# Материалы занятия

**Курс: Разработка интерфейса на JavaScript**

**Дисциплина: Основы JavaScript**

**Тема занятия №16: Взаимодействие с HTML. BOM, DOM**

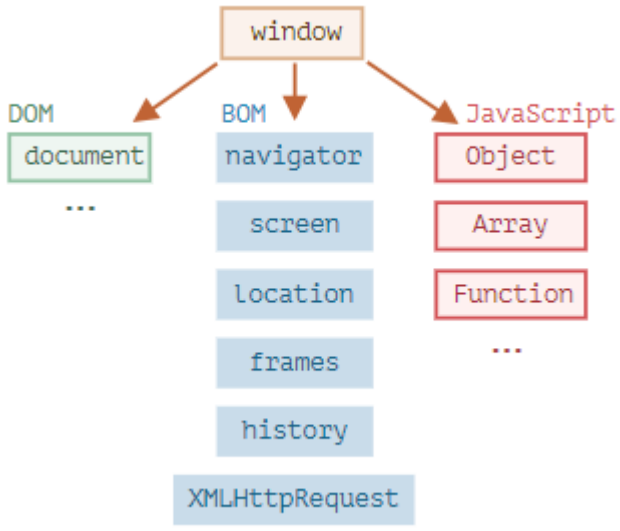
## Браузерное окружение, спецификации

Язык JavaScript изначально был создан для веб-браузеров. Но с тех пор он значительно эволюционировал и превратился в кроссплатформенный язык программирования для решения широкого круга задач.

Сегодня JavaScript может использоваться в браузере, на веб-сервере или в какой-то другой среде, даже в кофеварке. Каждая среда предоставляет свою функциональность, которую спецификация JavaScript называет *окружением*.

Окружение предоставляет свои объекты и дополнительные функции, в дополнение базовым языковым. Браузеры, например, дают средства для управления веб-страницами. Node.js делает доступными какие-то серверные возможности и так далее.

На картинке ниже в общих чертах показано, что доступно для JavaScript в браузерном окружении:




Как мы видим, имеется корневой объект window, который выступает в 2 ролях:

1. Во-первых, это глобальный объект для JavaScript-кода, об этом более подробно говорится в главе Глобальный объект.
2. Во-вторых, он также представляет собой окно браузера и располагает методами для управления им.

Например, здесь мы используем window как глобальный объект:

```
function sayHi() {
  alert("Hello");
}

// глобальные функции доступны как методы глобального объекта:
window.sayHi();
```

А здесь мы используем window как объект окна браузера, чтобы узнать его высоту:

```
alert(window.innerHeight); // внутренняя высота окна браузера
```

Существует гораздо больше свойств и методов для управления окном браузера. Мы рассмотрим их позднее.

## DOM (Document Object Model)

Document Object Model, сокращённо DOM – объектная модель документа, которая представляет все содержимое страницы в виде объектов, которые можно менять.

Объект document – основная «входная точка». С его помощью мы можем что-то создавать или менять на странице.

Например:

```
// заменим цвет фона на красный,
document.body.style.background = "red";

// а через секунду вернём как было
setTimeout(() => document.body.style.background = "", 1000);
```

Мы использовали в примере только document.body.style, но на самом деле возможности по управлению страницей намного шире. Различные свойства и методы описаны в спецификации:

## DOM – не только для браузеров

Спецификация DOM описывает структуру документа и предоставляет объекты для манипуляций со страницей. Существуют и другие, отличные от браузеров, инструменты, использующие DOM.

Например, серверные скрипты, которые загружают и обрабатывают HTML-страницы, также могут использовать DOM. При этом они могут поддерживать спецификацию не полностью.

## BOM (Browser Object Model)

Объектная модель браузера (Browser Object Model, BOM) – это дополнительные объекты, предоставляемые браузером (окружением), чтобы работать со всем, кроме документа.

Например:

- Объект navigator даёт информацию о самом браузере и операционной системе. Среди множества его свойств самыми известными являются: navigator.userAgent – информация о текущем браузере, и navigator.platform – информация о платформе (может помочь в понимании того, в какой ОС открыт браузер – Windows/Linux/Mac и так далее).
- Объект location позволяет получить текущий URL и перенаправить браузер по новому адресу.

