Изменение документа

Модификации DOM – это ключ к созданию «живых» страниц.

Здесь мы увидим, как создавать новые элементы «на лету» и изменять уже существующие.

Пример: показать сообщение

Рассмотрим методы на примере – а именно, добавим на страницу сообщение, которое будет выглядеть получше, чем `alert`.

Вот такое:

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

<div class="alert">
  <strong>Всем привет!</strong> Вы прочитали важное соо
</div>
```

Всем привет! Вы прочитали важное сообщение.

Это был пример HTML. Теперь давайте создадим такой же `div`, используя JavaScript (предполагаем, что стили в HTML или во внешнем CSS-файле).

Создание элемента

DOM-узел можно создать двумя методами:

`document.createElement(tag)`

Создаёт новый *элемент* с заданным тегом:

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

`document.createTextNode(text)`

Создаёт новый *текстовый узел* с заданным текстом:

```
let textNode = document.createTextNode('А вот и я');
```

Создание сообщения

В нашем случае сообщение – это `div` с классом `alert` и HTML в нём:

```
let div = document.createElement('div');
div.className = "alert";
div.innerHTML = "<strong>Всем привет!</strong> Вы прочи
```

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)`.

Вот полный пример:

```
<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 про

  document.body.append(div);
</script>
```

Вот методы для различных вариантов вставки:

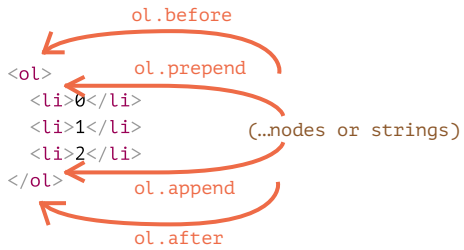
- `node.append(...nodes or strings)` – добавляет узлы или строки в конец `node`,
- `node.prepend(...nodes or strings)` – вставляет узлы или строки в начало `node`,
- `node.before(...nodes or strings)` –- вставляет узлы или строки до `node`,
- `node.after(...nodes or strings)` –- вставляет узлы или строки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

Вот пример использования этих методов, чтобы добавить новые элементы в список и текст до/после него:

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

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

  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>
```

```
before

1. prepend
2. 0
3. 1
4. 2
5. append

after
```

Наглядная иллюстрация того, куда эти методы вставляют:



Итоговый список будет таким:

```
1  before
2  <ol id="ol">
3    <li>prepend</li>
4    <li>0</li>
5    <li>1</li>
6    <li>2</li>
7    <li>append</li>
8  </ol>
9  after
```

Эти методы могут вставлять несколько узлов и текстовых фрагментов за один вызов.

Например, здесь вставляется строка и элемент:

```
1  <div id="div"></div>
2  <script>
3    div.before('<p>Привет</p>', document.createElement('h
4  </script>
```

Весь текст вставляется *как текст*.

Поэтому финальный HTML будет:

```
1  &lt;p&gt;Привет&lt;/p&gt;
2  <hr>
3  <div id="div"></div>
```

Другими словами, строки вставляются безопасным способом, как делает это `elem.textContent`.

Поэтому эти методы могут использоваться только для вставки DOM-узлов или текстовых фрагментов.

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

insertAdjacentHTML/Text/Element

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».

Например:

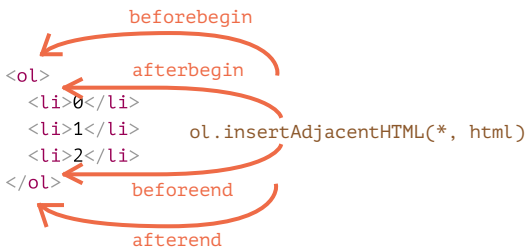
```
<div id="div"></div>
<script>
  div.insertAdjacentHTML('beforebegin', '<p>Привет</p>'
  div.insertAdjacentHTML('afterend', '<p>Пока</p>');
</script>
```

...Приведёт к:

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

Так мы можем добавлять произвольный HTML на страницу.

Варианты вставки:



Мы можем легко заметить сходство между этой и предыдущей картинкой. Точки вставки фактически одинаковые, но этот метод вставляет 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`.

Они существуют, в основном, чтобы унифицировать синтаксис. На практике часто используется только `insertAdjacentHTML`. Потому что для элементов и текста у нас есть методы `append/prepend/before/after` – их быстрее написать, и они могут вставлять как узлы, так и текст.

