

DOCUMENT OBJECT MODEL



ИЛЬЯ МЕДЖИДОВ



ИЛЬЯ МЕДЖИДОВ

CEO в RageMarket

 medzhidov@ragemarket.ru

ПЛАН ЗАНЯТИЯ

1. Вспоминаем DOM
2. DOM – события
3. Полноценная работа с атрибутами html-элементов
4. Работаем с классами как современный человек
5. Работаем со стилями (CSSOM, ты ли это?)
6. Определяем размеры и позицию элемента на странице
7. Немного работы с элементом
8. Навигация по элементам
9. Вспоминаем `onscroll`



ВСПОМНИМ ПРОШЛЫЕ ЗАНЯТИЯ

Вопрос: какие основные события в JavaScript вы знаете?

ВСПОМНИМ ПРОШЛЫЕ ЗАНЯТИЯ

Вопрос: какие основные события в JavaScript вы знаете?

Ответ:

- События мыши:
 - `click` – клик на левую кнопку мыши;
 - `contextmenu` – клик на правую кнопку мыши;
 - `mouseover` – когда мышь наводится на элемент;
 - `ondblclick` – двойной клик мыши;
- События клавиатуры:
 - `onkeypress` – когда нажимаем на кнопку;
 - `onkeyup` – когда отпускаем кнопку;
- События форм и элементов:
 - `onfocus` – когда элемент в фокусе;
 - `onblur` – когда элемент теряет фокус;
 - `onsubmit` – когда отправляем форму.



