# Объектная модель документа

[Смотреть материал на видео](https://www.youtube.com/watch?v=UbUQqcy7e9s&list=PLA0M1Bcd0w8zri9wWiDFmOukXhguEYBBO" \t "_blank)

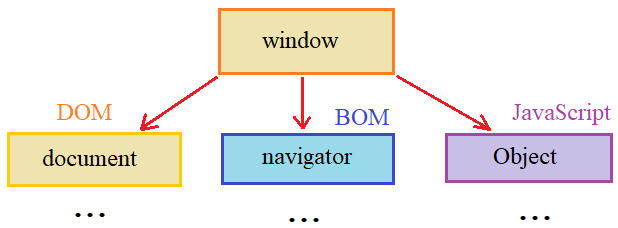
<https://proproprogs.ru/javascript_dom/obektnaya-model-dokumenta>

Программы, написанные на JavaScript, могут запускаться всюду, где установлена виртуальная JavaScript-машина. В основном – это браузеры и серверы, на которых установлен пакет Node.js. При этом, рассмотренные ранее, базовые конструкции языка должны одинаково выполняться в разных окружениях. Но каждое конкретное окружение дает некоторые дополнительные возможности. Например, при запуске JavaScript в браузере, мы имеем возможность управлять содержимым HTML-документа. А при выполнении скрипта на сервере, у него появляются дополнительные функции для работы с сетевыми запросами.

Начиная с этого занятия, мы будем рассматривать работу JavaScript в браузерной среде. В этом случае нам становится доступной:

* объектная модель документа (Document Object Model, DOM) – для управления содержимым HTML-документа;
* объектная модель браузера (Browser Object Model, BOM) – дополнительные объекты, предоставляемые браузером.

Во главе всех этих объектов стоит корневой объект window.



Причем, объект window глобальный также и для JavaScript. Например, используя объект window, мы можем получить высоту клиентской области браузера (в пикселах):

console.log(window.innerHeight);

а через объект document изменить стиль тега body:

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

через navigator получить сведения о платформе, на которой запущен браузер:

console.log(navigator.platform);

и так далее. Объекты DOM и BOM предоставляют функционал для работы с браузером и HTML-документом. И пришло время познакомиться с ними поближе.

## Дерево DOM

Как мы только что увидели, DOM представляет информацию HTML-документа в виде набора объектов. Например, document.body – это объект тега <body>. И так для всех тегов и их содержимых.

Предположим, что у нас имеется вот такая HTML-страница:

<!DOCTYPE html>

<html>

<head>

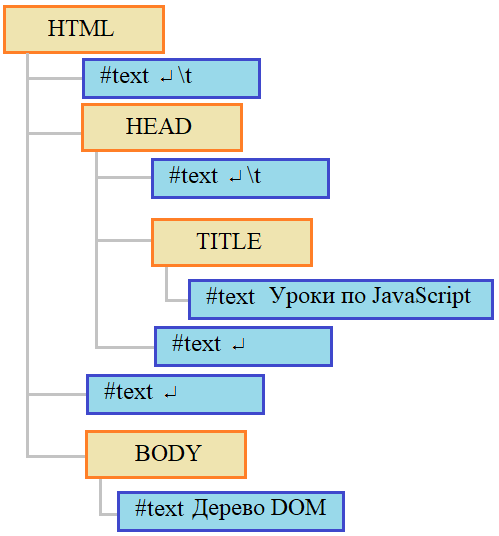
<title>Уроки по JavaScript</title>

</head>

<body>Дерево DOM</body>

</html>

Ее структура в виде DOM элементов будет выглядеть так:



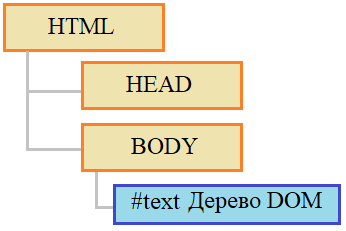
Здесь  - символ перевода строки (спецсимвол \n в JavaScript); \t – символ табуляции.

Каждый узел этого дерева – это объект. Соответственно, HTML – корневой узел, а HEAD и BODY – его дочерние элементы. Текст внутри элементов образует текстовые узлы (#text). Например, после тега <html> идет перевод на следующую строку – это элемент #text. После тега <head> также имеется перевод на следующую строчку с символом табуляции. И так далее. Все текстовые элементы обозначаются соответствующими узлами #text.

Если при построении дерева DOM браузер выявляет ошибки HTML-разметки, то он автоматически их корректирует и дерево создается всегда с корневым узлом HTML. Например, если мы создадим вот такой документ:

Дерево DOM

то браузер его автоматически достроит до вот такого:

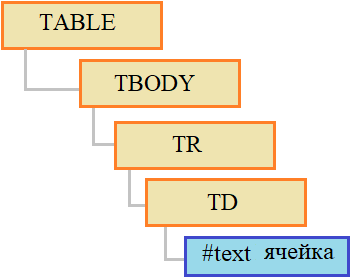


Если имеются какие-либо не закрытые теги (обычно это тег <p>), то в DOM-дереве все теги будут закрыты автоматически (правда, теги лучше закрывать при разметке, чтобы автоматическое закрытие не привело к ошибкам интерпретации HTML-документа).

Однако, в некоторых случаях автоматическое достроение дерева оказывается весьма полезным инструментом. Например, по стандарту DOM каждая таблица должна помещаться в тег <tbody>, но мы его обычно не пишем (это разрешено официально стандартом) и браузер добавляет его автоматически. Например:

<table><tr><td>ячейка</td></tr></table>

на уровне DOM выглядит так:



Эти моменты важно знать, чтобы избежать «сюрпризов» при работе с элементами через DOM.

Помимо тегов и текстовых полей в DOM есть еще один важный элемент – это комментарий. Да, комментарии тоже помещаются в DOM. Почему? Дело в том, что при построении дерева браузер руководствуется правилом: все что есть в HTML-документе должно быть в DOM. Поэтому и комментарии тоже помещаются в него. Например, вот такой наш прежний документ с комментарием:

<!DOCTYPE html>

<html>

<head>

<title>Уроки по JavaScript</title>

</head>

<body>

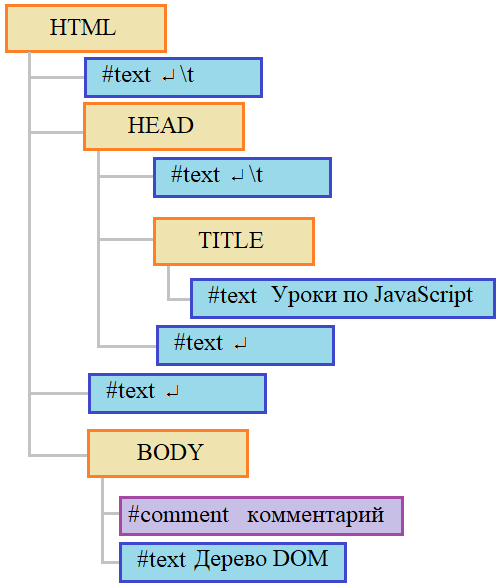
<!-- комментарий -->

Дерево DOM

</body>

</html>

Будет выглядеть так:



По идее директива <!DOCTYPE html> тоже помещается в это дерево как самый верхний узел, но мы ее не будем изображать для экономии пространства и времени.

Формально дерево DOM содержит 12 различных типов узлов. Но мы, в основном, будем использовать такие:

* document – объект, представляющий дерево DOM (точка входа в DOM);
* узлы-элементы – теги HTML-документа;
* текстовые узлы – элементы, содержащие текстовую информацию;
* комментарии – элементы с комментариями (иногда они используются JS для записи и чтения информации, скрытой от пользователя).

Чтобы увидеть дерево DOM в браузере, нужно открыть вкладку «Elements». Правда, дерево здесь отображается в несколько ином виде (текстовые узлы без полезной информации не показываются), но этого вполне достаточно, чтобы видеть его общую структуру.

# Навигация по дереву DOM

[Смотреть материал на видео](https://www.youtube.com/watch?v=X0GfPnS-zrc&list=PLA0M1Bcd0w8zri9wWiDFmOukXhguEYBBO" \t "_blank)

<https://proproprogs.ru/javascript_dom/navigaciya-po-derevu-dom>

На этом занятии рассмотрим способы навигации по DOM дереву и выбора нужного элемента в HTML-документе. Предположим, что у нас имеется вот такая страница:

<!DOCTYPE html>

<html>

<head>

<title>Уроки по JavaScript</title>

</head>

<body>

Абзац

<ul>

<li>Элемент списка</li>

</ul>

</body>

</html>

Мы не будем рисовать DOM-дерево этого документа, вы теперь можете делать это самостоятельно и, к тому же, сейчас в этом нет необходимости. Как мы говорили на предыдущем занятии, точкой входа для DOM является объект document. Чтобы с его помощью выбрать теги-объекты <html>, <head> и <body>, используются следующие свойства:

let html = document.documentElement;

let body = document.body;

let head = document.head;

console.log(html, body, head);

Для наглядности я буду писать программы непосредственно в тегах <script></script>, которые пропишем в конце документа:

<script></script>

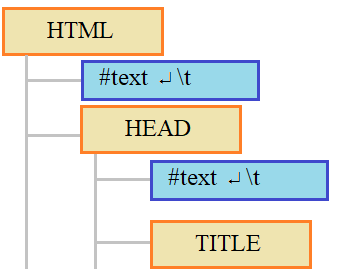
Обратите внимание, что мы прописали тег <script> в конце документа. Это сделано для того, чтобы браузер успел загрузить весь документ и создать полное DOM-дерево. Если мы его подключим, например, в разделе head, то переменная body будет равна null (что означает, что объект не существует). Вот это следует учитывать при работе с DOM: нужно быть уверенным, что браузер полностью загрузил документ, прежде чем работать с ним через JavaScript.

**Дочерние узлы и потомки**

Теперь посмотрим, как можно перебирать вложенные элементы этого дерева. Для начала введем два термина, чтобы мы понимали друг друга:

* **дочерние узлы** (или дети) – элементы, которые являются непосредственными детьми узла;
* **потомки** – все элементы, которые лежат внутри данного, включая детей, детей их детей и т.д.

Например, вот в этом фрагменте дерева head является дочерним узлом html, но title уже им не является, так как вложен в head. А вот потомки – это все вложенные элементы, значит, для html потомками будут и head и title. Вот это и гласят данные определения.



Для каждого узла дерева список его дочерних элементов можно получить через коллекцию childNodes, например, так:

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

    console.log( document.body.childNodes[i] );

}

Коллекция childNodes представляет собой псевдомассив, то есть, это итерируемый объект, но в нем отсутствуют методы массива. Из предыдущих занятий мы знаем, что для перебора элементов псевдомассива лучше использовать цикл for of:

**for** (let child of document.body.childNodes) {

    console.log( child );

}

Некоторые начинающим программисты применяют цикл for in для этой коллекции, что не стоит делать, по крайней мере, по двум причинам: это будет работать медленнее; будут доступны дополнительные свойства childNodes, которые обычно не нужны.

Если же мы хотим оперировать коллекцией childNodes как массивом, то ее можно преобразовать в массив:

let arr = Array.from(document.body.childNodes);

и, затем, использовать любые методы массивов, например:

arr.forEach((elem) => console.log(elem));

У каждого узла DOM-дерева имеются свойства **firstChild** и **lastChild**. Они обеспечивают доступ к первому и последнему дочернему элементу в списке childNodes. Например, так:

let html = document.documentElement;

let first = html.firstChild;

let last = html.lastChild;

console.log( first );         *//head*

console.log( last );          *//body*

По сути, это то же самое, что и такой выбор:

let html = document.documentElement;

let first = html.childNodes[0];

let last = html.childNodes[html.childNodes.length-1];

console.log( first );

console.log( last );

но записано в более краткой форме.

Если нам нужно узнать, есть ли у текущего элемента дочерние узлы, то используется метод hasChildNodes():

let html = document.documentElement;

console.log( html.hasChildNodes() );

Возвращает true, если есть и false, если нет.

Обратите внимание, что элементы коллекции childNodes доступны только для чтения. Менять их, используя синтаксис:

document.body.childNodes[0] = document.head.childNodes[0];

нельзя. Для удаления, добавления и изменения узлов дерева используются свои методы, о которых речь пойдет позже.

Наконец, если нам нужен только список дочерних тегов, то вместо коллекции childNodes используется коллекция children:

**for** (let child of document.documentElement.children) {

   console.log( child );

}

Теперь, мы видим только список тегов.

**Соседние элементы, родитель**

Теперь поговорим о переходах между соседними элементами. И введем такое понятие:

***Соседи****– это узлы, у которых один и тот же родитель.*

Например, вот в такой базовой структуре документа:

<html>

  <head>...</head><body>...</body>

</html>

объекты head и body являются соседями, т.к. у них общий родитель – объект html. Причем, для head объект body является **следующим правым**, а для body объект head – **предыдущий слева**. Отсюда название двух свойств:

* nextSibling – следующий сосед (справа);
* previousSibling – предыдущий сосед (слева).

Например (структура HTML-документа должна быть как выше):

console.log(document.head.nextSibling);

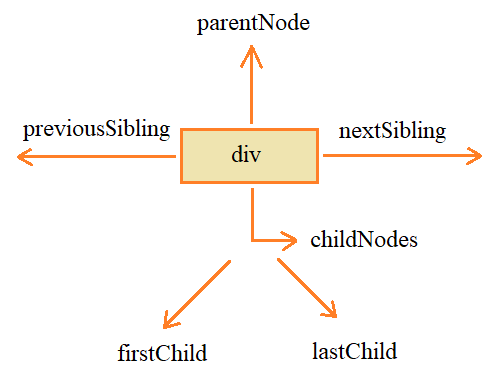
console.log(document.body.previousSibling);

В консоле увидим объект body и head.

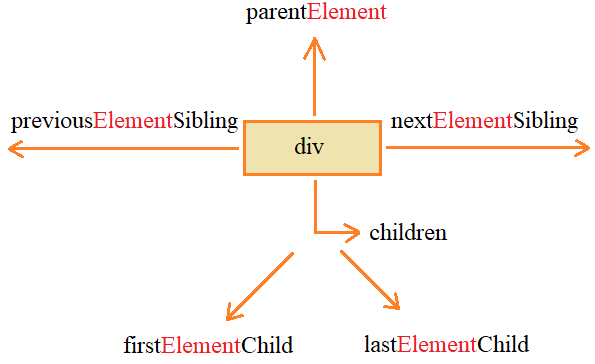
Наконец, если нам нужно получить объект родителя, то используется свойство parentNode:

console.log(document.body.parentNode);

Итак, относительно любого узла DOM-дерева можно осуществлять следующие переходы:



Если же нам нужны только узлы-элементы (как правило, это теги HTML-документа), то используется похожий набор свойств:



Здесь есть только одно ключевое отличие этих свойств (а, точнее, одного свойства parentElement), от ранее рассмотренных. В целом и parentNode и parentElement возвращают всегда один и тот же родительский элемент, кроме корневого элемента:

let html = document.documentElement;

console.log( html.parentNode ); *// выведет document*

console.log( html.parentElement ); *// выведет null*

Объект document не считается элементом HTML-страницы, поэтому ссылка parentElement для корня DOM-дерева возвращает null.

Некоторые типы DOM-элементов предоставляют для удобства дополнительные свойства, специфичные для их типа. Например, объект table имеет такие дополнительные свойства:

* 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).

Объекты tr имеют такие свойства:

* 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).

И объект td:

* td.cellIndex – номер ячейки в строке <tr>.

Они используются также как и описанные выше другие свойства узлов DOM-дерева. Например, если у нас имеется вот такая таблица:

<table id="table\_digs">

  <tr>

    <td>один</td><td>два</td>

  </tr>

</table>

То можно обратиться к ячейке со строкой два и вывести ее содержимое в консоль:

console.log( table\_digs.rows[0].cells[1].innerHTML );

Здесь свойство innerHTML содержит HTML-текст выбранной ячейки таблицы. О нем речь пойдет позже в других занятиях.

# Методы поиска элементов в DOM

[Смотреть материал на видео](https://www.youtube.com/watch?v=xuOYrDaS5lI&list=PLA0M1Bcd0w8zri9wWiDFmOukXhguEYBBO" \t "_blank)

<https://proproprogs.ru/javascript_dom/metody-poiska-elementov-dom>

На этом занятии рассмотрим методы поиска произвольных элементов в HTML-документе. Рассмотренные свойства на предыдущем занятии по навигации DOM-дереву хороши если элементы расположены рядом. Но что делать, если требуемый объект может находиться в самых разных местах. Как его искать? Вот об этом и пойдет сейчас речь.

В самом простом случае мы можем у любого тега HTML-документа прописать атрибут id с некоторым уникальным значением. Например:

<!DOCTYPE html>

<html>

<head>

    <title>Уроки по JavaScript</title>

</head>

<body>

<div id="div\_id">

    <p>Текст внутри блока div

</div>

<script>

</script>

</body>

</html>

Здесь у нас тег div имеет атрибут id со значением div\_id. Мы это значение придумываем сами, главное, чтобы оно было уникальным в пределах HTML-страницы. Теперь можно получить этот элемент div по этому id, где бы он ни находился в DOM-дереве. Для этого используется метод getElementById объекта document:

let divElem = document.getElementById('div\_id');

console.log( divElem );

Мы в методе getElementById в качестве аргумента указываем строку со значением атрибута id и на выходе получаем ссылку на этот элемент.

Или же можем получить доступ к этому элементу напрямую через глобальную переменную div\_id, которая автоматически создается браузером при формировании DOM-дерева:

console.log( div\_id );

Но этот второй способ имеет одну уязвимость: если мы создадим в скрипте переменную с таким же именем, то прежнее значение div\_id будет затерто:

let div\_id = "не тег div";

так что этот подход следует применять с большой осторожностью, лучше использовать метод getElementById.

В стандарте ES6+ появился новый метод поиска элементов querySelectorAll, который возвращает список элементов, удовлетворяющих CSS-селектору, который мы в нем указываем.

Например, добавим в HTML-документ вот такой маркированный список:

<ul>

<li>Солнце

<li>Меркурий

<li>Венера

<li>Земля

<li>Марс

</ul>

и вот такой нумерованный список:

<p>Звезды:

<ol>

<li>Сириус

<li>Альдебаран

<li>Капелла

<li>Ригель

</ol>

И теперь хотим выбрать все теги <li>, но только у маркированного списка. Для этого запишем метод querySelectorAll с таким CSS-селектором:

let list = document.querySelectorAll("ul > li");

**for**(let val of list)

   console.log(val);

Как видите, у нас были выбраны элементы только маркированного списка. Мало того, в методе querySelectorAll можно использовать псевдоклассы для указания более сложных CSS-селекторов, например, так:

let list = document.querySelectorAll("ul > li:first-child");

тогда мы увидим только первый элемент списка. И так далее. С помощью querySelectorAll можно довольно просто выбрать нужные элементы и далее производить с ними необходимые действия.

