нетология
университет интернет-профессий

ИЗМЕНЕНИЕ СТРУКТУРЫ HTML-ДОКУМЕНТА

ЕВГЕНИЙ КОРЫТОВ / РУКОВОДИТЕЛЬ ВЕБ РАЗРАБОТКИ, KEENETIC

ЕВГЕНИЙ КОРЫТОВ

Руководитель веб разработки, Keenetic

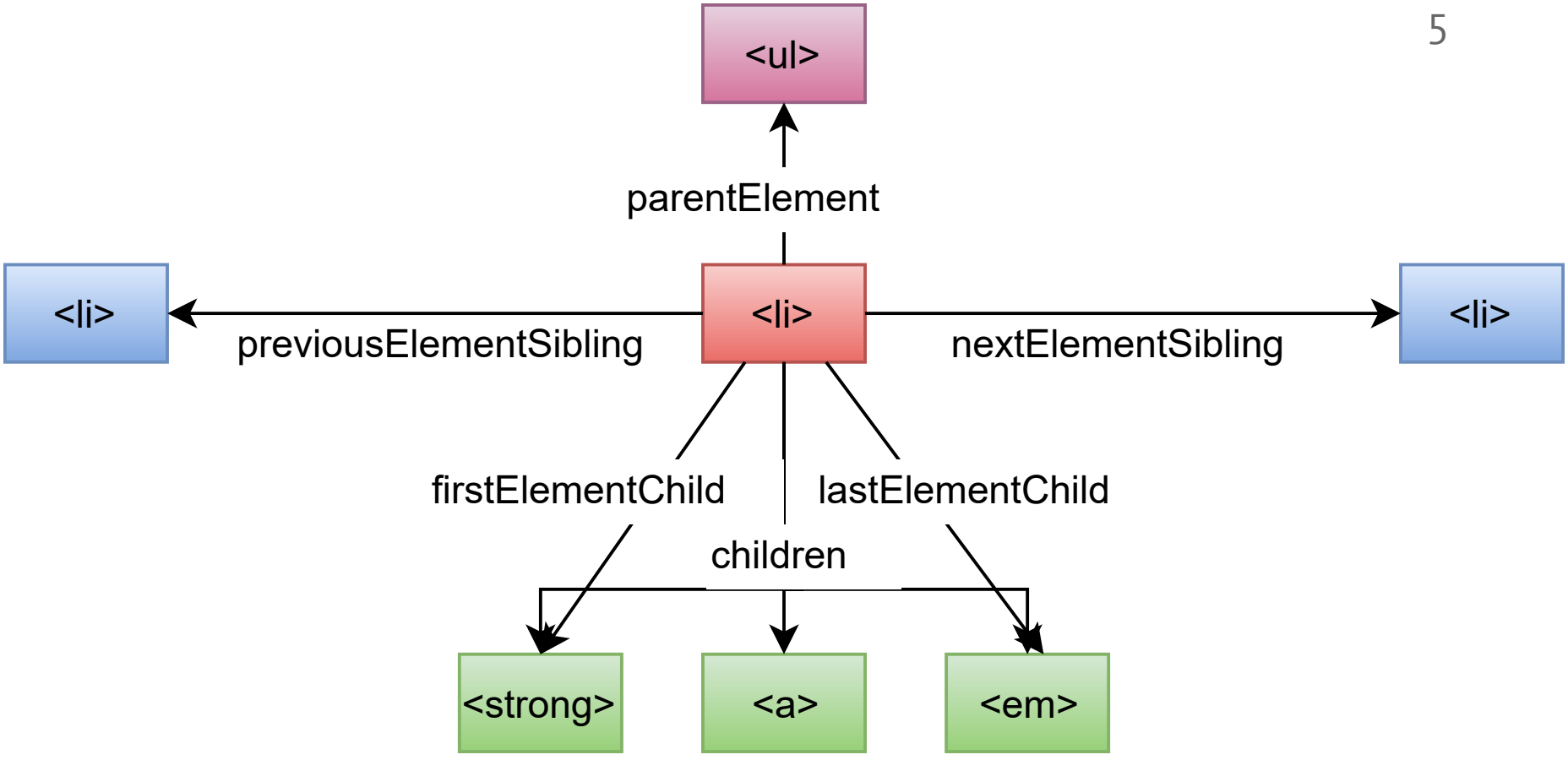
korytov.pro

korytoff@gmail.com

ПЛАН ЗАНЯТИЯ

1. Навигация по DOM
   - Родительский элемент
   - Соседи
   - А теперь все вместе
2. Модификация DOM-дерева
   - Свойства для чтения и записи содержимого элементов
   - Перемещение существующих узлов

НАВИГАЦИЯ ПО DOM



Браузер из HTML-разметки документа создает дерево объектов, с которыми мы можем взаимодействовать через JavaScript DOM API. В дереве у любого элемента есть родительский элемент, а также могут быть соседи и дочерние элементы. Для доступа к ним у каждого элемента DOM-дерева есть свойства, указывающие на его родителя, детей и соседей.

ПРИМЕР. КРАТКИЙ ВЫВОД С ПОДРОБНОСТЯМИ

Допустим, у нас есть краткий вывод, при клике на который мы хотим показать подробности. Для этого нам нужно обработать клик на `summary` и добавить или убрать класс `details-expanded` для родительского тего `details`:

```
<div class="details">
  <div class="summary">Хорошая погода</div>
  <p>В Москве ожидается +15 и солнце</p>
  <p>В Московской области до +18 без осадков</p>
</div>
```

ОБРАБОТЧИК СОБЫТИЯ

Добавим обработчик события `click` и напишем следующий код:

```
function expandDetails(event) {
  const details = document.querySelector('.details');
  details.classList.toggle('details-expanded');
}
const summaryItems = document
  .querySelectorAll('.summary');
Array.from(summaryItems).forEach(item => item
  .addEventListener('click', expandDetails));
```

РЕЗУЛЬТАТ ДОСТИГНУТ

Задача решена. Но не кажется ли вам, что в решении есть потенциальные проблемы?

