

ОБЪЕКТНАЯ МОДЕЛЬ ДОКУМЕНТА



МИХАИЛ КУЗНЕЦОВ / ING



МИХАИЛ КУЗНЕЦОВ

Developer ING



[@mkuznetcov](https://www.telegram.me/mkuznetcov)

ПЛАН ЗАНЯТИЯ

1. [Положение тега `<script>`](#)
2. [Загрузка DOM](#)
3. [События загрузки документа](#)
4. [Асинхронное и отложенное исполнение скриптов](#)
5. [Поиск элемента по CSS селектору](#)
6. [Обзор селекторов](#)
7. [Атрибуты и свойства элемента](#)
8. [Дата-атрибуты](#)
9. [Управление стилями](#)
10. [Содержимое тега](#)

ПОЛОЖЕНИЕ ТЕГА

<script>

ИЩЕМ ССЫЛКИ

Начнем с примера, в котором найдем на странице все теги `<a>` и выведем в консоль их количество:

```
const links = document.getElementsByTagName('a');  
console.log(`Найдено гиперссылок: ${links.length}`);
```

ПОДКЛЮЧАЕМ СКРИПТ

```
1 <a href="http://netology.ru/">Нетология</a>  
2 <a href="https://foxford.ru/">Фоксфорд</a>  
3 <script src="./links.js"></script>
```

РЕЗУЛЬТАТ ПОИСКА

Открыв консоль, мы увидим результат работы скрипта:

Найдено гиперссылок: 2

ВТОРОЙ ВАРИАНТ ПОДКЛЮЧЕНИЯ СКРИПТА

Изменим положение тега `<script>`, поставив его между ссылками:

```
1 <a href="http://netology.ru/">Нетология</a>  
2 <script src="./links.js"></script>  
3 <a href="https://foxford.ru/">Фоксфорд</a>
```


СКОЛЬКО ССЫЛОК НАЙДЕНО?

Результат в консоли:

Найдено гиперссылок: 1

Результат изменился. Теперь функция `getElementsByTagName` находит только один тег `<a>` в документе.

ОПРЕДЕЛЯЕМ ССЫЛКУ

Выясним, какой из тегов `<a>` находит скрипт, добавив вывод в консоль атрибута `href` ссылки:

```
1  const links = document.getElementsByTagName('a');
2  console.log(`Найдено гиперссылок: ${links.length}`);
3  for (const link of links) {
4      console.log(link.href);
5  }
```

РЕЗУЛЬТАТ ВЫВОДА

Заглянем в консоль и убедимся, что скрипт находит только первую ссылку:

```
http://netology.ru/
```

ТРЕТИЙ ВАРИАНТ ПОДКЛЮЧЕНИЯ

Теперь поместим тег `<script>` в самую первую строчку:

```
1 <script src="./links.js"></script>
2 <a href="http://netology.ru/">Нетология</a>
3 <a href="https://foxford.ru/">Фоксфорд</a>
```



ДРУГОЙ РЕЗУЛЬТАТ

Результат:

Найдено гиперссылок: 0

МЕСТО ПОДКЛЮЧЕНИЯ СКРИПТОВ

Теперь можно с уверенностью сказать, что мы можем найти только те теги в документе, которые расположены до тега `<script>`. Именно поэтому мы подключаем наш скрипт всегда в самом конце документа.

Почему так происходит?



ЗАГРУЗКА DOM

РАБОТА БРАУЗЕРА

Браузер формирует объектную модель документ не мгновенно. На создание объектов для каждого HTML-тега уходит некоторое время. Так как `<script>` тоже тег, то по спецификации браузер должен создать объекты для тегов, которые предшествуют ему, прежде, чем начнет выполнять скрипт из этого тега. Поэтому все эти теги будут доступны в DOM из скрипта.

Проще говоря, скрипт видит только DOM, построенный на момент его исполнения.

ПОЛОВИНА ТЕГА

А что, если тегу `<script>` предшествует только половина тега? А точнее, только его открывающая часть, например, `<html>`, `<head>`, `<body>` или `<div>`:

```
1 <div>
2   <div>
3     <script>
4       const divs = document.getElementsByTagName('div');
5       console.log(divs.length);
6     </script>
7   </div>
8 </div>
```



ОРИЕНТИРУЕМСЯ НА ОТКРЫВАЮЩИЙ ТЕГ

Результатом работы скрипта с предыдущего слайда будет 2.

Браузер создает элемент, как только встречает открывающий тег. Такой элемент еще не полностью сформирован, но мы уже можем его найти и, например, добавить обработчик события.

СТОИТ ЛИ ЖДАТЬ?

Может быть стоит немного подождать и продолжить, когда теги появятся?