Так что, вот альтернативный вариант показа сообщения:

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

<script>
  document.body.insertAdjacentHTML("afterbegin", `<div
    <strong>Всем привет!</strong> Вы прочитали важное с

```

```
15    </div>`);
    </script>
```


Удаление узлов

Для удаления узла есть методы `node.remove()`.

Например, сделаем так, чтобы наше сообщение удалялось через секунду:

```
<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 про

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

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

Все методы вставки автоматически удаляют узлы со старых мест.

Например, давайте поменяем местами элементы:

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

Пример копирования сообщения:

```
<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 соо
</div>

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

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

DocumentFragment

`DocumentFragment` является специальным DOM-узлом, который служит обёрткой 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.

Например, `getListContent` ниже генерирует фрагмент с элементами `<li>`, которые позже вставляются в `<ul>`:

```
<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>
```

Обратите внимание, что на последней строке с `(*)` мы добавляем `DocumentFragment`, но он «исчезает», поэтому структура будет:

```
<ul>
  <li>1</li>
  <li>2</li>
  <li>3</li>
</ul>
```

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

```
<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;
}
```

```
16  ul.append(...getListContent()); // append + оператор ".
17  </script>
```


Мы упоминаем `DocumentFragment` в основном потому, что он используется в некоторых других областях, например, для элемента template, который мы рассмотрим гораздо позже.

Устаревшие методы вставки/удаления

> **Старая школа**
>
> Эта информация помогает понять старые скрипты, но не нужна для новой разработки.

Есть несколько других, более старых, методов вставки и удаления, которые существуют по историческим причинам.

Сейчас уже нет причин их использовать, так как современные методы `append`, `prepend`, `before`, `after`, `remove`, `replaceWith` более гибкие и удобные.

Мы упоминаем о них только потому, что их можно найти во многих старых скриптах:

`parentElem.appendChild(node)`

Добавляет `node` в конец дочерних элементов `parentElem`.

Следующий пример добавляет новый `<li>` в конец `<ol>`:

```
1  <ol id="list">
2    <li>0</li>
3    <li>1</li>
4    <li>2</li>
5  </ol>
6
7  <script>
8    let newLi = document.createElement('li');
9    newLi.innerHTML = 'Привет, мир!';
10
11   list.appendChild(newLi);
12 </script>
```

`parentElem.insertBefore(node, nextSibling)`

Вставляет `node` перед `nextSibling` в `parentElem`.

Следующий пример вставляет новый элемент перед вторым `<li>`:

```
1  <ol id="list">
2    <li>0</li>
3    <li>1</li>
4    <li>2</li>
5  </ol>
6  <script>
7    let newLi = document.createElement('li');
8    newLi.innerHTML = 'Привет, мир!';
9
10   list.insertBefore(newLi, list.children[1]);
11 </script>
```

Чтобы вставить `newLi` в начало, мы можем сделать вот так:

```
1  list.insertBefore(newLi, list.firstChild);
```

`parentElem.replaceChild(node, oldChild)`

Заменяет `oldChild` на `node` среди дочерних элементов `parentElem`.

`parentElem.removeChild(node)`

Удаляет `node` из `parentElem` (предполагается, что он родитель `node`).

Этот пример удалит первый `<li>` из `<ol>`:

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

<script>
  let li = list.firstElementChild;
  list.removeChild(li);
</script>
```

Все эти методы возвращают вставленный/удалённый узел. Другими словами, `parentElem.appendChild(node)` вернёт `node`. Но обычно возвращаемое значение не используют, просто вызывают метод.

Несколько слов о «document.write»

Есть ещё один, очень древний метод добавления содержимого на веб-страницу: `document.write`.

Синтаксис:

```
<p>Где-то на странице...</p>
<script>
  document.write('<b>Привет из JS</b>');
</script>
<p>Конец</p>
```

Вызов `document.write(html)` записывает `html` на страницу «прямо здесь и сейчас». Строка `html` может быть динамически сгенерирована, поэтому метод достаточно гибкий. Мы можем использовать JavaScript, чтобы создать полноценную веб-страницу и записать её в документ.

