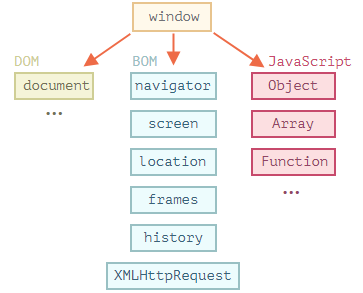
# Браузерное окружение, спецификации (Документ)

Сегодня JavaScript может использоваться в браузере, на веб-сервере или в какой-то другой среде, даже в кофеварке. Каждая среда предоставляет свою функциональность, которую спецификация JavaScript называет *окружением.* Окружение предоставляет свои объекты и дополнительные функции, в дополнение базовым языковым. Вот, что доступно для JavaScript в браузерном окружении:



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

1. Во-первых, это глобальный объект для JavaScript-кода, об этом более подробно говорится в главе Глобальный объект (в первой части).
2. Во-вторых, он также представляет собой окно браузера и располагает методами для управления им.

Например, здесь мы используем window как глобальный объект:

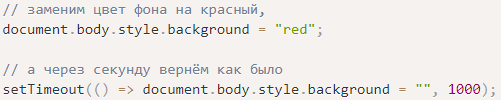


А здесь мы используем window как объект окна браузера, чтобы узнать его высоту:



## DOM (Document Object Model)

Document Object Model (DOM) – объектная модель документа, которая представляет все содержимое страницы в виде объектов, которые можно менять. Объект **document** – основная «входная точка». С его помощью мы можем что-то создавать или менять на странице, например:



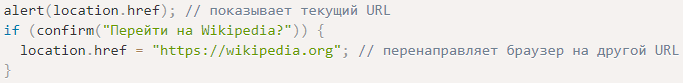
* **CSSOM для стилей -** Правила стилей CSS структурированы иначе чем HTML. Для них есть отдельная спецификация CSSOM, которая объясняет, как стили должны представляться в виде объектов, как их читать и писать. CSSOM используется вместе с DOM при изменении стилей документа. В реальности CSSOM требуется редко.

## 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** даёт информацию о самом браузере и операционной системе. Среди 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** позволяет получить текущий URL и перенаправить браузер по новому адресу:



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.

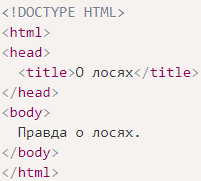
# DOM-дерево

В соответствии с объектной моделью документа (DOM), каждый HTML-тег является объектом. Вложенные теги являются «детьми» родительского элемента. Текст, который находится внутри тега, также является объектом.

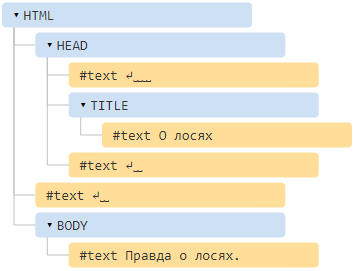
Например, **document.body** – объект для тега <body>. Если запустить этот код, то <body> станет красным на 3 секунды:



## Пример DOM (Структура)



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.

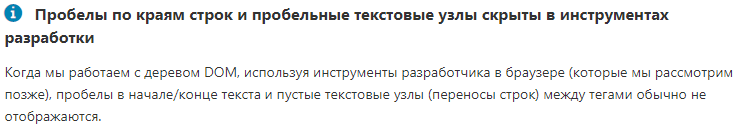
Обратите внимание на специальные символы в текстовых узлах:

* перевод строки: ↵ (в JavaScript он обозначается как \n)
* пробел: ␣

Пробелы и переводы строки – это полноправные символы, как буквы и цифры. Они образуют текстовые узлы и становятся частью дерева DOM.

Существует два исключения из этого правила:

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



## Автоисправление

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> в DOM.

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

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

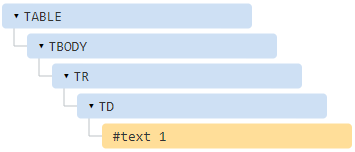
**Таблицы всегда содержат <tbody>**

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:



DOM-структура будет такой:



## Другие типы узлов

Есть и некоторые другие типы узлов, кроме элементов и текстовых узлов. Например, узел-комментарий. Зачем комментарий в DOM? Есть важное правило: если что-то есть в HTML, то оно должно быть в DOM-дереве.

Существует 12 типов узлов. Но на практике мы в основном работаем с 4 из них:

* **document** – «входная точка» в DOM.
* **узлы-элементы** – HTML-теги, основные строительные блоки.
* **текстовые узлы** – содержат текст.
* **комментарии** – иногда в них можно включить информацию, которая не будет показана, но доступна в DOM для чтения JS.

## Dev Tools

https://learn.javascript.ru/article/dom-nodes/elks.html

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. Либо правой кнопкой по элементу и «Посмотреть код».

В правой части инструментов разработчика находятся следующие подразделы:

* **Styles** – здесь мы видим CSS, применённый к текущему элементу: правило за правилом, включая встроенные стили (выделены серым). Почти всё можно отредактировать на месте, включая размеры, внешние и внутренние отступы.
* **Computed** – здесь мы видим 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 мы видим обработчики событий, привязанные к DOM-элементам (мы поговорим о них в следующей части учебника).
* … и т.д.

## Взаимодействие с консолью

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

Для начала:

1. На вкладке Elements выберите нужный элемент.
2. 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 мы можем запускать на них команды. Например $0.style.background = 'red' сделает выбранный элемент красным.

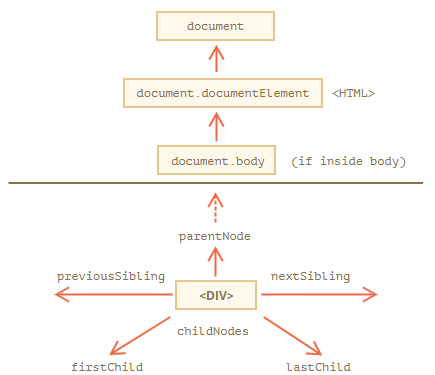
Есть и обратный путь: если есть переменная node, ссылающаяся на DOM-узел, можно использовать в консоли команду inspect(node), чтобы увидеть этот элемент во вкладке Elements.

Или мы можем просто вывести DOM-узел в консоль и исследовать «на месте», например, document.body: 

# Навигация по DOM-элементам

DOM позволяет нам 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 мы можем получить доступ к любому узлу.

Так выглядят основные ссылки, по которым можно переходить между узлами DOM:



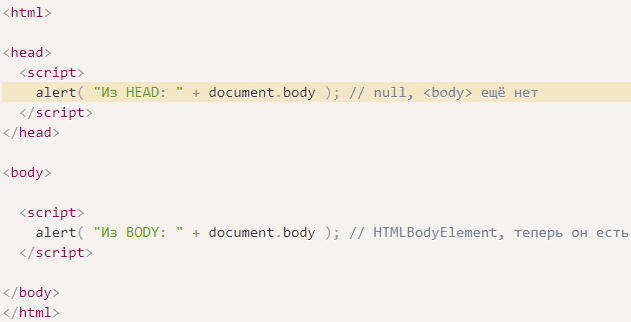
## Сверху: documentElement и body

Самые верхние элементы дерева доступны как свойства объекта document:

Самый верхний узел документа: document.documentElement. В DOM он соответствует тегу <html> (**<html> = document.documentElement)** или другой часто используемый DOM-узел – узел тега <body> (**<body> = document.body).** И, соответственно, Тег <head> доступен как document.head.

**Есть одна тонкость: document.body может быть равен null**

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

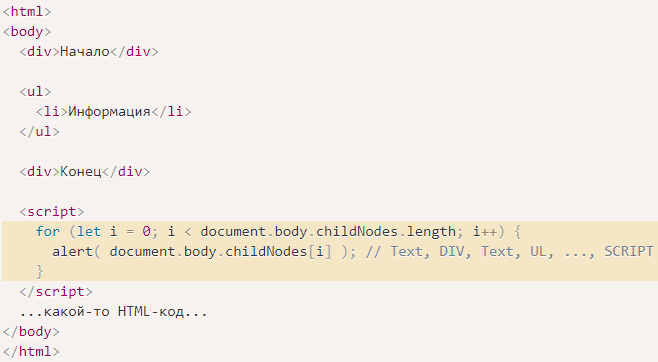
**В DOM значение null значит «не существует» или «нет такого узла».**

## Дети: childNodes, firstChild, lastChild

**Дочерние узлы** (или дети) – элементы, которые являются **непосредственными** детьми узла.

**Потомки** – все элементы, которые лежат внутри данного, включая детей, их детей и т.д.

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



*Обратим внимание на маленькую деталь. 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:



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

## DOM-коллекции

**childNodes** похож на массив. На самом деле это не массив, а **коллекция** – особый перебираемый объект-псевдомассив.

И есть два важных следствия из этого:

* Для перебора коллекции мы можем использовать for..of:



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

* Методы массивов не будут работать, потому что коллекция – это не массив. Если нам хочется использовать именно методы массива, то мы можем создать настоящий массив из коллекции, используя Array.from

**DOM-коллекции – только для чтения**

DOM-коллекции, и даже более – все навигационные свойства, перечисленные в этой главе, доступны только для чтения. Мы не можем заменить один дочерний узел на другой, просто написав childNodes[i] = .... Для изменения DOM требуются другие методы.

**DOM-коллекции живые**

Почти все DOM-коллекции, за небольшим исключением, живые. Другими словами, они отражают текущее состояние DOM. Если мы сохраним ссылку на *elem.childNodes* и добавим/удалим узлы в DOM, то они появятся в сохранённой коллекции автоматически.

 **Не используйте цикл for..in для перебора коллекций**

Коллекции перебираются циклом for..of (Symbol.iterator)

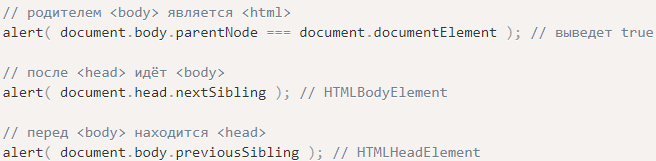
## Соседи и родитель

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



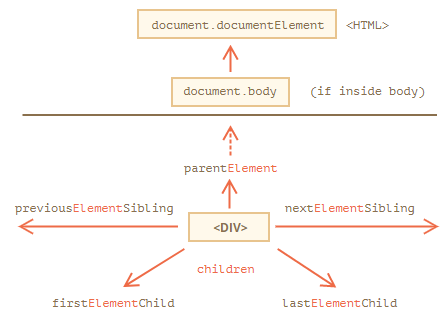
Говорят, что <body> – «следующий» или «правый» сосед <head>, также можно сказать, что <head> «предыдущий» или «левый» сосед <body>.

Следующий узел того же родителя (следующий сосед) – в свойстве **nextSibling**, а предыдущий – в **previousSibling**. Родитель доступен через **parentNode**.



## Навигация только по элементам

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. Рассмотрим дополнительный набор ссылок, которые учитывают только узлы-элементы:



Эти ссылки похожи на те, что 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.

 **Зачем нужен parentElement? Разве может родитель быть не элементом?**

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



Причина в том, что родителем корневого узла document.documentElement (<html>) является document. Но document – это не узел-элемент, так что parentNode вернёт его, а parentElement нет.

Эта деталь может быть полезна, если мы хотим пройти вверх по цепочке родителей от произвольного элемента elem к <html>, но не до document:



## Ещё немного ссылок: таблицы

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.

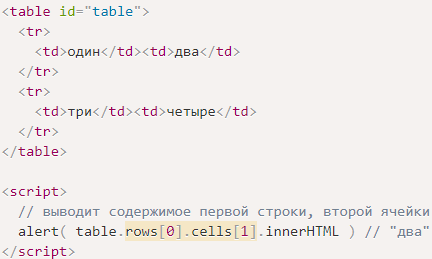
**<tr>: (***table.rows[i]***)**

* 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:



## Итого

Получив DOM-узел, мы можем перейти к его ближайшим соседям используя навигационные ссылки.

**Есть два основных набора ссылок:**

* Для всех узлов: parentNode, childNodes, firstChild, lastChild, previousSibling, nextSibling.
* *Только для узлов-элементов*: **parentElement, children, firstElementChild, lastElementChild, previousElementSibling, nextElementSibling.**

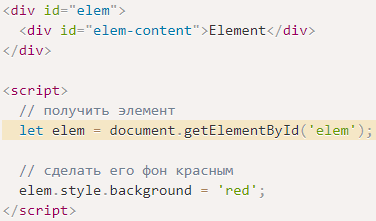
Некоторые виды DOM-элементов, например таблицы, предоставляют дополнительные ссылки и коллекции для доступа к своему содержимому.

# Поиск: getElement\*, querySelector\*

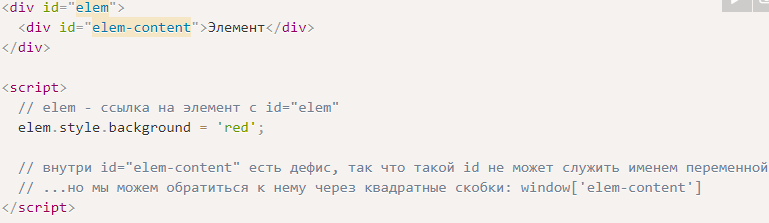
Свойства навигации по DOM хороши, когда элементы расположены рядом. А что, если нет? Как получить произвольный элемент страницы? Для этого в DOM есть дополнительные методы поиска.

## document.getElementById или просто id

Если у элемента есть атрибут id, то мы можем получить его вызовом document.getElementById(id), где бы он ни находился:



Также есть глобальная переменная с именем, указанным в id:



Но это только если мы не объявили в JavaScript переменную с таким же именем, иначе она будет иметь приоритет:



**Пожалуйста, не используйте такие глобальные переменные для доступа к элементам**

Это поведение соответствует стандарту, но поддерживается в основном для совместимости, как осколок далёкого прошлого. Браузер пытается помочь нам, смешивая пространства имён JS и DOM. Это удобно только для простых скриптов. В реальной жизни лучше использовать **document.getElementById.**

**В документе может быть только один элемент с данным id.**

**Только document.getElementById, а не anyElem.getElementById**

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

## elem.querySelectorAll

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



**Псевдоклассы тоже работают**

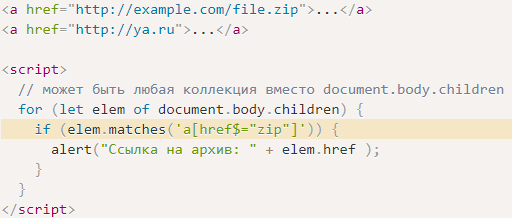
Псевдоклассы в CSS-селекторе, в частности :hover и :active, также поддерживаются. Например, *document.querySelectorAll(':hover')* вернёт коллекцию (в порядке вложенности: от внешнего к внутреннему) из текущих элементов под курсором мыши.

## querySelector

Метод **elem**.querySelector(css) возвращает первый элемент, соответствующий данному CSS-селектору. Результат такой же, как при вызове elem.querySelectorAll(css)[0], но более оптимально.

## matches

Метод **elem**.matches(css) ничего не ищет по DOM, а проверяет, удовлетворяет ли elem CSS-селектору, и возвращает true или false. Этот метод удобен, когда мы перебираем элементы (**например, в массиве или в чём-то подобном**) и пытаемся выбрать те из них, которые нас интересуют:

 (селектор атрибута) <http://htmlbook.ru/samcss/selektory-atributov> (знак $ означает, что значение атрибута должно заканчиваться соответствующим текстом)

## 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. То есть метод 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.



## 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.

Рассмотрим их для полноты картины, также вы можете встретить их в старом коде:

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

Примеры: <https://learn.javascript.ru/searching-elements-dom#getelementsby>

## Живые коллекции

<https://learn.javascript.ru/searching-elements-dom#zhivye-kollektsii>

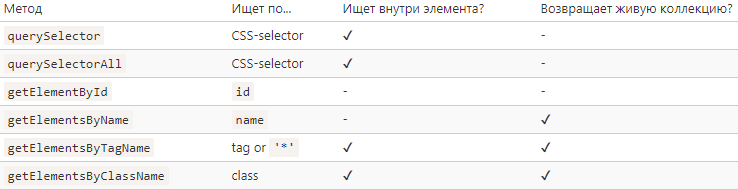
Живые коллекции всегда отражают текущее состояние документа и автоматически обновляются при его изменении.

## contains

**elemA.contains(elemB)** вернёт true, если elemB находится внутри elemA (elemB потомок elemA) или когда elemA==elemB.

## Итого

Есть 6 основных методов поиска элементов в DOM:



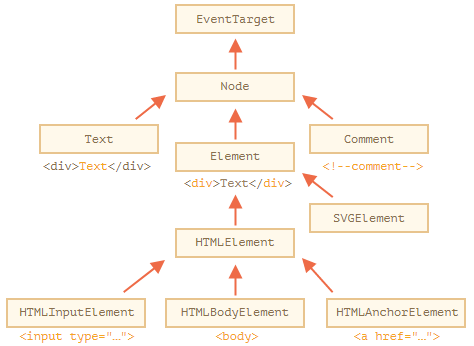
Кроме того:

* Есть метод **elem.matches(css)**, который проверяет, удовлетворяет ли элемент CSS-селектору.
* Метод **elem.closest(css)** ищет ближайшего по иерархии предка, соответствующему данному CSS-селектору. Сам элемент также включён в поиск.
* **elemA.contains(elemB)** вернёт true, если elemB находится внутри elemA (elemB потомок elemA) или когда elemA==elemB.

# Свойства узлов: тип, тег и содержимое

## Классы DOM-узлов

Каждый DOM-узел принадлежит соответствующему встроенному классу. Корнем иерархии является **EventTarget**, от него наследует **Node** и остальные DOM-узлы:



Существуют следующие классы:

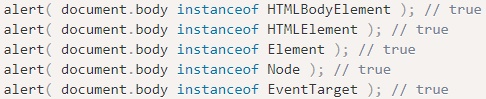
* [**EventTarget**](https://dom.spec.whatwg.org/#eventtarget) – это корневой «абстрактный» класс. Объекты этого класса никогда не создаются. Он служит основой, благодаря которой все DOM-узлы поддерживают так называемые «события», о которых мы поговорим позже.
* [**Node**](http://dom.spec.whatwg.org/#interface-node)– 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 определённые 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 определённые свойства и методы.

Так можно узнать имя класса DOM-узла:

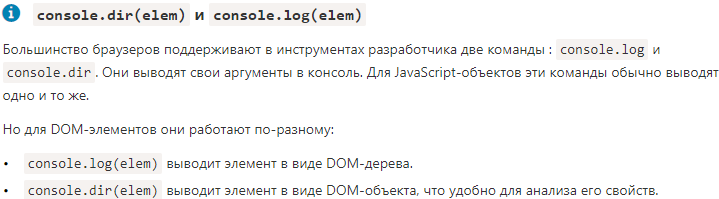


У каждого встроенного класса в свойстве constructor прототипа есть свойство name.

Или воспользоваться instanceof:



Как видно, DOM-узлы – это обычные JavaScript объекты. Для наследования они используют классы, основанные на прототипах.