ДВА ОДИНАКОВЫХ БЛОКА

С развитием проекта на странице может появить больше таких блоков. Например, два:

```
<div class="details">
  <div class="summary">Хорошая погода</div>
  <p>В Москве ожидается +15 и солнце</p>
  <p>В Московской области до +18, без осадков</p>
</div>
<p>Просто текст</p>
<div class="details">
  <div class="summary">Отличная погода</div>
  <p>В Омске ожидается +12 и небольшой дождь</p>
  <p>В Омской области до +15, облачно, без осадков</p>
</div>
```

ВОЗНИКШИЕ ПРОБЛЕМЫ

Теперь наш скрипт уже не решает поставленную задачу. Когда мы кликаем на второй блок, всё равно открывается первый. Почему?

БЕРЕМ ТОЛЬКО ПЕРВЫЙ ЭЛЕМЕНТ

Потому что при поиске по классу `details` мы сразу берем только первый элемент:

```
const details = document.querySelector('.details');
```

ИСПРАВЛЯЕМ СИТУАЦИЮ

Как это исправить? Возможные пути решения:

- Определить, на каком по счету `summary` мы кликнули, и выбирать `details`.
- Добавить `id` или разные классы элементам.
- Или еще как-то их связать между собой.

Все решения, по сути, сводятся к тому, чтобы установить какую-то связь между `summary` и его `details` и использовать её для поиска нужного элемента.

СУЩЕСТВУЮЩАЯ СВЯЗЬ

Но ведь между ними уже есть связь. Может быть, вы уже знаете, какая?

Вот фрагмент кода еще раз:

```
<div class="details">
  <div class="summary">Хорошая погода</div>
  …
</div>
```

Тут нам нужно вспомнить немного про то, что же из себя представляет DOM, с которым мы до сих пор взаимодействовали.

РОДИТЕЛЬ И ДЕТИ

Вы, наверное, уже догадались, что элементы `details` и `summary` связывают отношения родителя и ребенка, ведь `details` является родительским элементом для `summary`.

