JavaScript - работа с элементами

Поиск, Свойства, Атрибуты, Изменения документа, классы и стили (ближе к css);

Поиск

- document.getElementById
- querySelectorAll
- querySelector
- matches
- closest
- getElementsBy* (живые коллекции)

document.getElementById

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

Значение id должно быть уникальным

Значение id должно быть уникальным. В документе может быть только один элемент с данным id.

Если в документе есть несколько элементов с одинаковым значением id, то поведение методов поиска непредсказуемо. Браузер может вернуть любой из них случайным образом. Поэтому, пожалуйста, придерживайтесь правила сохранения уникальности id.

Только document.getElementById, а не anyElem.getElementById

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

```
<div id="elem">
  <div id="elem-content">Element</div>
</div>

<script>
  // получить элемент
  let elem = document.getElementById('elem');

  // сделать его фон красным
  elem.style.background = 'red';
</script>
```

```
<body>
  <div id="elem">
    <div id="elem-content">Элемент</div>
  </div>

  <script>
    // elem – ссылка на элемент с id="elem"
    elem.style.background = 'red';

    // внутри id="elem-content" есть дефис, так что такой id не может служить именем переменной
    // ...но мы можем обратиться к нему через квадратные скобки: window['elem-content']
  </script>
</body>
```

querySelectorAll

Самый универсальный метод поиска – это elem.querySelectorAll(css), он возвращает все элементы внутри elem, удовлетворяющие данному CSS-селектору.

Псевдоклассы тоже работают

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

```
<ul>
  <li>Этот</li>
  <li>тест</li>
</ul>
<ul>
  <li>полностью</li>
  <li>пройден</li>
</ul>
<script>
  let elements = document.querySelectorAll('ul > li:last-child');

  for (let elem of elements) {
    alert(elem.innerHTML); // "тест", "пройден"
  }
</script>
```

querySelector

Метод elem.querySelector(css) возвращает первый элемент, соответствующий данному CSS-селектору.

Иначе говоря, результат такой же, как при вызове elem.querySelectorAll(css)[0], но он сначала найдет все элементы, а потом возьмёт первый, в то время как elem.querySelector найдёт только первый и остановится. Это быстрее, кроме того, его короче писать.

matches

Метод elem.matches(css) ничего не ищет, а проверяет, удовлетворяет ли elem CSS-селектору, и возвращает true или false.

Этот метод удобен, когда мы перебираем элементы (например, в массиве или в чём-то подобном) и пытаемся выбрать те из них, которые нас интересуют.

```
<a href="http://example.com/file.zip">...</a>
<a href="http://ya.ru">...</a>

<script>
  // может быть любая коллекция вместо document.body.children
  for (let elem of document.body.children) {
    if (elem.matches('a[href$="zip"]')) {
      alert("Ссылка на архив: " + elem.href );
    }
  }
</script>
```

closest

Предки элемента – родитель, родитель родителя, его родитель и так далее. Вместе они образуют цепочку иерархии от элемента до вершины.

Метод elem.closest(css) ищет ближайшего предка, который соответствует CSS-селектору. Сам элемент также включается в поиск.

Другими словами, метод closest поднимается вверх от элемента и проверяет каждого из родителей. Если он соответствует селектору, поиск прекращается. Метод возвращает либо предка, либо null, если такой элемент не найден.

```
<h1>Содержание</h1>

<div class="contents">
  <ul class="book">
    <li class="chapter">Глава 1</li>
    <li class="chapter">Глава 2</li>
  </ul>
</div>

<script>
  let chapter = document.querySelector('.chapter'); // LI

  alert(chapter.closest('.book')); // UL
  alert(chapter.closest('.contents')); // DIV

  alert(chapter.closest('h1')); // null (потому что h1 – не предок)
</script>
```

getElementsBy* (живые коллекции)

Существуют также другие методы поиска элементов по тегу, классу и так далее.

На данный момент, они скорее исторические, так как querySelector более чем эффективен.

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

Возвращает коллекцию, а не элемент!

