



DOM



АЛЕКСЕЙ ДАЦКОВ



АЛЕКСЕЙ ДАЦКОВ

Руководитель разработки проекта в Берито



alex.smap@gmail.com



[alex.smap](https://t.me/alex.smap)



[alex.smap](https://vk.com/alex.smap)



ПЛАН ЗАНЯТИЯ




1. [Browser Environment](#)
2. [Обработка документа](#)
3. [BOM vs. DOM](#)
4. [Интерфейсы](#)
5. [Traversing и Manipulation](#)
6. [Базовые селекторы](#)
7. [Атрибуты](#)

ЗАДАЧА

Мы делаем интернет-портал и перед нами поставили задачу организовать автоматически обновляющийся блок последних новостей.

За примерами далеко ходить не нужно, один из таких блоков есть на главной странице Яндекса:

Сейчас в СМИ

-  Володин раскритиковал выступление Орешкина в Госдуме
-  BBC Индии опровергли сообщения о сбитом Пакистаном Су-30
-  В Совфеде оценили слова владимирской чиновницы о питании школьников

Но наш блок будет функционировать немного иначе: самая свежая новость должна появляться сверху, вытесняя самую старую.

HTML

В качестве формата, позволяющего отображать новости (да и весь портал) мы будем использовать язык HTML.

```
1  <!DOCTYPE html>
2  <html lang="en">
3  <head>
4    <title>News Portal</title>
5  </head>
6  <body>
7    <section>
8      <h2>Latest News</h2>
9      <!-- news list -->
10     <ul>
11       <li><a href="...">Third</a></li>
12       <li><a href="...">Second</a></li>
13       <li><a href="...">First</a></li>
14     </ul>
15   </section>
16 </body>
17 </html>
```

РЕАЛИЗАЦИЯ

Добавление новой новости сводится к тому, что мы в список `ul` добавляем новый `li` на первую позицию, а с последней позиции элемент `li` удаляем.

EMMET

Настоятельно рекомендуем вам изучить инструмент, который называется [Emmet](#). Он позволит не только быстро создавать HTML, но так же достаточно быстро запомнить синтаксис CSS-селекторов, который пригодится вам на протяжении всего курса.

Большинство редакторов кода (VS Code) в том числе, поддерживают Emmet *из коробки*.

В частности, базовую структуру документа можно быстро создать, набрав символ `!` и затем `Tab`. А дальше уже дополнять документ.



BROWSER ENVIRONMENT

ЗАГРУЗКА ДОКУМЕНТА БРАУЗЕРОМ

Когда пользователь вводит адрес определённого документа, браузер загружает его с использованием протокола HTTP, обрабатывает его и предоставляет нам API для взаимодействия с документом, другими объектами и окружение для исполнения нашего кода.

Нас пока не интересует, как браузер загружает документ и по каким протоколам, нас интересует лишь то, как производится обработка документа и как нам получать доступ к документу при помощи JavaScript.

ECMAScript и браузер

В стандарте ECMAScript сказано:



ECMAScript is an object-oriented programming language for performing computations and manipulating computational objects within a host environment.

A web browser provides an ECMAScript host environment for client-side computation including, for instance, objects that represent windows, menus, pop-ups, dialog boxes, text areas, anchors, frames, history, cookies, and input/output.

<https://www.ecma-international.org/ecma-262/6.0/>

Т.е. браузер предоставляет host environment и объекты для использования в JS.

ПЕРЕВОД



ECMAScript это объектно-ориентированный язык программирования для выполнения вычислений и обработки вычислительных объектов в среде размещения (host environment).

Веб-браузер предоставляет host environment ECMAScript для расчетов на клиентской стороне. Например, меню, всплывающих окон, диалоговых окон, текстовых областей, ссылок-якорей, рамок, историй, cookies и ввода/вывода.

[Оригинал](#)



БРАУЗЕРЫ

Поскольку браузеров много, должны существовать какие-то общие правила обработки документов и предоставления доступа к ним. Тёмные времена, когда у каждого браузера были свои правила, к нашей радости, прошли.



СТАНДАРТ HTML

Для обработки HTML на просторах интернета можно найти целых два описания:

- [HTML Living Standard от WHATWG](#)
- [HTML 5.2 W3C Recommendation](#)

Возникает резонный вопрос, почему спецификации две? И которой из них верить?

