Занятие 34

Библиотеки

Мы умеем создавать программы на странице с помощью тега <script>. Вспомним, что JavaScript-программа должна располагаться на странице после HTML-разметки, с которой будет работать программа. Также, возможно вынести весь JavaScript-код, содержащийся в теге <script> в отдельный файл, а в самом теге указать путь к этому файлу.

Рассмотрим пример из первого занятия, при этом перенесем JavaScript-код в отдельный файл:

**Файл index.html:**

<!DOCTYPE HTML>

<html>

<head>

    <title>Hello, world!</title>

</head>

<body>

<script type="text/javascript" src="hello.js"></script>

</body>

</html>

**Файл hello.js:**

alert('Hello, world!');

Как видите, мы создали файл hello.js с расширением .js - по названию языка. Такие файлы будут распознаваться вашими редакторами и IDE как файлы, содержащие JavaScript и подсвечивать синтаксис соответственным образом. Также, заметьте что файлы .js не содержат никакой дополнительной разметки - только чистый JavaScript.

В месте, где мы подключаем этот файл, мы указываем путь к файлу в атрибуте src тега <script>. Заметьте, что мы все равно обязаны закрыть тег, хотя в нем ничего нет. На этом моменте браузер загружает указанный файл и выполняет его, поведение аналогично тому, что если бы текст внутри этого файла был бы вставлен в страницу в тег <script>.

Как правило, такое вынесение кода в файл используют для создания так называемых **библиотек** - сборников функций, которые затем подключаются на нужных вам страницах сайта, чтобы избежать **дублирования** кода на страницах, либо для вынесения основной программы в отдельный файл.

Рассмотрим создание такой библиотеки на примере функции, которая будет создавать элемент для дальнейшей работы:

**Файл utils.js:**

const createOrderDiv = (name, amount) => {

  const div = document.createElement('div');

  div.innerHTML = name + ': <b>x ' + amount + '</b>';

  div.style.color = 'purple';

  return div;

};

**Файл index.html:**

<!DOCTYPE HTML>

<html>

<head>

  <title>Hello, world!</title>

  <script type="text/javascript" src="utils.js"></script>

</head>

<body>

<div id="orders"></div>

<script>

const container = document.getElementById('orders');

const items = [

  {name: 'Cheeseburger', amount: 2},

  {name: 'Fries', amount: 1},

  {name: 'Cola', amount: 2}

];

for (let i = 0; i < items.length; i++) {

  const item = items[i];

  const element = createOrderDiv(item.name, item.amount);

  container.appendChild(element);

}

</script>

</body>

</html>

В рассмотренном примере мы создали отдельный файл utils.js, в который положили нашу функцию, затем в основном файле index.html подключили этот файл в теге <head>, после чего написали второй скрипт в конце файла, в котором использовали функцию.

Так часто и делают - подключают библиотеки в начале файла, в <head>, в них как правило содержатся функции, которые никак не зависят от разметки на этом этапе, а код, относящийся к конкретной странице, пишется в теге <script> прямо на этой странице. Либо, все библиотеки располагаются в конце страницы, но основная программа в отдельном файле все равно должна быть подключена после всех остальных, чтобы иметь возможность обращаться к переменным, функциям, объектам, и т.п. из подключенных библиотек.

Есть еще несколько вариантов организации кода, о которых мы поговорим в дальнейших занятиях.

Еще про поиск элементов

Мы уже выяснили, что существует современный способ поиска элементов по CSS-селектору: document.querySelector и document.querySelectorAll.

Однако, во всех ли случаях нужно их использовать? Мы будем стараться совмещать их с более простыми существующими методами, чтобы получить оптимальный результат.

Будем придерживаться такого правила:

* document.getElementById - если элемент имеет определенный id. Мы знаем, что элемент на странице может быть только один с таким id.
* document.querySelector - для поиска элемента по CSS-селектору, когда мы хотим найти **один** элемент.
* document.querySelectorAll - для поиска элементов по CSS-селектору, когда хотим найти **много** элементов сразу.

