



Лекция 6

DOM

Что мы научимся делать?

- искать элементы;
- обращаться к элементам;
- менять содержимое HTML-документа;
- изменять стили;
- добавлять и удалять HTML-элементы;
- поговорим о шаблонах.

DOM (Document Object Model) –

это представление HTML-документа в виде **дерева объектов**, доступное для изменения через JavaScript. Из DOM легко можно управлять содержимым, стилями, значениями атрибутов.

Уровни DOM

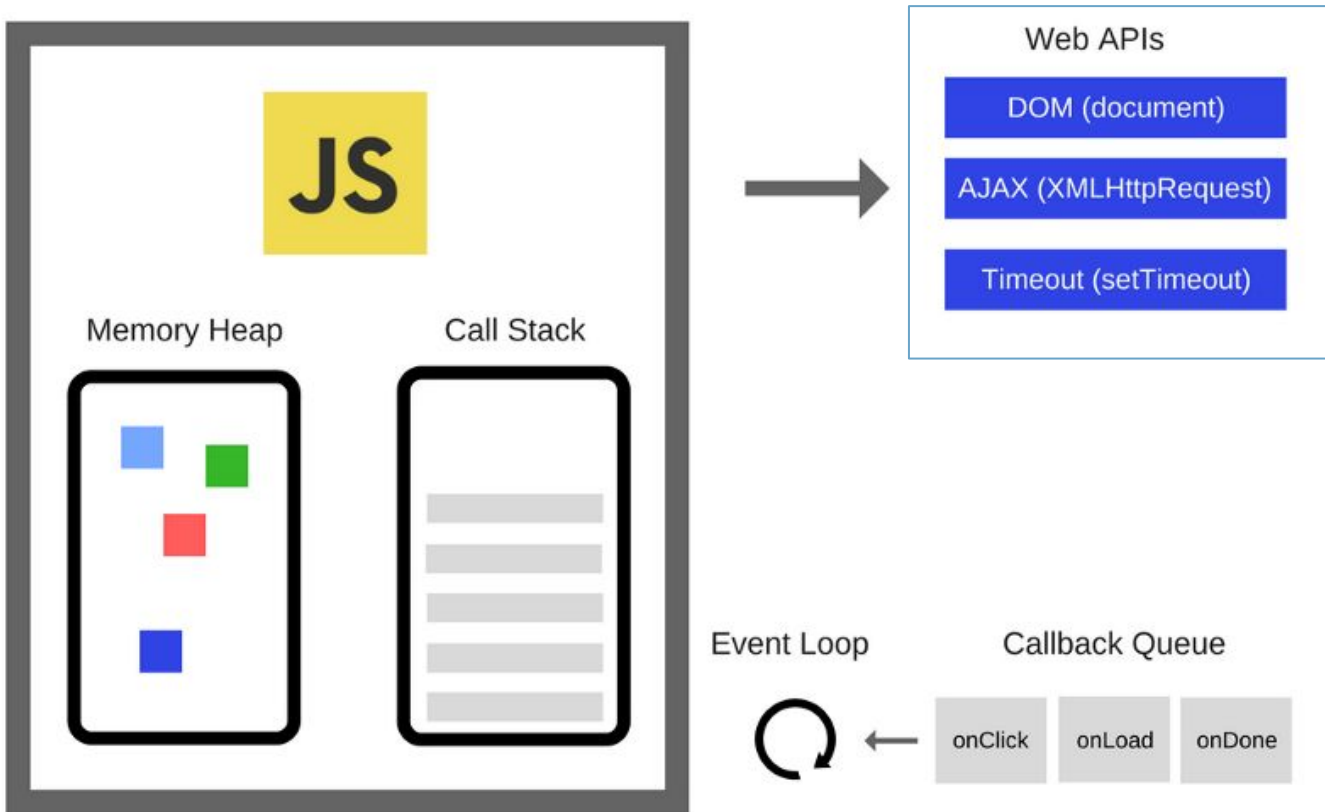
- **DOM 0** – изначально появившийся интерфейс, в котором содержатся некоторые примитивные методы и свойства. Не описан в каком-либо документе, но частично вошел в стандарт HTML 4.
- **DOM 1** – описал программные интерфейсы для работы с XML и с HTML. Появились некоторые коллекции элементов.