`parentElement`

Значит, имея доступ к `summary`, мы всегда можем получить доступ к `details` через свойство `parentElement`:

```
const details = summary.parentElement;
```

АДАПТИРУЕМ СКРИПТ

Перепишем наш код с использованием полученной информации:

```
function expandDetails(event) {
  const details = event.target.parentElement;
  details.classList.toggle('details-expanded');
}
const summaryItems = document
  .querySelectorAll('.summary');
Array.from(summaryItems).forEach(item => item
  .addEventListener('click', expandDetails));
```

Теперь этот код будет работать с любым количеством элементов на странице.

СЛАЙДЕР

Возьмем другой пример. Нам нужно сделать карусель.

У нас есть вот такая разметка:

```
<div class="slider">
  <ul>
    <li class="active">Первый слайд</li>
    <li>Второй слайд</li>
    <li>Третий слайд</li>
  </ul>
  <button class="slider-prev" disabled>Назад</button>
  <button class="slider-next">Вперед</button>
</div>
```

ОПИСАНИЕ РАБОТЫ СЛАЙДЕРА

Наша задача состоит в том, чтобы при нажатии кнопок `Вперед` и `Назад` класс `active` переходил вперед/назад. Если вперед/назад перейти невозможно, то соответствующая кнопка должна быть недоступна.

ЛИСТАЕМ ВПЕРЕД

Начнем с того, что добавим обработчики событий на кнопку управления `Вперед`:

```
function Slider(container) {
  const next = container.querySelector('.slider-next');
  next.addEventListener('click', event => {
    const currentSlide = container
      .querySelector('.active');
    currentSlide.classList.remove('active');
    const nextSlide = ???;
    nextSlide.classList.add('active');
  });
}
const sliders = document.querySelectorAll('.slider');
Array.from(sliders).forEach(item => Slider(item));
```

КАК ЛИСТАТЬ СЛАЙДЫ?

Как найти текущий слайд, мы знаем, но как перейти от текущего слайда к следующему?

ИЩЕМ СОСЕДЕЙ

Вспоминаем про DOM-дерево и про свойства, указывающие на соседей элемента, `nextElementSibling` и `previousElementSibling`:

```
const nextSlide = currentSlide.nextElementSibling;
```

ДОПИШЕМ СКРИПТ

Теперь код заработает:

```
next.addEventListener('click', event => {
  const currentSlide = container
    .querySelector('.active');
  currentSlide.classList.remove('active');
  const nextSlide = currentSlide.nextElementSibling;
  nextSlide.classList.add('active');
});
```

РАБОТА КНОПОК

Но как же нам включать и отключать кнопки управления, когда мы доходим до последнего слайда?

ЕСЛИ НЕТ СОСЕДЕЙ

Для этого нам нужно знать, когда у элемента есть соседи после него, а когда он является крайним. К счастью, `nextElementSibling` / `previousElementSibling` возвращают `null` в тех случаях, когда искомого соседа нет.

ПРОВЕРЯЕМ СОСЕДЕЙ

Вот такой код мы можем добавить в обработчик события:

```
next.disabled = nextSlide.nextElementSibling ?
  false : true;
```

СКРИПТ УПРАВЛЕНИЯ КНОПКАМИ

```
function Slider(container) {
 const next = container.querySelector('.slider-next');
 const prev = container.querySelector('.slider-prev');
 next.addEventListener('click', event =>
  moveSlide(true));
 prev.addEventListener('click', event =>
  moveSlide(false));
 // функция moveSlide
}
```

ФУНКЦИЯ ЛИСТАНИЯ СЛАЙДОВ

```
function moveSlide(isForward) {
 const currentSlide = container
  .querySelector('.active');
 const activatedSlide = isForward ?
  currentSlide.nextElementSibling :
  currentSlide.previousElementSibling;
 currentSlide.classList.remove('active');
 activatedSlide.classList.add('active');

 next.disabled = activatedSlide.nextElementSibling ?

  false : true;
 prev.disabled = activatedSlide.previousElementSibling ?
  false : true;
}
```

ОПРЕДЕЛЯЕМ СЛАЙДЫ

```
const sliders = document.querySelectorAll('.slider');
Array.from(sliders).forEach(item => Slider(item));
```

Весь код

А ТЕПЕРЬ ВСЕ ВМЕСТЕ

Для закрепления материала сделаем пример с навигацией по DOM-дереву, используя все известные нам навигационные свойства.