Рассмотрим некоторые примеры и вспомним некоторые CSS-селекторы:

**// найти один элемент с id="my-div"**

document.getElementById('my-div');

**// выбрать все элементы с class="some-class"**

document.querySelectorAll('.some-class');

**// выбрать все элементы input, с type="text" и class="to-validate"**

document.querySelectorAll('input[type="text"].to-validate');

**// выбрать все элементы span, находящиеся внутри div только на первом уровне вложенности (direct children)**

document.querySelectorAll('div > span');

Если поиск элемента не дал результата (такого элемента на странице не содержится), то методы getElementById и querySelector вернут вам null. Метод querySelectorAll вернет пустой NodeList (можно воспринимать, как пустой массив). Поэтому, проверить, что элемент не был найден, можно, например так:

// для одного элемента

const element = document.querySelector('.asdf');

// или

const element = document.getElementById('.asdf');

if (element !== null) {

**// элемент найден**

}

// для нескольких элементов

const elements = document.querySelectorAll('.asdf');

if (elements.length > 0) {

**// хотя бы один элемент найден**

}

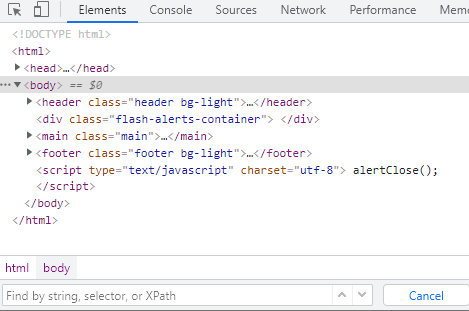
Если вы будете пытаться вызывать методы или обращаться к свойствам у элементов, которые не были найдены, вы соответственно увидите ошибку такого вида:

https://lh5.googleusercontent.com/A9XMlVdh04YMi6u4aiS9D33PKV4gmxpoDIzqOtrsV4IHDlY3f15XNLwaQT2jciwEZmo8Npl-VDlrAVlG_BrgxwW6XnEmSA20fLilBIG6K0-zAdbOOAYLnl6JNmSxGnVXeluIc5FF0LD2wEzUWIHbmdI3FPYHUilI

Т.к. нельзя обращаться к методам и свойствам у null, что логично.

Если вы будете искать несколько элементов, то скорее всего вам надо будет использовать цикл, для того, чтобы пройти по всем этим элементам. В этом случае, ошибки не будет, но тело цикла не отработает ни разу.

Для проверки, работает ли ваш CSS-селектор, вы также можете воспользоваться вкладкой "Elements" в Chrome Developer Tools, находясь в этой вкладке нажмите кнопку Ctrl+F для поиска элементов, внизу появится такое поле для поиска:



В него можно ввести CSS-селектор и найденные элементы будут подсвечены. Если элементы не найдены, вы увидите 0 из 0 элементов:

https://lh6.googleusercontent.com/Bkh1b07k_BTw8o5sgIwX0Ydg97KiKwQgzrhjFKO50R-BN8uut8sM_Q7Lhmzjf-skIjdCPWxYxxbxVONUBcAi4rq15efJNR_9SDBlphxZqvwTKJxlZS4AkkueigvLfgyCm40QfspCzOgZ34C4b15_PnFaB96axC3b

Внутри элементов также можно производить поиск, если вам надо найти какие-то дочерние элементы внутри уже найденного элемента. Для этого также можно воспользоваться методами .querySelector() и .querySelectorAll(), только вызывать их уже у родительского элемента, а не у объекта document.

<div id="container">

  <div class="element">Element 1</div>

  <div class="element">Element 2</div>

  <div class="element">Element 3</div>

  <div class="divider"><hr/></div>

</div>

<div class="element">Outside element</div>

<script>

const container = document.getElementById('container');

const elements = container.querySelectorAll('.element');

console.log(elements.length); // 3 внутренних элемента

</script>

Такой поиск можно использовать, когда мы не будем знать точно, в каком элементе мы производим поиск, иначе лучше использовать сложные селекторы:

const elements = document.querySelectorAll('#container .element');

В этом случае мы получим тот же результат. Но, если нам еще нужен будет отдельно найденный контейнер, для других дальнейших действий, то будет даже более предпочтителен первый вариант, с поиском внутри элемента.