Попробуем:

```
1  const links = document.getElementsByTagName('a');  
2  while (divs.length === 0);  
3  // !!! НЕ ПОВТОРЯТЬ, ПОВЕСИТ БРАУЗЕР !!!  
4  console.log(`Найдено гиперссылок: ${links.length}`);
```



НЕУДАЧНЫЙ ЭКСПЕРИМЕНТ

Если бы вы попробовали реализовать такой пример, то увидели бы, что ожидание длится вечно.

Всё дело в том, что браузер продолжит строить дерево DOM только после того, как скрипт выполнится. А точнее, его синхронная часть. Поэтому в процессе выполнения синхронного кода скрипта новых тегов в DOM не появится.

СКРИПТ СРАЗУ ПОСЛЕ НУЖНЫХ ТЕГОВ

Получается, если наш код взаимодействует с какими-то тегами, то его необходимо поместить после них?

Это плохое решение. Аргументы против:

1. Расположение элементов на странице или их порядок в коде может поменяться, так они могут оказаться ниже скрипта.
2. Теги `<script>`, расставленные внутри разметки, очень легко случайно удалить.
3. Ресурсы (стили, скрипты) лучше подключать в одном месте, чтобы было видно зависимости.
4. Пока выполняется скрипт, браузер не строит DOM дальше.

В КОНЕЦ ДОКУМЕНТА

Получается что мы вынуждены помещать скрипты в конец документа перед закрывающим тегом `<body>` ?

На самом деле, есть и другие варианты.

АСИНХРОННАЯ `setTimeout`

Я отмечал про синхронную часть скрипта. А, например, функция `setTimeout` асинхронная. Поставим таймер и подождем:

```
1 function init() {  
2   const links = document.getElementsByTagName('a');  
3   console.log(`Найдено гиперссылок: ${links.length}`);  
4 }  
5  
6 setTimeout(init, 1000);
```

РЕЗУЛЬТАТ С ТАЙМЕРОМ

Результат в консоли `Найдено гиперссылок: 2.`

Но такой вариант не является надежным или пуленепробиваемым. На самом деле, мы не знаем, сколько времени потребуется браузеру, чтобы загрузить весь документ и подготовить DOM к работе. Может, 1 секунда. А может, у нас EDGE 😄, а страница огромная.



ВЕРНЫЙ ВАРИАНТ

Поэтому самым надежным способом безопасно работать с DOM является работа с ним, когда браузер его полностью сформировал. Браузер может оповестить наш скрипт об этом.

Как? Конечно, с помощью специального события.



СОБЫТІЯ ЗАГРУЗКИ ДОКУМЕНТА

ЦЕЛЫХ ДВА СОБЫТИЯ

Когда браузер полностью сформировал DOM-дерево, генерируется событие `DOMContentLoaded`. Потом, когда браузер загрузит все ресурсы (стили, скрипты, изображения), то он также сгенерирует событие `load`:

- `DOMContentLoaded` — означает, что все DOM-элементы разметки уже созданы, можно их искать, вешать обработчики, создавать интерфейс, но при этом, возможно, ещё не догрузились какие-то картинки или стили.
- `load` — страница и все ресурсы загружены, используется редко, обычно нет нужды ждать этого момента.

СОБЫТИЕ DOMContentLoaded

Событие `DOMContentLoaded` происходит на `document` и поддерживается во всех браузерах, кроме **IE8-**. Обработчик на него вешается через `addEventListener`:

```
1 function init() {  
2     const links = document.getElementsByTagName('a');  
3     console.log(`Найдено гиперссылок: ${links.length}`);  
4 }  
5  
6 document.addEventListener('DOMContentLoaded', init);
```

РЕЗУЛЬТАТ ПОСЛЕ ЗАГРУЗКИ

Результат в консоли `Найдено гиперссылок: 2`. Вот теперь куда ни помести в коде наш скрипт, он дает один и тот же результат.

Обратите внимание, что через свойство `on*` это событие обработать нельзя.

Но это не является проблемой, так как мы договорились все события вешать через `addEventListener`.