W3C VS WHATWG

Взаимоотношения между W3C и WHATWG — долгая история. Связана она с зарождением HTML5 и разногласиями по поводу того, как должен выглядеть основной язык Web. Детальнее об этом вы можете узнать из книжек, выпущенных в 2010-2012 годах.

Для нас же ключевое следующее — именно в стандартах описаны ключевые вещи, которые позволяют понять, что вы можете делать в рамках API, предоставляемого вам браузером для работы с документом из JavaScript.

Нужно помнить, что стандарты W3C и WHATWG отличаются, но не слишком принципиально.

W3C VS WHATWG

Какой же из стандартов использовать в работе?

Оба + ещё подсматривать в caniuse.com, чтобы видеть уровень поддержки.

ЗАГРУЗКА ДОКУМЕНТА

Поскольку HTML это обычный текстовый файл, браузер занимается его обработкой в следующем формате*:

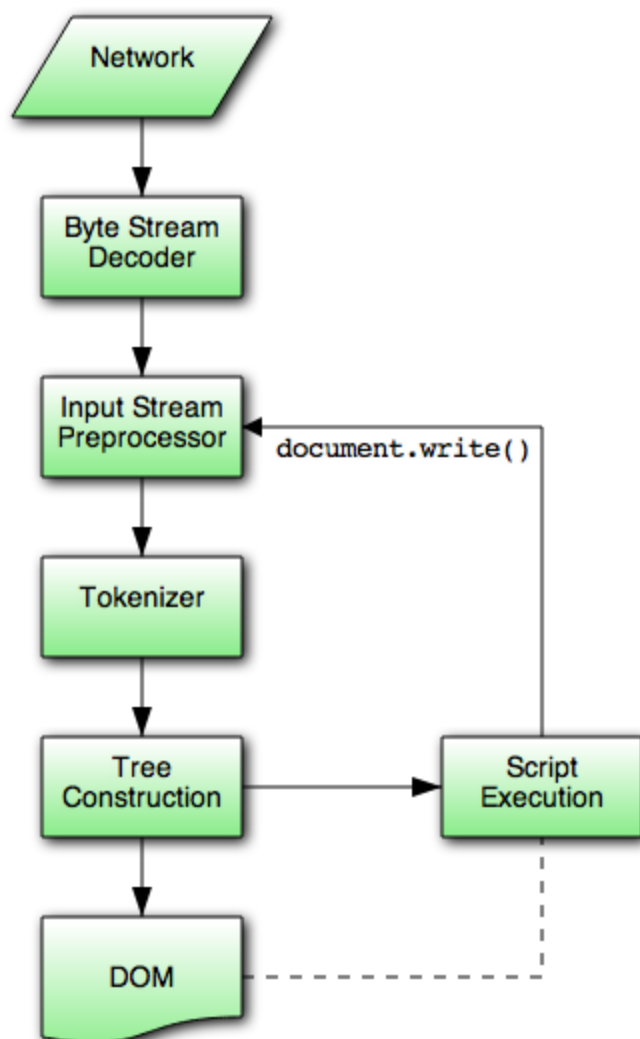
- для каждой открытой вкладки создаётся объект `Window`
- для каждого загруженного документа создаётся объект `Document`, который формируется при обработке текста HTML-страницы

* Мы рассматриваем упрощённый вариант, не учитывающий многих деталей.



ОБРАБОТКА ДОКУМЕНТА

ОБРАБОТКА ДОКУМЕНТА



В упрощённом виде можно считать, что браузер **последовательно** (сверху вниз) читает теги и создаёт для них объекты в памяти. И мы можем из JavaScript взаимодействовать с этими объектами.

Детально процесс загрузки описан в секции «Loading Web pages»:

- [W3C](#)
- [WHATWG](#)

ЗАЧЕМ СТОЛЬКО ТЕОРИИ?

Зачем мне эта теория, где же код?

Теория нужна, чтобы вы понимали, где искать информацию, которой нет в статьях и на StackOverflow.



BOM VS DOM



BOM

BOM (**B**rowser **O**bject **M**odel) — API, предоставляемое веб-браузером для взаимодействия с веб-браузером.



DOM

DOM (**D**ocument **O**bject **M**odel) — API, предоставляемое веб-браузером для взаимодействия с загруженным в память документом.