ВСПОМИНАЕМ DOM

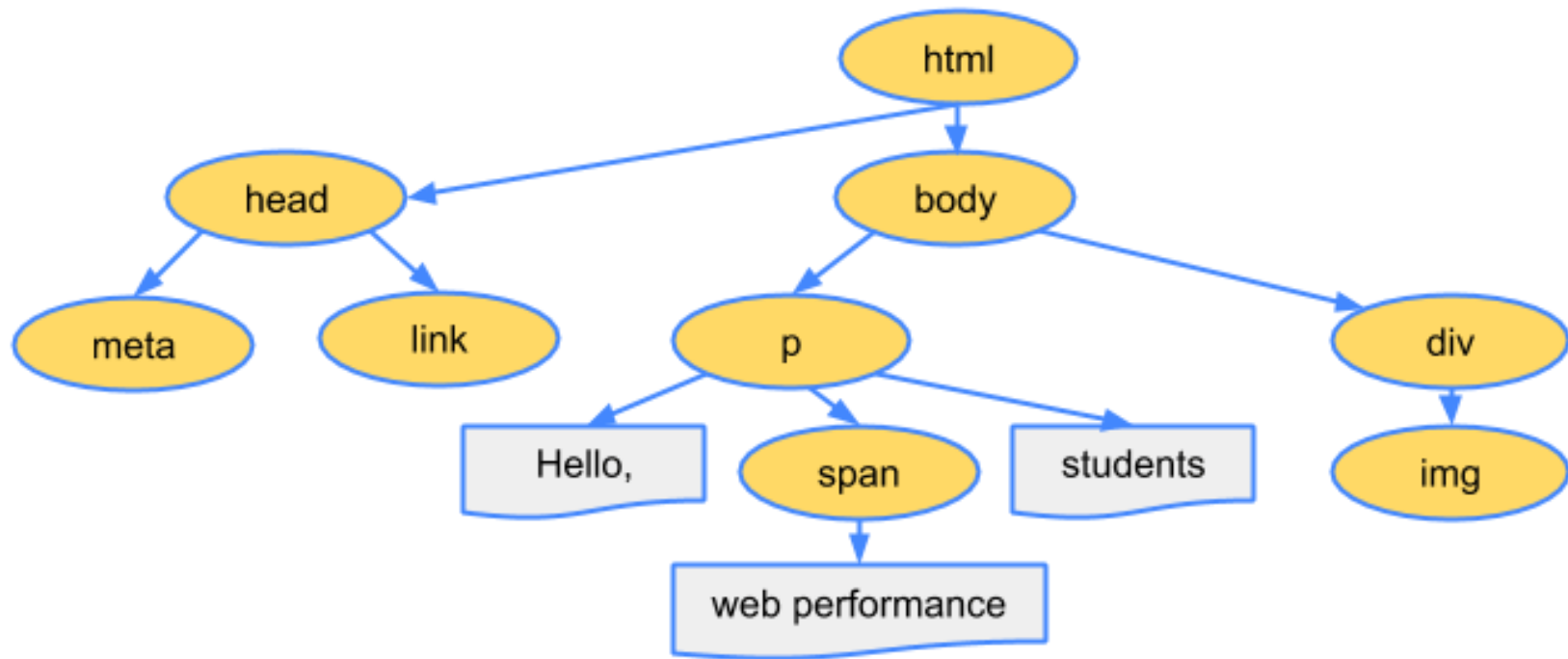


DOM

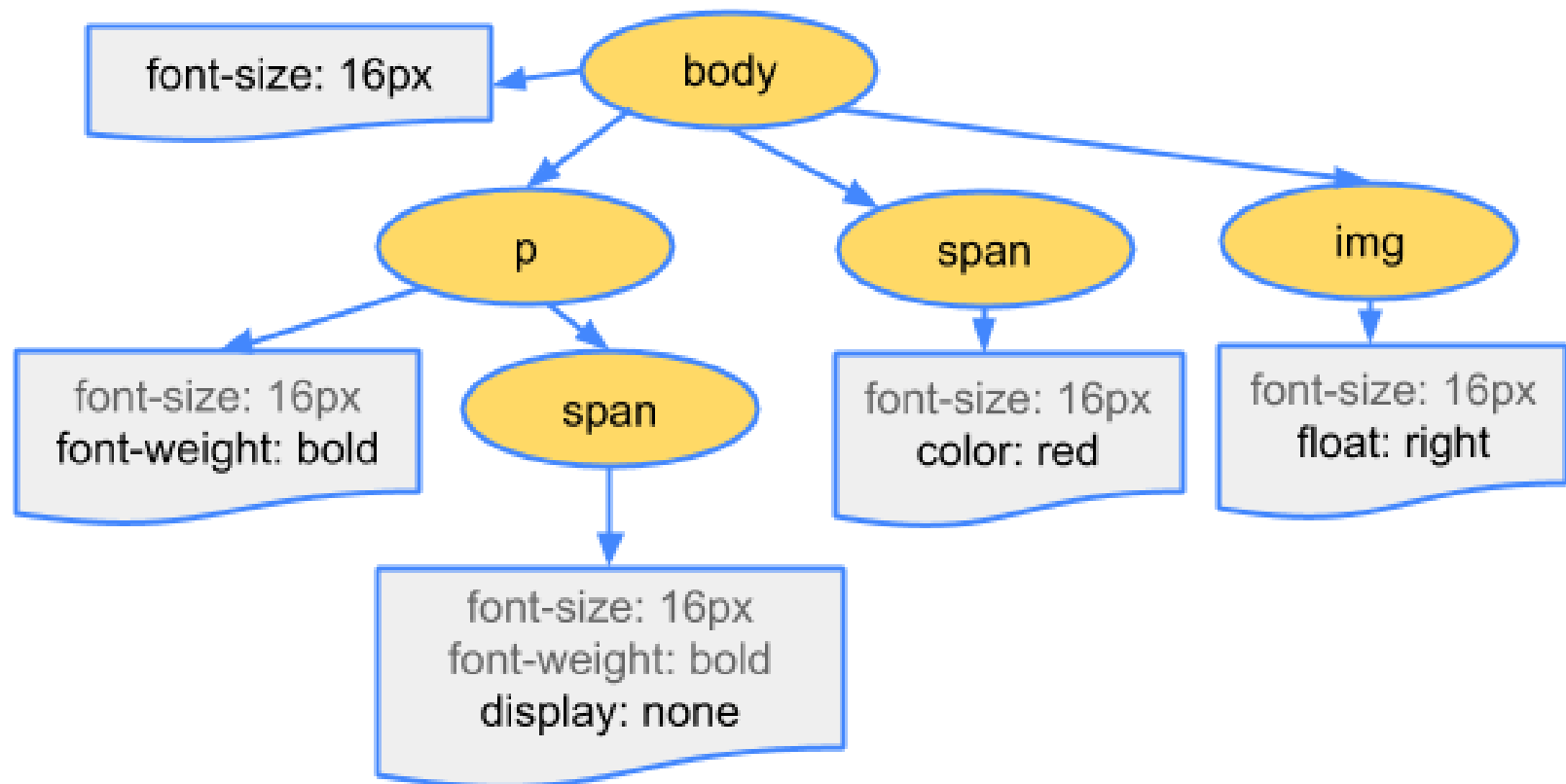
Как мы помним из первого занятия, DOM – объектная модель документа, которая формируется браузером перед тем, как вывести страницу на экран.

Вспоминаем этапы из лекции 1.1.

