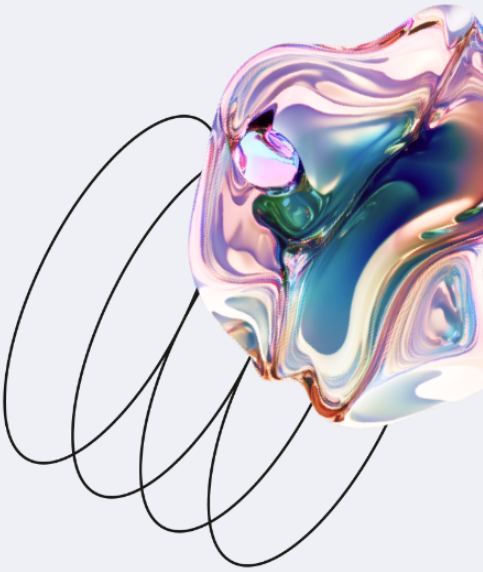
GeekBrains



DOM-дерево

JavaScript про API браузеров

Оглавление

Введение

На этом уроке мы поработаем непосредственно с DOM — деревом. Научимся привязывать JavaScript-код непосредственно к элементам дерева и работать с их содержимым.

Глобальный объект браузера

Ранее мы рассматривали, что такое глобальный объект. Тема наших уроков — браузерное окружение, а глобальный объект браузера — это window. И тема наших следующих уроков — рассмотреть его возможности подробнее.

Итак, рассмотрим структуру глобального объекта window. Состоит он из трёх основных блоков:

1. DOM
2. BOM
3. JavaScript

Собственно, JavaScript мы изучаем уже давно, про него не будем говорить подробнее. А остальные два пункта пора обсудить детально.

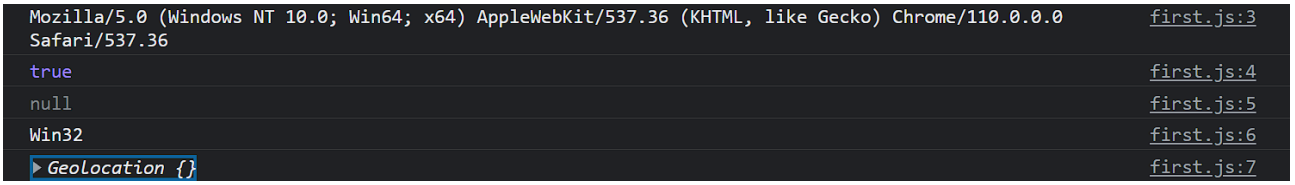
BOM

BOM (browser object model) — это свойства и методы глобального объекта, которые мы, по большей части, используем для работы со всем, кроме HTML-кода. Например:

- Хорошо известные нам функции alert/confirm/prompt. Они являются средствами коммуникации с пользователем, но не имеют отношения к HTML-коду.
- Объект navigator. Здесь нам выведется информация о браузере, операционной системе и их настройках, которые могут быть важны при выполнении нашей программы. Например:

```
1 console.log(navigator.userAgent);
2 console.log(navigator.cookieEnabled);
3 console.log(navigator.doNotTrack);
4 console.log(navigator.platform);
5 console.log(navigator.geolocation);
```

Выдаст:



Здесь:

- userAgent — информация о браузере;
- cookieEnabled — включены ли coockie;
- doNotTrack — включена ли опция запрета на отслеживание;
- platform — текущая ОС пользователя;
- geolocation — геолокация, в данном случае не активированная.

Есть ещё много возможностей, ознакомиться с которыми можно в спецификации HTML, частью которой является BOM.

- Объект location даёт информацию о текущем URL и позволяет перейти по новому пути. Например:

```
1 console.log(location);
2 location.href = './first.html';
```

Здесь мы вывели в консоль браузера сам объект location, чтобы посмотреть все его свойства и методы. Например, свойство href нам выдаст адрес текущей страницы, в него также можно записать новый адрес для перехода по нему. Здесь мы записали адрес той же страницы, что у нас и была. Как думаете, к чему привёл подобный шаг?

Вот вывод в консоль:

```
Location {ancestorOrigins: DOMStringList, href: 'file:///C:/repo/learn-js/first.html', origin: 'file://', protocol: 'file:', host: '', …}
  ancestorOrigins: DOMStringList {length: 0}
  assign: ƒ assign()
  hash: ""
  host: ""
  hostname: ""
  href: "file:///C:/repo/learn-js/first.html"
  origin: "file://"
  pathname: "/C:/repo/learn-js/first.html"
  port: ""
  protocol: "file:"
  reload: ƒ reload()
  replace: ƒ replace()
  search: ""
  toString: ƒ toString()
  valueOf: ƒ valueOf()
  Symbol(Symbol.toPrimitive): undefined
  [[Prototype]]: Location
⚠ Throttling navigation to prevent the browser from hanging. See https://crbug.com/1038223. Command line switch --disable-ipc-flooding-protection can be used to bypass the protection
```

Здесь мы увидели объект location с описанием опций положения нашего запускаемого файла со скриптом. А потом... предупреждение, что переходы по написанной нам ссылке прерваны. Почему прерваны? Потому что произошло зацикливание. Во второй строчке мы вызываем переход на эту же страницу, которая начинает выполняться заново. Снова происходит переход. Потом ещё. И так бесконечно, поэтому браузер решает прервать эту операцию во избежание зависания. И нам сообщается, что мы это можем отменить, но тогда браузер зависнет точно.