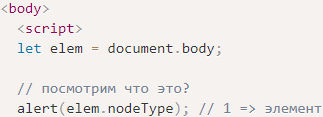


## Свойство «nodeType» (read only)

Свойство nodeType предоставляет ещё один, 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 (document)



В современном коде, чтобы узнать тип узла, можно использовать метод *instanceof* и другие способы проверить класс, но иногда *nodeType* проще использовать.

## Тег: nodeName и tagName (read only)

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

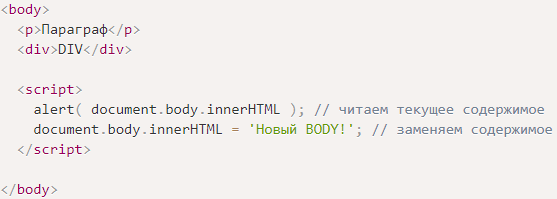


Свойство **tagName** есть только у элементов класса *Element*. Свойство **nodeName** определено для любых узлов *Node*:

* для элементов оно равно tagName.
* для остальных типов узлов (текст, комментарий и т.д.) оно содержит строку с типом узла.

## innerHTML: содержимое элемента (changeable)

Свойство innerHTML позволяет получить HTML-содержимое элемента в виде строки. Мы также можем изменять его:



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



* **Скрипты не выполнятся**

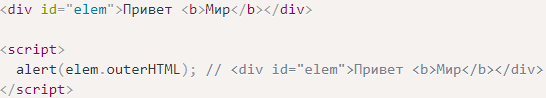
Если innerHTML вставляет в документ тег <script> – он становится частью HTML, но не запускается.

**Будьте внимательны: «innerHTML+=» осуществляет перезапись**

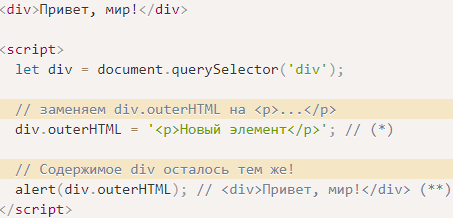
Так как содержимое «обнуляется» и переписывается заново, все изображения и другие ресурсы будут перезагружены. Из-за этого возникают неприятные побочные эффекты, к счастью, есть и другие способы добавить содержимое.

## outerHTML: HTML элемента целиком (changeable+)

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



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

****

Использование outerHTML не изменяет DOM-элемент, а удаляет его из внешнего контекста и вставляет вместо него новый HTML-код.

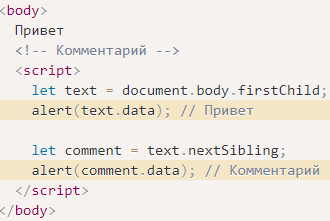
То есть, при div.outerHTML=... произошло следующее:

1. div был удалён из документа.
2. Вместо него был вставлен другой HTML <p>Новый элемент</p>.
3. В div (переменной) осталось старое значение. Новый HTML не сохранён ни в какой переменной.

Чтобы работать с новым элементом, который мы вставили, нужно получить на него новую ссылку, обратившись к DOM.

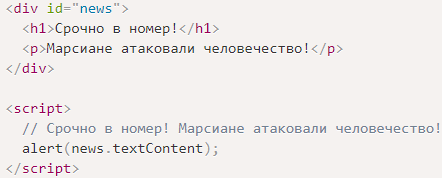
## data/nodeValue: содержимое текстового узла (changeable)

Свойство innerHTML есть только у узлов-элементов. У других типов узлов, в частности, у текстовых, есть свои аналоги. Будем использовать data, потому что оно короче:

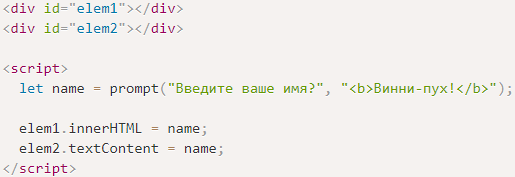


## textContent: просто текст (changeable)

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

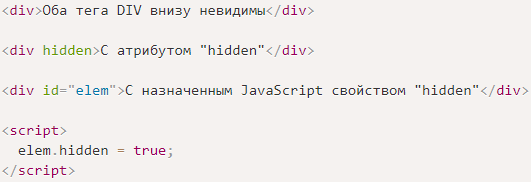


Также можно записывать текст в textContent, это позволяет писать текст «безопасным способом»:

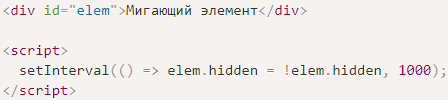
 - второй способ – безопасный, т.к. <b> прочтется просто как текст, а не как тег, пользователь не сможет натворить делов в этом случае.

## Свойство «hidden» (changeable)

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



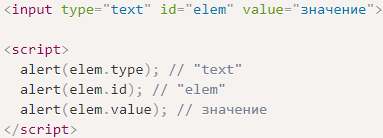
Мигающий элемент:



## Другие свойства

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…



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.

## Итого

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

Главные свойства DOM-узла:

* **nodeType**

Свойство nodeType позволяет узнать тип DOM-узла. Его значение – числовое: 1 для элементов,3 для текстовых узлов, и т.д. Только для чтения.

* **nodeName/tagName**

Для элементов это свойство возвращает название тега (записывается в верхнем регистре, за исключением XML-режима). Для узлов-неэлементов nodeName описывает, что это за узел. Только для чтения.

* **innerHTML**

Внутреннее HTML-содержимое узла-элемента. Можно изменять.

* **outerHTML**

Полный HTML узла-элемента. Запись в elem.outerHTML не меняет elem. Вместо этого она заменяет его во внешнем контексте.

* **nodeValue/data**

Содержимое узла-неэлемента (текст, комментарий). Эти свойства практически одинаковые, обычно мы используем data. Можно изменять.

* **textContent**

Текст внутри элемента: HTML за вычетом всех <тегов>. Запись в него помещает текст в элемент, при этом все специальные символы и теги интерпретируются как текст. Можно использовать для защиты от вставки произвольного HTML кода.

* **hidden**

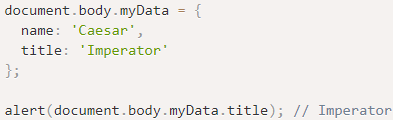
Когда значение установлено в true, делает то же самое, что и CSS display:none. (css свойство **visibility** умеет делать элемент прозрачным )

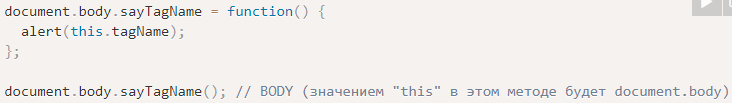
# Атрибуты и свойства

Когда браузер загружает страницу, он «парсит» HTML и генерирует из него DOM-объекты. Для узлов-элементов большинство стандартных HTML-атрибутов становятся свойствами DOM-объектов. Например, для тега <body id="page"> у DOM-объекта будет свойство body.id="page". Но преобразование атрибута в свойство может происходить не один-в-один!

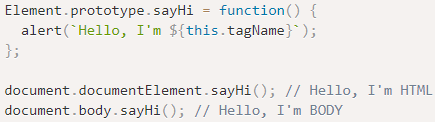
## DOM-свойства

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

 Можно и метод:

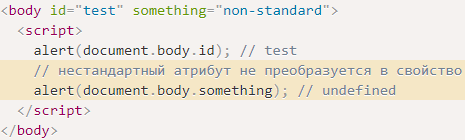


Также можно изменять встроенные прототипы, такие как **Element.prototype** и добавлять новые методы ко всем элементам:

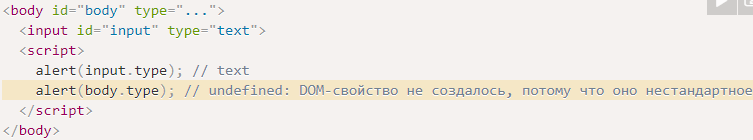


## HTML-атрибуты

Когда у элемента есть id или другой *стандартный атрибут*, создаётся соответствующее свойство. Но этого не происходит, если атрибут *нестандартный*:



стандартный атрибут для одного тега может быть нестандартным для другого. Например, атрибут "type" является стандартным для элемента <input> (HTMLInputElement), но не является стандартным для <body> (HTMLBodyElement):



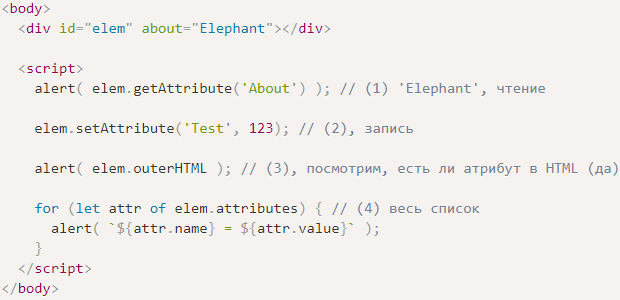
То есть для нестандартных атрибутов не будет соответствующих DOM-свойств. Но есть способ получить такие атрибуты:

* **elem.hasAttribute(name)** – проверяет наличие атрибута.
* **elem.getAttribute(name)** – получает значение атрибута.
* **elem.setAttribute(name, value)** – устанавливает значение атрибута.
* **elem.removeAttribute(name)** – удаляет атрибут.
* Получить все атрибуты элемента можно с помощью свойства **elem.attributes**: коллекция объектов, которая принадлежит ко встроенному классу Attr со свойствами name и value.

*Эти методы работают именно с тем, что написано в HTML.*

У HTML-атрибутов есть следующие особенности:

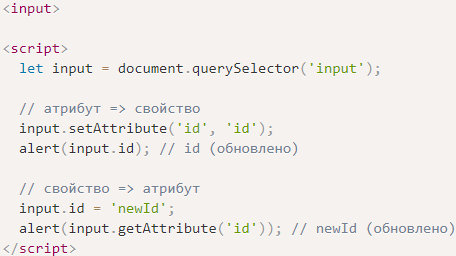
* Их имена регистронезависимы (id то же самое, что и ID).
* Их значения всегда являются строками.



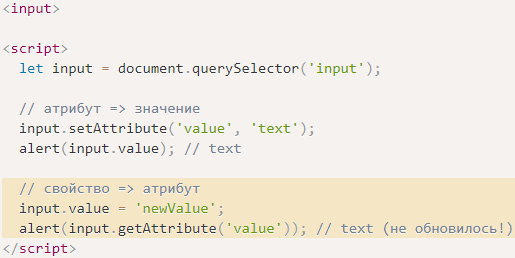
1. getAttribute('About') – здесь первая буква заглавная, а в HTML – строчная. Но это не важно: имена атрибутов регистронезависимы (приводятся к нижнему регистру).
2. Мы можем присвоить что угодно атрибуту, но это станет строкой. Поэтому в этой строчке мы получаем значение "123".
3. Все атрибуты, в том 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:

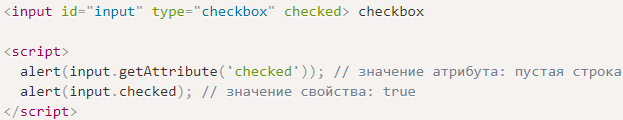


* Изменение **атрибута** value обновило свойство.
* Но изменение свойства не повлияло на атрибут.

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 Dom-свойство style является объектом:



Хотя большинство dom-свойств, всё же, строки.

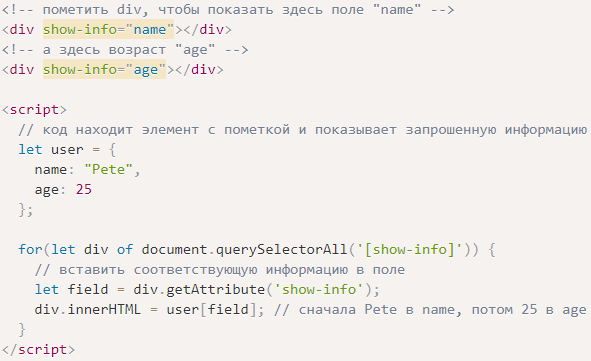
Некоторые из них, хоть и строки, могут отличаться от атрибутов. Например, DOM-свойство href всегда содержит полный URL, даже если атрибут содержит относительный URL или просто #hash:



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

## Нестандартные атрибуты, dataset

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

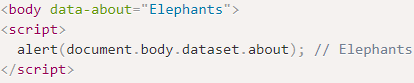


* Но с пользовательскими атрибутами могут возникнуть проблемы. Что если мы используем нестандартный атрибут для наших целей, а позже он появится в стандарте и будет выполнять какую-то функцию?

**Чтобы избежать конфликтов, существуют атрибуты вида data-\*.**

*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:



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

**Вот как можно управлять стилями с помощью атрибутов:**

*?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?:*

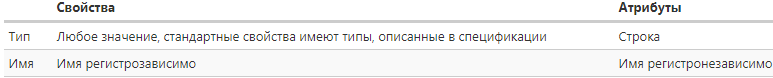


## Итого

* Атрибуты – это то, что написано в HTML.
* Свойства – это то, что находится в DOM-объектах.

Методы для работы с атрибутами:

* elem.hasAttribute(name) – проверить на наличие.
* elem.getAttribute(name) – получить значение.
* elem.setAttribute(name, value) – установить значение.
* elem.removeAttribute(name) – удалить атрибут.
* elem.attributes – это коллекция всех атрибутов.



В большинстве ситуаций предпочтительнее использовать DOM-свойства. Нужно использовать атрибуты только тогда, когда DOM-свойства не подходят, когда нужны именно атрибуты, например:

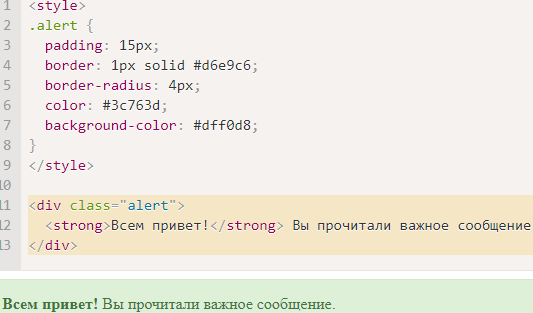
* Нужен нестандартный атрибут. Но если он начинается с data-, тогда нужно использовать dataset.
* Мы хотим получить именно то значение, которое написано в HTML. Значение DOM-свойства может быть другим, например, свойство href – всегда полный URL, а нам может понадобиться получить «оригинальное» значение.

# Изменение документа

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:



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

## Создание элемента

**document.createElement(tag)** - Создаёт новый *элемент* с заданным тегом:



**document.createTextNode(text)** - Создаёт новый *текстовый узел* с заданным текстом:



## Создание сообщения

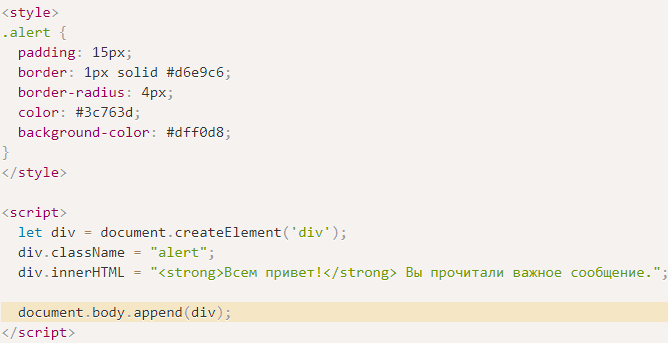
В нашем случае сообщение – это div с классом alert и HTML в нём:



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

## Методы вставки

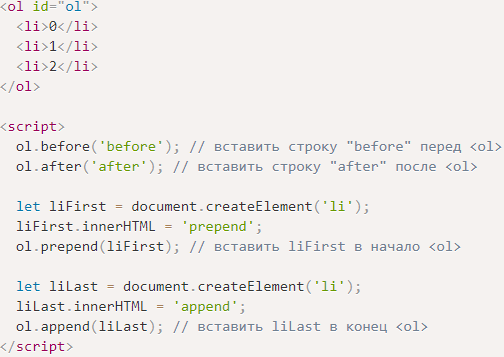
Чтобы наш div появился, нам нужно вставить его, например, в document.body с помощью метода **append**, в нашем случае: **document.body.append(div)**:



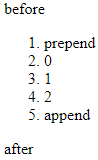
Вот методы для различных вариантов вставки:

* **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.

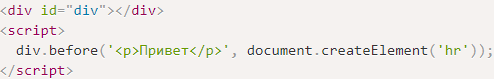
Примеры использования этих методов:



Результат:



Эти методы **могут вставлять несколько узлов** и текстовых фрагментов за вызов:



строки вставляются безопасным способом, как делает это elem.textContent, поэтому финальный HTML будет:



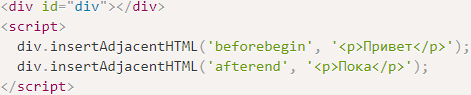
## insertAdjacentHTML/Text/Element

***elem.insertAdjacentHTML(where, html)*** - если мы хотим вставить HTML именно «как html», со всеми тегами и прочим, как делает это *elem.innerHTML*

Первый параметр – это специальное слово, указывающее, куда по отношению к elem производить вставку:

* "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»:

 Приведет к:



У этого 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 **ОДИН** элемент elem.

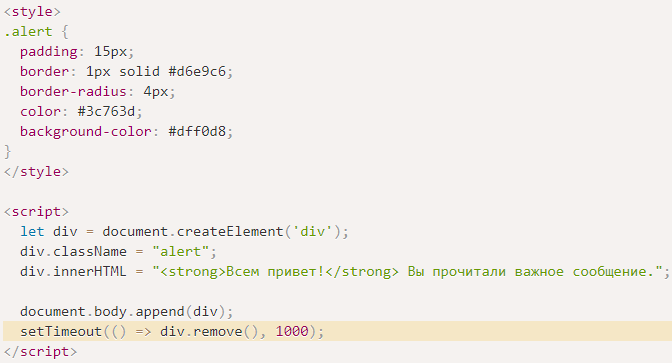
Они существуют, в основном, чтобы унифицировать синтаксис. На практике часто используется только insertAdjacentHTML. Потому что для элементов и текста у нас есть методы append/prepend/before/after – их быстрее написать, и они могут вставлять как узлы, так и текст.