HTML-РАЗМЕТКА

Возьмем такой документ:

```
<ul>
  <li class="active">Элемент <em>1</em></li>
  <li>Элемент <em>2</em></li>
  <li>Элемент <em>3</em></li>
</ul>
<div class="controls">
  <div>Текущий элемент: <strong class="activeElementInspector" /></div>
  <div>
    <button class="up">Родитель</button>
  </div>
  <div>
    <button class="left">Сосед слева</button>
    <button class="right">Сосед справа</button>
  </div>
  <div>
    <button class="down">Первый дочерний элемент</button>
  </div>
</div>
```

НАПИШЕМ СТИЛИ

```
.active {
  border: 2px solid green;
}

.controls {
  position: absolute;
  bottom: 0;
  background-color: lightgray;
  padding: 15px;
  text-align: center;
}
```

ПЕРЕМЕЩАЕМСЯ ПО DOM

Привязываемся к контролам

```
let activeElement = document.querySelector('.active');
const activeElementInspector = document
  .querySelector('.activeElementInspector');
const controls = document.querySelector('.controls');
const up = controls.querySelector('.up');
const left = controls.querySelector('.left');
const right = controls.querySelector('.right');
const down = controls.querySelector('.down');
```

ПОДСВЕЧИВАЕМ АКТИВНЫЙ ЭЛЕМЕНТ

Активный элемент будет выделен зеленой рамочкой, и будет выведено его имя:

```
controls.addEventListener('click', event => {
  activeElement.classList.remove('active');
  // конструкция switch
  activeElement.classList.add('active');
  updateControls();
});
```

switch

В зависимости он контрола выбираем активный элемент

```
switch (event.target) {
  case up:
    activeElement = activeElement.parentElement;
    break;
  case left:
    activeElement = activeElement.previousElementSibling;
    break;
  case right:
    activeElement = activeElement.nextElementSibling;
    break;
  case down:
    activeElement = activeElement.firstElementChild;
    break;
}
```

ОБНОВЛЯЕМ КОНТРОЛЫ

Если в какую-то сторону пойти нельзя, отключаем этот контрол

```
updateControls();

function updateControls() {
  activeElementInspector.innerHTML = Object
    .getPrototypeOf(activeElement).constructor.name;
  up.disabled = activeElement.parentElement ?
    false : true;
  left.disabled = activeElement.previousElementSibling ?
    false : true;
  right.disabled = activeElement.nextElementSibling ?
    false : true;
  down.disabled = activeElement.firstElementChild ?
    false : true;
}
```