Буква "s" отсутствует в названии метода getElementById, так как в данном случае возвращает один элемент. Но getElementsByTagName вернёт список элементов, поэтому "s" обязательна.

```
<table id="table">
  <tr>
    <td>Ваш возраст:</td>

    <td>
      <label>
        <input type="radio" name="age" value="young" checked> младше 18
      </label>
      <label>
        <input type="radio" name="age" value="mature"> от 18 до 50
      </label>
      <label>
        <input type="radio" name="age" value="senior"> старше 60
      </label>
    </td>
  </tr>
</table>

<script>
  let inputs = table.getElementsByTagName('input');

  for (let input of inputs) {
    alert( input.value + ': ' + input.checked );
  }
</script>
```

Живые коллекции

Все методы "getElementsBy*" возвращают живую коллекцию. Такие коллекции всегда отражают текущее состояние документа и автоматически обновляются при его изменении.

Напротив, querySelectorAll возвращает статическую коллекцию. Это похоже на фиксированный массив элементов.

```
<div>First div</div>

<script>
  let divs = document.getElementsByTagName('div');
  alert(divs.length); // 1
</script>

<div>Second div</div>

<script>
  alert(divs.length); // 2
</script>
```

```
<div>First div</div>

<script>
  let divs = document.querySelectorAll('div');
  alert(divs.length); // 1
</script>

<div>Second div</div>

<script>
  alert(divs.length); // 1
</script>
```

Свойства

- innerHTML
- outerHTML
- nodeValue/data
- textContent / innerText
- hidden
- dataset / нестандартные атрибуты

innerHTML: содержимое элемента

Свойство innerHTML позволяет получить HTML-содержимое элемента в виде строки.

Мы также можем изменять его. Это один из самых мощных способов менять содержимое на странице.

Скрипты не выполняются

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

«innerHTML+=» осуществляет перезапись

Другими словами, innerHTML+= делает следующее:

1. Старое содержимое удаляется.
2. На его место становится новое значение innerHTML (с добавленной строкой).

Так как содержимое «обнуляется» и переписывается заново, все изображения и другие ресурсы будут перезагружены.

```
<body>
  <p>Параграф</p>
  <div>DIV</div>

  <script>
    alert( document.body.innerHTML ); // читаем текущее содержимое
    document.body.innerHTML = 'Новый BODY!'; // заменяем содержимое
  </script>

</body>
```

```
elem.innerHTML += "...";
// это более короткая запись для:
elem.innerHTML = elem.innerHTML + "..."
```

outerHTML: HTML элемента целиком

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

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

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

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

```
<div id="elem">Привет <b>Мир</b></div>

<script>
  alert(elem.outerHTML); // <div id="elem">Привет <b>Мир</b></div>
</script>
```

```
<div>Привет, мир!</div>

<script>
  let div = document.querySelector('div');

  // заменяем div.outerHTML на <p>...</p>
  div.outerHTML = '<p>Новый элемент</p>'; // (*)

  // Содержимое div осталось тем же!
  alert(div.outerHTML); // <div>Привет, мир!</div> (**)
</script>
```

nodeValue/data: содержимое текстового узла

Свойство innerHTML есть только у узлов-элементов.

У других типов узлов, в частности, у текстовых, есть свои аналоги: свойства nodeValue и data. Эти свойства очень похожи при использовании, есть лишь небольшие различия в спецификации. Мы будем использовать data, потому что оно короче.

```
<body>
  Привет
  <!-- Комментарий -->
  <script>
    let text = document.body.firstChild;
    alert(text.data); // Привет

    let comment = text.nextSibling;
    alert(comment.data); // Комментарий
  </script>
</body>
```

textContent: просто текст

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

Намного полезнее возможность записывать текст в textContent, т.к. позволяет писать текст «безопасным способом».

- С innerHTML вставка происходит «как HTML», со всеми HTML-тегами.
- С textContent вставка получается «как текст», все символы трактуются буквально.

В большинстве случаев мы рассчитываем получить от пользователя текст и хотим, чтобы он интерпретировался как текст. Мы не хотим, чтобы на сайте появлялся произвольный HTML-код. Присваивание через textContent – один из способов от этого защититься.

```
<div id="news">
  <h1>Срочно в номер!</h1>
  <p>Марсиане атаковали человечество!</p>
</div>

<script>
  // Срочно в номер! Марсиане атаковали человечество!
  alert(news.textContent);
</script>
```

```
<div id="elem1"></div>
<div id="elem2"></div>

<script>
  let name = prompt("Введите ваше имя?", "<b>Винни-пух!</b>");

  elem1.innerHTML = name;
  elem2.textContent = name;
</script>
```

Свойство «hidden»

Атрибут и DOM-свойство «hidden» указывает на то, видим ли мы элемент или нет.