РЕШЕНИЕ ДЛЯ IE8-


Про поддержку аналогичного функционала в старых IE:

```
1  if (!document.addEventListener) {  
2      document.onreadystatechange = function () {  
3          if (document.readyState == "interactive") {  
4              init();  
5          }  
6      }  
7  }
```

СОБЫТИЕ `load`

Событие `load` на `window` срабатывает, когда загружается вся страница, включая ресурсы на ней — стили, картинки, ифреймы и т.п:

```
window.addEventListener('load', init);
```



АСИНХРОННОЕ И ОТЛОЖЕННОЕ ИСПОЛНЕНИЕ СКРИПТОВ

ЖДЕМ ЗАГРУЗКИ СКРИПТОВ

Если в документе есть теги `<script>`, то браузер обязан их выполнить до того, как построит DOM. Поэтому событие `DOMContentLoaded` ждёт загрузки и выполнения таких скриптов:

```
1 <script src="./links.js"></script>
2 <a href="http://netology.ru/">Нетология</a>
3 <a href="https://foxford.ru/">Фоксфорд</a>
4 <script>console.log('Теги пока не найдены');</script>
```

Результат:

```
Теги пока не найдены
Найдено гиперссылок: 2
```

ТОРМОЗЯЩИЙ ВНЕШНИЙ СКРИПТ

А что, если на странице подключается скрипт с внешнего ресурса (к примеру, реклама), и он тормозит? Событие `DOMContentLoaded` и связанные с ним действия могут сильно задержаться.

Исключением являются скрипты с атрибутами `async` и `defer`, которые подгружаются асинхронно.

«СИНХРОННОЕ» ПОВЕДЕНИЕ

Браузер загружает и отображает HTML постепенно. Особенно это заметно при медленном интернет-соединении: браузер не ждёт, пока страница загрузится целиком, а показывает ту часть, которую успел загрузить.

Если браузер видит тег `<script>`, то он по стандарту обязан:

- Загрузить файл скрипта (если есть атрибут `src`).
- Выполнить его.
- Показать оставшуюся часть страницы.

Такое поведение называют «синхронным». Как правило, оно вполне нормально, но есть важное следствие.

Если скрипт — синхронный,
то пока браузер его не
выполнит, часть страницы
под ним не будет показана.

РЕШЕНИЕ ПРОБЛЕМЫ

Решить эту проблему помогут атрибуты `async` или `defer`. Оба атрибута никак не влияют на встроенные в HTML скрипты, то есть на те, у которых нет атрибута `src`.

АТРИБУТ `defer`

Атрибут `defer` поддерживается всеми браузерами, включая самые старые IE. Скрипт выполняется асинхронно, но браузер гарантирует, что порядок скриптов с `defer` будет сохранён — они будут выполняться последовательно в том порядке, в котором расположены в документе.

ПОРЯДОК ВЫПОЛНЕНИЯ

Первым выполнится код из файла **first.js**, а **second.js** вторым:

```
<script src="./first.js" defer></script>  
<script src="./second.js" defer></script>
```

Скрипт **second.js**, даже если загрузился раньше, будет ожидать выполнения синхронной части кода из **first.js**.

СВЯЗАННЫЕ СКРИПТЫ

Поэтому атрибут `defer` используют в тех случаях, когда код одного скрипта использует ресурсы другого скрипта. Например если мы подключаем библиотеку, и наш скрипт, который её использует, и хотим их подключить асинхронно, то должны использовать `defer` и файл библиотеки подключить первым:

```
<script src="./lib.js" defer></script>  
<script src="./client.js" defer></script>
```


ОСОБЕННОСТЬ `defer`

Также важной особенностью скрипта, подключенного с атрибутом `defer`, является его исполнение после того, как построено DOM-дерево, но перед событием `DOMContentLoaded`. Это бывает удобно, когда мы в скрипте хотим работать с документом, и должны быть уверены, что он готов.

АТРИБУТ `async`

Атрибут `async` — поддерживается всеми браузерами, кроме **IE9-**. Скрипт выполняется асинхронно. То есть, при обнаружении `<script async src="...">` браузер не останавливает обработку страницы, а спокойно работает дальше. Когда скрипт будет загружен — он выполнится.

ПОРЯДОК ВЫПОЛНЕНИЯ

То есть в таком коде первым сработает тот скрипт, который раньше загрузится:

```
<script src="./first.js" async></script>  
<script src="./second.js" async></script>
```

Иногда это может быть **first.js** иногда **second.js**. Особенно разница может быть ощутима при существенной разнице в размере файлов и если они расположены на разных серверах. Так как порядок выполнения скриптов не гарантирован, нельзя подключать с `async` скрипты, от которых зависят какие-либо другие скрипты.

ТЕСТ НА ПОРЯДОК ВЫПОЛНЕНИЯ СКРИПТОВ

Для тестирования порядка исполнения воспользуемся методами объекта `console`, которые позволяют замерить промежуток времени:

- `time` – принимает название таймера *строка* и запускает его.
- `timeEnd` – принимает название таймера *строка*, останавливает таймер и выводит время.

