

ПМ3 Разработка модулей ПО.

**РО 3.2 Разрабатывать модули с применением DOM API,
Regexp, HTTP**

Тема 1. JS: DOM API.

Лекция 28. Поиск элементов и работа с коллекциями.

Цель занятия:

Научиться находить элементы в DOM различными методами, разбираться в типах коллекций (HTMLCollection, NodeList) и уметь использовать CSS-селекторы и атрибуты для выборки элементов.

Учебные вопросы:

- 1. Методы поиска элементов в DOM.**
- 2. HTMLCollection и NodeList.**
- 3. Поиск по CSS-селекторам и атрибутам.**
- 4. Практика.**

1. Методы поиска элементов в DOM.

◆ 1. getElementById(id)

- Находит один элемент по его уникальному **id**.
- Если не найден → вернёт **null**.
- Работает очень быстро (браузер оптимизирован под id).

```
<p id="info"></p>
```

```
let el = document.getElementById("info");  
el.textContent = "Hello!";  
console.log(el.textContent); // "Hello!"
```

◆ 2. `getElementsByClassName(className)`

- Возвращает живую коллекцию (`HTMLCollection`) всех элементов с данным классом.
- Если класс у элемента изменится → коллекция обновится.

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

```
let notes = document.getElementsByClassName("note");  
console.log(notes.length); // 2  
console.log(notes[0]); // <p class="note">Первый</p>
```

◆ 3. `getElementsByTagName(tagName)`

Возвращает живую коллекцию элементов по тегу (``, `<p>`, `<div>` и т.д.).

Можно вызывать у любого элемента, чтобы искать внутри него.

```
<ul>  
  <li>Элемент 1</li>  
  <li>Элемент 2</li>  
</ul>
```

```
let items = document.getElementsByTagName("li");  
console.log(items[0].textContent); // "Элемент 1"
```

◆ 4. `querySelector(cssSelector)`

- Принимает любой CSS-селектор.
- Возвращает первый подходящий элемент или `null`.
- Удобен, если нужен один конкретный элемент.

```
<p class="note">Пример</p>
```

```
let el = document.querySelector(".note");  
console.log(el.textContent); // "Пример"
```


◆ 5. `querySelectorAll(cssSelector)`

Возвращает статическую коллекцию (NodeList) всех элементов по селектору.

Коллекция не обновляется при изменении DOM.

У NodeList есть метод `forEach()` → удобно обходить.

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

```
let boxes = document.querySelectorAll(".box");  
boxes.forEach(box => box.style.border = "1px solid red");
```

Сравнительная таблица

Метод	Что возвращает	Живой/Статический	Когда использовать
<code>getElementById</code>	Один элемент	—	Уникальный id
<code>getElementsByClassName</code>	HTMLCollection (по классу)	Живой	Массовый выбор по классу
<code>getElementsByTagName</code>	HTMLCollection (по тегу)	Живой	Все элементы определённого тега
<code>querySelector</code>	Первый элемент	—	Любой CSS-селектор (один элемент)
<code>querySelectorAll</code>	NodeList (по селектору)	Статический	Массовый выбор через CSS-селекторы

`getElement*` → для простых случаев (id, класс, тег).

`querySelector*` → для сложных селекторов и комбинаций.

2. HTMLCollection и NodeList.

◆ Зачем нужны коллекции?

Когда мы ищем элементы в DOM (`getElementsByClassName`, `querySelectorAll` и др.), результатом часто является список элементов.

Это не массив, а специальные коллекции.

◆ HTMLCollection

Возвращается методами:

- `document.getElementsByTagName("div")`
- `document.getElementsByClassName("item")`
- `element.children`

Особенности:

- Это живая коллекция — автоматически обновляется при изменении DOM.
- Содержит только элементы (узлы типа `element`).

👉 Пример:

```
<body>
  <ul>
    <li>Первый</li>
    <li>Второй</li>
  </ul>

  <script>
    let items = document.getElementsByTagName("li");
    console.log(items.length); // 2
    document.querySelector("ul").append(document.createElement("li"));
    console.log(items.length); // уже 3 (коллекция обновилась!)
  </script>
```

◆ NodeList.