Вот как мы можем использовать объект location:

```
alert(location.href); // показывает текущий URL
if (confirm("Перейти на Wikipedia?")) {
  location.href = "https://wikipedia.org"; // перенаправляет браузер
}
```

Функции alert/confirm/prompt тоже являются частью BOM: они не относятся непосредственно к странице, но представляют собой методы объекта окна браузера для коммуникации с пользователем.

## Итого

Говоря о стандартах, у нас есть:

### Спецификация DOM

описывает структуру документа, манипуляции с контентом и события, подробнее на https://dom.spec.whatwg.org.

### Спецификация CSSOM

Описывает файлы стилей, правила написания стилей и манипуляций с ними, а также то, как это всё связано со страницей, подробнее на https://www.w3.org/TR/cssom-1/.

### Спецификация HTML

Описывает язык HTML (например, теги) и BOM (объектную модель браузера) – разные функции браузера: setTimeout, alert, location и так далее, подробнее на https://html.spec.whatwg.org. Тут берётся за основу спецификация DOM и расширяется дополнительными свойствами и методами.

Кроме того, некоторые классы описаны отдельно на https://spec.whatwg.org/.

Пожалуйста, заметьте для себя эти ссылки, так как по ним содержится очень много информации, которую невозможно изучить полностью и держать в уме.

Когда вам нужно будет прочитать о каком-то свойстве или методе, справочник на сайте Mozilla https://developer.mozilla.org/ru/ тоже очень хороший ресурс, хотя ничто не сравнится с чтением спецификации: она сложная и объёмная, но сделает ваши знания максимально полными.

## DOM-дерево

Основой HTML-документа являются теги.

В соответствии с объектной моделью документа («Document Object Model», коротко DOM), каждый HTML-тег является объектом. Вложенные теги являются «детьми» родительского элемента. Текст, который находится внутри тега, также является объектом.

Все эти объекты доступны при помощи JavaScript, мы можем использовать их для изменения страницы.

Например, document.body – объект для тега <body>.

Если запустить этот код, то <body> станет красным на 3 секунды:

```
document.body.style.background = 'red'; // сделать фон красным

setTimeout(() => document.body.style.background = '', 3000); // вернуть назад
```

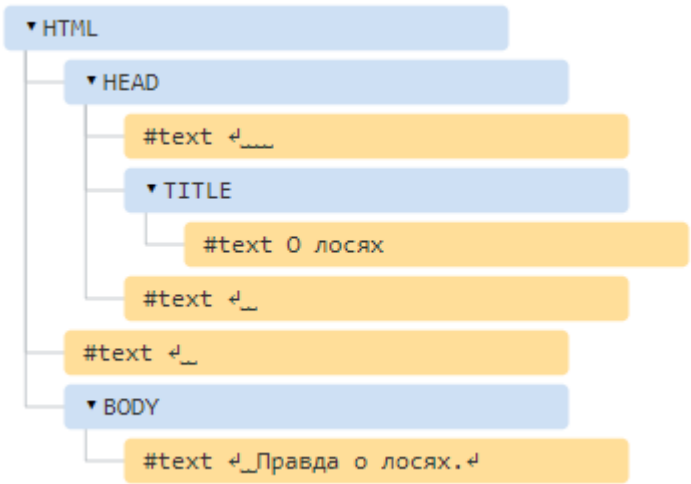
Это был лишь небольшой пример того, что может DOM. Скоро мы изучим много способов работать с DOM, но сначала нужно познакомиться с его структурой.

## Пример DOM

Начнём с такого, простого, документа:

```
<!DOCTYPE HTML>
<html>
<head>
  <title>О лосях</title>
</head>
<body>
  Правда о лосях.
</body>
</html>
```

DOM – это представление HTML-документа в виде дерева тегов. Вот как оно выглядит:




На рисунке выше узлы-элементы можно кликать, и их дети будут скрываться и раскрываться.

Каждый узел этого дерева – это объект.

Теги являются *узлами-элементами* (или просто элементами). Они образуют структуру дерева: <html> – это корневой узел, <head> и <body> его дочерние узлы и т.д.