ВЕСЬ КОД В ОДНОМ МЕСТЕ

Весь код

РЕАЛЬНЫЕ ПРОЕКТЫ

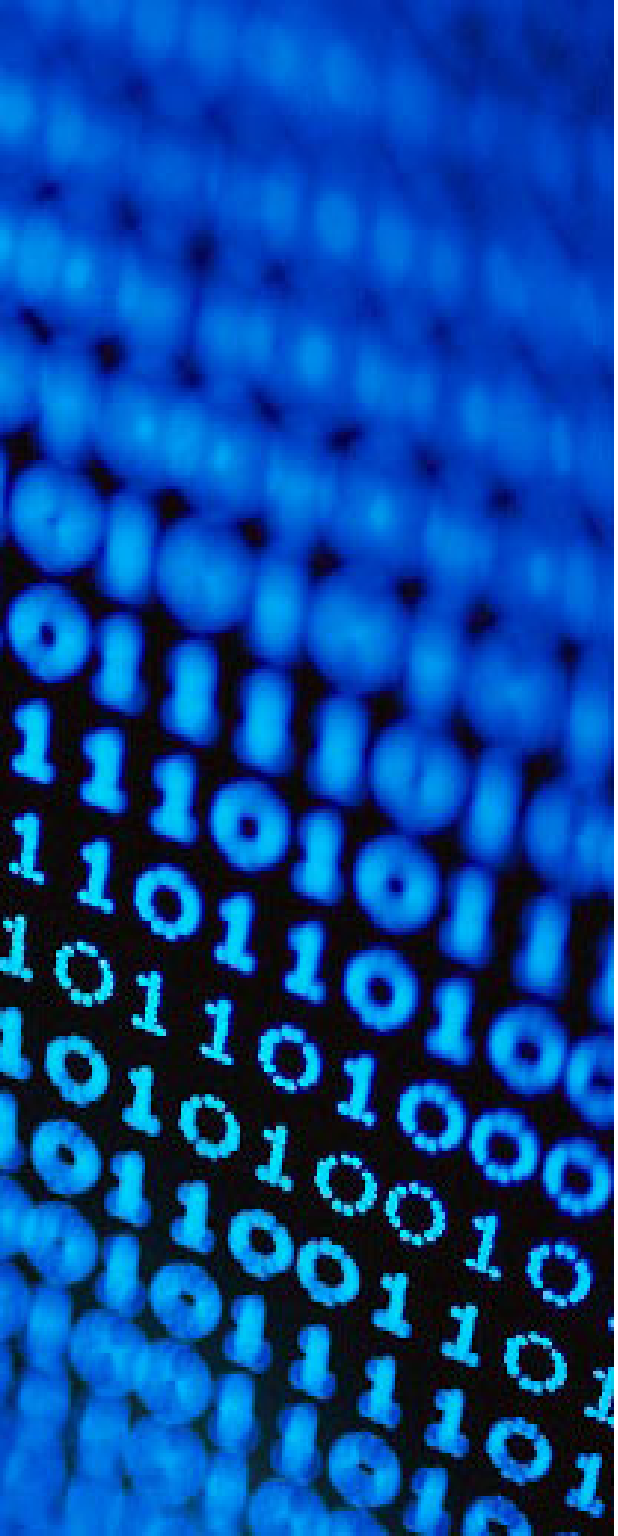
В реальных проектах нужно стараться не привязываться слишком тесно к структуре DOM и использовать для гибкой связи JavaScript и HTML data-атрибуты либо грамотно названные классы (например с префиксом `js-`, чтобы отличать их от классов, на которые привязываются стили).

МОДИФИКАЦИЯ DOM-ДЕРЕВА

РАЗДЕЛЕНИЕ СВОЙСТВ

Все способы работы с DOM-деревом можно условно разделить на следующие категории:

- Свойства для чтения и записи содержимого элементов;
- Перемещение существующих узлов.



«

Сериализация *– процесс перевода какой-либо структуры данных в последовательность битов.*

Кодирование данных последовательно по определению, и извлечение любой части сериализованной структуры данных требует, чтобы весь объект был считан от начала до конца и воссоздан.

`innerHTML`

Свойство `innerHTML` позволяет нам получить доступ к дочерним элементам в *сериализованном* виде в качестве HTML-разметки. Его можно использовать как для чтения, так и для записи.

МЕНЯЕМ СОДЕРЖИМОЕ АБЗАЦА

```
<p>
  <em>Я</em> абзац
</p>
<script>
  const p = document.querySelector('p');
  console.log(p.innerHTML); //<em>Я</em> абзац
  p.innerHTML = 'У меня <strong>поменяли</strong>
    содержимое!';
</script>
```

«ДОРОГОЙ» МЕТОД

Использование `innerHTML` для записи — всегда довольно дорогое удовольствие, так как браузеру приходится сначала парсить HTML-разметку и уже затем создавать сами элементы DOM. Если есть возможность, всегда лучше использовать более императивные методы работы с DOM, о которых мы будем говорить дальше.