DOM

DOM (document object model(— по сути, привязанный шаблон или HTML-документ, из которого был вызван этот скрипт. Он также является частью объекта window, и его можно менять под свои нужды. Этот объект называется document. Поменяем фоновый цвет страницы на фиолетовый:

```
1 document.body.style.backgroundColor = 'purple';
```

На самом деле, возможности изменения объекта document намного шире. Все они перечислены в спецификации. И в этом уроке мы разберём некоторые из них.

Но document может быть вызван не только в скрипте как часть window. Его ещё можно вызвать в таких случаях:

- В объекте iframe через свойство contentDocument.
- В качестве ответа responseXML объекта XMLHttpRequest.
- Из любого элемента или узла через свойство ownerDocument.

В зависимости от вида документа (т. е. HTML или XML) у объекта document могут быть доступны разные API.

Все объекты документов реализуют интерфейс Document. Таким образом, основные свойства и методы, описанные выше, доступны для всех видов документов. В современных браузерах некоторые документы (содержащие контент text/html) также реализуют интерфейс HTMLDocument, а SVG-документы реализуют интерфейс SVGDocument. В будущем все эти интерфейсы будут сведены в один — Document.

Document

В нашем редакторе VS Code есть функция генерации базового HTML-документа. Для этого нужно в пустом документе написать восклицательный знак и выбрать предложенное автозаполнение. Вот что нам сгенерируется:

```
1 <!DOCTYPE html>
```

```
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Document</title>
</head>
<body>

</body>
</html>
```

Расставим правильно отступы и добавим в тело вызов скрипта, с которым мы работали всё это время:

```
<!DOCTYPE html>
<html lang="en">
    <head>
        <meta charset="UTF-8">
        <meta http-equiv="X-UA-Compatible" content="IE=edge">
        <meta name="viewport" content="width=device-width, initial-scale=1.0">
        <title>Learn JS</title>
    </head>
    <body>
        <script src="first.js"></script>
    </body>
</html>
```

Здесь мы поставили отступы по узловым тегам. Эти теги образуют структуру DOM-дерева. В редакторе, при желании, их можно открывать и закрывать:

first.html > html

```
<!DOCTYPE html>
<html lang="en">
    <head>…
    </head>
    <body>
        <script src="first.js"></script>
    </body>
</html>
```

Внутри тегов образуются текстовые узлы. Запишем внутрь тега body текст для вывода на экран:

```
<body>
    <script src="first.js"></script>
    <span>Изучаем JavaScript</span>
</body>
```

Внутри тега span образовался текстовый узел. Он может включать в себя пробелы, знаки перевода строки и любые другие текстовые символы. У текстовых узлов не может быть потомков.

Есть пара нюансов из спецификации HTML:

- Если мы записываем что-то после тега body, то это содержимое переместится в его конец.
- Перед тегом head пробельные символы и перевод строки игнорируются.

В остальных случаях пробельные символы и переводы строк становятся текстовыми узлами документа.

Выведем в консоль наш документ:

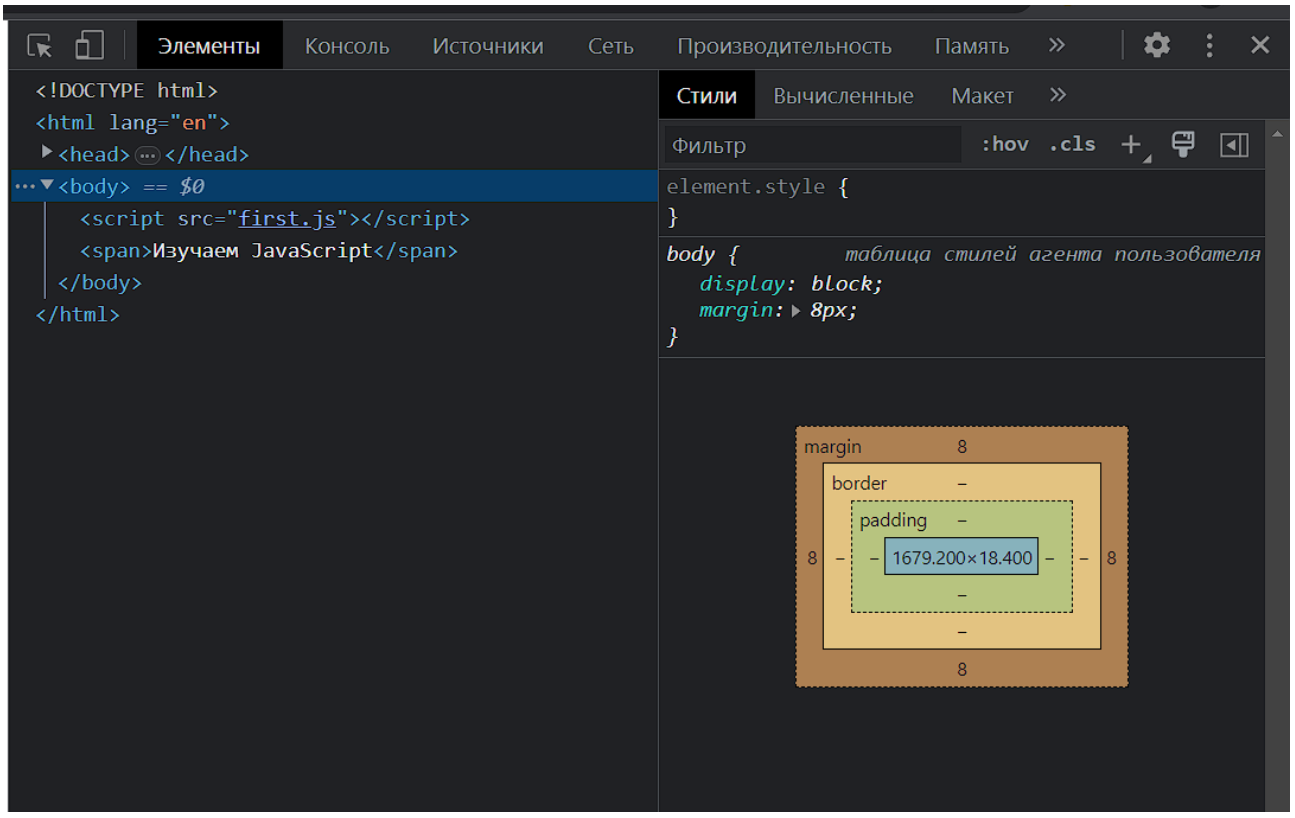
```
'use strict';

console.log(document);
```

```
▼#document                                                    first.js:3
    <!DOCTYPE html>
    <html lang="en">
    ▶<head>…</head>
    ▶<body>…</body>
    </html>
>
```