СОДЕРЖИМОЕ ФАЙЛОВ

Файл **async.js**:

```
console.timeEnd('async');
```

Файл **defer.js**:

```
console.timeEnd('defer');
```

СОЗДАЕМ ТАЙМЕР

Файл **timer.js**:

```
1  [  
2    'last',  
3    'async',  
4    'defer',  
5    'DOMContentLoaded',  
6    'load'  
7  ].forEach(title => console.time(title));  
8  document.addEventListener('DOMContentLoaded', () => {  
9    console.timeEnd('DOMContentLoaded');  
10 });  
11 window.addEventListener('load', () => {  
12   console.timeEnd('load');  
13 });
```

РАЗМЕТКА

```
1 <script src="timer.js"></script>
2 <script src="async.js" async></script>
3 <script src="defer.js" defer></script>
4
5 <a href="http://netology.ru/">Нетология</a>
6 <a href="https://foxford.ru/">Фоксфорд</a>
7
8 <script>console.timeEnd('last');</script>
```

[Весь код](#)

РЕЗУЛЬТАТ В КОНСОЛИ

```
last: 0.385009765625ms  
defer: 3.116943359375ms  
DOMContentLoaded: 3.5029296875ms  
async: 4.3408203125ms  
load: 5.928955078125ms
```

Время может быть разным при каждом новом запуске, но порядок исполнения скриптов будет всегда оставаться таким.

ВЫВОДЫ

- Если вам критично выполнить ваш код как можно раньше, подключите его перед `</body>`.
- Используйте `defer`, чтобы гарантировать последовательность выполнения скриптов, но при этом чтобы они выполнялись асинхронно.
- Используйте событие `DOMContentLoaded`, чтобы ваш код никак не зависел от способа подключения.
- Используйте `async` для скриптов, от которых не зависят другие скрипты, и которые нет необходимости выполнять как можно раньше.
- Используйте событие `load`, если хотите выполнить код после полной загрузки страницы.



ПОИСК ЭЛЕМЕНТА ПО CSS-СЕЛЕКТОРУ

НАЙДЕМ ВСЕ ССЫЛКИ

Для поиска нужного тега в HTML документе мы уже используем

`getElementById`, `getElementsByClassName`,
`getElementsByTagName`.

Давайте попробуем их использовать для поиска всех гиперссылок с классом `.download`:

```
1 function hasClass(className, node) {  
2     return node.classList.contains(className);  
3 }  
4  
5 const links = Array  
6     .from(document.getElementsByTagName('a'))  
7     .filter(hasClass.bind(null, 'download'));
```

ИЩЕМ ВЛОЖЕННЫЙ ``

А что, если мы хотим найти не сами ссылки, а теги `` с классом `.status` в них? Попробуем:

```
1 function extractStatus(node) {  
2   return Array  
3     .from(node.getElementsByTagName('span'))  
4     .filter(hasClass.bind(null, 'status'));  
5 }
```

Функция `extractStatus` найдет все `span` с классом `.status`, вложенные в тег, переданный в качестве аргумента.

ПРОДОЛЖАЕМ ПОИСКИ

```
1  const statuses = Array
2    .from(document.getElementsByTagName('a'))
3    .filter(hasClass.bind(null, 'download'))
4    .map(extractStatus)
5    .reduce((result, list) => result.concat(list), []);
```



«И ОНИ ПОСИДЕЛИ ЕЩЕ НЕМНОГО...»

Глядя на этот код, можно подумать что мы манипулируем большими данными. И суть решаемой задачи уже давно потерялась в этих деталях.

ПОСМОТРИМ НА CSS

Как бы мы решали эту же задачу в CSS, если бы нам потребовалось наделить эти статусы какими-то свойствами? Мы бы написали такой селектор:

```
1 | a.download span.status {  
2 |     /* свойства */  
3 | }
```

ДЕКЛАРАТИВНЫЙ СТИЛЬ

Подобный подход называется декларативным стилем. Мы говорим, что нам нужно сделать, а браузер знает, как найти нужные нам теги. И такой же стиль можно применить в DOM для поиска тегов.

Для этого есть два метода:

- `querySelectorAll` – принимает CSS-селектор, *строку*, и возвращает коллекцию элементов, которая соответствует этому селектору.
- `querySelector` – принимает CSS-селектор, *строку*, и возвращает первый соответствующий этому селектору элемент.

querySelectorAll В РАБОТЕ

Перепишем наше решение с использованием `querySelectorAll`:

```
const statuses = document
  .querySelectorAll('a.download span.status');
```

Всего лишь одна строка. Суть задачи доступна, а детали реализации скрыты и не отвлекают.

ТОЛЬКО ПЕРВЫЙ СТАТУС

А что, если нам нужны не все статусы, а только первый?