Технически, hidden работает так же, как style="display:none". Но его применение проще.

```
<div>Оба тега DIV внизу невидимы</div>

<div hidden>С атрибутом "hidden"</div>

<div id="elem">С назначенным JavaScript свойством "hidden"</div>

<script>
  elem.hidden = true;
</script>
```

```
<div id="elem">Мигающий элемент</div>

<script>
  setInterval(() => elem.hidden = !elem.hidden, 1000);
</script>
```

dataset / нестандартные атрибуты

При написании HTML мы используем много стандартных атрибутов. Но что насчет нестандартных, пользовательских? Давайте посмотрим, полезны они или нет, и для чего они нужны.

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

Язык HTML живой, он растет, появляется больше атрибутов, чтобы удовлетворить потребности разработчиков. В этом случае могут возникнуть неожиданные эффекты.

Чтобы избежать конфликтов, существуют атрибуты вида data-*.

Использование data-* атрибутов – валидный, безопасный способ передачи пользовательских данных.

Все атрибуты, начинающиеся с префикса «data-», зарезервированы для использования программистами. Они доступны в свойстве dataset.

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

```
<style>
  .order[data-order-state="new"] {
    color: green;
  }

  .order[data-order-state="pending"] {
    color: blue;
  }

  .order[data-order-state="canceled"] {
    color: red;
  }
</style>

<div id="order" class="order" data-order-state="new">
  A new order.
</div>

<script>
  // чтение
  alert(order.dataset.orderState); // new

  // изменение
  order.dataset.orderState = "pending"; // (*)
</script>
```

Изменение документа

- Создание элемента
- Методы вставки
- insertAdjacentHTML/Text/Element
- Удаление узлов
- Клонирование узлов: cloneNode
- DocumentFragment

Создание элемента

DOM-узел можно создать двумя методами:

document.createElement(tag) - Создает новый элемент с заданным тегом.

document.createTextNode(text) - Создает новый текстовый узел с заданным текстом.

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

Методы вставки

Чтобы наш div появился, нам нужно вставить его где-нибудь в document. Например, в document.body.

Для этого есть метод append, в нашем случае: document.body.append(div).

Методы для различных вариантов вставки:

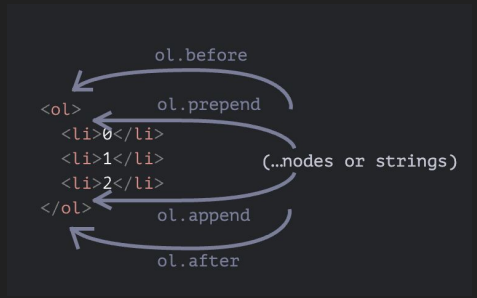
- node.append(...nodes or strings) – добавляет узлы или строки в конец node,
- node.prepend(...nodes or strings) – вставляет узлы или строки в начало node,
- node.before(...nodes or strings) –- вставляет узлы или строки до node,
- node.after(...nodes or strings) –- вставляет узлы или строки после node,
- node.replaceWith(...nodes or strings) –- заменяет node заданными узлами или строками.

```
<ol id="ol">
  <li>0</li>
  <li>1</li>
  <li>2</li>
</ol>

<script>
  ol.before('before'); // вставить строку "before" перед <ol>
  ol.after('after'); // вставить строку "after" после <ol>

  let liFirst = document.createElement('li');
  liFirst.innerHTML = 'prepend';
  ol.prepend(liFirst); // вставить liFirst в начало <ol>

  let liLast = document.createElement('li');
  liLast.innerHTML = 'append';
  ol.append(liLast); // вставить liLast в конец <ol>
</script>
```



insertAdjacentHTML/Text/Element

А что, если мы хотим вставить HTML именно «как html», со всеми тегами и прочим, как делает это elem.innerHTML?

С этим может помочь другой, довольно универсальный метод: elem.insertAdjacentHTML(where, html).

Первый параметр – это специальное слово, указывающее, куда по отношению к elem производить вставку. Значение должно быть одним из следующих:

- "beforebegin" – вставить html непосредственно перед elem,
- "afterbegin" – вставить html в начало elem,
- "beforeend" – вставить html в конец elem,
- "afterend" – вставить html непосредственно после elem.

