# 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.

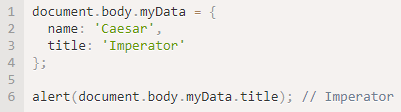
Например, для такого тега <body id="page"> у DOM-объекта будет такое свойство body.id="page".

## DOM-свойства

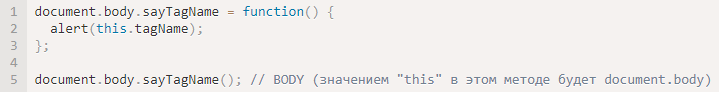
Ранее мы уже видели встроенные DOM-свойства. Их много. Но технически нас никто не ограничивает, и если этого мало – мы можем добавить своё собственное свойство.

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

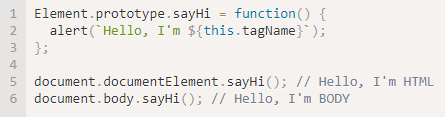
Например, создадим новое свойство для document.body:



Мы можем добавить и метод:



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



Итак, DOM-свойства и методы ведут себя так же, как и обычные объекты JavaScript:

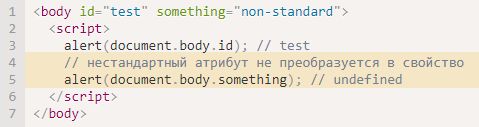
* Им можно присвоить любое значение.
* Они регистрозависимы (нужно писать elem.nodeType, не elem.NoDeTyPe).

## HTML-атрибуты

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.

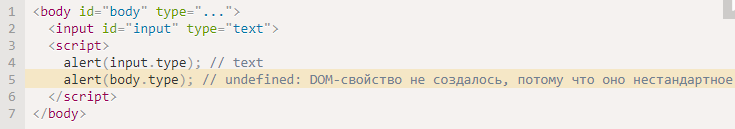
Таким образом, когда у элемента есть id или другой стандартный атрибут, создаётся соответствующее свойство. Но этого не происходит, если атрибут нестандартный.

Например:



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:



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

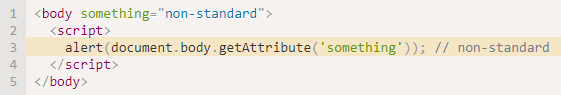
Конечно. Все атрибуты доступны с помощью следующих методов:

* 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/" \l "attr) со свойствами name и value.

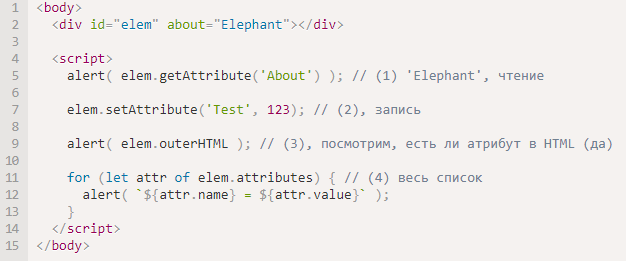
Вот демонстрация чтения нестандартного свойства:



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:



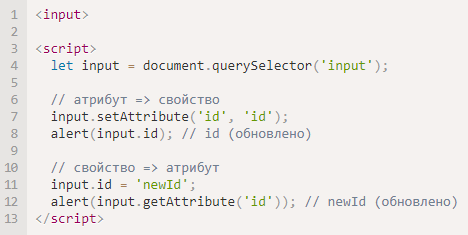
Пожалуйста, обратите внимание:

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.

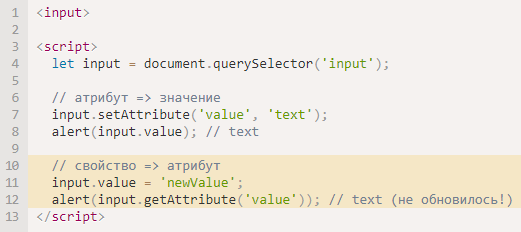
## Синхронизация между атрибутами и свойствами

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:



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



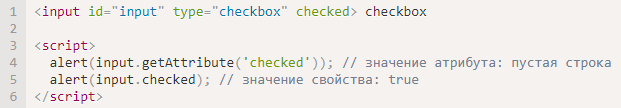
В примере выше:

* 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.

