Лекция 9

Изменение DOM
Атрибуты, классы и стили элементов

Манипуляции над DOM

Ранее мы разобрались с тем, как искать элементы и перемещаться по HTML-документу. Сегодня же мы поговорим о том, как при помощи JavaScript можно динамически изменять структуру этого документа. Достигается всё это с помощью встроенных методов объектов узлов (интерфейс **Node**). С их помощью мы можем перемещать, добавлять, изменять и удалять узлы DOM-дерева. Также существует возможность динамического изменения внешнего вида (CSS-стилей) узлов и даже значений их атрибутов.

Помимо изменения уже существующей структуры HTML-документа мы также можем добавлять динамически созданные элементы. При этом до момента непосредственного “вживления” элемента в DOM-дерево мы можем добавить в него какое-то содержимое, задать значения его атрибутам и даже стили. Делается всё это, конечно же, с помощью встроенных методов и свойств самих узлов или объекта **document** (частный случай **Node**).

Важно отметить, что процедуры изменения структуры DOM являются довольно трудоёмкими. Поэтому стоит всегда стремиться к минимизации операций такого рода, а в случаях, когда производительность в приложении критична, не бояться пользоваться дополнительными способами оптимизации.

Создание узлов

Начнём, пожалуй, с самого интересного. У объекта **document** есть встроенный метод **createElement(tagName, ?options)**, который нужен, очевидно, для создания **элементов**. Данный метод принимает два аргумента**(1)**, но в нативном JavaScript в большинстве случаев используется только первый из них. В качестве результата мы получаем специальный объект нашего элемента.

Например:

```
1   const divElement = document.createElement('div');
2   const unorderedListElement = document.createElement('ul');
```

На примере выше мы создали блочный элемент и элемент неупорядоченного списка. При вставке в документ они будут иметь вид ** ** и **
** соответственно. Таким образом мы можем создавать любые **элементы**. Однако, когда нам нужно создать текстовый узел или комментарий, можно также воспользоваться встроенными методами объекта **document**: **document.createTextNode(data)** и **document.createComment(data)**. Названия этих методов говорят сами за себя. Первый даёт нам возможность создавать текстовые узлы, а второй – комментарии.

(1) Второй аргумент options в методе createElement содержит всего одно поле is и нужен только в случаях использования пользовательских веб-компонентов.

Вставка узлов в документ

Как правило, мы создаём новые узлы и элементы, чтобы позже вставить их в наш документ. Автоматически при создании в DOM-дерево они не вставляются. Однако для этого существует много разных способов, а самые современные из них следующие:

- **node.append(…nodes** или **strings)** - вставляет переданные узлы/строки после всех дочерних узлов **node**;
- **node.prepend(…nodes** или **strings)** - вставляет переданные узлы/строки перед всеми дочерними узлами **node**;
- **node.before(…nodes** или **strings)** - вставляет переданные узлы/строки прямо перед узлом **node**;
- **node.after(…nodes** или **strings)** - вставляет переданные узлы/строки сразу после узла **node**;
- **node.replaceWith(…nodes** или **strings)** - вставляет переданные узлы/строки вместо узла **node**.

Рассмотрим их работу на примере.

Пример вставки узлов в документ

```
<div id="card">
    <h1>Header</h1>
    <p>Some text…</p>
    <!-- some important comment -->
</div>
```

nodes-insertion.js

nodes-insertion.js > ...

```
const card = document.getElementById('card');
const anotherImportantComment = document.createComment('another important comment');
const hintElement = document.createElement('h4');
card.append(anotherImportantComment, 'and some text');
card.prepend(hintElement);
card.before('before card...');
card.after('after card...');
```

Результат запуска кода из предыдущего примера

```
before card...
<div id="card">
    <h4></h4>
    <h1>Header</h1>
    <p>Some text…</p>
    <!-- some important comment -->
    <!--another important comment-->
    and some text
</div>
after card...
```

nodes-insertion.js

nodes-insertion.js > ...

```
const card = document.getElementById('card');
const anotherImportantComment = document.createComment('another important comment');
const hintElement = document.createElement('h4');
card.append(anotherImportantComment, 'and some text');
card.prepend(hintElement);
card.before('before card...');
card.after('after card...');
```

Вставка фрагментов HTML

Мы рассмотрели, как мы можем вставлять узлы и обычные строки в наш документ, но что если мы захотим в качестве значения для вставки передать, к примеру, следующую строку:

‘<h1>Unexpected!</h1>’

“Сломает” ли это структуру нашего документа? Появится ли из-за этого новый HTML-элемент? Нет. И нет. Методы, описанные ранее, вставляют строки в DOM безопасным способом. Это нужно, чтобы оградить документ от вмешательства в его структуру. Также безопасно можно вставить или изменить текст в документе с помощью свойства **node.textContent**. Однако тут стоит помнить, что новое значение для данного свойства заменит собой весь существовавший до этого контент узла.