Мы прямо в консоли увидим HTML-код, к которому привязан наш скрипт. А это значит, что мы можем его полностью или частично использовать в нашей программе. **Именно этому и будет посвящён наш курс.**

Естественно, постоянно выводить в консоль содержание документа не очень удобно, поэтому в средствах для разработчика есть специальная вкладка, в которой мы можем увидеть разметку страницы. Это первая вкладка и она называется «Элементы» (“Elements”):



В ней мы можем не только рассмотреть разметку, но и увидеть применённые к ней стили. И, при желании, попробовать их подредактировать прямо в браузере.

Но вернёмся к узлам нашего документа, ведь мы рассмотрели ещё не все. Их всего 12 типов:

```
[Exposed=Window]
interface Node : EventTarget {
  const unsigned short ELEMENT_NODE = 1;
  const unsigned short ATTRIBUTE_NODE = 2;
  const unsigned short TEXT_NODE = 3;
  const unsigned short CDATA_SECTION_NODE = 4;
  const unsigned short ENTITY_REFERENCE_NODE = 5; // legacy
  const unsigned short ENTITY_NODE = 6; // legacy
  const unsigned short PROCESSING_INSTRUCTION_NODE = 7;
  const unsigned short COMMENT_NODE = 8;
  const unsigned short DOCUMENT_NODE = 9;
  const unsigned short DOCUMENT_TYPE_NODE = 10;
  const unsigned short DOCUMENT_FRAGMENT_NODE = 11;
  const unsigned short NOTATION_NODE = 12; // legacy
```

Скриншот из [спецификации](#).

Как мы видим, три уже признаны устаревшими. Ещё два мы рассмотрели. Нам интересен ещё один — узел комментария (под номером 8). Напишем комментарий:

```
<!DOCTYPE html>
<html lang="en">
    <head>
        <meta charset="UTF-8">
        <meta http-equiv="X-UA-Compatible" content="IE=edge">
        <meta name="viewport" content="width=device-width, initial-scale=1.0">
        <title>Learn JS</title>
    </head>
    <body>
        <!-- It`s our first document! -->
        <script src="first.js"></script>
        <span>Изучаем JavaScript</span>
    </body>
</html>
```

Он тоже станет частью нашего DOM-дерева. Ведь всё, что написано в HTML-файле, туда обязательно попадёт.

На практике нам нужны только рассмотренные нами узлы. Сам объект document тоже является узлом — он в спецификации под номером 9. Ещё мы поработаем с атрибутами тегов. Например, для подключения нашего скрипта к HTML-коду необходимо в атрибуте src тега script указать путь к нашему файлу.

Если мы напишем некорректный HTML-код, браузер, по возможности, постарается его исправить при построении DOM-дерева. Как уже отмечалось ранее, текст после тега body перенесётся внутрь его. Но исправляется не только это.

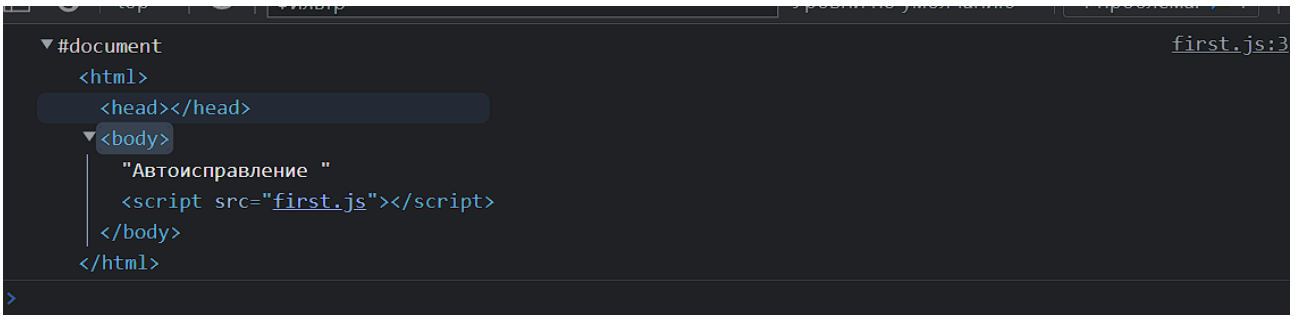
Например, если написать просто слово в HTML-документе, то недостающие теги будут поставлены. Сделаем такой HTML-документ:

```
1 Автоисправление
2
3 <script src="first.js"></script>
```