Если же нам нужно по определенному CSS-селектору найти только первый подходящий элемент, то для этого применяется другой метод querySelector:

let element = document.querySelector("ol > li");

console.log(element);

Здесь мы из нумерованного списка выбрали первый тег <li>. Конечно, здесь можно было бы использовать и предыдущий метод querySelectorAll, а затем, взять из списка только первый:

let element = document.querySelectorAll("ol > li")[0];

но это будет работать медленнее, так как все равно сначала будут находиться все подходящие элементы, а затем, браться первый. Поэтому, если нужно найти только первый подходящий, то следует использовать querySelector.

Следующий метод matches позволяет определить: подходит ли данный элемент под указанный CSS-селектор или нет. Если подходит, то возвращает true, иначе – false. Например, создадим такое оглавление:

<h1>О звездах</h1>

<div class="content-table">

<ul class="stars-list">

<li class="star">О сириусе</li>

<li class="star">Об альдебаране</li>

<li class="contact">Обратная связь</li>

</ul>

</div>

И мы, перебирая список пунктов меню, хотим выбрать только те элементы, у которых class равен star. Это можно сделать так:

let list = document.querySelectorAll("ul.stars-list > li");

**for**(let item of list) {

**if**(item.matches("li.star")) console.log(item);

}

Обратите внимание, что метод matches относится к объекту DOM, а не к document. Что, в общем-то логично, так как нам нужно проверить конкретный элемент на соответствие CSS-селектора. В результате, в консоле мы увидим первые два элемента:

<li class="star">О сириусе</li>

<li class="star">Об альдебаране</li>

Следующий метод elem.closest(css) ищет ближайшего предка, который соответствует CSS-селектору. Сам элемент также включается в поиск. Метод возвращает либо предка, либо null, если такой элемент не найден. Например:

let li = document.querySelector("li.star");

console.log(li.closest('.stars-list'));

console.log(li.closest('.content-table'));

console.log(li.closest('h1')); *// null*

Сначала мы выбираем первый элемент li пункта меню. Затем, с помощью метода closest ищем ближайшего родителя с классом stars-list. Находится список ul. Далее, ищем родителя с классом content-table. Находим блок div. Наконец, пытаемся найти родителя с тегом h1. Но его нет, так как h1 в документе не является родителем для объекта li. Получаем значение null.

В старых версиях языка JavaScript (стандарта ES5-) существуют следующие методы для поиска элементов:

* elem.getElementsByTagName(tag) ищет элементы с указанным тегом и возвращает их коллекцию. Указав "\*" вместо тега, можно получить всех потомков.
* elem.getElementsByClassName(className) возвращает элементы, которые имеют указанный CSS-класс.
* document.getElementsByName(name) возвращает элементы с заданным атрибутом name. (Используется очень редко).

Применение этих методов очевидно и я привел их здесь лишь для полноты картины, так как вы можете их встретить в старых скриптах. Вместо всех этих методов теперь используются методы querySelector и querySelectorAll. Они дают гораздо больше возможностей, чем старые методы.

Однако, между методами getElementsBy\* и querySelector, querySelectorAll есть одно важное отличие: методы getElementsBy\* возвращают, так называемую, «живую» коллекцию, то есть, они всегда отражают текущее состояние документа и автоматически обновляются при его изменении. Например, при загрузке и отображении такого HTML-документа:

<!DOCTYPE html>

<html>

<head>

<title>Уроки по JavaScript</title>

</head>

<body>

<h1>О звездах</h1>

<h2>Об альдебаране</h2>

<script>

let list = document.getElementsByTagName("h2");

</script>

<h2>О ригеле</h2>

<script>

**for**(let item of list) console.log(item);

</script>

</body>

</html>

Мы в консоле увидим список из двух тегов h2, хотя когда этот список формировался в HTML-документе был всего один тег h2. Второй добавился позже, автоматически. Но, если мы будем получать список этих элементов с помощью метода querySelectorAll:

   let list = document.querySelectorAll("h2");

то увидим только один тег h2. Так как здесь делается как бы снимок коллекции на текущий момент состояния HTML-документа и после этого никак не меняется. Вот этот момент при работе с этими методами следует иметь в виду.

Ну и в заключение этого занятия отметим еще один полезный метод

elemA.contains(elemB)

который возвращает значение true, если elemB является дочерним по отношению к elemA. И false в противном случае. Например, вот в этом документе:

<div class="content-table">

<ul class="stars-list">

<li class="star">О сириусе</li>

<li class="star">Об альдебаране</li>

<li class="contact">Обратная связь</li>

</ul>

</div>

Можно проверить: имеется ли список внутри тега div:

let div = document.querySelector("div.content-table");

let ul = document.querySelector("ul.stars-list");

**if**(div.contains(ul))

     console.log("ul внутри div");

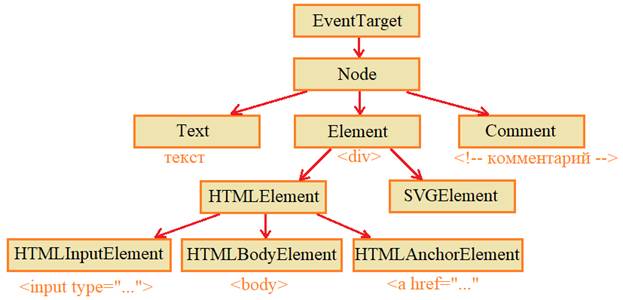
Вот такие основные методы поиска элементов в DOM-дереве есть в JavaScript.

# Свойства DOM-узлов

[Смотреть материал на видео](https://www.youtube.com/watch?v=tY5NgzEi03Q&list=PLA0M1Bcd0w8zri9wWiDFmOukXhguEYBBO" \t "_blank)

<https://proproprogs.ru/javascript_dom/svoystva-dom-uzlov>

На этом занятии мы с вами познакомимся с некоторыми часто используемыми свойствами DOM-узлов. Но для начала посмотрим на иерархию классов объектов, из которых составляется DOM-дерево.



Понимать этот рисунок можно так. Все элементы DOM содержат вот этот корневой класс EventTarget и класс Node, который наследуется от него. Далее, начинается специализация классов: Text – для текстовых элементов; Element – для тегов; Comment – для комментариев. Для чего введена такая специализация? В частности, это позволяет создавать специальные методы и свойства для различных типов DOM-узлов. Например, текстовые элементы имеют свои специфические свойства, которых нет у тегов-элементов или комментариев. А вот их общность наследования от классов Node и EventTarget предоставляет некоторые общие методы, например, обработку реакций на события от пользователей. Такого понимания этой иерархической структуры вполне достаточно для восприятия дальнейшей информации.

Итак, базовые классы EventTarget и Node являются абстрактными, то есть, они служат только для построения других (дочерних) классов, но сами по себе существовать не могут – только в составе других классов. Остальные объекты могут быть созданы и быть узлом DOM-дерева.

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

console.log( document.body.constructor.name );

или же можно проверить наследование с помощью оператора instanceof:

console.log( document.body **instanceof** HTMLBodyElement ); *// true*

console.log( document.body **instanceof** HTMLElement ); *// true*

console.log( document.body **instanceof** Element ); *// true*

console.log( document.body **instanceof** Text ); *// false*

Если мы хотим отобразить в консоле структуру объекта, то вместо console.log следует использовать console.dir:

console.dir(document.body);

а если записать

console.log(document.body);

то увидим содержимое тега body.

В старых стандартах языка JavaScript для определения типа узла DOM-дерева использовалось свойство nodeType, которое содержало числовое значение:

* elem.nodeType == 1 для узлов-элементов;
* elem.nodeType == 3 для текстовых узлов;
* …
* elem.nodeType == 9 для объектов документа.

console.log(document.body.nodeType);

Теперь используется оператор instanceof, хотя в ряде случаев бывает удобно обратиться к свойству nodeType.

Наконец, имя узла можно узнать по свойствам nodeName и tagName. У всех без исключения DOM-узлов имеется свойство nodeName, содержащее его название. Например, возьмем вот такой документ:

<!DOCTYPE html>

<html>

<head>

<title>Уроки по JavaScript</title>

</head>

<body><!-- комментарий --><h1>Заголовок страницы</h1>

<script>

</script>

</body>

</html>

И посмотрим свойство nodeName для комментария и тега:

let comm = document.body.firstChild;

console.log(comm.nodeName);

console.log(document.body.nodeName);

А вот если отобразить свойство tagName, то получим следующее:

console.log(comm.tagName);

console.log(document.body.tagName);

Видите, у комментария нет свойства tagName, оно есть только у тега-элемента и имеет то же значение, что и nodeName. Это справедливо и для всех остальных типов элементов: tagName – имя свойства тега-элемента; nodeName – имя свойства любого узла DOM-дерева. В общем случае мы всегда можем использовать nodeName для любых узлов, чтобы узнать его имя.

## innerHTML

У элементов-тегов имеется свойство innerHTML, позволяющее читать и менять содержимое любых тегов HTML-документа. Например, вот так

let h1 = document.body.firstChild.nextSibling;

console.log(h1.innerHTML);

можно вывести содержимое заголовка h1 в консоль. А вот так изменить его:

h1.innerHTML = "Измененный заголовок";

или, даже так:

h1.innerHTML = "Измененный <u>заголовок</u>";

Смотрите, в последнем примере мы добавили в заголовок не только текст, но и тег форматирования <u>. И этот тег был применен при отображении заголовка. Это важный момент: с помощью свойства innerHTML можно менять не только текст, но и структуру HTML-документа с добавлением новых тегов. Правда, здесь есть одно ограничение. Если мы вставляем тег скрипта <script>, то он добавляется на страницу, но не запускается. Например, пусть имеется тег:

<p id="para"></p>

и, далее в скрипте:

para.innerHTML = `<script>

alert("hello");

<\/script>`;

Скрипт не запустится. Также не следует полагать, что запись вида:

h1.innerHTML += " с добавленным текстом";

просто добавляет строку к уже имеющемуся содержимому. В этом случае сформируется новая строка:

"Измененный <u>заголовок</u> с добавленным текстом"

Затем, эта строка скопируется в h1.innerHTML и браузер полностью обновит весь заголовок. То есть, если вы, например, делаете чат и думаете таким образом добавлять в него информацию с новыми сообщениями, то это приведет не только к добавлению, но и обновлению содержимого всего чата. И это может быть нежелательным эффектом: представьте, что все картинки и видео у пользователя начинают грузиться заново! Наверное, это не то, что хотелось бы? Вот это следует учитывать при работе со свойством innerHTML.

## outerHTML

Следующее свойство тега-элемента outerHTML работает также как и innerHTML, но содержит информацию не только о содержимом тега, но и сам тег. С его помощью мы можем изменять тег целиком, то есть, на его место записывать другой тег. Например:

h1.outerHTML = "<h2>Измененный <u>заголовок</u> из h1 в h2";

Теперь у нас вместо заголовка h1 появляется заголовок h2 с новым содержимым. И здесь есть одна тонкость: переменная h1 продолжает содержать информацию о заголовке h1:

console.dir(h1);

То есть, мы на странице заменили этот заголовок, но в переменной прежняя информация продолжает существовать. В частности, если вывести содержимое свойства outerHTML, то увидим прежний заголовок:

console.log(h1.outerHTML);

Поэтому, как только мы заменяем какой-либо элемент через свойство outerHTML, прежнюю переменную следует либо обновить новой информацией, либо отбросить и в дальнейшем не использовать.

## nodeValue и data

Следующие два свойства – это nodeValue и data. Как мы отмечали выше, свойство innerHTML хранит информацию только о содержимом тегов-элементов. Доступ к данным остальных типов узлов осуществляется через эти два свойства: nodeValue и data. Они, в общем то, эквивалентны, поэтому на практике чаще используют data, так как оно  короче в записи. Например, отобразим содержимое комментария:

let comm = document.body.firstChild;

console.log(comm.data);

И в консоле увидим слово «комментарий», которое записано внутри комментария. Причем, эти свойства nodeValue и data отсутствуют у тегов-элементов. Если мы возьмем тег body и обратимся к свойству data, то получим значение undefined:

console.log(document.body.data);

## textContent

Следующее полезное свойство тегов – textContent, которое содержит чистый текст без тегов. Например, отобразим текст содержимое тега body, получим:

console.log(document.body.textContent);

Как видите в консоле у нас только текст, прописанный внутри этого элемента без каких-либо тегов.

Где и когда используется это свойство. Представьте, что вы создали страницу, где каждый пользователь может оставить комментарий. Ушлые посетители вашего сайта обязательно захотят попробовать свои недюжие хакерские способности, чтобы исказить содержимое страницы или даже нарушить работу всего сайта. Как они это будут делать? Да очевидно – прописывать теги в своих посланиях, в том числе и теги <script> с надеждой выполнить свой скрипт на вашей странице. Для защиты от таких нехитрых атак как раз и можно использовать свойство textContent следующим образом. Предположим, на нашей странице имеется раздел div с комментариями:

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

И мы просим пользователя ввести какую-либо строчку:

let msg = prompt("Ваш комментарий:", "");

и, затем, добавляем его в блок div:

comm.textContent = msg;

Если теперь пользователь вводит что-то вроде:

<script>Это мой вредоносный скрипт</script>

то в комментариях появится ровно такая строка вместе с тегами и никакого вреда сайту нанесено не будет. А вот, если бы мы использовали свойство innerHTML:

comm.innerHTML = msg;

то теги бы были прописаны в структуру документа:

это текст вместе с тегом

что мы и видим на нашей странице. Вот ключевое отличие между textContent и innerHTML.

## hidden

Данный атрибут позволяет скрывать элемент, то есть, не отображать его на экране браузера, например:

<div id="comm">Сообщения от пользователей</div>

и далее:

let comm = document.getElementById("comm");

comm.hidden = **true**;

Или так:

setInterval(()=> comm.hidden = !comm.hidden, 500 );

тогда наш блок div будет то пропадать, то появляться через каждые пол секунды. В целом, это свойство подобно свойству style="display:none", но его удобнее использовать.

Конечно, узлы DOM-дерева имеют и множество других свойств. Но, чтобы не перегружать занятие информацией, мы будем с ними постепенно знакомиться по мере прохождения материала.

# Нестандартные свойства DOM-узлов

[Смотреть материал на видео](https://www.youtube.com/watch?v=bwGyCpFadZg&list=PLA0M1Bcd0w8zri9wWiDFmOukXhguEYBBO" \t "_blank)

<https://proproprogs.ru/javascript_dom/nestandartnye-svoystva-dom-uzlov>

В наших прошлых занятиях мы не раз отмечали, что узлы DOM-дерева – это объекты языка JavaScript. Но раз это так, значит, мы можем добавлять в них свои собственные свойства. Например, возьмем объект body:

let body = document.body;

и пропишем в нем новое свойство myStyle:

body.myStyle = "color: red";

и отобразим его в консоле:

console.log(body.myStyle);

Видим, что у нашего объекта добавилось новое свойство. Мы можем даже поступить так:

body.myStyle = {

color: "red",

fontSize: 20

}

console.log(body.myStyle.fontSize);

И теперь, наше свойство ссылается на объект со стилями. Наконец, никто нам не мешает прописать и новый метод для объекта body:

body.getFontSize = **function**() {

**return** **this**.myStyle.fontSize;

}

console.log( body.getFontSize() );

Как видите это все работает, так как DOM-объекты – это обычные объекты JavaScript.

Однако, с добавлением новых свойств и методов нужно быть осторожными. Если какое-либо свойство уже существует и мы пытаемся его изменить, то оно либо изменится на другое или же возникнет ошибка, если свойство неизменяемое. Но как понять: какие свойства есть у DOM-объекта, а каких нет. Один из вариантов – просмотреть существующие свойства с помощью вывода объекта в консоль через метод dir:

console.dir( body );

Некоторые из этих свойств могут быть изменены, а другие – нет. Причем, значения этих свойств присваиваются автоматически браузером при создании DOM-дерева. Например, у нас имеется такая строчка:

<div id="comm">Сообщения от пользователей</div>

Здесь автоматически будет создан объект div со свойством id, равным comm. Можем в этом убедиться, если выведем его в консоль:

let div = document.getElementById("comm");

console.dir( div );

Вообще, всем стандартным атрибутам автоматически присваиваются соответствующие значения. Но, что если мы прописываем в теге какой-нибудь нестандартный атрибут, например:

<div id="comm" deflt="значение">Сообщения</div>

Тогда при попытке его вывести в консоль

console.log( div.deflt );

мы увидим значение undefined. Это говорит о том, что оно не создано в явном виде. Но это не значит, что его нет. Для работы с такими нестандартными свойствами следует пользоваться следующими методами:

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

Например:

console.log( div.getAttribute("deflt") );

div.setAttribute("deflt", "новое значение");

console.log( div.getAttribute("deflt") );

А полный список указанных атрибутов в HTML-документе, хранится в коллекции attributes:

**for**(let val of div.attributes)

console.log(val);

Здесь тонким моментом является то, что все значения нестандартных атрибутов могут быть только строками: не числами, не объектами, а только строками. Даже если мы указываем какой-то другой тип данных, он автоматически будет приведет к строке. Вот это следует иметь в виду.

Стандартные атрибуты могут принимать другие типы данных. Например, если у нас есть вот такой checkbox:

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

то при выводе типа свойства checked:

let inp = document.getElementById("input");

console.log( **typeof** inp.checked );

увидим булевый тип. Хотя большинство стандартных свойств все же являются строками.

Теперь давайте подробнее посмотрим для чего могут быть использованы нестандартные атрибуты. В первом случае они могут помочь нам выбрать необходимые узлы DOM-дерева и прописать им заданные свойства. Например, пусть у нас имеется вот такая таблица:

<table><tr>

<td cell-red="#CC0000">1</td><td cell-blue="#0000CC">2</td>

<td cell-red="#CC0000">3</td><td cell-blue="#0000CC">4</td>

</tr>

<tr>

<td cell-blue="#0000CC">5</td><td cell-red="#CC0000">6</td>

<td cell-blue="#0000CC">7</td><td cell-red="#CC0000">8</td>

</tr></table>

Для перебора всех ячеек с атрибутом cell-red можно воспользоваться таким скриптом:

**for**(let cell of document.querySelectorAll('[cell-red]')) {

   let attr = cell.getAttribute('cell-red');

   cell.style.background = attr;

}

Мы здесь с помощью querySelectorAll выбрали все элементы с атрибутом cell-red и, затем, присвоили им указанный цвет фона. В ряде случаев это может быть очень удобно.

