нетология

Конспект по теме 2.3 «Изменение структуры HTML-документа»

Содержание конспекта:

1. Содержимое HTML-элемента
2. Добавление HTML-разметки
3. Создание HTML-элементов
4. Удаление HTML-элементов
5. Замена и клонирование HTML-элементов

1. Содержимое HTML-элемента

Иногда нужно получить текст, находящийся внутри элемента. Для этого у каждого элемента есть свойство **textContent**, которое позволяет получить текстовый контент указанного узла и всех его потомков.
Это значение можно представить как сложение всех текстовых узлов, которые являются потомками узла, для которого вызывается свойство. Если элемент, для которого вызывается свойство **textContent**, содержит один дочерний текстовый узел, то свойство вернёт значение этого HTML-элемента.

Свойство innertext копирует текст между открывающим и закрывающим тегом элемента, отображаемого в веб-браузере.

В отличие от свойства **textContent** свойство **innerText** как бы копирует текст в веб-браузере, который отображается HTML-кодом, расположенным между открывающим и закрывающим тегом того элемента, для которого вызывается свойство, т. е. свойство **innerHTML** учитывает стили элементов, а именно — отображается элемент или нет, а, следовательно, и его содержимое в браузере. Это свойство также позволяет установить элементу заданный текстовый контент, т. е. заменить содержимое элемента, расположенное между его открывающим и закрывающим тегом, на указанное.

Исходный код страницы:

```
<div>
    <p>Old text</p>
</div>
```

Принцип работы свойства innerText можно рассмотреть на примере некоторого кода веб-страницы, в котором указан элемент абзаца. Предположим, что этот элемент содержит текстовый контент Old Text.
Выполнение p.innerText = “New Text” приведёт к изменению кода:

```
<div>
    <p>New Text!</p>   <!--Изменился внутренний текст-->
</div>
```

В таком случае при выполнении присвоения свойству **innerText** значения New Text приведет к замене содержимого тега **p**.

Свойство outertext позволяет получить текст элемента, а также заменить сам элемент.

Кроме свойства **innerText** также существует свойство **outerText**, которое возвращает текст аналогично свойству **innerText**. А вот при установлении значения свойству **outerText** для элемента, это свойство заменяет не только содержимое, расположенное между открывающим и закрывающим тегом элемента, но и сам элемент.

Принцип работы свойства outerText рассмотрим также на примере тега **p,** содержащего текст Old Text.
Исходный код страницы:

```
<div>
   <p>Old text</p>
</div>
```

Выполнение p.innerText = "New Text!" приведёт к изменению кода:

```
<div>
   New Text!   <!--Изменился весь элемент-->
</div>
```

В случае присвоения свойству outerText значения New Text изменится не только содержимое тега p, но и сам парный тег.

Иногда нужно получить или заменить не текст внутри элемента, а его содержимое вместе с HTML-тегами.
Для этого существуют свойства innerHTML и outerHTML. Они похожи на innerText и outerText, однако возвращают только HTML-разметку.

Пример:

```
<b><i>Super</i>Text!</b>
```

```
let b = document.getElementsByTagName("b")[0];
console.log(b.innerHTML);//<i>Super</i>Text!
console.log(b.outerHTML);//<b><i>Super</i>Text!</b>
```

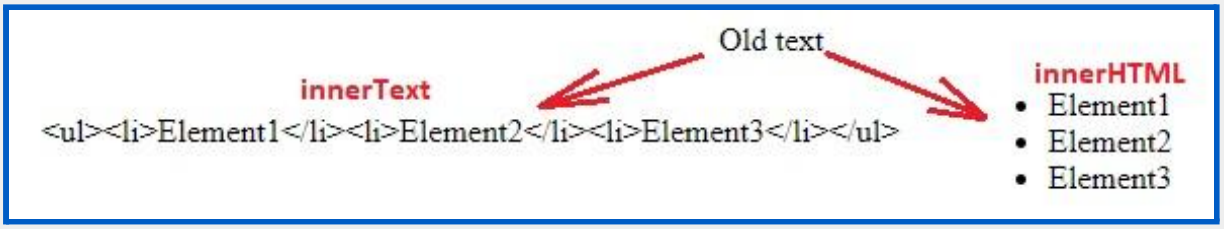
Различия между innerText и innerHTML
Допустим, у нас есть разметка:

```
<div>
  <p>Old text</p>
</div>
```

Что будет, если мы применим следующую операцию?
p.innerText = "
Element1
Element2
Element3
"

А если следующее присвоение?

p.innerHTML = "
Element1
Element2
Element3
"



Итоги:

inner*— что находится между тегами элемента;
outer*— содержимое элемента, включая теги;
*Text — текстовые значения;
*HTML — HTML-разметка.