Выведем в скрипте содержимое документа:

```
1 'use strict';
2
3 console.log(document);
```

Вот что у нас получится в качестве DOM-дерева:



Мы видим, что вся необходимая разметка была добавлена браузером автоматически.

🔥 Есть особый случай автоисправления — это таблицы. По стандарту DOM внутри таблиц должен быть тег tbody, но по стандарту HTML он необязателен. При его отсутствии, браузер его поставит автоматически.

Навигация по элементам

Как мы уже отмечали выше, все действия с HTML-разметкой браузера мы можем произвести с помощью объекта document, который, в свою очередь, является частью глобального объекта window. К трём корневым тегам HTML-документа мы можем обратиться напрямую как к свойствам объекта document:

```
1 document.documentElement; // тег <html>
2 document.body; // тег <body>
3 document.head; // тег <head>
```

Если тега не существует, то подобное обращение нам вернёт null. Например, если мы вызовем скрипт из head с обращением к body, то нам вернётся null, так как body ещё не сформирован:

```
<html>
    <head>
        <script>
            console.log(document.body);
        </script>
    </head>
    <body>
        <script>
            console.log(document.body);
        </script>
    </body>
</html>
```

Консоль:

Первый вывод тела нам выдал null, поскольку в head оно ещё не сформировано: документ формируется последовательно. Второй вывод нам выдаёт его код, всё работает правильно.

Теперь поговорим о навигации внутри тегов. Дочерними узлами назовём, лежащие непосредственно на один уровень вложенности внутри тега, а потомками все, без учёта уровня вложенности.

Навигация по дочерним элементам

Свойство firstChild даёт доступ к первому дочернему элементу. Свойство lastChild даст доступ к последнему дочернему элементу. Коллекция childNodes включает все дочерние узлы, включая текстовые. Коллекция children даст доступ к списку дочерних элементов-тегов. Посмотрим их работу на практике:

```html
<html>
    <body>
        <div>First element?</div>
        <!-- Comment -->
        Text element
        <br />
        <span>
            <span>grandchild element</span>
        </span>
        <script src="first.js"></script>
        <div>Last element?</div>
    </body>
</html>
```

Рассмотрим такую разметку и, для начала, выведем в консоль:

```js
console.log(document.body.firstChild);
console.log(document.body.lastChild);
console.log(document.body.childNodes);
console.log(document.body.children);
```

```
▼ #text i                                                    first.js:4
    assignedSlot: null
    baseURI: "file:///X:/last_repo/vsk/learn-js/first.html"
  ▶ childNodes: NodeList []
    data: "\n        "
    firstChild: null
    isConnected: true
    lastChild: null
    length: 9
  ▶ nextElementSibling: div
  ▶ nextSibling: div
    nodeName: "#text"
    nodeType: 3
    nodeValue: "\n        "
```

Проанализируем результаты. Первым элементом, который нам был выдан свойством firstChild и коллекцией childNodes, стал… текстовый элемент. Откуда же он взялся? Всё дело в том, что переводы строк и пробельные символы тоже являются текстовыми узлами. А у нас в документе переводы строк сделаны для удобства просмотра. И они записались как текстовые узлы. И отступы в тегах вместе с ними. Поэтому внутри тегов чаще всего первыми символами будут текстовые узлы, которые там имеются, хотя мы их не видим. Кроме этого, отметим, что в качестве результата нам выданы объекты, которые тоже, в свою очередь,

могут иметь те же свойства — firstChild, lastChild и остальные. И мы ими аналогичным образом можем пользоваться.