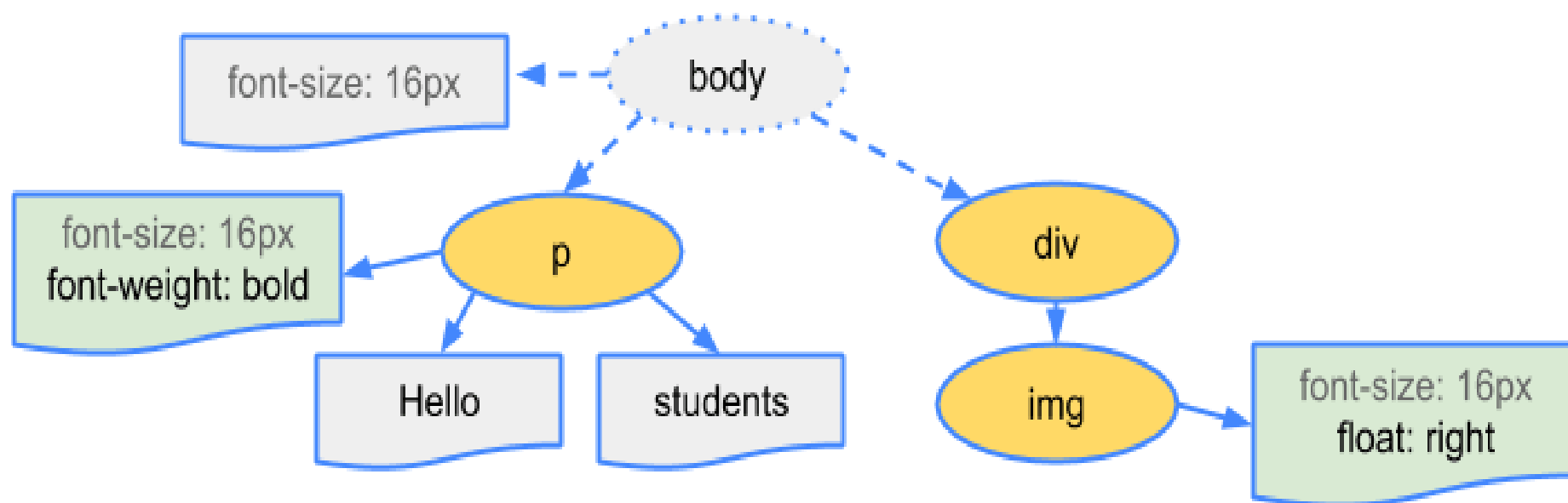
ПОСТРОЕНИЕ ОБЪЕКТНОЙ МОДЕЛИ ДОКУМЕНТА (DOM)



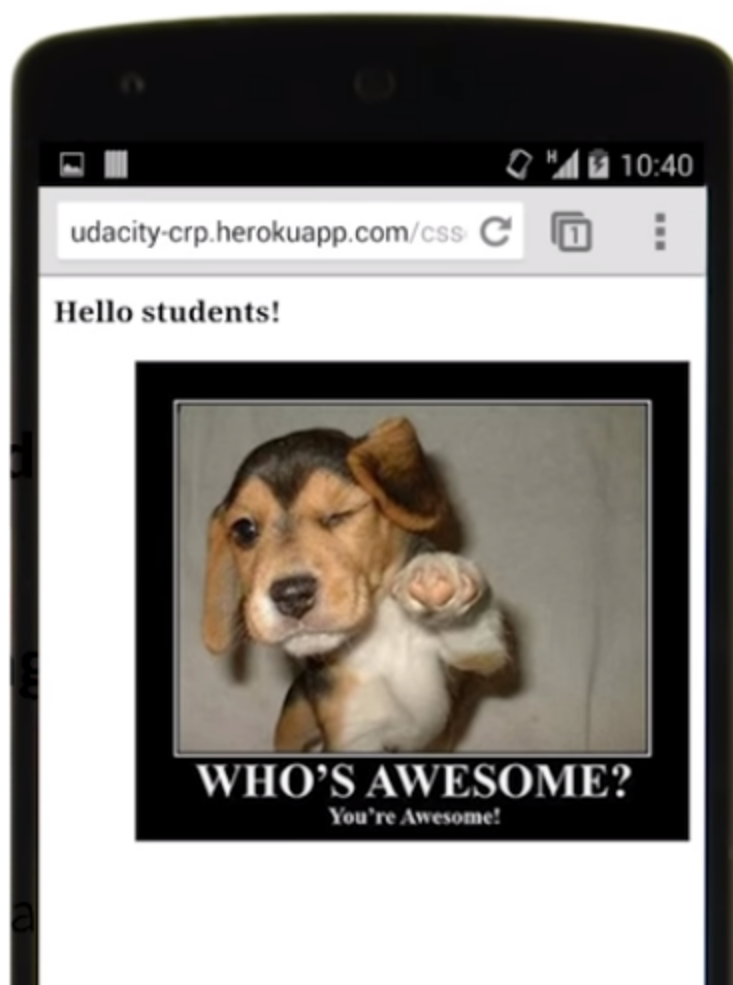
ПОСТРОЕНИЕ ОБЪЕКТНОЙ МОДЕЛИ CSS (CSSOM)



ПОСТРОЕНИЕ МОДЕЛИ ВИЗУАЛИЗАЦИИ



ОТРИСОВКА МАКЕТА СТРАНИЦЫ





DOM – СОБЫТІЯ



DOM – СОБЫТИЯ ЗАГРУЗКИ

Теперь мы познакомимся с событиями, которые позволяют нам отследить тот момент, когда страница загружена браузером и готова к работе.

Например, для того, чтобы запустить ту часть скрипта, что должна работать с DOM только после того, как он будет полностью сформирован.

Для этого существует два события.

