

# Тема 13. DOM: основы .

хекслет колледж



## Цель занятия:

Сформировать базовое понимание DOM и научить студентов получать элементы страницы и изменять их содержимое с помощью JavaScript.

# Учебные вопросы:

1. Понятие DOM. Структура DOM-дерева.
2. Основные способы доступа к элементам DOM
3. Работа с содержимым элементов
4. Работа с атрибутами элементов
5. Работа с CSS-классами через JavaScript
6. Создание и удаление элементов DOM

# 1. Понятие DOM. Структура DOM-дерева.

DOM (Document Object Model) — это объектная модель HTML-документа, которая создаётся браузером после загрузки страницы.

DOM представляет веб-страницу в виде иерархического дерева объектов, с которым JavaScript может взаимодействовать.

Иными словами, DOM — это форма представления HTML-документа, удобная для программной обработки.

# Что такое DOM на пальцах?

DOM — это ваш «пульт управления» страницей.

- HTML — это просто текст в файле. Его нельзя «потрогать» через JS.
- Браузер читает этот текст и превращает его в объекты (DOM).
- JS работает с этими объектами.

Главная мысль: Мы не правим файл `.html` напрямую. Мы меняем то, что пользователь видит в браузере прямо сейчас.

# HTML и DOM

HTML — это текстовый документ.

DOM — это результат интерпретации HTML браузером.

Последовательность работы:

- браузер загружает HTML;
- анализирует разметку;
- строит DOM-дерево;
- JavaScript получает к нему доступ через объект document.

# Структура DOM-дерева

Страница — это не плоский список, а иерархия (семья).

- **Родители:** Контейнеры, внутри которых кто-то есть (`<div>`, `<ul>`).
- **Дети:** Элементы внутри контейнеров (`<li>` внутри `<ul>`).
- **Текст:** Самое глубокое вложение — это само содержимое тегов.

# Пример HTML:

```
<!DOCTYPE html>
<html>
  <head>
    <title>Страница</title>
  </head>
  <body>
    <h1>Заголовок</h1>
    <p>Текст</p>
  </body>
</html>
```

# Структура DOM в инструментах разработчика:

The screenshot shows the 'Elements' tab in the Chrome DevTools interface. The DOM tree displays the following structure:

```
<!DOCTYPE html>
<html> == $0
  > <head> ...
  > <body>
    <h1>Заголовок</h1>
    <p>Текст</p>
  </body>
</html>
```

# Точка входа: объект `document`

Чтобы JS мог что-то найти на странице, ему нужен «корень».

- `document` — это главный встроенный объект, который содержит всё дерево вашего сайта.
- С него начинается любая работа с элементами.

Попробуем прямо сейчас (F12 -> Console):

```
› document.body.style.background = 'red';
```

## Вывод:

DOM — это объектное, иерархическое представление HTML-документа.

DOM-дерево отражает структуру страницы и позволяет JavaScript управлять содержимым и поведением сайта.

Понимание структуры DOM-дерева является основой для дальнейшего изучения работы с элементами и событиями.

## 2. Основные способы доступа к элементам DOM

### 1. getElementById()

Используется для поиска одного элемента по уникальному id.

```
<h1 id="title">Заголовок</h1>
```

```
const title = document.getElementById('title');  
console.log(title);
```

## Особенности `getElementById()` :

- возвращает один элемент;
- id должен быть уникальным;
- если элемент не найден — возвращается null.

## 2. `getElementsByClassName()`

Находит все элементы с указанным классом.

```
<p class="text">Первый</p>
<p class="text">Второй</p>
```

```
const texts = document.getElementsByClassName('text');
console.log(texts);
```

Особенности:

- возвращает коллекцию элементов;
- доступ к элементам по индексу (`texts[0]`);
- если элементов нет — возвращается пустая коллекция.

### 3. `getElementsByName()`

Позволяет получить все элементы с заданным HTML-тегом.

```
const paragraphs = document.getElementsByTagName('p');
```

Особенности:

- возвращает коллекцию элементов;
- используется реже, чем современные методы.