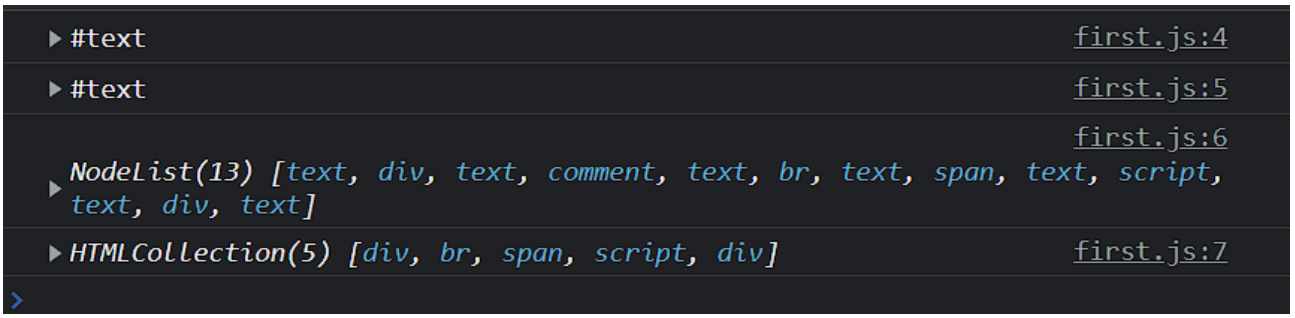
Теперь проанализируем, что же нам выдаётся в качестве последнего элемента. В разметке мы сделали Last element? . Но он, судя по изложенному выше, не должен становиться последним, так как переводы строки тоже присутствуют. Но, в итоге, у нас последним элементом становится... скрипт, из которого мы вызываем вывод в консоль. Почему же так происходит? Всё потому, что скрипты выполняются синхронно, и у нас вывод в консоль сработает раньше, чем сгенерируется конец документа. То же самое выдаёт предпросмотр коллекции childNodes, но в развёрнутом виде нам выдаётся уже полный документ, так как когда мы разворачиваем его, он успевает полностью сгенерироваться. Отсюда можно понять, что представленные свойства, хоть и являются доступными только для чтения, являются «живыми»: их содержание меняется оперативно, на лету.

Но что делать, чтобы скрипты не задерживали нам генерацию документа? Ведь некоторые узлы могут не сформироваться на момент выполнения скрипта, что ожидаемо приведёт к ошибкам!

- 💡 В стандарте HTML5 присутствуют атрибуты у тега script для подключения скрипта после формирования разметки. Это атрибуты async и defer.
- 💡 **Async** указывает браузеру загрузить скрипт, по возможности, асинхронно. Если в документе несколько скриптов, они могут быть загружены в произвольном порядке, по мере загрузки, это может привести к нежелательным последствиям.
- 💡 **Defer** загрузит скрипты синхронно после генерации разметки, но до события DOMContentLoaded (событие генерации DOM-дерева). Это сохранит порядок подключения скриптов.

  Эти атрибуты работают только при указании пути подключаемых скриптов в атрибуте src. Динамически подключаемые скрипты по умолчанию имеют атрибут async. При использовании модулей атрибут defer подключается по умолчанию.

Итак, подключим наш скрипт с атрибутом defer и посмотрим, что получится:



Последним элементом стал текстовый узел с такими данными:

```
1 data :: "\n \n\n\n"
```

Предпросмотр коллекций теперь у нас соответствует текущему содержимому, в котором содержатся все представленные узлы.

Работа со свойствами и коллекциями HTML

Рассмотрим ещё свойства элементов и попробуем с ними поработать. Обратиться к первому и последнему элементу мы можем с помощью firstElementChild и lastElementChild. К предыдущему элементу обращаемся через previousElementSibling, к следующему — через nextElementSibling. К «родителю» мы можем обратиться через parentElement, который, однако, у свойства document.documentElement будет равняться null.

Коллекции элементов — это объекты типа HTMLCollection, которые являются псевдомассивами. Мы помним, что псевдомассивы — это объекты, которые похожи на массивы, но ими не являются, поэтому мы не можем с ними работать как с массивами полностью. Но у нас есть метод Array.from(), который легко может такой объект переделать в массив.

Выше мы перечислили все основные свойства разнообразных узлов, которые могут быть как и текстовыми, так и тегами, либо любыми другими узлами документа. Все они, как выше упоминалось, доступны только для чтения.

Для перебора коллекций у нас есть цикл for(..of..). Он, в отличие от цикла for(..in..), тоже будет работать, но он не будет перебирать так называемые «лишние» свойства: length, item, value и т. п.

```
for(let val of document.body.children){
    console.log(val);
}
```

```
<div>First element</div>                                    first.js:5
<br>                                                        first.js:5
▶ <span> … </span>                                          first.js:5
<script defer src="first.js"></script>                      first.js:5
<div>Last element</div>                                     first.js:5
>
```

Сделаем в переборе коллекции вывод проверки, является ли он div:

```
for(let val of document.body.children){
    console.log(val.localName === 'div' ? "Это DIV": "Это не DIV");
}
```

```
Это DIV                                                     first.js:5
3 Это не DIV                                                first.js:5
Это DIV                                                     first.js:5
>
```

Здесь мы использовали свойство, присущее только элементу, в котором хранится его название.

Посмотрим соседей, потомков и родителя каждого элемента:

```
for(let val of document.body.children){
    console.log('————————next element————————');
    console.log(val.firstElementChild);
    console.log(val.lastElementChild);
    console.log(val.previousElementSibling);
    console.log(val.nextElementSibling);
```

```
7     console.log(val.parentElement);
8 }
```

```
-------next element----------                    first.js:5
null                                             first.js:6
null                                             first.js:7
null                                             first.js:8
  <br>                                           first.js:9
▶ <body> … </body>                               first.js:10
-------next element----------                    first.js:5
null                                             first.js:6
null                                             first.js:7
  <div>First element</div>                       first.js:8
▶ <span> … </span>                               first.js:9
▶ <body> … </body>                               first.js:10
-------next element----------                    first.js:5
  <span>grandchild element</span>                first.js:6
  <span>grandchild element</span>                first.js:7
  <br>                                           first.js:8
  <script defer src="first.js"></script>         first.js:9
▶ <body> … </body>                               first.js:10
-------next element----------                    first.js:5
null                                             first.js:6
null                                             first.js:7
▶ <span> … </span>                               first.js:8
  <div>Last element</div>                        first.js:9
▶ <body> … </body>                               first.js:10
-------next element----------                    first.js:5
null                                             first.js:6
null                                             first.js:7
  <script defer src="first.js"></script>         first.js:8
null                                             first.js:9
▶ <body> … </body>                               first.js:10
```

Свойства узлов

Выше вы уже видели, что все узлы делятся по типам, у каждого есть свои возможности для работы. А это означает, что для каждого типа узлов имеется свой встроенный класс. А у этого класса имеются свои свойства и методы, которые мы будем рассматривать далее.