DOMContentLoaded

Событие `DOMContentLoaded` срабатывает сразу после того, как браузер полностью загрузил HTML и построил DOM-дерево. Не дожидаясь загрузки и применения стилей\картинок\iframe и прочего.

```
1 <script>
2   const ready = function() {
3     const paragraphs = document.querySelectorAll("p");
4     console.log(paragraphs.length); // 5
5   }
6
7   document.addEventListener("DOMContentLoaded", ready);
8 </script>
9
10 <p>Абзац 1</p>
11     ...
12 <p>Абзац 5</p>
```

window.onload

Событие `load` на объекте `window` срабатывает только после того, как загрузится вся страница, включая все ресурсы на ней (стили\картинки\iframe и т.п.):

```
1  <script>
2      const load = function() {
3          console.log("Все картинки и ресурсы загружены");
4          console.log("Даже стили применены");
5      }
6
7      window.addEventListener("load", load);
8      // аналогично window.onload = load;
9  </script>
10
11  <link rel="stylesheet" href="style.css">
12  
13  <iframe src="iframe.html"></iframe>
```

ВОПРОСЫ

- Сможем ли мы узнать размер изображения по событию `DOMContentLoaded`?
- Всегда ли будет построен DOM по событию `window.onload`?

РЕЗЮМИРУЕМ И ОТВЕТЫ НА ВОПРОСЫ

- `DOMContentLoaded` на объекте `document` срабатывает сразу после построения DOM;
- `load` на объекте `window` срабатывает только после загрузки всех ресурсов на странице, включая стили\картинки\iframe и прочее.

А КАК ЖЕ АТТРИБУТЫ `defer` И `async` ?

Давайте рассмотрим, когда эти скрипты будут выполнены:

- `script` с атрибутом `defer` будет выполнен сразу после построения DOM, прямо перед вызовом события `DOMContentLoaded`;
- `script` с атрибутом `async` будет выполнен в тот же момент, как только загрузится. Это может произойти до того, как браузер успеет построить DOM.


ВОПРОС (РИТОРИЧЕСКИЙ)

А что будет лучше – использовать `script` с атрибутом `defer` или использовать `script` с атрибутом `async`, но внутри последнего выполнять код по событию `DOMContentLoaded`?

ОТВЕТ

Если нам не важно, когда будет выполнен конкретный скрипт, но важно, чтобы он был выполнен после загрузки DOM – используем `async` и событие `DOMContentLoaded`.

Если же нам нужно вызвать скрипт в определенный момент, после или перед выполнением других скриптов – используем `defer`.



ПОЛНОЦЕННАЯ РАБОТА С АТТРИБУТАМИ HTML- ЭЛЕМЕНТОВ

ПОЛНОЦЕННАЯ РАБОТА С АТТРИБУТАМИ HTML-ЭЛЕМЕНТОВ

Помимо того, что мы можем обращаться к стандартным атрибутам html-элементов через одноименные свойства (например, `src` у тега `img`), мы также можем устанавливать и читать абсолютно любые атрибуты html-элементов через два удобных метода.

element.getAttribute()

`element.getAttribute()` – возвращает значение указанного атрибута элемента. Если элемент не содержит данный атрибут, могут быть возвращены `null` или `""` (пустая строка).

```
1 <div not-usual="любое значение"></div>
2
3 <script>
4   const div = document.querySelector("div");
5   const attr = div.getAttribute("not-usual");
6   console.log(attr); // "любое значение"
7 </script>
```

element.setAttribute()

`element.setAttribute()` – добавляет новый атрибут или изменяет значение существующего атрибута у выбранного элемента. Принимает два аргумента: название атрибута и значение, которое нужно установить.

```
1 <div></div>
2
3 <script>
4   const div = document.querySelector("div");
5   div.setAttribute("abc", "любое значение")
6
7   let attrAbc = div.getAttribute("abc");
8   console.log(attrAbc) // "любое значение"
9 </script>
```

При этом при применении `element.setAttribute()` мы редактируем DOM, а значит, в коде HTML после выполнения предыдущего скрипта мы увидим:

```
<div abc="любое значение"></div>
```


ХРАНИМ ДАННЫЕ В dataset

Теперь мы знаем методы, при помощи которых можно создавать и читать какие угодно атрибуты html-элементов. Так почему бы там не хранить какие-либо данные, которые могут нам понадобиться в скриптах или стилях?

Для этого существуют специальные `data` атрибуты. Они отличаются как записью в html, так и способом работы с ними.

```
1 <!-- Обычный атрибут. Так писать нежелательно -->
2 <div custom="значение"></div>
3 <!-- Data-атрибут -->
4 <div data-custom="значение"></div>
```

Да, в html вся разница лишь в том, что перед названием атрибута мы используем префикс `data-`.

