Front-end

JavaScript

JS. BOM, DOM, API

- DOM

JS. BOM, DOM, API

Модификации DOM

document.createElement(tag)

Создаёт новый элемент с заданным тегом:

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

div.className = "alert";

div.innerHTML = "<strong>Всем привет!</strong> Вы прочитали важное сообщение.";
```

JS. BOM, DOM, API

Модификации DOM

Методы 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.

JS. BOM, DOM, API

Модификации DOM

Методы вставки: insertAdjacentHTML/Text/Element

Синтаксис

elem.insertAdjacentHTML (position, text) - HTML-строка, которая будет вставлена именно «как HTML»

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.

JS. BOM, DOM, API

Модификации DOM

Методы вставки: insertAdjacentHTML/Text/Element

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 (до самого element (до открывающего тега))

"afterbegin" – вставить html в начало elem (сразу после открывающего тега element (перед первым потомком))

"beforeend" – вставить html в конец elem (сразу перед закрывающим тегом element (после последнего потомка))

"afterend" – вставить html непосредственно после elem (после element (после закрывающего тега))

JS. BOM, DOM, API

Модификации DOM

Удаление узлов

Для удаления узла есть методы node.remove()

JS. BOM, DOM, API

Модификации DOM

Клонирование узлов: cloneNode

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.

JS. BOM, DOM, API

Модификации DOM

Устаревшие методы:

```
parent.appendChild(node);
parent.insertBefore(node, nextSibling);
parent.removeChild(node);
parent.replaceChild(newElem, node);
```

Все возвращают node.

JS. BOM, DOM, API

Модификации DOM

Атрибуты и свойства

Когда браузер загружает страницу, он «читает» (также говорят: «парсит») HTML и генерирует из него DOM-объекты. Для узлов-элементов большинство стандартных HTML-атрибутов автоматически становятся свойствами DOM-объектов.

Например, для такого тега <body id="page"> у DOM-объекта будет такое свойство body.id="page".

JS. BOM, DOM, API

Модификации DOM

В HTML у тегов могут быть атрибуты. Когда браузер парсит HTML, чтобы создать DOM-объекты для тегов, он распознаёт стандартные атрибуты и создаёт DOM-свойства для них.

Таким образом, когда у элемента есть id или другой стандартный атрибут, создаётся соответствующее свойство. Но этого не происходит, если атрибут нестандартный.

<body id="test" something="non-standard">

JS. BOM, DOM, API

Модификации DOM

Таким образом, для нестандартных атрибутов не будет соответствующих DOM-свойств. Есть ли способ получить такие атрибуты?

Конечно. Все атрибуты доступны с помощью следующих методов:

elem.hasAttribute(name) – проверяет наличие атрибута.

elem.getAttribute(name) – получает значение атрибута.

elem.setAttribute(name, value) – устанавливает значение атрибута.

elem.removeAttribute(name) – удаляет атрибут.

JS. BOM, DOM, API

Модификации DOM

Эти методы работают именно с тем, что написано в HTML.

Кроме этого, получить все атрибуты элемента можно с помощью свойства elem.attributes: коллекция объектов, которая принадлежит ко встроенному классу Attr со свойствами name и value.

JS. BOM, DOM, API

Модификации DOM

Нестандартные атрибуты, dataset

При написании HTML мы используем много стандартных атрибутов. Но что насчёт 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.

JS. BOM, DOM, API

Модификации DOM

Также они могут быть использованы, чтобы стилизовать элементы.

Например, здесь для состояния заказа используется атрибут order-state:

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

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

.order[order-state="canceled"] {
  color: red;
}
```

JS. BOM, DOM, API

Модификации DOM

Почему атрибут может быть предпочтительнее таких классов, как .order-state-new, .order-state-pending, order-state-canceled?

Это потому, что атрибутом удобнее управлять. Состояние может быть изменено достаточно просто:

```
// немного проще, чем удаление старого/добавление нового класса

div.setAttribute('order-state', 'canceled');
```

JS. BOM, DOM, API

Модификации DOM

Но с пользовательскими атрибутами могут возникнуть проблемы. Что если мы используем нестандартный атрибут для наших целей, а позже он появится в стандарте и будет выполнять какую-то функцию?

Чтобы избежать конфликтов, существуют атрибуты вида data-*.

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

Например, если у elem есть атрибут "data-about", то обратиться к нему можно как elem.dataset.about

```
<body data-about="Elephants">

alert(document.body.dataset.about); // Elephants
```

JS. BOM, DOM, API

Модификации DOM

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

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

JS. BOM, DOM, API

Коллекция — сущность, которая похожа на массив объектов, но при этом им не является, на самом деле это набор DOM-элементов. Стоит учесть, что фактически разные методы возвращают разные коллекции:

HTMLCollection — коллекция непосредственно HTML-элементов.

NodeList — коллекция узлов, более абстрактное понятие. Например, в DOM-дереве есть не только узлы-элементы, но также текстовые узлы, узлы-комментарии и другие, поэтому NodeList может содержать другие типы узлов.