> 💡 Мы видели, что метод console.log(), который мы использовали ранее, при выдаче узла выдаёт его HTML-представление. Чтобы мы могли видеть его в представленных классах, можно воспользоваться методом console.dir().

В TypeScript и в других языках есть понятие абстрактного класса. Класс называется абстрактным, если его экземпляры не создаются, а нужен он для наследования логики в классах-потомках. При этом зачастую он имеет абстрактные свойства и методы, которые в абстрактных классах не реализованы (имеют префикс abstract), но должны будут реализованы в классах-потомках.

Абстрактный класс EventTarget служит прототипом для всего нижеперечисленного. Его наследник Node тоже является абстрактным и является основой для узлов DOM. Остальные классы являются его наследниками. Их три: Document, Element и CharacterData. Document является прототипом для уже известного нам класса HTMLDocument, этот тип имеет объект document. Element — базовый класс для HTML-элементов. В нём содержатся методы для поиска элементов, которые мы рассмотрим в следующей главе. Он также является прототипом для классов HTMLElement, SVGElement и XMLElement. CharacterData является абстрактным классом для классов Text и Comment, которые являются текстовыми узлами и узлами комментариев соответственно.

HTMLElement является базовым классом для всех элементов, с ним мы и будем работать больше всего. У каждого элемента имеется свой подкласс, который содержит все специфичные свойства и методы для него.

nodeType

Свойство nodeType является частью перечисления (enum) с типом узла. Посмотрим в цикле тип каждого узла:

First.html, эту разметку мы использовали в предыдущих примерах:

```html
<html>
    <body>
        <div>First element</div>
        <!-- Comment  -->
        Text element
        <br />
        <span>
            <span>grandchild element</span>
        </span>
        <script defer src="first.js"></script>
        <div>Last element</div>
    </body>
</html>
```

First.js

```js
for(let val of document.body.childNodes){
    console.dir(val.nodeType);
}
```

Консоль:

```
3                                                   first.js:5
1                                                   first.js:5
3                                                   first.js:5
8                                                   first.js:5
3                                                   first.js:5
1                                                   first.js:5
3                                                   first.js:5
1                                                   first.js:5
3                                                   first.js:5
1                                                   first.js:5
3                                                   first.js:5
1                                                   first.js:5
3                                                   first.js:5
```

У нас вывелись три значения: 1 — означает узел элемента, 3 — текстовый узел, 8 — узел комментария. Все значения можно посмотреть на странице спецификации.

Название тега

Для получения названия тега есть два свойства: nodeName из класса Node и tagName класса Element. Нам будет выдано название в верхнем регистре. Если узел не является элементом, tagName нам вернёт undefined.

nodeValue и data

Эти два свойства дают нам доступ к содержимому текстового узла. Они практически идентичны, поэтому можно использовать любой. Выведем нашу разметку циклом:

```
1 for(let val of document.body.childNodes){
2     console.dir(val.nodeValue);
3 }
```

Смотрим в консоль. Здесь переводы строки уже выведены не символами, а применены «как есть в тексте».

```
                                                    first.js:5

null                                                first.js:5
                                                    first.js:5

 Comment                                            first.js:5
                                                    first.js:5
        Text element

null                                                first.js:5
                                                    first.js:5

null                                                first.js:5
                                                    first.js:5

null                                                first.js:5
                                                    first.js:5
```

```
null                                                    first.js:5
                                                        first.js:5
```

outerHTML

Свойство outerHTML даёт доступ к содержимому HTML-кода элемента целиком. Содержимое можно изменить, но не рекомендуется, так как сам элемент не изменится, но зато заменится во внешнем контексте. Это может привести к ошибкам.

hidden и style

Свойство hidden является булевым значением. Его изменять можно, если мы хотим скрыть или показать элемент. Сделаем имитацию светомузыки с его помощью:

```
<html>
    <body>
        <span id="light">Светомузыка!</span>
        <script>
            setInterval(()⇒ {
                light.hidden = ( parseInt(Math.random()*2) === 1);
                document.body.style.backgroundColor =
                `rgb(${parseInt(Math.random()*255)},
                ${parseInt(Math.random()*255)},
                ${parseInt(Math.random()*255)})`;
            }, 200);
        </script>
    </body>
</html>
```

Здесь ещё мы изменили метод style для body, задав с помощью классов css значение фона элемента, сделав цвет полностью случайным.

classList и className

Свойство className помогает обращаться к имени класса элемента и изменить его. Но его изменение заменяет содержимое атрибута class полностью. Свойство classList работает с одним классом. У него есть методы для добавления или удаления класса (add и remove), проверки наличия класса (contains) и метод toggle, который удалит его при наличии класса, а при отсутствии добавит.

innerHTML и textContent

Эти два свойства дают возможность изменения элемента. Если textContent изменяет содержимое текстового узла, то innerHTML позволяет изменить полностью разметку всего элемента. Главное отличие между свойствами — это то, что содержимое innerHTML воспринимается как HTML-код (правда, тег script в нём не запустится), то textContent любое содержимое вставит как строку.