ХРАНИМ ДАННЫЕ В dataset

Для того, чтобы прочитать или записать значение в `data`-атрибут, следует обратиться к нему как к свойству в `dataset` конкретного элемента, но используя `camelCase` нотацию:

```
1 <div data-custom-name="значение"></div>
2
3 <script>
4   const div = document.querySelector("div");
5   // Прочитаем значение
6   console.log(div.dataset.customName) // "значение"
7   // Запишем новое значение
8   div.dataset.customName = "новое значение 1"
9   // абсолютно идентично
10  div.dataset['customName'] = "новое значение 2"
11  // При этом стандартный способ также будет работать
12  let dataAttr = div.getAttribute("data-custom-name");
13  console.log(dataAttr); // "новое значение 2"
14 </script>
```

ВОПРОСЫ

- Можно ли задавать или читать `data`-атрибуты через `get/setAttribute`?
- В чём разница между обычными атрибутами и `data`-атрибутами?

РЕЗЮМИРУЕМ И ОТВЕТЫ НА ВОПРОСЫ

- `element.getAttribute()` – позволяет получить значение любого атрибута элемента;
- `element.setAttribute(name, value)` – позволяет установить любое значение любому атрибуту элемента;
- `element.dataset` – содержит все `data`-атрибуты элемента в виде свойств, но в нотации `camelCase`. Их можно как прочесть, так и записать в них новые значения.

**РАБОТАЕМ С КЛАССАМИ
КАК СОВРЕМЕННЫЙ
ЧЕЛОВЕК**

РАБОТАЕМ С КЛАССАМИ ПО-СОВРЕМЕННОМУ

Все вы помните, как мы работали с классами через `element.className`, который хранит все классы в виде одной строки с пробелами.

Давайте посмотрим на код, который позволяет при помощи свойства `className` добавить или удалить класс `selected` у элемента `div` при клике на него.

className

```
1 <div class='red'>Нажми на меня</div>
2
3 <script>
4   const div = document.querySelector('div');
5
6   function toggleSelectedClass() {
7     const classNames = div.className.split(' ');
8     const index = classNames.indexOf('selected');
9     if (index === -1) {
10      classNames.push('selected');
11    } else {
12      classNames.splice(index, 1);
13    }
14    div.className = classNames.join(' ');
15  }
16
17  div.addEventListener('click', toggleSelectedClass);
18 </script>
```

Из-за того, что `className` – строка, работать с несколькими классами не очень удобно. Поэтому появился новый интерфейс для работы с ними – через свойство `classList` и его методы.

classList

`classList` недоступен в старых версиях Internet Explorer (в 9 и ниже). Для полной информации по этому вопросу лучше обратиться к [caniuse](https://caniuse.com/classlist)

Вся работа происходит через методы объекта `classList`, вот основные:

- `add()` – для добавления класса;
- `remove()` – для удаления;
- `contains()` – для проверки, установлен ли такой класс или нет;
- `toggle()` – для переключения класса (если он уже был, то будет удален, если его не было – будет добавлен).

Тогда функцию `toggleSelectedClass()` из прошлого примера можно заметно укоротить:

```
function toggleSelectedClass() {  
    btn1.classList.toggle('selected');  
}
```

ПРИМЕР

Давайте придумаем пример посложнее. Будем добавлять класс `selected` только тем объектам-узлам, где уже есть класс `red`.

```
1 <button class='red'>Нажми на меня</button>
2 <button>Нажми на меня</button>
3
4 <script>
5     const buttons = document.querySelectorAll('button');
6
7     function addSelectedClassIfRed() {
8         if (this.classList.contains('red')) {
9             this.classList.add('selected');
10        }
11    }
12
13    for (const btn of buttons) {
14        btn.onclick = addSelectedClassIfRed;
15    }
16 </script>
```

РЕЗЮМИРУЕМ

Свойство `classList` имеет удобные методы для работы с классами html-элемента: `add` \ `remove` \ `toggle` \ `contains`.

В то время как свойство `className` всего лишь возвращает строковое значение атрибута `class`.



РАБОТАЕМ СО СТИЛЯМИ

РАБОТАЕМ СО СТИЛЯМИ ЧЕРЕЗ DOM

Каждому html-элементу можно задать стили внутри атрибута `style`. И, пользуясь этой лазейкой, в DOM было создано специальное свойство для html-элементов, которое позволяет задавать стили элементу, записывая их в этот атрибут.

Для этого существует свойство `element.style`, которое содержит все перечисленные в атрибуте `style` свойства, но названия этих свойств будут в `camelCase` нотации.

```
1 <div style="margin-top: 0px;">Текст</div>
2
3 <script>
4     const div = document.querySelector("div");
5     console.log(div.style.marginTop); // "0px"
6 </script>
```

Также через свойство `element.style` мы можем записать любое css-свойство, которое будет добавлено в атрибут `style` и применено браузером:

```
1 <div style="margin-top: 0px;">Текст</div>
2
3 <script>
4   const div = document.querySelector("div");
5   div.style.marginTop = "20px";
6
7   let styleAttr = div.getAttribute('style');
8   console.log(styleAttr) // "margin-top: 20px;"
9 </script>
```

ПОЛУЧАЕМ АКТУАЛЬНЫЕ ЗНАЧЕНИЯ СТИЛЕЙ

Учитывая, что свойство `style` содержит только инлайн-стили, как же нам узнать реальные применяемые стили для данного элемента, с учетом CSS-каскада? Для этого есть глобальная функция

`getComputedStyle(element[, ':pseudo'])`, которая возвращает вычисленные значения стилей для элемента. Для `getComputedStyle` важно указывать полное название свойства, например `marginTop` вместо `margin`.

