Вводная информация

В современном мире JavaScript имеет много сфер применения. На нём пишутся серверная и клиентская части веб-приложений, он используется для создания мобильных приложений, майнинга криптовалют и даже программирования микроконтроллеров.

Основная область применения — использование JavaScript в браузерной среде.

С момента начала эры интернета и создания первого браузера прошло более 30 лет. Это было сложное время: частые взлеты и падения, жесткая конкуренция, которая часто называется «войны браузеров». Софтверные компании создавали новые технологии и использовали их как конкурентное преимущество. Так появился JavaScript (разработки Netscape) и CSS (Microsoft).

Netscape и Microsoft старались добавить в свой браузер уникальные функции, выделиться и завоевать как можно больше пользователей. Особенно сильно из-за этой неразберихи и хаоса страдали веб-разработчики, вынуждены создавать сайты таким образом, чтобы они одинаково отображались в Internet Explorer и Netscape Navigator.

Со временем ситуация стала меняться к лучшему. Количество компаний, занимающихся разработкой браузеров росло, и пришло понимание, что используемые браузерами технологии требуют стандартизации. Появились такие организации, как World Wide Web Consortium (W3C), и группы опытных веб-разработчиков (TAG, WHATWG, TC39), которые совместными усилиями стали документировать и стандартизировать существующие технологии и предлагать новые.

Сейчас понятие «кросс-браузерная совместимость» практически ушло из лексикона современных веб-разработчиков, а устаревшие браузеры, которые доставляли им много проблем, исчезли из статистики, или их доля стремится к нулю.

Современные браузеры — это комплексные программные продукты, по сложности и функциональности сопоставимые с операционными системами. И действительно, если посмотрим на список существующих API браузера, найдем практически всё, что «умеют» приложения на наших с вами компьютерах и даже больше.

Десктопные и мобильные приложения часто имеют веб-версию. Это делает их более универсальными и снимает ограничения для пользователей, которым не надо иметь определённую систему или что-то скачивать.

API браузера встроены в веб-браузер и используют данные браузера и компьютерной среды для осуществления более сложных действий с этими данными. Они не часть языка, API браузера строятся на основе встроенных функций JavaScript для увеличения возможностей разработчиков при написании кода.

Начнём с самого базового API, отвечающее за программное (объектное) представление веб-страницы (HTML-документа), а именно: DOM, или Document Object Model (объектная модель документа).

Введение в DOM

Документ, загруженный в каждую вкладку браузера, представлен объектной моделью документа (DOM). Это представление «древовидной структуры», созданное браузером. Он позволяет легко получить доступ к структуре HTML путём использования языков программирования. Например, сам браузер использует его для применения стиля и другой информации к соответствующим элементам, поскольку он отображает страницу (рассмотрим это позже). Разработчики манипулируют DOM, используя JavaScript после отображения страницы.

Создадим простейшую HTML-страницу и посмотрим, как будет выглядеть её DOM-представление:

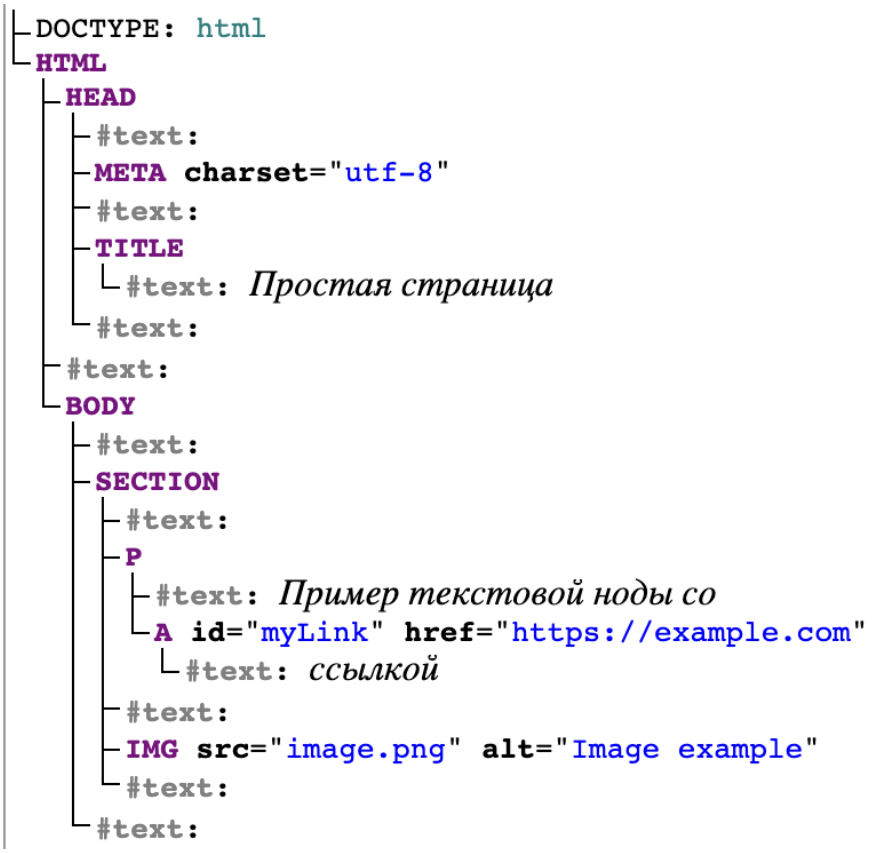
```
<!DOCTYPE html>
<html>

<head>
    <meta charset="utf-8">
    <title>Простая страница</title>
</head>

<body>
    <section>
        <p>Пример текстовой ноды со <a id="myLink"
href="https://example.com">ссылкой</a></p>
        <img src="image.png" alt="Image example">
    </section>
</body>

</html>
```

Используя онлайн-инструмент Live DOM Viewer, можно посмотреть, как выглядит эта страница в виде дерева элементов DOM:



Мы видим, что каждый HTML-элемент и текст в документе имеют собственную запись в дереве — каждый из них называется узлом (node). Для описания узла и его положения в дереве используются термины:

1. **Element node** — элемент, как он существует в DOM.
2. **Root node** — верхний узел в дереве, который в случае HTML всегда представляет собой HTML-узел. Другие типы разметки, такие как SVG и пользовательский XML, имеют разные корневые элементы.
3. **Child node** (узел-ребёнок) — узел, находящийся прямо внутри другого узла. Так, IMG в приведённом выше примере считается дочерним элементом SECTION.
4. **Descendant node** (узел-потомок) — узел внутри дочернего элемента. Так, IMG в приведённом выше примере считается дочерним элементом SECTION и потомком для родителя SECTION. IMG не ребёнок BODY, так как находится на двух уровнях ниже дерева в дереве, но он считается потомком BODY.
5. **Ancestor node** (узел-прародитель) — один из родительских узлов родителя текущего узла. Любой узел, для которого текущий узел представляется потомком, будет его прародителем.

6. **Parent node** (узел-родитель) — узел, в который входит текущий узел. Например, BODY — родительский узел SECTION в приведённом выше примере.
7. **Sibling nodes** (родственный узел) — узлы, лежащие на одном уровне в дереве DOM. Например, IMG и P — братья и сёстры в приведённом выше примере.
8. **Text node** — узел, содержащий текстовую строку.

Основы управления структурой DOM

Добавим элемент <script></script> прямо перед закрывающим тегом </body>. Чтобы управлять элементом внутри DOM, сначала надо выбрать его, а затем сохранить ссылку на него в переменной. Добавим в наш элемент script следующую строку:

```
<script>
   const link = document.querySelector('a')
</script>
```

В JavaScript есть множество способов выбора элемента и хранения указателя на него в переменной. Document.querySelector() — рекомендуемый современный подход, который считается удобным, потому что позволяет выбирать элементы, применяя селекторы CSS. Вышеупомянутый запрос querySelector() соответствует первому элементу <a>, который появляется в документе.