## DOM-свойства типизированы

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



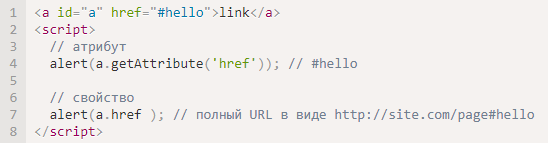
Есть и другие примеры. Атрибут style – строка, но свойство style является объектом:



Хотя большинство свойств, всё же, строки.

При этом некоторые из них, хоть и строки, могут отличаться от атрибутов. Например, DOM-свойство href всегда содержит полный URL, даже если атрибут содержит относительный URL или просто #hash.

Ниже пример:



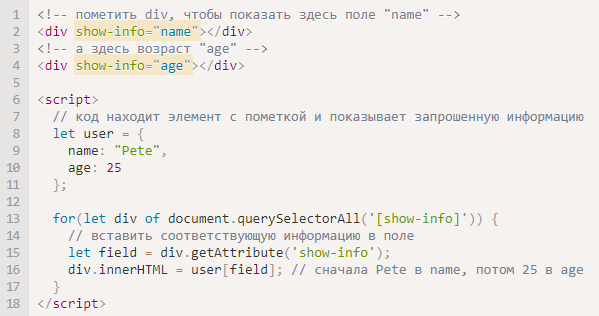
Если же нужно значение href или любого другого атрибута в точности, как оно записано в HTML, можно воспользоваться getAttribute.

## Нестандартные атрибуты, dataset

При написании HTML мы используем много стандартных атрибутов. Но что насчёт нестандартных, пользовательских? Во-первых, давайте посмотрим, полезны они или нет? Для чего они нужны?

Иногда нестандартные атрибуты используются для передачи пользовательских данных из HTML в JavaScript, или чтобы «помечать» HTML-элементы для JavaScript.

Как тут:



Также они могут быть использованы, чтобы стилизовать элементы.

Например, здесь для состояния заказа используется атрибут order-state:



Почему атрибут может быть предпочтительнее таких классов, как .order-state-new, .order-state-pending, order-state-canceled?

Это потому, что атрибутом удобнее управлять. Состояние может быть изменено достаточно просто:



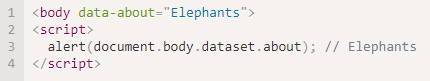
Но с пользовательскими атрибутами могут возникнуть проблемы. Что если мы используем нестандартный атрибут для наших целей, а позже он появится в стандарте и будет выполнять какую-то функцию? Язык HTML живой, он растёт, появляется больше атрибутов, чтобы удовлетворить потребности разработчиков. В этом случае могут возникнуть неожиданные эффекты.

Чтобы избежать конфликтов, существуют атрибуты вида [data-\*](https://html.spec.whatwg.org/" \l "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.

Как тут:



Атрибуты, состоящие из нескольких слов, к примеру data-order-state, становятся свойствами, записанными с помощью верблюжьей нотации: dataset.orderState.

Вот переписанный пример «состояния заказа»:



Использование data-\* атрибутов – валидный, безопасный способ передачи пользовательских данных.

Пожалуйста, примите во внимание, что мы можем не только читать, но и изменять data-атрибуты. Тогда CSS обновит представление соответствующим образом: в примере выше последняя строка (\*) меняет цвет на синий.

# Изменение документа

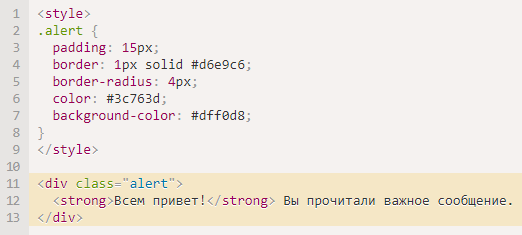
Модификации DOM – это ключ к созданию «живых» страниц.