DOM

В данной лекции нас будет интересовать по большей части только [DOM](#).

Примечание: в этой лекции мы не будем использовать шаблон Webpack, а ограничимся лишь live-server. Напоминаем, что для запуска live-server в нужном каталоге достаточно выполнить команду `npx live-server`.

ИНТЕРФЕЙС Document

В рамках DOM нам предоставляется объект `document`, соответствующий интерфейсу `Document`:

```
1  [Constructor, Exposed=Window]
2  interface Document : Node {
3      [SameObject] readonly attribute DOMImplementation implementation;
4      readonly attribute USVString URL;
5      readonly attribute USVString documentURI;
6      readonly attribute USVString origin;
7      readonly attribute DOMString compatMode;
8      readonly attribute DOMString characterSet;
9      readonly attribute DOMString charset; // historical alias of .characterSet
10     readonly attribute DOMString inputEncoding; // historical alias of .characterSet
11     readonly attribute DOMString contentType;
12
13     readonly attribute DocumentType? doctype;
14     readonly attribute Element? documentElement;
15     HTMLCollection getElementsByTagName(DOMString qualifiedName);
16     HTMLCollection getElementsByTagNameNS(DOMString? namespace, DOMString localName);
17     HTMLCollection getElementsByClassName(DOMString classNames);
```




ИНТЕРФЕЙСЫ



ИНТЕРФЕЙСЫ ЭТО...

В рамках спецификаций W3C и WHATWG интерфейс — это некий контракт, гарантирующий наличие определённых свойств и методов у объектов, соответствующих данному интерфейсу.

ПО СПЕЦИФИКАЦИИ



Большинство API, определенные в этой спецификации, являются интерфейсами, а не классами. Это означает, что фактической реализации нужно только предоставлять методы с определенными именами и указанную операцию, не создавая классы, которые напрямую связаны с интерфейсами.

НАСЛЕДОВАНИЕ ИНТЕРФЕЙСОВ

Наследование интерфейсов обозначается через символ `::`:

```
1 interface Document : Node {  
2     ...  
3 }
```

В JS интерфейсов пока нет, но соответствие объектов, предоставляемых браузером, интерфейсам в целом* поддерживается.

document





К предоставляемому объекту документа мы можем обращаться по имени `document`.

Давайте посмотрим, что он из себя представляет (F12 в консоли браузера):

```
console.dir(document);
```

ЦЕПОЧКА ПРОТОТИПОВ

Если пройти по цепочке прототипов, то мы увидим следующее:

- `HTMLDocument` 
- `Document` 
- `Node` 
- `EventTarget` 
- `Object`

`Object` вы уже знаете, с `EventTarget` мы будем разбираться на следующей лекции, а вот с остальными познакомимся сегодня.

GLOBAL OBJECT

В рамках BOM браузер предоставляет нам глобальный объект `window`. Большинство остальных объектов, предоставляемых браузером являются свойствами этого глобального объекта. В том числе и `document`:

```
document === window.document; // true
```

ПОСЛЕДНЯЯ ИНСТАНЦИЯ

Любое имя сначала ищется в текущей области видимости, затем в вышестоящей и в самую последнюю очередь — в глобальном объекте (в рамках браузера — `window`).

globalThis

В стандарте ES10 планируют ввести `globalThis` — имя, которое вне зависимости от окружения (Node.js, браузер) будет указывать на глобальный объект.

IMAGES & LINKS

Чтобы добраться до изображений и ссылок, можно воспользоваться свойствами `images` и `links` объекта `document`:

```
console.log(document.images);  
console.log(document.links);
```

HTMLImageElement

Причём изображения будут представлены типом `HTMLImageElement`.






Цепочка прототипов:

- `HTMLImageElement` 
- `HTMLElement` 
- `Element` 
- `Node` 
- `EventTarget` 
- `Object`

HTMLAnchorElement

А ссылки будут представлены типом `HTMLAnchorElement`.

Цепочка прототипов:

- `HTMLAnchorElement` 
- `HTMLElement` 
- `Element` 
- `Node` 
- `EventTarget` 
- `Object`



ДВА ВОПРОСА

1. Откуда всё берётся?
2. Как добраться до других элементов?

DOM TREE

Документ после обработки браузером превращается в древовидную структуру: DOM Tree или DOM-дерево. Эта структура состоит из объектов, соответствующих интерфейсу `Node`.