Возвращается методами:

- `document.querySelectorAll("div")`
- `element.childNodes`

Особенности:

- Может содержать не только элементы, но и текстовые/комментарии.
- Чаще всего это статическая коллекция (например, `querySelectorAll`).
- Есть исключения: `Node.childNodes` — живая `NodeList`.

👉 Пример:

```
<ul>  
  <li>Первый</li>  
  <li>Второй</li>  
</ul>
```

```
<script>  
  let items = document.querySelectorAll("li");  
  console.log(items.length); // 2  
  document.querySelector("ul").append(document.createElement("li"));  
  console.log(items.length); // всё ещё 2 (список статический)  
</script>
```

◆ Обход коллекций.

Коллекции похожи на массивы, но это не массивы:
У них нет методов `map`, `filter`, `reduce`.

Можно обходить:

```
for (let el of items) { } // работает  
items.forEach(el => { }); // работает у NodeList
```

Преобразование в массив:

```
let arr1 = Array.from(items);  
// или  
let arr2 = [...items];
```


Выводы:

- HTMLCollection → живая, только элементы.
- NodeList → обычно статическая, может содержать любые узлы.
- Для удобной работы коллекции можно превращать в массивы (Array.from, spread-оператор [...]).

3. Поиск по CSS-селекторам и атрибутам.

Зачем использовать селекторы?

Методы `querySelector` и `querySelectorAll` позволяют находить элементы не только по `id` или классу, а любым CSS-селектором. Это удобно, если:

- элементы сложно отличить только по тегу или классу;
- нужен первый/последний/конкретный потомок;
- требуется фильтрация по атрибутам (`href`, `data-*` и др.).

◆ Простые селекторы.

По тегу:

```
document.querySelector("p"); // первый <p>  
document.querySelectorAll("p"); // все <p>
```

По классу:

```
document.querySelector(".note"); // элемент с классом note  
document.querySelectorAll(".note"); // все элементы с этим классом
```

По id:

```
document.querySelector("#header"); // элемент с id="header"
```

◆ Комбинированные селекторы.

Потомки через пробел:

```
document.querySelector("ul li"); // первый <li> внутри <ul>
```

Непосредственный потомок (>):

```
document.querySelector("div > p"); // <p>, у которого родитель – div
```

Класс + тег:

```
document.querySelector("p.active"); // <p class="active">
```

Псевдоклассы (работают в JS так же, как в CSS):

```
document.querySelector("ul li:first-child"); // первый <li> в списке  
document.querySelector("ul li:last-child");  // последний <li>
```

◆ Работа с data-* атрибутами.

HTML5 позволяет создавать пользовательские атрибуты data-*:

```
<div data-role="admin" data-level="5">Пользователь</div>
```

Поиск через селекторы:

```
document.querySelector('[data-role="admin"]'); // элемент с data-role="admin"
```

Доступ к значению в JS:

```
let el = document.querySelector('[data-role="admin"]');  
console.log(el.dataset.role); // "admin"  
console.log(el.dataset.level); // "5"
```

Подробнее про **data-*** атрибуты и **dataset**.

◆ Что такое **data-*** атрибуты?

- Это специальные пользовательские атрибуты, которые начинаются с префикса **data-**.
- Стандарт HTML5 разрешает их использовать для хранения «мини-данных» прямо в HTML.
- Они не влияют на отображение страницы, но доступны в JS и CSS.

👉 Используются, когда нужно хранить небольшую информацию, связанную с элементом (например, **id** товара, роль пользователя, состояние кнопки).

Пример:

```
<button data-id="42" data-role="delete">Удалить</button>
```

Здесь:

`data-id="42"` — идентификатор (например, товара).

`data-role="delete"` — назначение кнопки.

◆ Как работать с ними в JS?

1. Через `getAttribute`:

```
let btn = document.querySelector("button");  
console.log(btn.getAttribute("data-id"));    // "42"  
console.log(btn.getAttribute("data-role"));  // "delete"
```

2. Через свойство dataset.