Хотя применение textContent безопаснее, innerHTML применяется чаще — всегда удобнее менять вёрстку на лету. Используя innerHTML надо учитывать нюансы — это, в первую очередь, может быть небезопасно. Во вторую очередь, перезапись значения innerHTML (например, оператором +=) приведёт к перезагрузке этого содержимого: заполненные формы сбросятся, выделение пропадёт. Вроде старая разметка осталась, но она была перезаписана.

```
<body>
    <h1>Заголовок</h1>
    <span>span</span>
    <script>
        let inner = document.body.innerHTML;
        let textContent = document.body.textContent;
        console.log(inner);  // 8 строка
        console.log(textContent);  // 9 строка
        document.body.innerHTML += '<h1>Новый BODY!<h1>'; // 10 строка
    </script>
</body>
```

Здесь вывод в консоль на 8 строке выдаст HTML-содержимое странице, а в 10 строке оно перезапишется:

А в консоли будет отличие innerHTML от textContent: в textContent пропали все теги разметки, осталось только текстовое наполнение.

Другие свойства

На самом деле, свойств гораздо больше. И они зависят от класса, к которому относится элемент. Например, в формах часто используется value — введённое в форму значение. Id — это соответствующий атрибут в любом теге, href- в ссылочном теге a и так далее.

В следующей главе мы рассмотрим, как найти элемент в документе, в том числе, и по атрибуту.

Поиск элементов

Поиск по id

В примерах в предыдущей главе мы находили элемент по id как переменную. В примере со светомузыкой у нас обращение к элементу с надписью идёт через id:

```
1 <span id="light">Светомузыка!</span>
```

```
1 light.hidden = (parseInt(Math.random()*2) == 1);
```

Но это не очень хорошо. id из разметки смешиваются с глобальными переменными скрипта и может произойти конфликт имён. Поэтому есть метод получения элемента по id в документе:

```
1 document.getElementById('light').hidden = ( parseInt(Math.random()*2) == 1);
```

Здесь id передаётся строчным параметром в метод getElementById(). Сам getElementById() идёт как метод класса document, так как элемент ищется всегда глобально в документе.

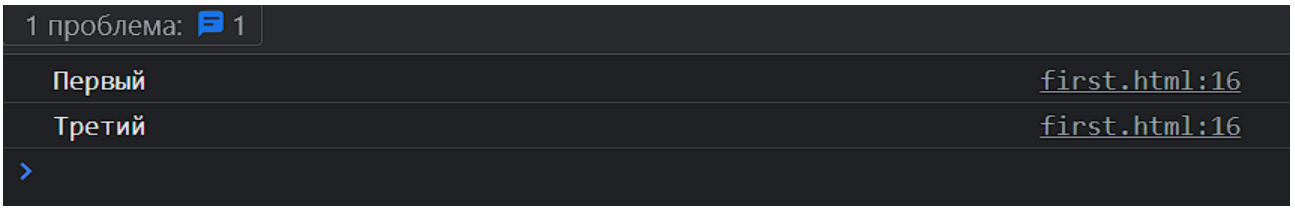
Надо отметить, что id в разметке должны быть уникальными, иначе уже будет конфликт имён в разметке. При этом будет использоваться только первый найденный элемент.

querySelector и querySelectorAll

Эти оба метода возвращают элементы по заданному css-селектору. querySelector вернёт первый подходящий элемент, querySelectorAll вернёт коллекцию элементов, удовлетворяющих поиску.

```
<html>
    <body>
        <table>
            <tr>
                <td>Первый</td>
                <td>Второй</td>
            </tr>
            <tr>
                <td>Третий</td>
                <td>Четвёртый</td>
            </tr>
        </table>
        <script>
            let elements = document.querySelectorAll('tr > td:first-child');
            for(let element of elements){
                console.log(element.textContent);
            }
        </script>
    </body>
</html>
```

Выберутся элементы:



getElementsBy

Существует семейство методов, ищущих элементы по какому-то признаку. Это может быть имя класса, название тега и так далее. Их можно назвать устаревшими,

хотя их использование вполне произойти и в новых скриптах. Их функционал покрывает querySelectorAll.

- element.getElementsByTagName(tag) ищет элементы по заданному тегу и возвращает их коллекцию. Передав звёздочку вместо тега, можно получить всех потомков.
- element.getElementsByClassName(className) возвращает элементы, которые имеют данный CSS-класс.
- document.getElementsByName(name) возвращает элементы с заданным атрибутом name.

Мы видим, что для каждого метода может быть несколько элементов в выдаче, то есть они возвращают коллекцию элементов. И так как элементов несколько, не забываем -s в названии метода (Elements).

Коллекции, как и в предыдущих случаях, являются «живыми», то есть всегда актуальны и изменяются автоматически при необходимости.

closest и matches

Эти два метода принимают в качестве параметра CSS-атрибут и работают от элемента.