Во втором случае мы можем использовать эти атрибуты в таблице стилей, например, так:

<style>

td[cell-red="#CC0000"] {

background: #CC0000;

}

td[cell-red="lightRed"] {

background: #FF9999;

}

</style>

И далее, через JavaScript мы можем управлять этими стилями так:

let flSkip = **false**;

**for**(let cell of document.querySelectorAll('[cell-red]')) {

**if**(!flSkip) cell.setAttribute('cell-red', 'lightRed');

   flSkip = !flSkip;

}

В ряде случаев это добавляет гибкости и элегантности в управлении элементами HTML-документа.

Но здесь, как всегда, есть одна тонкость: что если нестандартный атрибут cell-red появится в будущих версиях языка JavaScript и станет уже стандартным? Это может привести к непредсказуемому поведению нашей программы в будущем. Чтобы такого не было создатели языка JavaScript решили, что все атрибуты, начинающиеся с префикса

data-

будут заведомо восприниматься как нестандартные. Поэтому, наш скрипт лучше переписать так:

<table><tr>

<td data-cell-red="#CC0000">1</td>

<td data-cell-blue="#0000CC">2</td>

<td data-cell-red="#CC0000">3</td>

<td data-cell-blue="#0000CC">4</td>

</tr>

<tr>

<td data-cell-blue="#0000CC">5</td>

<td data-cell-red="#CC0000">6</td>

<td data-cell-blue="#0000CC">7</td>

<td data-cell-red="#CC0000">8</td>

</tr></table>

Стили будут выглядеть так:

<style>

td[data-cell-red="#CC0000"] {

background: #CC0000;

}

td[data-cell-red="lightRed"] {

background: #FF9999;

}

</style>

И скрипт примет вид:

let flSkip = **false**;

**for**(let cell of document.querySelectorAll('[data-cell-red]')) {

**if**(!flSkip) cell.setAttribute('data-cell-red', 'lightRed');

  flSkip = !flSkip;

}

Мало того, все эти свойства с префиксом data- становятся доступными через специальную коллекцию dataset, которая имеется у всех DOM-элементов:

**if**(!flSkip) cell.dataset.cellRed = 'lightRed';

причем, сам префикс data- опускается и оставшееся имя записывается по следующему правилу:

* если свойство использует дефисы, то в dataset оно прописывается в верблюжьей нотации (например, data-cell-red => cellRed);
* в остальных случаях просто используется оставшееся имя (например, data-color => color).

Вот так можно добавлять свои собственные свойства, методы и атрибуты в объекты DOM-дерева.

# Создание и добавление элементов DOM

[Смотреть материал на видео](https://www.youtube.com/watch?v=TjiaQrKA5fk&list=PLA0M1Bcd0w8zri9wWiDFmOukXhguEYBBO" \t "_blank)

<https://proproprogs.ru/javascript_dom/sozdanie-i-dobavlenie-elementov-dom>

На этом занятии мы поговорим о том как можно добавлять и удалять элементы в DOM-дереве, то есть, фактически в HTML-документе. И начнем с такого примера. Предположим, мы хотим создать вот такое сообщение:

<!DOCTYPE html>

<html>

<head>

<title>Уроки по JavaScript</title>

</head>

<style>

.msg {

width: 400px;

margin: 0 auto;

text-align: center;

background: #20454B;

padding: 5px;

border-radius: 5px;

color: #eee;

font-size: 24px;

}

</style>

<body>

<div class="msg">Важная информация!</div>

<script>

</script>

</body>

</html>

Но будем это делать посредством JavaScript. Поэтому тег div уберем. Теперь, чтобы создать элемент div, воспользуемся методом

document.createElement(tag)

и пропишем его так:

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

Далее, укажем класс стилей и содержимое:

div.className = "msg";

div.innerHTML = "Важная информация!";

Все, теперь переменная div является объектом для представления тега div с классом msg и строкой «Важная информация!». Но этот объект пока еще не является частью DOM-дерева, а значит, не содержится в HTML-документе. Если мы сейчас обновим страницу, то ничего на ней не увидим.

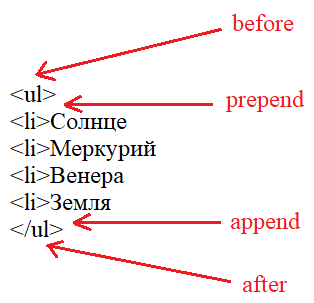
Для размещения объекта на HTML-странице существует несколько методов. Сейчас мы воспользуемся методом append, который может быть вызван из любого узла DOM-дерева для добавления объекта как дочернего в конец списка. Например, добавим наш div в раздел body:

document.body.append(div);

И если теперь посмотрим структуру HTML-документа, то увидим, что div идет последним в списке дочерних элементов раздела body.

Кроме метода append существуют такие методы вставки:

* 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.



Обратите внимание, что эти методы могут вставлять не только объекты-теги, но и текстовые элементы как строки. Чтобы все было понятнее, рассмотрим работу этих методов на таком примере. Пусть у нас имеется документ:

<h1>Солнечная система</h1>

<ul>

<li>Солнце

<li>Меркурий

<li>Венера

<li>Земля

</ul>

И выполним следующие операции:

let list = document.querySelector("ul");

list.before('before');

list.after('after');

В результате до и после списка вставятся текстовые элементы со значениями before и after. Далее:

let li\_1 = document.createElement('li');

li\_1.innerHTML = "первый элемент";

list.prepend(li\_1);

let li\_2 = document.createElement('li');

li\_2.innerHTML = "последний элемент";

list.append(li\_2);

будет добавлен первый и последний элемент в список. И, наконец, вот такая операция:

list.replaceWith(document.createElement("hr"),

                       "замена",

                       document.createElement("br"));

заменяет весь список новой информацией: тегом hr, строкой «замена» и тегом br.

Вот этот последний пример показывает, что данные методы могут принимать сразу более одного аргумента для вставки нескольких элементов.

Методы append/prepend/before/after позволяют также перемещать уже существующие элементы. То есть, когда мы вставляем имеющийся элемент, он автоматически удаляется со своего прежнего места и добавляется в новое. Например, у нас есть исходный список и мы хотим «солнце» поместить после «земли». Это можно сделать так:

let li = document.querySelector("ul.list > li:first-child");

let list = document.querySelector("ul.list");

list.append(li);

В JavaScript помимо создания элементов-тегов с помощью метода createElement можно создавать и текстовые элементы. Для этого следует использовать метод

document.createTextNode(text)

например, так:

let textNode = document.createTextNode('Текстовый элемент');

document.body.append(textNode);

В результате мы получим простой текст внутри HTML-документа.

## insertAdjacentHTML

Рассмотренные выше методы вставки добавляют либо отдельные теги, либо текстовую информацию. Причем, текст вставляется именно как текст, то есть, теги внутри текста будут отображены как текст. Например:

document.body.append("Текст с тегом");

Но, что если мы хотим вставить строку как HTML-строку с обработкой всех тегов, а не как обычный текст? Как раз для этого используется метод insertAdjacentHTML, который имеет следующий синтаксис:

elem.insertAdjacentHTML(where, html)

здесь where может принимать значения:

* "beforebegin" – для вставки html непосредственно перед elem;
* "afterbegin" – для вставки html как первого дочернего элемента в elem;
* "beforeend" – для вставки html как последнего дочернего элемента в elem;
* "afterend" – для вставки html непосредственно после elem.

Для примера возьмем вот такой список:

<ul>

<li>Меркурий

<li>Венера

<li>Земля

</ul>

И сделаем такие вставки:

let list = document.querySelector("ul.list");

list.insertAdjacentHTML("beforebegin", "Список планет<hr>");

list.insertAdjacentHTML("afterend", "<hr>Конец списка");

list.insertAdjacentHTML("afterbegin", "<li>Солнце");

list.insertAdjacentHTML("beforeend", "<li>Марс");

Смотрите как разместились эти элементы.



Существует еще два похожих метода, но более специализированных:

* insertAdjacentText(where, text) – для вставки строки текста text;
* insertAdjacentElement(where, elem) – для вставки элемента elem.

Например:

list.insertAdjacentText("**beforebegin", "Список планет<hr>**");

вставит текст как строку, а

let li = document.createElement("**li**");

li.innerHTML="**Солнце**";

list.insertAdjacentElement("**afterbegin**", li);

добавит соответствующий элемент.

Однако, на практике эти два метода почти не применяются. Вместо них удобнее использовать методы append/prepend/before/after просто потому, что легче писать. Если нам нужно удалить какой-либо узел из DOM-дерева, то для этого часто используется метод

node.remove()

Например, имеется список:

<ul class="list">

<li>Солнце

<li>Меркурий

<li>Венера

<li>Земля

<li>Марс

</ul>

И мы из него хотим из него удалять последние пункты, пока полностью не очистим. Это можно сделать так:

let idRemove = setInterval(**function**() {

let li = document.querySelector("ul.list > li:last-child");

**if**(li === **null**) {

clearInterval(idRemove);

alert("Список удален");

}

**else** li.remove();

}, 500);

## cloneNode

Следующий метод cloneNode позволяет создавать клон узла DOM-дерева. Он имеет следующий синтаксис:

elem.cloneNode(flDeep);

Если flDeep равен true, то создается «глубокий» клон объекта со всеми его свойствами и дочерними элементами. При значении false получаем клон без дочерних элементов.

Когда может пригодиться такая операция. Например, у нас есть вот такая таблица:

<table border=1 cellpadding="10">

<tr><td>Солнце</td><td>Звезда</td></tr>

<tr><td>Меркурий</td><td>Планета</td></tr>

</table>

Мы хотим добавить в нее еще одну строку с планетой Венерой. Это можно сделать путем клонирования существующей строки и добавления клона в конец таблицы:

let t = document.querySelector("table");

let r = document.querySelector("table>tbody>tr:last-child");

let row = r.cloneNode(**true**);

row.firstChild.innerHTML = "Венера";

t.append(row);

Все, теперь у нас появилась еще одна строчка с Венерой.

## DocumentFragment

В JavaScript есть DOM-объект специального вида – DocumentFragment. Он как бы образует фрагмент документа со своим DOM-деревом, то есть, со своим содержимым. Затем, это содержимое можно целиком вставить в любое место HTML-документа. Приведем такой простой пример. Предположим, имеется пустой список:

<ul></ul>

И мы хотим в него вставить заготовленный фрагмент списка. Это может выглядеть так:

let fr = **new** DocumentFragment();

let list=["Меркурий", "Венера", "Земля", "Марс"];

**for**(let p of list) {

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

item.innerHTML = p;

fr.append(item);

}

let ul = document.querySelector("ul");

ul.append(fr);

Но это лишь пример использования DocumentFragment. В данном конкретном случае все можно реализовать проще:

let list=["Меркурий", "Венера", "Земля", "Марс"];

let ul = document.querySelector("ul");

**for**(let p of list) {

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

item.innerHTML = p;

ul.append(item);

}

Вообще DocumentFragment используется крайне редко. Я здесь рассказал о нем, скорее, для целостности изложения материала. Хотя в ряде случаев он незаменим, например, при создании шаблонов документов. Но это выходит за рамки данного занятия.

Также приведу список некоторых устаревших методов вставки и удаления элементов, так как вы можете их встретить в старых скриптах и знали бы, что это такое:

* parent.appendChild(node) – добавляет элемент в конец списка дочерних элементов;
* parent.insertBefore(node, nextSibling) – вставляет элемент перед nextSibling;
* parent.removeChild(node) – удаляет элемент node (здесь parent является родителем node);
* parent.replaceChild(newElem, node) – заменяет дочерний элемент node на новый newElem.

Все эти методы работают вполне очевидным образом.

## document.write

Это довольно древний метод, который появился в JavaScript когда даже не было понятия DOM-объектов и DOM-дерева. Данный метод позволяет добавлять HTML-содержимое в то место, где записан и имеет такой синтаксис:

document.write(html);

Причем, срабатывает этот метод только в момент загрузки страницы. Если вызвать его после загрузки, то все содержимое HTML-документа будет удалено!

В качестве примера приведу такую страницу:

<h1>Солнечная система</h1>

<script>

document.write('<p>Строка вставленная методом write');

</script>

<ul>

<li>Солнце

<li>Меркурий

<li>Венера

<li>Земля

</ul>

Если же мы выполним этот метод после полной загрузки, то получим следующее:

setTimeout(**function**() {

    document.write('Строка вставленная методом write');

}, 500);

Видите? Все содержимое HTML-документа пропало и осталась только эта одна строка.

Но, несмотря на все эти особенности, метод document.write иногда применяют для динамического формирования HTML-страницы. Дело в том, что он срабатывает еще до построения DOM-дерева и все что он добавит в документ воспринимается браузером так, словно такая страница изначально была получена с сервера. И, в частности, так можно динамически добавлять теги <script> с соответствующим содержимым и они будут исполняться после загрузки документа. Это существенный плюс данного метода в сравнении с другими рассмотренными на этом занятии. Но, все-таки, он все реже применяется и постепенно уходит в небытие.

Итак, по итогам этого занятия вы должны знать следующие методы:

|  |  |  |
| --- | --- | --- |
| Методы создания узлов | | |
| document.createElement(tag) | создает указанный объект-тег | |
| document.createTextNode(value) | создает текстовый узел | |
| elem.cloneNode(deep) | создает клон элемента | |
| Методы вставки и удаления узлов | | |
| elem.append(node [или text]) | | добавляет node (или текст) как последний дочерний элемент |
| elem.prepend(node [или text]) | | добавляет node (или текст) как первый дочерний элемент |
| elem.before(node [или text]) | | добавляет node (или текст) непосредственно перед объектом elem |
| elem.after(node [или text]) | | добавляет node (или текст) непосредственно после объекта elem |
| elem.replaceWith(node [или text]) | | заменяет объект elem другим объектом или текстовым элементом |
| node.remove() | | удаляет элемент |
|  |  |  |

Также желательно знать и устаревшие методы, так как они могут встретиться в сторонних скриптах:

|  |  |
| --- | --- |
| Методы вставки и удаления узлов | |
| parent.appendChild(node) | добавляет node как последний дочерний элемент у parent |
| parent.insertBefore(node, nextSibling) | вставляет элемент перед nextSibling |
| parent.removeChild(node) | удаляет элемент node (здесь parent является родителем node) |
| parent.replaceChild(newElem, node) | заменяет дочерний элемент node на новый newElem |
| elem.insertAdjacentHTML(where, html) | вставляет фрагмент html в зависимости от значения where |
| Отдельный метод вставки в момент загрузки | |
| document.write(html) | вставка фрагмента html в момент загрузки HTML-страницы |

# Управление стилями

[Смотреть материал на видео](https://www.youtube.com/watch?v=sUcxJhVnoJI&list=PLA0M1Bcd0w8zri9wWiDFmOukXhguEYBBO" \t "_blank)

<https://proproprogs.ru/javascript_dom/upravlenie-stilyami>

В любом HTML-документе имеется оформление отображаемой информации. Например, цвет и размер шрифта, различные отступы между элементами на странице, тени и границы блоков и многое другое. Пришло время узнать как можно управлять таким оформлением на уровне JavaScript.

<!DOCTYPE html>

<html>

<head>

<title>Уроки по JavaScript</title>

</head>

<style>

.msg {

width: 400px;

margin: 0 auto;

text-align: center;

background: #20454B;

padding: 5px;

border-radius: 5px;

color: #eee;

font-size: 24px;

box-shadow: 5px 5px 4px 4px #aaa;

}

</style>

<body>

<div class="msg">Важная информация!</div>

<script>

</script>

</body>

</html>

На данный момент вам должно быть известно, что внешний вид того или иного тега в документе следует прописывать в каскадных таблицах стилей (CSS) и подключать через атрибут class, например:

<div class="msg">Информация</div>

Второй способ – это использовать атрибут style и уже в нем в виде строки записывать стили оформления:

<div style="position:absolute;top:200px;left:50%">

    Информация 2

</div>

Но такой подход следует использовать только в том случае, если нам по каким то причинам не подходит оформление через класс. Например, как показано здесь – указать положение элемента в окне браузера.

Итак, чтобы на уровне JavaScript узнать, какие классы подключены к определенному элементу, используется свойство объекта

elem. className

А для управления свойствами в атрибуте style, идет обращение к объекту:

elem.style

Например:

let elem = document.querySelector("div.msg");

console.log(elem.className);

let elem2 = elem.nextSibling.nextSibling;

console.log(elem2.style.top);

Здесь мы дважды обращаемся к свойству nextSibling для перехода ко второму div, так как не нужно забывать, что перенос строки – это текстовый элемент, стоящий между этими двумя дивами.

Оба этих свойства className и style доступны и для чтения и для записи. Например, определим вот такой класс стилей:

<style>

.message {

width: 200px;

padding: 10px;

background: #eee;

}

</style>

И заменим существующий на него:

let elem = document.querySelector("div.msg");

elem.className = "message";

Теперь наш div отображается с этими новыми стилями. Но что, если у нас есть вот такое расширение класса стилей:

.message.ex {

       color: #0000CC;

}

и мы хотим его применить к элементу div? Напомню, что через атрибут это расширение записывается так:

<div class="message ex">…</div>

то есть, через пробел. На уровне JavaScript для добавления (или удаления) таких дополнений используется коллекция

elem.classList

у которой есть такие методы:

* 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.

Добавим наше расширение к элементу div:

elem.classList.add("ex");

Теперь мы видим синий цвет текста, то есть, дополнение было успешно применено. По аналогии работают и все остальные методы свойства classList.

Свойство classList является итерируемым объектом, поэтому мы можем перебирать его элементы с помощью цикла for of:

**for**(let cl of elem.classList)

    console.log(cl);

Перейдем теперь к свойству style. Мы также можем напрямую управлять стилями элемента через него. Например, переместим наш второй div на несколько пикселей на экране браузера. Сделаем это так:

let elem2 = elem.nextSibling.nextSibling;

let coordY = 0;

let idMove = setInterval(**function**() {

       coordY += 5;

       elem2.style.top = coordY+"px";

}, 50);

setTimeout(()=>clearInterval(idMove), 3000);

В результате наш элемент будет динамически смещаться по вертикали в течение трех секунд. Обратите внимание на указание единиц измерения «px». Без этого свойство top работать не будет. Единицы измерения у всех подобных свойств, например, width, left, margin, font-size и т.п. указывать обязательно!

