нетология

Конспект по теме 2.1 Document Object Model

Содержание конспекта:

1. События Document Object Model
2. Работа с атрибутами HTML-элементов
3. Работа с классами
4. Определение размеров и позиций элементов
5. Навигация по элементам

1. События Document Object Model

DOM — объектная модель документа, которая формируется браузером перед тем, как вывести страницу на экран.

Познакомимся с событиями, которые позволяют нам отследить тот момент, когда страница загружена браузером и готова к работе. Это пригодится, чтобы запустить ту часть скрипта, что должна работать с DOM только после того, как он будет полностью сформирован.

Для этого существует два события:

1. 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:

```
<script>
  const ready = function() {
  const paragraphs = document.querySelectorAll("p");
    console.log(paragraphs.length); // 5
  }
  document.addEventListener("DOMContentLoaded", ready);
</script>

<p>Абзац 1</p>
    ...
<p>Абзац 5</p>
```

2. Событие 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.:

```
<script>
  const load = function() {
    console.log("Все картинки и ресурсы загружены");
    console.log("Даже стили применены");
  }

  window.addEventListener("load", load);
  // аналогично window.onload = load;
</script>

<link rel="stylesheet" href="style.css">
<img src="image.jpg">
<iframe src="iframe.html"></iframe>
```

Итоги

- 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.

Атрибуты defer и async

Рассмотрим, когда эти скрипты будут выполнены:

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

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 определённый момент, после или перед выполнением других скриптов — используем **defer**.

2. Работа с атрибутами HTML-элементов

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

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

```
<div not-usual="любое значение"></div>

<script>
  const div = document.querySelector("div");
  const attr = div.getAttribute("not-usual");
  console.log(attr); // "любое значение"
</script>
```

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

```
<div></div>

<script>
  const div = document.querySelector("div");
  div.setAttribute("abc", "любое значение")

  let attrAbc = div.getAttribute("abc");
  console.log(attrAbc) // "любое значение"
</script>
```

Применяя **element.setAttribute**(), мы редактируем DOM, а, значит, в коде HTML после выполнения предыдущего скрипта мы увидим:

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

Хранение данных в dataset

Для того, чтобы сохранять произвольные данные в атрибут элемента, существуют специальные data-атрибуты. С точки зрения HTML разница в том, что такие атрибуты начинаются с префикса data-. В JS же есть специальный интерфейс dataset для доступа к таким атрибутам:

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

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:

```
<div data-custom-name="значение"></div>

<script>
  const div = document.querySelector("div");
  // Прочитаем значение
  console.log(div.dataset.customName) // "значение"
  // Запишем новое значение
  div.dataset.customName = "новое значение 1"
  // абсолютно идентично
  div.dataset['customName'] = "новое значение 2"
  // При этом стандартный способ также будет работать
  let dataAttr = div.getAttribute("data-custom-name");
  console.log(dataAttr); // "новое значение 2"
</script>
```

Итоги

- 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 прочитать,так и записать в них новые значения.

3. Работа с классами

Свойство **className** объекта-узла хранит классы как единую строку, разделённую пробелами. И это не очень удобно, если нужно добавить или удалить какой-то класс.
В современных браузерах есть более удобный способ работать с классами — свойство **classList**.
Ранее 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:

```
<div class='red'>Нажми на меня</div>

<script>
  const div = document.querySelector('div');

  function toggleSelectedClass() {
    const classNames = div.className.split(' ');
    const index = classNames.indexOf('selected');
    if (index === -1) {
      classNames.push('selected');
    } else {
      classNames.splice(index, 1);
    }
    div.className = classNames.join(' ');
  }

  div.addEventListener('click', toggleSelectedClass);
</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.

Свойство **classList** недоступно в версиях Internet Explorer от 9 и старше.
Для полной информации по этому вопросу лучше обратиться к caniuse.

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 удалён, если его не было — будет добавлен.

Тогда функцию **toggleSelectedClass**() из прошлого примера можно заметно сократить:

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

Более сложный пример. Будем добавлять класс **selected** только тем объектам-узлам, где уже есть класс **red**:

```
<button class='red'>Нажми на меня</button>
<button>Нажми на меня</button>

<script>
  const buttons = document.getQuerySelectorAll('button');

    function addSelectedClassIfRed() {
      if (this.classList.contains('red')) {
        this.classList.add('selected');
      }
    }

  for (const btn of buttons) {
    btn.onclick = addSelectedClassIfRed;
  }
</script>
```

Итоги

- Свойство classList имеет удобные методы для работы с классами HTML-элемента: add, remove, toggle, contains.
- Свойство className всего лишь возвращает строковое значение атрибута class.

Работа со стилями через DOM

Каждому HTML-элементу можно задать стили внутри атрибута **style**. 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. В JS названия стилей нужно записывать через camelCase, а не через дефисы:

```
<div style="margin-top: 0px;">Текст</div>

<script>
  const div = document.querySelector("div");
  console.log(div.style.marginTop); // "0px"
</script>
```

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

```
<div style="margin-top: 0px;">Текст</div>

<script>
  const div = document.querySelector("div");
```

```
  div.style.marginTop = "20px";

  let styleAttr = div.getAttribute('style');
  console.log(styleAttr) // "margin-top: 20px;"
</script>
```

Получение актуальных значений стилей

Учитывая, что свойство **style** содержит только inline-стили, можно узнать реальные применяемые стили для данного элемента с учётом CSS-каскада. Для этого есть глобальная функция **getComputedStyle**(element[, ':pseudo']), которая возвращает вычисленные значения стилей для элемента. Для **getComputedStyle** важно указывать полное название свойства: **marginTop** вместо **margin**.
Но использовать **getComputedStyle** для считывания геометрии объекта не лучшее решении, поскольку полученная таким образом ширина будет зависеть от box-sizing, а для inline-элементов и вовсе будет выдавать auto.

Итоги

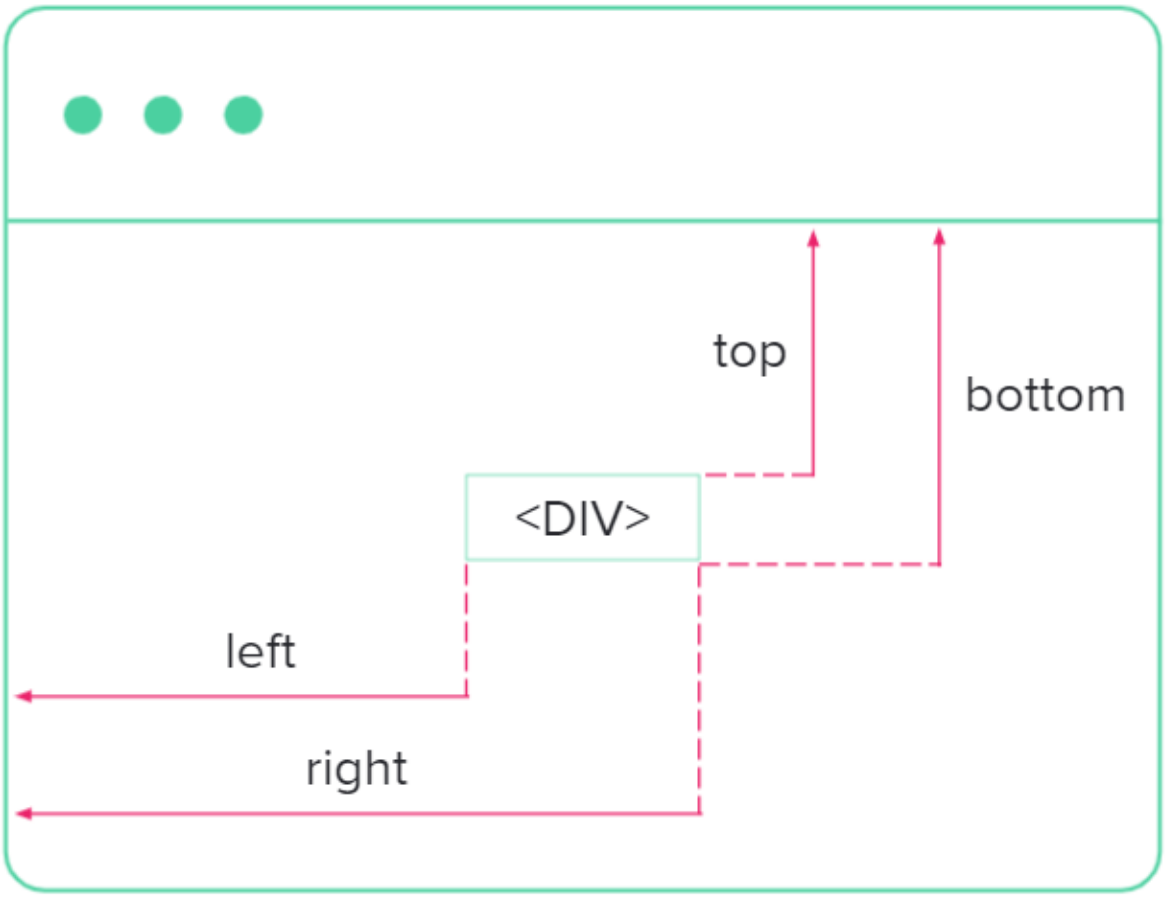
- Свойство element.style позволяет работать с inline-стилями любого HTML-элемента через одноимённые свойства, но в camelCase нотации.
- Метод getComputedStyle(element[, ':pseudo']) позволяет получить вычисленные браузером актуальные значения стилей, в том числе не заданных явно через CSS.

4. Определение размеров и позиций элементов

Для определения размеров и положения элемента на странице существует метод **getBoundingClientRect**(). С точки зрения этого метода элемент ограничен воображаемым прямоугольником, и мы можем получить его координаты.

Метод **element.getBoundingClientRect**() возвращает координаты элемента 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 пропадёт за верхней частью окна браузера, значения top и bottom будут отрицательными, но также будут рассчитаны.

Пример: необходимо обнаружить, появился ли элемент внутри видимой области браузера, используя дополнительное свойство **window.innerHeight**, которое всегда возвращает актуальную высоту видимой области внутри окна браузера:

```
<div></div>
<script>
  var isInViewport = function(element){
    const viewportHeight = window.innerHeight;
    const elementTop =
      element.getBoundingClientRect().top;

    return elementTop < viewportHeight ? true : false;
  };
  const div = document.querySelector('div');
  isInViewport(div);
</script>
```

Код, приведённый 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. Для того чтобы решить эту проблему, нам необходимо отслеживать не только свойство **top**, но и нижние координаты элемента.

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

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.

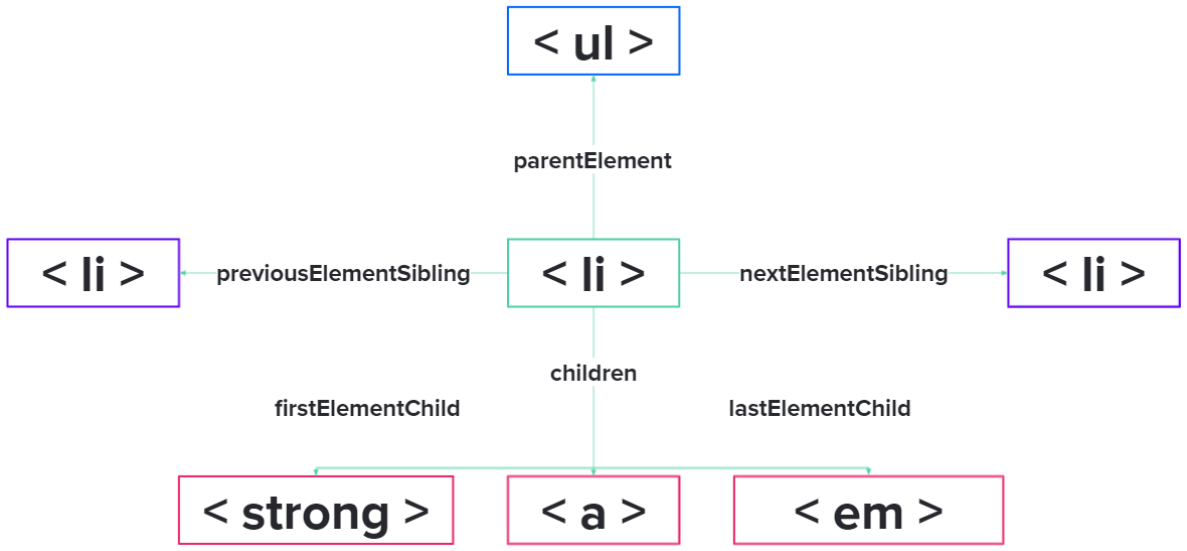
Получаем больше информации об элементе

Иногда нам необходимо узнать некоторую информацию о конкретном элементе. Например, когда внутри функции нам нужно совершить разные операции над абзацем или div, 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: **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**.

5. Навигация по элементам

В **DOM**-дереве у любого элемента могут существовать элементы, находящиеся выше и ниже по дереву, а также соседние элементы. Очевидно, что в **DOM-**дереве 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, указывающие на родителя, детей и соседей элемента:

- parentElement — родитель-элемент;
- previousElementSibling — предыдущий соседний элемент;
- nextElementSibling — следующий соседний элемент;
- firstElementChild — первый дочерний элемент;
- lastElementChild — 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:

```
<section class="modal">
```

```
  <p>
    Это модельное окно, их несколько на странице,
    поэтому использовать поиск родителя по классу нельзя
  </p>
  <button>Закрыть</button>
</section>

<script>
  const closeModal = function() {
    const modalParent = this.parentElement;
    modalParent.classList.add('hidden');
  };

  const modalButtons = document.querySelectorAll('.modal button');
  for ( const button of modalButtons ) {
    button.addEventListener('click', closeModal);
  }
</script>
```

Совершенно небольшое изменение приведет к полной неработоспособности скрипта:

```
<section class="modal">
  <p>
    Это модельное окно, их несколько на странице,
    поэтому использовать поиск родителя по классу нельзя
  </p>
  <div>
    <button>Закрыть</button>
  </div>
</section>
<script>
  const closeModal = function() {
    const modalParent = this.parentElement;
    modalParent.classList.add('hidden');
    // Класс будет добавлен элементу `div`
  };
  const modalButtons = document.querySelectorAll('.modal button');
  for ( const button of modalButtons ) {
    button.addEventListener('click', closeModal);
  }
</script>
```

Мы можем переписать наш скрипт, заменив **.parentElement** на **.parentElement.parentElement**, получив второго по старшинству родителя, но возникнет затруднение при увеличении их количества до 4 или 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:

```
<section class="modal">
```

```
  <p>
    Это модельное окно, их несколько на странице,
    поэтому использовать поиск родителя по классу нельзя
  </p>
  <div>
    ...
    <div>
      <button>Закрыть</button>
    </div>
    ...
  </div>
</section>

<script>
  const closeModal = function(){
    const modalParent = this.closest('.modal');
    modalParent.classList.add('hidden');
    // Класс всегда будет добавлен родителю `.modal`
  };

  const modalButtons = document.querySelectorAll('.modal button');

  for ( const button of modalButtons ){
    button.addEventListener('click', closeModal);
  }
</script>
```

Также зачастую в JS нет необходимости обращаться к следующему или к предыдущему элементу из-за возможности изменения структуры. Поэтому вместо **previous\nextElementSibling** часто используют удобную связку **closest** и **querySelector**:

```
<section class="modal">
  <p>
    Этот текст будет заменен
  </p>
  <div>
    ...
    <div>
      <button>Закрыть</button>
    </div>
    ...
  </div>
</section>

<script>
  const changeText = function () {
    const modalParent = this.closest('.modal');
    const modalParagraph = modalParent.querySelector('p');
    modalParagraph.textContent = "Вы нажали на кнопку!";
```

```
  };

  const modalButtons = document.querySelectorAll('.modal button');

  for (const button of modalButtons) {
    button.addEventListener('click', changeText);
  }
</script>
```

Есть также ещё методы, позволяющие сэкономить время:

- **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** вернёт 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.

Итоги

- Для навигации по DOM-элементам можно использовать: parentElement, nextElementSibling, previousElementSibling, firstElementChild, lastElementChild, children.
- 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).

Итоги по теме

- 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:

- Навигация по DOM-элементам
- Element.classList
- Использование data-* атрибутов
- window.getComputedStyle()