ДЕРЕВО НАШЕГО ДОКУМЕНТА

В частности, наш документ после обработки будет выглядеть вот так:

```
└─ DOCTYPE: html
  └─ HTML lang="en"
    └─ HEAD
      └─ #text:
        └─ TITLE
          └─ #text: News Portal
        └─ #text:
      └─ #text:
    └─ BODY
      └─ #text:
      └─ SECTION
        └─ #text:
        └─ H2
          └─ #text: Latest News
        └─ #text:
        └─ #comment: news list
        └─ #text:
        └─ UL
          └─ #text:
          └─ LI
            └─ #text:
            └─ A href="..."
              └─ IMG src="..."
                └─ #text: Third
            └─ #text:
          └─ #text:
        └─ #text:
      └─ #text:
```

ОБЪЕКТЫ ДЛЯ ВСЕГО

Но подождите, после тега `<head>` нет никакого текста. На самом деле там есть символ переноса строки `↵` и он тоже преобразуется в отдельный объект в дереве.

То же самое относится и к `DOCTYPE`, и к комментариям — для них создаются отдельные объекты в дереве.



СЕМЕЙНЫЕ ОТНОШЕНИЯ

Все Node, объединённые в DOM-дерево, состоят в родительско-дочерних отношениях (+sibling или соседские отношения) с корневым объектом в этом дереве (`root`).

ДОСТУП К РОДСТВЕННИКАМ

Интерфейс Node нам предоставляет свойства для доступа к родителям, детям и соседям:

```
readonly attribute Node? parentNode; // родительская Node
[SameObject] readonly attribute NodeList childNodes; // список дочерних Node
readonly attribute Node? firstChild; // первая дочерняя Node
readonly attribute Node? lastChild; // последняя дочерняя Node
readonly attribute Node? previousSibling; // предыдущая соседняя Node
readonly attribute Node? nextSibling; // следующая соседняя Node
```

ВАЖНО

- `readonly` означает, что вы не можете изменить значение свойства с помощью присваивания.
- `?` означает, что значением свойства может быть `null`.

МЕТОДЫ ИЗМЕНЕНИЯ

Кроме того, определяется ряд методов изменения DOM-дерева:

```
[CEReactions] Node insertBefore(Node node, Node? child);  
[CEReactions] Node appendChild(Node node);  
[CEReactions] Node replaceChild(Node node, Node child);  
[CEReactions] Node removeChild(Node child);
```

ДОСТУП К ЭЛЕМЕНТАМ

Мы можем добраться до нашего списка новостей следующим образом:

```
1  const ulNode = document
2    .childNodes[1]
3    .childNodes[2]
4    .childNodes[1]
5    .childNodes[3];
```

КЛЮЧЕВЫЕ МОМЕНТЫ:

- Нас в принципе не особо интересуют ноды с комментариями, `DOCTYPE` и символами переноса строки.
- Очень неудобно и чревато ошибками.
- Вполне может перестать работать, если мы будем использовать минимайзеры HTML (в рамках того же Webpack).

ПОРЯДОК СОЗДАНИЯ ОБЪЕКТОВ

Очень важно подключать скрипт перед `</body>` - это позволит быть уверенным, что вся структура до скрипта уже обработана и объекты для неё созданы.

Посмотрите, что будет, если код, указанный на предыдущем слайде подключить сразу после `<body>`.

NODE TYPES

В рамках DOM есть несколько различных типов `Node` :

```
1  const unsigned short ELEMENT_NODE = 1;
2  const unsigned short ATTRIBUTE_NODE = 2;
3  const unsigned short TEXT_NODE = 3;
4  const unsigned short CDATA_SECTION_NODE = 4;
5  const unsigned short ENTITY_REFERENCE_NODE = 5; // historical
6  const unsigned short ENTITY_NODE = 6; // historical
7  const unsigned short PROCESSING_INSTRUCTION_NODE = 7;
8  const unsigned short COMMENT_NODE = 8;
9  const unsigned short DOCUMENT_NODE = 9;
10 const unsigned short DOCUMENT_TYPE_NODE = 10;
11 const unsigned short DOCUMENT_FRAGMENT_NODE = 11;
12 const unsigned short NOTATION_NODE = 12; // historical
```


Element

`Element` — специальный интерфейс, описывающий `Node` с типом `ELEMENT_NODE`.