альтернативный вариант показа сообщения:

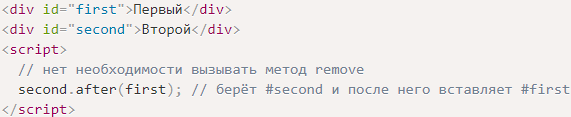


## Удаление узлов

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:**

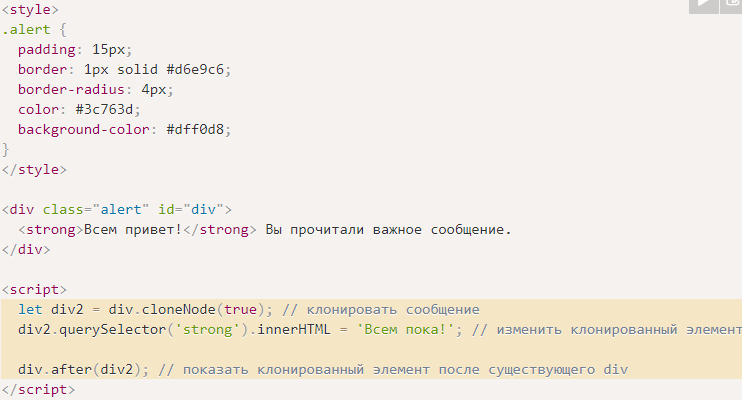


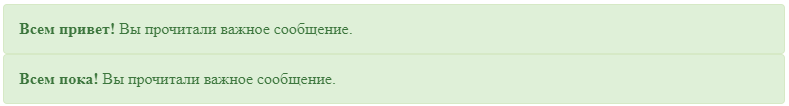
## Клонирование узлов: cloneNode

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

***elem.cloneNode(false)***, тогда клон будет без дочерних элементов.

Пример:



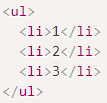


## DocumentFragment

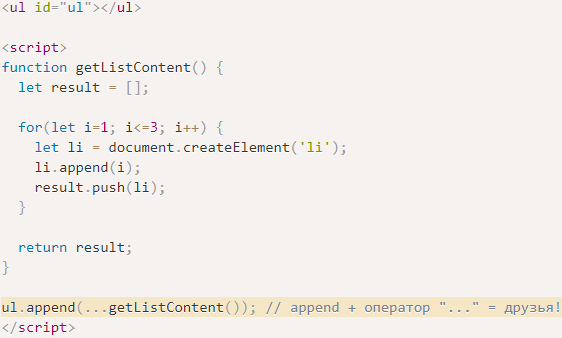
DocumentFragment является специальным DOM-узлом, который служит обёрткой для передачи списков узлов. Мы можем добавить к нему другие узлы, но когда мы вставляем его куда-то, он «исчезает», вместо него вставляется его содержимое.

Например, getListContent ниже генерирует фрагмент с элементами <li>, которые позже вставляются в <ul>:

 Итог:



DocumentFragment редко используется. Вместо него можно вернуть массив узлов:



## Устаревшие методы вставки/удаления

<https://learn.javascript.ru/modifying-document#ustarevshie-metody-vstavki-udaleniya>

## Несколько слов о «document.write» (legacy)

<https://learn.javascript.ru/modifying-document#neskolko-slov-o-document-write>

## Итого

* Методы для создания узлов:
  + document.createElement(tag) – создаёт элемент с заданным тегом,
  + document.createTextNode(value) – создаёт текстовый узел (редко используется),
  + elem.cloneNode(deep) – клонирует элемент, если deep==true, то со всеми дочерними элементами.
* Вставка и удаление:
  + node.append(...nodes or strings) – вставляет в node в конец,
  + node.prepend(...nodes or strings) – вставляет в node в начало,
  + node.before(...nodes or strings) – вставляет прямо перед node,
  + node.after(...nodes or strings) – вставляет сразу после node,
  + node.replaceWith(...nodes or strings) – заменяет node.
  + node.remove() – удаляет node.
* Если нужно вставить фрагмент HTML, то elem.insertAdjacentHTML(where, html) вставляет в зависимости от where:
  + "beforebegin" – вставляет html прямо перед elem,
  + "afterbegin" – вставляет html в elem в начало,
  + "beforeend" – вставляет html в elem в конец,
  + "afterend" – вставляет html сразу после elem.

Также существуют похожие методы elem.insertAdjacentText и elem.insertAdjacentElement, они вставляют текстовые строки и элементы, но они редко используются.

* Чтобы добавить HTML на страницу до завершения её загрузки:
  + document.write(html)

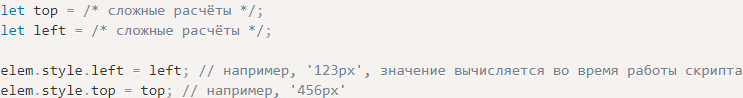
# Стили и классы

Как правило, существует два способа задания стилей для элемента:

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

JavaScript может менять и классы, и style.

Классы – предпочтительный вариант нежели style. Можно манипулировать свойством style только если классы «не могут справиться». Например, использование style приемлимо, если мы вычисляем координаты элемента динамически и хотим установить их из JavaScript:

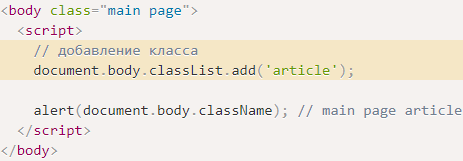


В других случаях, например, чтобы сделать текст красным, добавить значок фона – описываем это в CSS и добавляем класс (JavaScript может это сделать). Это более гибкое и лёгкое в поддержке решение.

## className и classList

Свойство ***elem.className*** соответствует атрибуту "class".

Если мы присваиваем что-то *elem.className*, то это заменяет всю строку с классами. Но часто мы хотим добавить/удалить один класс. Для этого есть другое свойство: **elem.classList** - это специальный объект с методами для добавления/удаления одного класса. Например:



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.

*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.

* Свойства с префиксом

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

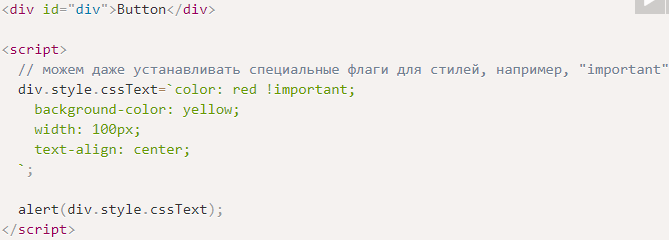


## Сброс стилей

Вместо *delete elem.style.display* мы должны присвоить ему пустую строку: *elem.style.display = ""*. 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.

* Полная перезапись **style.cssText**

Нельзя установить список стилей как, например, div.style="color: red; width: 100px", потому что div.style – это объект со свойствами, который нельзя перезаписать целиком. Для задания нескольких стилей в одной строке используется специальное свойство **style.cssText:**



Это свойство редко используется, потому что такое присваивание удаляет все существующие стили. То же самое можно сделать установкой атрибута: *div.setAttribute('style', 'color: red...').*

## Следите за единицами измерения

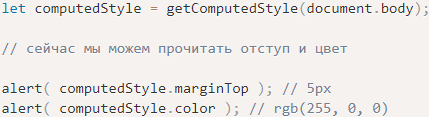
<https://learn.javascript.ru/styles-and-classes#sledite-za-edinitsami-izmereniya>

## Вычисленные стили: getComputedStyle

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

**getComputedStyle(element, [pseudo])** - Результат вызова – объект со стилями, похожий на elem.style, но с учётом всех CSS-классов. *pseudo -* указывается, если нужен стиль псевдоэлемента, например ::before. Пустая строка или отсутствие аргумента означают сам элемент.

Например:



* Вычисленное (computed) и окончательное (resolved) значения

Есть две концепции в CSS:

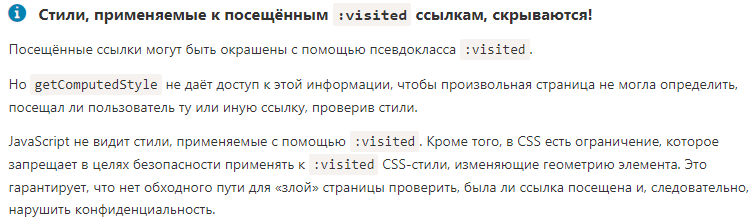
* **Вычисленное (computed) значение** – это то, которое получено после применения всех CSS-правил и CSS-наследования. Например, height:1em или font-size:125%.
* **Окончательное (resolved) значение** – непосредственно применяемое к элементу. Значения 1em или 125% являются относительными. Браузер берёт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

Давным-давно getComputedStyle был создан для получения вычисленных значений, но оказалось, что окончательные значения гораздо удобнее, и стандарт изменился.

Так что, в настоящее время getComputedStyle фактически возвращает окончательное значение свойства, для геометрии оно обычно в пикселях.

* getComputedStyle требует полное свойство!

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



## Итого

Для управления классами существуют два DOM-свойства:

* className – строковое значение, удобно для управления всем набором классов.
* classList – объект с методами add/remove/toggle/contains, удобно для управления отдельными классами.

Чтобы изменить стили:

* Свойство style является объектом со стилями в формате camelCase. Чтение и запись в него работают так же, как изменение соответствующих свойств в атрибуте "style". Чтобы узнать, как добавить в него important и делать некоторые другие редкие вещи – смотрите [документацию](https://developer.mozilla.org/ru/docs/Web/API/CSSStyleDeclaration).
* Свойство style.cssText соответствует всему атрибуту "style", полной строке стилей.

Для чтения окончательных стилей (с учётом всех классов, после применения CSS и вычисления окончательных значений) используется:

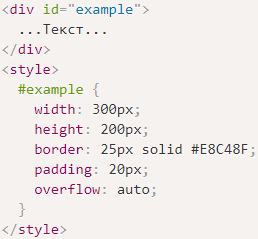
* Метод getComputedStyle(elem, [pseudo]) возвращает объект, похожий по формату на style. Только для чтения.

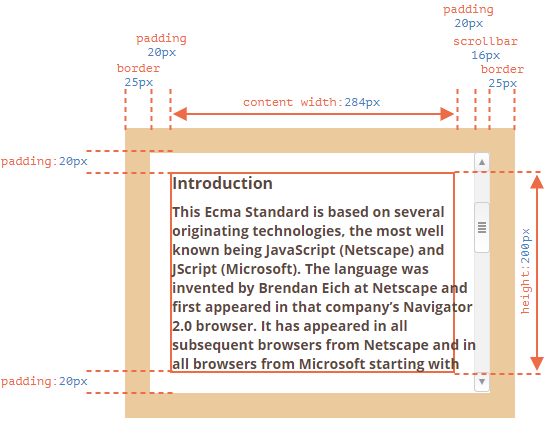
# Размеры и прокрутка элементов

Существует множество JavaScript-свойств, которые позволяют считывать информацию об элементе: ширину, высоту и другие геометрические характеристики. В этой главе мы будем называть их «**метрики**».

## Простой пример

В качестве примера демонстрации будем использовать следующий элемент:

 Элемент имеет рамку (border), внутренний отступ (padding) и прокрутку. 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:



* **Внимание, полоса прокрутки**

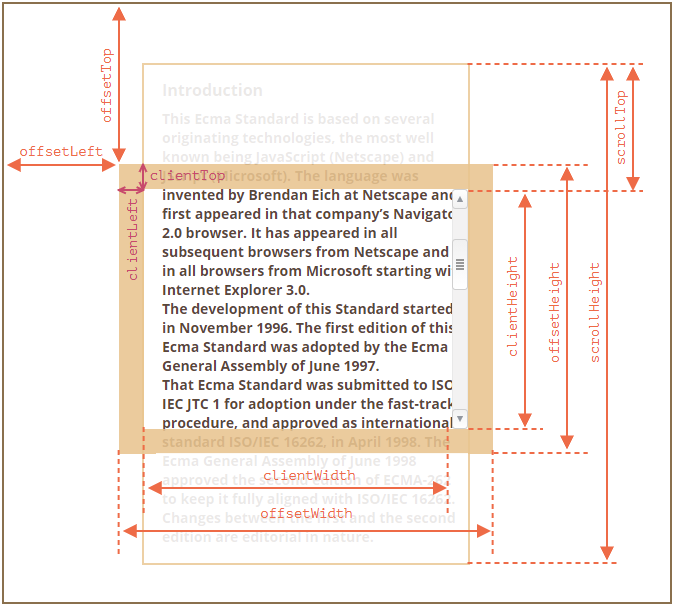
Некоторые браузеры (не все) отбирают место для полосы прокрутки, забирая его у области, отведённой для содержимого (помечена как «content width» выше). Таким образом, без учёта полосы прокрутки ширина области содержимого (content width) будет 300px, но если предположить, что ширина полосы прокрутки равна 16px (её точное значение зависит от устройства и браузера), тогда остаётся только 300 - 16 = 284px

* **Область padding-bottom (нижний внутренний отступ) может быть заполнена текстом**

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

## Метрики

Общая картина с геометрическими свойствами:



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

## offsetParent, offsetLeft/Top

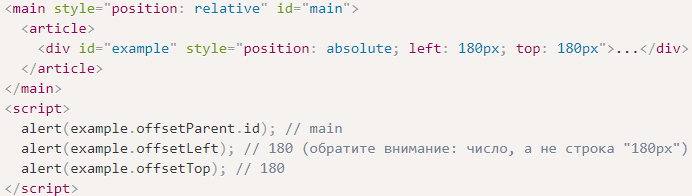
Эти свойства редко используются, но так как они являются «самыми внешними» метриками, мы начнём с них.

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

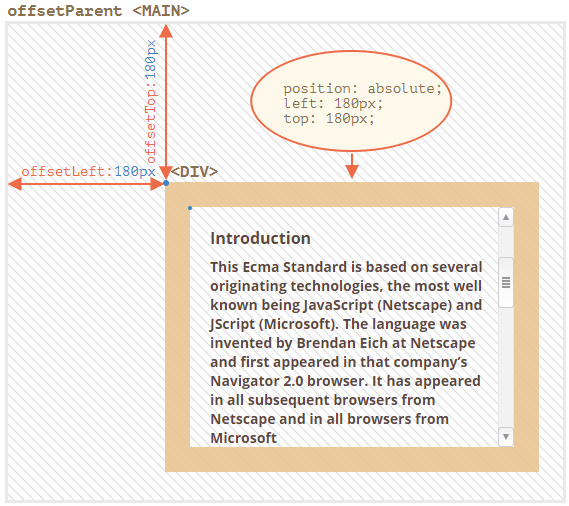
1. Является CSS-позиционированным (CSS-свойство position равно absolute, relative, fixed или sticky),
2. или <td>, <th>, <table>,
3. или <body>.

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):



*Если у родителя значение position задано как fixed, relative или absolute, то при position: absolute, отсчет координат ведется от края родительского элемента.*



**Координаты отсчитываются от *внутреннего* верхнего левого угла родителя**

Существует несколько ситуаций, когда offsetParent равно null:

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

## offsetWidth/Height (полный размер)

Содержат «внешнюю» ширину/высоту элемента, то есть его полный размер, включая рамки и паддинги.

* Метрики для не показываемых элементов равны нулю.

Координаты и размеры в JS 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 мы создали элемент, но ещё не вставили его в документ, или если у элемента (или у его родителя) display:none.

Мы можем использовать это, чтобы делать проверку на видимость:

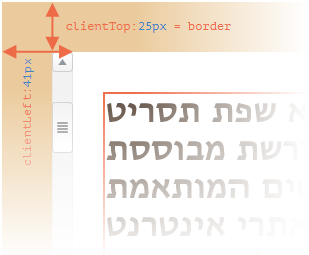


Заметим, что функция isHidden также вернёт true для элементов, которые в принципе показываются, но их размеры равны нулю (например, пустые <div>).

## clientTop/Left (рамки)

Это свойства для рамок (border)

Отступы внутренней части элемента от внешней. Документ может располагается справа налево (операционная система на арабском языке или иврите) тогда полоса прокрутки в этом случае находится слева, и тогда свойство **clientLeft** включает в себя ещё и ширину полосы прокрутки:

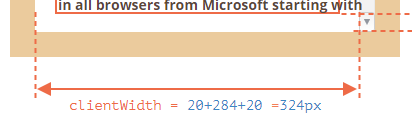


В обычной ситуации – это ширина рамки

## clientWidth/Height (content + padding)

Эти свойства – размер области внутри рамок элемента.

Они включают в себя ширину области содержимого вместе с внутренними отступами padding, но без прокрутки:

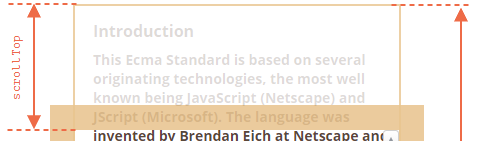


## scrollWidth/Height

Эти свойства – как clientWidth/clientHeight, но также включают в себя прокрученную (которую не видно) часть элемента.

## scrollLeft/scrollTop

Свойства scrollLeft/scrollTop – ширина/высота невидимой, прокрученной в данный момент, части элемента слева и сверху. Другими словами, свойство scrollTop – это «сколько уже прокручено вверх»:



* **Свойства scrollLeft/scrollTop можно изменять**

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.

## Не стоит брать width/height из CSS

<https://learn.javascript.ru/size-and-scroll#ne-stoit-brat-width-height-iz-css>

## Итого

Все свойства доступны только для чтения, кроме scrollLeft/scrollTop, изменение которых заставляет браузер прокручивать элемент.

# Размеры и прокрутка окна

## Ширина/высота окна

**window.innerWidth** – ширина окна вместе с полосой прокрутки

Чтобы получить ширину/высоту окна, можно взять свойства **clientWidth/clientHeight** из **document.documentElement** (HTML).

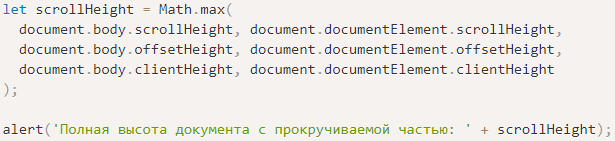
* **DOCTYPE – это важно**

Обратите внимание, что геометрические свойства верхнего уровня могут работать немного иначе, если в HTML нет *<!DOCTYPE HTML>.* Возможны странности.

## Ширина/высота документа (всего)

<https://learn.javascript.ru/size-and-scroll-window>

Чтобы надёжно получить полную высоту документа, нам следует взять максимальное из этих свойств:



Почему? Лучше не спрашивайте. Эти несоответствия идут с древних времён. Глубокой логики здесь нет.

## Получение текущей прокрутки документа

Надежнее так, чем *elem.scrollLeft/scrollTop*:



Эти свойства доступны только для чтения.

## Прокрутка: scrollTo, scrollBy, scrollIntoView

* **Для прокрутки страницы из JavaScript её DOM должен быть полностью построен.**

Например, если мы попытаемся прокрутить страницу из скрипта в <head>, это не сработает.

Обычные элементы можно прокручивать, изменяя scrollTop/scrollLeft. Мы можем 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 и **window.scrollTo(pageX,pageY)**.

* **scrollBy(x,y)** прокручивает страницу относительно её текущего положения. Например, scrollBy(0,10) прокручивает страницу на 10px вниз.
* **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 мы можем использовать scrollTo(0,0).

## 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.

## Запретить прокрутку

Иногда нам 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"**.

Аналогичным образом мы можем «заморозить» 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.

## Итого

Размеры:

**window.innerWidth** – ширина окна вместе с полосой прокрутки

* Ширина/высота видимой части документа (ширина/высота области содержимого): document.documentElement.clientWidth/Height
* Ширина/высота всего документа с прокрученной частью:
* let scrollHeight = Math.max(
* document.body.scrollHeight, document.documentElement.scrollHeight,
* document.body.offsetHeight, document.documentElement.offsetHeight,
* document.body.clientHeight, document.documentElement.clientHeight

);

Прокрутка:

* Прокрутку окна можно получить так: window.pageYOffset/pageXOffset.
* Изменить текущую прокрутку:
  + window.scrollTo(pageX,pageY) – абсолютные координаты,
  + window.scrollBy(x,y) – прокрутка относительно текущего места,
  + elem.scrollIntoView(top) – прокрутить страницу так, чтобы сделать elem видимым (выровнять относительно верхней/нижней части окна).