👉 `getElementsByClassName` и `getElementsByName` на практике почти не применяются, почти всегда используются `querySelector/All`

## 4. querySelector()

Универсальный и современный способ поиска элементов.  
Использует CSS-селекторы.

```
// Найти первый параграф
const firstParagraph = document.querySelector('p');

// Найти элемент с id="header"
const header = document.querySelector('#header');

// Найти элемент с class="button"
const button = document.querySelector('.button');

// Найти первую кнопку
const firstButton = document.querySelector('button');
```

## Особенности `querySelector()`:

- возвращает первый найденный элемент;
- поддерживает любые CSS-селекторы (#id, .class, tag, вложенные селекторы).

## 5. querySelectorAll()

Находит все элементы, соответствующие CSS-селектору.

```
// Найти ВСЕ параграфы
const allParagraphs = document.querySelectorAll('p');

// Найти ВСЕ элементы с class="button"
const allButtons = document.querySelectorAll('.button');

// Найти ВСЕ input'ы
const allInputs = document.querySelectorAll('input');
```

## Особенности `querySelectorAll()` :

- возвращает коллекцию элементов;
- поддерживает любые CSS-селекторы;
- часто используется в современной разработке.

## Важные замечания:

- Если элемент не найден, JavaScript не выдаёт ошибку, а возвращает null;
- Перед работой с элементом полезно проверять, что он найден;
- querySelector и querySelectorAll — предпочтительный выбор в большинстве случаев.

## Вывод:

- Доступ к элементам DOM — это процесс поиска HTML-элементов в структуре страницы с помощью методов объекта `document`.
- Правильный выбор метода позволяет эффективно управлять содержимым и поведением веб-страницы с помощью JavaScript.

### 3. Работа с содержимым элементов

После того как элемент найден в DOM, JavaScript позволяет читать и изменять его содержимое.

Это один из ключевых механизмов динамического поведения веб-страниц:

- изменение текста, вставка HTML, обновление данных без перезагрузки страницы.

## 1. `textContent`

Позволяет получать и изменять текстовое содержимое элемента.

```
<p id="info">Старый текст</p>
```

```
const info = document.getElementById('info');
```

```
console.log(info.textContent); // Старый текст  
info.textContent = 'Новый текст';
```

## Особенности:

- работает только с текстом;
- HTML-теги воспринимаются как обычный текст;
- безопасен (не выполняет HTML-код).

## 2. innerHTML

Позволяет читать и изменять HTML-содержимое элемента.

```
<div id="box"></div>
<script>
    const box = document.getElementById("box");
    box.innerHTML = "<p>text2</p>"
</script>
```

## Особенности:

- можно вставлять HTML-разметку;
- удобно для быстрого обновления разметки;
- требует осторожности при использовании данных от пользователя (след. слайд)

## XSS (Cross-Site Scripting) уязвимость innerHTML

- innerHTML интерпретирует строку как HTML-код. Если в строке есть теги <script>, <img> с обработчиками событий или стили, браузер их выполнит или отрисует.
- В современных браузерах работает защитный механизм, вставленные через innerHTML, не должны исполняться. Но проблема полностью не решена.

```
const box = document.getElementById("container");
box.innerHTML = `<img src='invalid-path' onerror='alert("XSS !")'>`;
```

### 3. innerText

Возвращает и изменяет только тот текст, который пользователь реально видит на экране, с учётом CSS-стилей.

```
<p id="p1">Текст</p>
<script>
    const p1 = document.getElementById("p1");
    p1.innerText = "Новый текст"
</script>
```

Добавление текста к существующему содержимому:

```
const message =  
document.getElementById('message');  
message.textContent += ' Дополнительный  
текст';
```

Полная замена содержимого элемента:

```
element.innerHTML = "";
```

## Вывод:

- JavaScript предоставляет несколько способов работы с содержимым DOM-элементов.
- Выбор между textContent, innerText и innerHTML зависит от того, нужно ли работать с простым текстом или с HTML-разметкой.