В целом старайтесь делать как можно меньше поиска элементов на страницах, если можно этого избежать.

Дерево элементов и навигация по элементам

Вы можете находить элементы не только с помощью методов поиска по CSS-селекторам, id или классам. Так как объектная модель документа содержит все элементы в дереве, можно находить элементы с помощью навигации по этому дереву элементов.

Все элементы начинаются с элемента document - это специальный родительский элемент для верхнего уровня элементов.

Предположим такую верстку:

<html>

<head></head>

<body>

  <div class="container">

    <ul>

      <li>One</li>

      <li>Two</li>

      <li>Three</li>

    </ul>

  </div>

</body>

</html>

Чтобы получить доступ к элементу body можно обратиться к нему, как к:

document.body

Затем, у body существует специальное свойство - childNodes, которое представляет собой коллекцию элементов. Мы уже знаем, что можем обращаться к коллекциям как к массивам, поэтому, в нашем случае,

document.body.children[0] даст нам элемент <div class="container">.

Также "шорткат" для этой операции - document.body.firstElementChild

Затем, мы можем получить точно также элемент <ul>, который находится внутри этого элемента, например. Сохраним его как отдельную переменную:

<script>

const ul = document.body.firstElementChild.firstElementChild;

</script>

Теперь, мы можем посмотреть на список дочерних элементов UL:

ul.children.length // 3

Длина списка дочерних элементов - 3. Соответственно, взять второй элемент мы можем как:

const li = ul.children[1] // <li>Two</li>

Также у элементов мы можем идти не только "вниз" по дереву, но также и "вверх", например найти его родителя. Для этого у элементов есть свойство parentElement.

li.parentElement; // <ul>

А также "вбок", находя соседние элементы. Для этого есть свойства nextSibling и previousSibling:

li.nextElementSibling // <li>Three</li>

li.previousElementSibling // <li>One</li>

Это нужно, когда вы например, нашли элемент с помощью методов поиска, а затем хотите взять родительский или соседний элемент для какой-то манипуляции.

Коллекции элементов

Давайте разберемся немного подробнее, что мы можем, а что не можем делать с коллекциями. Мы уже знаем, что мы можем обращаться к ним как к массивам, и использовать циклы for, или даже for..of для обработки.

Не рекомендуется для обработки использовать цикл for..in.

Коллекции элементов нельзя изменить, как массивы, то есть, допустим,

ul.children[1] = document.createElement('li'); // так делать нельзя

Все добавление/изменение элементов должно происходить с помощью методов, которые мы рассмотрели в прошлом занятии, то есть, appendChild, replaceChild, insertBefore, remove и т.п.

Однако, коллекции являются "живыми", то есть в них отражаются изменения, если мы изменим что-то с помощью методов манипуляции деревом элементов.

Например,

const newLi = document.createElement('li');

newLi.innerHTML = 'Zero';

ul.insertBefore(newLi, ul.firstChild);

console.log(ul.children.length) // 4

Коллекция дочерних элементов автоматически обновилась, т.к. добавился новый элемент.

У коллекций элементов нет тех же методов, что и у массивов. То есть, например, методы forEach, map, reduce, filter и т.п. - отсутствуют. Поэтому, для "перебора" коллекций всегда используем циклы for или for..of.

Поэтому, резюмируя, коллекции элементов ведут себя как массивы, но массивами не являются.

Атрибуты элементов

Если необходимо получить доступ к атрибутам элементов, то мы можем это сделать, с помощью методов getAttribute и setAttribute у элементов.

Например:

<div id="container" style="width: 100px;"></div>

<script>

const container = document.getElementById('container');

console.log(container.getAttribute('id'));    // "container"

console.log(container.getAttribute('style')); // "width: 100px;"

container.setAttribute('id', 'not-container');

</script>

Вызывая метод getAttribute, вы должны передать в него название атрибута, который вы хотите получить, и метод вернет его значение.