Так как `querySelectorAll` вернет коллекцию, то можем получить из нее нулевой элемент любым возможным способом, например деструкцией:

```
const [firstStatus] = document
  .querySelectorAll('a.download span.status');
```

Или просто возьмем элемент с индексом 0:

```
const firstStatus = document
  .querySelectorAll('a.download span.status')[0];
```

querySelector В РАБОТЕ

Но гораздо эффективнее для задачи, когда нам нужен только первый элемент, или предполагается что такой элемент вообще один, использовать метод `querySelector`. Перепишем код:

```
const firstStatus = document
  .querySelector('a.download span.status');
```

Иначе говоря, результат такой же, как и при `document.querySelectorAll(css)[0]`, но в последнем вызове сначала ищутся все элементы, а потом берётся первый, тогда как в `document.querySelector(css)` ищется только первый, то есть второй вариант эффективнее.

«Я ХОЧУ СЫГРАТЬ С ТОБОЙ В ОДНУ ИГРУ...»





«КТО ИЩЕТ, ТО ВСЕГДА НАЙДЕТ»

Вопрос: Что ищет селектор `a`?

«КТО ИЩЕТ, ТО ВСЕГДА НАЙДЕТ»

Вопрос: Что ищет селектор `a`?

Ответ: Все теги `<a>` в документе.

Для поиска по имени тега в JavaScript лучше использовать `document.getElementsByTagName('a')`.



«КТО ИЩЕТ, ТО ВСЕГДА НАЙДЕТ»

Вопрос: Что ищет селектор `#total`?

«КТО ИЩЕТ, ТО ВСЕГДА НАЙДЕТ»

Вопрос: Что ищет селектор `#total`?

Ответ: Тег с идентификатором (атрибут `id`) `total`.

Для поиска по идентификатору в JavaScript лучше использовать `document.getElementById('total')`.



«КТО ИЩЕТ, ТО ВСЕГДА НАЙДЕТ»

Вопрос: Что ищет селектор `.selected`?

«КТО ИЩЕТ, ТО ВСЕГДА НАЙДЕТ»

Вопрос: Что ищет селектор `.selected`?

Ответ: Теги содержащие класс (атрибут `class`) `selected`.

Для поиска по идентификатору в JavaScript лучше использовать `document.getElementsByClassName('selected')`.



«КТО ИЩЕТ, ТО ВСЕГДА НАЙДЕТ»

Вопрос: Что ищет селектор `.current > h2` ?

«КТО ИЩЕТ, ТО ВСЕГДА НАЙДЕТ»

Вопрос: Что ищет селектор `.current > h2`?

Ответ: Теги `h2` дочерние для тега с классом `current`.

Для поиска по более сложным селекторам в JavaScript используйте `querySelectorAll` или `querySelector`.

ИЩЕМ ИЗОЛИРОВАННО

Иногда нам нужно найти не все теги в документе, соответствующие селектору, а только те, что есть в элементе.

Методы `querySelectorAll` или `querySelector` есть у каждого элемента, и поэтому мы можем их использовать для поиска:

```
1 function getAllStatuses(node) {  
2     return node.querySelectorAll('a.download span.status');  
3 }  
4  
5 const main = document.getElementById('mainPanel');  
6 const allStatuses = getAllStatuses(document);  
7 const mainStatuses = getAllStatuses(main);
```

НЕСКОЛЬКО СЕЛЕКТОРОВ ЧЕРЕЗ ЗАПЯТУЮ

Как и в CSS, можно перечислять несколько селекторов через запятую:

```
const selectedAndCurrent = document
  .querySelectorAll('.selected, .current');
```

ИЩЕМ ПСЕВДОКЛАСС / ПСЕВДОЭЛЕМЕНТ

Если попробовать использовать псевдоэлемент или псевдокласс, то вернет пустую коллекцию, даже если они есть:

```
const before = document.querySelectorAll('div::before');  
console.log(before.length === 0);
```



ОБЗОР СЕЛЕКТОРОВ

ОСНОВНЫЕ ВИДЫ СЕЛЕКТОРОВ

- `*` — любые элементы.
- `div` — элементы с таким тегом.
- `#id` — элемент с данным `id`.
- `.class` — элементы с таким классом.
- `[name="value"]` — селекторы по атрибуту.

КОМБИНИРОВАНИЕ СЕЛЕКТОРОВ

- `.c1.c2` – элементы одновременно с двумя классами `c1` и `c2`.
- `a#id.c1.c2` – элемент `a` с данным `id`, классами `c1` и `c2`.

ОТНОШЕНИЯ

- `div p` – элементы `p`, являющиеся потомками `div`.
- `div > p` – только непосредственные потомки
- `div ~ p` – правые соседи: все `p` на том же уровне вложенности, которые идут после `div`.
- `div + p` – первый правый сосед: `p` на том же уровне вложенности, который идёт сразу после `div` (если есть).