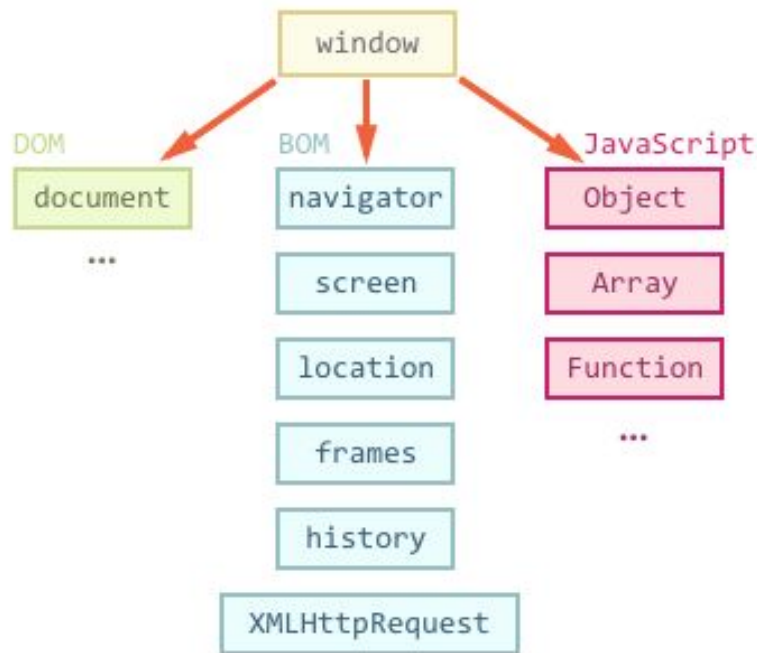
Уровни DOM

- **DOM 2** – наиболее важный уровень. Здесь описаны основные понятия и методы работы с XML и HTML. Впервые стандартизованы события, описана работа с CSS. Описаны **DOM2-Traversal-Range**, которые служат основой для обхода документа;
- **DOM 3** – привнес немного, но уменьшил количество багов и улучшил производительность;

Уровни DOM

- **DOM 4** — самый новый стандарт, улучшающий производительность и добавляющий новые методы для работы со страницей. Пока находится в активной разработке и много где доступен только с полифилами.