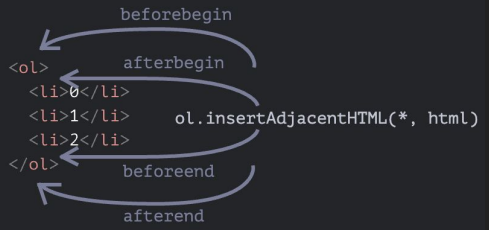
Второй параметр – это HTML-строка, которая будет вставлена именно «как HTML».

У метода есть два брата:

- elem.insertAdjacentText(where, text) – такой же синтаксис, но строка text вставляется «как текст», вместо HTML,
- elem.insertAdjacentElement(where, elem) – такой же синтаксис, но вставляет элемент elem.

```
<div id="div"></div>
<script>
  div.insertAdjacentHTML('beforebegin', '<p>Привет</p>');
  div.insertAdjacentHTML('afterend', '<p>Пока</p>');
</script>
```

```
<p>Привет</p>
<div id="div"></div>
<p>Пока</p>
```



Удаление узлов

Для удаления узла есть методы node.remove().

Если нам нужно переместить элемент в другое место – нет необходимости удалять его со старого.

Все методы вставки автоматически удаляют узлы со старых мест.

```
<style>
.alert {
  padding: 15px;
  border: 1px solid #d6e9c6;
  border-radius: 4px;
  color: #3c763d;
  background-color: #dff0d8;
}
</style>

<script>
  let div = document.createElement('div');
  div.className = "alert";
  div.innerHTML = "<strong>Всем привет!</strong> Вы прочитали важное сообщение.";

  document.body.append(div);
  setTimeout(() => div.remove(), 1000);
</script>
```

```
<div id="first">Первый</div>
<div id="second">Второй</div>
<script>
  // нет необходимости вызывать метод remove
  second.after(first); // берёт #second и после него вставляет #first
</script>
```

Клонирование узлов: cloneNode

Как вставить ещё одно подобное сообщение?

Мы могли бы создать функцию и поместить код туда. Альтернатива – клонировать существующий div и изменить текст внутри него (при необходимости).

Иногда, когда у нас есть большой элемент, это может быть быстрее и проще.

Вызов elem.cloneNode(true) создаёт «глубокий» клон элемента – со всеми атрибутами и дочерними элементами. Если мы вызовем elem.cloneNode(false), тогда клон будет без дочерних элементов.

```
<style>
.alert {
  padding: 15px;
  border: 1px solid #d6e9c6;
  border-radius: 4px;
  color: #3c763d;
  background-color: #dff0d8;
}
</style>

<div class="alert" id="div">
  <strong>Всем привет!</strong> Вы прочитали важное сообщение.
</div>

<script>
  let div2 = div.cloneNode(true); // клонировать сообщение
  div2.querySelector('strong').innerHTML = 'Всем пока!'; // изменить клонированный элемент

  div.after(div2); // показать клонированный элемент после существующего div
</script>
```

DocumentFragment

DocumentFragment является специальным DOM-узлом, который служит оберткой для передачи списков узлов.

Мы можем добавить к нему другие узлы, но когда мы вставляем его куда-то, он «исчезает», вместо него вставляется его содержимое.

DocumentFragment редко используется. Зачем добавлять элементы в специальный вид узла, если вместо этого мы можем вернуть массив узлов?

```
<ul id="ul"></ul>

<script>
function getListContent() {
  let fragment = new DocumentFragment();

  for(let i=1; i<=3; i++) {
    let li = document.createElement('li');
    li.append(i);
    fragment.append(li);
  }

  return fragment;
}

ul.append(getListContent()); // (*)
</script>
```

```
<ul id="ul"></ul>

<script>
function getListContent() {
  let result = [];

  for(let i=1; i<=3; i++) {
    let li = document.createElement('li');
    li.append(i);
    result.push(li);
  }

  return result;
}

ul.append(...getListContent()); // append + оператор "..." = друзья!
</script>
```

Стили и классы

- className и classList
- Свойство style
- Сброс стилей
- Вычисленные стили: getComputedStyle

className и classList

Когда-то давно в JavaScript существовало ограничение: зарезервированное слово типа "class" не могло быть свойством объекта. Это ограничение сейчас отсутствует, но в то время было невозможно иметь свойство elem.class.