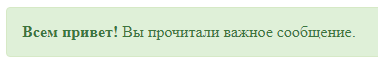
Здесь мы увидим, как создавать новые элементы «на лету» и изменять уже существующие.

## Пример: показать сообщение

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

Вот такое:





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

## Создание элемента

DOM-узел можно создать двумя методами:

**document.createElement(tag)**

Создаёт новый элемент с заданным тегом:



**document.createTextNode(text)**

Создаёт новый *текстовый узел* с заданным текстом:



### **Создание сообщения**

В нашем случае сообщение – это div с классом alert и HTML в нём:



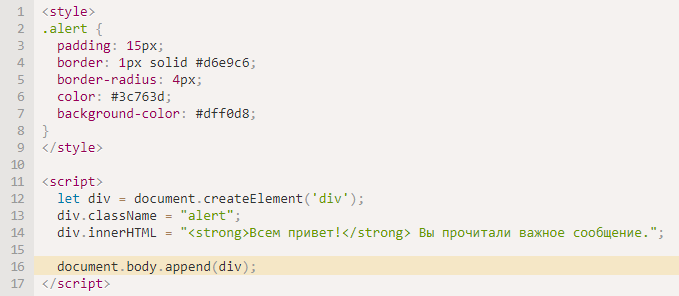
Мы создали элемент, но пока он только в переменной. Мы не можем видеть его на странице, поскольку он не является частью документа.

## Методы вставки

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).

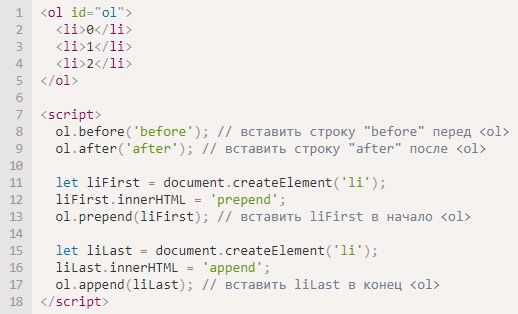
Вот полный пример:

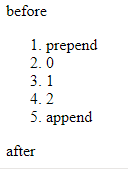


Вот методы для различных вариантов вставки:

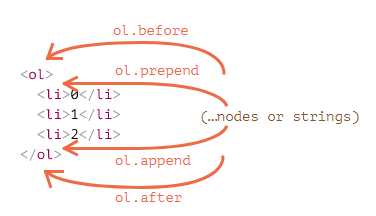
* node.append(...nodes or strings) – добавляет узлы или строки в конец node,
* node.prepend(...nodes or strings) – вставляет узлы или строки в начало node,
* node.before(...nodes or strings) –- вставляет узлы или строки до node,
* node.after(...nodes or strings) –- вставляет узлы или строки после node,
* 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:

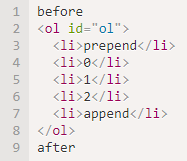




Наглядная иллюстрация того, куда эти методы вставляют:

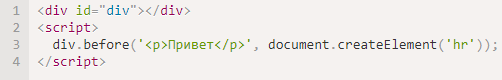


Итоговый список будет таким:



Эти методы могут вставлять несколько узлов и текстовых фрагментов за один вызов.

Например, здесь вставляется строка и элемент:



Весь текст вставляется как текст.

Поэтому финальный HTML будет:



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

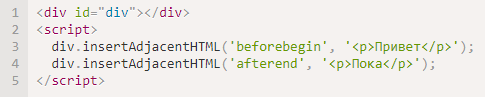
С этим может помочь другой, довольно универсальный метод: elem.insertAdjacentHTML(where, html).

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:

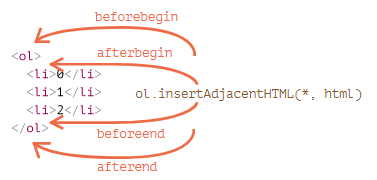


…Приведёт к:



Так мы можем добавлять произвольный HTML на страницу.