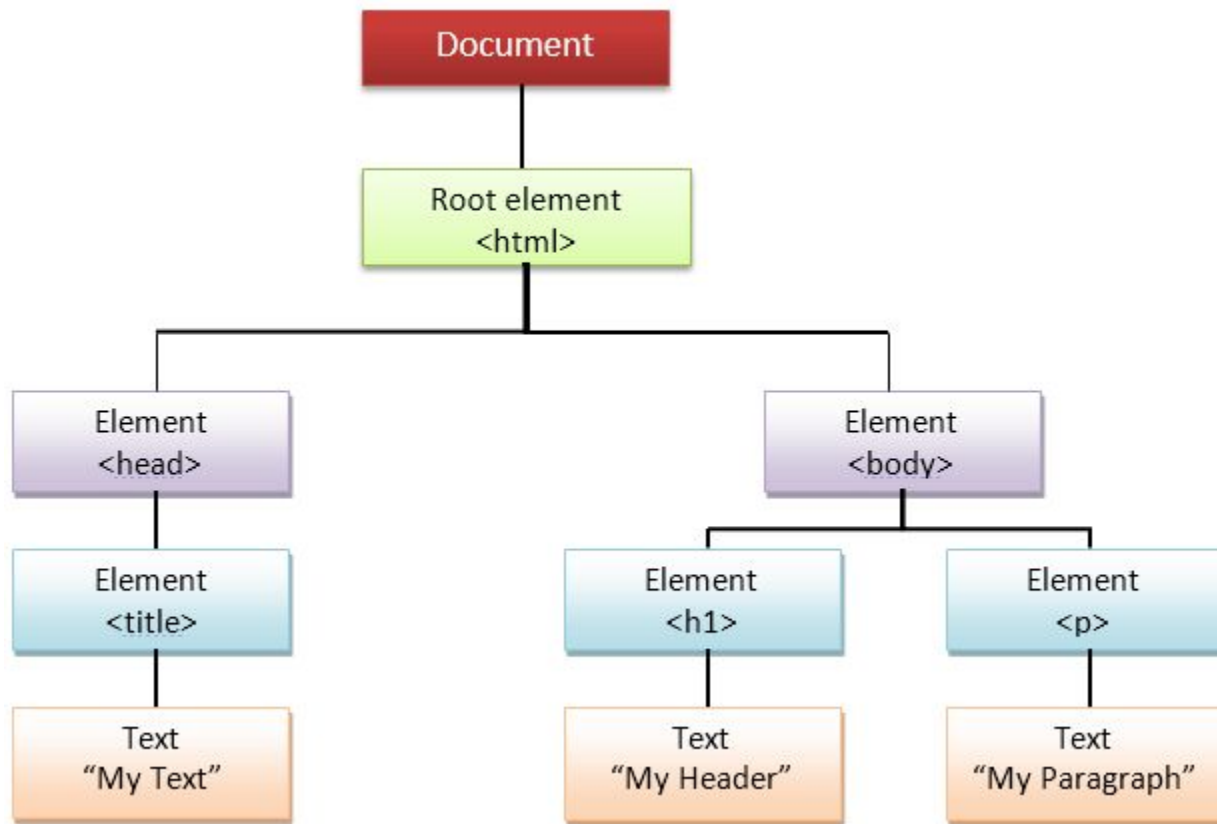




Сам по себе язык JavaScript не предусматривает работы с браузером.

Он вообще не знает про HTML. Но позволяет легко расширять себя новыми функциями и объектами.

Представление страницы в JavaScript



Некоторые основные узлы (ноды):

- Теги образуют **узлы-элементы** (element node). Естественным образом одни узлы вложены в другие. Структура дерева образована исключительно за счет них.
- Текст внутри элементов образует **текстовые узлы** (text node). Текстовый узел содержит исключительно строку текста и не может иметь потомков, то есть он всегда на самом нижнем уровне.

Как найти элемент на странице?

HTML коллекции

- `document.all;`
- `document.anchors;`
- `document.body;`
- `document.documentElement;`
- `document.embeds;`
- `document.forms;`
- `document.head;`
- `document.images;`
- `document.links;`
- `document.scripts;`
- `document.title.`

Поиск по классу, тегу или идентификатору

`document.getElementById(elementId: DOMString)` –

принимает в качестве аргумента идентификатор и возвращает элемент.

`document.getElementsByClassName(classNames: DOMString)` –

принимает в качестве аргумента любой класс и возвращает коллекцию.

`document.getElementsByTagName(localName: DOMString)` –

принимает в качестве аргумента любой тег и возвращает коллекцию.

Первый метод всегда
возвращает ноду, а второй и
третий – коллекцию

Коллекция != не массив

Они похожи на массивы, поскольку имеют индексы и к ним можно обратиться, как к обычному массиву (`element[1]`, `form[2]`). Однако, у них **отсутствуют методы массивов** и для их перебора используют цикл ***for***.

Для перебора также нельзя использовать ***for ... in***, который кроме индексов будет выводить еще и лишнюю для нас информацию.

Крутые методы querySelectorAll и querySelector

`document.querySelectorAll(selectors: DOMString)` –

принимает в качестве аргумента любой CSS-селектор и возвращает **NodeList**.

`document.querySelector(selectors: DOMString)` –

принимает в качестве аргумента любой CSS-селектор и возвращает **первый найденный** на странице элемент. Аналог `document.querySelectorAll('.someclass')[0]`, однако работает быстрее.