Когда вам всё же необходимо вставить фрагмент HTML, самым простым способом будет использовать свойство **element.innerHTML**. При изменении этого свойства, строка, переданная в качестве нового значения, прежде чем заменить собой предыдущий контент элемента, преобразуется к HTML. Однако пользоваться этим свойством нужно аккуратно и всегда проверять, что идёт на вход, чтобы предотвратить XSS атаки. Проверять можно как вручную (что не всегда лучший вариант, т.к. можно что-то упустить), так и воспользоваться каким-нибудь готовым решением.

Ещё методы для вставки HTML, элементов и текста

Кроме свойства **element.innerHTML**, элементы имеют метод **element.insertAdjacentHTML(where, html)**, помогающий достичь примерно того же самого. Метод принимает два обязательных аргумента: специальную строку, указывающую куда именно нужно по отношению к **element** вставить новый фрагмент HTML, и непосредственно сам HTML-код. Возможные значения аргумента **where**:

- “**beforebegin**" — вставить HTML-код перед элементом (похож на **node.before**);
- “**afterbegin**” — вставить HTML-код перед дочерними узлами элемента (похож на **node.prepend**);
- “**beforeend**” — вставить HTML-код после дочерних узлов элемента (похож на **node.append**);
- “**afterend**” — вставить HTML-код после элемента (похож на **node.after**).

Кроме этого существуют ещё два очень похожих на метода:

- ~~**element.insertAdjacentText(where, text)**~~ – безопасно вставить текст по расположению **where**;
- ~~**element.insertAdjacentElement(where, element)**~~ – вставить элемент по расположению **where**.

Однако используются они крайне редко.

Удаление узлов

Удаление узлов происходит очень просто. Для этого лучше всего воспользоваться методом **node.remove(),** мы его вызываем прямо на удаляемом узле.

Кроме явного удаления узлов можно прибегнуть и к другим способам. Как говорилось ранее, при изменении значений свойств **node.textContent** и **element.innerHTML** старое содержимое узла перезаписывается новым. Таким образом также происходит удаление узлов.

Клонирование узлов

Создать клон узла очень просто. Нужно просто воспользоваться встроенным методом **node.cloneNode(?deep)**. Данный метод принимает один необязательный параметр, который указывает, стоит ли производить глубокое клонирование объекта. Если параметр не указан или равен **false**, то клон будет представлять из себя лишь сам узел без его предков.

Во время перемещения узлов они не клонируются, а перемещаются сами. То есть фактически элементы при попытке добавления их в разные части документа каждый раз “перетаскиваются” с одного места на другое. Чтобы сделать несколько одинаковых элементов их каждый раз нужно клонировать или создавать заново. Например:

```
1   const sectionElement = document.createElement('section');
2   const divElement = document.createElement('div');
3   sectionElement.append(divElement);
4   sectionElement.append(divElement);
5   sectionElement.append(divElement);
```

После выполнения кода из примера элемент **<section></section>** будет содержать всего один элемент ** ** внутри себя.

Устаревшее

До прихода современных методов вставки, удаления и замены узлов таких как: **append**, **before**, **replaceWith** и т.д., — в обиходе был ряд других методов. Сегодня они уже почти не используются. Главные причины этому: меньшая гибкость и большая трудоемкость в использовании. Тем не менее, мы не можем их не упомянуть, так как их всё ещё можно обнаружить в старом коде.

- **node.appendChild(newNode)** – вставляет **newNode** после всех дочерних узлов узла **node**;
- **node.removeChild(nodeToRemove)** – удаляет **nodeToRemove** из дочерних узлов **node**;
- **node.insertBefore(newNode, nextSibling)** – вставляет **newNode** перед **nextSibling** внутри узла **node**;
- **node.replaceChild(newNode, oldChildNode)** – заменяет **oldChildNode** на **newNode** среди дочерних узлов **node**.

Можно заметить, что раньше для вставки, замены и даже удаления элементов приходилось всегда производить операцию через родительский элемент.

** Все вышеперечисленные методы, в отличие от их современных аналогов, возвращают вставленный/удалённый узел.

Кроме этих методов существует ещё один более «древний» – document.write(html). Сегодня у него

Работа с отображением элементов

Довольно часто в рамках разработки приложений нам, основываясь на каком-то состоянии, нужно отобразить один и тот же элемент по-разному. Или же просто динамически изменить внешний вид как всего элемента, так и отдельной его части. Всего этого, естественно, можно достичь с помощью JavaScript и встроенных свойств и методов элементов.

Изменить внешний вид элемента с помощью JavaScript можно тремя основными способами:

1. Изменить атрибуты элемента;
2. Изменить набор классов элемента;
3. Изменить стили элемента.

Очень важное замечание!

В ситуациях, когда стоит вопрос изменить внешний вид элемента средствами JavaScript или CSS, всегда отдавайте предпочтение именно CSS! Используйте JavaScript только тогда, когда возможностей CSS в вашем случае недостаточно.