ПРИМЕР ИСПОЛЬЗОВАНИЯ `innerHTML`

Типичный пример использования `innerHTML` — это когда мы хотим асинхронно подгрузить с сервера часть страницы при помощи XHR-запроса:

```
const container = querySelector('.ajax-container');
const request = new XMLHttpRequest();
request.open('GET', 'http://some-url', true);
request.onload = function() {
  if (request.status >= 200 && request.status < 400) {
    container.innerHTML = request.responseText;
  }
};
request.send();
```

ВСТАВЛЯЕМ ДАННЫЕ

В этом примере мы получаем готовую HTML-разметку с сервера и скармливаем ее `innerHTML` для парсинга.

В серьезных приложениях мы скорее будем получать с сервера JSON-данные и генерировать на их основе DOM при помощи императивных методов генерации DOM либо JS-библиотек.

XSS-атака *— это определенным образом встроенный в страницу сайта-жертвы скрипт, который будет выполнен при ее посещении.*

Часто используется злоумышленниками с целью кражи персональных данных пользователей или для взлома сайта.

ПРЕДОСТЕРЕЖЕНИЕ

Важно с осторожностью использовать `innerHTML` для записи данных, полученных от пользователя, чтобы не нарваться на XSS-атаку. Если вам не нужно работать с HTML-разметкой, то лучше использовать свойство `textContent`.

`textContent`

`textContent` работает похожим образом с `innerHTML`, но он не сериализует элементы в HTML, а представляет их в текстовом виде, выкидывая все теги.

Особенно это важно при записи: `textContent` работает быстрее, чем `innerHTML`, и не подвержен риску XSS-атаки.

СНОВА МЕНЯЕМ СОДЕРЖИМОЕ АБЗАЦА

```
<p>
  <em>Я</em> абзац
</p>
<script>
  const p = document.querySelector('p');
  console.log(p.textContent); // Я абзац
  p.textContent = "У меня <strong>поменяли</strong>
    содержимое!";
  // <strong> останется просто текстом
</script>
```

`outerHTML`

Свойство `outerHTML` работает похожим образом с `innerHTML`, но также включает сам элемент, а не только его дочерние элементы.

Его также можно использовать для замены элемента на произвольный набор тегов.

ЗАМЕНЯЕМ АБЗАЦ НА `div`

```
<p>
  <em>Я</em> абзац
</p>
<script>
  const p = document.querySelector('p');
  console.log(p.outerHTML); // <p><em>Я</em> абзац</p>
  p.outerHTML = '<div>Меня заменили на
    совсем другой тег!</div>';
  console.log(p.nodeName);
  // подстава, все еще "P", а не "DIV"!
</script>
```

ОСОБЕННОСТИ РАБОТЫ

Как вы заметили, у `outerHTML` есть один подвох, о котором важно помнить: после того, как мы заменили элемент при помощи `outerHTML`, переменная, которая ссылалась на старый элемент, все еще будет показывать на него.

`appendChild`

Метод `node.appendChild(someNode)` добавляет узел `someNode` в качестве последнего дочернего узла у узла `node`. Если узел `someNode` уже находился в DOM-дереве, то он будет перемещен из предыдущего места.

ПЕРЕМЕЩАЕМ АБЗАЦ

Переместим уже существующий абзац внутри DOM-дерева:

```
<p>Переместите меня внутрь div</p>
<div>
  <h1>Первый дочерний элемент</h1>
</div>
<script>
  const p = document.querySelector('p');
  const div = document.querySelector('div');
  div.appendChild(p);
</script>
```

РЕЗУЛЬТАТ ПЕРЕМЕЩЕНИЯ

В итоге абзац будет перемещен внутрь `div` и мы получим такую структуру:

```
<div>
  <h1>Первый дочерний элемент</h1>
  <p>Переместите меня внутрь div</p>
</div>
```

`cloneNode`

Метод `cloneNode` создает копию узла. `cloneNode(true)` клонирует узел вместе со всеми дочерними узлами.

Этот метод может быть полезен, если бы в прошлом примере мы захотели, чтобы абзац остался на своем старом месте и одновременно оказался бы в новом.