# Координаты

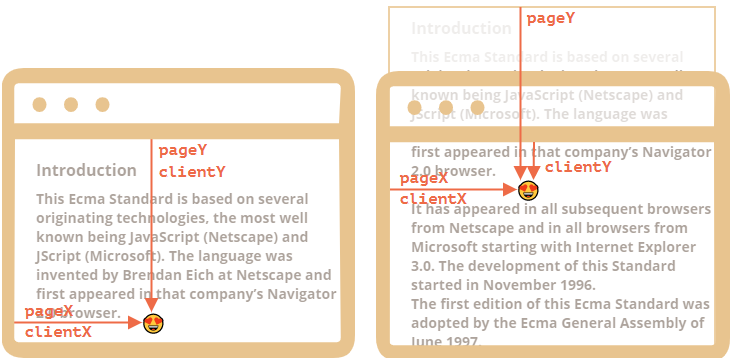
Большинство соответствующих методов JavaScript работают в одной из двух указанных ниже систем координат:

* **Относительно окна браузера** – как в position:fixed, отсчёт идёт от верхнего левого угла окна. Мы будем обозначать эти координаты как ***clientX/clientY***
* **Относительно документа** – как в position:absolute на уровне документа, отсчёт идёт от верхнего левого угла документа. мы 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.*

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

**elem.getBoundingClientRect()** возвращает координаты в контексте окна для минимального по размеру прямоугольника, который заключает в себе элемент elem, в виде объекта встроенного класса DOMRect.

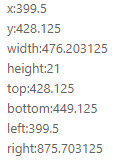
*Основные свойства объекта типа DOMRect:*

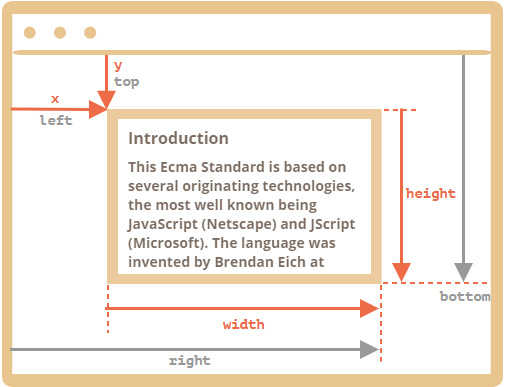
* **x/y – X/Y**-координаты начала прямоугольника относительно окна,
* **width/height** – ширина/высота прямоугольника (могут быть отрицательными).

*Дополнительные, «зависимые», свойства:*

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

Пример:



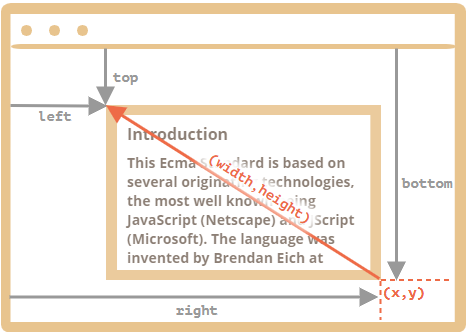


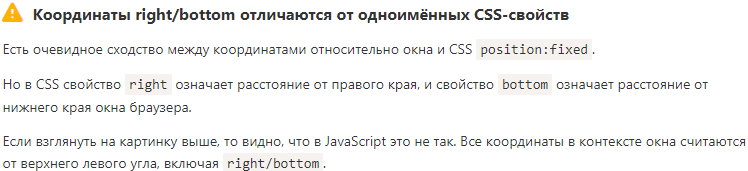
Координаты могут быть отрицательными. Например, если страница прокручена так, что элемент elem ушёл вверх за пределы окна, то вызов elem.getBoundingClientRect().top вернёт отрицательное значение.

* *Зачем вообще нужны зависимые свойства? Для чего существуют top/left, если есть x/y?*

top/left, то они на самом деле не всегда равны x/y. Технически, значения width/height могут быть отрицательными. Это позволяет задать «направленный» прямоугольник, например, для выделения мышью с отмеченным началом и концом. То есть, отрицательные значения width/height означают, что прямоугольник «растет» влево-вверх из правого угла.

Вот прямоугольник с отрицательными width и height (например, width=-200, height=-100):





## elementFromPoint(x, y)

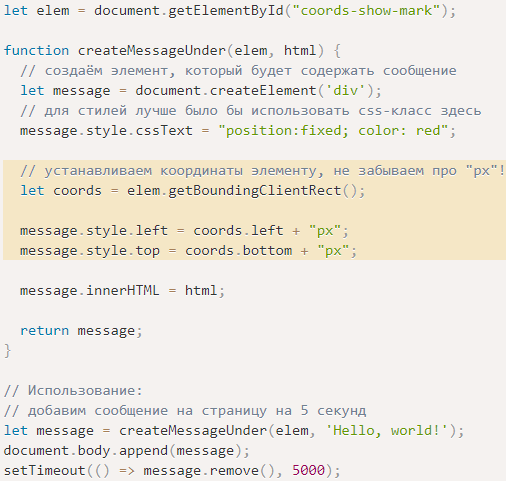
**document.elementFromPoint(x, y)** возвращает самый глубоко вложенный элемент в окне, находящийся по координатам (x, y).

* Для координат за пределами окна метод elementFromPoint возвращает null

## Применение для fixed позиционирования

Чаще всего нам нужны координаты для позиционирования чего-либо.

Чтобы показать что-то около нужного элемента, мы можем 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:



сообщение позиционируется с помощью position:fixed, поэтому оно остаётся всегда на том же самом месте в окне при прокрутке страницы.

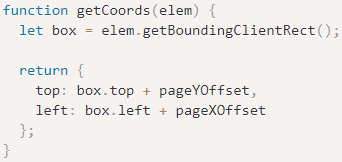
## Координаты относительно документа

В CSS координаты относительно документа соответствуют свойству position:absolute на самом верхнем уровне вложенности. Мы можем воспользоваться свойствами position:absolute и top/left, чтобы привязать что-нибудь к конкретному месту в документе. Но сначала нужно получить верные координаты.

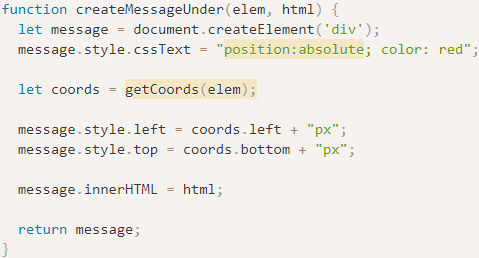
Не существует стандартного метода, который возвращал бы координаты элемента относительно документа, но мы можем написать его сами:

* 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:



Модифицированная функция createMessageUnder:



## Итого

Любая точка на странице имеет координаты:

* Относительно окна браузера – elem.getBoundingClientRect().
* Относительно документа – elem.getBoundingClientRect() плюс текущая прокрутка страницы.

Координаты в контексте окна подходят для использования с position:fixed, а координаты относительно документа – для использования с position:absolute.

Иногда будет лучше применить одну систему координат, а иногда – другую, как это и происходит с позиционированием в CSS, где мы выбираем между absolute и fixed.

# Введение в браузерные события (Введение в события)

Событие – это сигнал от браузера о том, что что-то произошло. Все DOM-узлы подают такие сигналы (хотя события бывают и не только в DOM).

**События мыши:**

* **click** – происходит, когда кликнули на элемент левой кнопкой мыши (на устройствах с сенсорными экранами оно происходит при касании).
* **contextmenu** – происходит, когда кликнули на элемент правой кнопкой мыши.
* **mouseover / mouseout** – когда мышь наводится на / покидает элемент.
* **mousedown / mouseup** – когда нажали / отжали кнопку мыши на элементе.
* **mousemove** – при движении мыши.

**События на элементах управления:**

* **submit** – пользователь отправил форму <form>.
* **focus** – пользователь фокусируется на элементе, например нажимает на <input>.

**Клавиатурные события:**

* **keydown** и **keyup** – когда пользователь нажимает / отпускает клавишу.

**События документа**:

* **СDOMContentLoaded –** когда HTML загружен и обработан, DOM документа полностью построен и доступен.

**CSS events:**

* **transitionend –** когда CSS-анимация завершена.

Существует множество других событий.

## Обработчики событий, способы назначения

Это функция, которая сработает, как только событие произошло.

* **Использование атрибута HTML**

Обработчик может быть назначен прямо в разметке, в атрибуте, который называется *on<событие>*. Например, чтобы назначить обработчик события click на элементе input, можно использовать атрибут onclick, вот так:



Атрибут HTML-тега – не самое удобное место для написания большого количества кода, поэтому лучше создать отдельную JavaScript-функцию и вызвать её там.

* **Использование свойства DOM-объекта**

Можно назначать обработчик, используя свойство DOM-элемента on<событие>:



Если обработчик задан через атрибут, то браузер читает HTML-разметку, создаёт новую функцию из содержимого атрибута и записывает в свойство. Обработчик всегда хранится в свойстве DOM-объекта, а атрибут – лишь один из способов его инициализации.

*Удалить такой обработчик можно назначением elem.onclick = null.*

## Доступ к элементу через this

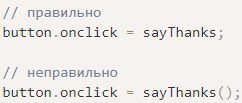
Внутри обработчика события, this ссылается на текущий элемент, то есть на тот, на котором, как говорят, «висит» (т.е. назначен) обработчик.

В коде ниже button выводит своё содержимое, используя this.innerHTML:



## Частые ошибки

Когда обработчику присваивается готовая функция, она должна быть присвоена как sayThanks, а не sayThanks():



А вот в разметке, в отличие от свойства, скобки нужны:



Потому что при создании обработчика браузером из атрибута, он автоматически создаёт функцию с телом из значения атрибута: sayThanks():



**Не используйте setAttribute для обработчиков.** Так не сработает:



**Регистр**

OnClicK в разметке не чувствителен к регистру, а DOM – свойство и **addEventListener** – да

## addEventListener

Фундаментальный недостаток описанных выше способов назначения обработчика – невозможность повесить несколько обработчиков на одно событие.

Поэтому существует способ назначения обработчиков при помощи специальных методов **addEventListener** и **removeEventListener**. Они свободны от указанного недостатка.

**Синтаксис:**



* **event -** Имя события, например "click".
* **handler -** Ссылка на функцию-обработчик.
* **options -** Дополнительный объект со свойствами:

*once:* если true, тогда обработчик будет автоматически удалён после выполнения.

*capture:* фаза, на которой должен сработать обработчик, подробнее об этом будет рассказано в главе Всплытие и погружение. Так исторически сложилось, что options может быть false/true, это тоже самое, что {capture: false/true}.

*passive:* если true, то указывает, что обработчик никогда не вызовет preventDefault(), подробнее об этом будет рассказано в главе «Действия браузера по умолчанию».

Для удаления обработчика следует использовать **removeEventListener**:



* *Удаление требует именно ту же функцию*

Нужно передать ссылку на ту же функцию, если передать анонимную функцию с тем же кодом – не получится. Если функцию-обработчик не сохранить где-либо, мы не сможем её удалить. Нет метода, который позволяет получить из элемента обработчики событий, назначенные через addEventListener.

Метод **addEventListener** позволяет добавлять несколько обработчиков на одно событие одного элемента:

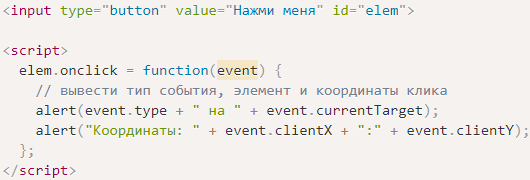


* *Обработчики некоторых событий можно назначать только через addEventListener*

Но их совсем немного

## Объект события

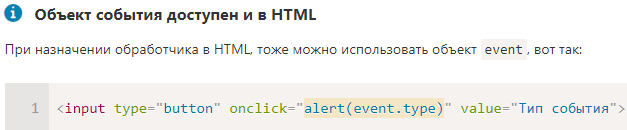
Чтобы хорошо обработать событие, могут понадобиться детали того, что произошло. Не просто «клик» или «нажатие клавиши», а также – какие координаты указателя мыши, какая клавиша нажата и так далее. Когда происходит событие, браузер создаёт объект события, записывает в него детали и передаёт его в качестве аргумента функции-обработчику:



Некоторые свойства объекта event:

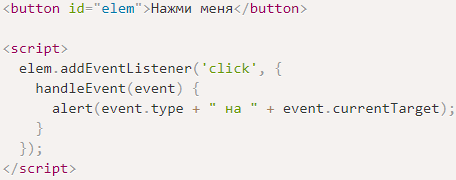
* **event.type -** тип события, в данном случае "click".
* **event.currentTarget -** элемент, на котором сработал обработчик. Значение – обычно такое же, как и у this, но если обработчик является функцией-стрелкой или при помощи bind привязан другой объект в качестве this, то мы можем получить элемент из event.currentTarget.
* **event.clientX / event.clientY -** координаты курсора в момент клика относительно окна, для событий мыши.
* Если назначили на таблицу обработчик и кликнули на ее ячейку – **e.target** будет ячейка, а **e.currentTarget** – таблица. То есть **e.currentTarget –** это элемент, на котором висит текущий обработчик

**Не важно, как вы назначаете обработчик – он получает объект события первым аргументом. Этот объект содержит подробности о том, что произошло.**



## Объект-обработчик: handleEvent

Мы можем назначить обработчиком не только функцию, но и объект при помощи addEventListener. В этом случае, когда происходит событие, вызывается метод объекта **handleEvent**:



Также можем использовать класс:



Метод handleEvent не обязательно должен выполнять всю работу сам. Он может вызывать другие методы, которые заточены под обработку конкретных типов событий, вот так:



## Итого

Есть три способа назначения обработчиков событий:

* Атрибут HTML: onclick="...".
* DOM-свойство: elem.onclick = function.
* Специальные методы: elem.addEventListener(event, handler[, phase]) для добавления, removeEventListener для удаления.

# Всплытие и погружение

Этот обработчик для <div> сработает, если вы кликните по любому из вложенных тегов, будь то <em> или <code>:



## Всплытие

Когда на элементе происходит событие, обработчики сначала срабатывают на нём, потом на его родителе, затем выше и так далее, вверх по цепочке предков. Этот процесс называется «всплытием», потому что события «всплывают» от внутреннего элемента вверх через родителей.

* *Почти все события всплывают*.

Например, событие focus не всплывает. В дальнейшем мы увидим и другие примеры. Однако, стоит понимать, что это скорее исключение.

## event.target

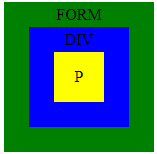
Самый глубокий элемент, который вызывает событие, называется целевым элементом, и он доступен через event.target.

Отличия от **this** (=**event.currentTarget**):

* **event.target** – это «целевой» элемент, на котором произошло событие, в процессе всплытия он неизменен.
* **This (event.currentTarget**): – это «текущий» элемент, до которого дошло всплытие, на нём сейчас выполняется (висит) обработчик.

Например, если стоит только один обработчик form.onclick на форме, то он «поймает» все клики внутри формы. Где бы ни был клик внутри – он всплывёт до элемента <form>, на котором сработает обработчик:





## Прекращение всплытия

Всплытие идёт с «целевого» элемента прямо наверх. По умолчанию событие будет всплывать до элемента <html>, а затем до объекта document, а иногда даже до window, вызывая все обработчики на своём пути. Но любой промежуточный обработчик может решить, что событие полностью обработано, и остановить всплытие. Для этого нужно вызвать метод **event.stopPropagation()**:



* *event.stopImmediatePropagation()*

event.stopPropagation() препятствует продвижению события дальше, но на текущем элементе все обработчики будут вызваны. Для того, чтобы полностью остановить обработку, существует метод **event.stopImmediatePropagation()**. Он не только предотвращает всплытие, но и останавливает обработку событий на текущем элементе.

* *Не прекращайте всплытие без необходимости!*

## Погружение

Очень редко используется в реальном коде, однако тоже может быть полезным.

<https://learn.javascript.ru/bubbling-and-capturing#pogruzhenie>

## Итого

При наступлении события – самый глубоко вложенный элемент, на котором оно произошло, помечается как «целевой» (event.target).

* Затем событие сначала двигается вниз от корня документа к event.target, по пути вызывая обработчики, поставленные через addEventListener(...., true), где true – это сокращение для {capture: true}.
* Далее обработчики вызываются на целевом элементе.
* Далее событие двигается от event.target вверх к корню документа, по пути вызывая обработчики, поставленные через on<event> и addEventListener без третьего аргумента или с третьим аргументом равным false.

Каждый обработчик имеет доступ к свойствам события event:

* event.target – самый глубокий элемент, на котором произошло событие.
* event.currentTarget (=this) – элемент, на котором в данный момент сработал обработчик (тот, на котором «висит» конкретный обработчик)
* event.eventPhase – на какой фазе он сработал (погружение=1, фаза цели=2, всплытие=3).

Любой обработчик может остановить событие вызовом event.stopPropagation(), но делать это не рекомендуется, так как в дальнейшем это событие может понадобиться, иногда для самых неожиданных вещей.

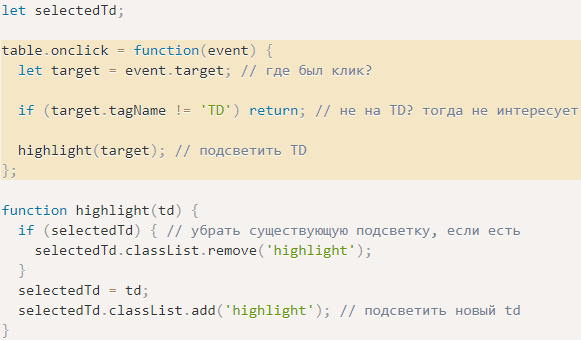
В современной разработке стадия погружения используется очень редко, обычно события обрабатываются во время всплытия. И в этом есть логика.

# Делегирование событий

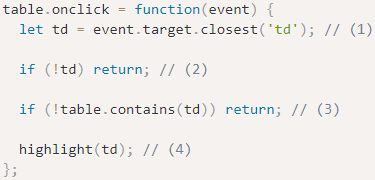
Всплытие и перехват событий позволяет реализовать один из самых важных приёмов разработки – делегирование. Идея в том, что если у нас есть много элементов, события на которых нужно обрабатывать похожим образом, то вместо того, чтобы назначать обработчик каждому, мы ставим один обработчик на их общего предка.

Из него можно получить целевой элемент event.target, понять на каком именно потомке произошло событие и обработать его.

Например, повесим обработчик клика на таблицу и будет отслеживать клики на ячейки:



Но td может содержать вложенные тэги и target.tagName будет не ‘td’, но по факту клик был внутри ячейки, вот решение:

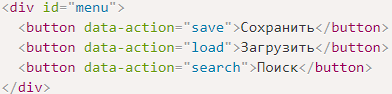


## Применение делегирования: действия в разметке

Например, нам нужно сделать меню с разными кнопками: «Сохранить (save)», «Загрузить (load)», «Поиск (search)» и т.д. И есть объект с соответствующими методами save, load, search… Как их состыковать? Первое, что может прийти в голову – это найти каждую кнопку и назначить ей свой обработчик среди методов объекта. Но существует более элегантное решение. Мы можем добавить один обработчик для всего меню и атрибуты data-action для каждой кнопки в соответствии с методами, которые они вызывают:



Обработчик считывает содержимое атрибута и выполняет метод. Взгляните на рабочий пример:





Обратите внимание, что метод this.onClick в строке, отмеченной звёздочкой (\*), привязывается к контексту текущего объекта this. Это важно, т.к. иначе this внутри него будет ссылаться на DOM-элемент (elem), а не на объект Menu, и this[action] будет не тем, что нам нужно.