Работа с атрибутами элементов

Для работы с атрибутами интерфейс **Element** предоставляет нам целый ряд встроенных методов:

- **element.hasAttribute(attrName)** — проверяет, есть ли у элемента атрибут с таким именем;
- **element.getAttribute(attrName)** — получает значение атрибута элемента;
- **element.setAttribute(attrName, value)** — устанавливает значение атрибута;
- **element.removeAttribute(attrName)** — удаляет атрибут элемента.

Кроме этого есть свойство **element.attributes**, которое возвращает весь список атрибутов элементов в виде коллекции (перебираемого псевдомассива). У каждого атрибута из этой коллекции можно получить имя (поле **name**) и значение (поле **value**).

В большинстве случаев напрямую с атрибутами работать не приходится. Самые распространённые случаи работы с ними следующие: изменение состояние поля для ввода на **disabled** или **readonly**, отображение состояния **checked** в чекбоксах и т.п.

Работа с классами элементов

Для работы с классами интерфейс **Element** предоставляет нам целых два встроенных свойства: **className** (строка, содержащая все классы элемента, разделённые пробелом) и **classList** (коллекция (перебираемый псевдомассив) всех названий классов элемента в виде строк). В подавляющем большинстве случаев предпочтительным способом для работы с классами будет именно второй вариант. Помимо более удобного формата представления набора классов (в виде коллекции, а не строки со значениями через пробел), свойство **classList** содержит в себе целый ряд вспомогательных методов:

- **element.classList.add('class')** — добавить класс с названием **class** элементу **element**;
- **element.classList.remove('class')** — удалить класс элемента **element** с названием **class**;
- **element.classList.toggle('class')** — если класс с названием **class** у элемента **element** уже есть то удалить его, если нет — добавить;
- **element.classList.contains('class')** — определить, есть ли класс с названием **class** у элемента **element**.

Работа со стилями элементов

Любому HTML-элементу можно добавить атрибут **style**. Получить значение этого атрибута можно как обычным способом (с помощью метода **getAttribute**), так и напрямую обратившись к свойству элемента под таким же названием. В JavaScript нам, естественно, удобнее работать именно со свойством. Тем более данное свойство содержит в себе не просто строковое представление наших стилей (как это делает атрибут), а целый объект. Изменяя данный объект мы тут же меняем внешний вид нашего элемента. Например, присвоение **element.style.fontSize = '24px'(1)** задаст значение параметру размера шрифта в 24px. Здесь важно обратить внимание на то, что стандартные названия CSS-свойств с дефисами (в стиле *kebab-case*) превращаются в названия в стиле *camelCase*. При этом названия с браузерными префиксами так и вовсе пишутся в стиле *PascalCase* (например, **MozBorderRadius**, **WebkitAnimation** и т.п.).

Для очищения какого-то из ранее записанных в объект **style** стилей понадобится присвоить ему пустую строку или значение **null**. Иначе свойство попросту не удалится.

(1) Обратите внимание, что CSS-свойства, ожидающие значения с размерностями, не будут работать в случае передачи обычного числа или строки, содержащей число без размерности.

Задание сразу нескольких стилей

Задать элементу сразу несколько стилей можно с помощью специального свойства **element.style.cssText**. Данное свойство должно принимать на вход строку, содержащую валидный CSS-код формата: “**свойство 1: значение; свойство 2: значение…**”. Таким образом в эту строку можно вписать все необходимые стили в привычном для нас виде, да ещё и за один подход. Однако тут стоит понимать, что при изменении значения данного свойства ВСЕ ранее заданные через объект **style** стили затираются (даже если новые стили не совпадают со старыми). Поэтому лучше всего свойство **element.style.cssText** применять только тогда, когда мы точно знаем, что ничего нужного у нас не затрётся (например, во время создания элемента).

Получение стилей элемента

Чтобы получить значение стилей вашего элемента не хватит просто обратиться к свойству **style** и по имени (например, **backgroundColor**) взять необходимое значение. В лучшем случае мы просто получим значение записанное туда ранее, а в худшем – пустую строку. Происходит так потому, что данное свойство никак не связано со стилями, которые мы указываем обычным способом в тегах **<style></style>** (как правило в рамках тега **head** нашего HTML-документа) или же в отдельных CSS-файлах.

Для получения реальных значений стилей, с учётом нашего CSS-кода, нужно воспользоваться методом **window.getComputedStyle(elem, ?pseudo)(1)**. Он вернёт нам огромный объект, содержащий все стили элемента в виде пар “ключ (название свойства): значение”. При этом в полученном объекте всегда лучше обращаться именно с конкретными значениями по типу **marginTop** или **paddingLeft**, т.к. их обобщённые аналоги (например, **margin**, **padding**) могут вести себя в разных браузерах по-разному. Также стоит отметить, что **window.getComputedStyle(elem)** возвращает объект, ключи свойств которого написаны в стиле *camelCase*.

(1) Второй параметр метода нужен для получения стилей псевдо-элементов. Пример: “:after”.