Текст внутри элементов образует *текстовые узлы*, обозначенные как #text. Текстовый узел содержит в себе только строку текста. У него не может быть потомков, т.е. он находится всегда на самом нижнем уровне.

Например, в теге <title> есть текстовый узел "О лосях".

Обратите внимание на специальные символы в текстовых узлах:

- перевод строки: ↵ (в JavaScript он обозначается как \n)
- пробел: ␣

Пробелы и переводы строки – это полноправные символы, как буквы и цифры. Они образуют текстовые узлы и становятся частью дерева DOM. Так, в примере выше в теге <head> есть несколько пробелов перед <title>, которые образуют текстовый узел #text (он содержит в себе только перенос строки и несколько пробелов).

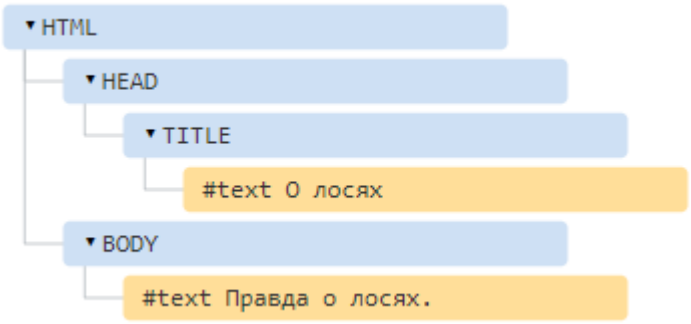
Существует всего два исключения из этого правила:

1. По историческим причинам пробелы и перевод строки перед тегом <head> игнорируются
2. Если мы записываем что-либо после закрывающего тега </body>, браузер автоматически перемещает эту запись в конец body, поскольку спецификация HTML требует, чтобы всё содержимое было внутри <body>. Поэтому после закрывающего тега </body> не может быть никаких пробелов.

В остальных случаях всё просто – если в документе есть пробелы (или любые другие символы), они становятся текстовыми узлами дерева DOM, и если мы их удалим, то в DOM их тоже не будет.

Здесь пробельных текстовых узлов нет:

```
<!DOCTYPE HTML>
<html><head><title>О лосях</title></head><body>Правда о лосях.</body></html>
```




Пробелы по краям строк и пробельные текстовые узлы скрыты в инструментах разработки. Когда мы работаем с деревом DOM, используя инструменты разработчика в браузере (которые мы рассмотрим позже), пробелы в начале/конце текста и пустые текстовые узлы (переносы строк) между тегами

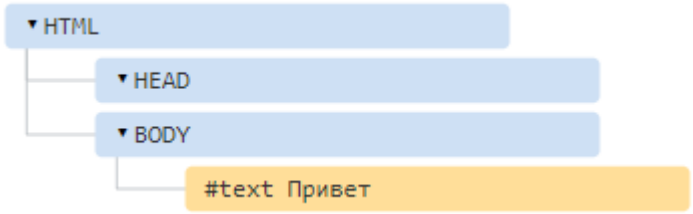
обычно не отображаются. Таким образом инструменты разработки экономят место на экране. В дальнейших иллюстрациях DOM мы также будем для краткости пропускать пробельные текстовые узлы там, где они не имеют значения. Обычно они не влияют на то, как отображается документ.

## Автоисправление

Если браузер сталкивается с некорректно написанным HTML-кодом, он автоматически корректирует его при построении DOM.

Например, в начале документа всегда должен быть тег <html>. Даже если его нет в документе – он будет в дереве DOM, браузер его создаст. То же самое касается и тега <body>.

Например, если HTML-файл состоит из единственного слова "Привет", браузер обернёт его в теги <html> и <body>, добавит необходимый тег <head>, и DOM будет выглядеть так:

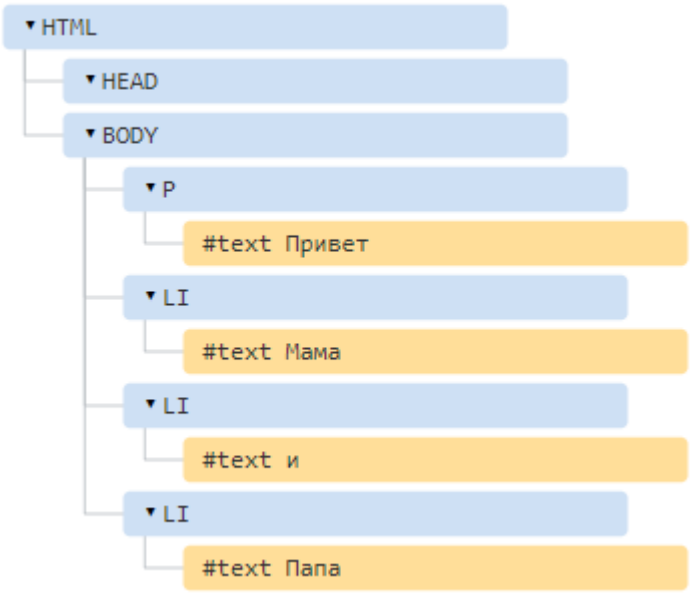



При генерации DOM браузер самостоятельно обрабатывает ошибки в документе, закрывает теги и так далее.

Есть такой документ с незакрытыми тегами:

```
<p>Привет
<li>Мама
<li>и
<li>Папа
```

…Но DOM будет нормальным, потому что браузер сам закроет теги и восстановит отсутствующие детали:



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

Даже директива <!DOCTYPE...>, которую мы ставим в начале HTML, тоже является DOM-узлом. Она находится в дереве DOM прямо перед <html>. Мы не будем рассматривать этот узел, мы даже не рисуем его на наших диаграммах, но он существует.

Даже объект document, представляющий весь документ, формально является DOM-узлом.

Существует 12 типов узлов. Но на практике мы в основном работаем с 4 из них:

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.

## Взаимодействие с консолью

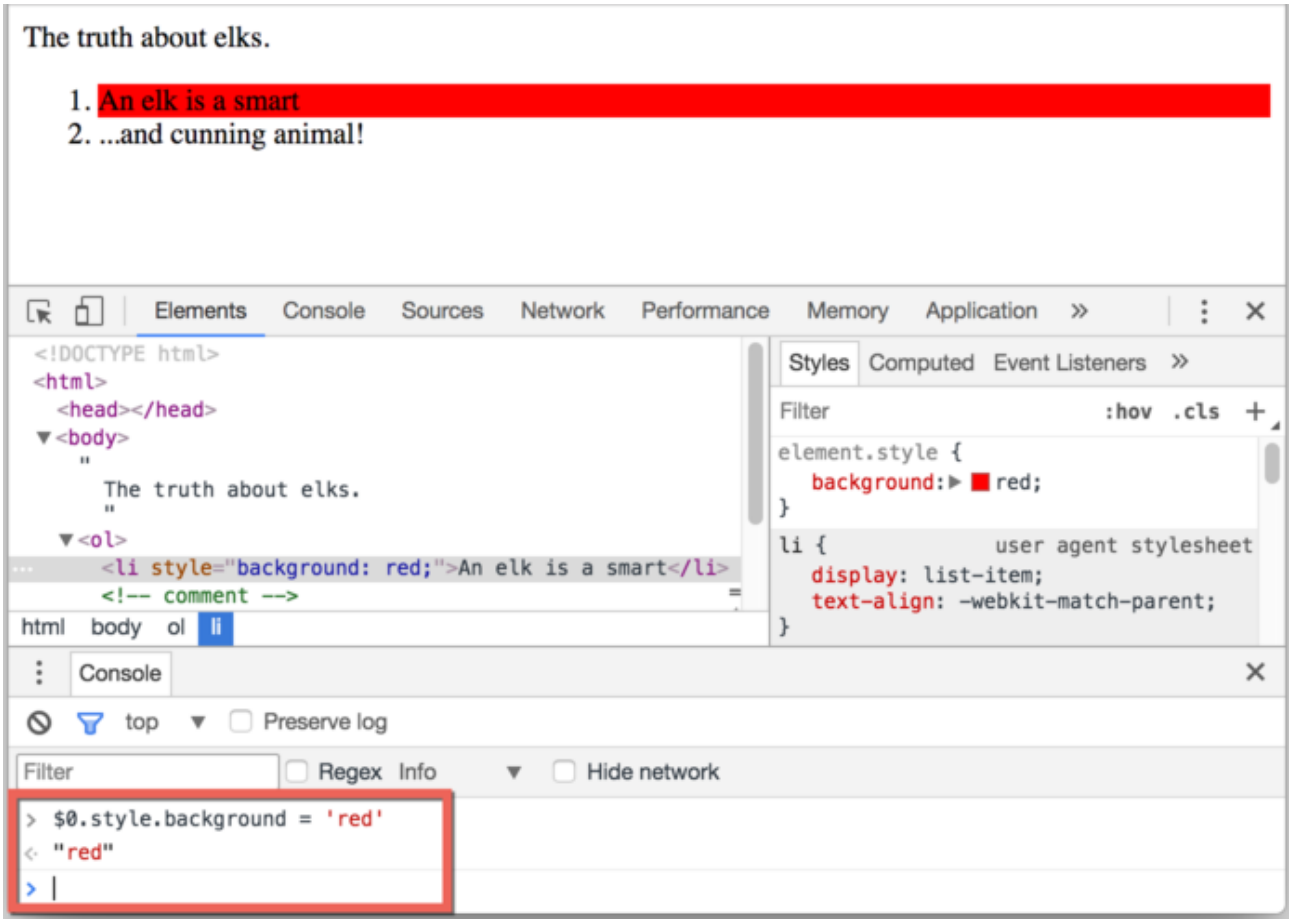
При работе с DOM нам часто требуется применить к нему JavaScript. Например: получить узел и запустить какой-нибудь код для его изменения, чтобы посмотреть результат. Вот несколько подсказок, как перемещаться между вкладками Elements и Console.