Мы также можем использовать классы .action-save, .action-load, но подход с использованием атрибутов data-action является более семантичным. Их можно использовать и для стилизации в правилах CSS.

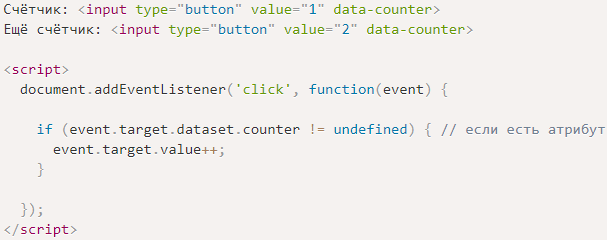
## Приём проектирования «поведение»

Приём проектирования «поведение» состоит из двух частей:

1. Элементу ставится пользовательский атрибут, описывающий его поведение.
2. При помощи делегирования ставится обработчик на документ, который ловит все клики (или другие события) и, если элемент имеет нужный атрибут, производит соответствующее действие.

## Поведение: «Счётчик»

Например, здесь HTML-атрибут data-counter добавляет кнопкам поведение: «увеличить значение при клике»:



Элементов с атрибутом data-counter может быть сколько угодно. Новые могут добавляться в HTML-код в любой момент. При помощи делегирования мы фактически добавили новый «псевдостандартный» атрибут в HTML, который добавляет элементу новую возможность («поведение»).

## Поведение: «Переключатель» (Toggler)

Ещё один пример поведения. Сделаем так, что при клике на элемент с атрибутом data-toggle-id будет скрываться/показываться элемент с заданным id:



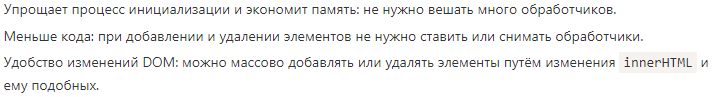
Ещё раз подчеркнём, что мы сделали. Теперь для того, чтобы добавить скрытие-раскрытие любому элементу, даже не надо знать JavaScript, можно просто написать атрибут data-toggle-id.

## Итого

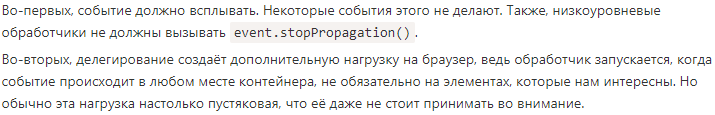
Алгоритм делегирования:

1. Вешаем обработчик на контейнер.
2. В обработчике проверяем исходный элемент event.target.
3. Если событие произошло внутри нужного нам элемента, то обрабатываем его.

Зачем использовать:



Конечно, у делегирования событий есть свои ограничения:



# Действия браузера по умолчанию

Многие события автоматически влекут за собой действие браузера.

Например:

* Клик по ссылке инициирует переход на новый URL.
* Нажатие на кнопку «отправить» в форме – отсылку её на сервер.
* Зажатие кнопки мыши над текстом и её движение в таком состоянии – инициирует его выделение.

Если обрабатываем событие в JS часто такое действие браузера не нужно. Но его можно отменить.

## Отмена действия *браузера*. preventDefault().

* Основной способ – это воспользоваться объектом **event**. Для отмены *действия* ***браузера*** существует стандартный метод **event.preventDefault().**
* Если же обработчик назначен через on<событие> (не через addEventListener), то также можно вернуть false из обработчика.



* *Возвращать true не нужно*

Обычно значение, которое возвращает обработчик события, игнорируется.

Единственное исключение – это return false из обработчика, назначенного через on<событие>.

В других случаях return не нужен, он никак не обрабатывается.

## Пример: меню

<https://learn.javascript.ru/default-browser-action#primer-menyu>

Там же про вытекание некоторых событий из других

## Опция «passive» для обработчика

Необязательная опция **passive: true** для addEventListener сигнализирует браузеру, что обработчик не собирается выполнять preventDefault().

Опция passive: true сообщает браузеру, что обработчик не собирается отменять прокрутку. Тогда браузер начинает её немедленно, обеспечивая максимально плавный интерфейс, параллельно обрабатывая событие.

<https://learn.javascript.ru/default-browser-action#optsiya-passive-dlya-obrabotchika>

## event.defaultPrevented

Свойство **event.defaultPrevented** установлено в true, если действие по умолчанию было предотвращено, и false, если нет. Практическое применение:

<https://learn.javascript.ru/default-browser-action#event-defaultprevented>

## Итого

Действий браузера по умолчанию достаточно много:

* **mousedown** – начинает выделять текст (если двигать мышкой).
* **click на <input type="checkbox">** – ставит или убирает галочку в input.
* **submit** – при нажатии на <input type="submit"> или при нажатии клавиши Enter в форме данные отправляются на сервер.
* **keydown** – при нажатии клавиши в поле ввода появляется символ.
* **contextmenu** – при правом клике показывается контекстное меню браузера.

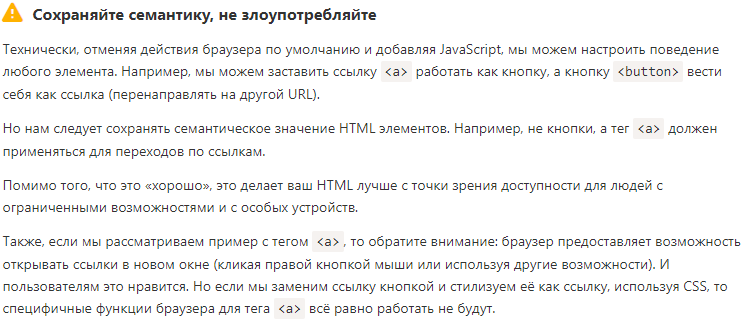
…и многие другие…

Все эти действия можно отменить, если мы хотим обработать событие исключительно при помощи JavaScript.

Чтобы отменить действие браузера по умолчанию, используйте event.preventDefault() или return false. Второй метод работает, только если обработчик назначен через on<событие>.

Опция passive: true для addEventListener сообщает браузеру, что действие по умолчанию не будет отменено. Это очень полезно для некоторых событий на мобильных устройствах, таких как touchstart и touchmove, чтобы сообщить браузеру, что он не должен ожидать выполнения всех обработчиков, а ему следует сразу приступать к выполнению действия по умолчанию, например, к прокрутке.

Если событие по умолчанию отменено, то значение event.defaultPrevented становится true, иначе false.



# Генерация пользовательских событий

Можно не только назначать обработчики, но и генерировать события из JavaScript-кода. Пользовательские события могут быть использованы при создании графических компонентов. Например, корневой элемент меню, реализованного при помощи JS, может генерировать события, относящиеся к этому меню: open (меню раскрыто), select (выбран пункт меню) и т.п. А другой код может слушать эти события и узнавать, что происходит с меню.

Можно генерировать не только совершенно новые, придуманные нами события, но и встроенные, такие как click, mousedown и другие.

## Конструктор Event

Встроенные классы для событий формируют иерархию аналогично классам для DOM-элементов. Её корнем является встроенный класс Event.



Где:

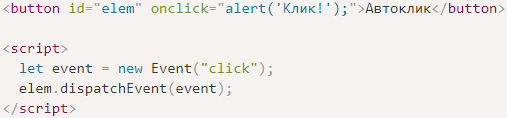
* **type** – тип события, строка, например "click" или же любой придуманный нами – "my-event".
* **options** – объект с тремя необязательными свойствами:

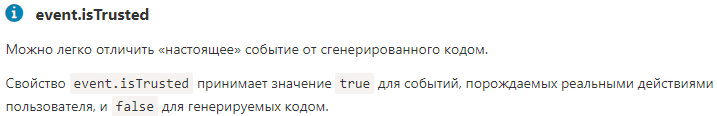
1. ***bubbles***: true/false – если true, тогда событие всплывает.
2. ***cancelable***: true/false – если true, тогда можно отменить действие по умолчанию. Позже мы разберём, что это значит для пользовательских событий.
3. ***composed***: true/false – если true, тогда событие будет всплывать наружу за пределы Shadow DOM. Позже мы разберём это в разделе Веб-компоненты.

По умолчанию все три свойства установлены в false: {bubbles: false, cancelable: false, composed: false}.

## Метод dispatchEvent

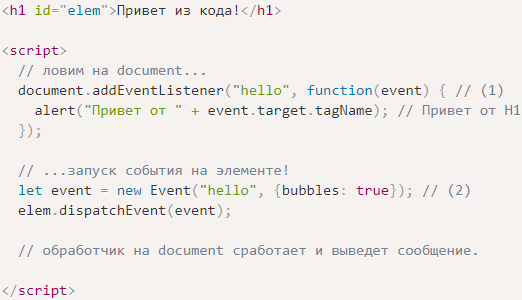
После того, как объект события создан, мы должны запустить его на элементе, вызвав метод **elem.dispatchEvent(event)**. Затем обработчики отреагируют на него, как будто это обычное браузерное событие. Если при создании указан флаг bubbles, то оно будет всплывать.

 В данном примере мы создали свое событие и тут же в коде запустили его на элементе. Далее обработчик в атрибуте onclick его обработает.



## Пример всплытия

Мы можем создать всплывающее событие с именем "hello" и поймать его на document. Всё, что нужно сделать – это установить флаг bubbles в true:



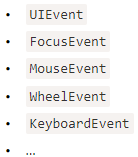
Обратите внимание:

1. Мы должны использовать addEventListener для наших собственных событий, т.к. on<event>-свойства существуют только для встроенных событий, то есть document.onhello не сработает.
2. Мы обязаны передать флаг bubbles:true, иначе наше событие не будет всплывать.

Механизм всплытия идентичен как для встроенного события (click), так и для пользовательского события (hello). Также одинакова работа фаз всплытия и погружения.

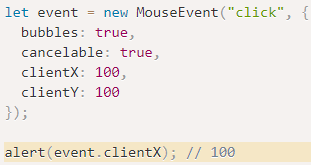
## MouseEvent, KeyboardEvent и другие

Для некоторых конкретных типов событий есть свои специфические конструкторы. Вот небольшой список конструкторов для различных событий пользовательского интерфейса, которые можно найти в спецификации UI Event:



Стоит использовать их вместо new Event, если мы хотим создавать такие события. К примеру, ***new MouseEvent("click").***

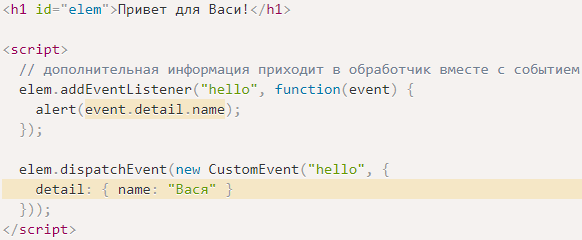
Специфический конструктор позволяет указать стандартные свойства для данного типа события. Например, clientX/clientY для события мыши (в объекте options):

 Впрочем, использование конкретного конструктора не является обязательным, можно обойтись Event, а свойства записать в объект отдельно, после создания, вот так: event.clientX=100. Здесь это скорее вопрос удобства и желания следовать правилам. События, которые генерирует браузер, всегда имеют правильный тип.

## Пользовательские события

Для генерации событий новых типов, таких как "hello", нужно использовать конструктор ***new CustomEvent***. Технически CustomEvent абсолютно идентичен Event за исключением одного:

У второго аргумента-объекта есть дополнительное свойство detail, в котором можно указывать информацию для передачи в событие:



Свойство detail может содержать любые данные.

## event.preventDefault()

Для многих браузерных событий есть «действия по умолчанию», такие как переход по ссылке, выделение и т.п.

Для новых, пользовательских событий, браузерных действий, конечно, нет, но код, который генерирует такое событие, может предусматривать какие-то свои действия после события.

*Вызов event.preventDefault() является возможностью для обработчика события сообщить в сгенерировавший событие код, что эти действия надо отменить.*

*Тогда вызов elem.dispatchEvent(event) возвратит false.* И код, сгенерировавший событие, узнает, что продолжать не нужно.

Ниже вы можете видеть кролика #rabbit и функцию hide(), которая при вызове генерирует на нём событие "hide", уведомляя всех интересующихся, что кролик собирается спрятаться.

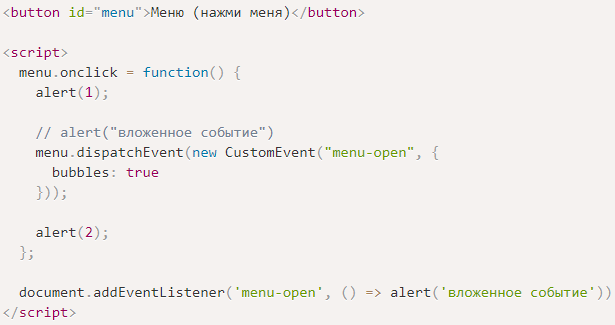
Любой обработчик может узнать об этом, подписавшись на событие hide через rabbit.addEventListener('hide',...) и, при желании, отменить действие по умолчанию через event.preventDefault(). Тогда кролик не исчезнет:



## Вложенные события обрабатываются синхронно

Обычные события обрабатываются асинхронно. То есть, если браузер обрабатывает onclick и в процессе этого произойдёт новое событие, то оно ждёт, пока закончится обработка onclick. Исключением является ситуация, когда событие инициировано из обработчика другого события. Тогда управление сначала переходит в обработчик вложенного события и уже после этого возвращается назад.

В примере ниже событие menu-open обрабатывается синхронно во время обработки onclick:

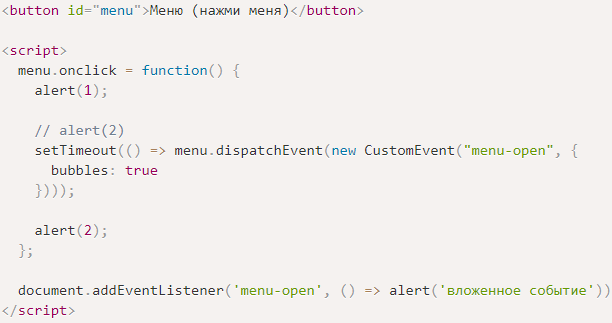


Порядок вывода: 1 → вложенное событие → 2.

Обратите внимание, что вложенное событие menu-open успевает всплыть и запустить обработчик на document. Обработка вложенного события полностью завершается до того, как управление возвращается во внешний код (onclick).

Это справедливо не только для dispatchEvent, но и для других ситуаций. JavaScript в обработчике события может вызвать другие методы, которые приведут к другим событиям – они тоже обрабатываются синхронно.

Если нам это не подходит, то мы можем либо поместить dispatchEvent (или любой другой код, инициирующий события) в конец обработчика onclick, либо, если это неудобно, можно обернуть генерацию события в setTimeout с нулевой задержкой:



## Итого

Чтобы сгенерировать событие из кода, вначале надо создать объект события.

Базовый конструктор Event(name, options) принимает обязательное имя события и options – объект с двумя свойствами:

* bubbles: true чтобы событие всплывало.
* cancelable: true если мы хотим, чтобы event.preventDefault() работал.

Особые конструкторы встроенных событий MouseEvent, KeyboardEvent и другие принимают специфичные для каждого конкретного типа событий свойства. Например, clientX для событий мыши.

Для пользовательских событий стоит применять конструктор CustomEvent. У него есть дополнительная опция detail, с помощью которой можно передавать информацию в объекте события. После чего все обработчики смогут получить к ней доступ через event.detail.

Несмотря на техническую возможность генерировать встроенные браузерные события типа click или keydown, пользоваться ей стоит с большой осторожностью.

Весьма часто, когда разработчик хочет сгенерировать встроенное событие – это вызвано «кривой» архитектурой кода.

Как правило, генерация встроенных событий полезна в следующих случаях:

* Либо как явный и грубый хак, чтобы заставить работать сторонние библиотеки, в которых не предусмотрены другие средства взаимодействия.
* Либо для автоматического тестирования, чтобы скриптом «нажать на кнопку» и посмотреть, произошло ли нужное действие.

Пользовательские события со своими именами часто создают для улучшения архитектуры, чтобы сообщить о том, что происходит внутри наших меню, слайдеров, каруселей и т.д.

# Основы событий мыши

Сразу заметим: эти события бывают не только из-за мыши, но и эмулируются на других устройствах, в частности, на мобильных, для совместимости.

## Типы событий мыши

Мы можем разделить события мыши на две категории: **«простые**» и **«комплексные»**.

**Простые события**

Самые часто используемые простые события:

* *mousedown/mouseup* - Кнопка мыши нажата/отпущена над элементом.
* *mouseover/mouseout* - Курсор мыши появляется над элементом и уходит с него
* *mousemove* - Каждое движение мыши над элементом генерирует это событие.
* *сontextmenu* - Вызывается при попытке открытия контекстного меню, как правило, нажатием правой кнопки мыши. Но, заметим, это не совсем событие мыши, оно может вызываться и специальной клавишей клавиатуры.

**Комплексные события**

*сlick* **-** Вызывается при mousedown , а затем mouseup над одним и тем же элементом, если использовалась левая кнопка мыши.

*dblclick* **-** Вызывается двойным кликом на элементе.

Комплексные события состоят из простых, поэтому в теории мы могли бы без них обойтись. Но хорошо, что они существуют, потому что работать с ними очень удобно.

## Порядок событий

Например, клик мышью вначале вызывает mousedown, когда кнопка нажата, затем mouseup и click, когда она отпущена. В случае, когда одно действие инициирует несколько событий, порядок их выполнения фиксирован. То есть обработчики событий вызываются в следующем порядке: mousedown → mouseup → click.

## Получение информации о кнопке: which

Свойство which позволяет определить нажатую кнопку мыши. Это свойство не используется для событий click и contextmenu, поскольку первое происходит только при нажатии левой кнопкой мыши, а второе – правой.

Но если мы отслеживаем mousedown и mouseup, то оно нам нужно, потому что эти события срабатывают на любой кнопке, и which позволяет различать между собой «нажатие правой кнопки» и «нажатие левой кнопки».

Есть три возможных значения:

* event.which == 1 – левая кнопка
* event.which == 2 – средняя кнопка
* event.which == 3 – правая кнопка

## Модификаторы: shift, alt, ctrl и meta

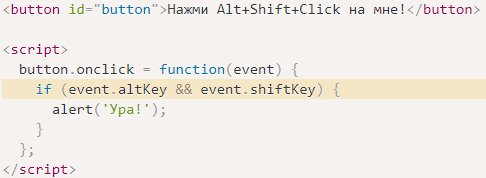
Все события мыши включают в себя информацию о нажатых клавишах-модификаторах.

Свойства объекта события:

* *shiftKey*: Shift
* *altKey*: Alt (или Opt для Mac)
* *ctrlKey*: Ctrl
* *metaKey*: Cmd для Mac

Они равны true, если во время события была нажата соответствующая клавиша.

Например, кнопка внизу работает только при комбинации Alt+Shift+клик:



* *Внимание: обычно на Mac используется клавиша Cmd вместо Ctrl*
* *Не забывайте про мобильные устройства*

Комбинации клавиш на клавиатуре – это хорошее дополнение к рабочему процессу. Если у пользователя есть клавиатура – они работают. Ну а если на его устройстве её нет – должен быть другой способ сделать то же самое.