При работе с DOM-элементами тип коллекции значительной роли не играет, поэтому для удобства будем рассматривать их как одну сущность — коллекцию.

Во время работы с коллекциями можно столкнуться с поведением, которое покажется странным, если не знать один нюанс — они бывают живыми (динамическими) и неживыми (статическими). То есть либо реагируют на любое изменение DOM, либо нет. Вид коллекции зависит от способа, с помощью которого она получена.

JS. BOM, DOM, API

Разница между живыми и неживыми коллекциями

Допустим, в разметке есть список книг:

```
<ul class="books">
  <li class="book book--one"></li>
  <li class="book book--two"></li>
  <li class="book book--three"></li>
</ul>
```

JS. BOM, DOM, API

Для взаимодействия с книгами получим с помощью JavaScript список всех нужных элементов. Чтобы в дальнейшем увидеть разницу между видами коллекций, используем разные способы поиска элементов — свойство children и метод querySelectorAll:

```
let booksList = document.querySelector(`.books`);

let liveBooks = booksList.children;


// Выведем все дочерние элементы списка .books

console.log(liveBooks);

let notLiveBooks = document.querySelectorAll(`.book`);


// Выведем коллекцию, содержащую все элементы с классом book

console.log(notLiveBooks);
```

JS. BOM, DOM, API

Пока никакой разницы не видно. В обоих случаях console.log выведет одни и те же элементы. Но что, если попробовать удалить из DOM одну из книг?

```
let booksList = document.querySelector(`.books`);
let liveBooks = booksList.children;

// Удалим первую книгу
liveBooks[0].remove();
// Получим 2
console.log(liveBooks.length);
// Получим элемент book--two, который теперь стал первым в коллекции
console.log(liveBooks[0]);
let notLiveBooks = document.querySelectorAll(`.book`);

// Удалим первую книгу
notLiveBooks[0].remove();
// Получим 3
console.log(notLiveBooks.length);
// Получим ссылку на удалённый элемент book--one
console.log(notLiveBooks[0]);
```

JS. BOM, DOM, API

В первом случае информация о количестве элементов внутри коллекции автоматически обновилась после удаления одного элемента из DOM — эта коллекция живая. Во втором случае в переменной notLiveBooks хранится первоначальное состояние коллекции, которое было актуально на момент вызова метода querySelectorAll. Эта коллекция неживая, она ничего не знает об изменении DOM. При этом доступна ссылка на удалённый элемент book--one, которого фактически больше нет в DOM.

JS. BOM, DOM, API

Стили и классы

До того, как начнёте изучать способы работы со стилями и классами в JavaScript, есть одно важное правило. Надеемся, это достаточно очевидно, но мы всё равно должны об этом упомянуть.

Как правило, существует два способа задания стилей для элемента:

Создать класс в CSS и использовать его:

Писать стили непосредственно в атрибуте style: .

JavaScript может менять и классы, и свойство style.

Классы – всегда предпочтительный вариант по сравнению со style. Мы должны манипулировать свойством style только в том случае, если классы «не могут справиться».

Например, использование style является приемлемым, если мы вычисляем координаты элемента динамически и хотим установить их из JavaScript

JS. BOM, DOM, API

Для классов было введено 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

JS. BOM, DOM, API

Мы можем работать как со строкой полного класса, используя className, так и с отдельными классами, используя classList. Выбираем тот вариант, который нам удобнее.

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

JS. BOM, DOM, API

Свойство elem.style – это объект, который соответствует тому, что написано в атрибуте "style". Установка стиля elem.style.width="100px" работает так же, как наличие в атрибуте style строки width:100px.

Для свойства из нескольких слов используется camelCase

JS. BOM, DOM, API

Не забудьте добавить к значениям единицы измерения.

Например, мы должны устанавливать 10px, а не просто 10 в свойство elem.style.top. Иначе это не сработает

JS. BOM, DOM, API

Событие – это сигнал от браузера о том, что что-то произошло. Все DOM-узлы подают такие сигналы (хотя события бывают и не только в DOM).

Вот список самых часто используемых DOM-событий, пока просто для ознакомления:

События мыши:

- click – происходит, когда кликнули на элемент левой кнопкой мыши (на устройствах с сенсорными экранами оно происходит при касании).
- contextmenu – происходит, когда кликнули на элемент правой кнопкой мыши.
- mouseover / mouseout – когда мышь наводится на / покидает элемент.
- mousedown / mouseup – когда нажали / отжали кнопку мыши на элементе.
- mousemove – при движении мыши.

JS. BOM, DOM, API

События на элементах управления:

- submit – пользователь отправил форму <form>.
- focus – пользователь фокусируется на элементе, например нажимает на <input>.

Клавиатурные события:

- keydown и keyup – когда пользователь нажимает / отпускает клавишу.

События документа:

- DOMContentLoaded – когда HTML загружен и обработан, DOM документа полностью построен и доступен.

CSS events:

- transitionend – когда CSS-анимация завершена.

JS. BOM, DOM, API

Обработчики событий

Событию можно назначить обработчик, то есть функцию, которая сработает, как только событие произошло.

Именно благодаря обработчикам JavaScript-код может реагировать на действия пользователя.

Есть несколько способов назначить событию обработчик.

Обработчик может быть назначен прямо в разметке, в атрибуте, который называется on<событие>.

Например, чтобы назначить обработчик события click на элементе input, можно использовать атрибут onclick, вот так:

```
<input value="Нажми меня" onclick="alert('Клик!')" type="button">
```

JS. BOM, DOM, API

Обработчики событий

При клике мышкой на кнопке выполнится код, указанный в атрибуте onclick.

Обратите внимание, для содержимого атрибута onclick используются одинарные кавычки, так как сам атрибут находится в двойных. Если мы забудем об этом и поставим двойные кавычки внутри атрибута, вот так: onclick="alert("Click!")", код не будет работать.

Атрибут HTML-тега – не самое удобное место для написания большого количества кода, поэтому лучше создать отдельную JavaScript-функцию и вызвать её там.

JS. BOM, DOM, API

Обработчики событий

Использование свойства DOM-объекта

Можно назначать обработчик, используя свойство DOM-элемента on<событие>.

К примеру, elem.onclick:

```
<input id="elem" type="button" value="Нажми меня!">

elem.onclick = function() {
  alert('Спасибо');
 };
```

JS. BOM, DOM, API

Обработчики событий

Так как у элемента DOM может быть только одно свойство с именем onclick, то назначить более одного обработчика так нельзя.

Обработчиком можно назначить и уже существующую функцию

```
function sayThanks() {
  alert('Спасибо!');
}

elem.onclick = sayThanks;
```

JS. BOM, DOM, API

Обработчики событий

Доступ к элементу через this

Внутри обработчика события this ссылается на текущий элемент, то есть на тот, на котором, как говорят, «висит» (т.е. назначен) обработчик.

В коде ниже button выводит своё содержимое, используя this.innerHTML:

```
<button onclick="alert(this.innerHTML)">Нажми меня</button>
```

JS. BOM, DOM, API

Обработчики событий

addEventListener

Фундаментальный недостаток описанных выше способов назначения обработчика – невозможность повесить несколько обработчиков на одно событие.

Например, одна часть кода хочет при клике на кнопку делать её подсвеченной, а другая – выдавать сообщение.

Мы хотим назначить два обработчика для этого. Но новое DOM-свойство перезапишет предыдущее:

```
input.onclick = function() { alert(1); }
// ...
input.onclick = function() { alert(2); } // заменит предыдущий обработчик
```

JS. BOM, DOM, API

addEventListener

Разработчики стандартов достаточно давно это поняли и предложили альтернативный способ назначения обработчиков при помощи специальных методов addEventListener и removeEventListener. Они свободны от указанного недостатка.

Синтаксис добавления обработчика:

element.addEventListener(event, handler, [options]);

event - Имя события, например "click".

handler - Ссылка на функцию-обработчик.

options - Дополнительный объект со свойствами:

once: если true, тогда обработчик будет автоматически удалён после выполнения.

capture: фаза, на которой должен сработать обработчик. Так исторически сложилось, что options может быть false/true, это то же самое, что {capture: false/true}.

passive: если true, то указывает, что обработчик никогда не вызовет preventDefault()

Для удаления обработчика следует использовать removeEventListener:

element.removeEventListener(event, handler, [options]);

JS. BOM, DOM, API

addEventListener

Для удаления обработчика следует использовать removeEventListener:

```
element.removeEventListener(event, handler, [options]);
```

Удаление требует именно ту же функцию

Для удаления нужно передать именно ту функцию-обработчик которая была назначена.

Вот так не сработает:

```
elem.addEventListener( "click" , () => alert('Спасибо!'));
// ....
elem.removeEventListener( "click", () => alert('Спасибо!'));
```

JS. BOM, DOM, API

addEventListener

Если функцию обработчик не сохранить где-либо, мы не сможем её удалить. Нет метода, который позволяет получить из элемента обработчики событий, назначенные через addEventListener

Метод addEventListener позволяет добавлять несколько обработчиков на одно событие одного элемента

JS. BOM, DOM, API

addEventListener

Объект события

Чтобы хорошо обработать событие, могут понадобиться детали того, что произошло. Не просто «клик» или «нажатие клавиши», а также – какие координаты указателя мыши, какая клавиша нажата и так далее.

Когда происходит событие, браузер создаёт объект события, записывает в него детали и передаёт его в качестве аргумента функции-обработчику.

JS. BOM, DOM, API

addEventListener

Объект события

Некоторые свойства объекта event:

event.type - тип события, в данном случае "click".

event.currentTarget

Элемент, на котором сработал обработчик. Значение – обычно такое же, как и у this, но если обработчик является функцией-стрелкой или при помощи bind привязан другой объект в качестве this, то мы можем получить элемент из event.currentTarget.

event.clientX / event.clientY

Координаты курсора в момент клика относительно окна, для событий мыши.