## 4. Работа с атрибутами элементов

Атрибуты — это дополнительные свойства HTML-элементов, которые задаются в разметке и влияют на их поведение или внешний вид (например: `id`, `class`, `src`, `href`, `title`, `disabled` и т.д.).

JavaScript позволяет получать, изменять, добавлять и удалять атрибуты элементов во время работы страницы.

# Получение значения атрибута

Используется метод `getAttribute()`.

```
const link = document.querySelector('a');
const url = link.getAttribute('href');

console.log(url);
```

Метод возвращает:

- значение атрибута, если он существует;
- `null`, если атрибут не найден.

# Изменение и добавление атрибута

Используется метод `setAttribute()`.

```
const img = document.querySelector('img');
img.setAttribute('src', 'image.jpg');
img.setAttribute('alt', 'Описание изображения');
```

Если атрибут:

- уже существует — его значение будет изменено;
- не существует — он будет создан.

# Проверка наличия атрибута

Метод `hasAttribute()` возвращает `true` или `false`.

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

```
if (button.hasAttribute('disabled')) {  
    console.log('Кнопка отключена');  
}
```

## Специальный случай: `class` и `id`.

Хотя `class` и `id` являются атрибутами, для работы с ними часто используются более удобные свойства:

```
element.id = 'main';
```

```
element.className = 'container';
```

# Когда используется работа с атрибутами

Работа с атрибутами применяется для:

- изменения ссылок и изображений;
- включения и отключения элементов формы;
- добавления вспомогательной информации (title, aria-атрибуты);
- динамического изменения поведения элементов интерфейса.

## Итог:

- Методы `getAttribute`, `setAttribute`, `removeAttribute` и `hasAttribute` позволяют гибко управлять HTML-элементами и адаптировать страницу без перезагрузки.

## 5. Работа с CSS-классами через JavaScript

Для управления внешним видом элементов страницы JavaScript часто изменяет CSS-классы.

Это считается правильной практикой, так как стили остаются в CSS, а логика — в JavaScript.

# Свойство classList

Для работы с CSS-классами используется объект `classList`, доступный у каждого DOM-элемента.

```
const box = document.querySelector('.box');
```

# Добавление класса — `classList.add()`

```
box.classList.add('active');
```

Добавляет указанный класс элементу.  
Если класс уже есть — повторно добавлен  
не будет.

# Удаление класса — classList.remove()

```
box.classList.remove('active');
```

Удаляет класс у элемента, если он существует.

# Переключение класса — `classList.toggle()`

```
box.classList.toggle('active');
```

- если класса нет → добавляет;
- если класс есть → удаляет.

Часто используется для:

- меню,
- модальных окон,
- вкладок.

## Проверка наличия класса — classList.contains()

```
if (box.classList.contains('active')) {  
    console.log('Элемент активен');  
}
```

Возвращает true или false.

# Пример практического использования:

HTML:

```
<div class="card"></div>
```

CSS:

```
.card {  
    background: #lightgray;  
}  
.card.active {  
    background: #lightgreen;  
}
```

# JavaScript:

```
const card = document.querySelector('.card');
card.classList.add('active');
```

В результате элемент изменит внешний вид  
за счёт CSS-класса.

# Почему лучше использовать классы, а не inline-стили?

Не рекомендуется:

```
element.style.backgroundColor = 'red';
```

Предпочитительно:

```
element.classList.add('error');
```

## Преимущества:

- код чище и понятнее;
- стили централизованы в CSS;
- упрощается поддержка и доработка интерфейса.

# Итог:

- `classList` — основной инструмент управления классами;
- JS управляет состоянием, CSS — отображением;
- работа с классами — основа интерактивных интерфейсов и адаптивных элементов.

# 6. Создание и удаление элементов DOM

JavaScript позволяет динамически создавать, добавлять и удалять HTML-элементы во время работы страницы. Это используется для формирования списков, карточек, сообщений, элементов интерфейса без перезагрузки страницы.

# Создание новых элементов

Для создания элемента используется метод  
**document.createElement()**.

```
const paragraph = document.createElement('p');
```

На этом этапе элемент:

- существует в памяти;
- ещё не добавлен на страницу.