Чтобы сделать что-то с несколькими элементами, используется Document.querySelectorAll(). Он выбирает все элементы, которые соответствуют селектору и сохраняет ссылки на них в коллекцию элементов NodeList. Рассмотрим коллекции элементов NodeList и HTMLCollection (псевдомассивы) чуть позже.

Есть более старые методы для захвата ссылок на элементы. Например:

1. **Document.getElementById()** выбирает элемент с заданным значением атрибута id — передаётся функции как параметр.
2. **Document.getElementsByTagName()** возвращает коллекцию HTMLCollection, содержащую все элементы на странице этого типа — передаётся функции как параметр. Например, <p>, <a> и т. д.
3. **Document.getElementsByClassName()** возвращает коллекцию HTMLCollection дочерних элементов, соответствующих всем указанным именам классов — передаётся функции как параметр. Например, 'class-one class-two' и т. д.

Эти три метода работают в более старых браузерах, чем современные методы, такие как **querySelector()**. Но они не такие удобные, так как синтаксис CSS-селекторов более гибкий и позволяет задавать сложные критерии поиска.

Все описанные методы вызываются также, применительно к любому элементу: возвращаются лишь те элементы, которые считаются потомками указанного корневого элемента и удовлетворяют условиям поиска, то есть имеют указанные теги, классы и **id**.

В этом примере все варианты возвращают тот же элемент <a>:

```
<script>
  let link
  link = document.querySelectorAll('a')[0]
  link = document.querySelector('#myLink')
  link = document.getElementById('myLink')
  link = document.getElementsByTagName('a')[0]
</script>
```

Итак, теперь у нас есть ссылка на элемент, хранящаяся в переменной. Мы можем начать ей манипулировать, используя доступные элементу свойства и методы. Эти свойства и методы определены на таких интерфейсах, как **HTMLAnchorElement**, в случае <a>, его родительский интерфейс **HTMLElement** и **Node**, который представляет все узлы в DOM.

Прежде всего изменим текст внутри ссылки, обновив значение свойства Node.textContent. Добавим следующую строку ниже предыдущей:

```
link.textContent = 'Новый текст ссылки'
```

Мы также можем изменить URL-адрес, на который указывает ссылка, чтобы он не попадал в неправильное место при нажатии.

Добавим следующую строку:

```
link.href = 'https://google.com'
```

Создание и добавление новых узлов

Посмотрим, как создаются новые элементы.

Возвращаясь к текущему примеру, начнем с получения ссылки на наш элемент <section>. Добавим следующий код внизу существующего скрипта:

```
const sectionElement = document.querySelector('section')
```

Теперь создадим новый абзац, используя Document.createElement(), и передадим ему текстовое содержимое, как и раньше:

```
const paragraphElement = document.createElement('p')
paragraphElement.textContent = 'Новый текст параграфа'
```

Добавим новый абзац в конец раздела, воспользовавшись **Node.appendChild()**:

```
sectionElement.appendChild(paragraphElement)
```

Добавим дополнительный текстовый узел в наш новый абзац.

Сначала создадим текстовый узел, используя Document.createTextNode(). Теперь возьмём ссылку на абзац и добавим к нему дополнительный текстовый узел:

```
const paragraphElementText = document.createTextNode('Содержимое текстовой
ноды')
paragraphElement.appendChild(paragraphElementText)
```

Это большая часть того, что требуется для добавления узлов в DOM. Воспользуемся этими методами при построении динамических интерфейсов.

Клонирование узлов

Когда узел уже добавлен в дерево, повторное добавление его другому родителю приводит к удалению из текущего. Для копирования узла применяется метод **Node.cloneNode(deep)**. Он возвращает дубликат узла, из которого этот метод вызван. Метод принимает единственный логический параметр **deep**, определяющий, надо ли клонировать дочерние элементы узла.