Далее, если нужно через style обратиться к свойству, состоящему из составного имени, например,

font-size

то следует использовать верблюжью форму записи:

elem.style.fontSize = "24px";

Если в имени свойства имеется префикс:

-moz-border-radius, -webkit-border-radius

то оно начинается с заглавной буквы:

elem.style.MozBorderRadius = "5px";

Наконец, если нам нужно «сбросить» какое-либо свойство через style, то для этого достаточно присвоить ему пустую строку. Например:

let elem2 = elem.nextSibling.nextSibling;

let idMove = setInterval(**function**() {

**if**(!elem2.style.display) elem2.style.display = "none";

**else** elem2.style.display = "";

}, 100);

setTimeout(()=>clearInterval(idMove), 3000);

Мы здесь проверяем: если свойство elem2.style.display содержит пустую строку, то присваиваем значение «none», чтобы скрыть элемент. Иначе, «сбрасываем» это свойство, присваивая пустую строку. В результате элемент div начинает мигать с интервалом в 100 мс.

Во всех наших примерах со style мы работали с отдельными свойствами: top, display и т.п. Но что если мы хотим прописать сразу несколько свойств в виде строки:

`color:red; font-size: 18px; background-color: green`

Для этого следует все это присваивать специальному свойству cssText следующим образом:

elem.style.cssText = `color:red; font-size: 18px; background-color: green`;

Теперь к элементу применяются все эти стили. Однако, на практике такой подход используется крайне редко, так как подобные стили следует прописывать в CSS и подключать через класс. Это более гибкий и удобный инструмент, чем свойство cssText.

Наконец, следует знать, что свойства в style не синхронизированы со свойствами, указанными в CSS и подключенные к данному элементу. Например, наш элемент:

let elem = document.querySelector("div.msg");

elem.className = "message";

elem.classList.add("ex");

Имеет свойства, прописанные в классе message и его расширении ex. Но, при попытке их вывода через style:

console.log(elem.style.width);

увидим пустую строку. Но, что если мы хотим увеличить ширину на 20 пикселей? Как нам получить текущие свойства элемента? Для этого существует специальный метод:

let objStyles = getComputedStyle(element, [pseudo]);

который возвращает объект objStyles со всеми текущими (актуальными) стилями для элемента, указанным в аргументе element. Если же нам нужен стиль не самого элемента, а его псевдоэлемента, например, :hover, то этот псевдокласс и указывается в аргументе pseudo. Если он не нужен, то не пишем его. Например:

let objStyles = getComputedStyle(elem);

console.log(objStyles.width);

Теперь мы видим корректное значение 200px. Здесь стоит обращать внимание на то, что метод getComputedStyle корректно возвращает точные свойства, вроде paddingTop/paddingBottom/paddingLeft вместо общего свойства padding. Дело в том, что некоторые браузеры некорректно или совсем не формируют общие свойства вроде margin, padding и им подобные, поэтому всегда обращайтесь к соответствующим конкретным значениям!

Также есть одно ограничение на получение стилей псевдокласса :visited у посещенных ссылок. Доступ к нему закрыт на уровне JavaScript-машины, чтобы скрипт не мог знать: была ли посещена та или иная ссылка.

Итак, на этом занятии мы с вами рассмотрели способы управления оформлением элементов HTML-страницы через CSS-классы и свойство style. Теперь у вас есть базовая информация как все это работает.

# Метрики HTML-документа

[Смотреть материал на видео](https://www.youtube.com/watch?v=TVbZI3dRu6Q&list=PLA0M1Bcd0w8zri9wWiDFmOukXhguEYBBO" \t "_blank)

<https://proproprogs.ru/javascript_dom/metriki-html-dokumenta>

На этом занятии мы рассмотрим метрики, используемые для позиционирования элементов и их содержимого на экране браузера. Чтобы лучше их понять, предположим, что у нас есть вот такой документ:

<!DOCTYPE html>

<html>

<head>

<title>Уроки по JavaScript</title>

<style>

body {

margin: 0;

}

.message {

border: 20px solid #007700;

width: 300px;

height: 200px;

overflow: auto;

padding: 10px;

background: #eee;

font-size: 24px;

margin: 50px 0 0 100px;

}

</style>

</head>

<body>

<div class="message">

<h3>Уроки по JavaScript</h3>

<p>На этом занятии мы рассмотрим метрики, используемые для

позиционирования элементов и их содержимого

на экране браузера. Чтобы лучше их понять, рассмотрим такой рисунок.</p>

</div>

<script>

</script>

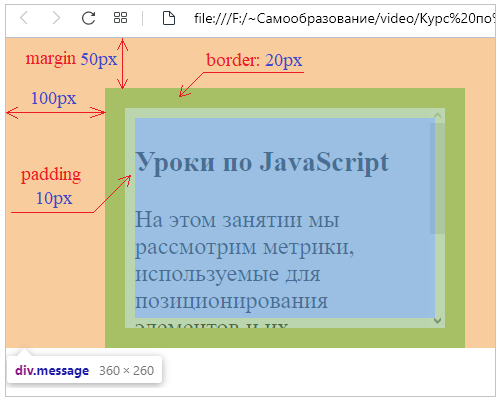
</body>

</html>

Который на экране браузера выглядит так. Обратите внимание, что в документе обязательно должен быть указан

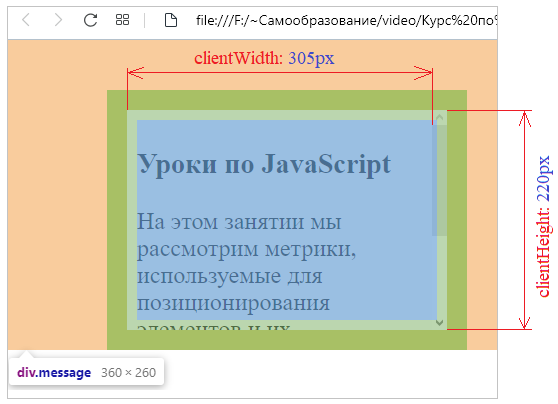
<!DOCTYPE html>

который говорит браузеру, что HTML-страницу следует воспринимать по стандарту HTML 5.x. От этого могут зависеть значения метрик, о которых здесь пойдет речь.



Теперь смотрите, у нашего блока div реальные размеры составляют 360 на 260 пикселей и имеется множество различных отступов: margin, border и padding. Так вот, ко всем этим параметрам блока div можно получить доступ из JavaScript. Они и называются метриками. Рассмотрим их по порядку. Начнем со свойств:

* elem.clientWidth – содержит ширину клиентской области (в пикселах);
* elem.clientHeight – содержит высоту клиентской области (в пикселах).



Выведем их в консоль:

let elem = document.querySelector("div.message");

console.log(elem.clientWidth, elem.clientHeight);

Увидим значения: 305 и 220 пикселей. Откуда взялись такие числа? Изначально мы задавали ширину, равную 300 пикселей. К этому значению прибавляются отступы padding по 10 пикселей и вычитается ширина полосы прокрутки, которая в данном случае съедает 15 пикселей, получаем:

clientWidth = 300+10+10-15 = 305

Аналогично вычисляется размер клиентской области по вертикали: в ней учитывается базовая высота блока div в 200 пикселей и по 10 пикселей отступы padding, имеем:

clientHeight = 200+10+10 = 220

Обратите внимание, что эти свойства доступны только для чтения, изменять их, например, вот так:

elem.clientHeight = 100;

не получится. Если нам нужно изменить высоту элемента, то следует использовать соответствующие свойства из style:

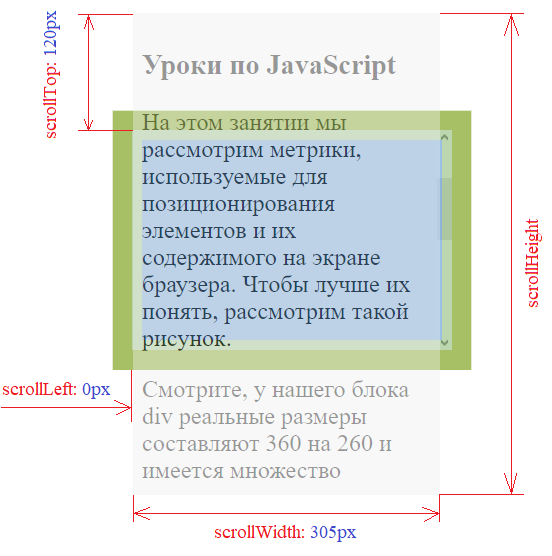
elem.style.height = 100+"px";

с обязательным указанием единиц измерения (в данном случае – пиксели). Аналогично через свойство width меняется ширина блока.

Следующие метрики:

* elem.scrollTop – высота прокрутки содержимого блока;
* elem.scrollLeft – величина прокрутки содержимого в левую сторону блока;
* elem.scrollWidth – ширина содержимого с учетом прокрутки;
* elem.scrollHeight – высота содержимого с учетом прокрутки.

Все эти свойства для наглядности изображены на этом рисунке.



Чтобы увидеть значения изменения этих величин, преобразуем немного наш документ, поместим внутрь нашего блока еще один блок

<div style="width: 500px">

и будем отображать величины каждую секунду:

let elem = document.querySelector("div.message");

setInterval(**function**() {

     console.log(elem.scrollTop, elem.scrollLeft, elem.scrollWidth, elem.scrollHeight);

}, 1000);

Как видите, при прокрутке содержимого, у нас меняются значения scrollLeft и scrollTop. Остальные два параметра остаются без изменений, как это и должно быть.

Величины scrollLeft и scrollTop можно менять и осуществлять программный скроллинг содержимого. Например, если мы пропишем вот такой код:

elem.scrollTop = 100;

то при обновлении документа увидим смещение содержимого по вертикали. Аналогично и для

elem.scrollLeft += 20;

мы его здесь ему присвоили значение через оператор +=, то есть, к предыдущему значению добавили еще 20 пикселей. Так тоже можно делать.

А вот свойства scrollWidth и scrollHeight менять уже нельзя, они доступны только для чтения.

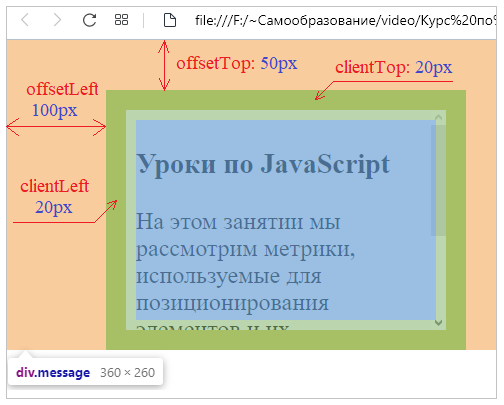
Следующие свойства:

* elem.offsetLeft – содержит смещение блока от левого края;
* elem.offsetTop – содержит смещение блока от верхнего края.

Данные свойства обычно отсчитывают смещения относительно окна браузера (элемента body), но в общем случае относительно своего предка, указанного в свойстве offsetParent:

console.log(elem.offsetParent);

В данном случае увидим элемент body. Визуально их можно представить так:



Выведем их в консоль

console.log(elem.offsetLeft, elem.offsetTop);

и убедимся, что они принимают заданные в CSS-стилях значения. На этом же рисунке отмечены еще два свойства:

* elem.clientLeft – смещение слева внутренней части области от края элемента (в данном случае блока div);
* elem.clientTop – смещение сверху внутренней части области от края элемента (в данном случае блока div).

В нашем примере эти свойства будут равны 20 пикселам – толщине рамки, указанной в CSS-свойстве border. Выведем их в консоль:

console.log(elem.clientLeft, elem.clientTop);

и видим значения 20.

Наконец, последние две метрики:

* elem.offsetWidth – ширина элемента с учетом всех отступов;
* elem.offsetHeight – высота элемента с учетом всех отступов.

То есть, в нашем случае – это будут полные размеры блока div. Выведем их в консоль:

console.log(elem.offsetWidth, elem.offsetHeight);

и видим значения 360 на 260 пикселей.

Вот мы с вами рассмотрели метрики DOM-элементов и теперь вы знаете как их использовать. Однако, может возникнуть вопрос: можно ли брать эти же значения непосредственно из стилей объекта, которые можно получить с помощью функции:

let styles = getComputedStyle(elem, [pseudo]);

о которой мы говорили на предыдущем занятии? В действительности нет, так как реальные значения элементов в окне браузера могут отличаться от значений свойств, указанных в CSS. Кроме того, в CSS свойства могут быть вообще не указаны, например, высота элемента – она часто определяется реальным содержимым. Или же, ширина заданная как auto:

width: auto;

не может быть переведена в пиксели функцией getComputedStyle: мы там так auto и увидим. Поэтому, если вам нужны актуальные значения метрик элемента, то следует использовать свойства, рассмотренные на этом занятии.

# Размеры и скорллинг HTML-документа

[Смотреть материал на видео](https://www.youtube.com/watch?v=fkwgAOI-01g&list=PLA0M1Bcd0w8zri9wWiDFmOukXhguEYBBO" \t "_blank)

<https://proproprogs.ru/javascript_dom/razmery-i-skorlling-html-dokumenta>

Продолжим рассматривать метрики и начнем с вопроса определения ширины и высоты всего клиентского окна браузера. Для этого можно обратиться к элементу-тегу html и посмотреть его значения свойств:

* clientWidth – ширина клиентской области (за вычетом полосы вертикального скроллинга);
* clientHeight – высота клиентской области (за вычетом полосы горизонтального скроллинга).

Например, так:

let html = document.documentElement;

console.log(html.clientWidth, html.clientHeight);

Другой способ – это обратиться к свойствам innerWidth/innerHeight объекта window:

* window.innerWidth – полная ширина окна браузера вместе с полосой прокрутки (если она есть);
* window.innerHeight – полная высота окна браузера вместе с горизонтальной полосой прокрутки (если она есть).

Например, добавим в документ вот такой список:

<ul style="font-size: 20px">

<li>offsetParent<li>offsetLeft

<li>offsetTop<li>offsetWidth

<li>offsetHeight<li>clientTop

<li>clientLeft<li>clientWidth

<li>clientHeight<li>scrollWidth

<li>scrollHeight<li>scrollLeft

<li>scrollTop<li>innerHeight

<li>innerWidth

</ul>

так, чтобы у нас появилась вертикальная полоса прокрутки. И дополнительно выведем в консоль свойства:

console.log(window.innerWidth, window.innerHeight);

Как видите, ширина окна в innerWidth не учитывает полосу прокрутки, а свойство clientWidth – учитывает. Вот этот момент следует иметь в виду при разработке скриптов.

## Высота и ширина HTML-документа

Теперь предположим, что мы хотим определить всю ширину и высоту документа, загруженного в браузер. Формально, для этого следует обратиться опять же к тегу-элементу html и просмотреть его свойства scrollWidth/scrollHeight:

let html = document.documentElement;

console.log(html.scrollWidth, html.scrollHeight);

Как видите мы получаем вполне корректные значения при наличии прокрутке и при ее отсутствии. Но, к сожалению, не все браузеры адекватно определяют эти свойства при отсутствии полос прокрутки. И на практике программисты поступают так. Для надежного вычисления высоты всего документа, берут максимальное значение из следующих свойств:

let scrollHeight = Math.max(

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

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

      document.body.clientHeight, document.documentElement.clientHeight

);

console.log(scrollHeight);

Почему это так работает? Думаю, этого никто не знает. И эта проблема наблюдается только для всего документа. Для отдельного элемента все работает через свойства scrollWidth/scrollHeight. А для всей страницы приходится использовать такой своеобразный «костыль» (как говорят программисты). Просто запомните этот момент: для корректного определения высоты (или ширины) всего документа в браузере, нужно брать максимальное значение среди этих свойств (для ширины будут соответствующие свойства, где вместо Height следует записать Width).

## Получение текущего положения прокрутки

Далее, чтобы определить положение прокрутки всего документа, можно опять же воспользоваться объектом html и посмотреть значения его свойств scrollLeft/scrollTop, о которых мы говорили на предыдущем занятии. Например, так:

let html = document.documentElement;

console.log(body.scrollTop);

И, в общем то этим можно пользоваться без каких-либо проблем. Но в большинстве старых браузеров это свойство следовало получать при обращении к элементу body так:

console.log(document.body.scrollTop);

А в новых – это, наоборот, не работает. Но мы можем сделать все очень надежно, воспользовавшись свойствами:

* window.pageXOffset – смещение документа в окне браузера по оси Ox;
* window.pageYOffset – смещение документа в окне браузера по оси Oy.

Например:

console.log(window.pageYOffset);

получаем корректные значения положения скроллинга документа по вертикали. Здесь следует только помнить, что эти свойства доступны только для чтения, так что управлять скроллингом через них не получится.

## Прокрутка (скроллинг) документа

Теперь посмотрим, как же все-таки осуществлять прокрутку всего документа. Во-первых, следует помнить, что прокрутка документа работает только после загрузки всей HTML-страницы, когда построено DOM-дерево. До этого рассматриваемые ниже методы скроллинга работать не будут. Так что, вот это имейте в виду.

Далее, в современных браузерах прокручивать документ можно через свойства объекта html:

* document.documentElement.scrollTop
* document.documentElement.scrollLeft

В некоторых старых браузерах этот вариант будет работать через body:

* document.body.scrollTop
* document.body.scrollLeft

Поэтому, для написания надежного кода, лучше использовать специальные методы скроллинга объекта window:

* window.scrollBy(offX, offY) – прокручивает страницу относительно её текущего положения на смещения offX, offY пикселей;
* window.scrollTo(pageX, pageY) – прокручивает страницу до указанных координат pageX и pageY.

Например, вот такой скрипт:

setInterval(**function**() {

    window.scrollBy(0, 5);

}, 100);

будет прокручивать страницу вниз на 5 пикселей каждые 100 мс. А если прописать вот так:

setInterval(**function**() {

    window.scrollTo(0, 5);

}, 100);

то мы будем все время переходить в одну и ту же позицию на 5 пикселей по вертикальному скроллингу (вручную переводим вниз, он возвращается). И, например, строка:

window.scrollTo(0, 0);

будет возвращать нас просто к началу документа.