Этот метод пришёл к нам со времён, когда ещё не было ни DOM, ни стандартов... Действительно старые времена. Он всё ещё живёт, потому что есть скрипты, которые используют его.

В современных скриптах он редко встречается из-за следующего важного ограничения:

Вызов `document.write` работает только во время загрузки страницы.

Если вызвать его позже, то существующее содержимое документа затрётся.

Например:

```
<p>Через одну секунду содержимое этой страницы будет за
<script>
  // document.write через 1 секунду
  // вызов происходит после того, как страница загрузит
  setTimeout(() => document.write('<b>...Этим.</b>'), 1
</script>
```

Так что после того, как страница загружена, он уже непригоден к использованию, в отличие от других методов DOM, которые мы рассмотрели выше.

Это его недостаток.

Есть и преимущество. Технически, когда `document.write` запускается во время чтения HTML браузером, и что-то пишет в документ, то браузер воспринимает это так, как будто это изначально было частью загруженного HTML-документа.

Поэтому он работает невероятно быстро, ведь при этом *нет модификации DOM*. Метод пишет прямо в текст страницы, пока DOM ещё в процессе создания.

Так что, если нам нужно динамически добавить много текста в HTML, и мы находимся на стадии загрузки, и для нас очень важна скорость, это может помочь. Но на практике эти требования редко сочетаются. И обычно мы можем увидеть этот метод в скриптах просто потому, что они старые.

Итого

- Методы для создания узлов:
  - `document.createElement(tag)` – создаёт элемент с заданным тегом,
  - `document.createTextNode(value)` – создаёт текстовый узел (редко используется),
  - `elem.cloneNode(deep)` – клонирует элемент, если `deep==true`, то со всеми дочерними элементами.
- Вставка и удаление:
  - `node.append(...nodes or strings)` – вставляет в `node` в конец,
  - `node.prepend(...nodes or strings)` – вставляет в `node` в начало,
  - `node.before(...nodes or strings)` – вставляет прямо перед `node`,
  - `node.after(...nodes or strings)` – вставляет сразу после `node`,
  - `node.replaceWith(...nodes or strings)` – заменяет `node`.
  - `node.remove()` – удаляет `node`.
- Устаревшие методы:
  - `parent.appendChild(node)`
  - `parent.insertBefore(node, nextSibling)`
  - `parent.removeChild(node)`
  - `parent.replaceChild(newElem, node)`

  Все эти методы возвращают `node`.
- Если нужно вставить фрагмент HTML, то `elem.insertAdjacentHTML(where, html)` вставляет в зависимости от `where`:
  - `"beforebegin"` – вставляет `html` прямо перед `elem`,
  - `"afterbegin"` – вставляет `html` в `elem` в начало,
  - `"beforeend"` – вставляет `html` в `elem` в конец,
  - `"afterend"` – вставляет `html` сразу после `elem`.

  Также существуют похожие методы `elem.insertAdjacentText` и `elem.insertAdjacentElement`, они вставляют текстовые строки и элементы, но они редко используются.
- Чтобы добавить HTML на страницу до завершения её загрузки:
  - `document.write(html)`

  После загрузки страницы такой вызов затирает документ. В основном встречается в старых скриптах.

Задачи

createTextNode vs innerHTML vs textContent

важность: 5

У нас есть пустой DOM-элемент `elem` и строка `text`.

Какие из этих 3-х команд работают одинаково?

1. `elem.append(document.createTextNode(text))`
2. `elem.innerHTML = text`

3. `elem.textContent = text`

решение

Очистите элемент

важность: 5

Создайте функцию `clear(elem)`, которая удаляет всё содержимое из `elem`.

```
<ol id="elem">
  <li>Привет</li>
  <li>Мир</li>
</ol>

<script>
  function clear(elem) { /* ваш код */ }

  clear(elem); // очищает список
</script>
```

решение

Почему остаётся "aaa"?

важность: 1

Запустите этот пример. Почему вызов `remove` не удалил текст `"aaa"`?

```
<table id="table">
  aaa
  <tr>
    <td>Тест</td>
  </tr>
</table>

<script>
  alert(table); // таблица, как и должно быть

  table.remove();
  // почему в документе остался текст "aaa"??
</script>
```

решение

Создайте список

важность: 4

Напишите интерфейс для создания списка.