Для начала:

1. На вкладке Elements выберите первый элемент <li>.
2. Нажмите Esc – прямо под вкладкой Elements откроется Console.

Последний элемент, выбранный во вкладке Elements, доступен в консоли как $0; предыдущий, выбранный до него, как $1 и т.д.

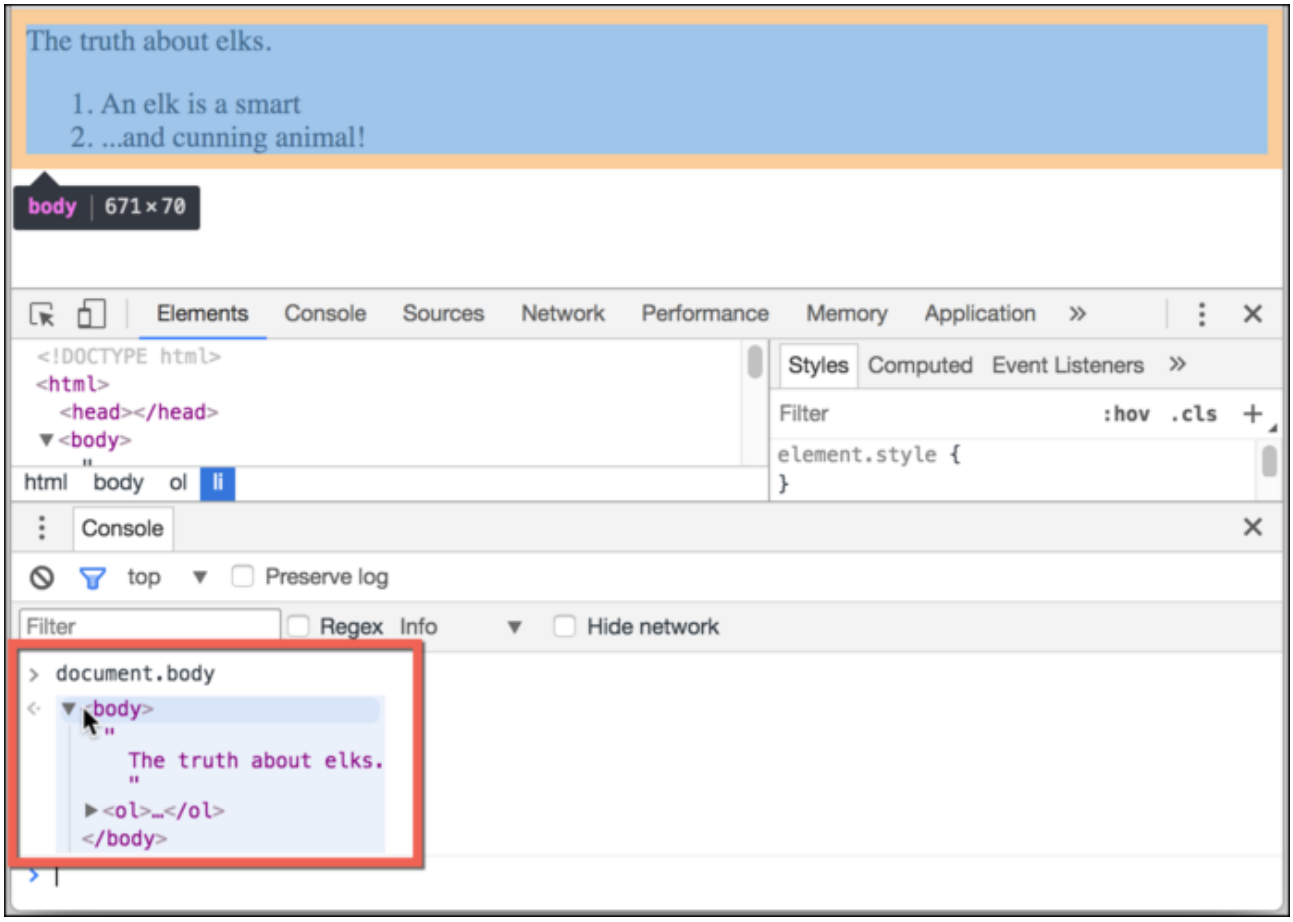
Теперь мы можем запускать на них команды. Например $0.style.background = 'red' сделает выбранный элемент красным, как здесь:




Это мы посмотрели, как получить узел из Elements в Console.

Есть и обратный путь: если есть переменная node, ссылающаяся на DOM-узел, можно использовать в консоли команду inspect(node), чтобы увидеть этот элемент во вкладке Elements.

Или мы можем просто вывести DOM-узел в консоль и исследовать «на месте», как document.body ниже:




Это может быть полезно для отладки. В следующей главе мы рассмотрим доступ и изменение DOM при помощи JavaScript.

Инструменты разработчика браузера отлично помогают в разработке: мы можем исследовать DOM, пробовать с ним что-то делать и смотреть, что идёт не так.

## Итого

HTML/XML документы представлены в браузере в виде DOM-дерева.

- Теги становятся узлами-элементами и формируют структуру документа.
- Текст становится текстовыми узлами.
- … и т.д. Всё, что записано в HTML, есть и в DOM-дереве, даже комментарии.

Для изменения элементов или проверки DOM-дерева мы можем использовать инструменты разработчика в браузере.

## Поиск: getElement*, querySelector*

Свойства навигации по DOM хороши, когда элементы расположены рядом. А что, если нет? Как получить произвольный элемент страницы?

Для этого в DOM есть дополнительные методы поиска.

### document.getElementById или просто id

Если у элемента есть атрибут id, то мы можем получить его вызовом document.getElementById(id), где бы он ни находился.

Например:

```
<div id="elem">
  <div id="elem-content">Element</div>
</div>

<script>
  // получить элемент
  let elem = document.getElementById('elem');

  // сделать его фон красным
  elem.style.background = 'red';
</script>
```

Пожалуйста, не используйте такие глобальные переменные для доступа к элементам. Это поведение соответствует стандарту, но поддерживается в основном для совместимости, как осколок далёкого прошлого.

Браузер пытается помочь нам, смешивая пространства имён JS и DOM. Это удобно для простых скриптов, которые находятся прямо в HTML, но, вообще говоря, не очень хорошо. Возможны конфликты имён. Кроме того, при чтении JS-кода, не видя HTML, непонятно, откуда берётся переменная.

В реальной жизни лучше использовать document.getElementById.

Значение id должно быть уникальным. Значение id должно быть уникальным. В документе может быть только один элемент с данным id. Если в документе есть несколько элементов с одинаковым значением id, то поведение методов поиска непредсказуемо. Браузер может вернуть любой из них случайным образом. Поэтому, пожалуйста, придерживайтесь правила сохранения уникальности id.