Для полноты картины давайте рассмотрим ещё один метод scrollIntoView, который имеет следующий синтаксис:

elem.scrollIntoView(top = true);

Данный метод существует у всех объектов-тегов DOM-дерева и прокручивает документ так, чтобы elem оказался вверху окна браузера (если значение top=true), или внизу, при значении top=false. Пусть, в начале документа имеется заголовок:

<h1 id="header\_1">Список свойств метрики</h1>

и мы хотим прокрутить документ, чтобы он стал виден вверху страницы:

setTimeout(**function**() {

    header\_1.scrollIntoView();

});

Обратите внимание, что мы вызвали этот метод через планировщик отложенного вызова – функцию setTimeout с нулевой задержкой. Если мы просто в скрипте напишем:

header\_1.scrollIntoView();

то это работать не будет. Как мы отмечали в самом начале, все методы скроллинга работают после полной загрузки документа и построения DOM-дерева. И, вызывая scrollIntoView через setTimeout, мы как раз и выполняем это условие.

## Запрет прокрутки документа

Наконец, бывают ситуации, когда нам нужно запретить прокручивать документ в браузере пользователю. В частности, эта необходимость возникает при отображении какого-либо блока поверх документа и мы не хотим, чтобы при этом работал общий скроллинг. Например, как в этой игре: при отображении блока div, имитирующего диалоговое окно, у всего документа отсутствуют полосы прокрутки.

Это делается очень простой конструкцией:

document.body.style.overflow = "hidden";

И теперь, при обновлении документа у нас пропадает вертикальная полоса прокрутки. Для восстановления полос, свойство overflow нужно сбросить, присвоив ей пустую строку:

document.body.style.overflow = "";

Итак, на этом занятии мы с вами рассмотрели способы определения размеров клиентского окна браузера, HTML-документа в целом и способы его прокрутки.

# Расположение элементов в HTML-документе

[Смотреть материал на видео](https://www.youtube.com/watch?v=bahITnOqXnc&list=PLA0M1Bcd0w8zri9wWiDFmOukXhguEYBBO" \t "_blank)

<https://proproprogs.ru/javascript_dom/raspolozhenie-elementov-v-html-dokumente>

При разметке HTML-страницы можно использовать два принципиально разных способа позиционирования элемента поверх остальных элементов:

* position: fixed – позиционирование относительно окна браузера, не зависящее от прокрутки документа;
* position: absolute – позиционирование относительно документа (зависит от величины прокрутки).

Чтобы были понятнее эти различия, приведу такой пример. Пусть у нас имеется вот такой документ:

<!DOCTYPE html>

<html>

<head>

<title>Уроки по JavaScript</title>

<style>

#wnd {z-index: 999; position: fixed; width: 200px;

overflow: hidden;background: #CC0000;color: #eee;

left: 100px;top: 50px;text-align: center;

}

</style>

</head>

<body>

<h1>Позиционирование элементов на странице</h1>

<div id="wnd"><p>Элемент с позиционированием fixed</div>

<ul style="font-size: 20px">

<li>offsetParent<li>offsetLeft<li>offsetTop

<li>offsetWidth<li>offsetHeight<li>clientTop

<li>clientLeft<li>clientWidth<li>clientHeight

<li>scrollWidth<li>scrollHeight<li>scrollLeft

<li>scrollTop<li>innerHeight<li>innerWidth

</ul>

<script>

</script>

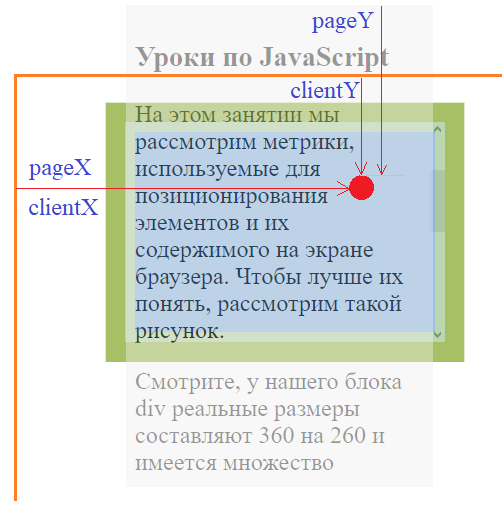
</body>

</html>

Если мы его отобразим в окне браузера, то увидим красное окно (элемент div) поверх всей страницы с положением, не зависящем от скроллинга. Но, если заменить стиль на:

position: absolute

то при прокрутке элемент начинает также смещаться вместе со всей страницей. Фактически, мы здесь имеем следующую картину (см. рисунок). В режиме fixed координаты left и top верхнего левого угла элемента div отсчитываются от границ клиентской области окна браузера (обозначим эти параметры через clientX, clientY). В режиме absolute отсчет идет от начала документа – параметры pageX, pageY.



Теперь поставим перед собой такую задачу: нужно с помощью скрипта расположить наш блок div точно по центру окна браузера. Если мы запишем такую программу:

let wnd = document.getElementById("wnd");

let centerX = document.documentElement.clientWidth/2;

let centerY = document.documentElement.clientHeight/2;

wnd.style.left = centerX + "px";

wnd.style.top = centerY + "px";

То по центру будет верхний левый угол, а не сам блок. Здесь из значений centerX, centerY нужно еще вычесть половину ширины и высоты блока div. Эти величины возьмем из свойств offsetWidth/offsetHeight нашего элемента, получим:

wnd.style.left = centerX-wnd.offsetWidth/2 + "px";

wnd.style.top = centerY-wnd.offsetHeight/2 + "px";

Все, теперь блок встал точно по центру окна. Так вот, если мы теперь захотим узнать все координаты и размеры блока div, то всю эту информацию можно получить с помощью метода

let coords = elem.getBoundingClientRect();

Например, добавим строчки:

let coords = wnd.getBoundingClientRect();

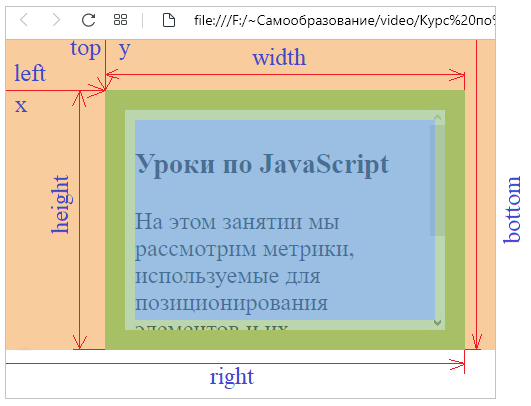
console.log(coords);

И обновим документ. В консоле увидим параметры:

x/y, top/left, width/height, bottom/right

Все они представлены на рисунке ниже. Причем,

right = left+width  
bottom = top+height



Использовать эти методы бывает очень удобно для получения актуальных значений координат и размеров какого-либо элемента-тега в HTML-документе.

Здесь вы можете заметить два дублирующих свойства: top = y и left = x. Зачем нужны x и y? Дело в том, что при отрицательных значениях width или height значения x и y будут соответствовать нижнему правому углу, а top и left продолжат указывать на верхний левый. Правда такие моменты редко используются на практике, да и некоторые браузеры не возвращают величины x/y (например, IE и Edge).

Также следует иметь в виду, что параметры right и bottom не имеют ничего общего с аналогичными параметрами в CSS. Они только называются одинаково, но их значения совершенно разные.

Если мы прокрутим страницу так, чтобы элемент ушел из области видимости, то получим отрицательные значения координат по высоте. Что, в общем-то логично, так как координаты отсчитываются от верхней и левой границ клиентского окна браузера.

## elementFromPoint(x, y)

Далее, рассмотрим метод документа

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

который возвращает самый глубокий в иерархии элемент, на который указывает координата (x,y). Это бывает полезно, когда мы хотим узнать, что находится в данной позиции HTML-страницы. Для примера возьмем прежний документ и удалим ранее написанный скрипт. У нас блок div окажется поверх пунктов списка. Теперь возьмем координату (101, 110) так, чтобы мы указывали и на блок div и на список. Вызовем метод elementFromPoint с этими значениями и посмотрим, что он нам вернет:

let wnd = document.getElementById("wnd");

let elem = document.elementFromPoint(102, 110);

console.log(elem);

Получили элемент div. Но что тогда означает фраза «самый глубокий в иерархии элемент»? Разве мы не должны были бы получить элемент списка, так как он находится глубже на странице? Не совсем так: список и div являются сестринскими элементами, то есть, у них один родитель. В этом случае возвращается тот, что находится выше. А вот если мы поместим координату над списком:

let elem = document.elementFromPoint(102, 120);

то как раз и получим самый глубокий элемент, то есть, тег <li>, а не тег <ul>, который является родительским по отношению к li. Вот так следует понимать работу этого метода.

Этот метод возвращает элементы находящиеся в области видимости окна браузера, то есть, когда координаты x/y находятся в клиентской области. В других случаях, например, при отрицательных значениях координат, мы получим значение null.

## Вычисление координат относительно документа

Часто в практике программирования требуется получить координаты какого-либо элемента относительно HTML-документа, а не окна браузера. Как это можно сделать? Ранее рассмотренный метод

elem.getBoundingClientRect()

возвращает координаты относительно окна (clientX, clientY). Значит, чтобы получить координаты относительно документа к ним нужно прибавить значения прокрутки документа по вертикали и горизонтали:

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

Вспоминая предыдущее занятие, величины прокрутки можно взять из свойств объекта window:

window.pageYOffset и window.pageXOffset

В результате, можно реализовать следующую функцию для вычисления координат элемента относительно документа:

let wnd = document.getElementById("wnd");

console.log( getCoordFromDocument(wnd) );

**function** getCoordFromDocument(elem) {

let coords = elem.getBoundingClientRect();

**return** {

  top: coords.top + window.pageYOffset,

  left: coords.left + window.pageXOffset

};

}

Если мы теперь будем прокручивать страницу, то значения координат будут оставаться неизменными, т.к. их значение отсчитывается от начала документа.

Итак, мы с вами рассмотрели способы позиционирования элементов в окне браузера и научились определять и задавать их координаты положения в HTML-документе.

# Обработчики событий

[Смотреть материал на видео](https://www.youtube.com/watch?v=wkzX9RtQKg4&list=PLA0M1Bcd0w8zri9wWiDFmOukXhguEYBBO" \t "_blank)

<https://proproprogs.ru/javascript_dom/obrabotchiki-sobytiy>

Начиная с этого занятия, мы подробно рассмотрим такую тему как события. Что такое событие? Это своего рода «сигнал» от браузера что, что-то произошло. Например:

* пользователь кликает мышкой по HTML-странице (браузер генерирует событие onclick);
* пользователь перемещает курсор мыши (браузер генерирует событие mousemove);
* пользователь нажимает на клавишу клавиатуры (браузер генерирует события keydown/keyup);
* когда полностью загружается HTML-документ браузер генерирует событие DOMContentLoaded).

и так далее. Существует много разных событий с возможностью реакции на них из программы на JavaScript. То есть, когда от браузера приходит определенное событие, мы можем вызвать привязанную к этому событию функцию. Такая функция называется **обработчиком события**.

В самом простом случае это можно сделать так. Предположим, у нас есть ссылка:

<!DOCTYPE html>

<html>

<head>

<title>Уроки по JavaScript</title>

</head>

<body>

<p><a id="link" href="#" onclick="showMessage()">Нажми меня</a>

<script>

</script>

</body>

</html>

и в ней указан атрибут onclick. Через этот атрибут происходит «привязка» функции showMessage к событию onclick. То есть, при щелчке мышью по ссылке будет вызвана функция showMessage. Сама же функция должна быть определена в скрипте, например, так:

**function** showMessage() {

    alert("Hello!");

}

Теперь, при обновлении документа и щелчке мышью по ссылке на экране браузера будет отображаться окно с сообщением «Hello!».

А можно ли программно применить обработчик для события onclick? Да, можно. Давайте для этого создадим еще один элемент:

<p><input type="button" value="Событие onclick" />

и в скрипте для тега <input> и свойству onclick присвоим ссылку на функцию-обработчик:

let inp = document.querySelector("input");

inp.onclick = showMessage;

Обратите внимание, мы здесь передаем ссылку на функцию, поэтому записываем ее без круглых скобок в конце. Если прописать вот так:

inp.onclick = showMessage();

то свойству onclick будет присвоен результат вызова функции, то есть, значение undefined и обработки события уже не будет.

Вот эти два способа (через атрибут onclick и свойство onclick) для назначения обработчиков, в действительности, работают одинаково: в обоих случаях при событии onclick соответствующего элемента будет вызвана функция showMessage. Но есть и отличия. При записи через атрибут происходит фактически следующее:

a.onclick = **function**(event) {

    showMessage();

}

то есть, свойству onclick присваивается ссылка на анонимную функцию с аргументом event (о нем мы поговорим позже) и в ней уже вызывается указанная функция showMessage. Именно поэтому в атрибуте onclick мы прописываем функцию showMessage() с круглыми скобками.

На практике используют оба способа для назначения обработчика какого-либо события. И в обоих случаях обработчик будет храниться в соответствующем свойстве (например, onclick, как в нашем скрипте).

Если необходимо сбросить обработчик и не реагировать, например, на событие onclick, то ему присваивается значение null:

inp.onclick = showMessage;

inp.onclick = **null**;

Теперь при нажатии на кнопку ничего не будет происходить.

В теле каждой функции-обработчика можно использовать ключевое слово this, которое ссылается на элемент DOM-дерева, на котором висит этот обработчик. Например, перепишем нашу функцию так:

**function** showMessage() {

    alert(**this**.tagName);

}

Нажимая на кнопку, получаем значение input, а при нажатии на ссылку – значение undefined. Почему у ссылки undefined? Дело в том, что обработчик ссылки прописан через атрибут onclick и вызов осуществляется из анонимной функции. В результате контекст this теряется и становится undefined. Исправить это можно несколькими способами. Ну, во-первых, назначить обработчик через свойство onclick:

link.onclick = showMessage;

А, во-вторых, использовать аргумент event анонимной функции для обращения к контексту:

**function** showMessage(event) {

    alert(event.currentTarget.tagName);

}

Здесь свойство currentTarget объекта event содержит ссылку на узел DOM-дерева, из которого пришло событие onclick. Что еще содержит объект event? У него имеются такие свойства:

* event.type – тип события (например, для onclick будет содержать строку «click»);
* event.clientX / event.clientY – координаты мыши в момент возникновения события (относительно границ окна браузера).

И ряд других в зависимости от типа события, о которых мы будем говорить на следующих занятиях.

У рассмотренных двух способов назначения обработчиков событий имеется одно существенное ограничение: мы не можем назначить несколько разных функций для одного и того же события. Тем не менее, иногда, в этом возникает необходимость. И разработчики языка JavaScript учли этот момент. Для назначения некоторому элементу (element) нескольких обработчиков для события event следует использовать метод

element.addEventListener(event, handler[, options]);

здесь event – строка с названием события; handler – ссылка на функцию-обработчик; options – необязательный параметр, о котором мы поговорим на последующих занятиях.

Итак, зададим два обработчика для события onclick конпки:

inp.addEventListener("click", showMessage);

inp.addEventListener("click", **function**(event) {

    console.log(event.clientX, event.clientY);

});

Теперь, при нажатии на кнопку мы увидим окно с сообщение «input» и в консоли координаты мыши в момент нажатия на кнопку.

Обратите внимание, если у нас будет задан еще один обработчик через свойство onclick:

inp.onclick = **function**() {

     console.log("обработчик onclick");

};

то он будет третьим. То есть, метод addEventListener не использует свойство onclick для назначения обработчиков, он реализует другой механизм на основе паттерна проектирования «слушатели». Поэтому на практике используют или метод addEventListener или свойство onclick, но не оба вместе.

Для удаления обработчиков, созданных с помощью addEventListener, используется метод

element.removeEventListener(event, handler[, options]);

Все аргументы здесь аналогичны аргументам метода addEventListener. Например, удалим обработчик showMessage тега input:

inp.removeEventListener("click", showMessage);

Теперь, при обновлении документа, мы увидим только сообщения в консоли. А вот удалить обработчик анонимной функции не получится, так как у нас нет ссылки на нее, а значит, мы не можем указать нужный handler у метода removeEventListener. Вот это следует иметь в виду при использовании анонимных функций в качестве обработчиков событий.

Также некоторым событиям можно назначать обработчики только с помощью метода addEventListener. Например, событию «transitionend», которое возникает при завершении CSS-анимации. Предположим, у нас имеется вот такой стиль:

<style>

input {

transition: width 1s;

width: 200px;

}

.expand { width: 400px; }

</style>

И при нажатии на кнопку мы добавляем стиль expand:

<p><input type="button" onclick="this.classList.toggle('expand')" value="Событие onclick" />

Далее, создаем обработчик события:

inp.addEventListener("transitionend", showMessage);

Теперь, при нажатии на кнопку, она постепенно увеличится и по окончании анимации вызовется метод showMessage.

В заключение этого занятия я приведу пример назначения объекта в качестве обработчика события. Да, так тоже можно делать и выглядит это следующим образом:

inp.addEventListener("click", {

handleEvent(event) {

  console.log("Событие: " + event.type);

}

});

Смотрите, здесь внутри объекта задан специальный метод  handleEvent, который автоматически вызывается, если объект назначен в качестве обработчика события. То есть, это стандартизированный метод, который должен присутствовать в объекте для обработки события.

То же самое можно сделать и через класс, например, так:

class MyHandler {

handleEvent(event) {

  console.log("Событие: " + event.type);

}

}

let inp = document.querySelector("input");

inp.addEventListener("click", **new** MyHandler());

Вот так в JavaScript назначаются и удаляются обработчики событий.

# Погружение и всплытие событий

[Смотреть материал на видео](https://www.youtube.com/watch?v=p78ff9QzWzU&list=PLA0M1Bcd0w8zri9wWiDFmOukXhguEYBBO" \t "_blank)

<https://proproprogs.ru/javascript_dom/pogruzhenie-i-vsplytie-sobytiy>

На этом занятии вы узнаете как происходит распространение событий в браузерной среде. Для начала давайте представим, что у нас есть вот такой документ, состоящий из трех вложенных элементов:

<!DOCTYPE html>

<html>

<head>

<title>Уроки по JavaScript</title>

<style>

body {font-size: 20px;}

#first {background: #CC4444; width: 100px; padding: 10px}

#second {background: #00CC00; padding: 10px}

#third {background: #4444CC; padding: 0 10px 0 10px}

</style>

</head>

<body>

<div id="first">div

<p id="second">p

<span id="third">span</span>

</p>

</div>

<script>

</script>

</body>

</html>

И мы на блок div вешаем обработчик:

onclick="showTag(event)"

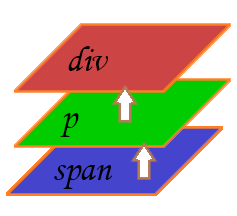
который выводит в консоль имя тега элемента:

**function** showTag(event) {

     console.log(event.currentTarget.tagName);

}

Если мы теперь кликнем мышкой по любому из блоков, то сработает этот обработчик и в консоли отобразится «DIV». Почему так происходит? Дело в том, что события в браузерной среде, срабатывая на любом дочернем элементе, переходят к его родителю, и там тоже возникает такое же событие. И так далее до самого корневого объекта.