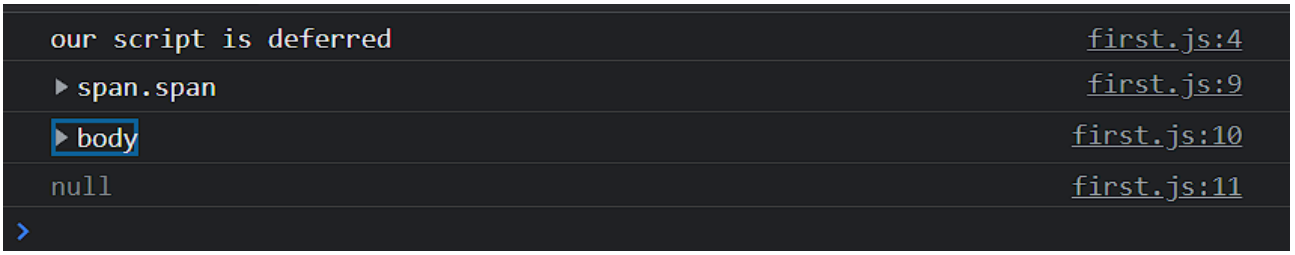
element.matches(css) проверяет, удовлетворяет ли element заданному атрибуту и возвращает true или false.

Метод element.closest(css) проверяет сам элемент и его предков на соответствие CSS-атрибуту. Возвращает первый удовлетворяющий элемент, либо null, если такой элемент не найден.

```
<html>
    <body>
        <span class="span">
            <span class="grandchild">grandchild element</span>
        </span>
        <script defer src="first.js"></script>
        <div class="last">Last element</div>
```

```
    </body>
</html>
```

```
for(let val of document.body.children){
    if(val.matches('script[defer]')) console.log('our script is deferred');
}
let grandChild = document.querySelector('.grandchild');
console.log(grandChild.closest('.span'));
console.log(grandChild.closest('body'));
console.log(grandChild.closest('last'));
```



Последний вывод в консоль у нас не является родителем элемента с классом grandchild, поэтому выдан null. В первом выводе в лог мы нашли тег script с атрибутом defer и вывели это в консоль.

Методы редактирования DOM-дерева

До этого мы изменяли элементы, редактировали их свойства и даже редактировали HTML-код с помощью свойства innerHTML. В этой главе мы рассмотрим работу по добавлению отдельного элемента, его копированию, вставке и удалению.

Создание

Для создания имеются два метода:

- document.createTextNode(‘текст’) — создаёт текстовый узел.
- document.createElement('body') — создаёт элемент.

Узлы создаются и возвращаются этими методами, но пока никуда не вставляются. Для вставки существуют отдельные методы.

Вставка

- node.prepend(...узлы или строки) — вставляет узлы или строки в начало node.
- node.append(...узлы или строки) — добавляет узлы или строки в конец node.
- node.before(...узлы или строки) — вставляет узлы или строки до node.
- node.after(...узлы или строки) — вставляет узлы или строки после node.
- node.replaceWith(...узлы или строки) — заменяет node заданными узлами или строками.

Либо есть универсальный метод, принимающий в параметрах место, куда вставлять элемент: element.insertAdjacentHTML(куда, html). Здесь куда — это место, куда нужно вставить:

- beforebegin — вставить html непосредственно перед element.
- afterbegin — вставить html в начало element.
- beforeend — вставить html в конец element.
- afterend — вставить html непосредственно после element.

У этого метода есть два «брата», которые вставляют текст и элемент:

- element.insertAdjacentText(куда, текст) — вставляет текст.
- element.insertAdjacentElement(куда, Элемент) — вставляет элемент Элемент.

Удаление

Удаление производит метод узла node.remove(). Отметим, что при перемещении метода на старом месте метод удалится автоматически.

Клонирование

Для клонирования элементов существует метод element.cloneNode(глубоко). Параметр «глубоко» определяет, насколько глубоко будет клонирован элемент: при установке true клонируется элемент со всеми вложенными элементами и атрибутами. При false не будет вложенных элементов.

Разберём пример

```
<html>
    <body>
        <script defer src="first.js"></script>
    </body>
</html>
```

Разметку мы оставили только с вызовом скрипта:

```
let div = document.createElement('div');
div.innerHTML = "<strong>Всем привет!</strong> Я - новенький элемент!";
document.body.append(div);
let div2 = div.cloneNode(true);
div2.innerHTML += " Второй";
setTimeout(() => div.insertAdjacentElement('afterend', div2), 1000);
setTimeout(() => div.remove(), 2000);
```

Мы сначала создали элемент. Потом добавили текстовое содержимое и вставили после body. Второй элемент был склонирован из первого, добавили ему пометку, что он второй. И далее с помощью двух таймаутов «оживляем» документ — через секунду вставляем второй элемент, а ещё через секунду — удаляем первый.

Заключение

В этом уроке мы разобрали подробно объект document и способы работы с его помощью с DOM-деревом. Разобрали, что такое узлы и элементы, получили к ним доступ, поуправляли их содержимым и научились редактировать их положение в DOM-дереве.

Далее нас ждёт знакомство с другими средствами API браузера — разберём события и формы и научимся работать с сетевыми запросами.