## querySelectorAll

Самый универсальный метод поиска – это elem.querySelectorAll(css), он возвращает все элементы внутри elem, удовлетворяющие данному CSS-селектору.

Следующий запрос получает все элементы <li>, которые являются последними потомками в <ul>:

```
<ul>
  <li>Этот</li>
  <li>тест</li>
</ul>
<ul>
  <li>полностью</li>
  <li>пройден</li>
</ul>
<script>
  let elements = document.querySelectorAll('ul > li:last-child');

  for (let elem of elements) {
    alert(elem.innerHTML); // "тест", "пройден"
  }
</script>
```

Этот метод действительно мощный, потому что можно использовать любой CSS-селектор.

### Псевдоклассы тоже работают

Псевдоклассы в CSS-селекторе, в частности :hover и :active, также поддерживаются. Например, document.querySelectorAll(':hover') вернёт коллекцию (в порядке вложенности: от внешнего к внутреннему) из текущих элементов под курсором мыши.

### querySelector

Метод elem.querySelector(css) возвращает первый элемент, соответствующий данному CSS-селектору.

Иначе говоря, результат такой же, как при вызове elem.querySelectorAll(css)[0], но он сначала найдёт *все* элементы, а потом возьмёт первый, в то время как elem.querySelector найдёт только первый и остановится. Это быстрее, кроме того, его короче писать.

### matches

Предыдущие методы искали по DOM.

Метод elem.matches(css) ничего не ищет, а проверяет, удовлетворяет ли elem CSS-селектору, и возвращает true или false.

Этот метод удобен, когда мы перебираем элементы (например, в массиве или в чём-то подобном) и пытаемся выбрать те из них, которые нас интересуют.

Например:

```
<a href="http://example.com/file.zip">...</a>
<a href="http://ya.ru">...</a>

<script>
  // может быть любая коллекция вместо document.body.children
  for (let elem of document.body.children) {
    if (elem.matches('a[href$="zip"]')) {
      alert("Ссылка на архив: " + elem.href );
    }
  }
</script>
```

## closest

*Предки* элемента – родитель, родитель родителя, его родитель и так далее. Вместе они образуют цепочку иерархии от элемента до вершины.

Метод elem.closest(css) ищет ближайшего предка, который соответствует CSS-селектору. Сам элемент также включается в поиск.

Другими словами, метод closest поднимается вверх от элемента и проверяет каждого из родителей. Если он соответствует селектору, поиск прекращается. Метод возвращает либо предка, либо null, если такой элемент не найден.

Например:

```
<h1>Содержание</h1>

<div class="contents">
  <ul class="book">
    <li class="chapter">Глава 1</li>
    <li class="chapter">Глава 2</li>
  </ul>
</div>

<script>
  let chapter = document.querySelector('.chapter'); // LI

  alert(chapter.closest('.book')); // UL
  alert(chapter.closest('.contents')); // DIV

  alert(chapter.closest('h1')); // null (потому что h1 - не предок)
</script>
```

## Итого

Есть 6 основных методов поиска элементов в DOM:

| Метод | Ищет по... | Ищет внутри элемента? | Возвращает живую коллекцию? |
|---|---|---|---|
| `querySelector` | CSS-selector | ✓ | - |
| `querySelectorAll` | CSS-selector | ✓ | - |
| `getElementById` | `id` | - | - |
| `getElementsByName` | `name` | - | ✓ |
| `getElementsByTagName` | tag or `'*'` | ✓ | ✓ |
| `getElementsByClassName` | class | ✓ | ✓ |

Безусловно, наиболее часто используемыми в настоящее время являются методы querySelector и querySelectorAll, но и методы getElement(s)By* могут быть полезны в отдельных случаях, а также встречаются в старом коде.

Кроме того:

- Есть метод elem.matches(css), который проверяет, удовлетворяет ли элемент CSS-селектору.
- Метод elem.closest(css) ищет ближайшего по иерархии предка, соответствующему данному CSS-селектору. Сам элемент также включён в поиск.

И, напоследок, давайте упомянем ещё один метод, который проверяет наличие отношений между предком и потомком:

- elemA.contains(elemB) вернёт true, если elemB находится внутри elemA (elemB потомок elemA) или когда elemA==elemB.