи JavaScript, как в примере ниже:



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

Мигающий элемент:



**[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…