Это называется всплытием события. И мы можем в этом убедиться, назначив обработчик каждому тегу:

<div id="first" onclick="showTag(event)">div

<p id="second" onclick="showTag(event)">p

<span id="third" onclick="showTag(event)">span</span>

</p>

</div>

Теперь, при обновлении документа, мы видим три тега в порядке возникновения события на каждом из них. Причем, это событие дойдет до самого корня DOM-дерева – до объекта document.

Так всплывают почти все события в JavaScript. Но есть исключения. Например, событие **focus** не всплывает. Но пока не заостряйте на этом свое внимание, просто имейте в виду, что всплытие – это не абсолютное правило, есть исключения. Если на каком-то уровне обработки нам необходимо узнать: от какого элемента пришло событие, то можно воспользоваться свойством

event.target

Например, перепишем обработчик с выводом этой информации в консоль:

**function** showTag(event) {

    console.log(event.currentTarget.tagName +

          " от " + event.target.tagName);

}

Видите отличие между свойствами currentTarget и target: первое указывает на текущий элемент, в котором обрабатывается событие, а второе – на элемент, в котором изначально произошло событие.

Иногда возникает необходимость прервать процесс всплытия и остановить обработку событий. Это реализуется с помощью метода:

event.stopPropagation();

Добавим его в наш обработчик и убедимся, что событие теперь останавливается на том элементе, где и возникло.

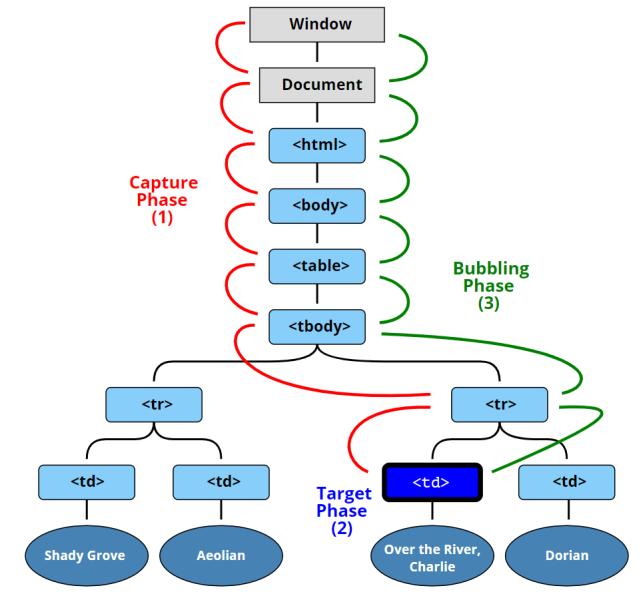
Однако, если у элемента имеется несколько обработчиков, срабатывающих на одно и то же событие (как мы говорили на предыдущем занятии, они добавляются с помощью метода **document.addEventListener**(…)), то для остановки их всех следует использовать метод:

event.stopImmediatePropagation();

Так как с помощью stopPropagation останавливается только текущий обработчик, в котором он и был вызван.

Небольшая рекомендация начинающим программистам: не вызывайте без строгой необходимости методы остановки процесса всплытия события, так как ряд элементов имеют автоматические обработчики стандартных событий. Например, при клике по ссылке осуществляется переход по указанному адресу. Если событие не дойдет, то и перехода не будет. Так что, прерывая этот процесс, вы должны четко понимать, что делаете!

На самом деле, события, прежде чем всплыть, сначала погружаются. Вот картинка, взятая из спецификации языка JavaScript, которая представляет жизненный цикл события.



Смотрите, в первой фазе (красная линия) событие опускается до целевого элемента, по которому кликнули мышью (это ячейка таблицы – тег td). Достигнув элемента, событие переходит во вторую фазу. После этого начинается третья фаза – фаза всплытия (зеленая линия), то есть переход по родительским элементам.

Так вот, в JavaScript можно отслеживать события не только на стадии всплытия, но и на стадии погружения. Для этого обработчики следует добавлять с помощью метода addEventListener, у которого третий параметр устанавливается в значение true:

elem.addEventListener(..., {capture: true});

или просто писать true:

elem.addEventListener(..., true);

Перепишем наш скрипт в таком виде:

first.addEventListener("click", showTag, **true**);

second.addEventListener("click", showTag, **true**);

third.addEventListener("click", showTag, **true**);

**function** showTag(event) {

    console.log(event.eventPhase + ": " +

              event.currentTarget.tagName +

             " от " + event.target.tagName);

}

Теперь, при клике по элементу span, мы увидим как событие идет вниз от блока div и обратно от span к div. То есть, если в методе addEventListener третий аргумент равен true, то обработчик перехватывает событие в стадии погружения. Если же этот аргумент равен false – то на стадии всплытия. Кроме того, свойство eventPhase показывает нам номер фазы: 1 – погружение; 2 – достижение целевого элемента; 3 – всплытие. Причем, на целевом элементе span срабатывают два обработчика: и на погружение и на всплытие.

Если мы теперь захотим удалить какой-либо обработчик с помощью метода removeEventListener, то должны также указывать и фазу, для которой этот обработчик применен. Например, если записать вот так:

first.removeEventListener("click", showTag);

то обработчик showTag удален не будет, т.к. здесь по умолчанию идет удаление на всплытие. В нашем случае следует прописать так:

first.removeEventListener("click", showTag, **true**);

Теперь, при клике по span мы не увидим div на стадии погружения – обработчик был успешно удален.

Вот так в браузерной среде происходит движение большинства событий.

# Делегирование событий

[Смотреть материал на видео](https://www.youtube.com/watch?v=KjacEXfpleU&list=PLA0M1Bcd0w8zri9wWiDFmOukXhguEYBBO" \t "_blank)

<https://proproprogs.ru/javascript_dom/delegirovanie-sobytiy>

На предыдущем занятии мы с вами познакомились с таким важным механизмом как погружение и всплытие событий для вложенных элементов. Так вот, используя механизм всплытия, в JavaScript реализуется такой прием разработки как **делегирование событий**. Что это такое? Предположим, у нас есть вот такая страница с неким меню:

<!DOCTYPE html>

<html>

<head>

<title>Уроки по JavaScript</title>

<style>

#main\_menu {

margin: 10px; padding: 0;

list-style: none; font-size: 24px;

width: 200px; background: #eee;

cursor: pointer;

}

#main\_menu li { margin-bottom: 10px; padding: 5px;}

#main\_menu li:hover {background: #FF4EF5;}

</style>

</head>

<body>

<ul id="main\_menu">

<li>Открыть</li><li>Сохранить</li>

<li>Рисовать</li><li>Редактировать</li>

<li>Настройки</li>

</ul>

<script>

</script>

</body>

</html>

И мы, естественно, хотим обрабатывать клики по каждому из этих пунктов. Конечно, можно повесить обработчик на каждый элемент li. Но можно сделать лучше: повесить один общий обработчик на элемент ul, который благодаря механизму всплытия будет перехватывать все клики от дочерних элементов li.

Добавим в скрипт следующие строчки:

main\_menu.addEventListener("click", **function**(event) {

      let li = event.target;

      console.log(li.innerHTML);

});

Теперь кликая по элементам li мы в консоли видим соответствующие строчки. Однако, если кликнуть между этими элементами, то сработает обработчик для ul и мы увидим все пункты меню. Это не то, что нам нужно. Поэтому немного перепишем обработчик:

      let li = event.target.closest('li');

**if**(li == **null**) **return**;

Здесь метод closest вернем ближайший дочерний элемент li, а если его нет, то значение null. Далее, мы проверяем: если li равно null, то завершаем работу обработчика. Обновим документ и видим, что теперь все работает гораздо лучше.

Но нам в нашей задаче нужно не получать содержимое элементов li, а выполнять определенные действия при выборе того или иного пункта меню. Как это лучше сделать? В практике программирования на JavaScript существует один очень эффективный и элегантный способ решить эту задачу. Для этого нам понадобятся нестандартные атрибуты, начинающиеся с префикса data-. Добавим к нашим тегам li следующие параметры:

<li data-command="open">Открыть</li><li data-command="save">Сохранить</li>

<li data-command="draw">Рисовать</li><li data-command="edit">Редактировать</li>

<li data-command="customize">Настройки</li>

А в обработчике будем читать этот атрибут:

console.log(li.dataset.command);

Теперь при клике по пункту меню будем получать команду, которую следует выполнить. Для удобства все эти команды можно прописать как методы некоторого объекта:

let menuActs = {

     open() {alert("open...");},

     save() {alert("save...");},

};

И вызывать их вот таким универсальным кодом:

    let act = li.dataset.command;

**if**(act && menuActs[act] !== **undefined**) menuActs[act]();

Щелкая по первым двум пунктам, откроется окно с сообщением. А при кликах по другим пунктам ничего не будет происходить, т.к. указанные методы не будут найдены в объекте menuActs.

Как видите, благодаря делегированию событий, получили довольно красивое и элегантное решение для обработки элементов пунктов меню, не привязанное к конкретной разметке. В дальнейшем можно относительно просто добавлять и менять это меню без существенной перестройки кода.

Кстати, используя прием с нестандартными атрибутами, в JavaScript удобно реализовывать обработчики общего назначения. Например, для отображения или сокрытия с экрана браузера некоторого элемента. Для примера, предположим, что у нас есть вот такая кнопка:

<p><input type="button" data-toggle-id="main\_menu" value="Показать/Скрыть меню" />

с атрибутом data-toggle-menu и значением main\_menu. Создадим универсальный обработчик, который будет использовать этот атрибут, чтобы показать или скрыть элемент с указанным id:

document.addEventListener("click", **function**(event) {

   let id = event.target.dataset.toggleId;

**if**(!id) **return**;

   let elem = document.getElementById(id);

**if**(!elem) **return**;

   elem.hidden = !elem.hidden;

});

Обратите внимание, что общий обработчик мы вешаем на объект document, который является корневым для DOM-дерева, то есть, все элементы внутри HTML-документа являются дочерними для этого объекта. Поэтому, когда событие всплывает до document, мы берем ссылку event.target на элемент, в котором событие произошло и смотрим значение его атрибута data-toggle-id. Если оно существует, то ищем элемент с указанным id, и если такой элемент находится, то меняем его видимость на противоположное значение. В результате наше меню то скрывается, то показывается.

Здесь может возникнуть вопрос: почему мы повесили универсальный обработчик через метод addEventListener, а не через свойство onclick объекта document? Дело в том, что на свойстве onclick уже может висеть какой-либо нужный обработчик и меняя его можно порушить работу сайта. Тогда как метод addEventListener лишь добавляет еще один обработчик к уже существующим и никаких дополнительных проблем это не должно вызывать. Особенно так удобно делать при разработке больших проектов, работая в команде: каждый программист может через addEventListener назначить свои обработчики и это никак не повлияет на работоспособность всего проекта.

Во второй части этого занятия рассмотрим события, которые браузер выполняет автоматически. Что это за события? Например, если мы вместо кнопки разместим вот такую ссылку:

<p><a href="ex1.htm" data-toggle-id="main\_menu">Показать/Скрыть меню</a>

то наш скрипт перестанет работать, так как браузер при клике будет выполнять переход к странице ex1.htm, то есть, перегружать текущий документ. Как сделать так, чтобы этого не происходило, то есть отменить действие браузера по умолчанию? Здесь есть два способа: наше событие onclick должно вернуть значение false:

onclick="return false;"

или же выполнить метод event.preventDefault() в обработчике события:

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

a.onclick = **function**(event) {

   event.preventDefault();

}

Когда может понадобиться такое прерывание действия браузера по умолчанию? Предположим, что мы создаем контекстное меню для всего документа:

document.addEventListener("contextmenu", **function**(event) {

      console.log("Контекстное меню документа");

});

Нажимая на правую кнопку мыши, мы увидим в консоли сообщение и контекстное меню по умолчанию от браузера. Но мы не хотим, чтобы стандартное меню отображалось. Для этого добавим в обработчик строку:

event.preventDefault();

Теперь щелкая правой кнопкой мыши, будем видеть только сообщение в консоли.

Далее, нам говорят, что нужно еще сделать отдельное контекстное меню для элементов главного меню. И мы добавляем в наш скрипт такие строчки:

main\_menu.oncontextmenu = **function**(event) {

   console.log("Контекстное меню главного меню");

   event.preventDefault();

}

Но при щелчке правой кнопкой мыши, мы теперь видим два контекстных меню, так как событие oncontextmenu всплыло до уровня document и там тоже сработал обработчик этого события. Естественным желанием будет остановить всплытие события:

event.stopPropagation();

и вроде бы все заработало как надо. Только теперь обработчики верхнего уровня не поймают это событие. А что, если среди них есть тот, что считает статистику вызова контекстных меню на странице? Лучшим решением будет в обработчике верхнего уровня отследить: было ли уже вызвано контекстное меню в обработчике нижнего уровня и если так, то ничего не делать. Это можно реализовать так:

* убрать строчку event.stopPropagation();
* и добавить первой строкой if (event.defaultPrevented) return; в document.addEventListener

Теперь наше событие продолжает всплывать, но контекстное меню документа будет отображаться, только если никакое другое ранее не было вызвано. Благодаря свойству

event.defaultPrevented

мы можем узнать было ли в дочерних элементах вызван метод

event.preventDefault();

и если да, то defaultPrevented принимает значение true. Иначе, оно будет равно false. Как видите, все достаточно просто.

Последнее, что отметим на этом занятии, в методе addEventListener можно использовать вот такой необязательный третий аргумент:

{passive: true}

который сигнализирует браузеру, что добавляемый обработчик события не собирается прерывать действие по умолчанию браузера. Например:

document.addEventListener("contextmenu", **function**(event) {

**if** (event.defaultPrevented) **return**;

      console.log("Контекстное меню документа");

      event.preventDefault();

}, {passive: **true**});

Здесь возникнет ошибка, так как мы вызываем event.preventDefault(), хотя указали, что не будем этого делать. Зачем вообще нужно заранее говорить JavaScript-машине, что мы не собираемся прерывать обработку события? Дело в том, что некоторые действия, вроде скроллинга документа, особенно на смартфонах, желательно выполнять быстро и без задержек. Чтобы увеличить время реакции мы можем сразу сказать, что можно делать скроллинг, не дожидаясь окончания обработки текущего события в нашем скрипте. В некоторых браузерах (Firefox, Chrome) это свойство {passive: true} установлено сразу по умолчанию и это следует учитывать при написании программ.

Итак, на этом занятии мы рассмотрели механизм делегирования событий и узнали как можно обрабатывать действия браузера по умолчанию.

# События мыши

[Смотреть материал на видео](https://www.youtube.com/watch?v=HtVPuyzb3c8&list=PLA0M1Bcd0w8zri9wWiDFmOukXhguEYBBO" \t "_blank)

<https://proproprogs.ru/javascript_dom/sobytiya-myshi>

Ранее в скриптах мы с вами часто использовали событие click – щелчок левой кнопкой мыши. В действительности – это составное событие, то есть, мы сначала нажимаем левую кнопку, затем отпускаем и только потом генерируется событие click. Причем в момент нажатия и отпускания кнопки возникают свои события:

mousedown/mouseup – нажатие и отпускание кнопки мыши.

Давайте в этом убедимся, возьмем вот такой HTML-документ:

<!DOCTYPE html>

<html>

<head>

<title>Уроки по JavaScript</title>

</head>

<body>

<script>

document.addEventListener("mouseup",

()=>console.log("mouseup"));

document.addEventListener("click",

()=>console.log("click"));

document.addEventListener("mousedown",

()=>console.log("mousedown"));

</script>

</body>

</html>

В нем прописаны три обработчика на события click, mousedown и mouseup. Теперь кликнем по документу и в консоли увидим эти три события в порядке:

mousedown → mouseup → click

И это стандартизированный порядок: данные события возникают именно в такой последовательности. Для двойного клика тоже есть свое событие dblclick. Добавим его:

document.addEventListener("dblclick",

            ()=>console.log("dblclick"));

Теперь при двойном щелчке левой кнопкой мыши, мы видим такие события:

mousedown → mouseup → click → mousedown → mouseup → click → dblclick

Эта последовательность также стандартизирована и события следуют друг за другом именно в таком порядке.

Для событий mousedown и mouseup в объекте event полезным является еще одно свойство:

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

Отобразим его:

document.addEventListener("mousedown",

      (event)=>console.log("mousedown: "+event.which));

document.addEventListener("mouseup",

      (event)=>console.log("mouseup: "+event.which));

Теперь мы знаем какая кнопка мыши была нажата при возникновении данных событий.

Также часто совместно с мышью используется клавиатура. Например, в графических редакторах клавиши Alt (или Opt для Mac), Ctrl, Shift и Cmd для Mac добавляют функционал манипулятору «мышь». Чтобы в событиях мыши проверить: была ли дополнительно нажата еще какая-либо функциональная клавиша, используются следующие свойства объекта event:

* event.shiftKey – true, если нажата клавиша Shift и false в противном случае;
* event.altKey – true при нажатом Alt (или Opt для Mac) и false иначе;
* event.ctrlKey – true при нажатом Ctrl и false иначе;
* event.metaKey – true при нажатом Cmd (для Mac) и false иначе.

Обрабатывать их достаточно просто, например так:

document.addEventListener("click",

      (event) => {

**if**(event.shiftKey && event.ctrlKey)

                console.log("click");

});

Мы здесь увидим сообщение «click» только если при клике были нажаты клавиши Shift и Ctrl.

Обратите внимание, когда мы пишем браузерные скрипты, то предполагаем, что они могут быть запущены на самых разных компьютерах с разными ОС, в том числе и Mac OS. Так вот, в ОС Mac пользователи вместо клавиши Ctrl нажимают на клавишу Cmd. И это следует учесть в скрипте, например, так:

**if**(event.shiftKey && (event.ctrlKey || event.metaKey))

           console.log("click");

Здесь мы проверяем не только Ctrl, но и свойство metaKey, отвечающее за нажатие на кнопку Cmd.