# Изменение созданного элемента

После создания можно задать:

- текстовое содержимое;
- атрибуты;
- CSS-классы.

```
paragraph.innerText = 'Новый абзац';  
paragraph.classList.add('text');
```

# Добавление элемента в DOM

Чтобы элемент появился на странице, его нужно добавить в DOM-дерево.

Добавление в конец  
**parent.append(child);**

Пример:

```
const container = document.querySelector('.container');
container.append(paragraph);
```

Элемент будет добавлен в конец контейнера.

# Добавление в начало

Пример:

```
const container = document.querySelector('.container');
```

```
container.prepend(paragraph);
```

# Добавление рядом с элементом

Пример:

**element.before(newElement);**

**element.after(newElement);**

Пример:

```
const ul = document.querySelector('ul');
```

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

```
li.innerText = 'Новый пункт';
```

```
ul.append(li);
```

# Удаление элементов

Удаление самого элемента

**paragraph.remove();**

Элемент полностью удаляется из DOM.

Удаление дочернего элемента через родителя

**container.removeChild(paragraph);**

# Создание и удаление элементов применяется для:

- генерации списков;
- сообщений об ошибках или успехе;
- карточек товаров;
- динамических блоков интерфейса.

## Итог:

- createElement() — создаёт элемент;
- append(), prepend() — добавляют элемент в DOM;
- before(), after() — вставляют элемент рядом;
- remove() — удаляет элемент со страницы.

Эти операции являются основой динамических веб-интерфейсов и активно используются совместно с событиями, которые изучаются на следующей лекции.

# Пример: Переключение темной темы:

Чтобы сайт реагировал на пользователя,  
нужно соединить три шага:  
Найти — Послушать — Изменить.

HTML:

```
<body>
|  <button class="btn-toggle">click</button>
</body>
```

# CSS:

```
.btn-toggle {  
    padding: 10px 20px;  
    border-radius: 5px;  
    border: 1px solid #333;  
}  
  
.dark-theme {  
    background-color: #2c3e50; /* Темный фон */  
    color: #ecf0f1; /* Светлый текст */  
}
```

# JS:

```
// 1. Находим кнопку и корпус страницы
const btn = document.querySelector('.btn-toggle');
const body = document.querySelector('body');

// 2. Слушаем клик (Событие)
btn.addEventListener('click', () => {
    // 3. Меняем CSS-класс
    body.classList.toggle('dark-theme');
});
```

# Как это работает:

- `addEventListener` — «ухо», которое ждет клика.
- `toggle` — умный метод: если класса нет — добавит, если есть — удалит.

Результат: Одной строчкой кода мы создали интерактив, который видит пользователь.

# Итоги лекции:

- DOM — это живое дерево элементов. Мы меняем объекты в браузере, а не текст в файле.
- document — корень этого дерева и наш главный инструмент для поиска элементов.
- Меняем всё: JavaScript может на лету изменить текст, картинку или структуру страницы без её перезагрузки.
- Стили — через классы: Вместо того чтобы менять цвета в JS, мы просто переключаем CSS-классы через `classList`. Это чище и удобнее.
- Динамика: Мы можем создавать элементы из «пустоты» и добавлять их на страницу, когда это нужно (например, новое сообщение в чате).
- DOM — это база: Без понимания того, как найти и изменить элемент, невозможно создать ни одно современное приложение.

# Контрольные вопросы:

- Зачем нам нужен DOM, если у нас уже есть HTML-файл?
- В чем универсальность **querySelector** и что он вернет, если на странице 10 одинаковых кнопок?
- **textContent** vs **innerHTML**: какой метод вы выберете для вставки имени пользователя из базы данных и почему?
- Почему профи предпочитают **classList.add()** вместо прямого изменения **style.color**?
- Как заставить новый элемент, созданный через **createElement**, появиться на экране?

# Домашнее задание:

<https://ru.hexlet.io/courses/js-basics>

хекслет колледж

@HEXLY.KZ