Но есть одно но. Использовать `getComputedStyle` для считывания геометрии объекта не лучшая идея. Например, полученная таким образом ширина будет зависеть от `box-sizing`. А для inline-элементов и вовсе будет выдавать `auto`.

РЕЗЮМИРУЕМ

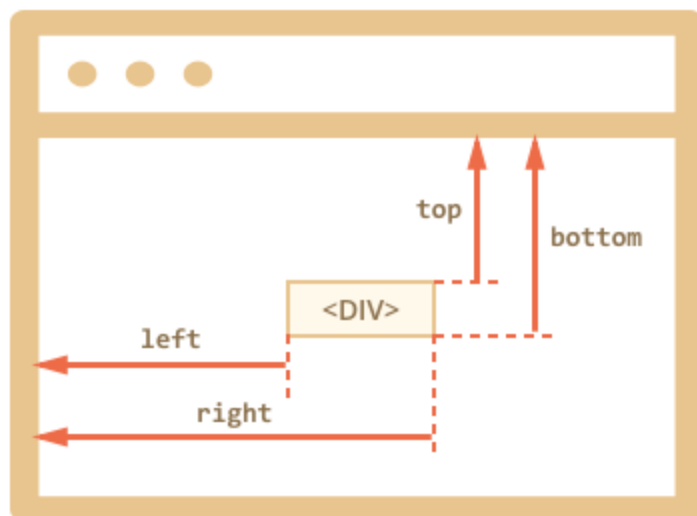
- Свойство `element.style` позволяет работать с inline стилями любого html-элемента через одноименные свойства, но в `camelCase` нотации;
- Метод `getComputedStyle(element[, ':pseudo'])` позволяет получить вычисленные браузером актуальные значения стилей, в том числе не заданных явно через CSS.

ОПРЕДЕЛЯЕМ РАЗМЕРЫ И ПОЗИЦИЮ ЭЛЕМЕНТА НА СТРАНИЦЕ

ОПРЕДЕЛЯЕМ РАЗМЕРЫ И ПОЗИЦИЮ ЭЛЕМЕНТА НА СТРАНИЦЕ

Метод `element.getBoundingClientRect()` возвращает координаты элемента, под которыми понимаются размеры «воображаемого прямоугольника», который охватывает весь элемент.

Координаты возвращаются в виде объекта со свойствами:



- `top` – Y-координата верхней границы элемента;
- `left` – X-координата левой границы;
- `right` – X-координата правой границы;
- `bottom` – Y-координата нижней границы.

Эти координаты высчитываются от границ видимой области окна браузера, поэтому, если прокрутить страницу выше или ниже, координаты изменятся.

Так, если мы пролистаем ниже элемента и он пропадет за верхней частью окна браузера – значения `top` и `bottom` будут отрицательными, но также будут рассчитаны.

Теперь мы можем решить тривиальную задачу: обнаружить, появился ли элемент внутри видимой области браузера, используя дополнительное свойство `window.innerHeight`, которое всегда возвращает актуальную высоту видимой области внутри окна браузера:

```
1 <div></div>
2
3 <script>
4   const isInViewport = function(element){
5     const viewportHeight = window.innerHeight;
6     const elementTop =
7       element.getBoundingClientRect().top;
8
9     return elementTop > viewportHeight ? true : false;
10  };
11
12  const div = document.querySelector('div');
13  isInViewport(div);
14 </script>
```

Код, приведенный выше, не совершенен и учитывает только один нюанс: долистал ли пользователь до элемента или нет. Если он его уже пролистал и элемент пропал из области видимости – функция всё равно вернёт true. А исправлять этот нюанс вам придётся в своём домашнем задании.

Также некоторые современные браузеры добавляют к результату `getBoundingClientRect` свойства для ширины и высоты: `width` / `height`, но их можно получить и простым вычитанием: `height = bottom - top`, `width = right - left`.

РЕЗЮМИРУЕМ

- Метод `element.getBoundingClientRect()` позволяет получить положение элемента относительно видимой области окна браузера, а также получить или рассчитать высоту и ширину элемента;
- Свойство `window.innerHeight` содержит текущую высоту видимой области внутри окна браузера, `window.innerWidth` – ширину.



