**Краткое перечисление основных свойств и методов DOM:**

**https://learn.javascript.ru/dom-cheatsheet**

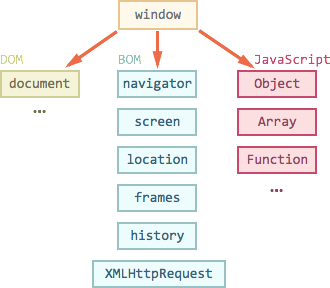
**ЭЛЕМЕНТЫ JAVASCRIPT В СТРУКТУРЕ HTML**

Объектная модель документа (**DOM** – document object model) – для работы с содержимым страницы

Объектная модель браузера (**BOM**) - это объекты для работы с чем угодно, кроме документа:

Объект **navigator** содержит общую информацию о браузере и операционной системе. Особенно примечательны два свойства: **navigator.userAgent** — содержит информацию о браузере и **navigator.platform** — содержит информацию о платформе, позволяет различать Windows/Linux/Mac и т.п.

Объект **location** содержит информацию о текущем URL страницы и позволяет перенаправить посетителя на новый URL

****

**Глобальный обьект window**

**window** стоит на вершине иерархии javascropt в браузере. У этого объекта двоякая позиция — он с одной стороны является глобальным объектом в JavaScript, с другой — содержит свойства и методы для управления окном браузера.

В обьекте window есть много свойст и конструкторов, в том числе Array, Function, Object и т.д.

Обьявленная глобально переменная **var a === window.a**

У обьекта window есть **свойство** **window**, которое ссылается на обьтект window. Чистая рекурсия

**МОДЕЛЬ BOM (Oбьектная модель браузера)**

**Методы Location**

**location.reload() -** страница будет перезагружаться бесконечно

**location.assign()** - тоже самое, что присваивание href какого-то пути

**location.replace()** - тоже самое, но браузер уберет из истории текущую страницу, потому мы не сможем вернуться на нее, нажав кнопку "назад"

**encodeURI()** заменяет человеческие символы на нечеловеческий код последовательностей

**decodeURI()** - берет код с последовательностями и возвращает нормальную строку

Функции **alert/confirm/prompt** — тоже входят в **BOM**.

**Свойства Location**

Позволяет работать с адресной строкой браузера. Посмотреть в бразуере

console.log(window.location)

Все свойства доступны для записи. Hash - один из способов сохранения состояния в одностраничных приложениях

location.hash = "anything"; //sorax:8000/#anything

**window.onhashchange -** событие, которое следит за изменением hash

**loccation.href** = "http://google.com" можно полностью изменить адрес на свой

**loccation** = "http://google.com" можно присваивать адрес напрямую, так же будет работать

**location.toString** - вернет значение location, потому при конкатенации можно писать сразу **location:**

console.log("Current URL is " + location) //выведется текущий адрес

Свойства **hash, host, hostname, origin, pathname, port, protocol** - лишь декомпозиция полного URL

**location.search** = "mysearch" // часть URL, которая идет после знака **?,** т.е. значения, которые передаются на сервер **get-запросом**

**Обьект History**

**history.PushState** - меняет историю в реальном времени на то, что подставляем

**history.replaceState** - заменяет последнюю запись в истории.

При помощи обьекта мы можем управлять историей браузера. Когда пользователь посещает страницу, длинна history увелчивается на единицу. Получать ссылки из истории невозможно, по причинам безопасности

console.log(history.length)

history.back() - перейти на предыдущую страницу

history.forward() - перейти на следующую страницу

history.go(-2) - перейти на 2 страницы назад

Для работы с историей приложений, которые работают без перезагрузки страницы (одностраничные приложения), кнопки "вперед" и "назад" теряют смысл. Существует два метода работы с ними:

* **Хеши в url, что бы можно было менять контент при изменении хеша:**

<body>

<main class="main-container">

<ul class="top-nav">

<li><a href="#home">Домой</a></li> // в ссылках нужен хеш - решетка

<li><a href="#features">Особенности</a></li>

<li><a href="#partners">Партнеры</a></li>

<li><a href="#contacts">Контакты</a></li>

</ul>

</main>

<div class="content h2" style="text-align: center">Сейчас открыта страничка Домой</div>

</body>

Что пишем в JS:

var links, updateState, contentEl;

contentEl = document.querySelector(".content");

links = {

home: "Сейчас открыта страничка Домой",

features: "Сейчас открыта страничка Особенности",

partners: "Сейчас открыта страничка Партнеры",

contacts: "Сейчас открыта страничка Контакты"

};

updateState = function(){

var content = links[location.hash.slice(1)];//обрезаем первый символ (решетку)

contentEl.innerHTML = content || "Page not found"; //если хеш отсутствует

};

window.addEventListener("hashchange", updateState); //работает только под window

window.addEventListener("load", updateState); //при перезагрузке ссылка остается

* **HTML5 History API, пример с той же разметкой, переписан только JS:**

**var links, updateState, contentEl, topMenu, activeButton;**

**contentEl = document.querySelector(".content");**

**topMenu = document.querySelector(".top-nav");**

**links = {**

**home: "Сейчас открыта страничка Домой",**

**features: "Сейчас открыта страничка Особенности",**

**partners: "Сейчас открыта страничка Партнеры",**

**contacts: "Сейчас открыта страничка Контакты"**

**};**

**updateState = function(state){** //заменяем содержимое и добавляем класс active для li

**if(!state) return;**

**contentEl.innerHTML = links[state.page];**

**activeButton(state);**

**};**

**activeButton = function(state){** //<li class="active">

**[].slice.call(topMenu.querySelectorAll("a")).forEach(function(e){**

**var classList = e.parentNode.classList;**

**state.page === e.getAttribute('href')**

**? classList.add('active')**

**: classList.remove('active');**

**});**

**};**

**window.addEventListener("popstate", function(e){**

**updateState(e.state)**

**});**

**topMenu.addEventListener("click", function(e){**

**var state;**

**if (e.target.tagName !== "A") return;**

**state = { page: e.target.getAttribute('href') };**

**history.pushState(state,'', state.page);**

**updateState(state);**

**e.preventDefault();**

**});**

**Обьект Screen**

**screen.width, screen.height** - можно узнать разрешение экрана пользователя

**screen.availWidth, screen.availHeight** - максимально доступное разрешение, в котором можно открыть браузер **не** в полноэкранном режиме

**screen.colorDepth** - глубина цвета, количество бит на пиксель

**МОДЕЛЬ DOM (Обьектная модель документа)**

**Навигация по DOM-узлам**

**Всё, что есть в HTML, находится и в DOM.**

**DOM-модель** — это внутреннее представление **HTML**-страницы в виде дерева.

Все элементы страницы, включая теги, текст, комментарии, являются **узлами** **DOM**.