АТТРИБУТЫ И СВОЙСТВА ЭЛЕМЕНТА

ИЗВЕСТНЫЕ АТТРИБУТЫ

Мы уже знаем, что многие атрибуты тегов, например, `id`, `src`, `href`, `title` и др, доступны `HTMLElement` в качестве свойства. И изменение свойства приводит к изменению атрибута.

Также мы знаем, что есть и исключения. Например, для атрибута `class` свойство названо `className`, а еще есть более удобное свойство для работы с классами `classList`.

ОБЪЕКТ HTMLElement

Так как `HTMLElement` просто объект, мы можем добавлять в него различные свойства. Давайте попробуем:

```
const item = document.querySelector('.catalog > a');  
item.likes = 42;
```

Появится ли такой атрибут у тега `<a>`? Нет!

СВЯЗЬ СВОЙСТВА-АТТРИБУТЫ

И это вовсе не связано с тем, что такого атрибута нет в спецификации. Просто в DOM при создании `HTMLElement` для тега связь свойств элемента устанавливается только с узким набором атрибутов тега.

СМОТРИМ НА `href`

И не всегда отличия только в названии:

```
1 <nav class="catalog">
2   <a href="./path/to/item.html">Товар 1</a>
3 </nav>
4 <script>
5   const item = document.querySelector('.catalog > a');
6   console.log(item.href);
7   // http://example.com/section/path/to/item.html
8 </script>
```

То, что записано в атрибуте `href` тега и то, что доступно в свойстве `href` элемента — различается. Потому что по спецификации свойство `href` должно содержать полный URL-адрес. И так не только со ссылками.

АТРИБУТЫ `<audio>`

В теге `<audio>` мы работали со свойством `controls`, которое имеет значение `true` или `false`. А атрибут может быть вообще без значения, а может содержать любую строку.

МЕТОДЫ ДЛЯ РАБОТЫ С АТТРИБУТАМИ

- `node.hasAttribute(name)` – принимает название атрибута, строка, и вернет `true`, если такой атрибут в теге присутствует, иначе вернет `false`.
- `node.getAttribute(name)` – принимает название атрибута, строка, и вернет значение атрибута, либо пустую строку, если атрибута нет (многие браузеры могут вернуть `null` в этом случае).
- `node.setAttribute(name, value)` – принимает название атрибута, строка, и значение атрибута, строка, создает атрибут с таким значением, если его у тега не было, иначе переписывает значение атрибута.
- `node.removeAttribute(name)` – принимает название атрибута, строка, и удаляет атрибут.

Названия атрибутов автоматически приводятся к нижнему регистру.

ИСПОЛЬЗУЕМ `getAttribute`

Перепишем пример со ссылкой так, чтобы в консоль выводилось значение, которое указано в атрибуте, а не полный путь. Для этого используем `getAttribute`:

```
1 <nav class="catalog">
2   <a href="./path/to/item.html">Товар 1</a>
3 </nav>
4 <script>
5   const item = document.querySelector('.catalog > a');
6   console.log(item.getAttribute('href'));
7   // ./path/to/item.html
8 </script>
```

ДОБАВИМ СВОЙ АТТРИБУТ

Добавим новый атрибут через `setAttribute`:

```
const item = document.querySelector('.catalog > a');  
item.setAttribute('likes', 42);
```

Атрибут появился. Но изобретать свои атрибуты — плохой стиль.



ДАТА-АТТРИБУТЫ

ИСПОЛЬЗУЕМ ЛЮБЫЕ АТТРИБУТЫ

В HTML5 появилась возможность использовать любые атрибуты у тегов для собственных нужд. Давайте договоримся называть их дата-атрибуты. Это рабочие лошадки, которые позволяют передать из HTML-разметки в JavaScript или обратно дополнительные данные. Они задаются точно так же, как и обычные, но с префиксом `data-`.

ДОБАВИМ ЛАЙКИ

Работать с ними можно так же, как с остальными атрибутами:

```
1 <nav class="catalog">
2   <a href="/path/to/item.html" data-likes="42">
3     Товар 1
4   </a>
5 </nav>
6 <script>
7   const item = document.querySelector('.catalog > a');
8   console.log(item.getAttribute('data-likes'));
9   // 42
10 </script>
```

СВОЙСТВО dataset

Но в `HTMLElement` для работы с дата-атрибутами реализовано дополнительное свойство `dataset`. И все заданные в теге дата-атрибуты становятся его свойствами:

```
1 <nav class="catalog">
2   <a href="/path/to/item.html" data-likes="42">Товар 1</a>
3 </nav>
4 <script>
5   const item = document.querySelector('.catalog > a');
6   console.log(item.dataset.likes);
7   // 42
8 </script>
```

Без префикса `data-`.