НЕМНОГО РАБОТЫ С ЭЛЕМЕНТОМ

ПОЛУЧАЕМ БОЛЬШЕ ИНФОРМАЦИИ ОБ ЭЛЕМЕНТЕ

Иногда нам необходимо узнать некоторую информацию о конкретном элементе. Например, когда внутри функции нам нужно совершить разные операции над абзацем или `div` 'ом, но мы не уверены, что именно из этого нам будет передано.

Для этого можно воспользоваться несколькими простыми свойствами узла:

- `Element.tagName` – содержит название тега в верхнем регистре, например `div.tagName == "DIV"`;
- `Node.nodeName` – содержит название узла, но в случае, если узлом является html-элемент – вернёт `tagName`;
- `Element.textContent` – содержит весь текст внутри элемента;
- `Node.nodeValue` – содержимое текстового узла или комментария. При применении к html-элементу всегда вернёт `null`.

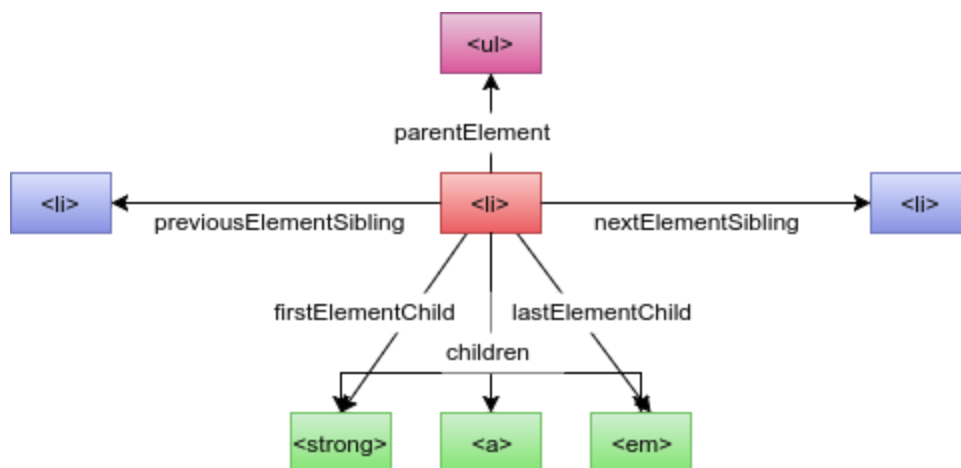


НАВИГАЦИЯ ПО ЭЛЕМЕНТАМ

НАВИГАЦИЯ ПО ЭЛЕМЕНТАМ

Очевидно, что в дереве DOM у любого элемента есть родительский элемент, а также могут быть соседи и дочерние элементы.

Для доступа к ним у каждого элемента DOM-дерева есть свойства, указывающие на его родителя, детей и соседей:



СВОЙСТВА

Давайте рассмотрим все свойства, а после попытаемся при помощи них решить простую задачу:

- `parentElement` – *родитель-элемент*;
- `previousElementSibling`, `nextElementSibling` – соседи-элементы, *предыдущий* и *следующий*;
- `firstElementChild`, `lastElementChild` – соответственно, *первый* и *последний* дочерний элемент;
- `children` – только *дочерние узлы-элементы*.

ЗАДАЧА

Делаем кнопку, которая будет скрывать модальное окно при клике на неё, добавляя класс `hidden` к родителю `.modal`:

```
1 <section class="modal">
2   <p>
3     Это модельное окно, их несколько на странице,
4     поэтому использовать поиск родителя по классу нельзя
5   </p>
6   <button>Закрыть</button>
7 </section>
8
9 <script>
10   const closeModal = function() {
11     const modalParent = this.parentElement;
12     modalParent.classList.add('hidden');
13   };
14
15   const modalButtons = document.querySelectorAll('.modal button');
16
17   for ( const button of modalButtons ) {
18     button.addEventListener('click', closeModal);
19   }
20 </script>
```

Но совершенно небольшое изменение приведет к полной неработоспособности скрипта:


```
1 <section class="modal">
2   <p>
3     Это модельное окно, их несколько на странице,
4     поэтому использовать поиск родителя по классу нельзя
5   </p>
6   <div>
7     <button>Закрыть</button>
8   </div>
9 </section>
10
11 <script>
12   const closeModal = function() {
13     const modalParent = this.parentElement;
14     modalParent.classList.add('hidden');
15     // Класс будет добавлен элементу `div`
16   };
17
18   const modalButtons = document.querySelectorAll('.modal button');
19
20   for ( const button of modalButtons ) {
21     button.addEventListener('click', closeModal);
22   }
23 </script>
```

Да, мы можем переписать наш скрипт, заменив `.parentElement` на `.parentElement.parentElement`, получив второго по старшинству родителя. Но что будет, когда их станет 4? А если 10?

Задачу может облегчить простой метод `Element.closest()`. Он возвращает ближайший родительский элемент (или сам элемент), который соответствует заданному CSS-селектору. Или `null`, если таковых элементов вообще нет.

Давайте перепишем наш код используя новый метод:

```
1 <section class="modal">
2   <p>
3     Это модельное окно, их несколько на странице,
4     поэтому использовать поиск родителя по классу нельзя
5   </p>
6   <div>
7     ...
8     <div>
9       <button>Закрыть</button>
10    </div>
11    ...
12  </div>
13 </section>
14
15 <script>
16   const closeModal = function(){
17     const modalParent = this.closest('.modal');
18     modalParent.classList.add('hidden');
19     // Класс всегда будет добавлен родителю `.modal`
20   };
21
22   const modalButtons = document.querySelectorAll('.modal button');
23
24   for ( const button of modalButtons ){
25     button.addEventListener('click', closeModal);
26   }
27 </script>
```



Также, зачастую в JS нет необходимости обращаться к следующему или к предыдущему элементу, опять же из-за возможности изменения структуры, поэтому вместо `previous\nextElementSibling` часто используют удобную связку `closest` и `querySelector`:


```
1 <section class="modal">
2   <p>
3     Этот текст будет заменен
4   </p>
5   <div>
6     ...
7     <div>
8       <button>Закрыть</button>
9     </div>
10    ...
11  </div>
12 </section>
13
14 <script>
15   const changeText = function () {
16     const modalParent = this.closest('.modal');
17     const modalParagraph = modalParent.querySelector('p');
18     modalParagraph.textContent = "Вы нажали на кнопку!";
19   };
20
21   const modalButtons = document.querySelectorAll('.modal button');
22
23   for (const button of modalButtons) {
24     button.addEventListener('click', changeText);
25   }
26 </script>
```

Есть также ещё пара методов, что могут вам сэкономить кучу времени:

- `Element.matches()` – вернёт `true` или `false`, в зависимости от того, соответствует ли элемент указанному CSS-селектору.
- `Element.contains(child)` – возвращает `true`, если `Element` содержит `child` или `Element == child`.

НАВИГАЦИЯ ПО УЗЛАМ В DOM

Выше мы рассмотрели способы навигации по элементам DOM, но также есть и способ навигации **по узлам**. Это значит, что если функция `nextElementSibling` вернет нам следующий html-элемент, то соответствующая ей функция для навигации по узлам вернёт нам любой следующий узел и не важно, будет это элемент, текст или комментарий. Конечно, названия этих методов отличаются, но мы просто их перечислим, чтобы знать:

- `parentNode` – узел-родитель;
- `previousSibling`, `nextSibling` – предыдущий и следующий узел;
- `firstChild`, `lastChild` – первый и последний дочерний узел;
- `childNodes` – все дочерние узлы.

РЕЗЮМИРУЕМ

- Для навигации по DOM-элементам можно использовать:
`parentElement`, `nextElementSibling`,
`previousElementSibling`, `firstElementChild`,
`lastElementChild`, `children`;
- Но гораздо удобнее использовать связку двух методов:
`Element.closest` (ближайший родительский элемент, который соответствует заданному CSS-селектору) и `Element.querySelector`.
Из-за возможного внесения изменений в структуру html-кода;
- Есть также способ навигации непосредственно по узлам с похожими по названию методами;
- Можно сравнить любой элемент на соответствие css-селектору при помощи метода `Element.matches`;
- И проверить, содержит ли один элемент внутри себя другой при помощи `Element.contains(childElement)`.



ВСПОМИНАЕМ
onscroll


onscroll

Помните метод `onscroll` ?

В любом случае он вам понадобится при выполнении домашнего задания. И поскольку вы уже взрослые и знаете, что такое `addEventListener` – в домашнем задании нужно будет работать с данным событием именно через этот метод.

ФИНАЛЬНОЕ РЕЗЮМЕ

- Новые события для выполнения кода после загрузки DOM и после загрузки всех ресурсов браузера;
- Новые методы для работы с атрибутами html-элементов;
- Специальное свойство для работы с `data`-атрибутами;
- Удобная работа с классами через `Element.classList`;
- Работаем со стилями через `Element.style`;
- Определяем размеры и положение элемента через `getBoundingClientRect`;
- Умеем перемещаться по DOM во всех направлениях. Как по узлам, так и по элементам.



МАТЕРИАЛЫ, ИСПОЛЬЗОВАННЫЕ ПРИ ПОДГОТОВКЕ ЛЕКЦИИ

- <https://learn.javascript.ru/traversing-dom>
- <https://developer.mozilla.org/.../classList>
- https://developer.mozilla.org/.../Using_data_attributes
- <https://developer.mozilla.org/.../getComputedStyle>

ДОМАШНЕЕ ЗАДАНИЕ

Давайте посмотрим ваше [домашнее задание](#).

- Вопросы по домашней работе задаем в Slack!
- Работы должны соответствовать принятому [стилю оформления кода](#).
- Зачет по домашней работе проставляется после того, как приняты все **3 задачи**.



Задавайте вопросы и напишите отзыв о лекции!

ИЛЬЯ МЕДЖИДОВ

 medzhidov@ragemarket.ru