У элементов **DOM** есть **свойства и методы**, которые позволяют изменять их.

Пробелы до <head> по стандарту игнорируются, а любое содержимое после </body> не создаёт узла, браузер переносит его внутрь, в конец body.

Всего различают 12 типов узлов, но на практике мы работаем с четырьмя из них:

* **Документ** — точка входа в DOM.
* **Элементы** — основные строительные блоки.
* **Текстовые узлы** — содержат, собственно, текст.
* **Комментарии** — иногда в них можно включить информацию, которая не будет показана, но доступна из JS.

Важный «особый случай» при работе с DOM — **таблицы**. По стандарту DOM они обязаны иметь **<tbody>,** однако в HTML их можно написать без него. В этом случае браузер добавляет **<tbody>** самостоятельно.

Узел HTML можно получить как document.documentElement, а BODY — как document.body

**Работа с console в элементах разработчика Chrome**

Зачастую бывает нужно выбрать элемент DOM и сделать с ним что-то на JavaScript.

Находясь во вкладке **Elements**, откройте консоль нажатием **Esc** (или перейдите на вкладку **Console**).

Последний элемент, выбранный во вкладке **Elements**, доступен в консоли как **$0**, предыдущий — **$1** и так далее:

$0.style.backgroundColor = "red";

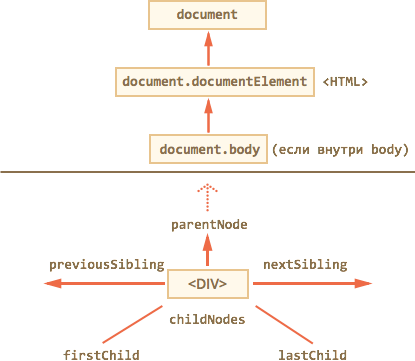
Для **поиска элементов** в консоли есть два специальных метода:

**$$("div.my")** — ищет все элементы в DOM по данному CSS-селектору.

**$("div.my")** — ищет первый элемент в DOM по данному CSS-селектору.

Более полная документация по методам консоли доступна на страницах **Console API Reference** для Chrome и Command Line API для Firebug, а также на firebug.ru.

**НАВИГАЦИЯ ПО DOM-ЭЛЕМЕНТАМ**

****

<HTML> = document.documentElement

<BODY> = document.body

<HEAD> = document.head

* Каскадность DOM: если скрипт находится в <head>, то в нём будет недоступен document.body.
* В мире DOM в качестве значения, обозначающего «нет такого элемента» или «узел не найден», используется не undefined, а **null**

**childNodes** хранит **все дочерние элементы первого уровня**, включая **текстовые узлы**. Список дочерних элементов — только для чтения!

**.children –** свойство Элемента. Это только дочерние узлы-элементы, то есть соответствующие тегам, и они включают в себя все уровни вложенности, в отличии от **childNodes**

**elem.parentNode** возвращает родитель элемента

**parentElement** — родитель-элемент. Новый элемент, разница с **parentNode:** если у элемента нету родителя, parentElement == null, т.к. он работает только в <html>. А **parentNode** выдаст верхний узел.

**hasChildNodes()** –проверяет, имеет ли элемент детей

**firstChild** –ссылку на первый узел (элемент, текст, комментарий)

**firstElementChild** –возвращает ссылку на первый дочерний элемент (htm-тэг)

**previousElementSibling,** **nextElementSibling** — соседи-элементы.

**DOM-коллекции**, такие как childNodes и другие, не являются JavaScript-массивами, и у них нету методов, которые есть у массивов. **Так же, для коллекций неправильно работает for...in**

Для того, что бы сделать из них массив, есть 2 метода:

1. Скопировать методы из массива

var elems = document.documentElement.childNodes;

[].call(elems)

1. Сделать из коллекции массив

elems = Array.prototype.slice.call(elems); // теперь elems - массив

**Для таблиц свои методы:**

TABLE

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 несколько.

THEAD/TFOOT/TBODY

tbody.rows — коллекция строк TR секции.

TR

tr.cells — коллекция ячеек TD/TH

tr.sectionRowIndex — номер строки в текущей секции THEAD/TBODY

tr.rowIndex — номер строки в таблице

TD/TH

td.cellIndex — номер ячейки в строке

Пример использования:

<table>

<tr>

<td>один</td> <td>два</td>

</tr>

<tr>

<td>три</td> <td>четыре</td>

</tr>

</table>

<script>

var table = document.body.children[0];

alert( table.rows[0].cells[0].innerHTML ) // "один"

</script>

**ПОИСК: getElement() & querySelector()**

Результаты поиска **getElementsBy\*** — живые! При изменении документа — изменяется и результат запроса. Полезно, когда нужно оперативно реагировать на меняющийся DOM, однако, эти методы медленне других поисковых.

**document.getElementById('ID') –** самый быстрый способ найти элемент. Ищет по всему документу, но необходим уникальный **id**

**document.getElementsByTagName('div') -** в отличие от getElementById, который существует только в контексте document, метод getElementsByTagName может искать внутри **любого элемента**.

document.getElementsByTagName('\*'); // получить все элементы документа

elem.getElementsByTagName('\*'); // получить всех потомков элемента elem

Возвращается **коллекция**, а не элемет. Если необходимо что-то получить от найденных тегов, например value, для начала их надо перебрать в массиве или получить элемент по номеру

document.getElementsByTagName('input')[0].value = 5;

**getElementsByClassName** - возвращает коллекцию элементов с классом className. Находит элемент и в том случае, если у него несколько классов, а искомый – один из них.

<div class="article">Статья</div>

<div class="long article">Длинная статья</div>

<script>

var articles = document.getElementsByClassName('article');

alert( articles.length ); // 2, найдёт оба элемента

</script>

**elem.querySelectorAll(css)** возвращает все элементы внутри elem, удовлетворяющие CSS-селектору css. Этот перебор происходит очень быстро, так как осуществляется непосредственно движком браузера, а не JavaScript-кодом.