Варианты вставки:



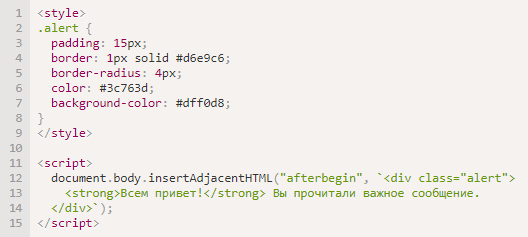
Мы можем легко заметить сходство между этой и предыдущей картинкой. Точки вставки фактически одинаковые, но этот метод вставляет HTML.

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:



## Удаление узлов

Для удаления узла есть методы node.remove().

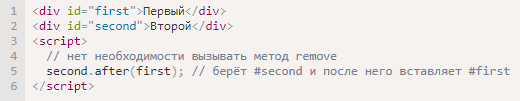
Например, сделаем так, чтобы наше сообщение удалялось через секунду:



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:



## Клонирование узлов: cloneNode

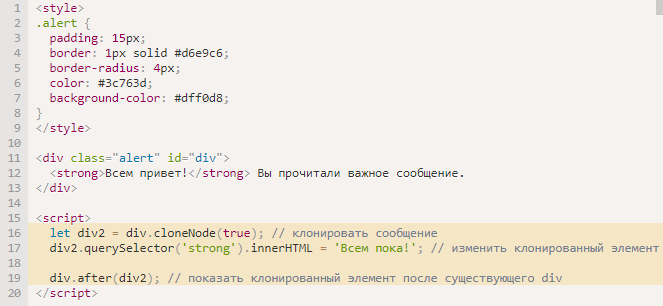
Как вставить ещё одно подобное сообщение?

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:



# Стили и классы

До того, как начнёте изучать способы работы со стилями и классами в JavaScript, есть одно важное правило.

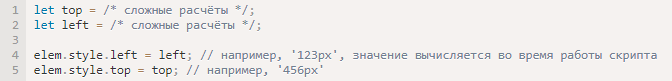
Как правило, существует два способа задания стилей для элемента:

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. Мы должны манипулировать свойством style только в том случае, если классы «не могут справиться».

Например, использование style является приемлемым, если мы вычисляем координаты элемента динамически и хотим установить их из JavaScript:



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

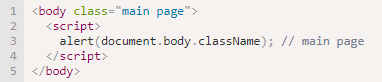
## className и classList

Изменение класса является одним из наиболее часто используемых действий в скриптах.

Когда-то давно в JavaScript существовало ограничение: зарезервированное слово типа "class" не могло быть свойством объекта. Это ограничение сейчас отсутствует, но в то время было невозможно иметь свойство elem.class.

Поэтому для классов было введено схожее свойство "className": elem.className соответствует атрибуту "class".

Например:

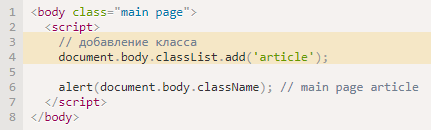


Если мы присваиваем что-то elem.className, то это заменяет всю строку с классами. Иногда это то, что нам нужно, но часто мы хотим добавить/удалить один класс.

Для этого есть другое свойство: elem.classList.

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

Например:

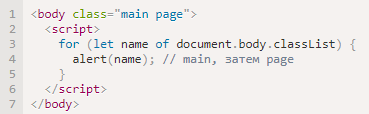


Так что мы можем работать как со строкой полного класса, используя className, так и с отдельными классами, используя classList. Выбираем тот вариант, который нам удобнее.

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.

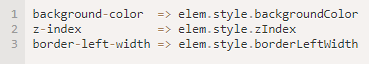
Кроме того, classList является перебираемым, поэтому можно перечислить все классы при помощи for..of:



## 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:



Например:

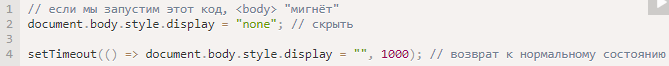


## Сброс стилей

Иногда нам нужно добавить свойство стиля, а потом, позже, убрать его.