2. Добавление HTML-разметки

Иногда нужно не заменить всё, что находится внутри HTML-элемента, а добавить какую-либо разметку. Для таких случаев используется метод **insertAdjacentHTML**().
Метод **insertAdjacentHTML**() более гибкий, чем свойство **innerHTML**. При присвоении свойства **innerHTML** у элемента меняется вся внутренняя разметка, а при использовании метода **insertAdjacentHTML**() разметка добавляется в указанную позицию.

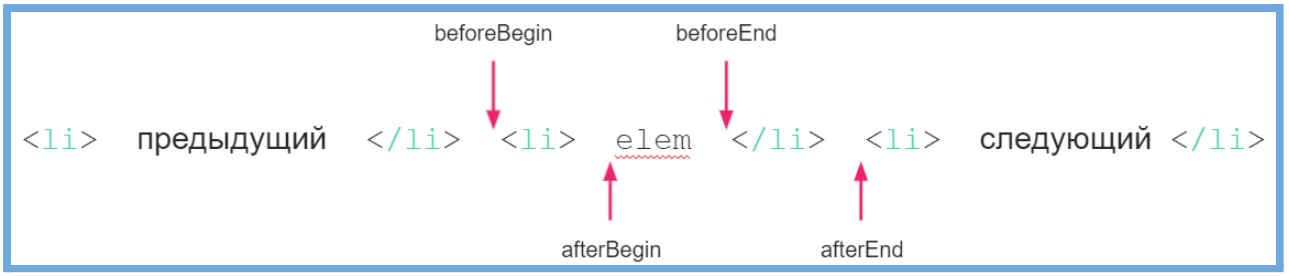
Синтаксис метода выглядит так:

```
elem.insertAdjacentHTML(where, html);
```

- **elem** — элемент, которому добавляется разметка;
- **where** — позиция, куда добавляется разметка;
- **html** — вставляемая разметка.

Значение **where** может быть одним из нескольких значений:

- **beforeBegin** — перед elem;
- **afterBegin** — внутрь elem, в самое начало;
- **beforeEnd** — внутрь elem, в конец;
- **afterEnd** — после elem.



Рассмотрим пример:

```
<ul class="exclusive">
  <li>Element 1</li>
  <li>Element 2</li>
  <li>Element 3</li>
</ul>
```

При выполнении следующих команд:

```
let li = document.getElementsByTagName("li");
li[0].insertAdjacentHTML("afterBegin","<a href='http://google.com'>Google</a>");
li[1].insertAdjacentHTML("beforeBegin","<li>Element 1.5</li>");
li[2].insertAdjacentHTML("beforeEnd","<a href='http://google.com'>Google</a>");
```

Получим следующую разметку:

```
<ul class="exclusive">
  <li>
    <a href="http://google.com">Google</a>
    Element 1
  </li>
  <li>Element 1.5</li>
  <li>Element 2</li>
  <li>
    Element 3
    <a href="http://google.com">Google</a>
  </li>
</ul>
```

Как и с **innerText/innerHTML**, для метода **insertAdjacentHTML**() есть схожие методы:

- **insertAdjacentText**() — добавляет текст в позицию другого элемента;
- **insertAdjacentElement**() — добавляет элемент в позицию другого элемента.

3. Создание HTML-элементов

Для манипулирования HTML-элементами рассмотрим следующие методы:

1. document.createElement
2. removeChild/remove
3. appendChild
4. insertBefore
5. replaceChild
6. cloneNode

Иногда нужно создать элемент. Например, для дальнейшей вставки его с помощью метода **insertAdjacentElement**().