Также не нужно забывать и про пользователей мобильных устройств с отсутствующей клавиатурой. Так что они вряд ли смогут перемещать курсор мыши с нажатой клавишей Shift!

Кроме клавиш, объект event для событий от мыши, содержит свойства:

* clientX/clientY – координаты курсора мыши относительно окна браузера;
* pageX/pageY – координаты курсора мыши относительно HTML-документа.

Например, для события mousemove будем выводить координаты курсора мыши в консоль:

document.addEventListener("mousemove",

  (event)=>console.log("mousemove: "+event.clientX+", "+event.clientY));

Аналогично работают свойства pageX/pageY.

Следующие два события мыши

* mouseover – возникает при наведении курсора мыши на элемент HTML-документа;
* mouseout – возникает при покидании курсора мыши элемента HTML-документа.

Например, у нас есть вот такое содержимое страницы:

<div class="parent">DIV

<p class="child">P</p>

</div>

Со стилями:

<style>

.parent {background: #CC4444; padding: 10px 0 10px 10px;

display: inline-block;width: 100px;}

.child {background: #00CC00; padding: 10px;}

</style>

И добавим обработчик события mouseover:

document.addEventListener("mouseover", (event)=>{

   let prop = "target = "+event.target.tagName;

**if**(event.relatedTarget != **null**)

      prop+=", relatedTarget = "+event.relatedTarget.tagName;

      console.log("mouseover: "+prop);

});

Здесь у объекта event мы используем два новых свойства:

* target – ссылка на объект над которым находится курсор мыши;
* relatedTarget – ссылка на объект, с которого ушел курсор мыши.

Давайте посмотрим, как это все будет работать. Объясняем.

А вот для другого события mouseout эти свойства работают с точностью наоборот:

* relatedTarget – ссылка на объект над которым находится курсор мыши;
* target – ссылка на объект, с которого ушел курсор мыши.

Изменим в обработчике событие на mouseout и обновим документ. Видите, эти свойства работают именно так, что логично, так как событие mouseout возникает при покидании элемента, значит target будет тем элементом, с которого мы ушли, а relatedTarget – элементом где мы сейчас находимся.

События mouseover и mouseout всплывают от целевого элемента, где они произошли и доходят до корня DOM-дерева – объекта document. Например, если повесить событие mouseover на тег div:

let div = document.querySelector("div.parent");

div.addEventListener("mouseover", showInfo);

**function** showInfo(event) {

let prop = "target = "+event.target.tagName;

**if**(event.relatedTarget != **null**)

  prop+=", relatedTarget = "+event.relatedTarget.tagName;

let name = event.currentTarget.tagName;

console.log(name+" mouseover: "+prop);

}

То при наведении курсора на дочерний элемент p, увидим событие mouseover. Причем при переходе с дочернего элемента и обратно также генерируется это событие.

Если же мы хотим чтобы событие возникало один раз при наведении на него курсора мыши, то можно воспользоваться эквивалентными событиями:

mouseenter и mouseleave

которые похожи на mouseover/mouseout, но:

1. Не всплывают.
2. Событие mouseenter генерируется когда указатель оказывается над элементом и при этом не важно: потомок это элемента или сам элемент.
3. Событие mouseleave срабатывает при покидании элемента целиком (дочерние элементы здесь не учитываются).

Например, заменим событие mouseover на mouseenter, получим. Смотрите, теперь при переходе между дочерним элементом и div дополнительных событий не возникает. Аналогично работает и mouseleave.

Из-за того, что события mouseenter и mouseleave не всплывают, они не могут быть делегированы обработчику верхнего уровня, например, для объекта document:

document.addEventListener("mouseenter", showInfo);

так работать не будет. Но, если прописать событие

document.addEventListener("mouseover", showInfo);

то мы его успешно перехватываем. Вот эти ограничения следует учитывать при использовании событий mouseenter и mouseleave.

Итак, на этом занятии мы с вами рассмотрели работу следующих событий мыши:

|  |  |
| --- | --- |
| mousedown/mouseup | При нажатии и отпускании кнопки мыши |
| click | При щелчке левой кнопкой мыши |
| dblclick | При двойном щелчке левой кнопкой мыши |
| mousemove | При перемещении мыши |
| mouseover/mouseout | При наведении и покидании курсора мыши на элемент |
| mouseenter/mouseleave | При наведении и покидании курсора мыши на элемент (не всплывают и не учитывают дочерние элементы) |

Также мы с вами рассмотрели следующие свойства объекта event:

|  |  |
| --- | --- |
| event.which | Номер нажатой кнопки мыши |
| event.shiftKey/altKey/ctrlKey/metaKey | Флаги, указывающие на соответствующую нажатую кнопку клавиатуры |
| event.clientX/clientY | Координаты курсора мыши относительно окна браузера |
| event.pageX/pageY | Координаты курсора мыши относительно HTML-документа |
| event.target/ relatedTarget | Ссылки на объекты в зависимости от типов события: mouseover/mouseout |

# События клавиатуры keydown и keyup

[Смотреть материал на видео](https://www.youtube.com/watch?v=EK91UGx6O7c&list=PLA0M1Bcd0w8zri9wWiDFmOukXhguEYBBO" \t "_blank)

<https://proproprogs.ru/javascript_dom/sobytiya-klaviatury-keydown-i-keyup>

На предыдущем занятии мы с вами рассматривали события от мыши. Продолжим тему событий и рассмотрим события от клавиатуры:

* keydown – возникает при нажатии клавиши;
* keyup – возникает при отпускании клавиши.

Добавим эти обработчики в наш документ для поля ввода input:

<!DOCTYPE html>

<html>

<head>

<title>Уроки по JavaScript</title>

</head>

<body>

<input id="inp" onkeydown="keyHandler(event)" onkeyup="keyHandler(event)" />

<script>

**function** keyHandler(event) {

console.log(event.type);

}

</script>

</body>

</html>

Если мы в поле ввода нажмем на клавишу клавиатуры, то увидим сначала событие keydown, а затем, событие keyup. Или же, если нажать и держать клавишу, то при автоповторе будет возникать несколько событий keydown и при отпускании – одно событие keyup. Для событий, возникающих при автоповторе, свойство event.repeat равно true. Так что, если нам нужно узнать как было сгенерировано текущее событие, то достаточно прочитать это свойство:

console.log("repeat: "+event.repeat);

Теперь посмотрим, как можно определить какая была нажата клавиша и как получить символ, связанный с ней. Для этого в объекте event есть два таких свойства:

* event.key – символ нажатой клавиши;
* event.code – код нажатой клавиши.

В чем между ними разница? Давайте выведем их в консоль и посмотрим:

console.log("key: "+event.key+", code: "+event.code);

Теперь, нажимая на клавишу ‘a’ мы видим, что свойство key соответствует символу этой клавиши, а code – ее коду (строке KeyA). Причем, этот код остается неизменным и для заглавных символов и при русской раскладке клавиатуры. То есть, если нам нужно просто понять: какая клавиша была нажата, то следует использоваться свойство code, а если нужен символ, связанный с этой клавишей – то используется свойство key.

Вообще, возвращаемые коды клавиш формируются по такому правилу:

* 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.

В старых версиях языка JavaScript использовалось еще одно событие от клавиатуры – keypress. Однако его работа в разных браузерах сильно различается, поэтому в новых спецификациях данное событие помечено как устаревшее и вместо него следует пользоваться событиями keydown и keyup. Также устаревшими считаются следующие свойства объекта event при работе с клавиатурой:

event.keyCode, event.charCode, event.which

И ими теперь лучше не пользоваться.

Далее рассмотрим событие scroll, связанное с прокруткой всего документа или же с содержимым отдельного элемента. Добавим в наш HTML-документ вот такой список:

<ul style="font-size: 20px">

<li>offsetParent<li>offsetLeft<li>offsetTop

<li>offsetWidth<li>offsetHeight<li>clientTop

<li>clientLeft<li>clientWidth<li>clientHeight

<li>scrollWidth<li>scrollHeight<li>scrollLeft

<li>scrollTop<li>innerHeight<li>innerWidth

</ul>

И повесим обработчик события scroll на объект window:

<input id="inp" style="position: fixed" onkeydown="keyHandler(event)" onkeyup="keyHandler(event)" />

window.addEventListener("scroll", **function**(event) {

        inp.value = pageYOffset;

});

Теперь при прокрутке в нашем поле ввода будет отображаться смещение документа по вертикали.

Вообще событие scroll довольно часто используется в практике программирования, например, для автоматической подзагрузки данных, когда пользователь прокручивает страницу до конца.

Также обратите внимание, что для данного события нельзя отменить действие браузера по умолчанию, используя метод

event.preventDefault()

так как фактическая прокрутка отрабатывается до возникновения этого события в JavaScript. Если по каким-то причинам нужно запретить пользователю прокручивать документ, то это лучше сделать через CSS свойство overflow: hidden:

document.body.style.overflow = "hidden";

# События при загрузке HTML-документа

[Смотреть материал на видео](https://www.youtube.com/watch?v=KQKMRebk5uU&list=PLA0M1Bcd0w8zri9wWiDFmOukXhguEYBBO" \t "_blank)

<https://proproprogs.ru/javascript_dom/sobytiya-pri-zagruzke-html-dokumenta>

На этом занятии рассмотрим события, возникающие при загрузке HTML-документа и при его обновлении в браузере. Они такие:

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

В каких случаях используются эти события? Например, DOMContentLoaded полезно для инициализации каких-либо интерфейсных функций, прописанных в скрипте. Событие load может использоваться для динамической загрузки ресурсов (например, в браузерных играх следует подождать, пока все картинки загрузятся). При возникновении события beforeunload можно предупредить пользователя, что он не сохранил какие-либо изменения на своей странице (это, в частности, можно встретить в youtube). Ну а на событие unload можно повесить обработчик сбора статистики, чтобы определить время, которое пользователь провел на странице.

Рассмотрим работу этих событий подробнее и начнем с первого DOMContentLoaded. Возьмем вот такой HTML-документ с отображением рисунка:

<!DOCTYPE html>

<html>

<head>

<title>Уроки по JavaScript</title>

</head>

<body>

<p><img id="image" src="/htm/javascript\_dom/files/images/winter.jpg?nocashe=1" />

<script>

</script>

</body>

</html>

И добавим обработчик для DOMContentLoaded. Это можно сделать только через метод addEventListener объекта document:

document.addEventListener("DOMContentLoaded", ready);

**function** ready(event) {

    console.log("DOMContentLoaded");

    console.log(`Изображение: ${image.offsetWidth}x${image.offsetHeight}`);

}

В обработчике мы выводим размер изображения. Обновляем документ и видим, что размер нулевой. Это как раз и говорит о том, что событие DOMContentLoaded возникает до загрузки ресурсов (если, конечно, они не были закэшированы).

В целом, все понятно и просто. Но, как всегда, есть один нюанс. Данное событие отрабатывает только после выполнения всех скриптов, записанных в HTML-документе. Например, если мы добавим вот такой скрипт в конец документа:

<script>

console.log("вызов метода write");

document.write('Добавляем тег p на страницу');

</script>

То сначала сработает он, а потом уже событие DOMContentLoaded. Обновляем страницу и видим этот эффект.

Почему разработчики языка JavaScript реализовали такую последовательность вызовов? Дело в том, что в момент загрузки страницы скрипты могут вносить изменения в DOM-дерево: удалять или добавлять какие-то элементы. Поэтому, нужно сначала полностью сформировать DOM, а значит, выполнить скрипты, и только потом сгенерировать сообщение DOMContentLoaded. Так что такое поведение браузера вполне логично.

Из этого правила есть только два исключения:

* Скрипты с атрибутом async (который мы рассмотрим немного позже), не блокируют DOMContentLoaded.
* Скрипты, сгенерированные динамически при помощи document.createElement('script') и затем добавленные на страницу, также не блокируют это событие.

С каскадными таблицами стилей (CSS) все несколько иначе. Изначально, событие DOMContentLoaded возникает не дожидаясь загрузки стилей, так как они непосредственно не затрагивают DOM-дерево. Однако, если на странице есть скрипт, который использует CSS-стили, например, такой:

<script>

*// скрипт не выполняется, пока не загрузятся стили*

console.log(getComputedStyle(document.body).marginTop);

</script>

То браузер будет вынужден сгенерировать событие DOMContentLoaded после загрузки стилей. Вот эти нюансы следует иметь в виду при работе с этим событием.

Следующее событие load для объекта window работает очень просто: оно генерируется браузером, когда HTML-документ полностью загружен вместе со всеми связанными с ним ресурсами:

window.onload = **function**(event) {

   console.log("load");

   console.log(`Изображение: ${image.offsetWidth}x${image.offsetHeight}`);

}

Здесь при обновлении документа мы увидим полные размеры изображения.

Следующее событие unload объекта window выполняется когда пользователь практически покинул HTML-страницу. Обычно, в этот момент отправляется статистика поведения пользователя на странице серверу:

window.addEventListener("unload", **function**() {

   console.log("отправка данных на сервер");

});

Если мы теперь станем обновлять станицу, то в консоли будет появляться данное сообщение. К сожалению, продемонстрировать работу этого события как то приятнее не получается. Поэтому, далее, пара слов теории. Для отправки данных серверу в объекте navigator существует такой специальный метод:

navigator.sendBeacon(url, data);

Здесь url – это путь к скрипту на сервере, который будет принимать данные data. Особенность этого метода в том, что при закрытии страницы нет необходимости дожидаться окончания его работы. Браузер будет его выполнять в фоне даже при отсутствии ранее открытого документа. Поэтому, если вы хотите что-то отправить на сервер при закрытии HTML-страницы, то это лучше всего делать через этот метод. На практике он реализуется примерно так:

window.addEventListener("unload", **function**() {

navigator.sendBeacon("/analytics.php", JSON.stringify(myData));

};

И работает по такой схеме:

* данные отправляются по POST-запросу;
* размер данных ограничен 64 Кб.

Также в этом событии нельзя предупредить пользователя о каких-то несохраненных данных на странице и прервать закрытие документа. Если нужно выполнить такие действия, то для этого используется событие

**beforeunload**

объекта window следующим образом:

window.onbeforeunload = **function**() {

**return** **false**;

};

Теперь, при обновлении документа будет появляться окно, спрашивающее пользователя: действительно ли он хочет перезагрузить страницу. Это должно привлечь его внимание и напомнить о возможных последствиях такого действия, например, на несохранение каких-то данных.

Ранее, в браузерах можно было писать свои сообщения в таких окнах, примерно так:

window.onbeforeunload = **function**() {

**return** "Не уходи не попрощавшись!";

};

В этом случае также будет появляться окно с предупреждением, но к с частью строка сообщения в новых браузерах игнорируется и заменяется стандартным предупреждением, так как программисты прошлого злоупотребляли этой возможностью.

## readyState

В заключение занятия отметим свойство

document.readyState

которое в момент загрузки HTML-документа принимает следующие значения:

* "loading" – документ в процессе загрузки;
* "interactive" – документ был полностью прочитан (парсинг документа завершен);
* "complete" – документ был полностью прочитан и все ресурсы (изображения, стили и т.п.) тоже загружены.

В ряде случаев это свойство бывает весьма полезно. Например, мы вызываем функцию, но не уверены, что DOM-дерево полностью построено. Поэтому, делаем такую проверку:

removeImage();

**function** removeImage() {

**if**(document.readyState == "loading") {

  console.log("документ грузится, вешаем обработчик");

  document.addEventListener("DOMContentLoaded", removeImage);

}

**else** {

  console.log("удаляем изображение");

  document.body.remove(image);

}

}

По аналогии могут быть обработаны и остальные свойства.

Для полноты картины пару слов о событии readystatechange, которое появилось до событий

DOMContentLoaded, load, unload, beforeunload

и в старых версиях JavaScript процесс загрузки документа контролировался через него. Например, так:

document.addEventListener('readystatechange', **function**() {

         console.log(document.readyState);

});

Теперь при обновлении страницы мы можем увидеть изменение состояний свойства document.readyState в процессе загрузки. Однако такой механизм отслеживания ушел в прошлое и сейчас уже нет смысла о нем подробно говорить.

Итак, на этом занятии мы с вами рассмотрели события

DOMContentLoaded, load, unload, beforeunload

и поговорили о свойстве

document.readyState

которое дополняет работу с этими событиями.

# События load, error, атрибуты async, defer

[Смотреть материал на видео](https://www.youtube.com/watch?v=lOzpaYsmI9U&list=PLA0M1Bcd0w8zri9wWiDFmOukXhguEYBBO" \t "_blank)

<https://proproprogs.ru/javascript_dom/sobytiya-load-error-atributy-async-defer>

На предыдущем занятии мы узнали о событиях, которые происходят во время загрузки HTML-документа:

DOMContentLoaded, load, unload, beforeunload

Но, что если мы хотим отследить загрузку отдельных элементов документа, например, изображения или скрипта:

img, script

Оказывается для всех элементов, имеющих атрибут src, генерируется событие

load

при завершении загрузки содержимого элемента. Например, повесим обработчик такого события на загрузку изображения:

<!DOCTYPE html>

<html>

<head>

<title>Уроки по JavaScript</title>

</head>

<body>

<p><img id="image" src="/htm/javascript\_dom/files/images/winter.jpg?nocashe=1" />

<script>

image.onload = **function**(event) {

console.log(`Изображение: ${image.offsetWidth}x${image.offsetHeight}`);

}

</script>

</body>

</html>

Теперь после загрузки изображения мы увидим в консоли его размеры. То же самое можно сделать и со скриптом:

<script src="js/ex1.js" onload="execute()"></script>

А в скрипте выше прописать эту функцию:

**function** execute() {

     console.log("скрипт загружен");

}

При обновлении документа мы увидим сообщение, что скрипт загружен. Для чего нам может понадобиться обработчик события load для скриптов? Например, если нужно вызвать некоторую функцию, объявленную в ex1.js. Очевидно, это можно сделать только после его загрузки:

**function** execute() {

         \_\_fromJS();

}

Причем, во всех случаях обработчик execute() должен быть объявлен до строчки

<script src="js/ex1.js" onload="execute()"></script>

Или же, мы можем сделать так: создать скрипт динамически:

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

script.src = "js/ex1.js";

script.onload = execute;

document.head.append(script);

И тогда местоположение обработчика не будет иметь значения.