Например, чтобы скрыть элемент, мы можем задать elem.style.display = "none".

Затем мы можем удалить свойство style.display, чтобы вернуться к первоначальному состоянию. Вместо delete elem.style.display мы должны присвоить ему пустую строку: elem.style.display = "".

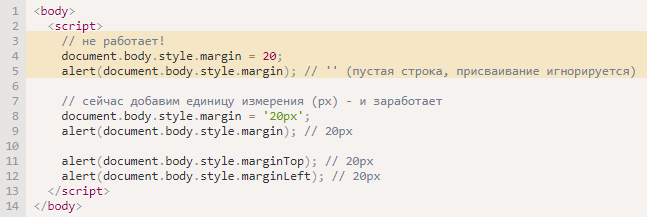


Если мы установим в style.display пустую строку, то браузер применит CSS-классы и встроенные стили, как если бы такого свойства style.display вообще не было.

## Следите за единицами измерения

Не забудьте добавить к значениям единицы измерения.

Например, мы должны устанавливать 10px, а не просто 10 в свойство elem.style.top. Иначе это не сработает:



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

## Вычисленные стили: getComputedStyle

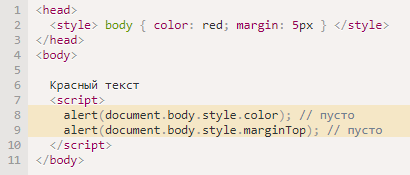
Итак, изменить стиль очень просто. Но как его прочитать?

Например, мы хотим знать размер, отступы, цвет элемента. Как это сделать?

**Свойство style оперирует только значением атрибута "style", без учёта CSS-каскада.**

Поэтому, используя elem.style, мы не можем прочитать ничего, что приходит из классов CSS.

Например, здесь style не может видеть отступы:



…Но что, если нам нужно, скажем, увеличить отступ на 20px? Для начала нужно его текущее значение получить.

Для этого есть метод: getComputedStyle.

Синтаксис:



**element**

Элемент, значения для которого нужно получить

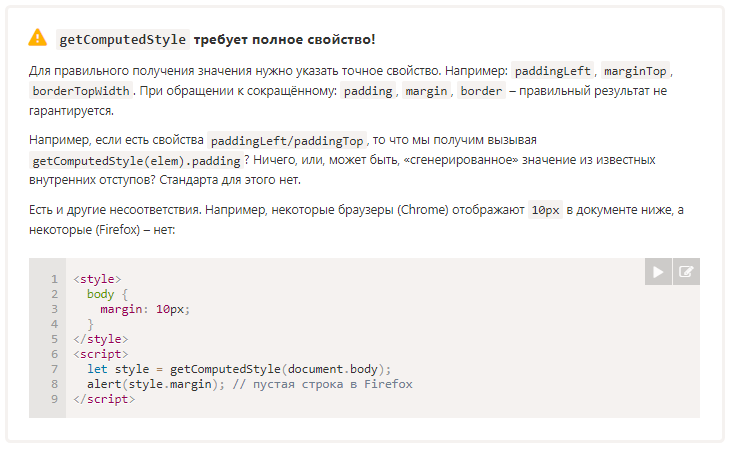
**pseudo**

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:





# Свойства и методы формы

Формы и элементы управления, такие как <input>, имеют множество специальных свойств и событий.

Работать с формами станет намного удобнее, когда мы их изучим.

## Навигация: формы и элементы

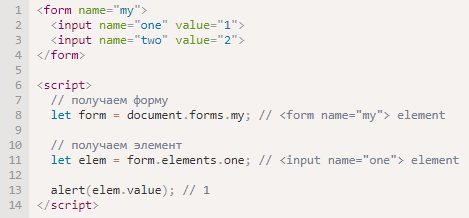
Формы в документе входят в специальную коллекцию document.forms.