Создание элемента страницы

Для создания элемента с нуля из JavaScript существует специальный метод объекта document — **createElement**().

```
let element = document.createElement('tag_name');
```

Созданным элементом можно пользоваться так же, как и любым элементом, полученным из **DOM**-дерева.
Явным отличием между созданным элементом и элементом из DOM-дерева является то, что элемент в **DOM**-дереве уже там находится и, скорее всего, отображается на странице.

Создания текстового элемента

Для создания текстового элемента существует метод **createTextNode**(), который похож на метод **createElement**() и отличается только тем, что создаёт не HTML-узел, а текстовый узел:

```
let element = document.createTextNode('simple text');
```

Добавления элемента-потомка

Созданные элементы можно добавлять внутрь других элементов через метод **appendChild**(), который необходимо вызывать на родительском элементе. Добавление происходит в конец родительского элемента. Если добавляемый элемент уже был на странице, то он будет перемещён в новое место, а в старом его больше не останется.

Для добавление текста есть схожий метод — **append**()

Если **appendChild**() добавляет в конец, то метод **insertBefore**() добавляет перед необходимым элементом, например:

parentElem.insertBefore(elem, nextSibling)

Вставляет elem в коллекцию детей parentElem перед элементом nextSibling.

Внимание. Если вторым аргументом указать null,то **insertBefore** сработает как **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 перемещён из предыдущего места.

```
let b = document.getElementsByTagName("b")[0];
let deletableElement = document.getElementsByClassName("deletable")[0];
b.appendChild(deletableElement);   // Добавляем HTML-элемент

b.append("Inserted text");   // Добавляем текст
```

Пример добавления элемента перед нужным элементом

Рассмотрим пример HTML-документа, в котором элемент b содержит три вложенных элемента: i, текстовую ноду и контейнер с классом deletable.

Исходный код страницы:

```
<b>
  <i>Super</i>
  Text!Inserted text
  <div class="deletable">deleted</div>
</b>
```

При добавлении:

b.insertBefore(document.createTextNode('Вставляемый_текст'),b.children[0]);

В таком случае использование метода **InsertBefore**, применяемого к элементу b, позволит нам добавить необходимую разметку или текстовый узел. При вызове метода **InsertBefore** необходимо передать два аргумента.

Первый из которых — это аргумент, содержащий элемент для вставки, а второй — указывающий на то, перед каким элементом необходимо добавить. Если вторым аргументом передать null, то вставка будет произведена в самый конец.

Результат выполнения метода:

```
<b>
  Вставляемый_текст
  <i>Super</i>
  Text!Inserted text
  <div class="deletable">deleted</div>
</b>
```

В нашем примере, в результате вызова метода **insertBefore** для элемента b, второй аргумент указывает на необходимость добавления элемента в качестве нулевого вложенного элемента.
Таким образом, надпись «Вставляемый_текст» отобразится на странице перед всеми ранее описанными элементами.

4. Удаление HTML-элементов

Иногда приходится удалять элементы из DOM-дерева. Для таких случаев существует метод **removeChild**().

Метод **removeChild**() вызывается на элементе, у которого необходимо удалить дочерний элемент. Аргументом функции является дочерний (удаляемый) элемент.

Вызов метода:

```
let deletableElement = document.getElementsByClassName("deletable")[0];
let childOfDeletable = deletableElement.childNodes[0];// элемент для удаления
deletableElement.removeChild(childOfDeletable);
```

В современных браузерах также есть более короткий в написании метод **remove**() без параметров, который вызывается у элемента для его удаления:

```
let deletableElement = document.getElementsByClassName("deletable")[0];
deletableElement.remove(); // удалить элемент
```

Можно удалять элементы из DOM-дерева, как с помощью родительского элемента, так и с помощью метода remove() без необходимости в родительском элементе.
Будьте внимательны при удалении элементов страницы, ведь исчезновение хотя бы одного действительно важного элемента может нарушить логику работы страницы в целом.

5. Замена и клонирование HTML-элементов

JavaScript предоставляет нам множество инструментов для работы с элементами. Например, у нас есть возможность поменять один HTML-элемент на другой с помощью метода **replaceChild**, который при размещении на родительском элементе позволяет нам заменить элемент-потомок.

```
replacedNode = parentNode.replaceChild(newChild, oldChild);
```

- parentNode — элемент, у которого будет заменяться дочерний элемент;
- oldChild — элемент, который будет убран;
- newChild — элемент, который будет добавлен на место прошлого;
- replacedNode — заменённый элемент (тоже самое, что и oldChild).