На базе этого интерфейса уже строятся такие интерфейсы как `HTMLElement` (для HTML-элементов — тегов), `SVGElement` (для SVG-элементов). Кроме того, он содержит ряд методов, позволяющих организовать поиск.

ParentNode

Mixin `ParentNode` содержит свойства и методы, позволяющие получать из родительско-дочерних отношений только элементы:

```
1 interface mixin ParentNode {
2   [SameObject] readonly attribute HTMLCollection children;
3   readonly attribute Element? firstElementChild;
4   readonly attribute Element? lastElementChild;
5   readonly attribute unsigned long childElementCount;
6
7   [CEReactions, Unscopable] void prepend((Node or DOMString)... nodes);
8   [CEReactions, Unscopable] void append((Node or DOMString)... nodes);
9
10  Element? querySelector(DOMString selectors);
11  [NewObject] NodeList querySelectorAll(DOMString selectors);
12 };
13 Document includes ParentNode;
14 DocumentFragment includes ParentNode;
15 Element includes ParentNode;
```

ChildNode

Mixin `ChildNode` содержит методы, позволяющие манипулировать положениями `Node` в DOM-дереве:

```
1 interface mixin ChildNode {  
2     [CEReactions, Unscopable] void before((Node or DOMString)... nodes);  
3     [CEReactions, Unscopable] void after((Node or DOMString)... nodes);  
4     [CEReactions, Unscopable] void replaceWith((Node or DOMString)... nodes);  
5     [CEReactions, Unscopable] void remove();  
6 };  
7 DocumentType includes ChildNode;  
8 Element includes ChildNode;  
9 CharacterData includes ChildNode;
```

ПОДДЕРЖКА В БРАУЗЕРАХ

К сожалению, не все методы, перечисленные в интерфейсах DOM, поддерживаются на должном уровне браузерами. Поэтому мы будем использовать только самые поддерживаемые:

This is an experimental technology

Check the [Browser compatibility table](#) carefully before using this in production.

The `ChildNode.before` method inserts a set of `Node` or `DOMString` objects in the children list of this `ChildNode`'s parent, just before this `ChildNode`. `DOMString` objects are inserted as equivalent `Text` nodes.

Browser compatibility [↗](#)

[↗ Update compatibility data on GitHub](#)

Desktop						Mobile							
Chrome	Edge	Firefox	Internet Explorer	Opera	Safari	Android webview	Chrome for Android	Edge Mobile	Firefox for Android	Opera for Android	Safari on iOS	Samsung Internet	
Basic support													
54	17	49	No	39	No	54	54	No	49	39	No	6.0	
	Full support						No support						

ИТОГО

У нас есть:

1. Интерфейс `Node`, который поддерживает методы манипуляции DOM-деревом.
2. Mixin `ParentNode` (включён в интерфейсы `Document` и `Element`), позволяющий получать доступ только к элементам и содержащий методы `querySelector` и `querySelectorAll`.
3. Интерфейсы `Document` и `Element`, поддерживающие методы `getElementById`, `getElementsByClassName`, `getElementsByTagName`.
4. Интерфейс `Document` содержащий метод `createElement`.



TRAVERSING \mathbb{N} MANIPULATION

УДАЛЕНИЕ САМОЙ СТАРОЙ НОВОСТИ

Решим сначала задачу «вытеснения» самой старой новости. Поскольку метод `remove` у `Node` поддерживается не везде, мы будем использовать `removeChild` у родительской `Node`.

Нам нужно:

1. Найти родительский элемент.
2. Найти «последний» дочерний элемент в этом родителе.
3. Если дочерних элементов 5, удалить последний.

ПОИСК ЭЛЕМЕНТА

Искать элементы мы можем несколькими способами (включая их комбинации):

- через родительско-дочерние отношения;
- `getElementById` – поиск по атрибуту `id` элемента (можем назначить `ul` какой-то идентификатор и таким образом его найти);
- `getElementsByClassName` – поиск по атрибуту `class` элемента;
- `getElementsByTagName` – поиск по имени тега элемента;
- `querySelector` / `querySelectorAll` – поиск по CSS-селектору.