Это – так называемая «именованная» коллекция: мы можем использовать для получения формы как её имя, так и порядковый номер в документе.



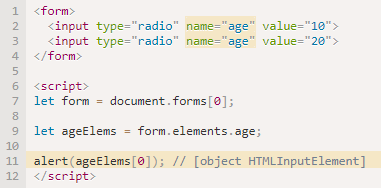
Когда мы уже получили форму, любой элемент доступен в именованной коллекции form.elements.

Например:

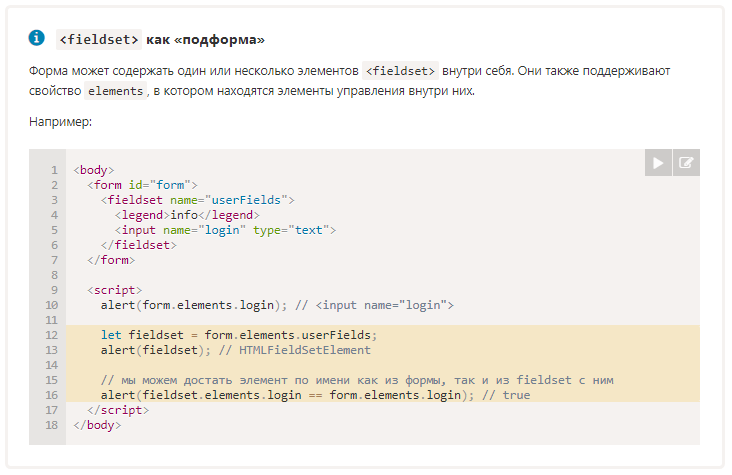


Может быть несколько элементов с одним и тем же именем, это часто бывает с кнопками-переключателями radio.

В этом случае form.elements[name] является коллекцией, например:



Эти навигационные свойства не зависят от структуры тегов внутри формы. Все элементы управления формы, как бы глубоко они ни находились в форме, доступны в коллекции form.elements.

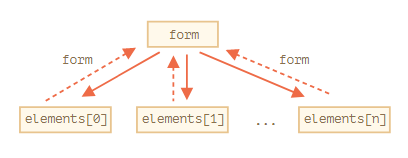




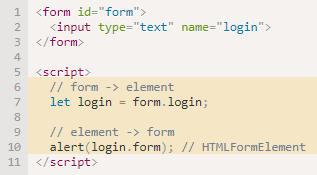
## Обратная ссылка: element.form

Для любого элемента форма доступна через element.form. Так что форма ссылается на все элементы, а эти элементы ссылаются на форму.

Вот иллюстрация:



Пример:



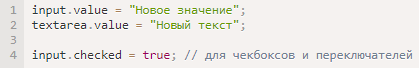
## Элементы формы

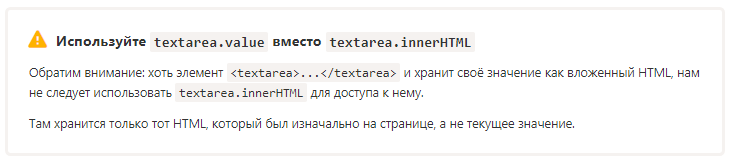
Рассмотрим элементы управления, используемые в формах.

### [input и textarea](https://learn.javascript.ru/form-elements" \l "input-i-textarea)

К их значению можно получить доступ через свойство input.value (строка) или input.checked (булево значение) для чекбоксов.

Вот так:





### **select и option**

Элемент <select> имеет 3 важных свойства:

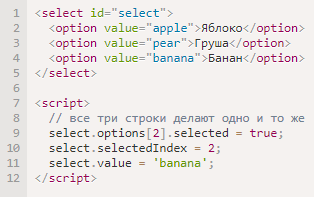
1. select.options – коллекция из подэлементов <option>,
2. select.value – значение выбранного в данный момент <option>,
3. select.selectedIndex – номер выбранного <option>.

Они дают три разных способа установить значение в <select>:

1. Найти соответствующий элемент <option> и установить в option.selected значение true.
2. Установить в select.value значение нужного <option>.
3. Установить в select.selectedIndex номер нужного <option>.