ЛЮБОЕ СВОЙСТВО

Любое свойство, добавленное в `dataset`, автоматически становится дата-атрибутом, а значение приводятся к строке:

```
1 <nav class="catalog">
2   <a href="./path/to/item.html">Товар 1</a>
3 </nav>
4 <script>
5   const item = document.querySelector('.catalog > a');
6   item.dataset.liked = true;
7   item.dataset.from = ['Иван', 'Мария'];
8   item.dataset.owner = { name: 'Денис' };
9 </script>
```

РЕЗУЛЬТАТ

```
1 <a href="./path/to/item.html"  
2   data-liked="true"  
3   data-from="Иван,Мария"  
4   data-owner="[object Object]">Товар 1</a>
```

РЕГИСТРЫ ИМЕН

В названии HTML-атрибутов принят «змеиный» регистр, **слова-разделяются-дефисом**. Для свойств в JavaScript принят нижний «верблюжий» регистр, когда второе и следующие слова с **большойБуквы**. При формировании имен свойств и атрибутов преобразование выполняется автоматически.

РАЗНОЕ НАПИСАНИЕ

Используйте дефис в HTML и заглавные буквы в JavaScript для разделения слов:

```
1 <nav class="catalog">
2   <a href="./path/to/item.html" data-user-likes="42">
3     Товар 1</a>
4 </nav>
5 <script>
6   const item = document.querySelector('.catalog > a');
7   console.log(item.dataset.userLikes); // 42
8   item.dataset.finalPrice = 8999;
9 </script>
```

HTML ПОСЛЕ ИЗМЕНЕНИЙ

HTML будет выглядеть вот так после изменения:

```
<a  
  href="/path/to/item.html"  
  data-user-likes="42"  
  data-final-price="8999">  
  Товар 1  
</a>
```



УПРАВЛЕНИЕ СТИЛЯМИ

УПРАВЛЯЕМ СТИЛЯМИ

С помощью свойства `style` у `HTMLElement` можно абсолютно так же управлять стилями элемента. Свойство `style` это объект. Правила преобразования названий CSS-свойств в параметры объекта `style` и обратно точно такие же, как и для `dataset`.

ПОКАЗЫВАЕМ ТОВАР ПО КЛИКУ

Реализуем с помощью стиля `display` скрытие и показ товара в каталоге по клику на кнопку:

```
1 <button id="showCatalog">Показать товар</button>
2 <nav class="catalog">
3   <a href="/path/to/item.html" data-user-likes="42">
4     Товар 1
5   </a>
6 </nav>
```


ДОПИШЕМ СКРИПТ

```
1 function toggleItem() {  
2     if (item.style.display === 'none') {  
3         item.style.display = 'initial';  
4     } else {  
5         item.style.display = 'none';  
6     }  
7 }  
8 const trigger = document.getElementById('showCatalog');  
9 const item = document.querySelector('.catalog > a');  
10  
11  
trigger.addEventListener('click', toggleItem);
```

РАЗДЕЛЯЕМ ЛОГИКУ И ПРЕДСТАВЛЕНИЕ

В целом с развитием CSS, появлением там переходов, и с появлением удобной манипуляции классами элемента, управление стилями в JavaScript отходит на второй план. К ней прибегают только в ситуациях, когда количество состояний довольно большое. В остальных случаях JavaScript лишь переключает классы, соответствующие состоянию, а анимация переходом между состояниями описывается в CSS. В этом случае происходит разделение логики и представления.

ВЫНЕСЕМ СТИЛИ ОТОБРАЖЕНИЯ ЭЛЕМЕНТОВ В CSS

Напишем стили в файле **style.css** и подключим его в **index.html**:

```
1  .catalog a {  
2      opacity: 1;  
3      transition: all 0.5s ease-out;  
4  }  
5  .catalog a.hidden {  
6      opacity: 0;  
7  }
```

```
1  <head>  
2      <link rel="stylesheet" href="./style.css" />  
3  </head>
```

ИСПОЛЬЗУЕМ CSS-АНИМАЦИЮ

Перепишем разметку с использованием класса и CSS-анимации:

```
1 <button id="showCatalog">Показать товар</button>
2 <nav class="catalog">
3   <a href="./path/to/item.html" data-user-likes="42"
4     class="hidden">Товар 1</a>
5 </nav>
```

УПРАВЛЯЕМ КЛАССАМИ

```
1 function toggleItem() {  
2     item.classList.toggle('hidden');  
3 }  
4 const trigger = document.getElementById('showCatalog');  
5 const item = document.querySelector('.catalog > a');  
6  
7 trigger.addEventListener('click', toggleItem);
```

[Весь код](#)



МЕНЯЕМ ТЕКСТ НА КНОПКЕ

Нам нужно еще поменять подпись на кнопке. Для этого пришло время узнать, как можно менять содержимое тега.