## Координаты: clientX/Y, pageX/Y

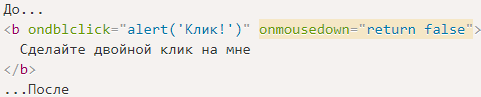
Все события мыши имеют координаты двух видов:

* Относительно окна: clientX и clientY.
* Относительно документа: pageX и pageY.

## Отключаем выделение

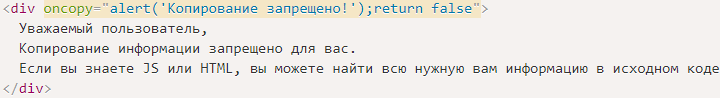
Двойной клик мыши имеют побочный эффект, который может быть неудобен в некоторых интерфейсах: он выделяет текст. Если зажать левую кнопку мыши и, не отпуская кнопку, провести мышью, то также будет выделение, которое в интерфейсах может быть «не кстати». Есть несколько способов запретить выделение, о которых вы можете прочитать в главе “Selection и Range”.

Самый простой вариант: отменить действие браузера по умолчанию при событии *mousedown*, это отменит оба этих выделения:



* Предотвращение копирования

Если мы хотим отключить выделение для защиты содержимого страницы от копирования, то мы можем использовать другое событие: oncopy.



# Движение мыши: mouseover/out, mouseenter/leave

В этой главе мы более подробно рассмотрим события, возникающие при движении указателя мыши над элементами страницы.

## События mouseover/mouseout, relatedTarget

Событие **mouseover** происходит в момент, когда курсор оказывается над элементом, а событие **mouseout** – в момент, когда курсор уходит с элемента.



Данные события имеют свойство **relatedTarget.**

Для события **mouseover**:

* **event.target** – это элемент, на который курсор перешёл.
* **event.relatedTarget** – это элемент, с которого курсор ушёл (relatedTarget → target).

Для события **mouseout** наоборот:

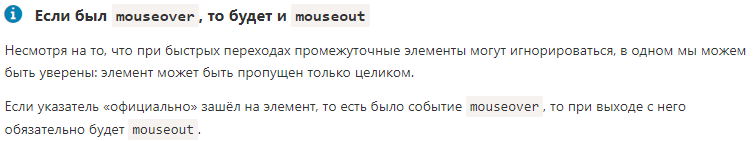
* **event.target** – это элемент, с которого курсор ушёл.
* **event.relatedTarget** – это элемент, на который курсор перешёл (target → relatedTarget).
* *Свойство relatedTarget может быть null*

Это нормально и означает, что указатель мыши перешёл не с другого элемента, а из-за пределов окна браузера. Или же, наоборот, ушёл за пределы окна.

## Пропуск элементов

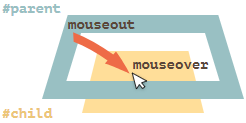
Событие mousemove происходит при движении мыши. Однако, это не означает, что указанное событие генерируется при прохождении каждого пикселя. Браузер периодически проверяет позицию курсора и, заметив изменения, генерирует события mousemove. Это означает, что если пользователь двигает мышкой очень быстро, то некоторые DOM-элементы могут быть пропущены:





## Событие mouseout при переходе на потомка

Важная особенность события mouseout – оно генерируется в том числе, когда указатель переходит с элемента на его потомка. То есть, визуально указатель всё ещё на элементе, но мы получим mouseout!

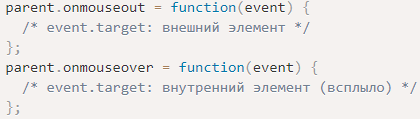


**По логике браузера, курсор мыши может быть только над одним элементом в любой момент времени – над самым глубоко вложенным элементом и верхним по z-index.**

Таким образом, если курсор переходит на другой элемент (пусть даже дочерний), то он покидает предыдущий.

**Важная деталь!** Событие mouseover, происходящее на потомке, всплывает. Поэтому если на родительском элементе есть такой обработчик, то оно его вызовет.

При переходе с родителя элемента на потомка – на родителе сработают два обработчика: и *mouseout* и *mouseover*:



Если код внутри обработчиков не смотрит на target, то он подумает, что мышь ушла с элемента parent и вернулась на него обратно. Но это не так! Мышь никуда не уходила, она просто перешла на потомка.

Если при уходе с элемента что-то происходит, например, запускается анимация, то такая интерпретация происходящего может давать нежелательные побочные эффекты.

Чтобы этого избежать, можно смотреть на relatedTarget и, если мышь всё ещё внутри элемента parent, то игнорировать такие события.

Или же можно использовать другие события: mouseenter и mouseleave

## События mouseenter и mouseleave

События **mouseenter/mouseleave** похожи на mouseover/mouseout. Они тоже генерируются, когда курсор мыши переходит на элемент или покидает его.

Но есть и пара важных отличий:

* Переходы внутри элемента, на его потомки и с них, не считаются.
* События **mouseenter/mouseleave** не всплывают.

Когда указатель появляется над элементом – генерируется **mouseenter**, причём не имеет значения, где именно указатель: на самом элементе или на его потомке.

## Делегирование событий

События **mouseenter/leave** просты и легки в использовании. Но они не всплывают. Таким образом, мы не можем их делегировать.

Что ж, не проблема – будем использовать mouseover/mouseout. Пример:

<https://learn.javascript.ru/mousemove-mouseover-mouseout-mouseenter-mouseleave#delegirovanie-sobytiy>

# Drag'n'Drop с событиями мыши

## Алгоритм Drag’n’Drop

Базовый алгоритм Drag’n’Drop выглядит так:

1. При mousedown – готовим элемент к перемещению, если необходимо (например, создаём его копию).
2. Затем при mousemove передвигаем элемент на новые координаты путём смены left/top и position:absolute.
3. При mouseup – остановить перенос элемента и произвести все действия, связанные с окончанием Drag’n’Drop.

Пример с перетаскиванием мяча:

<https://learn.javascript.ru/mouse-drag-and-drop#algoritm-drag-n-drop>

## Цели переноса (droppable)

Обычно мы берём один элемент и перетаскиваем в другой. Например, «файл» в «папку» или что-то ещё. Абстрактно говоря, мы берём перетаскиваемый (draggable) элемент и помещаем его в другой элемент «цель переноса» (droppable).

Нам нужно знать:

* куда пользователь положил элемент в конце переноса, чтобы обработать его окончание
* и, желательно, над какой потенциальной целью (элемент, куда можно положить, например, изображение папки) он находится в процессе переноса, чтобы подсветить её.

<https://learn.javascript.ru/mouse-drag-and-drop#tseli-perenosa-droppable>

- **document.elementFromPoint** - для проверки того, над каким элементом мы «летим», в любое время. И обработать окончание переноса, когда оно случится.

# Клавиатура: keydown и keyup

## События keydown и keyup

Событие keydown происходит при нажатии клавиши, а keyup – при отпускании.

* **event.code** позволяет получить «физический код клавиши».
* **event.key -** позволяет получить символ.

Например одну и ту же клавишу Z можно нажать с клавишей Shift и без. В результате получится два разных символа: z в нижнем регистре и Z в верхнем регистре.

Свойство **event.key (символ)** –будет отличаться в таком случае. Но **event.code** всегда будет тот же: «KeyZ». Даже если сменить язык, code не измениться.

* Буквенные клавиши имеют коды по типу "Key<буква>": "KeyA", "KeyB" и т.д.
* Коды числовых клавиш строятся по принципу: "Digit<число>": "Digit0", "Digit1" и т.д.
* Код специальных клавиш – это их имя: "Enter", "Backspace", "Tab" и т.д.
* *Регистр важен: "KeyZ", а не "keyZ"*

## Автоповтор

При долгом нажатии клавиши возникает автоповтор: keydown срабатывает снова и снова, и когда клавишу отпускают, то отрабатывает keyup. Так что ситуация, когда много keydown и один keyup, абсолютно нормальна.

Для событий, вызванных автоповтором, у объекта события свойство event.repeat равно true.

## Действия по умолчанию

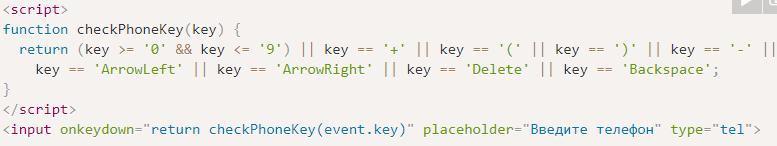
Действия по умолчанию весьма разнообразны, много чего можно инициировать нажатием на клавиатуре.

Для примера:

* Появление символа (самое очевидное).
* Удаление символа (клавиша Delete).
* Прокрутка страницы (клавиша PageDown).
* Открытие диалогового окна браузера «Сохранить» (Ctrl+S)
* …и так далее.

Предотвращение стандартного действия с помощью event.preventDefault() работает практически во всех сценариях, кроме тех, которые происходят на уровне операционной системы. Например, комбинация Alt+F4 инициирует закрытие браузера в Windows, что бы мы ни делали в JavaScript.

Для примера, <input> ниже ожидает телефонный номер, так что ничего кроме чисел, +, () или - принято не будет:



Альтернатива – отслеживать событие input, оно генерируется после любых изменений в поле <input>, и мы можем проверять новое значение и подчёркивать/изменять его, если оно не подходит.

# События указателя

События указателя (Pointer events) – это современный способ обработки ввода с помощью различных указывающих устройств, таких как мышь, перо/стилус, сенсорный экран и так далее.

## Типы событий указателя



Для каждого mouse<события> есть соответствующее pointer<событие>, которое играет аналогичную роль. Также есть 3 дополнительных события указателя, у которых нет соответствующего аналога mouse...

## Свойства событий указателя

События указателя содержат те же свойства, что и события мыши, например **clientX/Y**, **target** и т.п., и несколько дополнительных:

* **pointerId** – уникальный идентификатор указателя, вызвавшего событие.

Идентификатор генерируется браузером. Это свойство позволяет обрабатывать несколько указателей, например сенсорный экран со стилусом и мульти-тач (увидим примеры ниже).

* **pointerType** – тип указывающего устройства. Должен быть строкой с одним из значений: «mouse», «pen» или «touch».

Мы можем использовать это свойство, чтобы определять разное поведение для разных типов указателей.

* **isPrimary** – равно true для основного указателя (первый палец в мульти-тач).

Некоторые устройства измеряют область контакта и степень надавливания, например пальца на сенсорном экране, для этого есть дополнительные свойства (почти не используются):

* **width** – ширина области соприкосновения указателя (например, пальца) с устройством. Если не поддерживается, например мышью, то всегда равно 1.
* **height** – высота области соприкосновения указателя с устройством. Если не поддерживается, например мышью, то всегда равно 1.
* **pressure** – степень давления указателя в диапазоне от 0 до 1. Для устройств, которые не поддерживают давление, принимает значение 0.5 (нажато) либо 0.
* **tangentialPressure** – нормализованное тангенциальное давление.
* **tiltX, tiltY, twist** – специфичные для пера свойства, описывающие положение пера относительно сенсорной поверхности.

## Мульти-тач

События указателя позволяют обрабатывать мульти-тач с помощью свойств **pointerId** и **isPrimary**.

Вот что происходит, когда пользователь касается сенсорного экрана в одном месте, а затем в другом:

* При касании первым пальцем:

происходит событие **pointerdown** со свойством *isPrimary=true* и некоторым *pointerId.*

* При касании втором и последующими пальцами (при остающемся первом):

происходит событие **pointerdown** со свойством *isPrimary=false* и уникальным *pointerId* для каждого касания.

Если коснуться экрана 5 пальцами одновременно, получим 5 событий pointerdown, каждое со своими координатами и индивидуальным pointerId.

Мы можем отслеживать несколько касающихся экрана пальцев, используя их pointerId. Когда пользовтаель перемещает, а затем убирает палец, получаем события pointermove и pointerup с тем же pointerId, что и при событии pointerdown.

## Событие: pointercancel

Событие pointercancel происходит, когда текущее действие с указателем по какой-то причине прерывается, и события указателя больше не генерируются.

К таким причинам можно отнести:

* Указывающее устройство было физически выключено.
* Изменилась ориентация устройства (перевернули планшет).
* Браузер решил сам обработать действие, считая его жестом мыши, масштабированием и т.п.

Пример с мячиком: <https://learn.javascript.ru/pointer-events#sobytie-pointercancel>

**Предотвращайте действие браузера по умолчанию, чтобы избежать pointercancel**

Нужно сделать две вещи (для примера с мячиком):

1. Предотвратить запуск встроенного drag’n’drop
   * Мы можем сделать это, задав ball.ondragstart = () => false, как описано в статье [Drag'n'Drop с событиями мыши](https://learn.javascript.ru/mouse-drag-and-drop).
   * Это работает для событий мыши.
2. Для устройств с сенсорным экраном существуют другие действия браузера, связанные с касаниями, кроме drag’n’drop. Чтобы с ними не возникало проблем:
   * Мы можем предотвратить их, добавив в CSS свойство #ball { touch-action: none }.
   * Затем наш код начнёт корректно работать на устройствах с сенсорным экраном

## Захват указателя

Захват указателя – особая возможность событий указателя.

**elem.setPointerCapture(pointerId)** – привязывает события с данным pointerId к elem. После такого вызова все события указателя с таким pointerId будут иметь elem в качестве целевого элемента (как будто произошли над elem), вне зависимости от того, где в документе они произошли. Другими словами, elem.setPointerCapture(pointerId) меняет target всех дальнейших событий с данным pointerId на elem.

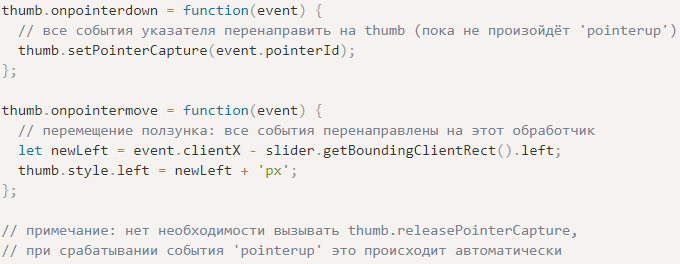
Эта привязка отменяется:

* автоматически, при возникновении события pointerup или pointercancel,
* автоматически, если elem удаляется из документа,
* при вызове elem.releasePointerCapture(pointerId).

**Захват указателя используется для упрощения операций с переносом (drag’n’drop) элементов.**

Пример со слайдером: Захват указателя позволяет привязать pointermove к thumb, а не к document и избежать любых подобных проблем:

* Мы можем вызывать *thumb.setPointerCapture(event.pointerId)* в обработчике pointerdown,
* Тогда дальнейшие события указателя до pointerup/cancel будут привязаны к thumb.
* Затем, когда произойдёт pointerup (передвижение завершено), привязка будет автоматически удалена, нам об этом не нужно беспокоиться.



## Итого

События указателя позволяют одновременно обрабатывать действия с помощью мыши, касания и пера, в едином фрагменте кода.

События указателя расширяют события мыши. Мы можем заменить mouse на pointer в названиях событий и код продолжит работать для мыши, при этом получив лучшую поддержку других типов устройств.

При обработке переносов и сложных касаний, которые браузер может попытаться обработать сам, **не забывайте отменять действие брауера** и ставить touch-events: none в CSS для элементов, с которыми мы взаимодействуем.

Дополнительные возможности событий указателя:

* Поддержка мультитач с помощью pointerId и isPrimary.
* Особые свойства для определённых устройств, такие как pressure, width/height и другие.
* Захват указателя: мы можем перенаправить все события указателя на определённый элемент до наступления события pointerup/pointercancel.

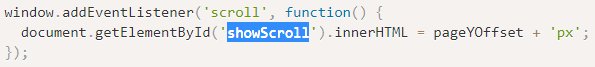
# Прокрутка

Событие прокрутки scroll позволяет реагировать на прокрутку страницы или элемента.

Можно:

* Показать/скрыть дополнительные элементы управления или информацию, основываясь на том, в какой части документа находится пользователь.
* Подгрузить данные, когда пользователь прокручивает страницу вниз до конца.

Вот небольшая функция для отображения текущей прокрутки в элементе с id showScroll' :



Событие scroll работает как на window, так и на других элементах, на которых включена прокрутка.

## Предотвращение прокрутки

Нельзя предотвратить прокрутку, используя event.preventDefault() в обработчике onscroll, потому что он срабатывает после того, как прокрутка уже произошла.

Но можно предотвратить прокрутку, используя event.preventDefault() на событии, которое вызывает прокрутку, например, на событии keydown для клавиш pageUp и pageDown.

Если поставить на них обработчики, в которых вызвать event.preventDefault(), то прокрутка не начнётся.

Способов инициировать прокрутку много, поэтому более надёжный способ – использовать CSS, свойство overflow.

# Свойства и методы формы

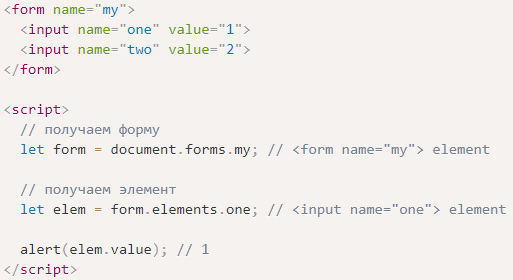
## Навигация: формы и элементы

Формы в документе входят в специальную коллекцию *document.forms.*

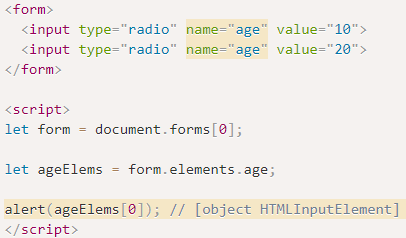
Это так называемая «именованная» коллекция: мы можем использовать для получения формы как её имя, так и порядковый номер в документе:



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



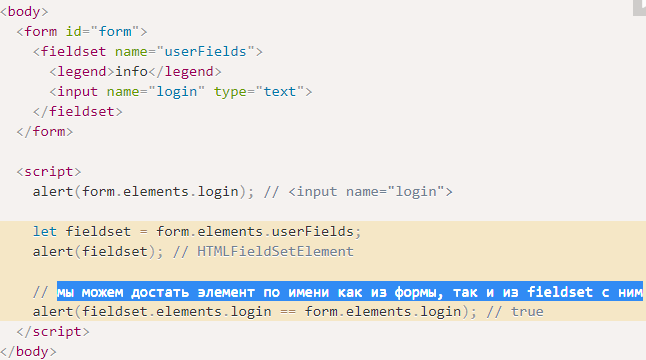
Может быть несколько элементов с одним и тем же именем, это часто бывает с кнопками-переключателями radio. В этом случае form.elements[name] является коллекцией, например:



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

* *<fieldset> как «подформа»*

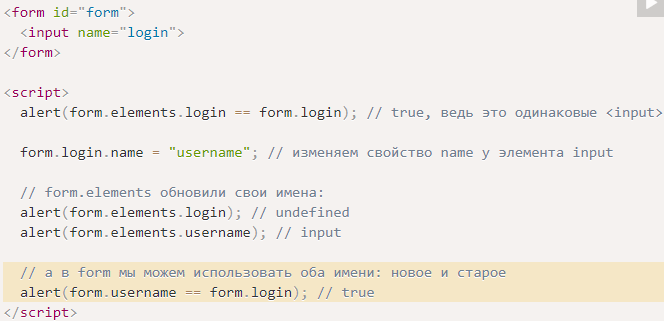
Форма может содержать один или несколько элементов <fieldset> внутри себя. Они также поддерживают свойство elements, в котором находятся элементы управления внутри них. Мы можем достать элемент по имени как из формы, так и из fieldset с ним:



* *Сокращённая форма записи: form.name*

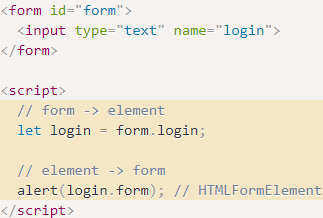
Вместо form.elements.login мы можем написать form.login.