Для каждого пункта:

1. Запрашивайте содержимое пункта у пользователя с помощью `prompt`.
2. Создавайте элемент `<li>` и добавляйте его к `<ul>`.
3. Процесс прерывается, когда пользователь нажимает `Esc` или вводит пустую строку.

Все элементы должны создаваться динамически.

Если пользователь вводит HTML-теги -– пусть в списке они показываются как обычный текст.

Демо в новом окне

решение

Создайте дерево из объекта

важность: 5

Напишите функцию `createTree`, которая создаёт вложенный список `ul/li` из объекта.

Например:

```
let data = {
  "Рыбы": {
    "форель": {},
    "лосось": {}
  },

  "Деревья": {
    "Огромные": {
      "секвойя": {},
      "дуб": {}
    },
    "Цветковые": {
      "яблоня": {},
      "магнолия": {}
    }
  }
};
```

Синтаксис:

```
let container = document.getElementById('container');
createTree(container, data); // создаёт дерево в контей
```

Результат (дерево):

- Рыбы
  - форель
  - лосось
- Деревья
  - Огромные
    - секвойя
    - дуб
  - Цветковые
    - яблоня
    - магнолия

Выберите один из двух способов решения этой задачи:

1. Создать строку, а затем присвоить через `container.innerHTML`.
2. Создавать узлы через методы DOM.

Если получится – сделайте оба.

P.S. Желательно, чтобы в дереве не было лишних элементов, в частности -– пустых `<ul></ul>` на нижнем уровне.

Открыть песочницу для задачи.

решение

Выведите список потомков в дереве

важность: 5

Есть дерево, организованное в виде вложенных списков `ul/li`.

Напишите код, который добавит каждому элементу списка `<li>` количество вложенных в него элементов. Узлы нижнего уровня, без детей – пропускайте.

Результат:

- Животные [9]
  - Млекопитающие [4]
    - Коровы
    - Ослы
    - Собаки
    - Тигры
  - Другие [3]
    - Змеи
    - Птицы
    - Ящерицы
- Рыбы [5]
  - Аквариумные [2]
    - Гуппи
    - Скалярии
  - Морские [1]

Открыть песочницу для задачи.

решение

Создайте календарь в виде таблицы

важность: 4

Напишите функцию `createCalendar(elem, year, month)`.

Вызов функции должен создать календарь для заданного месяца `month` в году `year` и вставить его в `elem`.

Календарь должен быть таблицей, где неделя – это `<tr>`, а день – это `<td>`. У таблицы должен быть заголовок с названиями дней недели, каждый день – `<th>`, первым днём недели должен быть понедельник.

Например, `createCalendar(cal, 2012, 9)` сгенерирует в `cal` следующий календарь:

пн	вт	ср	чт	пт	сб	вс
					1	2
3	4	5	6	7	8	9
10	11	12	13	14	15	16
17	18	19	20	21	22	23
24	25	26	27	28	29	30

P.S. В этой задаче достаточно сгенерировать календарь, кликабельным его делать не нужно.

Открыть песочницу для задачи.

решение

Цветные часы с использованием setInterval

важность: 4

Создайте цветные часы как в примере ниже:

10:06:38
Старт Стоп

Для стилизации используйте HTML/CSS, JavaScript должен только обновлять время в элементах.

Открыть песочницу для задачи.

Раздел
Документ
Навигация по уроку
Пример: показать сообщение
Создание элемента
Методы вставки
insertAdjacentHTML/Text/Element
Удаление узлов
Клонирование узлов: cloneNode
DocumentFragment
Устаревшие методы вставки/удаления
Несколько слов о «document.write»
Итого
Задачи (10)
Комментарии
Поделиться
Редактировать на GitHub

решение

Вставьте HTML в список

важность: 5

Напишите код для вставки `<li>2</li><li>3</li>` между этими двумя `<li>`:

```
1  <ul id="ul">
2    <li id="one">1</li>
3    <li id="two">4</li>
4  </ul>
```

решение

Сортировка таблицы

важность: 5

Таблица:

Имя	Фамилия	Возраст
John	Smith	10
Pete	Brown	15
Ann	Lee	5
...	...	...

Может быть больше строк.

Напишите код для сортировки по столбцу `"name"`.

Открыть песочницу для задачи.

решение

Комментарии

перед тем как писать...