Исходный код страницы:

```
<b>
  <i>Super</i>
  Text!Inserted text
  <div class="deletable">deleted</div>
</b>
```

Разберём принцип работы метода на примере некоторой разметки. Элемент b в качестве вложенных элементов имеет тег i, текстовую ноду и контейнер с классом deletable. Тогда при изменении:

```
let div = document.createElement('div');
b.replaceChild(div, b.children[0]);
```

В таком случае вызов метода replaceChild с аргументом div, хранящим в себе созданный нами ранее контейнер с аргументом b.children по индексу 0... Получим:

```
<b>
  <div></div>
  Text!Inserted text
  <div class="deletable">deleted</div>
</b>
```

Приведёт к изменению исходного кода страницы. Таким образом, на месте элемента **i** появится созданный нами элемент **div**.

Однако использование метода replaceChild ограничивает нас необходимостью поиска родительского элемента той области, в которой нам необходимо осуществить преобразование.

Избежать подобных сложностей поможет метод **replaceWith**(), который вызывается на самом элементе для замены и заменяет исходный элемент на новый, переданный первым аргументом.

Замена самого элемента

Если нужно заменить сам элемент, то можно использовать метод **replaceWith**(), который используется на заменяемом элементе и в качестве аргумента принимает элемент, на который будет заменён.

Пример замены элемента

Исходный код страницы:

```
<b>
  <div></div>
  Text!Inserted text
  <div class="deletable">deleted</div>
</b>
```

При добавлении:

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

Результат выполнения кода:

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

Полностью удалит начальный элемент и разместит вместо него div-элемент.

Изменение DOM-дерева

Допустим, есть список, в который нужно добавить несколько элементов. Можно сделать цикл, который на каждой итерации будет добавлять элемент списка в список, или сгенерировать коллекцию элементов списка, а затем их все разом добавить. Есть ли разница в этих подходах, если да, то какая? Какой подход предпочтительней и почему?

Каждое изменение DOM-дерева приводит к перерисовке всей страницы. А это одна из самых продолжительных операций. Поэтому при большом количестве изменений DOM-дерева могут возникнуть проблемы с производительностью.

Клонирование элемента

Помимо такой операции, как замена, достаточно часто необходимо создать полную копию элемента либо вместе с потомками, либо отдельно от них. Для этого существует метод **cloneNode**, который принимает единственный аргумент. Если этот аргумент равен истине, то клонирование происходит в так называемом глубоком режиме, будут скопированы сам элемент и все его дочерние элементы. Например:

```
let clonedNode = node.cloneNode(deep);
```

Этот метод делает копию элемента **node**. Если аргумент deep присутствует и равен **true**, то копируются и дочерние элементы. Если **false**, то копируется только сам элемент:

```
let ul = document.getElementsByTagName("ul")[0];
  // Копирование с дочерними элементами
  console.log(ul.cloneNode(true));   // <ul class="exclusive">...</ul>
  // Копирование списка без элементов
  console.log(ul.cloneNode(false));   // <ul class="exclusive"></ul>
```

Таким образом, для замены элементов в современных браузерах проще использовать метод **replaceWith**, а вот клонирование может производиться как с дочерними элементами, так и без них — используйте тот вариант, который лучше подходит для поставленной задачи.

Итоги по теме

- Метод document.createElement служит для создания элементов;
- Методы removeChild() и remove() используются для удаления элемента или его потомков;
- Методы appendChild() и append() служат для добавления различного содержимого к элементам;
- Метод insertBefore() используется для добавления внутрь элемента перед другими;
- Методы replaceChild() и replace() служат для изменения или замены элемента;
- Метод cloneNode() служит для копирования элемента;
- Каждое изменение DOM-дерева ведёт к перерисовке всей веб-страницы.

Весь код, используемый в лекции, — листинг кода

Материалы, использованные при подготовке:

- Навигация по DOM-элементам
- Добавление и удаление узлов
- Мультивставка: insertAdjacentHTML и DocumentFragment
- Изменение страницы посредством DOM
- Изменение элементов с помощью DOM (youtube)
- Добавление и удаление элементов с помощью DOM (youtube)