Это также работает, но есть небольшая проблема: если мы получаем элемент, а затем меняем его свойство name, то он всё ещё будет доступен под старым именем (также, как и под новым):



## Обратная ссылка: element.form

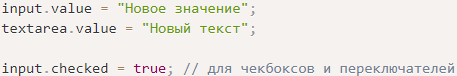
Для любого элемента форма доступна через element.form. Так что форма ссылается на все элементы, а эти элементы ссылаются на форму:

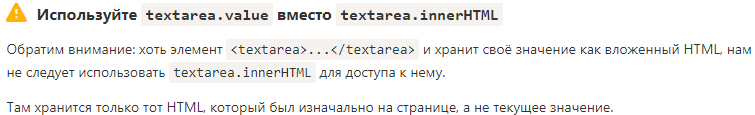


## Элементы формы

* **input и textarea**

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





* **select и option**

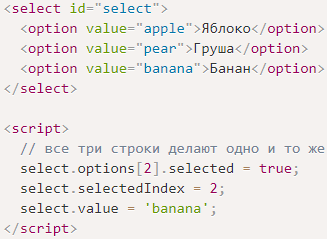
Элемент **<select>** имеет 3 важных свойства:

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>.

Пример:



**<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:



**new Option**

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).

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).

# Фокусировка: focus/blur

Элемент получает фокус, когда пользователь кликает по нему или использует клавишу Tab. Также существует HTML-атрибут autofocus, который устанавливает фокус на элемент, когда страница загружается. Есть и другие способы получения фокуса.

*Фокусировка обычно означает*: «приготовься к вводу данных на этом элементе», это хороший момент, чтобы инициализовать или загрузить что-нибудь.

*Потеря фокуса (blur)* обычно означает «данные введены», и мы можем выполнить проверку введённых данных или даже отправить эти данные на сервер и так далее.

## События focus/blur

Событие focus вызывается в момент фокусировки, а blur – когда элемент теряет фокус. Используем их для валидации (проверки) введённых данных:

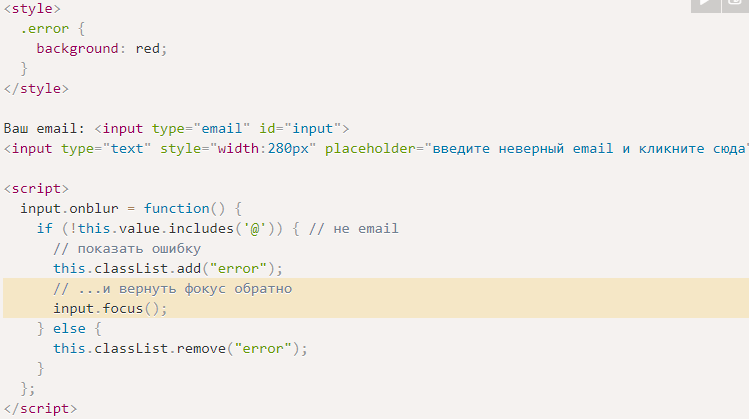


Современный HTML позволяет делать валидацию с помощью атрибутов required, pattern и т.д. Иногда – это всё, что нам нужно. JavaScript можно использовать, когда мы хотим больше гибкости. А ещё мы могли бы отправлять изменённое значение на сервер, если оно правильное.

## Методы focus/blur

Методы elem.focus() и elem.blur() устанавливают/снимают фокус.

Например, запретим посетителю переключаться с поля ввода, если введённое значение не прошло валидацию:



Отметим, что мы не можем «отменить потерю фокуса», вызвав event.preventDefault() в обработчике onblur потому, что onblur срабатывает после потери фокуса элементом.

* *Потеря фокуса, вызванная JavaScript*

Потеря фокуса может произойти по множеству причин.

Одна из них – когда посетитель кликает куда-то ещё. Но и JavaScript может быть причиной, например:

alert переводит фокус на себя – элемент теряет фокус (событие blur), а когда alert закрывается – элемент получает фокус обратно (событие focus).

Если элемент удалить из DOM, фокус также будет потерян. Если элемент добавить обратно, то фокус не вернётся.

Из-за этих особенностей обработчики focus/blur могут сработать тогда, когда это не требуется.

## Включаем фокусировку на любом элементе: tabindex

Многие элементы по умолчанию не поддерживают фокусировку. focus/blur гарантирована только для элементов, с которыми посетитель может взаимодействовать: <button>, <input>, <select>, <a> и т.д. Элементы форматирования <div>, <span>, <table> – по умолчанию не могут получить фокус. Метод elem.focus() не работает для них, и события focus/blur никогда не срабатывают.

Это можно изменить HTML-атрибутом tabindex.

Любой элемент поддерживает фокусировку, если имеет **tabindex**. Значение этого атрибута – порядковый номер элемента, когда клавиша Tab (или что-то аналогичное) используется для переключения между элементами. То есть: если у нас два элемента, первый имеет tabindex="1", а второй tabindex="2", то находясь в первом элементе и нажав Tab – мы переместимся во второй.

Порядок перебора таков: сначала идут элементы со значениями tabindex от 1 и выше, в порядке tabindex, а затем элементы без tabindex (например, обычный <input>). При совпадающих tabindex элементы перебираются в том порядке, в котором идут в документе.

**Есть два специальных значения:**

* tabindex="0" ставит элемент в один ряд с элементами без tabindex. То есть, при переключении такие элементы будут после элементов с tabindex ≥ 1.

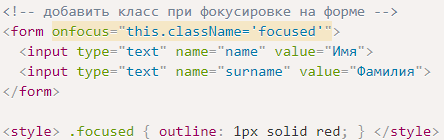
Обычно используется, чтобы включить фокусировку на элементе, но не менять порядок переключения. Чтобы элемент мог участвовать в форме наравне с обычными <input>.

* tabindex="-1" позволяет фокусироваться на элементе только программно. Клавиша Tab проигнорирует такой элемент, но метод elem.focus() будет действовать.



## События focusin/focusout, всплытие

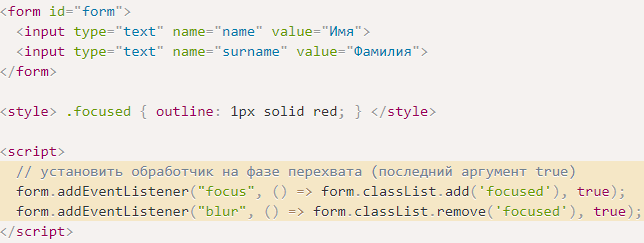
События focus и blur не всплывают. Например, мы не можем использовать onfocus на <form>, чтобы подсветить её:



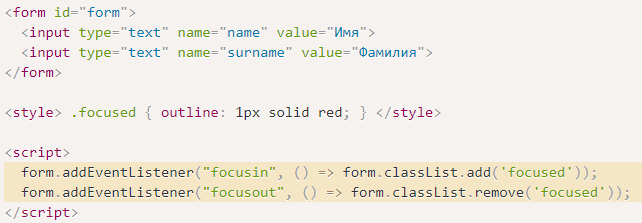
Событие срабатывает только на таргете, и не срабатывает на родителях.

У этой проблемы два решения.

*Первое:*– focus/blur не всплывают, но передаются вниз на фазе перехвата:



*Второе решение*: события focusin и focusout – такие же, как и focus/blur, но они всплывают.



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

## Событие: change

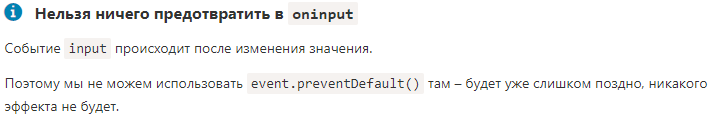
Событие change срабатывает по окончании изменения элемента. Для текстовых <input> это означает, что событие происходит при потере фокуса.

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

## Событие: input

Событие input срабатывает каждый раз при изменении значения. Оно работает при любых изменениях значений, даже если они не связаны с клавиатурными действиями: вставка с помощью мыши или распознавание речи.

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

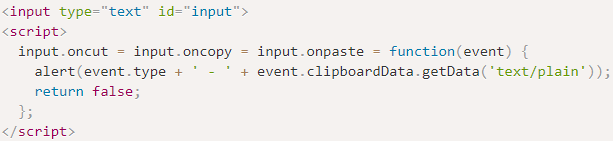


## События: cut, copy, paste

Эти события происходят при вырезании/копировании/вставке данных. Они относятся к классу ClipboardEvent 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:



# 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.

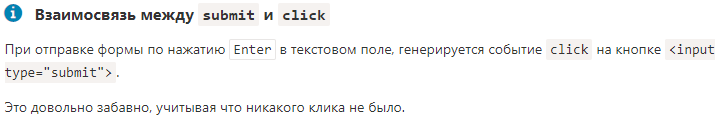
Метод form.submit() позволяет инициировать отправку формы из JavaScript. Мы можем использовать его для динамического создания и отправки наших собственных форм на сервер.

## 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.

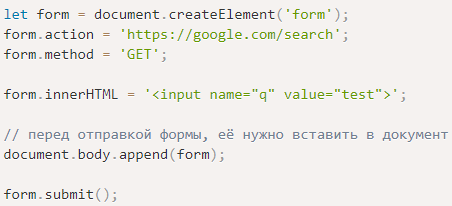
Оба действия сгенерируют событие submit на форме. Обработчик может проверить данные, и если есть ошибки, показать их и вызвать event.preventDefault(), тогда форма не будет отправлена на сервер.



## 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:



# Страница: DOMContentLoaded, load, beforeunload, unload

У жизненного цикла HTML-страницы есть три важных события:

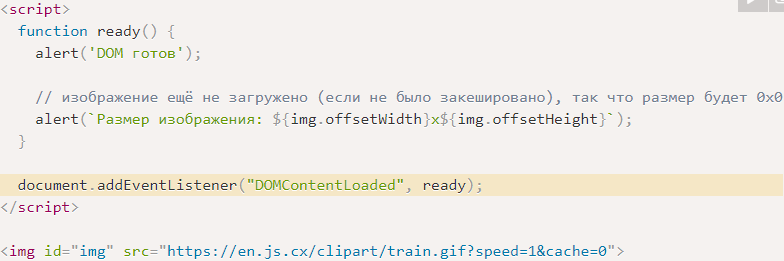
1. **DOMContentLoaded** – браузер полностью загрузил HTML, было построено DOM-дерево, но внешние ресурсы, такие как картинки <img> и стили, могут быть ещё не загружены.
2. **load** – браузер загрузил HTML и внешние ресурсы (картинки, стили и т.д.).
3. **beforeunload/unload** – пользователь покидает страницу.

Каждое из этих событий может быть полезно:

1. *DOMContentLoaded* – DOM готов, так что обработчик может искать DOM-узлы и инициализировать интерфейс.
2. *Load* - внешние ресурсы были загружены, стили применены, размеры картинок известны и т.д.
3. *Beforeunload -* пользователь покидает страницу. Мы можем проверить, сохранил ли он изменения и спросить, на самом ли деле он хочет уйти.
4. *Unload* - пользователь почти ушёл, но мы всё ещё можем запустить некоторые операции, например, отправить статистику.

## DOMContentLoaded

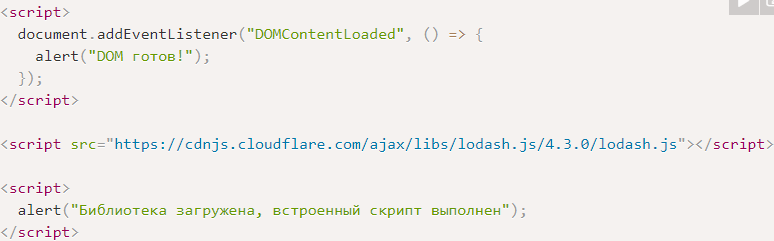
Cрабатывает на объекте document. Например:



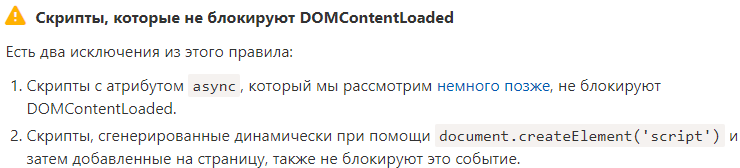
В этом примере обработчик DOMContentLoaded запустится, когда документ загрузится, так что он увидит все элементы, включая расположенный ниже <img>. Но он не дожидается, пока загрузится изображение. Поэтому alert покажет нулевой размер.

## DOMContentLoaded и скрипты

Когда браузер обрабатывает HTML-документ и встречает тег <script>, он должен выполнить его перед тем, как продолжить строить DOM. Это делается на случай, если скрипт захочет изменить DOM. Пример:

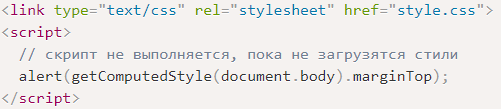


В примере выше мы сначала увидим «Библиотека загружена…», а затем «DOM готов!» (все скрипты выполнены).



## DOMContentLoaded и стили

Внешние таблицы стилей не затрагивают DOM, поэтому DOMContentLoaded их не ждёт. Но здесь есть подводный камень. Если после стилей у нас есть скрипт, то этот скрипт должен дождаться, пока загрузятся стили:



Причина в том, что скрипту может понадобиться получить координаты или другие свойства элементов, зависящих от стилей, как в примере выше. Естественно, он должен дождаться, пока стили загрузятся.

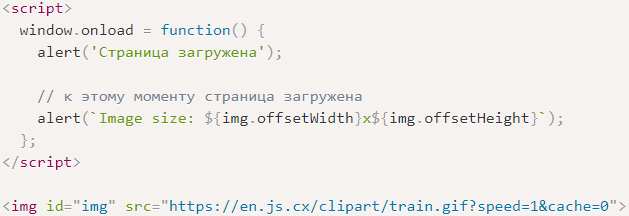
## Встроенное в браузер автозаполнение

<https://learn.javascript.ru/onload-ondomcontentloaded#vstroennoe-v-brauzer-avtozapolnenie>

## window.onload

Событие load на объекте window наступает, когда загрузилась вся страница, включая стили, картинки и другие ресурсы.

В примере ниже правильно показаны размеры картинки, потому что window.onload дожидается всех изображений:

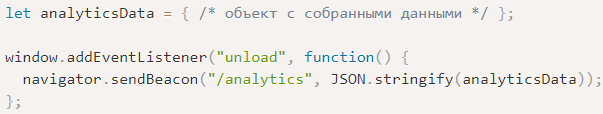


## window.onunload

Когда посетитель покидает страницу, на объекте window генерируется событие unload. В этот момент стоит совершать простые действия, не требующие много времени, вроде закрытия связанных всплывающих окон. Обычно здесь отсылают статистику. Предположим, имеется какая=то статистика использования страницы.

Естественно, событие unload – это тот момент, когда пользователь нас покидает и мы хотим сохранить эти данные. Для этого существует специальный метод **navigator.sendBeacon(url, data)**, описанный в спецификации <https://w3c.github.io/beacon/>.

Он посылает данные в фоне. Переход к другой странице не задерживается: браузер покидает страницу, но всё равно выполняет sendBeacon:



* Отсылается POST-запрос.
* Мы можем послать не только строку, но так же формы и другие форматы, как описано в главе Fetch, но обычно это строковый объект.
* Размер данных ограничен 64 Кб.

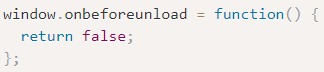
К тому моменту, как sendBeacon завершится, браузер уже покинет страницу, так что возможности обработать ответ сервера не будет (для статистики он обычно пустой).

Для таких запросов с закрывающейся страницей есть специальный флаг keepalive в методе [fetch](https://learn.javascript.ru/fetch) для общих сетевых запросов. Вы можете найти больше информации в главе [Fetch API](https://learn.javascript.ru/fetch-api).

Если мы хотим отменить переход на другую страницу, то здесь мы этого сделать не сможем. Но сможем в другом месте – в событии **onbeforeunload.**

## window.onbeforeunload

Если мы отменим это событие, то браузер спросит посетителя, уверен ли он.



После этого кода, когда пользователь закрывает вкладку или обновляет страницу – будет спрашиваться подтверждение.

## readyState

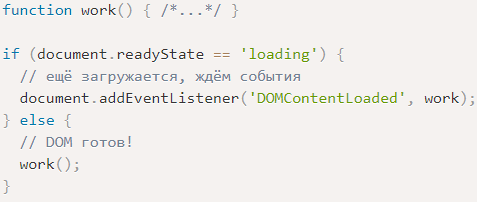
Что произойдёт, если мы установим обработчик DOMContentLoaded после того, как документ загрузился? Естественно, он никогда не запустится.

Есть случаи, когда мы не уверены, готов документ или нет. Мы бы хотели, чтобы наша функция исполнилась, когда DOM загрузился, будь то сейчас или позже.

Свойство document.readyState показывает нам текущее состояние загрузки. Есть три возможных значения:

* "loading" – документ загружается.
* "interactive" – документ был полностью прочитан.
* "complete" – документ был полностью прочитан и все ресурсы (такие как изображения) были тоже загружены.

Так что мы можем проверить document.readyState и, либо установить обработчик, либо, если документ готов, выполнить код сразу же:



Также есть событие readystatechange, которое генерируется при изменении состояния, так что мы можем вывести все эти состояния таким образом:



## Итого

События загрузки страницы:

* **DOMContentLoaded** генерируется на document, когда DOM готов. Мы можем применить JavaScript к элементам на данном этапе.

Скрипты, вроде <script>...</script> или <script src="..."></script> блокируют DOMContentLoaded, браузер ждёт, пока они выполнятся.

Изображения и другие ресурсы тоже всё ещё могут продолжать загружаться.

* Событие **load** на window генерируется, когда страница и все ресурсы загружены. Мы редко его используем, потому что обычно нет нужды ждать так долго.
* Событие **beforeunload** на window генерируется, когда пользователь покидает страницу. Если мы отменим событие, браузер спросит, на самом ли деле пользователь хочет уйти (например, у нас есть несохранённые изменения).
* Событие **unload** на window генерируется, когда пользователь окончательно уходит, в обработчике мы можем делать только простые вещи, которые ни о чём не спрашивают пользователя и не заставляют его ждать. Из-за этих ограничений оно редко используется. Мы можем послать сетевой запрос с помощью navigator.sendBeacon.
* **document.readyState** – текущее состояние документа, изменения можно отследить с помощью события readystatechange:

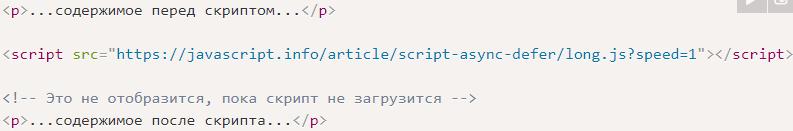
1. loading – документ грузится.
2. interactive – документ прочитан, происходит примерно в то же время, что и DOMContentLoaded, но до него.
3. complete – документ и ресурсы загружены, происходит примерно в то же время, что и window.onload, но до него.