NodeList != HTMLCollection

В отличие от коллекций, NodeList имеет в прототипе некоторые удобные методы. Поэтому при использовании `querySelector()` или `querySelectorAll()` можно перебирать полученный псевдомассив при помощи **forEach**.

matches и matchMedia

`window.matchMedia(query: string)` –

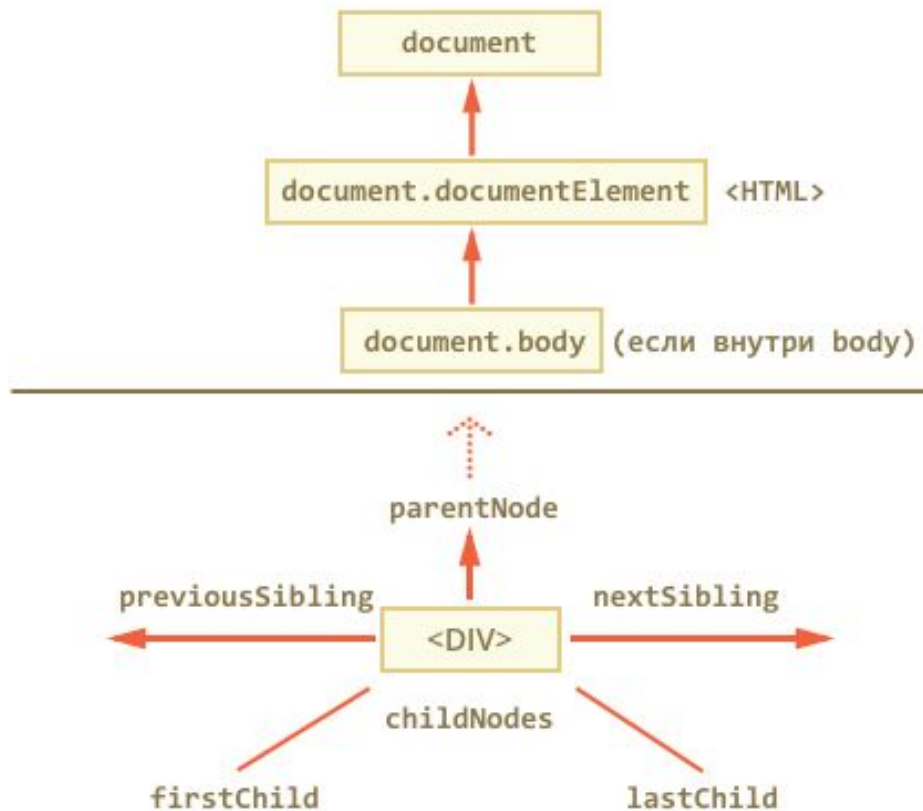
принимает в качестве аргумента @media-выражение и возвращает true или false в зависимости от его истинности.

`document.matches(selectors: DOMString)` –

проверяет, удовлетворяет ли заданный элемент указанному селектору. Возвращает true или false и бывает очень полезен при переборе элементов и в связке с `matchMedia()`.

Навигация по дереву

Навигация по дереву



У каждой ноды есть набор свойств, при помощи которых можно удобно обращаться к соседним, дочерним и родительским.

Это часто бывает необходимо при создании сложных интерфейсов.

Навигация от дочернего к родительскому

Node.parentElement – возвращает родителя узла DOM Element, или null если узел не имеет родителя, или его родитель не DOM Element
принимает в качестве аргумента @media-выражение и возвращает true или false в зависимости от его истинности. DOM 4

Node.parentNode – возвращает родителя определенного элемента DOM дерева. Возвращает null если элемент только был создан и еще не добавлен в DOM дерево. DOM 2-3

Навигация от родительского к дочерним

Node.children – возвращает живую коллекцию (HTMLCollection) дочерних элементов узла. Если у узла детей нет, она будет пустой. Коллекция не включает текстовые узлы и комментарии. **DOM 1**

Node.childNodes – возвращает коллекцию дочерних элементов данного элемента включая текстовые узлы и комментарии. **DOM 2-3**

Навигация от родительского к дочерним

Node.firstChild – возвращающее первого потомка узла в дереве или null, если узел является бездетным. Если узел это document, он возвращает первый узел в списке своих прямых детей. DOM 1-2

Node.lastChild – возвращает последнего потомка в узле. DOM 2

Node.previousSibling – возвращает узел предшествующий указанному в родительском элементе childNodes, или null, если указанный узел первый в своём родителе. DOM 1-2-3

Навигация от родительского к дочерним

`Node.nextSibling` – возвращает узел, непосредственно следующий за данным узлом в списке `childNodes` его родительского элемента, или `null` если данный узел последний в этом списке.