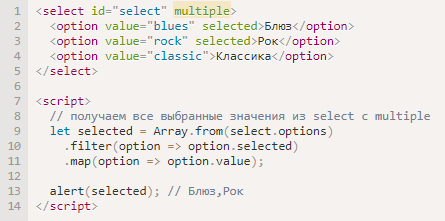
Первый способ наиболее понятный, но (2) и (3) являются более удобными при работе.

Вот эти способы на примере:



В отличие от большинства других элементов управления, <select> позволяет нам выбрать несколько вариантов одновременно, если у него стоит атрибут multiple. Эту возможность используют редко, но в этом случае для работы со значениями необходимо использовать первый способ, то есть ставить или удалять свойство selected у подэлементов <option>.

Их коллекцию можно получить как select.options, например:



### **new Option**

Элемент <option> редко используется сам по себе, но и здесь есть кое-что интересное.

В спецификации есть красивый короткий синтаксис для создания элемента <option>:



Параметры:

* text – текст внутри <option>,
* value – значение,
* defaultSelected – если true, то ставится HTML-атрибут selected,
* selected – если true, то элемент <option> будет выбранным.

Тут может быть небольшая путаница с defaultSelected и selected. Всё просто: defaultSelected задаёт HTML-атрибут, его можно получить как option.getAttribute('selected'), а selected – выбрано значение или нет, именно его важно поставить правильно. Впрочем, обычно ставят оба этих значения в true или не ставят вовсе (т.е. false).

Пример:



Тот же элемент, но выбранный:



Элементы <option> имеют свойства:

**option.selected**

Выбрана ли опция.

**option.index**

Номер опции среди других в списке <select>.

**option.text**

Содержимое опции (то, что видит посетитель).

# События: change, input, cut, copy, paste

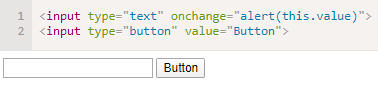
Давайте рассмотрим различные события, сопутствующие обновлению данных.

## Событие: change

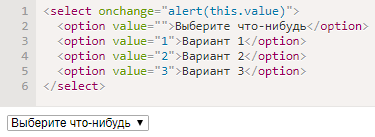
Событие change срабатывает по окончании изменения элемента.

Для текстовых <input> это означает, что событие происходит при потере фокуса.

Пока мы печатаем в текстовом поле в примере ниже, событие не происходит. Но когда мы перемещаем фокус в другое место, например, нажимая на кнопку, то произойдёт событие change:



Для других элементов: select, input type=checkbox/radio событие запускается сразу после изменения значения:

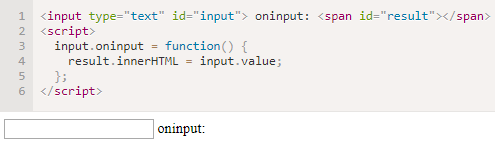


## Событие: input

Событие input срабатывает каждый раз при изменении значения.

В отличие от событий клавиатуры, оно работает при любых изменениях значений, даже если они не связаны с клавиатурными действиями: вставка с помощью мыши или распознавание речи при диктовке текста.

Например:

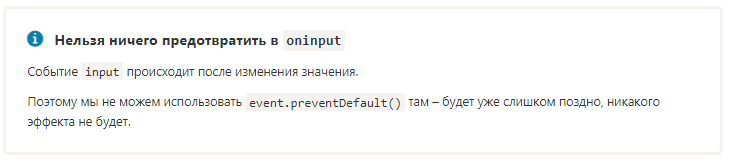






Если мы хотим обрабатывать каждое изменение в <input>, то это событие является лучшим выбором.

С другой стороны, событие input не происходит при вводе с клавиатуры или иных действиях, если при этом не меняется значение в текстовом поле, т.е. нажатия клавиш ⇦, ⇨ и подобных при фокусе на текстовом поле не вызовут это событие.