Вызывая метод setAttribute, вы должны передать в него два аргумента - название атрибута и его значение, при этом существующий атрибут перезапишется на новое значение, а если его раньше не было, то он добавится.

Это можно использовать, например, для изменения классов:

container.setAttribute('class', 'my-container');

Однако, для классов есть специальное свойство classList, которое позволяет манипулировать классами гораздо проще:

container.classList.add('my-container'); // добавить класс

container.classList.add('active', 'other'); // добавить сколько угодно классов, передавая сколько угодно аргументов.

container.classList.remove('other') // удалить класс

container.classList.toggle('active') // добавить класс, если его раньше не было, или удалить, если он раньше был.

container.classList.replace('my-container', 'cnt') // заменить один класс на другой

Также, если надо назначить элементу заранее известный класс (например, при его создании), то можно воспользоваться свойством className:

const newItem = document.createElement('div');

newItem.className = 'element item active'; // сразу 3 класса

container.append(newItem);

Немного про jQuery. Vanilla JS

Мы сейчас с вами изучаем стандартные способы манипуляции DOM, без каких-либо библиотек. Этот способ называется в коммьюнити - Vanilla JS (ванильный JS). Однако, для удобства в свое время были придуманы дополнительные библиотеки, которые облегчают манипуляцию элементами, т.к. многое в Vanilla JS произошло исторически и не все является достаточно удобным.

Поэтому, мы можем попробовать библиотеку jQuery, однако в будущем пользоваться ей не будем. Причина - в том, что в современных библиотеках построения интерфейсов для frontend, таких как Angular, VueJS, React и т.п. свои собственные способы манипуляции элементами, и jQuery будет только лишним. Мы изучаем и будем продолжать изучать способы Vanilla JS, т.к. при работе с такими библиотеками все-равно иногда приходится делать что-то "вручную", и для этого вы не сможете использовать jQuery как правило. А если даже и сможете, это будет считаться дурным тоном использовать лишнюю библиотеку для того, что можно сделать и без неё.

Поэтому, давайте посмотрим на то, как скачать/подключить jQuery, а также как сделать в ней некоторые основные действия, которые мы уже прошли в "Ванильном JS"

JS файл библиотеки можно найти на странице <http://jquery.com/download/>

Подключить библиотеку к странице можно несколькими способами, в частности:

* Скачать библиотеку и использовать как локальный JS-файл
* Использовать библиотеку с сервера "раздачи контента" (CDN)

Скачаем development-версию (разработческую) библиотеки. Она отличается от production-версии (производственной) тем, что содержит всю библиотеку в "соединенном" виде, то есть представляет из себя огромной длины JS-файл со всем кодом, и при желании мы сможем его посмотреть, а также воспользоваться дебаггером для того, чтобы понимать, что происходит.

В production-же версии файл "минифицирован", что означает, что все инструкции были склеены в одну строку, автоматическим образом были изменены названия локальных переменных и функций, и поэтому "залезть" внутрь и просмотреть его будет сложнее. Плюс production-версии в меньшем размере, что позволяет немного сэкономить количество отдаваемых с сервера данных и таким образом ускорять загрузку страницы.

После того как мы скачали файл, надо его подключить на страницу следующим образом:

...

<script src="jquery-X.X.X.js"></script>

...

Где X.X.X - версия, которую мы скачиваем с сайта jQuery. На момент написания данного материала, версия на сайте - 3.5.1. Для удобства мы будем обращаться к данному файлу как просто "jquery.js". Мы можем также переименовать данный файл, чтобы было удобнее его подключать.

Подразумевается, что файл jquery.js лежит в одной директории что и файл index.html. Как мы уже знаем, js-файлы библиотек подключаются как правило внутрь тега <head>. Теперь jQuery подключен на страницу.

Также можно подключить jQuery из CDN - для этого выберем на сайте в разделе "Using jQuery with a CDN" ссылку на сайт - <http://code.jquery.com>. Затем нажмем на "Uncompressed". Получим готовый кусочек кода для вставки на страницу.