ПОИСК ПО `id`

```
1 <ul id="news">
2   ...
3 </ul>
```

```
1 try {
2   const newsEl = document.getElementById('news');
3   // null, если элемент не найден
4   const newsToRemoveEl = newsEl.lastElementChild;
5   // null, если нет детей
6   if (newsEl.childElementCount === 3) {
7     newsEl.removeChild(newsToRemoveEl);
8   }
9 } catch (e) {
10   // TODO: log & send error somewhere
11 }
```

УЗКИЕ МЕСТА

1. Удалить `Node` может только непосредственный родитель, в противном случае получим `DOMException`!
2. Если передать `null` в `removeChild` получим `TypeError`.

УДАЛЕНИЕ ЭЛЕМЕНТА

1. При удалении элемента удаляются и его дочерние элементы.
2. Элемент перестаёт отображаться на странице (так как удалён из иерархии DOM-дерева).
3. Но если вы сохранили ссылку на объект в памяти, то он останется в памяти, несмотря на то, что его нет в DOM.

ПОИСК ПО class

```
1 <ul class="news">
2   ...
3 </ul>
```

```
1 try {
2   const NEWS_LIMIT = 3;
3   const newsEl = document.getElementsByClassName('news')[0];
4   const newsToRemoveEl = newsEl.lastElementChild; // null, если нет детей
5   if (newsEl.childElementCount === NEWS_LIMIT) {
6     newsEl.removeChild(newsToRemoveEl);
7   }
8 } catch (e) {
9   // TODO: log & send error somewhere
10 }
```

HTMLCollection

`document.getElementsByClassName` возвращает объект типа `HTMLCollection`. Этот объект поддерживает индексацию и итерирование, но работать с обычным массивом JS бывает гораздо удобнее.

ПРЕОБРАЗОВАТЬ В МАССИВ

Вы можете преобразовать `HTMLCollection` в массив с помощью разных вариантов:

- `[...collection]`
- `Array.from(collection)`

ВОПРОС

Что должно быть реализовано в объекте для поддержки итерирования (через `for ... of`)?

LIVE-КОЛЛЕКЦИЯ

`HTMLCollection` является live-коллекцией. Что это значит?

```
1  const collection = document.getElementsByClassName('news');
2  const array = [...collection];
3  const newsEl = collection[0];
4  newsEl.parentElement.removeChild(newsEl);
5
6  console.log(collection);
7  console.log(array);
```


ПОИСК ПО tagName

Работа по имени тега аналогична работе по имени класса (возвращается так же `HTMLCollection`), но использовать его для поиска через `document` неоправдано.

ВАРИАНТЫ

Идеи, предложенные на данном слайде могут быть оспорены, но мы придерживаемся следующих соображений:

1. Искать от `document` 'а через родительско-дочерние отношения — неудобно, так как структура документа может измениться.
2. `getElementById` в простейшем случае будет работать, но накладывает на нас ограничения уникальности идентификатора (мы не сможем на одной странице разместить два «виджета» новостей).
3. `getElementsByTagName` недостаточно специфичен, если использовать через `document`.
4. `getElementsByClassName` прекрасно подходит, но необходимо разделять CSS-классы, используемые для стилизации, и CSS-классы, используемые для поиска объектов в DOM.
5. `querySelector` / `querySelectorAll` — наш выбор вкупе с `data-` атрибутами.

АТРИБУТЫ

Для каждого элемента браузер создаёт объект, соответствующий определённым интерфейсам. Например, для изображений это был `HTMLImageElement`. Набор атрибутов для каждого элемента фиксирован и описан в спецификации:

```
1 interface HTMLImageElement : HTMLElement {
2     attribute DOMString alt;
3     attribute DOMString src;
4     attribute DOMString srcset;
5     attribute DOMString sizes;
6     attribute DOMString? crossOrigin;
7     attribute DOMString useMap;
8     attribute DOMString longDesc;
9     attribute boolean isMap;
10    attribute unsigned long width;
11    attribute unsigned long height;
12    readonly attribute unsigned long naturalWidth;
13    readonly attribute unsigned long naturalHeight;
14    readonly attribute boolean complete;
15    readonly attribute DOMString currentSrc;
16    attribute DOMString referrerPolicy;
17 };
```

data- АТРИБУТЫ

Спецификация HTML предлагает нам возможность создавать собственные атрибуты к html-элементам, которые будут являться валидными при соблюдении следующего условия: имя атрибута должно начинаться с префикса `data-`.