Он просто перебирает все подэлементы внутри элемента elem(или по всему документу, если вызов в контексте документа) и проверяет каждый элемент на соответствие запросу query. querySelectorAll собирает найденные элементы в «псевдомассив»: внутреннюю структуру данных, по сути аналогичную массиву JavaScript.

var elements = document.querySelectorAll('ul > li:last-child');

**elem.querySelector(css)** возвращает не все, а только первый элемент, соответствующий CSS-селектору css.

**elem.matches(css)** ничего не ищет, а проверяет, удовлетворяет ли elem селектору css. Он возвращает true либо false. Этот метод бывает полезным, когда мы перебираем элементы (в массиве или по обычным навигационным ссылкам) и пытаемся отфильтровать те из них, которые нам интересны.

var elem = document. querySelector("active");

if(elem.matches("a[href$="zip"]") {alert("Hola")

**elem.closest(css)** ищет ближайший элемент выше по иерархии DOM, подходящий под CSS-селектор css. Сам элемент тоже включается в поиск.

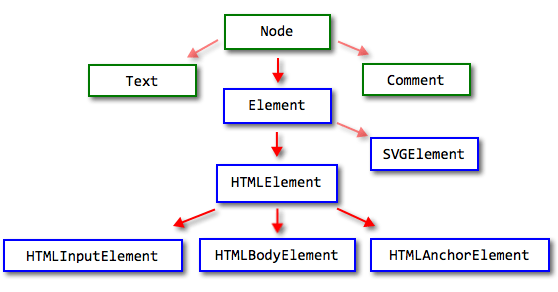
var elem = document.querySelector('li ');

elem.closest('.active') //найдет первый li с классом .active

**СВОЙСТВО УЗЛОВ**

Классы DOM образуют иерархию.

Основной объект в ней: **Node**, от которого наследуют остальные – text, element и comment

****

Узнать класс узла - привести его к строке:

alert(document.body); // [object HTMLBodyElement]

* **console.log** выводит элемент в виде, удобном для исследования HTML-структуры.
* **console.dir** выводит элемент в виде JavaScript-объекта, удобно для анализа его свойств.

Тип узла содержится в его свойстве **nodeType**

**nodeName и tagName** содержат название(тег) элемента узла. При помощи tagName мы можем работать только с элементами, а nodeName может что-то сказать и о других типах узлов (коментарии, пробелы, текст и т.д)

<body>

<!-- комментарий -->

<script>

// для комментария

alert( document.body.firstChild.nodeName ); // #comment

alert( document.body.firstChild.tagName ); // undefined

// для документа

alert( document.nodeName ); // #document, т.к. корень DOM -- не элемент

alert( document.tagName ); // undefined

</script>

</body>

**innerHTML -** внутреннее содержимое узла-элемента в виде HTML. Можно изменять.

Значение, возвращаемое innerHTML — всегда валидный HTML-код. При записи можно попробовать записать что угодно, но браузер исправит ошибки. Понимает теги внутри строки

<body>

<script>

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

alert( document.body.innerHTML ); // тест (исправлено)

</script>

</body>

Добавление **innerHTML+=**  осуществляет перезапись

Так как новое значение записывается с нуля, то все изображения и другие ресурсы будут перезагружены. Если в элементе много контента, задержка будет заметна

**outerHTML** - полный HTML узла-элемента. При записи в elem.outerHTML переменная elem сохраняет старое значение, она "не живая". Потому если еще нужно значение этого элемента, его нужно переназначить после замены

<div>Привет, Мир!</div>

<script>

var div = document.body.children[0];

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

div.outerHTML = '<p>Новый элемент!</p>';

// ... но содержимое div.outerHTML осталось тем же, несмотря на "перезапись"

alert( div.outerHTML ); // <div>Привет, Мир!</div>

</script>

**data.** Свойство innerHTML есть только у узлов-элементов**.** Содержимое других узлов, например, текстовых или комментариев, доступно на чтение и запись через свойство **data.**

Содержимое **других узлов**, например, текстовых или комментариев, доступно на чтение и запись через свойство **data**

<body>

Привет

<!-- Комментарий -->

<script>

document.body.childNodes[0] //содержит привет, т.к. это текст вне тэга

document.body.childNodes[1].data //комментарий тоже вне тэга

}

</script>

</body>

**elem.textContent** содержит только текст внутри элемента, за вычетом всех <тегов>. Оно берет и показывает текст внутри тегов, теги просто удаляя, как если бы текст выглядел на странице. Но при записи через textContent, все теги будут расценены как текст, в отличии от innerHTML:

elem.textContent = "<b>Винни-пух</b>" // "<b>Винни-пух</b>

elem.innerHTML = "<b>Винни-пух</b>" // Винни-пух

**hidden.** Технически, атрибут **hidden** работает так же, как **style="display:none".** Но его проще поставить через JavaScript (меньше букв), и могут быть преимущества для скринридеров и прочих нестандартных браузеров.

elem.hidden = true; // элемент будет скрыт, как при display:none

**Исследовать элемент**, что бы узнать какие у него свойства, проще всего через **console.dir**

**DOM: ПОЛИФИЛЫ**

Проверить на поддерживаемость:[**http://caniuse.com**](http://caniuse.com)

Обычно полифилл состоит из двух частей:

* Проверка, есть ли встроенная возможность.
* Эмуляция, если её нет.

if (document.documentElement.firstElementChild === undefined) {

Object.defineProperty(Element.prototype, 'firstElementChild', {

get: function() {

var el = this.firstChild;

do {

if (el.nodeType === 1) {

return el;

}

el = el.nextSibling;

} while (el);

return null;

}

});

}

Полифилы работают только для тех тегов, стандарт которых поддерживает наличие свосйства

**АТРИБУТЫ И DOM-СВОЙСТВА**

При чтении HTML браузер генерирует DOM-модель. При этом большинство стандартных HTML-атрибутов становятся свойствами соответствующих объектов.

Например, если тег выглядит как **<body id="page">,** то у объекта будет свойство **body.id = "page"**

**Атрибуты** — это то, что написано в HTML, видны в innerHTML

**Свойство** — это то, что находится в свойстве DOM-объекта, не видны в innerHTML

А действительно нужны атрибуты очень редко – лишь в следующих трёх случаях:

* Когда нужно кросс-браузерно получить нестандартный HTML-атрибут.
* Когда нужно получить «оригинальное значение» стандартного HTML-атрибута, например, <input value="...">.
* Когда нужно получить список всех атрибутов, включая пользовательские. Для этого используется коллекция attributes.

**Атрибуты**

Элементам DOM, с другой стороны, соответствуют HTML-теги, у которых есть текстовые атрибуты. Атрибуты всегда видны в innerHTML, в отличии от свойств обьектов

**elem.hasAttribute(name)** – проверяет наличие атрибута

**elem.getAttribute(name)** – получает значение атрибута

**elem.setAttribute(name, value)** – устанавливает атрибут

**elem.removeAttribute(name)** – удаляет атрибут

Эти методы возвращают именно то значение, которое находится в HTML.

Также все атрибуты элемента можно получить с помощью свойства **elem.attributes**, которое содержит псевдо-массив объектов типа Attr.

Синхронизация **не гарантирует** **одинакового** значения в атрибуте и свойстве.

Для примера, посмотрим, что произойдет с атрибутом "**href**" при изменении свойства:

<a id="a" href="#"></a>

<script>

a.href = '/';

alert( 'атрибут:' + a.getAttribute('href') ); // '/'

alert( 'свойство:' + a.href ); // полный URL

</script>

Это происходит потому, что атрибут может быть любым, а свойство **href**, в соответствии со спецификацией W3C, должно быть полной ссылкой.

Стало быть, если мы хотим именно то, что в HTML, то нужно обращаться через атрибут.

Обычно синхронизация односторонняя: **свойство зависит от атрибута, но не наоборот.**

Атрибуту "**class**" соответствует свойство **className.**

<body class="main page">

<script>

alert( document.body.className ); // main page

document.body.className = "class1 class2"; // поменять класс элемента

</script>

</body>

Атрибуту **for (<label for="...">)** соответствует свойство с названием **htmlFor**.

Свойство **classList** — это объект для работы с классами. Методы **classList**:

**elem.classList.contains("class")** — возвращает true/false, в зависимости от того, есть ли у элемента класс class.

**elem.classList.add/remove("class")** — добавляет/удаляет класс class

**elem.classList.toggle("class")** — если класса class нет, добавляет его, если есть — удаляет.

Для нестандартных атрибутов DOM-свойство не создаётся.

<div id="elem" href="http://ya.ru" about="Elephant"></div>

<script>

alert( elem.id ); // elem

alert( elem.about ); // undefined

</script>

**Свойство dataset, data-атрибуты**

С помощью нестандартных атрибутов можно привязать к элементу данные, которые будут доступны в JavaScript. Как правило, это делается при помощи атрибутов с названиями, начинающимися на **data-,** например:

<div id="elem" data-about="Elephant" data-user-location="street"></div>

<script>

alert( elem.getAttribute('data-about') ); // Elephant

alert( elem.getAttribute('data-user-location') ); // street

</script>

Стандарт HTML5 специально разрешает атрибуты **data-\*** и резервирует их для пользовательских данных.

При этом во всех браузерах, кроме IE10-, к таким атрибутам можно обратиться не только как к атрибутам, но и как к свойствам, при помощи специального свойства **dataset**:

<div id="elem" data-about="Elephant" data-user-location="street">

По улице прошёлся слон. Весьма красив и толст был он.

</div>

<script>

alert( elem.dataset.about ); // Elephant

alert( elem.dataset.userLocation ); // street

</script>

Обратим внимание — название **data-user-location** трансформировалось в **dataset.userLocation.** Дефис превращается в большую букву.

Пример: Сделайте желтыми внешние ссылки, добавив им класс external. Все ссылки без href, без протокола и начинающиеся с http://internal.com считаются внутренними. Решение:

// ищем все ссылки, у которых в href есть протокол,

// но адрес начинается не с http://internal.com

var css = 'a[href\*="://"]:not([href^="http://internal.com"])';

var links = document.querySelectorAll(css);

for (var i = 0; i < links.length; i++) {

links[i].classList.add('external');

}

**Необходимо стараться использовать везде свойства, а атрибуты - только когда необходимо**

**МЕТОДЫ CONTAINS И COMPAREDOCUMENTPOSITION**

**contains** для проверки на вложенность. Возвращает true, если parent содержит child или parent == child.

/\*Возвращает true, если parent содержит child или parent == child.\*/

var result = parent.contains(child);

**compareDocumentPosition** для порядка узлов

Биты Число Значение

000000 0 nodeA и nodeB — один и тот же узел

000001 1 Узлы в разных документах (или один из них не в документе)

000010 2 nodeB предшествует nodeA (в порядке обхода документа)

000100 4 nodeA предшествует nodeB

001000 8 nodeB содержит nodeA

010000 16 nodeA содержит nodeB

100000 32 Зарезервировано для браузера

<p>...</p>

<ul>

<li>1.1</li>

</ul>

<script>

var p = document.body.children[0];

var ul = document.body.children[1];

var li = ul.children[0];

// 1. <ul> находится после <p>

alert( ul.compareDocumentPosition(p) ); // 2 = 10

// 2. <p> находится до <ul>

alert( p.compareDocumentPosition(ul) ); // 4 = 100

// 3. <ul> родитель <li>

alert( ul.compareDocumentPosition(li) ); // 20 = 10100

// 4. <ul> потомок <body>

alert( ul.compareDocumentPosition(document.body) ); // 10 = 1010

</script>

**ДОБАВЛЕНИЕ, УДАЛЕНИЕ И КОПИРОВАНИЕ УЗЛОВ**

**document.createElement(tag)** создает новый элемент с указанным тегом:

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

**document.createTextNode(text)** создает новый текстовый узел с данным текстом. не понимает теги внутри строки, в отличии от **innerHTML:**

<div id="elem"></div>

<script>

var text = '<b>текст</b>';

elem.appendChild(document.createTextNode(text)); //<b>текст</b>, выведет теги

</script>

**parentElem.appendChild(elem)** добавляет elem в конец дочерних элементов parentElem.

**parentElem.insertBefore(elem, nextSibling)** вставляет elem в коллекцию детей parentElem, перед элементом nextSibling.

<ol id="list">

<li>0</li>

<li>1</li>

<li>2</li>

</ol>

<script>

var newLi = document.createElement('li');

newLi.innerHTML = 'Привет, мир!';

</script>

Для вставки элемента в начало достаточно указать, что вставлять будем перед первым потомком:

list.insertBefore(newLi, list.children[1]);

Если вторым аргументом указать **null**, то **insertBefore** сработает как **appendChild**:

parentElem.insertBefore(elem, null) //=== parentElem.appendChild(elem)

**elem.cloneNode(true)** создаст «глубокую» копию элемента — вместе с атрибутами, включая подэлементы. Если же вызвать с аргументом false, то копия будет сделана без дочерних элементов.

<script>

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

div.className = "alert alert-success";

div.innerHTML = "<strong>Ура!</strong> Вы прочитали это важное сообщение.";

document.body.insertBefore(div, document.body.firstChild);

var div2 = div.cloneNode(true); // создать копию узла

div2.querySelector('strong').innerHTML = 'Супер!'; // копию можно подправить

// вставим её после текущего сообщения

div.parentNode.insertBefore(div2, div.nextSibling); \*

</script>

\*Для вставки нам нужен будущий родитель. Мы, возможно, не знаем, где точно находится div (или не хотим зависеть от того, где он), но если нужно вставить рядом с div, то родителем определённо будет div.parentNode.

Мы хотели бы вставить после div, но метода insertAfter нет, есть только insertBefore, поэтому вставляем перед его правым соседом div.nextSibling.

**parentElem.removeChild(elem)** удаляет elem из списка детей parentElem.

**parentElem.replaceChild(newElem, elem)** среди детей parentElem удаляет elem и вставляет на его место newElem. Оба этих метода возвращают удаленный узел, то есть elem.

**elem.remove()**  удаляет элемент напрямую, не требуя ссылки на родителя. Это зачастую удобнее, чем removeChild. НЕ работает в IE11-

<script>

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

div.className = "alert alert-success";

div.innerHTML = "<strong>Ура!</strong> Вы прочитали это важное сообщение.";

document.body.appendChild(div);

setTimeout(function() {

div.parentNode.removeChild(div); //через секунду сообщение пропадёт

}, 1000);

</script>

**Мультивставка: insertAdjacentHTML и DocumentFragment**

Обычно, в браузерах такая схема работает быстрее - полностью создать список «вне DOM», а потом - вставить в документ:

var ul = document.createElement ('ul');

for (...) ul.appendChild (li) // Сначала вставить узлы

document.body.appendChild (ul); // Затем в документ

**insertAdjacentHTML** позволяет вставлять произвольные HTML в любое место документа, в том числе и между узлами. **БЕЗ перезаписи,** в отличии от innerHTML

elem.insertAdjacentHTML (where, html)

**html** - Строка HTML, которую нужно вставить

**where** Куда по отношении к elem вставлять срока. Всего четыре варианта:

* beforeBegin - перед elem.
* afterBegin - внутрь elem, в самое начало.
* beforeEnd - внутрь elem, в конец.
* afterEnd - после elem.

<ul>

<li>1</li>

<li>2</li>

<li>5</li>

</ul>

<script>

var ul = document.body.children[0];

var li5 = ul.children[2];

//вставим пропущенные элементы списка перед <li>5</li>:

li5.insertAdjacentHTML("beforeBegin", "<li>3</li><li>4</li>");

</script>

**elem.insertAdjacentElement(where, newElem)** — вставляет в произвольное место не строку HTML, а элемент newElem.

**elem.insertAdjacentText(where, text)** — создаёт текстовый узел из строки text и вставляет его в указанное место относительно elem.

**document.createDocumentFragment() – для старых браузеров.** У метода нет обычных свойств DOM-узлов, таких как innerHTML, tagName и т.п. Это не узел. Когда DocumentFragment вставляется в DOM — то он исчезает, а вместо него вставляются его дети.

**node.append(...nodes)** — вставляет nodes в конец node,

**node.prepend(...nodes)** — вставляет nodes в начало node,

**node.after(...nodes)** — вставляет nodes после узла node,

**node.before(...nodes)** — вставляет nodes перед узлом node,

**node.replaceWith(...nodes)** — вставляет nodes вместо node.

Во всех этих методах nodes — DOM-узлы или строки, в любом сочетании и количестве. Причём строки вставляются именно как **текстовые узлы**, в отличие от insertAdjacentHTML. Методы новые, в старых браузерах могут не работать

<script>

var p = document.createElement('p'); // добавим элемент в конец <body>

document.body.append(p);

var em = document.createElement('em');

em.append('Мир!');

p.append("Привет, ", em); // вставить в параграф текстовый и обычный узлы

p.after(document.createElement('hr')) // добавить элемент после <p>

</script>

**document.write** – работает только пока HTML-страница находится в процессе загрузки. Он дописывает текст в текущее место HTML ещё до того, как браузер построит из него DOM.метод пишет текст прямо в HTML, как будто он там всегда был. Нет никаких ограничений на содержимое document.write.

**СТИЛИ ЭЛЕМЕНТОВ: STYLE**

**background-color =>** elem.style.backgroundColor

**z-index =>** elem.style.zIndex

**border-left-width =>** elem.style.borderLeftWidth

**elem.style.cssFloat =>** используется вместо flloat (слово зарезервировано)

Чтобы сбросить поставленный стиль, присваивают в style пустую строку: elem.style.width=""

При сбросе свойства style стиль будет взят из CSS.

elem.style.display = "none".

elem.style.display = ""

**style.cssText** позволяет поставить стиль целиком в виде строки. При установке style.cssText все предыдущие свойства style удаляются.

<script>

var div = document.body.children[0];

div.style.cssText="color: red !important; \

background-color: yellow; \

width: 100px; \

text-align: center; \

blabla: 5; \ //будет проигнорирован, браузер не знает это свойство

";

alert(div.style.cssText);

</script>

**Чтение свойств стилей**

**window.getComputedStyle** позволяет получить текущее используемое значение свойств стилей. Надо указывать полное свойство, при обращении к сокращенному: padding, margin, border — правильный результат не гарантируется.

getComputedStyle(element[, pseudo])

**element** - элемент, значения для которого нужно получить

**pseudo -** Указывается, если нужен стиль псевдо-элемента, например "::before". Пустая строка или отсутствие аргумента означают сам элемент.

<style>

body {

margin: 10px

}

</style>

<body>

<script>

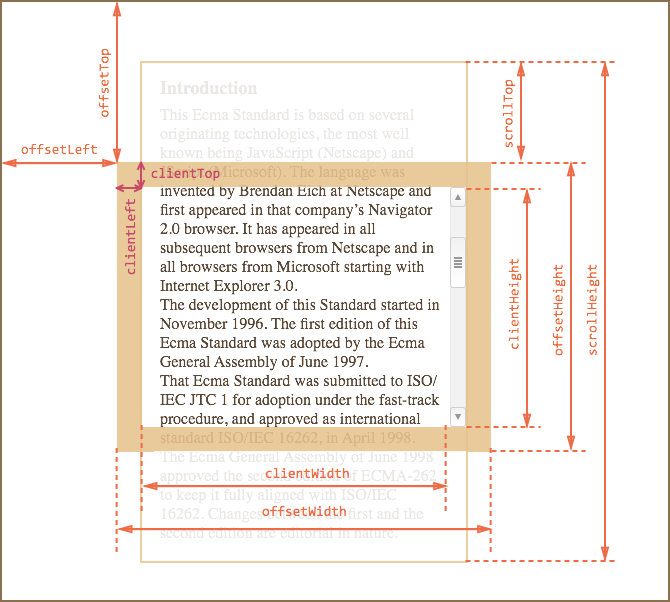
var computedStyle = getComputedStyle(document.body);

alert( computedStyle.marginTop ); // выведет отступ в пикселях

alert( computedStyle.color ); // выведет цвет

</script>

</body>

**РАЗМЕРЫ И ПРОКРУТКА ЭЛЕМЕНТОВ**

**offsetParent** — «родитель по дереву рендеринга» — ближайшая ячейка таблицы, body для статического позиционирования или ближайший позиционированный элемент для других типов позиционирования.

**offsetLeft/offsetTop** — позиция в пикселях левого верхнего угла блока, относительно его offsetParent.

**offsetWidth/offsetHeight** — «внешняя» ширина/высота блока, включая рамки и скролл

**clientLeft/clientTop** — отступ области содержимого от левого-верхнего угла элемента. Если операционная система располагает вертикальную прокрутку справа, то равны ширинам левой/верхней рамки, если же слева (ОС на иврите, арабском), то clientLeft включает в себя прокрутку.

**clientWidth/clientHeight** — ширина/высота содержимого вместе с полями padding, но без border и полосы прокрутки. Ширина\высота видимого окна. **clientWidth** использовать предпочтительнее, чем getComputedStyle

**scrollWidth/scrollHeight** — ширина/высота содержимого, включая прокручиваемую область. Включает в себя padding и не включает полосы прокрутки.

**scrollLeft/scrollTop** — ширина/высота прокрученной части документа, считается от верхнего левого угла.

Все свойства, доступны только для чтения, кроме **scrollLeft/scrollTop**. Изменение этих свойств заставляет браузер прокручивать элемент.

Пример: Свойство **elem.scrollTop** содержит размер прокрученной области при отсчете сверху. А как подсчитать размер прокрутки снизу? Решение:

elem.scrollHeight - elem.scrollTop - elem.clientHeight

Узнать ширину скроллбара:

var scrollbarWidth = div.offsetWidth - div.clientWidth;

Центрировать элемент ball внутри элмента field:

ball.style.marginLeft = ((field.clientWidth - ball.offsetWidth) / 2) + "px";

**РАЗМЕРЫ И ПРОКРУТКА СТРАНИЦЫ**

Те прокрутки, которые применимы к элементам, на страницах могут глючить – особенности браузеров

Надёжно **определить размер страницы с учетом прокрутки можно**, взяв максимум из нескольких свойств:

var scrollHeight = Math.max(

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

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

document.body.clientHeight, document.documentElement.clientHeight

);

alert( 'Высота с учетом прокрутки: ' + scrollHeight );

**window.pageXOffset \ window.pageYOffset** читаются, что бы получить область текущей прокрутки:

alert( 'Текущая прокрутка сверху: ' + window.pageYOffset );

alert( 'Текущая прокрутка слева: ' + window.pageXOffset );

**scrollBy(x,y)** прокручивает страницу относительно текущих координат.

Например, кнопка ниже прокрутит страницу на 10px вниз:

window.scrollBy(0,10) //прокрутить страницу на 10px вниз

**scrollTo(pageX,pageY)** прокручивает страницу к указанным координатам относительно документа.

Он эквивалентен установке свойств scrollLeft/scrollTop.

Чтобы прокрутить в начало документа, достаточно указать координаты (0,0).

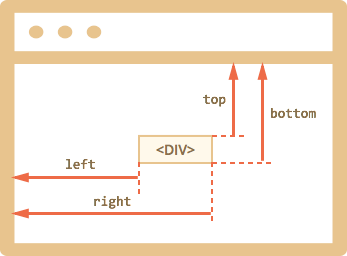
window.scrollTo(0,0)

**elem.scrollIntoView(top)** вызывается на элементе и прокручивает страницу так, чтобы элемент оказался вверху, если параметр top равен true, и внизу, если **top** равен **false**. Причем, если параметр top не указан, то он считается равным true.

this.scrollIntoView(false) //элемент окаэется внизу странички

**document.body.style.overflow = "hidden"**  запретить прокрутку страницы

**КООРДИНАТЫ В ОКНЕ**

**getBoundingClientRect()** возвращает координаты элемента, под которыми понимаются размеры «воображаемого прямоугольника», который охватывает весь элемент.Координаты относительно окна не учитывают прокрутку, они высчитываются от границ текущей видимой области (т.е. независимо от того, прокручен элемент или нет)

Возвращается обьект со свойствами:

top — Y-координата верхней границы элемента,

left — X-координата левой границы,

right — X-координата правой границы,

bottom — Y-координата нижней границы.

Если внутри блока есть строки(inline-элементы), они тоже образают прямоугольники произвольной длины.

Все эти прямоугольники можно получить с помощью **elem.getClientRects().** А метод **elem.getBoundingClientRect()** возвращает один охватывающий прямоугольник для всех **getClientRects().**

**elementFromPoint(x, y)** возвращает элемент, который находится на координатах (x, y) относительно окна. Код ниже ниже выделяет и выводит тег у элемента, который сейчас в середине окна:

var centerX = document.documentElement.clientWidth / 2;

var centerY = document.documentElement.clientHeight / 2;

var elem = document.elementFromPoint(centerX, centerY);

elem.style.background = "red";

Для координат вне окна elementFromPoint возвращает null

**КООРДИНАТЫ В ДОКУМЕНТЕ**

Если провести аналогию с CSS, то **координаты относительно окна** — это **position:fixed,** а **относительно документа** — **position:absolute** (при позиционировании вне других элементов, естественно).

К сожалению, готовой функции для получения координат элемента относительно страницы нет. Но её можно легко написать самим.

Эти две системы координат жёстко связаны: pageY = clientY + текущая вертикальная прокрутка **(pageYOffset/pageXOffset)**

function getCoords(elem) { // кроме IE8-

var box = elem.getBoundingClientRect();

return {

top: box.top + pageYOffset,

left: box.left + pageXOffset //координаты положения элемента + прокрутка

};

}

**screenX/screenY** – координаты на экране. Размеры экрана хранятся в глобальной переменной screen:

alert( screen.width + ' x ' + screen.height ); // общая ширина/высота

// доступная ширина/высота (за вычетом таскбара и т.п.)

alert( screen.availWidth + ' x ' + screen.availHeight );

Координаты левого-верхнего угла браузера на экране хранятся в **window.screenX, window.screenY** (не поддерживаются IE8-):

alert( "Браузер находится на " + window.screenX + "," + window.screenY );

**БРАУЗЕРНЫЕ СОБЫТИЯ**

**Использование атрибута HTML**

Обработчик может быть назначен прямо в разметке, в атрибуте, который называется on<событие>.

<button onclick="alert(this.innerHTML)">Нажми меня</button>

Если использоваться **this** - ссылается на элемент, на котором он сработал

**Использование свойства DOM-объекта**

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

<input id="elem" type="button" value="Нажми меня" />

<script>

elem.onclick = function() {

alert( 'Спасибо' );

};

</script>

Так как DOM-свойство onclick, в итоге, одно, то назначить более одного обработчика так нельзя - он будет перезаписан

Убрать обработчик: **elem.onclick = null**

**addEventListener**

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

event **-** Имя события, например click

handler **-** Ссылка на функцию, которую надо поставить обработчиком.

phase - Необязательный аргумент, «фаза», на которой обработчик должен сработать. Пример:

<script> //пример – достать и скрыть кнопку при нажатии на нее

var button = document.querySelector("button"); //достаем кнопку

function hide(){

button.hidden = true; //при вызове функции скрываем

}

/\*назначаем прослушку. Функция заходит без скобок, т.к.у нее нету return, будет undefined если она вызовется /\*

button.addEventListener("click", hide); событие – не onclick! И в кавычках

</script>

**removeEventListener –** для удаления нужно передать именно ту функцию-обработчик которая была назначена.

button.removeEventListener("click", hide);

Если функцию не сохранить где-либо, а просто передать в **addEventListener**, то потом получить её обратно, чтобы снять обработчик, будет невозможно.

**addEventListener** позволяет добавлять несколько обработчиков на одно событие одного элемента. Так же, есть события, которые будут работать только через addEventListener, например событие **transitionend**

**ПОРЯДОК ОБРАБОТКИ СОБЫТИЙ**

* JavaScript выполняется в едином потоке. Современные браузеры позволяют порождать подпроцессы **Web Workers,** они выполняются параллельно и могут отправлять/принимать сообщения, но не имеют доступа к DOM.
* Обычно события становятся в очередь и обрабатываются в порядке поступления, **асинхронно**, независимо друг от друга.
* **Синхронными** являются вложенные события, инициированные из кода.
* Чтобы сделать событие гарантированно асинхронным, используется вызов через **setTimeout(func, 0).** Отложенный вызов через **setTimeout(func, 0)** используется не только в событиях, а вообще — всегда, когда мы хотим, чтобы некая функция func сработала после того, как текущий скрипт завершится.

**Обьект события**

Детали произошедшего браузер записывает в «объект события», который передаётся первым аргументом в обработчик.

**event.type** - Тип события, в данном случае click

**event.currentTarget** - Элемент, на котором сработал обработчик. Значение — в точности такое же, как и у this, но бывают ситуации, когда обработчик является методом объекта и его this при помощи bind привязан к этому объекту, тогда мы можем использовать event.currentTarget.

**event.clientX / event.clientY** - Координаты курсора в момент клика (относительно окна)

<input type="button" value="Нажми меня" id="elem">

<script>

elem.onclick = function(event) {

// вывести тип события, элемент и координаты клика

alert(event.type + " на " + event.currentTarget);

alert(event.clientX + ":" + event.clientY);

}

</script>

**ВСПЛЫТИЕ И ПЕРЕХВАТ**

При наступлении события обработчики сначала срабатывают на самом вложенном элементе, затем на его родителе, затем выше и так далее, вверх по цепочке вложенности.

<form onclick="alert('form')">FORM

<div onclick="alert('div')">DIV

<p onclick="alert('p')">P</p>

<!—при клике по <P> сработают поочередно все верхние onclick--->

</div>

</form>

Самый глубокий элемент, который вызывает событие, называется «целевым» или «исходным» элементом и доступен как **event.target.**

Отличия от this (=event.currentTarget):

* event.target — это исходный элемент, на котором произошло событие, в процессе всплытия он неизменен.
* this — это текущий элемент, до которого дошло всплытие, на нём сейчас выполняется обработчик.
* Например, если стоит только один обработчик form.onclick, то он «поймает» все клики внутри формы. Где бы ни был клик внутри — он всплывёт до элемента <form>, на котором сработает обработчик.
* this (=event.currentTarget) всегда будет сама форма, так как обработчик сработал на ней.
* event.target будет содержать ссылку на конкретный элемент внутри формы, самый вложенный, на котором произошёл клик.

**Прекращение всплытия**

**event.stopPropagation()** – для прекращения всплытия. Например, здесь при клике на кнопку обработчик body.onclick не сработает:

<body onclick="alert('сюда обработка не дойдёт')">

<button onclick="event.stopPropagation()">Кликни меня</button>

</body>

**event.stopImmediatePropagation()** - не только предотвращает всплытие, но и останавливает обработку событий на текущем элементе.

Убийство всплытия кликов нежелательно, т.к. может влиять на некоторые другие инструменты, напрмер Яндекс-метрика по подсчету кликов не будет работать над областью, где клики «убиты»

**Погружение**

Погружение тоже самое, что и всплытие, но в обратном порядке. Т.е. когда вешаем обработчик на дочерний элемент, погружение перехватит сначала сигнал на родительском, а уже потом на дочернем

* Событие сначала идет сверху вниз. Эта стадия называется «стадия перехвата» (**capturing stage**).
* Событие достигло целевого элемента. Это — «стадия цели» (**target stage**).
* После этого событие начинает всплывать. Это — «стадия всплытия» (**bubbling stage**).

Чтобы поймать событие на стадии перехвата, нужно использовать третий аргумент **addEventListener:**

Если аргумент true, то событие будет перехвачено по дороге вниз.

Если аргумент false, то событие будет поймано при всплытии.

**ДЕЛЕГИРОВАНИЕ СОБЫТИЙ**

Что бы не вешать много обработчиков на каждый обьект, можно повесить один обработчик на их общего предка. Из него можно получить целевой элемент **event.target**, понять на каком именно потомке произошло событие и обработать его.

Например у нас есть таблица с множеством ячеек. Мы хотим подсветить ячейку, на которй произошел клик, другим цветом:

table.onclick = function(event) {

var target = event.target;

// цикл двигается вверх от target к родителям до table

while (target != table) {

if (target.tagName == 'TD') {

// нашли элемент, который нас интересует!

highlight(target);

return;

}

target = target.parentNode;

}

// возможна ситуация, когда клик был вне <td>

// если цикл дошёл до table и ничего не нашёл,

// то обработчик просто заканчивает работу

}

**ПРИЕМ ПРОЕКТИРОВАНИЯ «ПОВЕДЕНИЕ»**

Шаблон проектирования «поведение» (behavior) позволяет задавать хитрые обработчики на элементе декларативно, установкой специальных HTML-атрибутов и классов.

**ПРИМЕР 1** скрывает и показывает форму input при клике по кнопке

<button data-toggle-id="subscribe-mail">

Показать форму подписки

</button>

<form id="subscribe-mail" hidden>

Ваша почта: <input type="email">

</form>

<script>

document.onclick = function(event) {

var target = event.target;

var id = target.getAttribute('data-toggle-id');

if (!id) return;

var elem = document.getElementById(id);

elem.hidden = !elem.hidden;

};

</script>

**ПРИМЕР 2.** Нам нужно сделать меню с разными кнопками: «Сохранить», «Загрузить», «Поиск» и т.д. И есть объект с соответствующими методами: save, load, search и т.п…

Добавляем один обработчик на всё меню, а для каждой кнопки в специальном атрибуте, который мы назовем data-action (можно придумать любое название, но data-\* является валидным в HTML5), укажем, что она должна вызывать

<div id="menu">

<button data-action="save">Сохранить</button>

<button data-action="load">Загрузить</button>

<button data-action="search">Поиск</button>

</div>

<script>

function Menu(elem) {

this.save = function() {

alert( 'сохраняю' );

};

this.load = function() {

alert( 'загружаю' );

};

this.search = function() {

alert( 'ищу' );

};

var self = this;

/\*чтобы сохранить ссылку на объект Menu. Иначе обработчик просто бы не смог вызвать методы Menu, потому что его собственный this ссылается на элемент.\*/

elem.onclick = function(e) {

var target = e.target;

var action = target.getAttribute('data-action');

if (action) {

self[action]();

}

};

}

new Menu(menu);

</script>

**ДЕЙСТВИЯ БРАУЗЕРА ПО УМОЛЧАНИЮ**

**Отмена действий браузера**

Есть два способа отменить действие браузера:

Основной способ — это воспользоваться объектом события. Для отмены действия браузера существует стандартный метод **event.preventDefault().**

Если же обработчик назначен через onсобытие (не через addEventListener), то можно просто вернуть **false** из обработчика.

В следующем примере при клике по ссылке переход не произойдет:

<a href="/" onclick="return false">Нажми здесь</a>

<a href="/" onclick="event.preventDefault()">здесь</a>

Пример с отменой перехода по клику на <a href=”…”> в меню:

menu.onclick = function(event) {

if (event.target.nodeName != 'A') return;

var href = event.target.getAttribute('href');

alert( href ); // может быть подгрузка с сервера, генерация интерфейса и т.п.

return false; // отменить переход по url

};

**ГЕНЕРАЦИЯ СОБЫТИЙ НА ЭЛЕМЕНТАХ**

**В 98% случаев, когда разработчик начинающего или среднего уровня хочет сгенерировать встроенное событие — это вызвано «кривой» архитектурой кода, и взаимодействие нужно на уровне выше.**

События можно генерировать вручную. Спосбо для IE11+

**var event = new Event(тип события[, флаги]);**

Тип события — может быть как своим, так и встроенным, к примеру "click".

Флаги — объект вида{ bubbles: true/false, cancelable: true/false **},** где свойство bubbles указывает, всплывает ли событие, а cancelable — можно ли отменить действие по умолчанию.

Флаги по умолчанию: {bubbles: false, cancelable: false}

**elem.dispatchEvent(event)** **—** запускается, чтобы инициировать событие

<button id="elem" onclick="alert('Клик');">Автоклик</button>

<script>

var event = new Event("click");

elem.dispatchEvent(event);

</script>

На сгенерированном событии, как и на встроенном браузерном, обработчик может вызвать метод event.preventDefault(). Тогда dispatchEvent возвратит false.

**event.isTrusted** **—** единственный способ, которым код может отличить реальное нажатие от программного, является проверка его свойства. Если true - посетитель кликнул сам, false — если событие инициировал скрипт.

**При создании события браузер автоматически ставит следующие свойства:**

isTrusted: false — означает, что событие сгенерировано скриптом, это свойство изменить невозможно.

target: null — это свойство ставится автоматически позже при dispatchEvent.

type: тип события — первый аргумент new Event.

bubbles, cancelable — по второму аргументу new

Другие свойства события, если они нужны, например координаты для события мыши — можно присвоить в объект события позже, например:

var event = new Event("click", {bubbles: true, cancelable: false});

event.clientX = 100;

event.clientY = 100;

**Конструкторы MouseEvent, KeyboardEvent и другие**

Вот список конструкторов для различных событий интерфейса

* UIEvent
* FocusEvent
* MouseEvent
* WheelEvent
* KeyboardEvent
* CompositionEvent

Специфический конструктор позволяет указать стандартные свойства для данного типа события.Например, **clientX/clientY** для события мыши:

var e = new MouseEvent("click", {

bubbles: true,

cancelable: true,

clientX: 100,

clientY: 100

});

alert( e.clientX ); // 100

**CustomEvent -** специфический конструктор для генерации своих, нестандартных, событий. У второго аргумента-объекта есть дополнительное свойство **detail**, в котором можно указывать информацию для передачи в событие.

<h1 id="elem">Привет для Васи!</h1>

<script>

elem.addEventListener("hello", function(event) {

alert( event.detail.name );

}, false);

var event = new CustomEvent("hello", {

detail: { name: "Вася" }

});

elem.dispatchEvent(event);

</script>

**Для IE9-IE11 используется немного другой подход:**

//эти два подхода абсолютно идентичны

var event = new Event("click", { // современный стандарт

bubbles: true,

cancelable: true

});

var event = document.createEvent("Event"); // старый стандарт

event.initEvent("click", true, true);