После того, как jQuery был подключен, нам становится доступна функция "доллар", с помощью которой мы будем осуществлять все то, что нам предоставляет jQuery. Одной из основных возможностей - это поиск элементов. Посмотрим, как делать уже известные нам вещи с помощью jQuery:

<div id="sample-div">Sample</div>

<script>

  const sample = $('#sample-div');

  sample.html('Change sample text');

</script>

Для поиска элементов на странице мы вызываем jQuery-функцию $() (функцию-доллар), и передаем в нее строку, представляющую из себя CSS селектор. То есть, на первый взгляд, эта функция ведет себя примерно как рассмотренная ранее функция document.querySelectorAll. Однако, как мы посмотрим чуть далее, эта функция также позволяет принимать множество различных аргументов и вести себя по-разному в зависимости от аргументов. Посмотрим на второй пример, поиск по классу:

<div class="element">Element</div>

<div class="element">Element</div>

<div class="element">Element</div>

<script>

  const elements = $('.element');

  for(let i = 0; i < elements.length; i++) {

    const element = $(elements[i]);

    element.html(element.html() + (i + 1));

  }

</script>

Обратите внимание, что получив элементы, мы можем "пройтись" по ним, как по обычному массиву.  При этом обратите внимание, что каждый элемент этого "массива" - это обычный HTML-элемент, то есть для того, чтобы использовать jquery-функции у этих элементов, нам приходится еще раз передавать их в функцию-доллар:

const element = $(elements[i]);

Функция-доллар возвращает коллекцию jQuery с одним этим самым элементом внутри. Теперь мы можем у этой коллекции вызывать jQuery-специфичные методы (такие как .html()).

Обратите внимание, что вызывая методы у jquery-коллекций, которые как мы поняли, могут содержать 0, 1 и более элементов, эти методы будут применяться ко всем элементам этой коллекции без исключения. Например:

<div class="element">Element 1</div>

<div class="element">Element 2</div>

<div class="element">Element 3</div>

<script>

  const elements = $('.element');

  elements.html('New text'); // поменяет текст у всех трех элементов

</script>

Если коллекция не содержит элементов, то метод все равно вызовется и при этом не вызовет никакой ошибки!

const elements = $('.not-existing-class');

elements.html('Some text'); // просто отработает, ошибки не вызовет

Для того, чтобы создать новый элемент, необходимо передать строку с HTML этого элемента во всю ту же функцию jQuery:

const newDiv = $('<div class="item">Hamburger <b>x 1</b></div>');

Как видите, мы можем передавать готовый HTML, и тогда функция jQuery вернет элемент. Если HTML будет неверным, функция может вернуть неверный результат, либо сообщить об ошибке.

Мы можем также передавать туда "незакрытые теги" для более понятного кода, в том случае если нам нужно всего лишь создать элемент:

const anotherDiv = $('<div>');

Заметьте, что если бы мы передали туда строку 'div' без значков тега, это бы воспринималось как CSS-селектор, то есть "найди все элементы div".

Чтобы вставить элемент на страницу, мы будем использовать метод .append, который есть у всех jQuery-элементов:

$('#container').append(newDiv);

Метод .append работает также, как и метод .appendChild - он добавляет элемент "в конец" - после последнего дочернего элемента родительского элемента-контейнера.

Также существует метод .prepend, который добавляет элемент "в начало" - перед первым дочерним элементом:

$('#container').prepend(newDiv);

Также существуют аналог метода insertBefore, только у него иной интерфейс: он вызывается на элементе, который надо вставить, а параметр - это элемент или селектор, перед которым надо вставить этот элемент:

<p id="first">First paragraph</p>

<p id="second">Second paragraph</p>

<p id="message" style="color: red;">Not like others</p>

<script>

$('#message').insertBefore('#second');

</script>

Удаление элементов производится через похожий метод .remove()

$('#message').remove();

Атрибуты элементов получаются и изменяются с помощью метода .attr()

Стили элементов получаются и изменяются с помощью метода .css()

Добавление и изменение классов с помощью методов .addClass(), .removeClass(), .toggleClass().

Документация: <https://api.jquery.com/>