DOM 1-2

`Node.previousElementSibling` и **`Node.nextElementSibling`** – возвращают только элементные ноды.

DOM 1-2

Как влиять на HTML?

Как писать в документе?

`document.write(text...: DOMString)` –

дописывает текст в документ, пока он еще не загружен. В качестве аргумента можно передать любой текст, на страницу он вставляется, как есть. Является небезопасным и используется очень редко.

`document.getElementById(id).innerHTML` –

позволяет получить содержимое элемента в виде строки. Этим же методом строку можно присвоить указанному элементу. Метод является безопасным, поскольку браузер автоматически исправляет ошибки.

Как получать и изменять значения атрибута?

`document.element.attribute` –

позволяет получить значение атрибута.

`document.element.attribute = 'new value';` –

изменяет значение атрибута.

Атрибуты

Как получать и изменять значения атрибута?

`element.hasAttribute(name: DOMString)` – принимает атрибут и возвращает true, если он задан или false, если нет;

`element.removeAttribute(name: DOMString)` – принимает атрибут и удаляет его;

`element.getAttribute(name: DOMString)` – принимает атрибут и возвращает его значение;

`element.setAttribute(name: DOMString, value: DOMString)` – принимает атрибут, его значение и устанавливает его.

data-* атрибуты

Зачем нужны data-* атрибуты?

HTML5 спроектирован с возможностью расширения данных ассоциированных с каким-либо элементом, но в то же время не обязательно имеющих определённое значение. data-* атрибуты позволяют хранить дополнительную информацию в стандартных элементах HTML, без хаков вроде нестандартных атрибутов, лишних DOM-свойств или **Node.setUserData()**.

Как работать с data-* атрибутами?

Для работы с атрибутами можно использовать метод `getAttribute()` с параметром, равным полному имени атрибута. Но есть и более простой способ, используя объект `dataset`.

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

```
article.dataset.columns // "3"
```

```
article.dataset.indexNumber // "12314"
```

```
article.dataset.parent // "cars"
```

ClassList API

Как получать и изменять значения атрибута?

`element.classList.add(String [,String])` – добавляет элементу указанные классы;

`element.classList.remove(String [,String])` – удаляет у элемента указанные классы;

`element.classList.toggle(name: DOMString)` – если класс у элемента отсутствует - добавляет, а иначе - убирает;

`element.classList.contains(name: DOMString)` – проверяет, есть ли данный класс у элемента (вернет true или false).

Как получать и изменять значения атрибута?

`element.classList.item(Number)` – результат аналогичен вызову
`element.classList[Number];`

`element.classList.length` – возвращает количество классов у
элемента;

Как влиять на CSS?

Как получать и изменять значения стилей?

`element.style.property` –

возвращает строку со значением указанного свойства.

`element.style.property = 'value'` –

задает свойству указанное значение.

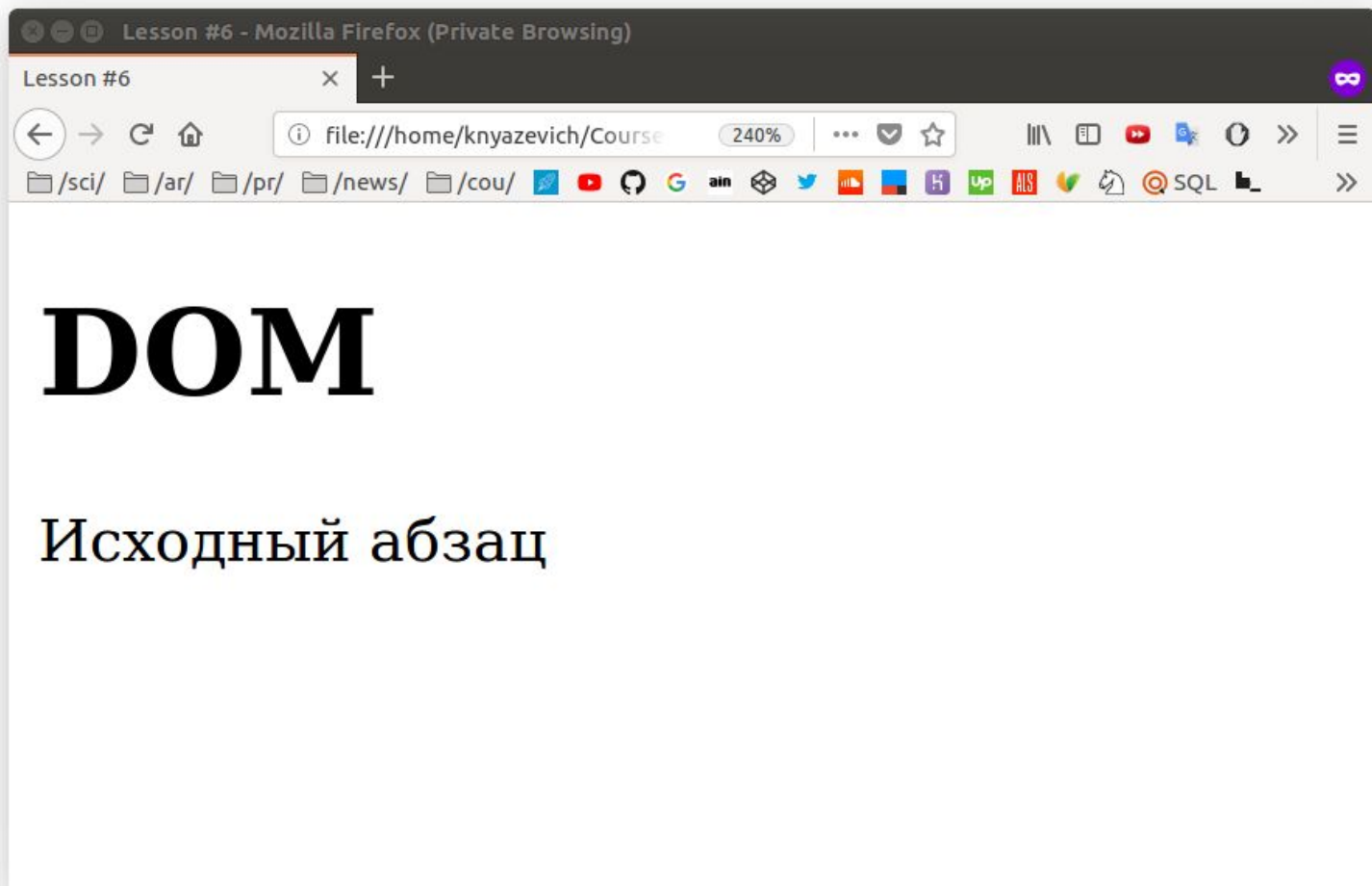
Если свойство состоит из нескольких слов, записанных через дефис, то к нему обращаются, **используя camelCase** (*backgroundColor, borderRadius, webkitBoxSizing*).

Как получать и изменять значения стилей?

```
element.style.cssText = `  
    background-color: #ddd;  
    color: #111;  
    border: 1px solid #010101;  
`;
```

позволяет указать сразу несколько свойств для элемента.

Добавляем и удаляем ноды






```
<body>
  <h1>DOM</h1>

  <!-- Блок для нового элемента -->
  <div class="container">
    <p class="child">Исходный абзац</p>
  </div>

  <!-- Подключим скрипт на страницу -->
  <script src="dom.js"></script>
</body>
```