Данный подход широко используется в работе, в том числе при организации автоматизированного тестирования, когда выбираются атрибуты `data-id` и `data-testid`. При этом исключается вероятность пересечения с CSS-классами, используемыми для оформления.

querySelector(All)

Методы `querySelector` / `querySelectorAll` используют спецификацию CSS-селекторов для выбора:

1. `Node` для `querySelector`.
2. `NodeList` для `querySelectorAll`.

`NodeList` как и `HTMLCollection` является live-коллекцией и конвертируется в JS-массив через `spread` или `Array.from`.



БАЗОВЫЕ СЕЛЕКТОРЫ

БАЗОВЫЕ СЕЛЕКТОРЫ

- По id: `#id`
- По тегу: `tagname`
- По классу: `.class`
- По атрибуту:

`[attr]`

`[attr=value]`

`[attr~=value]`

`[attr|=value]`

`[attr^=value]`

`[attr$=value]`

`[attr*=value]`


a + b

a + b – один родитель и **b** следует непосредственно за **a** :

```
1  <parent>
2    <a></a>
3    <b></b>
4  </parent>
```



 $a \sim b$

$a \sim b$ – один родитель и b следует за a , но необязательно непосредственно:

```
1  <parent>
2    <a></a>
3    <c></c>
4    <b></b>
5  </parent>
```

a > b

a > b – **b** является непосредственным ребёнком **a** :

```
1 | <a>
2 |   <b></b>
3 | </a>
```


a b

a b – **b** является ребёнком **a**, но не обязательно непосредственным:

```
1  <a>
2    <c>
3      <b></b>
4    </c>
5  </a>
```



ПОДРОБНЕЕ

Детальнее с селекторами вы можете ознакомиться на странице [MDN](#).

querySelector

```
1 <section data-widget="news-widget">
2   ...
3 </section>
```

```
1 try {
2   const NEWS_LIMIT = 3;
3   const newsEl = document.querySelector('[data-widget=news-widget] ul');
4   // возвращает первый найденный
5   const newsToRemoveEl = newsEl.lastElementChild;
6   // null, если нет детей
7   if (newsEl.childElementCount === NEWS_LIMIT) {
8     newsEl.removeChild(newsToRemoveEl);
9   }
10 } catch (e) {
11   // TODO: log & send error somewhere
12 }
```

Для поддержки нескольких виджетов необходимо будет `querySelector` заменить на `querySelectorAll` и организовать цикл.



ВОПРОСЫ ПРОИЗВОДИТЕЛЬНОСТИ

Поиск по селектору работает гораздо медленнее поиска по идентификатору и классу. Это стоит учесть при оптимизации вашего приложения.

Но стоит помнить и ключевое правило оптимизации: преждевременная оптимизация — корень многих проблем.

ЭМУЛИРУЕМ СЕРВЕР

Итак, мы научились находить нужного нам родителя и удалять «самую старую» новость. Пришло время научиться создавать новую.

Будем эмулировать приход новости с сервера через функцию `setInterval(callback, interval)`, которая запускает callback с определённым интервалом.

СОЗДАНИЕ ЭЛЕМЕНТА

Для создания элемента в интерфейсе `Document` существует метод `createElement`:

```
1  const liEl = document.createElement('li');
2  const linkEl = document.createElement('a');
3  const imgEl = document.createElement('img');
4
5  newsEl.insertBefore(liEl, newsEl.firstChild);
6  liEl.appendChild(linkEl);
7  linkEl.insertBefore(imgEl, linkEl.firstChild);
```

Вопрос: почему мы использовали именно `insertBefore`?



НЕ ВИДЕН БЕЗ РОДИТЕЛЯ

Важно: пока элементу не назначить родителя, он не будет отображаться в дереве DOM.

insertBefore(node, null)

`insertBefore(node, null)` работает аналогично `appendChild(node)`.