# Скрипты: async, defer

В современных сайтах скрипты обычно «тяжелее», чем HTML: они весят больше, дольше обрабатываются. Когда браузер загружает HTML и доходит до тега <script>...</script>, он не может продолжать строить DOM. Он должен сначала выполнить скрипт.

Это ведёт к двум важным проблемам:

1. Скрипты не видят DOM-элементы ниже себя, поэтому к ним нельзя добавить обработчики и т.д.
2. Если вверху страницы объёмный скрипт, он «блокирует» страницу. Пользователи не видят содержимое страницы, пока он не загрузится и не запустится:



Конечно, есть пути, как это обойти. Например, мы можем поместить скрипт внизу страницы.

Но это решение далеко от идеального. Например, браузер замечает скрипт (и может начать **загружать** его) только после того, как он полностью загрузил HTML-документ. В случае с длинными HTML-страницами это может создать заметную задержку.

К счастью, есть два атрибута тега <script>, которые решают нашу проблему: **defer** и **async**.

## Defer

Атрибут defer сообщает браузеру, что он должен продолжать обрабатывать страницу и **загружать** скрипт в фоновом режиме, а затем запустить этот скрипт, когда он загрузится:



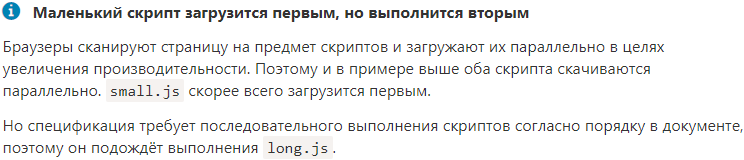
* Скрипты с defer никогда не блокируют страницу.
* Скрипты с defer всегда выполняются, когда дерево DOM готово, но до события DOMContentLoaded.

Следующий пример это показывает:



1. Содержимое страницы отобразится мгновенно.
2. Событие DOMContentLoaded подождёт отложенный скрипт. Оно будет сгенерировано, только когда скрипт (2) будет загружен и выполнен.

Отложенные с помощью defer скрипты сохраняют порядок относительно друг друга, как и обычные скрипты. Поэтому, если сначала загружается большой скрипт, а затем меньшего размера, то последний будет ждать.





## Async

Атрибут async означает, что скрипт абсолютно независим:

* Страница не ждёт асинхронных скриптов, содержимое обрабатывается и отображается.
* Событие DOMContentLoaded и асинхронные скрипты не ждут друг друга:
  + DOMContentLoaded может произойти как до асинхронного скрипта (если асинхронный скрипт завершит загрузку после того, как страница будет готова),
  + …так и после асинхронного скрипта (если он короткий или уже содержится в HTTP-кеше)
* Остальные скрипты не ждут async, и скрипты c async не ждут другие скрипты.

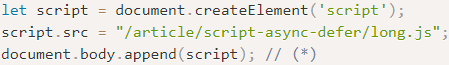
Так что если у нас есть несколько скриптов с async, они могут выполняться в любом порядке. То, что первое загрузится – запустится в первую очередь.

Асинхронные скрипты очень полезны для добавления на страницу сторонних скриптов: счётчиков, рекламы и т.д. Они не зависят от наших скриптов, и мы тоже не должны ждать их:



## Динамически загружаемые скрипты

Мы можем также добавить скрипт и динамически, с помощью JavaScript:



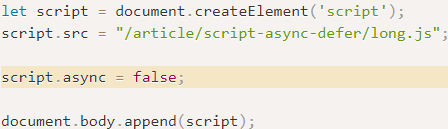
Скрипт начнёт загружаться, как только он будет добавлен в документ (\*).

**Динамически загружаемые скрипты по умолчанию ведут себя как «async».**

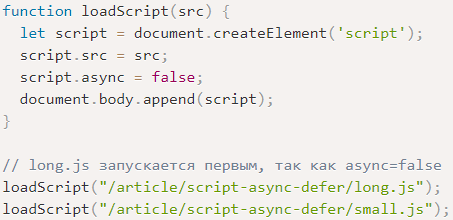
То есть:

* Они никого не ждут, и их никто не ждёт.
* Скрипт, который загружается первым – запускается первым

Мы можем изменить относительный порядок скриптов с «первый загрузился – первый выполнился» на порядок, в котором они идут в документе (как в обычных скриптах) с помощью явной установки свойства async в false:



Например, здесь мы добавляем два скрипта. Без script.async=false они запускались бы в порядке загрузки (small.js скорее всего запустился бы раньше). Но с этим флагом порядок будет как в документе:

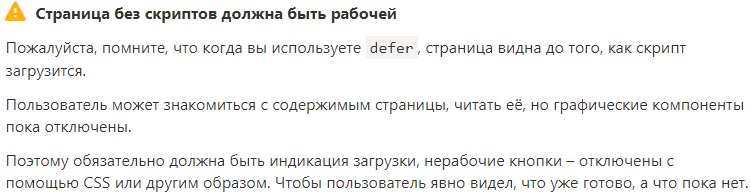


## Итого

У async и defer есть кое-что общее: они не блокируют отрисовку страницы. Так что пользователь может просмотреть содержимое страницы и ознакомиться с ней сразу же.

Но есть и значимые различия:





На практике defer используется для скриптов, которым требуется доступ ко всему DOM и/или важен их относительный порядок выполнения.

А async хорош для независимых скриптов, например счётчиков и рекламы, относительный порядок выполнения которых не играет роли.

# Загрузка ресурсов: onload и onerror

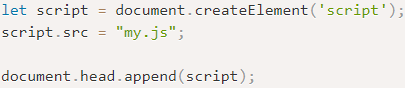
Браузер позволяет отслеживать загрузку сторонних ресурсов: скриптов, ифреймов, изображений и др.

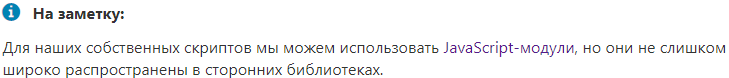
Для этого существуют два события:

* **load** – успешная загрузка,
* **error** – во время загрузки произошла ошибка.

## Загрузка скриптов

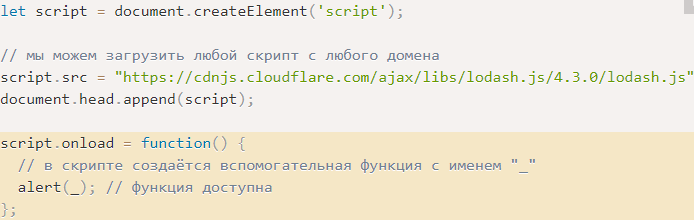
Допустим, нам нужно загрузить сторонний скрипт и вызвать функцию, которая объявлена в этом скрипте. Мы можем загрузить этот скрипт динамически:





## script.onload

Событие load срабатывает после того, как скрипт был загружен и выполнен:

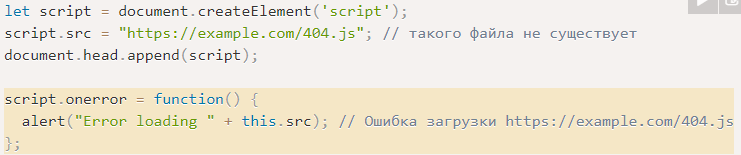


Таким образом, в обработчике onload мы можем использовать переменные, вызывать функции и т.д., которые предоставляет нам сторонний скрипт.

## script.onerror

Ошибки, которые возникают во время загрузки скрипта, могут быть отслежены с помощью события error.

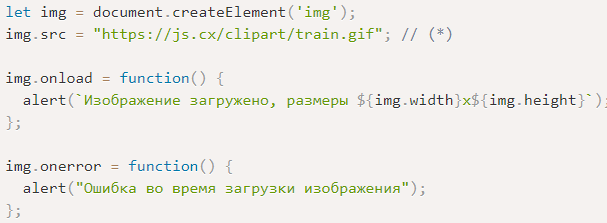
Например, давайте запросим скрипт, которого не существует:





## Другие ресурсы

События load и error также срабатывают и для других ресурсов, а вообще, для любых ресурсов, у которых есть внешний src:



Однако есть некоторые особенности:

* Большинство ресурсов начинают загружаться после их добавления в документ. За исключением тега <img>. Изображения начинают загружаться, когда получают src (\*).
* Для <iframe> событие load срабатывает по окончании загрузки как в случае успеха, так и в случае ошибки.

## Ошибка в скрипте с другого источника

<https://learn.javascript.ru/onload-onerror#oshibka-v-skripte-s-drugogo-istochnika>

# Событийный цикл: микрозадачи и макрозадачи

Поток выполнения в браузере, равно как и в Node.js, основан на *событийном цикле.*

## Событийный цикл

Идея событийного цикла очень проста. Есть бесконечный цикл, в котором движок JavaScript ожидает задачи, исполняет их и снова ожидает появления новых.

Общий алгоритм движка:

1. Пока есть задачи:

* выполнить их, начиная с самой старой

1. Бездействовать до появления новой задачи, а затем перейти к пункту 1

Это формализация того, что мы наблюдаем, просматривая веб-страницу. Движок JavaScript большую часть времени ничего не делает и работает, только если требуется исполнить скрипт/обработчик или обработать событие.

Примеры задач:

* Когда загружается внешний скрипт <script src="...">, то задача – это выполнение этого скрипта.
* Когда пользователь двигает мышь, задача – сгенерировать событие mousemove и выполнить его обработчики.
* Когда истечёт таймер, установленный с помощью setTimeout(func, ...), задача – это выполнение функции func
* И так далее.

Задачи поступают на выполнение – движок выполняет их – затем ожидает новые задачи.

Может так случиться, что задача поступает, когда движок занят чем-то другим, тогда она ставится **в очередь.** Очередь, которую формируют такие задачи, называют **«очередью макрозадач»** (**macrotask queue**, термин v8).

****

Например, когда движок занят выполнением скрипта, пользователь может передвинуть мышь, тем самым вызвав появление события mousemove, или может истечь таймер, установленный setTimeout, и т.п. Эти задачи формируют очередь, как показано на иллюстрации выше.

Задачи из очереди исполняются по правилу «первым пришёл – первым ушёл». Когда браузер заканчивает выполнение скрипта, он обрабатывает событие mousemove, затем выполняет обработчик, заданный setTimeout, и так далее.

Отметим две детали:

1. Рендеринг (отрисовка страницы) никогда не происходит во время выполнения задачи движком. Не имеет значения, сколь долго выполняется задача. Изменения в DOM отрисовываются только после того, как задача выполнена.
2. **Если задача выполняется очень долго**, то браузер не может выполнять другие задачи, обрабатывать пользовательские события, поэтому спустя некоторое время браузер предлагает «убить» долго выполняющуюся задачу. Такое возможно, когда в скрипте много сложных вычислений или ошибка, ведущая к бесконечному циклу.

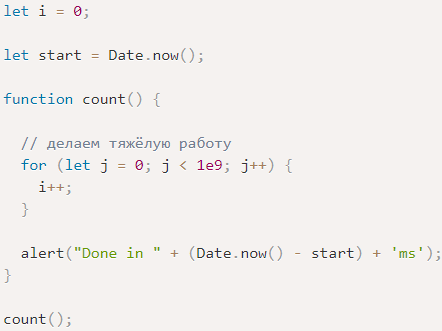
## Пример 1: разбиение «тяжёлой» задачи.

Пока движок занят тяжелой задачей, он не может делать ничего, связанного с DOM, не может обрабатывать пользовательские события и т.д. Возможно даже «подвисание» браузера, что совершенно неприемлемо.

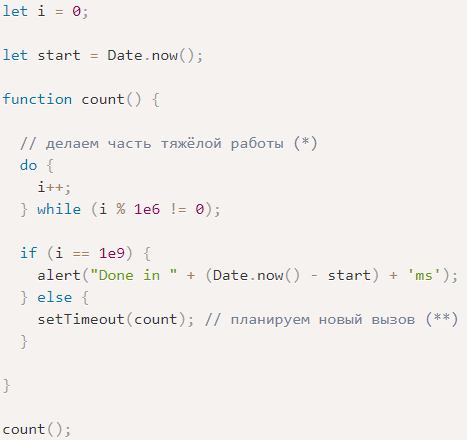
Мы можем избежать этого, разбив задачу на части. Сделать первые 100 частей, затем запланировать setTimeout (с нулевой задержкой) для разметки следующих 100 строк и т.д.

Чтобы продемонстрировать такой подход, давайте будем использовать для простоты функцию, которая считает от 1 до 1000000000.

При таком коде, браузер зависнем на некоторое время и не сможет даже вызывать обработчики событий на кнопках и т.д:



Давайте разобьём задачу на части, воспользовавшись вложенным setTimeout:



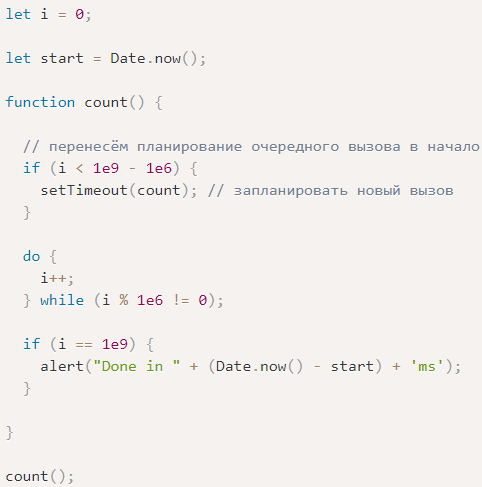
Теперь интерфейс браузера полностью работоспособен во время выполнения «счёта».

Один вызов count делает часть работы (\*), а затем, если необходимо, планирует свой очередной запуск (\*\*):

1. Первое выполнение производит счёт: i=1…1000000.
2. Второе выполнение производит счёт: i=1000001…2000000.
3. …и так далее.

Теперь если новая сторонняя задача (например, событие onclick) появляется, пока движок занят выполнением 1-й части, то она становится в очередь, и затем выполняется, когда 1-я часть завершена, перед следующей частью. Периодические возвраты в событийный цикл между запусками count дают движку достаточно «воздуха», чтобы сделать что-то ещё, отреагировать на действия пользователя.

Чтобы сократить разницу ещё сильнее, давайте немного улучшим наш код. Мы перенесём планирование очередного вызова в начало count():



Откуда сокращение времени? Как вы помните, в браузере есть минимальная задержка в 4 миллисекунды при множестве вложенных вызовов setTimeout. Даже если мы указываем задержку 0, на самом деле она будет равна 4 мс (или чуть больше). Поэтому чем раньше мы запланируем выполнение – тем быстрее выполнится код.

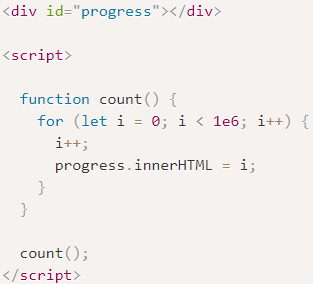
## Пример 2: индикация прогресса

Ещё одно преимущество разделения на части крупной задачи в браузерных скриптах – это возможность показывать индикатор выполнения.

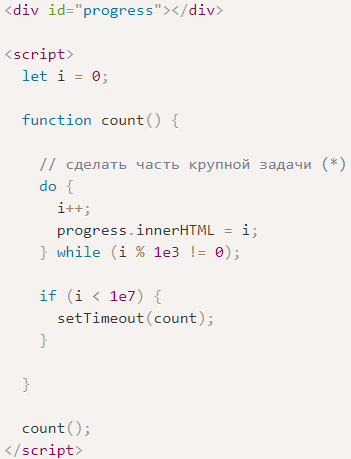
Обычно браузер отрисовывает содержимое страницы после того, как заканчивается выполнение текущего кода. Не имеет значения, насколько долго выполняется задача. Изменения в DOM отображаются только после её завершения.

С одной стороны, это хорошо, потому что наша функция может создавать много элементов, добавлять их по одному в документ и изменять их стили – пользователь не увидит «промежуточного», незаконченного состояния. Это важно, верно?

В примере ниже изменения i не будут заметны, пока функция не завершится, поэтому мы увидим только последнее значение i:



Если мы разобьём тяжёлую задачу на части, используя setTimeout, то изменения индикатора будут отрисованы в промежутках между частями:

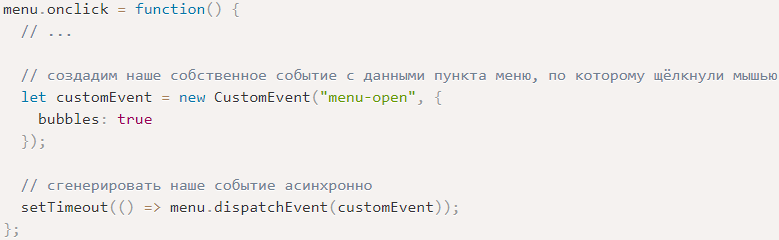


Теперь <div> показывает растущее значение i – это своего рода индикатор выполнения.

## Пример 3: делаем что-нибудь после события

В обработчике события мы можем решить отложить некоторые действия, пока событие не «всплывёт» и не будет обработано на всех уровнях. Мы можем добиться этого, обернув код в setTimeout с нулевой задержкой.

В главе [Генерация пользовательских событий](https://learn.javascript.ru/dispatch-events) мы видели пример: наше событие menu-open генерируется через setTimeout, чтобы оно возникло после того, как полностью обработано событие «click»:



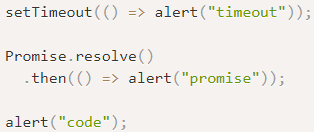
## Макрозадачи и Микрозадачи

Микрозадачи приходят только из кода. Обычно они создаются промисами: выполнение обработчика .then/catch/finally становится микрозадачей. Микрозадачи также используются «под капотом» await, т.к. это форма обработки промиса.

Также есть специальная функция queueMicrotask(func), которая помещает func в очередь микрозадач.

**Сразу после каждой МАКРОзадачи движок исполняет все задачи из очереди микрозадач перед тем, как выполнить следующую макрозадачу или отобразить изменения на странице, или сделать что-то ещё.**

Пример:



1. **code** появляется первым, т.к. это обычный синхронный вызов.
2. **promise** появляется вторым, потому что .then проходит через очередь микрозадач и выполняется после текущего синхронного кода.
3. **timeout** появляется последним, потому что это макрозадача.

**Все микрозадачи завершаются до обработки каких-либо событий или рендеринга, или перехода к другой макрозадаче.**

Это важно, так как гарантирует, что общее окружение остаётся одним и тем же между микрозадачами – не изменены координаты мыши, не получены новые данные по сети и т.п.

Если мы хотим запустить функцию асинхронно (после текущего кода), но до отображения изменений и до новых событий, то можем запланировать это через queueMicrotask.

Вот пример с индикатором выполнения, похожий на предыдущий, но в этот раз использована функция queueMicrotask вместо setTimeout. Обратите внимание – отрисовка страницы происходит только в самом конце. Как и в случае обычного синхронного кода:

