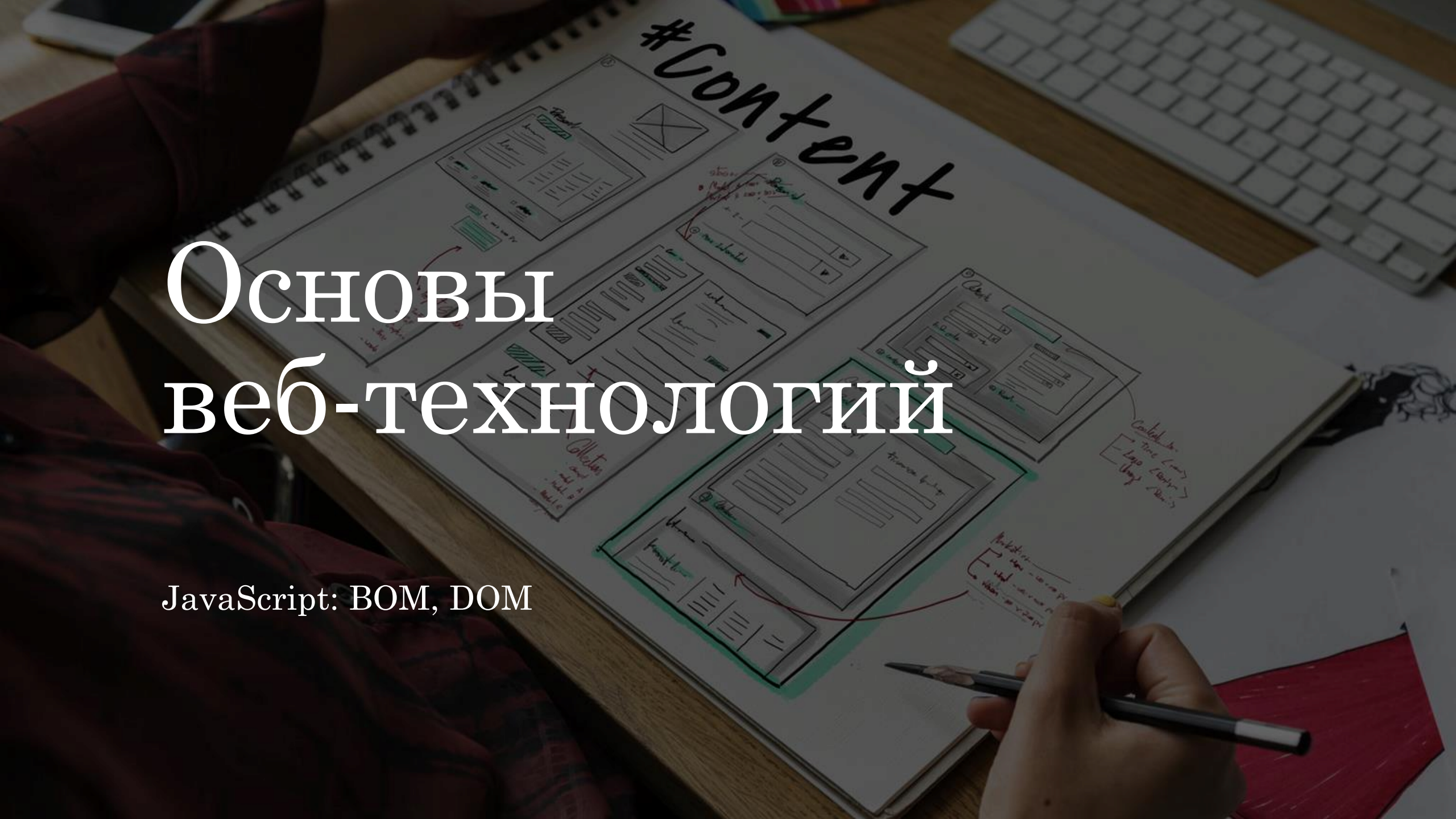


Основы веб-технологий

JavaScript: BOM, DOM

Browser Object Model

BOM (Browser Object Model, Объектная модель браузера) — межплатформенный, независимый от языка интерфейс для работы с окном браузера. Содержит набор свойств и методов, позволяющих получить доступ непосредственно к текущей вкладке и ряду функций браузера. Включает объект работы с историей, местоположением и другое.

Веб-страницы бывают статическими и динамическими, последние отличаются тем, что в них используются сценарии (программы) на языке JavaScript.

В сценариях JavaScript браузер веб-разработчику предоставляет множество "готовых" объектов, с помощью которых он может взаимодействовать с элементами веб-страницы и самим браузером. В совокупности все эти объекты составляют объектную модель браузера (BOM – Browser Object Model).

На самом верху этой модели находится глобальный объект window. Он представляет собой одно из окон или вкладку браузера с его панелями инструментов, меню, строкой состояния, HTML страницей и другими объектами. Доступ к этим различным объектам окна браузера осуществляется с помощью следующих основных объектов: navigator, history, location, screen, document и т.д. Так как данные объекты являются дочерними по отношению к объекту window, то обращение к ним происходит как к свойствам объекта window.

Browser Object Model

```
> window
◂ ▶ Window {0: Window, window: Window, self: Window, document: document, name: '', location: Location, …}
> window.location
◂ ▶ Location {ancestorOrigins: DOMStringList, href: 'chrome://new-tab-page/', origin: 'chrome://new-tab-page', protocol: 'chrome:', ho
    st: 'new-tab-page', …}
> window.screen
◂ ▼ Screen {availWidth: 1920, availHeight: 1040, width: 1920, height: 1080, colorDepth: 24, …} i
      availHeight: 1040
      availLeft: 0
      availTop: 0
      availWidth: 1920
      colorDepth: 24
      height: 1080
      isExtended: false
      onchange: null
    ▶ orientation: ScreenOrientation {angle: 0, type: 'landscape-primary', onchange: null}
      pixelDepth: 24
      width: 1920
    ▶ [[Prototype]]: Screen
>
```

Browser Object Model

Объектная модель браузера не стандартизована в спецификации, и поэтому её реализация может отличаться в разных браузерах.

Основные объекты Browser Object Model: window, navigator, history, location, screen, document.

`window` – самый главный объект в браузере, который отвечает за одно из окон (вкладок) браузера. Он является корнем иерархии всех объектов доступных веб-разработчику в сценариях JavaScript. Объект `window` кроме глобальных объектов (`document, screen, location, navigator` и др.) имеет собственные свойства и методы.

Если в браузере открыть несколько вкладок (окон), то браузером будет создано столько объектов window, сколько открыто этих вкладок (окон). Т.е. каждый раз открывая вкладку (окно), браузер создаёт новый объект window связанный с этой вкладкой (окном).

Browser Object Model

Объект navigator

navigator – информационный объект с помощью которого Вы можете получить различные данные, содержащиеся в браузере:

- информацию о самом браузере в виде строки (User Agent);
- внутреннее "кодовое" и официальное имя браузера;
- версию и язык браузера;
- информацию о сетевом соединении и местоположении устройства пользователя;
- информацию об операционной системе и многое другое.

Объект history

history – объект, который позволяет получить историю переходов пользователя по ссылкам в пределах одного окна (вкладки) браузера. Данный объект отвечает за кнопки forward (вперёд) и back (назад). С помощью методов объекта history можно имитировать нажатие на эти кнопки, а также переходить на определённое количество ссылок в истории вперёд или назад.

Browser Object Model

Объект location

location – объект, который отвечает за адресную строку браузера. Данный объект содержит свойства и методы, которые позволяют: получить текущий адрес страницы браузера, перейти по указанному URL, перезагрузить страницу и т.п.

Объект screen

screen – объект, который предоставляет информацию об экране пользователя: разрешение экрана, максимальную ширину и высоту, которую может иметь окно браузера, глубина цвета и т.д.

Объект document

document – HTML документ, загруженный в окно (вкладку) браузера. Он является корневым узлом HTML документа и "владельцем" всех других узлов: элементов, текстовых узлов, атрибутов и комментариев. Объект document содержит свойства и методы для доступа ко всем узловым объектам. document как и другие объекты, является частью объекта window и, следовательно, он может быть доступен как window.document.

Document Object Model

DOM (Document Object Model, Объектная модель документа) — межплатформенный, независимый от языка интерфейс для работы с HTML-документом. Содержит набор свойств и методов позволяющих искать, создавать и удалять элементы, реагировать на действия пользователя и другое.

DOM – это объектная модель документа, которую браузер создаёт в памяти компьютера на основании HTML-кода.

DOM выполняет две роли: является объектным представлением HTML-документа и действует как интерфейс, соединяющий страницу с языком программирования, например JavaScript.

Каждый элемент в документе, весь документ в целом, заголовок, ссылка, абзац — это части DOM этого документа, поэтому все они могут изменяться с помощью JavaScript.

Все объекты и методы, которые предоставляет браузер описаны в спецификации HTML DOM API, поддерживаемой W3C. С помощью них мы можем читать и изменять документ в памяти браузера.

Document Object Model

Все, что есть в `HTML`, даже комментарии, является частью DOM.

Директива `<!DOCTYPE...>`, тоже является DOM-узлом. Она находится в дереве DOM прямо перед `<html>`.

Объект `document`, представляющий весь документ, формально является DOM-узлом.

Основные типы узлов DOM:

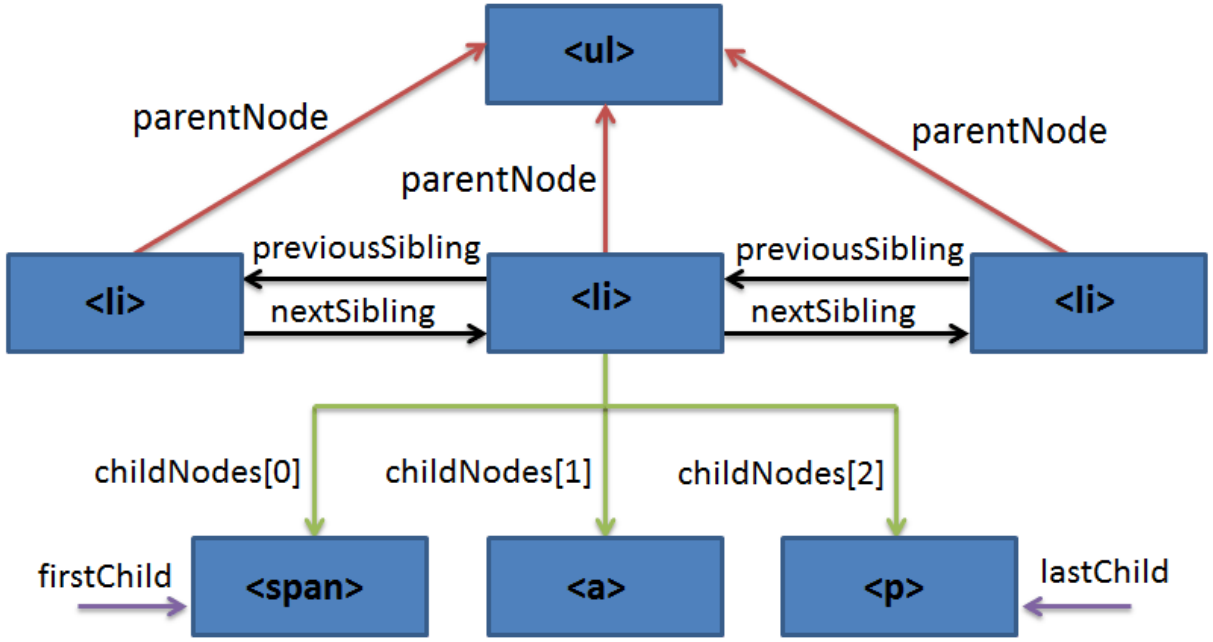
1. document – «входная точка» в DOM;
2. узлы-элементы – HTML-теги, основные строительные блоки;
3. текстовые узлы – содержат текст;
4. комментарии – иногда в них можно включить информацию, которая не будет показана, но доступна в DOM для чтения JS.

Навигация по DOM

DOM предоставляет широкий спектр возможностей при работе с HTML-элементом и его содержимым, но для этого сначала нужно получить ссылку на элемент. Доступ к DOM начинается с объекта `document`, от него можно добраться до любых узлов.

Узлы HTML-дерева имеют иерархическое отношение друг к другу. Термины `ancestor` (предок), `descendant` (потомок), `parent` (родитель), `child` (ребенок) и `sibling` (сосед) используются для описания отношений.

.



Навигация по DOM

Основные элементы можно получить следующим образом:

```
const html = document.documentElement;
const head = document.head;
const body = document.body;
const title = document.title;
```

Навигация по DOM

Свойства для навигации между узлами:

- `elem.parentNode` - выберет родителя elem
- `elem.childNodes` – коллекция (псевдомассив NodeList), хранит все дочерние элементы, включая текстовые.
- `elem.children` – коллекция (псевдомассив HTMLCollection), хранит только дочерние узлы-элементы, то есть соответствующие тегам.
- `elem.firstChild` - выберет первый дочерний элемент внутри elem, включая текстовые узлы.
- `elem.firstElementChild` - выберет первый дочерний узел-элемент внутри elem.
- `elem.lastChild` - выберет последний дочерний элемент внутри elem, включая текстовые узлы.
- `elem.lastElementChild` - выберет последний дочерний узел-элемент внутри elem.
- `elem.previousSibling` - выберет элемент "слева" от elem (его предыдущего соседа)
- `elem.previousElementSibling` - выберет узел-элемент "слева" от elem (его предыдущего соседа)
- `elem.nextSibling` - выберет элемент "справа" от elem (его следующего соседа)
- `elem.nextElementSibling` - выберет узел-элемент "справа" от elem (его предыдущего соседа)

Поиск узлов-элементов

`elem.querySelector(selector)`

Используется когда мы заведомо знаем, что подходящий элемент только один.

Возвращает первый найденный элемент внутри `elem`, соответствующий CSS-селектору `selector`. Если ничего не найдено, вернет `null`.

`elem.querySelectorAll(selector)`

Используется когда мы заведомо знаем, что подходящих элементов более одного.

Возвращает коллекцию (псевдомассив) всех элементов внутри `elem`, удовлетворяющих CSS-селектору `selector`. Если ничего не найдено, вернет пустой псевдомассив.

```
const elem1= document.querySelector("ul");  // первый элемент, найденный  по названию элемента
console.log(elem1);
const elem2= document.querySelector("#head");  // первый (единственный) элемент, найденный по id
console.log(elem2);
const elem3= document.querySelector(".menu-item"); // первый элемент, найденный по классу
console.log(elem3);
const elems1= document.querySelectorAll(".menu-item");    // Коллекция NodeList элементов с указанным
классом
console.log(elems1);
const elems2= document.querySelectorAll(".menu-item .active"); Коллекция NodeList
console.log(elems2);
```

Поиск узлов-элементов

Дополнительные методы для поиска узлов:

```
const elems1 = document.getElementsByTagName("li"); // HTMLCollection элементов li
const elem2 = document.getElementById("head"); // элемент id="head"
console.log(elem2);
const elems3 = document.getElementsByClassName("menu-item"); // HTMLCollection элементов   class="menu-
item"

const elems4 = document.getElementsByClassName("active menu-item"); // HTMLCollection элементов
class="menu-item active"
console.log(elems4);

//Нужно быть уверенным, что есть нужный идентификатор
const elem5 = document.getElementById("menu").getElementsByClassName("menu-item");
console.log(elem5);
```

DOM-коллекции

DOM-коллекции (псевдомассивы) HTMLCollection, NodeList, которые получаются при использовании childNodes, children и других методов, не являются JavaScript-массивами. В них нет методов массивов, таких как map, push, pop и других. При этом структура и некоторые свойства DOM-коллекции имеют много общего с массивом. Например, у неё тоже есть свойство length, и элементы коллекции можно перебирать в цикле for...of, потому что это перечисляемая сущность. Для коллекций NodeList также работает forEach.

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: `childNodes`, `children`, `getElementsBy...` возвращают живые коллекции, а `querySelectorAll` - неживые.

Если мы хотим использовать методы массивов для коллекции, то ее нужно преобразовать в массив, например, с помощью метода Array.from. Преобразованная в массив коллекция – статична.

```
const childNodes = body.childNodes
const arrayChilds = Array.from(childNodes);
```

Обход элементов

```
const elems1 = document.querySelectorAll("div"); //NodeList
console.log(elems1);
// вариант 1, в прототипе NodeList есть forEach
elems1.forEach((el)=> {
    console.log(el);
})
// вариант 2
for (let el of elems1){
    console.log(el);
}

const elems2 = document.getElementsByTagName("div"); //HTMLCollection
console.log(elems2);

for (let el of elems2){
    console.log(el);
}
```

Свойство innerHTML

elem.innerHTML — свойство, позволяет получить содержимое элемента, включая теги, в виде строки. Значение, возвращаемое innerHTML — всегда валидный HTML-код и оно доступно как для чтения, так и для записи. Если записать в innerHTML элемента строку с HTML-тегами, то браузер ее распарсит и превратит их в валидные DOM-узлы.

```
const elem = document.querySelector("#id123");
elem.innerHTML = '<p>Абзац.</p>';
elem.innerHTML += '<a href="">И ссылочка</a>';
```

Такой код говорит браузеру распарсить строку, проверить на наличие тегов, если нашел таковые, то создать DOM-элементы и вставить их в правильном порядке.

Свойство Node.textContent

`elem.textContent` — свойство, содержит текстовый контент внутри элемента. Доступно для записи, при чем вне зависимости, что будет передано в textContent, данные всегда будут записаны как текст.

```
const elem = document.querySelector("ul");
console.log(elem.textContent); // текстовый контент внутри ul
elem.textContent = "Ой, куда все делось?!"  //заменили все теги li на текст
```

Создание и добавление узлов

Мы можем не только выбирать уже существующие, но и удалять, а так же создавать новые элементы, после чего добавлять их в документ.

`document.createElement(tagName)`

Создает HTML-элемент по указанному имени тега и возвращает ссылку на него как результат своего выполнения. `tagName` - это строка, указывающая тип создаваемого элемента. Элемент создается в памяти, в DOM его еще нет.

```
const heading = document.createElement('h1');
console.log(heading); // <h1></h1>

heading.textContent = 'Новый заголовок';
console.log(heading); // <h1>Новый заголовок </h1>

const image = document.createElement('img');
image.setAttribute('src', '01.jpg');
image.setAttribute('alt', 'Африка');

console.log(image); // <img src="01.jpg" alt="Африка">
```

`document.createTextNode('text') // создает текстовый узел`

Создание и добавление узлов

Чтобы созданный элемент был отображен на странице, его необходимо добавить к уже существующему элементу в DOM. Допустим, что добавляем в некий элемент parentElem, для этого есть методы.

- `parentElem.appendChild(elem)` – добавляет elem в конец дочерних элементов parentElem.
- `parentElem.insertBefore(elem, nextSibling)` – добавляет elem в коллекцию детей `parentElem`, перед элементом `nextSibling`. Если вторым аргументом указать `null`, тогда `insertBefore` сработает как `appendChild`.

Если элемент для вставки – это существующий узел, то он изымается из своего старого места и ставится на новое. Отсюда вытекает правило — один и тот же узел не может быть одновременно в двух местах.

```
const body = document.body;
body.appendChild(heading);
body.appendChild(image);

const elem = document.querySelector("ul");
body.insertBefore(image, elem);
```

Создание и добавление узлов

Есть методы, которые позволяют вставить что угодно и куда угодно. Во всех этих методах, nodes – DOM-узлы или строки, в любом сочетании и количестве. Причём строки вставляются как текстовые узлы.

```
<!-- elem.before(nodes)  -->
<div class="example">
    <!-- elem.prepend(nodes)  -->
    <ul>
        <li>1</li>
        <li>2</li>
        <li>3</li>
    </ul>
    <!-- elem.append(nodes)  -->
</div>
<!-- elem.after(nodes)  -->
```

```
const elem = document.querySelector(".example")
const newH1 = document.createElement('h1');
newH1.textContent = "Новый заголовок";
elem.prepend(newH1);
```

Метод insertAdjacentHTML()

Метод парсит указанную строку как HTML и добавляет результирующие узлы в указанное место DOM-дерева. Не делает повторный рендеринг для существующих элементов внутри элемента-родителя на котором используется. Это позволяет избежать дополнительного этапа сериализации, делая его быстрее, чем непосредственная манипуляция innerHTML.

```
element.insertAdjacentHTML(position, string)
```

`position` — позиция относительно элемента:

```
<!-- 'beforebegin' - перед element  -->
<div class="example">
    <!-- 'afterbegin' - внутрь element, в самое начало контента  -->
    <ul>
        <li>1</li>
        <li>2</li>
        <li>3</li>
    </ul>
    <!-- 'beforeend' - внутрь element, в самый конец контента -->
</div>
<!-- 'afterend' - после element   -->
```

У этого метода есть братья-близнецы. Их синтаксис, за исключением последнего параметра, полностью совпадает с `insertAdjacentHTML`. Вместе они образуют универсальный швейцарский нож для вставки чего угодно куда угодно.

`elem.insertAdjacentElement(position, elem)` — вставляет в произвольное место не HTML-строку, а элемент elem.
`elem.insertAdjacentText(position, text)` — создаёт текстовый узел из строки text и вставляет его в указанное место относительно elem.

Метод insertAdjacentHTML()

```
const list = document.querySelector('#menu');
list.insertAdjacentHTML('beforebegin', '<h2>beforebegin заголовок</h2>');
list.insertAdjacentHTML('afterbegin', '<li>afterbegin item</li>');
list.insertAdjacentHTML('beforeend', '<li>beforeend item</li>');
list.insertAdjacentHTML('afterend', '<p>afterend текст</p>');
```

Клонирование узла

Представим, что у нас есть элемент с текстом, и мы хотим вставить такой же элемент в другую часть документа. Мы уже знаем, что каждый элемент может существовать в `document` в одном экземпляре. Но элемент можно клонировать и работать с этим клоном (точной копией).
Также в ряде случаев гораздо эффективнее клонировать существующий элемент, а потом внести изменения при необходимости, а не создавать новый элемент. В частности, если элемент большой, то клонировать его будет быстрее, чем пересоздавать.

`elem.cloneNode(true)` – создаст глубокую копию элемента – вместе с атрибутами, включая все поддерево. Если же вызвать с аргументом `false`, то копия будет сделана без дочерних элементов.

```
const elem = document.querySelector("#lorem p");
for(let i = 0; i<5; i++){
    const elemNew = elem.cloneNode(true);
    elem.parentNode.append(elemNew);
}
console.log(elem)
```

Удаление узлов

Для того, чтобы удалить узел существуют два метода. Первый, более старый метод, работающий во всех браузерах, позволяет удалить ребенка `elem` из родителя `parent`. В таком случае необходимо иметь ссылку как на родителя, так и на ребенка.

```
parent.removeChild(elem)
```

Более современный метод, но с гарантированной поддержкой только в новых браузерах, он вызывается на самом элементе elem, который необходимо удалить.

```
elem.remove()
```

```
body.removeChild(heading2);
// или
heading2.remove();
```

Свойство Element.classList

Объект содержит методы для работы с классами элемента.

- elem.classList.add(cls) - добавляет класс cls в список классов элемента
- elem.classList.remove(cls) - удаляет класс cls из списка классов элемента
- elem.classList.toggle(cls) - если класса cls нет - добавляет его, если есть- удаляет
- elem.classList.contains(cls) - возвращает true или false в зависимости от того, есть ли у элемента класс cls
- elem.classList.replace(oldClass, newClass) - заменяет существующий класс на указанный

```
const elem = document.querySelector("#lorem");
console.log(elem.classList); // DOMTokenList(2) ['ex-1', 'bg-1', value: 'ex-1 bg-1']
console.log(elem.classList.contains("ex-1")); // true
elem.classList.remove("ex-1");
console.log(elem.classList); // DOMTokenList(1) ['bg-1', value: 'bg-1’]
elem.classList.add("new-class");
console.log(elem.classList); // DOMTokenList(2) ['new-class', 'bg-1', value: 'new-class bg-1’]
// можно добавлять множественные классы
elem.classList.add("a", "b", "c");
console.log(elem.classList);

elem.classList.toggle("hidden"); // добавит, если класса нет
elem.classList.toggle("hidden"); // удалит, если класс есть

// classList – это псевдомассив, в прототипе которого есть метод forEach,
elem.classList.forEach(cls => {
  console.log(cls);
});
```

Свойство HTMLElement.style

Свойство `HTMLElement.style` используется для получения и установки встроенных стилей. Возвращает объект `CSSStyleDeclaration`, который содержит список всех свойств, определенных только во **встроенном** стиле элемента (то есть определенных через аттрибут `style` элемента), а не весь CSS. Свойство можно как читать, так и записывать. При записи свойства записываются в camelCase (верблюжьей нотации), то есть `background-color` превращается в `element.style.backgroundColor` и т. д.

Для получения всех стилей элемента (наследуемых, заданных в css-файле и вычисленных в соответствии со специфичностью необходимо использовать метод `window.getComputedStyle(element, pseudoElt)`

```
const elem = document.querySelector(".active a");
console.log(elem.style.color); // получили информацию об встроенном свойстве color
elem.style.color  = "green";  // изменили цвет

const computedStyle = window.getComputedStyle(elem, null);
console.log(computedStyle.paddingLeft); // получили информацию об свойстве из css
console.log(parseInt(computedStyle.paddingLeft)); // Преобразовали 20px в 20
```