КОПИРУЕМ АБЗАЦ

Вот как это можно сделать:

```
<p>Скопируйте меня внутрь div</p>
<div>
  <h1>Первый дочерний элемент</h1>
</div>
<script>
  const p = document.querySelector('p').cloneNode(true);
  const div = document.querySelector('div');

  div.appendChild(p);
</script>
```

ВЕШАЕМ СОБЫТИЕ ЗАНОВО

Важно помнить, что `cloneNode` (так же, как и все остальные методы) не сохраняет обработчики событий, их нужно будет привязывать заново либо использовать делегацию событий.

`insertBefore`

Метод `parentElement.insertBefore(newNode, referenceNode)` добавляет `newNode` внутри `parentElement`, перед `referenceNode`.

Если `referenceNode` равен `null`, то узел добавится последним, как и в случае с `appendChild`.

`insertAfter`?

Интересно, что метода `insertAfter` не существует. Чтобы эмулировать его поведение, можно использовать свойство `nextSibling` вот так:

```
parentElement.insertBefore(newElement, referenceElement
  .nextSibling);
```

Хитрость в том, что если `referenceElement` является последним, то `nextSibling` будет равен `null`, и следовательно, `newElement` добавится в конец.

`removeChild`

Метод `removeChild` отсоединяет узел от его родителя в DOM, по сути, удаляя его из DOM-дерева, но не из памяти, и возвращает ссылку на отсоединенный узел. Как и любой другой JS-объект, узел будет находиться в памяти, пока на него существует ссылка. Дальше с этим узлом можно продолжать работать, например, снова подключить его к DOM при помощи методов, описанных выше.

УДАЛЯЕМ АБЗАЦ

```
<p>Удали меня</p>
<script>
  const p = document.querySelector('p');
  p.parentNode.removeChild(p);
  console.log(p);
  /* <p>Удали меня</p> <-- я все еще жив, пока на меня
  кто-то ссылается, только вне DOM */
</script>
```

ИТОГИ

НАВИГАЦИЯ ПО DOM

- У любого элемента есть свойства, которые определяют родственные связи внутри DOM-дерева.
- Родительская связь — универсальный инструмент, работающий без дополнительных классов/идентификаторов.
- `parentElement` дает доступ к родительскому элементу.
- `nextElementSibling` ищет следующего соседа.
- `previousElementSibling` ищет предыдущего соседа.
- Если соседей нет, то свойства возвращают `null`.
- В реальных проектах чаще стоит использовать data-атрибуты или специальные классы.

ЧТЕНИЕ И ЗАПИСЬ СОДЕРЖИМОГО В DOM

- `innerHTML` позволяет вставлять элементы вместе с HTML-разметкой.
- Метод `innerHTML` «дорогой» и уязвимый. Старайтесь не использовать его без необходимости.
- `textContent` вставляет только текст, игнорируя теги.
- `outerHTML` заменяет весь элемент целиком на произвольный набор тегов и содержимого.
- После замены элемента при помощи `outerHTML` переменная, указывающая на исходный элемент, так и будет на него указывать.

ПЕРЕМЕЩЕНИЕ СУЩЕСТВУЮЩИХ УЗЛОВ

– `appendChild` добавляет новый или перемещает существующий узел последним в родительский узел.

– `cloneNode` копирует элемент.

– `cloneNode(true)` копирует узел вместе со всеми дочерними узлами.

– `cloneNode` не копирует обработчик события, его надо вешать заново.

– `insertBefore` позволяет добавить узел перед другим узлом внутри родителя.

– Если предшествующий узел не указан (`null`), то новый узел встает последним.

– `insertAfter` не существует, но можно его эмулировать.

– `removeChild` удаляет узел из DOM, но сохраняет в памяти. В результате работы возвращает ссылку на удаленный узел.

нетология
университет интернет-профессий

Задавайте вопросы и напишите отзыв о лекции!

ЕВГЕНИЙ КОРЫТОВ


korytov.pro


korytoff@gmail.com