Клонирование узлов копирует все атрибуты и их значения, включая слушателей событий, добавленных с использованием HTML-атрибутов, то есть <button onclick="..."">. Слушатели событий, добавленные методом **addEventListener()** или назначенные через свойства элемента, то есть node.onclick = fn, копируются.

Дубликат узла, возвращенного .**cloneNode(),** не считается частью документа и не имеет родителя, пока не добавится в другой узел, то есть часть документа, используя **Node.appendChild()** или другой метод.

Если **deep** установлен как **false** (значение по умолчанию), дочерние узлы, включая текстовые, не клонируются.

Если deep установлен как **true**, все поддеревья, включая текст, тоже копируется. Для пустых узлов, то есть элементов <img> и <input>, не имеет значения, установлен ли **deep** как **true** или **false**.

Важно! ***cloneNode()*** *иногда приводит к дублированию идентификаторов элементов в документе.*

Если исходный узел имеет идентификатор (атрибут **id**), и клон размещён в том же документе, идентификатор изменяется, чтобы быть уникальным. Атрибут name также нуждается в изменении.

```
const sectionElement = document.querySelector('section')
const sectionElementClone = sectionElement.cloneNode(true)
const link = sectionElementClone.querySelector('#myLink')
link.id = 'myLinkClone'
document.body.appendChild(sectionElementClone)
```

Удаление узлов

Удалить узел из DOM можно двумя способами:

- через ссылку на родительский элемент, используя **parentNode.removeChild(child);**
- применив метод **Element.remove().**

Удалим ранее созданный клон секции:

```
const sectionElementClone = document.querySelectorAll('section')[1]
sectionElementClone.parentNode.removeChild(sectionElementClone)
// sectionElementClone.remove()
```

Замена узла

DOM API содержит методы для замены одного узла другим или несколькими узлами. Как и в случае с удалением доступны два разных способа:

- через ссылку на родительский элемент, используя **parentNode.replaceChild(newChild, oldChild);**
- применив метод **Element.replaceWith(...nodes).**

```
const divElement = document.createElement('div')
const paragraphElement = document.createElement('p')
divElement.appendChild(paragraphElement)

const spanElement = document.createElement('span')
const strongElement = document.createElement('strong')
paragraphElement.replaceWith(spanElement, strongElement)

console.log(divElement.outerHTML)
```

Работа с свойствами и методами

```
<div class="product">
   <img class="product__img" width="250" height="350" src="img/photo1.jpg"
alt="photo product">
   <h3>Название футболки</h3>
   <p>Lorem ipsum dolor sit, amet consectetur adipisicing elit. Sequi,
reiciendis!</p>
   <button class="button">Купить</button>
</div>
<script>
   const buttonEl = document.querySelector('.button');
   const productImg = document.querySelector('.product__img');

   productImg.onclick = function() {
       productImg.src = 'img/photo2.jpg';
   }

   buttonEl.onclick = function() {
       buttonEl.textContent = 'Товар добавлен в корзину'
   }
</script>
```

В данном примере у нас есть блок продукта, внутри которого содержится изображение товараа, заголовок, описание и кнопка купить.

Наша задача сделать так, чтобы при клике на изображение появлялось новое изображение, а при клике на кнопку пить, текст кнопки менялся на “Товар добавлен в корзину”.

Первым делом нам необходимо найти элементы, с которыми будем работать, для этого используем конструкцию

```
const buttonEl = document.querySelector('.button');

const productImg = document.querySelector('.product__img');
```

Далее нам необходимо добавить обработчик **onclick** на каждый из этих элементов

Используя знания из нашего урока, мы можем использовать метод .**textContent** который позволяет менять значения текста внутри тега, теперь при клике на кнопку текст поменяется. Аналогично добавим метод .**src** который поможет нам изменить изображение товара, при клике на него.

Дополнительные материалы

1. Статья о псевдомассивах.

Используемые источники

1. Интерфейсы веб API.
2. Document.querySelector().