АТРИБУТЫ

АТТРИБУТЫ И ТЕКСТ

Элемент добавлен, но в данный момент не отображается ни текст, ни изображение. Когда мы прописывали элементы в HTML, то для задания изображения использовали атрибут `src`, адреса ссылки — `href`, а текст располагали внутри тега:

```
1  const liEl = document.createElement('li');
2  const linkEl = document.createElement('a');
3  const imgEl = document.createElement('img');
4
5  linkEl.href = 'https://netology.ru'; // ссылка
6  linkEl.textContent = 'Стартовал курс по Advanced JS in HTML'; // текст
7  imgEl.src = 'netology.png' // картинка
8
9  newsEl.insertBefore(liEl, newsEl.firstChild);
10 liEl.appendChild(linkEl);
11 linkEl.insertBefore(imgEl, linkEl.firstChild);
```



ОДНОИМЁННЫЕ СВОЙСТВА

Для большинства атрибутов, определённых в стандарте HTML, нам предоставляются одноимённые свойства объектов.

ИСКЛЮЧЕНИЯ

Но есть и исключения. Самые часто встречающиеся:

- `class` — используются свойства `className` и `classList`.
- `data-*` — используются свойство `dataset`.

getAttribute / setAttribute

Важно понимать отличие атрибутов, существующих в HTML, от свойств, существующих в объекте. Так в интерфейсе `Element` определены методы для работы с атрибутами:

- `setAttribute`
- `getAttribute`

СРАВНИМ

Попробуем сравнить значения:

```
console.log(imgEl.src); // полный url  
console.log(imgEl.getAttribute('src')); // netology.png
```


МЕНЯЕТСЯ ВЕЗДЕ

При изменении свойства `src` или атрибута `src`, соответствующие атрибуты / свойства так же обновятся.

Но это утверждение верно не для всех атрибутов и свойств.

СЛИШКОМ СЛОЖНО

А что, если иерархия создаваемого элемента не такая простая и содержит много вложенных элементов с большим количеством свойств?

В таком случае придётся либо писать шаблонизатор, либо использовать `innerHTML`.

innerHTML

Браузер прекрасно справляется с работой, связанной с преобразованием HTML в дерево DOM. Почему бы не воспользоваться этой возможностью?

```
1  const liEl = document.createElement('li');
2  const link = 'https://netology.ru';
3  const img = 'netology.png';
4  const text = 'Стартовал курс по Advanced JS in HTML';
5  liEl.innerHTML = `
6      <li>
7          <a href="${link}"> ${text}</a>
8      </li>
9  `;
10
11 newsEl.insertBefore(liEl, newsEl.firstChild);
```

ВНЕКЛАСНОЕ ЧТЕНИЕ

Методы `cloneNode` и `replaceChild` мы оставим вам на самостоятельное изучение. Вы попрактикуетесь в их использовании в домашних заданиях к следующим лекциям.

Не забывайте, что источниками информации для вас в первую очередь должны служить сайт MDN и спецификации.



РАЗДЕЛЕНИЕ КОДА

Рекомендации по поводу создания тестируемого кода:

1. Разделять код, взаимодействующий с BOM / DOM и не взаимодействующий с BOM / DOM по разным модулям.
2. Не использовать внутри методов свойства глобального объекта, а передавать их в методы / конструкторы в качестве параметров.



СИНХРОНИЗАЦИЯ

Наш скрипт получился достаточно простым, но уже сейчас встаёт вопрос синхронизации: данные хранятся «в памяти» (в виде JS-объектов), а их отображение — в дереве DOM.

И в такой модели нам придётся либо постоянно синхронизировать их друг с другом, либо рассмотреть иной подход. Об этом мы поговорим в следующих лекциях.



ИТОГИ

ИТОГИ

Сегодня мы изучили:

- Существует два стандарта HTML: WATWG и W3C.
- Как браузер загружает и обрабатывает документ.
- Чем отличается BOM от DOM.
- Как вырастить своё DOM-дерево.
- Объект `document` и глобальный объект `window`.
- Как получить доступ к отдельным элементам, их атрибутам и тексту.
- В каких семейных отношениях находятся элементы.
- Основные методы поиска элемента по классу, идентификатору или селектору.

ИТОГИ

- Что такое `HTMLCollection` и почему её называют live-коллекцией.
- Где полезны data-атрибуты.
- Создание элемента при помощи `createElement` и вставка в HTML при помощи `insertBefore(node, null)`.



Задавайте вопросы и напишите отзыв о лекции!

АЛЕКСЕЙ ДАЦКОВ

 alex.smap@gmail.com

 [alex.smap](https://t.me/alex.smap)

 [alex.smap](https://vk.com/alex.smap)

<