Поэтому для классов было введено схожее свойство "className": elem.className соответствует атрибуту "class".

Если мы присваиваем что-то elem.className, то это заменяет всю строку с классами. Иногда это то, что нам нужно, но часто мы хотим добавить/удалить один класс.

Для этого есть другое свойство: elem.classList.

elem.classList – это специальный объект с методами для добавления/удаления одного класса.

Методы classList:

- elem.classList.add/remove("class") – добавить/удалить класс.
- elem.classList.toggle("class") – добавить класс, если его нет, иначе удалить.
- elem.classList.contains("class") – проверка наличия класса, возвращает true/false.

Кроме того, classList является перебираемым, поэтому можно перечислить все классы при помощи for..of:

```
<body class="main page">
  <script>
    alert(document.body.className); // main page
  </script>
</body>
```

```
<body class="main page">
  <script>
    // добавление класса
    document.body.classList.add('article');

    alert(document.body.className); // main page article
  </script>
</body>
```

```
<body class="main page">
  <script>
    for (let name of document.body.classList) {
      alert(name); // main, затем page
    }
  </script>
</body>
```

Свойство style

Свойство elem.style – это объект, который соответствует тому, что написано в атрибуте "style".

Свойства из одного слова записываются так же, с маленькой буквы:

```
background  => elem.style.background
top         => elem.style.top
opacity     => elem.style.opacity
```

Для свойств из нескольких слов используется camelCase:

```
background-color  => elem.style.backgroundColor
z-index           => elem.style.zIndex
border-left-width => elem.style.borderLeftWidth
```

Свойства с префиксом

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

```
button.style.MozBorderRadius = '5px';
button.style.WebkitBorderRadius = '5px';
```

Сброс стилей

Иногда нам нужно добавить свойство стиля, а потом, позже, убрать его.

Например, чтобы скрыть элемент, мы можем задать elem.style.display = "none".

Затем мы можем удалить свойство style.display, чтобы вернуться к первоначальному состоянию. Вместо delete elem.style.display мы должны присвоить ему пустую строку: elem.style.display = "".

Если мы установим в style.display пустую строку, то браузер применит CSS-классы и встроенные стили, как если бы такого свойства style.display вообще не было.

```
// если мы запустим этот код, <body> "мигнёт"
document.body.style.display = "none"; // скрыть

setTimeout(() => document.body.style.display = "", 1000);
// возврат к нормальному состоянию
```

Вычисленные стили: getComputedStyle

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

Поэтому, используя elem.style, мы не можем прочитать ничего, что приходит из классов CSS.

Для этого есть метод: getComputedStyle(element, [pseudo]).

element - Элемент, значения для которого нужно получить

pseudo - Указывается, если нужен стиль псевдоэлемента, например ::before. Пустая строка или отсутствие аргумента означают сам элемент.

Результат вызова – объект со стилями, похожий на elem.style, но с учётом всех CSS-классов.

```
<head>
  <style> body { color: red; margin: 5px } </style>
</head>
<body>

  Красный текст
  <script>
    alert(document.body.style.color); // пусто
    alert(document.body.style.marginTop); // пусто
  </script>
</body>
```

```
<head>
  <style> body { color: red; margin: 5px } </style>
</head>
<body>

  <script>
    let computedStyle = getComputedStyle(document.body);

    // сейчас мы можем прочитать отступ и цвет

    alert( computedStyle.marginTop ); // 5px
    alert( computedStyle.color ); // rgb(255, 0, 0)
  </script>

</body>
```

Стили, применяемые к посещенным :visited ссылкам, скрываются!

Посещённые ссылки могут быть окрашены с помощью псевдокласса :visited.

Но getComputedStyle не даёт доступ к этой информации, чтобы произвольная страница не могла определить, посещал ли пользователь ту или иную ссылку, проверив стили.

JavaScript не видит стили, применяемые с помощью :visited. Кроме того, в CSS есть ограничение, которое запрещает в целях безопасности применять к :visited CSS-стили, изменяющие геометрию элемента. Это гарантирует, что нет обходного пути для «злой» страницы проверить, была ли ссылка посещена и, следовательно, нарушить конфиденциальность.

Ресурсы

Поиск элементов по документу - ТЫК

Свойства элементов и узлов - ТЫК

Атрибуты и свойства - ТЫК

Изменение документа (HTML) - ТЫК

Стили и классы - ТЫК