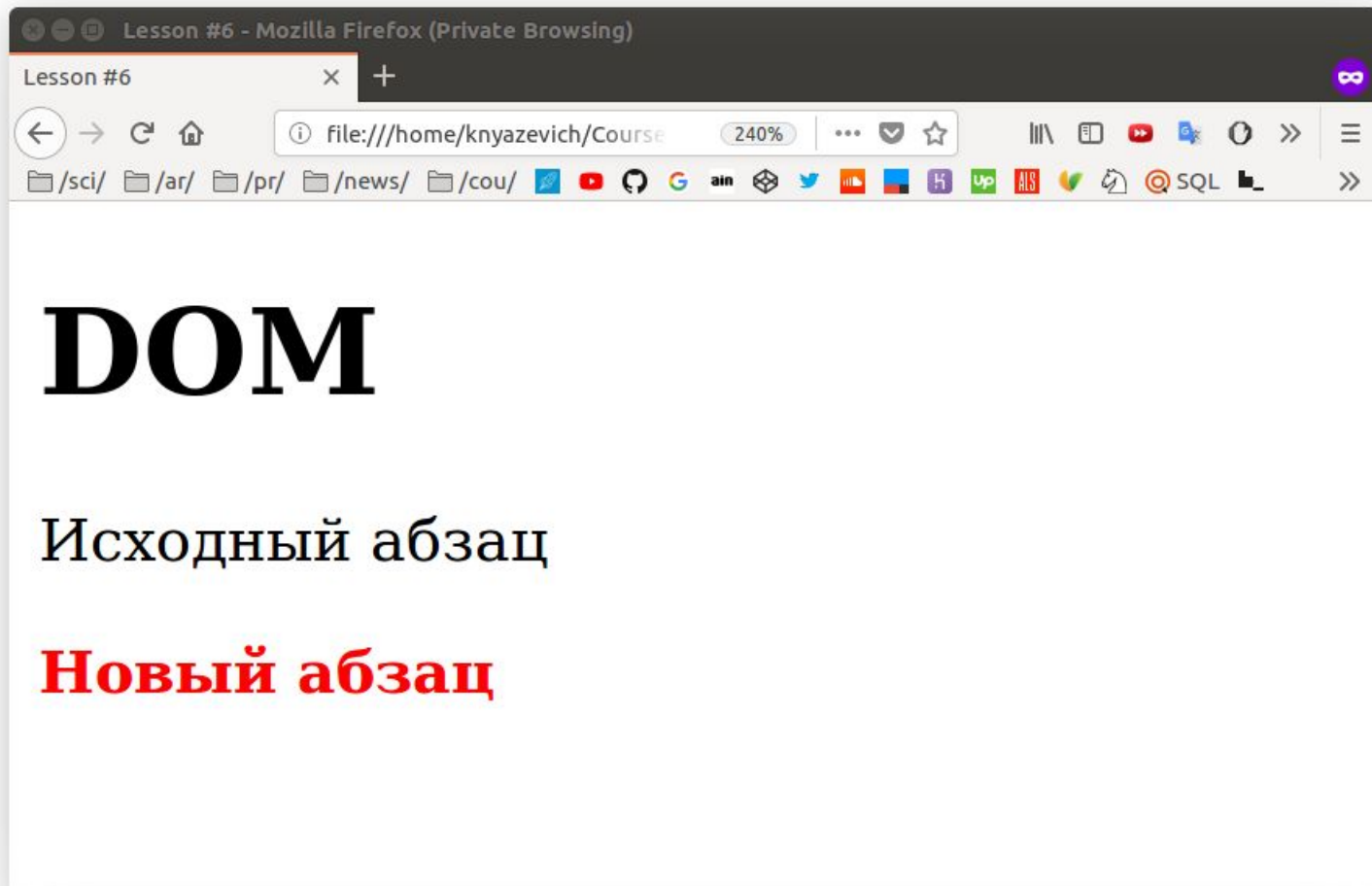
```
// Создаем элементную ноду
const paragraph = document.createElement( 'p' );

// Создаем текстовую ноду
const textNode = document.createTextNode( 'Новый абзац' );

// После этого мы добавляем текстовую ноду в элементную:
paragraph.appendChild( textNode );

// Немного стилизуем
paragraph.style.color = 'red';
paragraph.style[ 'font-weight' ] = '700';

// Теперь новый элемент можно добавить в HTML
const container = document.querySelector( '.container' );
container.appendChild( paragraph );
```



Удаляем элемент



```
const container = document.querySelector('.container');  
const child = document.querySelector('.child');  
  
container.removeChild(child);  
  
// Можно использовать метод element.remove(), который  
// работает аналогично, но воспринимается легче  
  
child.remove();
```