СОДЕРЖИМОЕ ТЕГА

СОДЕРЖИМОЕ HTML-ТЕГА

У `HTMLElement` есть свойство `innerHTML`, которое позволяет работать с HTML-содержимым элемента как со строкой. Давайте выведем текст кнопки в консоль:

```
1 function toggleItem() {  
2     item.classList.toggle('hidden');  
3     console.log(trigger.innerHTML);  
4     // Показать товар  
5 }
```

Видно, что это то, что находится между открывающим и закрывающим тегом. Без самого тега.

«СКРЫТЬ / ПОКАЗАТЬ ТОВАР»

Поменяем текст в зависимости от того, установлен класс `hidden` или нет:

```
1 function toggleItem() {  
2   item.classList.toggle('hidden');  
3   trigger.innerHTML = item.classList.contains('hidden') ?  
4     'Показать товар' : 'Скрыть товар';  
5 }
```

[Live Demo](#)

ДОБАВИМ КАРТИНКУ

Так как это содержимое тега, а тег может содержать другие теги, давайте добавим картинку:

```
1 function toggleItem() {  
2     item.classList.toggle('hidden');  
3     trigger.innerHTML = item.classList.contains('hidden') ?  
4     ' Показать товар' :  
5     ' Скрыть товар';  
6 }
```

ДОПУЩЕНА ОШИБКА

А что будет, если сформировать HTML не правильно? Давайте попробуем:

```
1 function toggleItem() {  
2   item.classList.toggle('hidden');  
3   trigger.innerHTML = item.classList.contains('hidden') ?  
4     '<strong>Показать</em> товар' :  
5     'Скрыть товар';  
6 }
```



БРАУЗЕР НАЧЕКУ

Браузер исправит HTML-код автоматически, ровно так же, как он это делает, когда есть ошибки в самом HTML. Но полагаться на это точно не стоит.

ОСОБЕННОСТИ DOM

- Скрипту доступна только та часть DOM, которая была обработана до начала исполнения скрипта.
- Браузер создает элемент, как только встречает открывающий тег.
- Функция `setTimeout` – асинхронная.
- Самый надежный способ работы с DOM – работать с ним, когда браузер его полностью сформировал.
- Событие, генерируемое по окончании формирования DOM-дерева – `DOMContentLoaded`.
- `DOMContentLoaded` происходит на `document`.
- Событие, генерируемое по окончании загрузки всех ресурсов на странице – `load`.
- Событие `load` происходит на `window`.

АСИНХРОННАЯ ЗАГРУЗКА

- Атрибуты `async` и `defer` не влияют на встроенные в HTML скрипты.
- Скрипты с `defer` будут выполняться по порядку.
- Используйте скрипты с `defer` для связанных скриптов.
- Скрипты с `defer` выполняются после построения DOM-дерева, но до события `DOMContentLoaded`.
- Браузер не останавливает обработку страницы при обнаружении скрипта с `async`.

ДЕКЛАРАТИВНЫЕ ПОИСКИ

- `querySelectorAll` возвращает коллекцию элементов.
- `querySelector` возвращает первый найденный элемент.
- Эти методы есть у каждого элемента.
- Можно искать несколько селекторов через запятую.
- Не работает при поиске псевдоклассов / псевдоэлементов.

АТТРИБУТЫ ТЕГОВ

- При создании `HTMLElement` для тега связь свойство устанавливается с ограниченным набором атрибутов.
- Проверяем, есть ли атрибут – `node.hasAttribute(name)`.
- Проверяем значение атрибута – `node.getAttribute(name)`.
- Создаем атрибут со значением или переписываем существующее – `node.setAttribute(name, value)`.
- Удаляем атрибут – `node.removeAttribute(name)`.
- Не изобретайте своих атрибутов. Используйте атрибуты с `data-`.
- Все дата-атрибуты становятся свойствами `dataset`.
- Любое свойство становится атрибутом, а значение присваивается новому атрибуту.

УПРАВЛЕНИЕ СТИЛЯМИ

- Для управления стилями элемента используется свойство `style` у `HTMLElement`.
- Но лучше хранить все стили в CSS, управляя только классами элемента.
- `innerHTML` позволяет работать с содержимым элемента как со строкой.
- Можно добавлять с его помощью и новые теги.



НЕТОЛОГИЯ
университет интернет-професий

Задавайте вопросы и напишите отзыв о лекции!

МИХАИЛ КУЗНЕЦОВ



[@mkuznetcov](https://www.instagram.com/mkuznetcov)