У каждого элемента есть свойство dataset.

Это объект, где ключи соответствуют data-* атрибутам.

```
let btn = document.querySelector("button");  
console.log(btn.dataset.id);    // "42"  
console.log(btn.dataset.role);  // "delete"
```


Внимание:

В dataset имена пишутся в camelCase:

- data-user-name → dataset.userName
- data-max-value → dataset.maxValue

◆ Использование в CSS.

Атрибуты data-* можно использовать в селекторах:

```
<style>  
  button[data-role="delete"] {  
    color:  red;  
  }  
</style>
```

```
<button data-role="delete">delete</button>
```

◆ Когда использовать data-*?

- ✅ Удобны для «мини-данных» в HTML (id, роль, состояние).
- ✅ Не требуют дополнительных запросов на сервер.
- ❌ Но не стоит хранить в них большие объёмы данных или чувствительную информацию (пароли, токены).

👉 Вывод:

- data-* — способ хранить пользовательские данные в HTML.
- В JS доступны через `element.dataset` как свойства объекта.
- В CSS — через селекторы по атрибутам.

👉 Вывод:

- `querySelector*` работает со всеми CSS-селекторами.
- Можно комбинировать селекторы для точного поиска.
- `data-*` атрибуты — гибкий способ хранить «мини-данные» внутри разметки, легко читаются через `dataset`.

4. Практика работы с DOM: поиск и коллекции.

◆ 1. Изменить текст элемента по id

```
<h1 id="title">Старый заголовок</h1>
<script>
  let h1 = document.getElementById("title");
  h1.textContent = "Новый заголовок!";
</script>
```


◆ 2. Подсветить группу элементов по классу.

```
<ul>
  <li class="item">Элемент 1</li>
  <li class="item">Элемент 2</li>
  <li class="item">Элемент 3</li>
</ul>

<script>
  let items = document.getElementsByClassName("item");
  for (let el of items) {
    el.style.background = "yellow";
  }
</script>
```

◆ 3. Выделить элементы с data-role и изменить стиль.

```
<button data-role="admin">Admin</button>
<button data-role="user">User</button>

<script>
  let buttons = document.querySelectorAll("[data-role]");
  buttons.forEach(btn => {
    btn.style.border = "2px solid red";
  });
</script>
```

◆ 4. Пройтись циклом по коллекции и добавить изменения.

```
<ul>
  <li>Первый</li>
  <li>Второй</li>
  <li>Третий</li>
</ul>

<script>
  let lis = document.querySelectorAll("li");
  lis.forEach((li, index) => {
    li.textContent = `${index + 1}. ${li.textContent}`;
  });
</script>
```

◆ 5. Комбинированный пример: работа с разными селекторами.

```
<div id="box" class="container" data-type="info">  
  Привет, я контейнер!  
</div>  
  
<script>  
  let box = document.querySelector("#box.container[data-type='info']");  
  box.style.color = "blue";  
  box.style.fontWeight = "bold";  
</script>
```

Вывод:

- Методы поиска позволяют находить элементы по id, классу, тегу, селекторам и атрибутам.
- С коллекциями (HTMLCollection, NodeList) можно работать циклом или через `forEach`.
- Часто практическая задача: найти группу элементов и массово изменить их стиль или текст.

Контрольные вопросы:

- Чем отличается getElementById от querySelector?
- В чём разница между HTMLCollection и NodeList?
- Что значит, что коллекция «живая»?
- Как выбрать все <p> с классом .active?
- Как обратиться к data-* атрибуту через JS?

Домашнее задание:

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

9 Манипулирование DOM-деревом

Учимся менять DOM-дерево, добавлять и удалять элементы

10 Управление узлами DOM

Учимся модифицировать элементы, разбираем разницу между атрибутами и свойствами

11 Полифиллы

Выясняем, как нивелировать различия между браузерами при работе с DOM

12 Введение в события

Материалы лекций:

<https://github.com/ShViktor72/Education2025>

Обратная связь:

colledge20education23@gmail.com