Но, как вы понимаете, при загрузке ресурсов могут происходить ошибки. Например, если указать неверный путь к файлу:

script.src = "js/ex11.js";

то при загрузке произойдет ошибка и событие load не сгенерируется для данного элемента. Для обработки таких ошибок можно использовать событие

error

следующим образом:

script.onerror = execute\_error;

и пропишем обработчик:

**function** execute\_error() {

     console.log("ошибка при загрузке ресурса");

}

Теперь, при обновлении документа, мы увидим строчку с сообщением об ошибке. В идеале здесь можно выполнить некоторые действия для минимизации вреда от такой ситуации.

Аналогичный обработчик можно повесить и для изображения:

image.onerror = () => console.log("error load image");

## async, defer

Во второй части занятия поговорим об атрибутах тега <script>. Они добавляют гибкости при подключении внешних скриптов к HTML-документу. И начнем с атрибута defer. Представьте, что у нас имеется внешний скрипт, который имеет большой объем и долго загружается с сервера:

<p>Текст до скрипта

<script src="http://funnyballs.ru/long\_script.js">

</script>

<p>Текст после скрипта

Тогда мы увидим сначала первый абзац и только после загрузки – второй абзац. То есть, наш HTML-документ будет подвисать и ждать окончания загрузки внешнего ресурса. Если у нас всего один такой внешний скрипт, то в самом простом случае его можно просто расположить в конце документа:

<p>Текст до скрипта

<script defer src="http://funnyballs.ru/long\_script.js">

</script>

<p>Текст после скрипта

Но это далеко не самое лучшее решение. В таких ситуациях можно воспользоваться атрибутом defer, который сообщает браузеру, чтобы он загружал скрипт в фоновом режиме, продолжая загружать основной HTML-документ. Как только скрипт загрузится он начнет выполняться.

Если добавить этот атрибут в наш тег:

<p>Текст до скрипта

<script defer src="http://funnyballs.ru/long\_script.js">

</script>

<p>Текст после скрипта

то он выполнится до появления какого-либо абзаца, так как браузер загрузит и выполнит его быстрее, чем отобразит теги p. Это как раз и есть эффект атрибута defer – браузер обрабатывал скрипт в фоне. Причем, скрипты с этим атрибутом выполнятся только после того, как браузер целиком загрузит всю страницу и построит DOM-дерево, но до возникновения события DOMContentLoaded. Мы это можем увидеть на следующем примере:

<script>

document.addEventListener('DOMContentLoaded', **function**() {

console.log("DOMContentLoaded");

});

</script>

Далее, если на нашей странице имеется несколько скриптов с атрибутом defer:

<script defer src=" http://funnyballs.ru/long\_script.js"></script>

<script defer src="js/ex2.js"></script>

то они будут выполняться в порядке записи в HTML-документе, то есть, сначала выполнится ex1.js, а затем, ex2.js вне зависимости от того, который из них будет загружен раньше. Например, если скрипт ex1.js окажется большим, а ex2.js – маленьким, то выполнение второго скрипта будет отложено до тех пор, пока не загрузится первый большой скрипт и не выполнится.

Иногда такое поведение не то, что нам бы хотелось. Допустим, что для нас было бы лучше, если бы скрипты запускались в порядке их загрузки. В этом случае следует воспользоваться атрибутом

async

который делает скрипт совершенно независимым от других скриптов, а также от события DOMContentLoaded: оно может наступить и раньше и позже, в зависимости от процесса загрузки скрипта.

Такие асинхронные скрипты очень полезны для добавления на страницу сторонних скриптов: счётчиков, рекламы и т.п. Они не зависят от других скриптов, и не вызывают задержек при загрузке и отображении всего HTML-документа.

Кстати, если в JavaScript-программе скрипт создается динамически, как в одном из наших примеров:

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

script.src = "js/ex1.js";

script.onload = execute;

document.head.append(script);

то он запускается в асинхронном режиме.

Вот так работают атрибуты

async, defer

и события

load, error

для отслеживания загрузки ресурсов отдельных элементов.

# Навигация и обработка элементов форм

[Смотреть материал на видео](https://www.youtube.com/watch?v=rujZthbjRXQ&list=PLA0M1Bcd0w8zri9wWiDFmOukXhguEYBBO" \t "_blank)

<https://proproprogs.ru/javascript_dom/navigaciya-i-obrabotka-elementov-form>

На этом занятии поговорим о способах навигации по элементам форм и доступу к их значениям. Для начала предположим, что у нас имеется вот такой документ с формой:

<!DOCTYPE html>

<html>

<head>

<title>Уроки по JavaScript</title>

</head>

<body>

<form name="reg">

<p><input name="user" value="" />

<p><input name="sex" type="radio" checked />М

<input name="sex" type="radio" />Ж

<p>Город: <select name="city">

<option>Москва</option>

<option>Санкт-Петербург</option>

<option>Тверь</option>

</select>

<p>Согласие <input name="agree" type="checkbox" />

<p><textarea name="about" rows="10" cols="30">О себе</textarea>

<p><input type="submit" name="submit">

</form>

<script>

</script>

</body>

</html>

И мы хотим получить доступ к форме в нашем скрипте. Для этого можно использовать специальную коллекцию

document.forms

которая содержит все формы текущего HTML-документа. В данном случае эта коллекция будет содержать один элемент (одну форму), к которой можно обратиться либо по индексу (отсчет начинается с индекса 0):

let frm = document.forms[0];

или по имени формы (указанное в атрибуте name):

let frm = document.forms.reg;

console.dir(frm);

Каждый элемент формы, в свою очередь, доступен через коллекцию

form.elements

например, так:

let user = frm.elements[0];

console.dir(user);

или по имени (по значению атрибута name):

let user = frm.elements.user;

Обратите внимание, что теги, не относящиеся к элементам формы, игнорируются. У нас это тег <p>.

В случае, если имеется несколько элементов с одинаковым именем (в нашем случае – это значение sex для выбора пола пользователя), то возвращается коллекция соответствующих элементов:

let sex = frm.elements.sex;

console.dir(sex);

И, далее, к каждому конкретному элементу можно обратиться по индексу:

console.dir(sex[0]);

В JavaScript допускается использовать более короткую запись для доступа к элементам формы согласно синтаксису:

form[index/name]

Например, так:

let user = frm[0];

let sex = frm.sex;

console.dir(sex);

Результат будет тем же. Также внутри формы можно использовать подформы, заданные тегом

<fieldset>[элементы подформы]</fieldset>

Например:

<fieldset name="info">

<p>Согласие <input name="agree" type="checkbox" />

<p><textarea rows="10" cols="30">О себе</textarea>

</fieldset>

Тогда внутри формы мы можем обратиться целиком к этому фрагменту следующим образом:

let info = frm.info;

и далее, обратиться к элементу подформы через коллекцию elements:

console.dir(info.elements[0]);

console.dir(info.elements.about);

Иногда это бывает удобно. Каждый элемент формы имеет ссылку на саму форму, которой он принадлежит:

console.dir(user.form);

Теперь, когда мы разобрались с навигацией по элементам форм, посмотрим как можно извлекать значения из них значения. Начнем с тегов

input и textarea

У них есть следующие свойства:

* input.value – значение атрибута value;
* input.checked – значение атрибута checked;
* textarea.value – содержимое поля textarea.

Обратите внимание, к полю textarea следует обращаться через свойство value, а не через innerHTML. Примеры:

let user = frm.user;

let sex = frm.sex;

let about = frm.about;

console.log(user.value, sex[0].checked, about.value);

Следующий тег

select

Он имеет три распространенных свойства:

* select.options – коллекция из подэлементов <option>;
* select.value – значение выбранного в данный момент <option>;
* select.selectedIndex – номер выбранного <option>.

Посмотрим их работу на следующем примере:

console.dir(city.options);

console.log(city.value);

console.log(city.selectedIndex);

Последние два свойства доступны и для записи значений:

city.value = "Тверь";

будет выбран пункт с таким городом;

city.selectedIndex = 1;

будет выбран второй пункт списка. Но что делать, если наш список имеет режим множественного выбора:

<select name="city" multiple>

В этом случае следует оперировать коллекцией options. Вот так мы можем выбрать сразу два последних элемента:

city.options[1].selected = **true**;

city.options[2].selected = **true**;

А так выбрать все отмеченные пункты:

let selected = Array.from(city.options)

      .filter(option => option.selected)

**for**(let c of selected)

      console.log(c);

Или, если нам нужны только города, то:

let selected = Array.from(city.options)

        .filter(option => option.selected)

        .map(option => option.value);

Вот так на уровне JavaScript выполняется навигация по формам и доступ к значениям их элементов.

# Управление фокусировкой

[Смотреть материал на видео](https://www.youtube.com/watch?v=-FhsHyujmto&list=PLA0M1Bcd0w8zri9wWiDFmOukXhguEYBBO" \t "_blank)

Рассмотренные на предыдущем занятии элементы формы:

input, textarea, select

обладают свойством фокусировки, то есть, когда мы кликаем по ним, они получают фокус и мы можем изменять состояние этих элементов. Например, в такой странице:

<!DOCTYPE html>

<html>

<head>

<title>Уроки по JavaScript</title>

</head>

<body>

<form name="reg">

<p><input name="user" value="" />

<p><input name="sex" type="radio" checked />М

<input name="sex" type="radio" />Ж

<p>Город: <select name="city">

<option>Москва</option>

<option>Санкт-Петербург</option>

<option>Тверь</option>

</select>

<p>Согласие <input name="agree" type="checkbox" />

<p><textarea name="about" rows="10" cols="30">О себе</textarea>

<p><input type="submit" name="submit">

</form>

<script>

</script>

</body>

</html>

щелкая по формам ввода, видим синюю подсветку элементов. Это как раз и есть визуальный признак получения фокуса. Когда мы уходим с элемента, он теряет фокус. Так вот, при получении или потери фокуса возникают события:

* focus – при получении фокуса;
* blur – при потере фокуса.

Давайте в этом убедимся. Повесим на первый элемент ввода обработчики этих двух событий:

let frm = document.forms[0];

frm.user.onfocus = **function**(event) {

    console.log("получение фокуса");

};

frm.user.onblur = **function**(event) {

    console.log("потеря фокуса");

};

Теперь, щелкая по элементу, появится сообщение «получение фокуса», то есть, сработало событие focus. А при переходе к другому элементу появляется событие «потеря фокуса», то есть, сработало событие blur.

Когда могут понадобиться такие события? Например, для проверки корректности ввода данных в поле ввода. Для простоты положим, что имя пользователя не должно содержать символа #. Такую проверку можно сделать так:

frm.user.onblur = **function**(event) {

**if**(frm.user.value.includes('#'))

               frm.user.classList.add('error');

**else** frm.user.classList.remove('error');

};

То есть, когда в поле ввода содержится символ #, то оно принимает красный фон. Соответственно стиль можно прописать так:

<style>

.error { background: #CC0000; }

</style>

А в обработчике фокуса будем удалять этот класс, то есть, убирать красный фон:

frm.user.classList.remove('error');

Однако, обратите внимание, стандарты HTML 5 и выше позволяют в полях ввода прописывать атрибуты

required, pattern

для автоматической проверки корректности введенных данных. Так что скрипт следует использовать, только если недостаточно встроенных способов валидации данных.

Также следует иметь в виду, что события focus и blur не всплывают, то есть, мы не можем перехватить их обработчиком элемента более высокого уровня в иерархии DOM-дерева. Например, если поставить обработчик onfocus на форму:

frm.onfocus = **function**() {

     console.log("form: focus");

}

то он никогда не будет срабатывать. Однако, тут есть один нюанс: события focus и blur хоть и не всплывают, но зато имеют фазу погружения и мы можем их перехватить на форме в этот момент:

frm.addEventListener("focus", **function**() {

         console.log("form: focus");

}, **true**);

Теперь, кликая по элементам формы, мы будем видеть сообщения в консоли. Но, все же, если нам нужно перехватывать события фокусировки, то лучше использовать для этого специальные события:

focusin и focusout

которые работают также как и focus и blur, но всплывают. Обработчики этих событий можно добавлять только через метод addEventListener, следующим образом:

frm.addEventListener("focusin", **function**() {

            console.log("form: focusin");

});

Теперь событие успешно перехватывается на этапе всплытия.

В JavaScript существуют методы:

* elem.focus() – для установки фокуса на элемент;
* elem.blur() – для снятия фокуса с элемента.

Например, так:

frm.about.focus();

и при обновлении документа мы увидим фокусировку на теге textarea. Аналогично работает второй метод. Но здесь есть несколько тонких моментов. Во-первых, эти методы могут не работать в браузерах Firefox. Во-вторых, бывают случаи, когда фокусировка с элемента будет сниматься автоматически. Например, при появлении модальных окон, вызванных методами:

alert(), prompt(), confirm()

Так что полагаться целиком на методы focus и blur не стоит и в практике программирования ими стараются не пользоваться. Вместо этого лучше полагаться на события focus и blur и реализовывать логику через их обработчики. Или же, используя псевдокласс :focus, определять стили сфокусированного элемента.

Работая с событиями focus и blur, следует иметь в виду, что далеко не все элементы HTML-документа по умолчанию поддерживают фокусировку. Например, элементы:

div, p, table, span, img и т.п.

Добавим div на нашу страницу:

<div id="block" style="height:50px; background: #777"></div>

И пропишем для него два обработчика:

block.onfocus = **function**(event) {

      console.log("focus");

};

block.onblur = **function**(event) {

      console.log("blur");

};

Кликая по блоку div мы не увидим никаких сообщений в консоли. Это, как раз, и говорит о том, что данный элемент не имеет фокусировки. Но ее можно включить, если прописать вот такой атрибут:

tabindex="1"

Теперь, при щелчке мыши по блоку div, будут срабатывать эти два события. Что это за атрибут tabindex? Он указывает в какой последовательности следует переключать фокусы между элементами, при нажатии на клавишу Tab. Так что, если мы какой-то элемент включаем в эту последовательность, то он должен обладать фокусировкой. И браузеры ее включают, даже если, ранее элемент не обладал таким свойством. Причем, это свойство можно установить и непосредственно в скрипте, используя свойство tabIndex:

block.tabIndex = 1;

Обратите внимание на его написание: здесь буква I заглавная.

У этого атрибута (или же свойства) есть два специальных значения:

* tabindex="0" располагает элемент на уровне с другими элементами без атрибута tabindex, но обладающих фокусировкой;
* tabindex="-1" позволяет фокусироваться на элементе только программно.

Ну и, наконец, последнее. Для получения текущего элемента с фокусировкой, используется свойство:

let active = document.activeElement;

console.log(active);

И в консоли мы увидим элемент textarea, который имеет фокус на странице.

# События форм

[Смотреть материал на видео](https://www.youtube.com/watch?v=kcA5zYeeXB8&list=PLA0M1Bcd0w8zri9wWiDFmOukXhguEYBBO" \t "_blank)

<https://proproprogs.ru/javascript_dom/sobytiya-form>

Это занятие начнем с рассмотрения событий

change, input, cut, copy, paste

для элементов формы input и select. Предположим, что у нас имеется вот такая форма:

<!DOCTYPE html>

<html>

<head>

<title>Уроки по JavaScript</title>

</head>

<body>

<form name="reg">

<p><input name="user" value="" />

<p><input name="sex" type="radio" checked />М

<input name="sex" type="radio" />Ж

<p>Город: <select name="city">

<option>Москва</option>

<option>Санкт-Петербург</option>

<option>Тверь</option>

</select>

<p>Согласие <input name="agree" type="checkbox" />

<p><textarea name="about" rows="10" cols="30">О себе</textarea>

<p><input type="submit" name="submit">

</form>

<script>

</script>

</body>

</html>

Как правильно узнать: в какой момент следует считывать значение, например, из поля ввода user? Для этого используется событие change, возникающее после потери фокуса элементом. Например:

let frm = document.forms[0];

frm.user.onchange = **function**() {

            console.log(frm.user.value);

}

Теперь, при вводе значения и перемещения фокуса на следующий элемент, срабатывает событие change и в консоль выводится введенное значение. Причем, если значение в элементе не было изменено, то событие не сработает. Это принципиальное отличие события change от blur.

Если же повесить такой обработчик на элементы radio или checkbox, то оно будет срабатывать сразу при их изменении:

frm.sex[0].onchange = **function**() {

            console.log("sex 0: "+frm.sex[0].checked);

}

frm.sex[1].onchange = **function**() {

            console.log("sex 1: "+frm.sex[1].checked);

}

frm.agree.onchange = **function**() {

            console.log("agree: "+frm.agree.checked);

}

То же самое и для элемента select:

frm.city.onchange = **function**() {

            console.log(frm.city.selectedIndex);

}

Для поля ввода есть еще одно полезное событие

input

которое тут же срабатывает при изменении значения, например:

frm.user.oninput = **function**() {

         console.log(frm.user.value);

}

Если же значение не меняется, а мы просто перемещаем курсор по тексту или выделяем его, то это событие не происходит. Его удобно использовать, если нам нужно «на лету» анализировать введенные данные пользователем.

Действие этого события по умолчанию нельзя отменить методом:

event.preventDefault()

так как оно происходит уже после того, как браузер его обработал.

Следующие три события:

* cut – при вырезании фрагмента в буфер обмена;
* copy – при копировании фрагмента в буфер обмена;
* paste – при вставке фрагмента из буфера обмена.

Реализуются следующим образом:

frm.user.oncut = **function**(event) {

            console.log("cut");

}

frm.user.oncopy = **function**(event) {

            console.log("copy");

}

frm.user.onpaste = **function**(event) {

            console.log("paste");

}

Причем, все они могут быть отменены методом

event.preventDefault();

то есть, браузер по умолчанию не будет выполнять вырезание/копирование/вставку фрагментов текста.

## submit

Следующее событие submit возникает при отправке формой запроса на сервер. Реализуется оно следующим образом:

frm.onsubmit = **function**(event) {

            alert("отправка формы");

}

В этом обработчике мы можем, например, проверить корректность заполнения данных формы и прервать отправку, при возникновении ошибок:

frm.onsubmit = **function**(event) {

**if**(!frm.user.value) {

              event.preventDefault();

              console.log("поле user пустое");

     }

}

Также в JavaScript существует метод:

form.submit()

который отправляет данные на сервер, но только для формы, созданной непосредственно в скрипте. При этом событие submit не генерируется. Например:

<p onclick="sendForm()">Отправить форму</p>

и обработчик:

**function** sendForm() {

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

form.method = 'GET';

form.innerHTML = '<input name="q" value="test">';

document.body.append(form);

form.submit();

}

Обратите внимание, если мы будем использовать форму документа:

reg.submit();

то этот метод работать не будет!

Итак, на этом занятии мы с вами рассмотрели метод submit и события:

change, input, cut, copy, paste, submit.