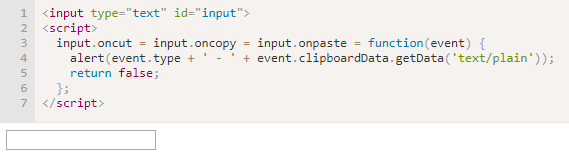
## События: cut, copy, paste

Эти события происходят при вырезании/копировании/вставке данных.

Они относятся к классу [ClipboardEvent](https://www.w3.org/TR/clipboard-apis/" \l "clipboard-event-interfaces) и обеспечивают доступ к копируемым/вставляемым данным.

Мы также можем использовать event.preventDefault() для предотвращения действия по умолчанию, и в итоге ничего не скопируется/не вставится.

Например, код, приведённый ниже, предотвращает все подобные события и показывает, что мы пытаемся вырезать/копировать/вставить:



Технически, мы можем скопировать/вставить всё. Например, мы можем скопировать файл из файловой системы и вставить его.

Существует список методов [в спецификации](https://www.w3.org/TR/clipboard-apis/#dfn-datatransfer) для работы с различными типами данных, чтения/записи в буфер обмена.

Но обратите внимание, что буфер обмена работает глобально, на уровне ОС. Большинство браузеров в целях безопасности разрешают доступ на чтение/запись в буфер обмена только в рамках определённых действий пользователя, к примеру, в обработчиках событий onclick.

Также запрещается генерировать «пользовательские» события буфера обмена при помощи dispatchEvent во всех браузерах, кроме Firefox.

# Отправка формы: событие и метод submit

При отправке формы срабатывает событие submit, оно обычно используется для проверки (валидации) формы перед её отправкой на сервер или для предотвращения отправки и обработки её с помощью JavaScript.

Метод form.submit() позволяет инициировать отправку формы из JavaScript. Мы можем использовать его для динамического создания и отправки наших собственных форм на сервер.

Давайте посмотрим на них подробнее.

## Событие: submit

Есть два основных способа отправить форму:

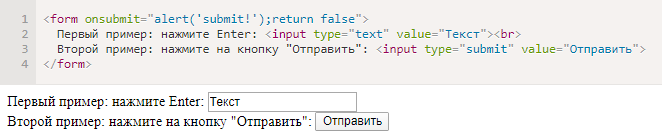
1. Первый – нажать кнопку <input type="submit"> или <input type="image">.
2. Второй – нажать Enter, находясь на каком-нибудь поле.

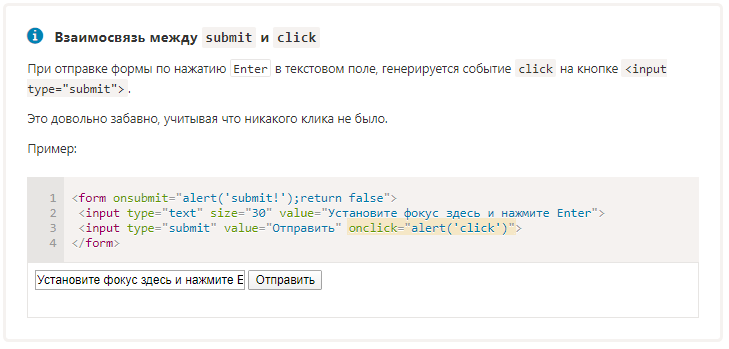
Оба действия сгенерируют событие submit на форме. Обработчик может проверить данные, и если есть ошибки, показать их и вызвать event.preventDefault(), тогда форма не будет отправлена на сервер.

В примере ниже:

1. Перейдите в текстовое поле и нажмите Enter.
2. Нажмите <input type="submit">.

Оба действия показывают alert и форма не отправится благодаря return false:





## Метод: submit

Чтобы отправить форму на сервер вручную, мы можем вызвать метод form.submit().

При этом событие submit не генерируется. Предполагается, что если программист вызывает метод form.submit(), то он уже выполнил всю соответствующую обработку.

Иногда это используют для генерации формы и отправки её вручную, например так:

