

# JavaScript в браузере

## начало

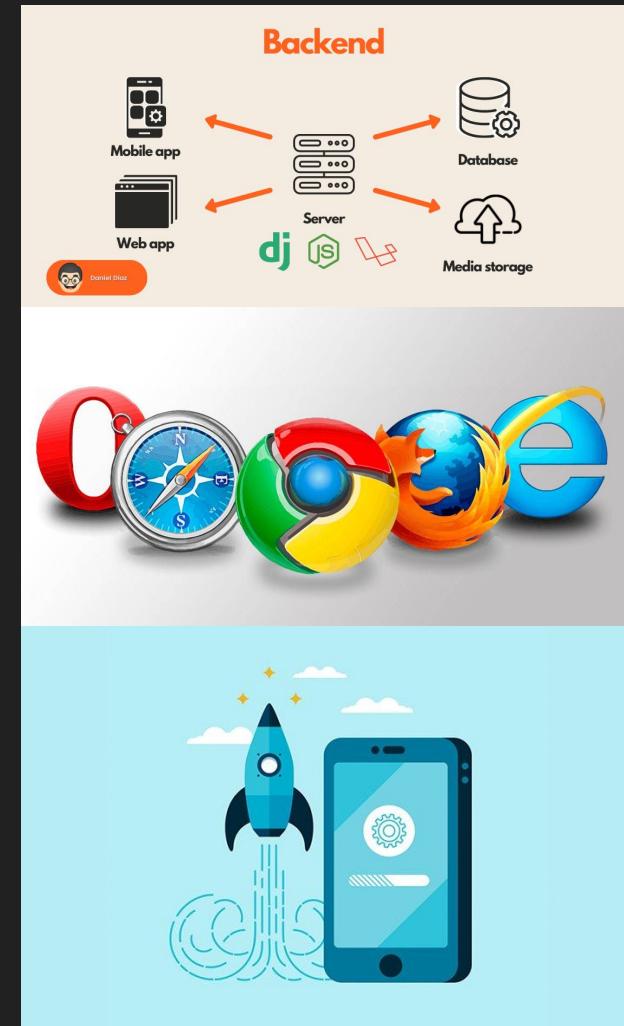
DOM, BOM и как подключать JS в HTML

# Как используется JavaScript

Язык JavaScript изначально был создан для веб-браузеров. Но с тех пор он значительно эволюционировал и превратился в кроссплатформенный язык программирования для решения широкого круга задач.

Сегодня JavaScript может использоваться в браузере, на веб-сервере или в какой-то другой среде, даже в кофеварке. Каждая среда предоставляет свою функциональность, которую спецификация JavaScript называет окружением.

Окружение предоставляет свои объекты и дополнительные функции, в дополнение базовым языковым. Браузеры, например, дают средства для управления веб-страницами. Node.js делает доступными какие-то серверные возможности и так далее.



# Движки и среды исполнения JavaScript кода

Движки и среды выполнения JavaScript связаны, но тем не менее они разные. Движок JavaScript реализовывает ECMAScript согласно стандарту ECMA-262. Данный стандарт определяет основную функциональность JavaScript без какой-либо реализации ввода и вывода. Среда выполнения JavaScript - это хост ECMAScript, который встраивает движок JavaScript и дополняет его различной функциональностью для ввода и вывода наряду с тем, что еще нужно для самой среды выполнения. Дополнительная функциональность может включать в себя DOM в веб-браузере или доступ к файловой системе в серверных средах выполнения. Сами среды выполнения не имеют каких-либо обязательств для следования другим стандартам и вправе определять свое собственное необходимое им API, именно поэтому Node.js, Deno и Bun имеют разное API для работы с файловой системы.

# Где можно использовать JavaScript

Web разработка - разработка front-end части:

- Нативный (ванильный) JS + HTML + CSS
- Jquery + HTML + CSS
- Фреймворки (React, Vue, Angular)

Серверная разработка - разработка back-end части:

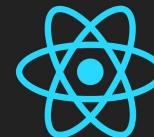
- Node.js (не очень оптимизировано но супер популярный)
- Bun.js (только реализовалась в сентябре 2023)
- Deno.js (круто модно но мало библиотек)
- Фреймворки (nest.js, next.js, nuxt.js, express.js + prisma)

Мобильная разработка:

- React native

Десктопная разработка:

- Electron.js



React Native



# JavaScript в Web разработке

Среда исполнения JavaScript-а - браузер

Каждая вкладка или iframe создают свой поток исполнения JavaScript кода

Также есть возможность запуска второго потока исполнения в одной вкладке (реализуется с помощью web-worker-а)

Более того существует service-worker - выполняет роль прокси-сервера, находящегося между веб-приложением и браузером, а также сетью (если доступна).

# Web Workers API

Web Workers это механизм, который позволяет скрипту выполнять в фоновом потоке, который отделен от основного потока веб-приложения. Преимущество заключается в том, что ресурсоемкие вычисления могут выполняться в отдельном потоке, позволяя запустить основной (обычно пользовательский) поток без блокировки и замедления.

Это настоящая асинхронность реализуемая многопоточностью

# Service worker

Service worker — это событийно-управляемый worker, регистрируемый на уровне источника и пути. Он представляет собой JavaScript-файл, который может контролировать веб-страницу/сайт, с которым он ассоциируется, перехватывать и модифицировать запросы навигации и ресурсов, очень гибко кешировать ресурсы, для того чтобы предоставить вам полный контроль над тем, как приложение ведет себя в определённых ситуациях (например, когда сеть не доступна).

Своего рода middleware (прокси) который позволяет работать приложению в оффлайн режиме (не совмещен с основным потоком исполнения)

# JavaScript в HTML или в файле JS

## Встроенные скрипты

Вы можете вставить JavaScript-код непосредственно в HTML-документ, используя элемент `<script>` внутри `<head>` или `<body>`

## Внешние скрипты

Часто лучше размещать JavaScript-код в отдельном файле и подключать его к HTML-документу. Это улучшает читаемость кода и его повторное использование.

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>Document</title>
  <script>
    // Ваш JavaScript-код здесь
    console.log('Hello, world!');
  </script>
</head>
<body>
  <h1>Hello, World!</h1>
</body>
</html>
```

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>Document</title>
</head>
<body>
  <h1>Hello, World!</h1>
  <script src="script.js"></script>
</body>
</html>
```

# Подключение JavaScript в HTML файл

Скрипты можно размещать в различных частях HTML-документа:

## Внутри <head>

Если вы хотите, чтобы ваш JavaScript-код выполнялся сразу после загрузки HTML-кода, разместите элемент `<script>` внутри `<head>`.

Однако это может замедлить загрузку страницы, так как браузер должен сначала загрузить и выполнить скрипт, прежде чем отобразить содержимое страницы.

## Внутри <body>

Лучше всего размещать элемент `<script>` перед закрывающим тегом `</body>`, чтобы HTML-код страницы загружался и отображался до выполнения JavaScript-кода. Это улучшает восприятие скорости загрузки страницы.

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>Document</title>
  <script src="script.js"></script>
</head>
<body>
  <h1>Hello, World!</h1>
</body>
</html>
```

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>Document</title>
</head>
<body>
  <h1>Hello, World!</h1>
  <script src="script.js"></script>
</body>
</html>
```

# Атрибуты <script>

Вы также можете использовать атрибуты defer и async для управления выполнением скриптов:

## defer

Атрибут defer откладывает выполнение скрипта до тех пор, пока весь HTML-документ не будет полностью загружен и разобран. Скрипты с defer выполняются в том порядке, в котором они указаны в документе.

## async

Атрибут async указывает, что скрипт должен быть выполнен асинхронно, как только он будет загружен. Скрипты с async могут выполняться в любом порядке, что может привести к непредсказуемому поведению, если у вас есть зависимости между скриптами.

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>Document</title>
  <script src="script.js" defer></script>
</head>
<body>
  <h1>Hello, World!</h1>
</body>
</html>
```

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>Document</title>
  <script src="script.js" async></script>
</head>
<body>
  <h1>Hello, World!</h1>
</body>
</html>
```

## На практике

Наиболее часто используемый и рекомендуемый способ — это размещение элемента `<script>` перед закрывающим тегом `</body>`, чтобы минимизировать задержки при загрузке и отображении страницы.

Атрибутами `async` и `defer` в основном пользуются для оптимизации под капотом фреймворков

По факту фреймворки по типу React, Vue, Angular сами предоставляют шаблон подключений при компиляции и сборке проекта

В чем и есть один из их плюсов - разработчик тратит время на реализацию функционала нежели тратит время на оптимизацию кода

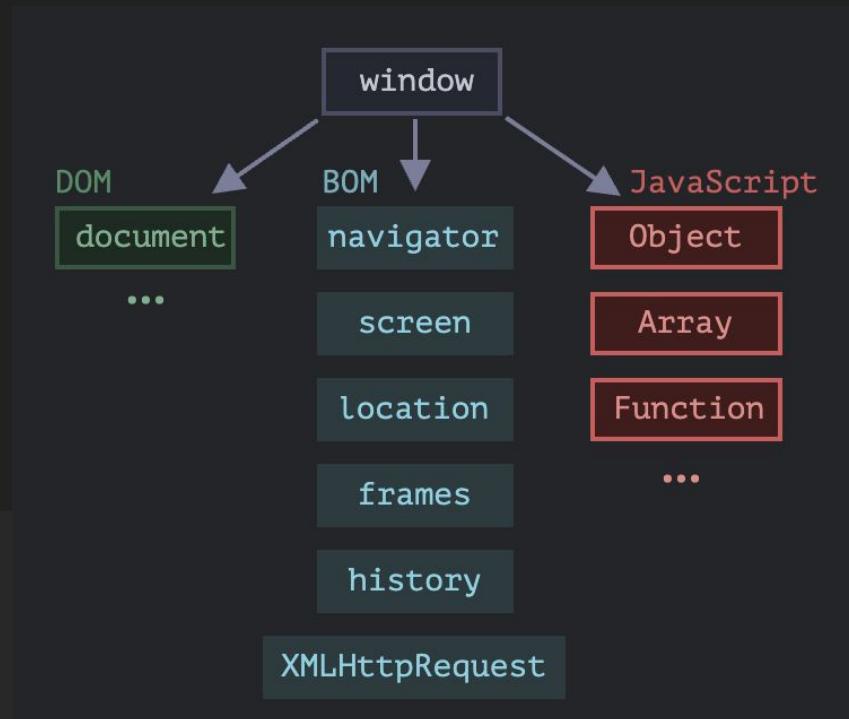
# Браузерное окружение, спецификации - WEB API

Во-первых, это глобальный объект для JavaScript-кода

Во-вторых, он также представляет собой окно браузера и располагает методами для управления им.

Отображение this в браузере - обратиться к контексту глобального объекта (window)

```
index.js:1
▼ Window ⓘ
▶ alert: f alert()
▶ atob: f atob()
▶ blur: f blur()
▶ btoa: f btoa()
▶ caches: CacheStorage {}
▶ cancelAnimationFrame: f cancelAnimationFrame()
▶ cancelIdleCallback: f cancelIdleCallback()
▶ captureEvents: f captureEvents()
```



# Web APIs

API расшифровывается как Application Programming Interface (интерфейс прикладного программирования). Это набор правил и инструментов, позволяющих различным программным приложениям взаимодействовать друг с другом. API определяет методы и данные, которые одно приложение может использовать для работы с другим, обеспечивая стандартизованный способ коммуникации между ними.

(Web Application Programming Interfaces) представляют собой интерфейсы, предоставляемые браузерами и другими веб-платформами для выполнения различных задач, таких как манипулирование документами, выполнение сетевых запросов, хранение данных и многое другое. Они позволяют разработчикам создавать сложные и интерактивные веб-приложения.

Так же Web API предоставляет функционал асинхронности в JS (event-loop)

# DOM (Document Object Model)

Document Object Model, сокращённо DOM – объектная модель документа, которая представляет все содержимое страницы в виде объектов, которые можно менять.

Объект `document` – основная «входная точка». С его помощью мы можем что-то создавать или менять на странице.

Спецификация DOM описывает структуру документа и предоставляет объекты для манипуляций со страницей. Существуют и другие, отличные от браузеров, инструменты, использующие DOM.

Например, серверные скрипты, которые загружают и обрабатывают HTML-страницы, также могут использовать DOM. При этом они могут поддерживать спецификацию не полностью.

JS index.js

1 `console.log(document);`

```
▼ #document (file:///Users/iskander/Projects/index.html)
  <!DOCTYPE html>
  <html lang="en">
    ▼ <head>
      <meta charset="UTF-8">
      <meta name="viewport" content="width=device-width, initial-scale=1.0">
      <title>Document</title>
    </head>
    ▼ <body>
      <script src="./index.js"></script>
    </body>
  </html>
```

# BOM (Browser Object Model)

Объектная модель браузера (Browser Object Model, BOM) – это дополнительные объекты, предоставляемые браузером (окружением), чтобы работать со всем, кроме документа.

## Объект window

window - основной объект BOM, представляющий окно браузера. Все остальные объекты BOM являются его свойствами.

Например:

- Объект navigator дает информацию о самом браузере и операционной системе. Среди множества его свойств самыми известными являются: navigator.userAgent – информация о текущем браузере, и navigator.platform – информация о платформе (может помочь в понимании того, в какой ОС открыт браузер – Windows/Linux/Mac и так далее).
- Объект location позволяет получить текущий URL и перенаправить браузер по новому адресу.
- Функции alert/confirm/prompt тоже являются частью BOM: они не относятся непосредственно к странице, но представляют собой методы объекта окна браузера для коммуникации с пользователем.

# DOM-дерево

Основой HTML-документа являются теги.

В соответствии с объектной моделью документа («Document Object Model», коротко DOM), каждый HTML-тег является объектом. Вложенные теги являются «детёми» родительского элемента. Текст, который находится внутри тега, также является объектом.

Все эти объекты доступны при помощи JavaScript, мы можем использовать их для изменения страницы.

Например, `document.body` – объект для тега `<body>`.

Каждый узел этого дерева – это объект.

Теги являются узлами-элементами (или просто элементами). Они образуют структуру дерева: `<html>` – это корневой узел, `<head>` и `<body>` его дочерние узлы и т.д.

Текст внутри элементов образует текстовые узлы, обозначенные как `#text`. Текстовый узел содержит в себе только строку текста. У него не может быть потомков, т.е. он находится всегда на самом нижнем уровне.

# Автоисправление

Если браузер сталкивается с некорректно написанным HTML-кодом, он автоматически корректирует его при построении DOM.

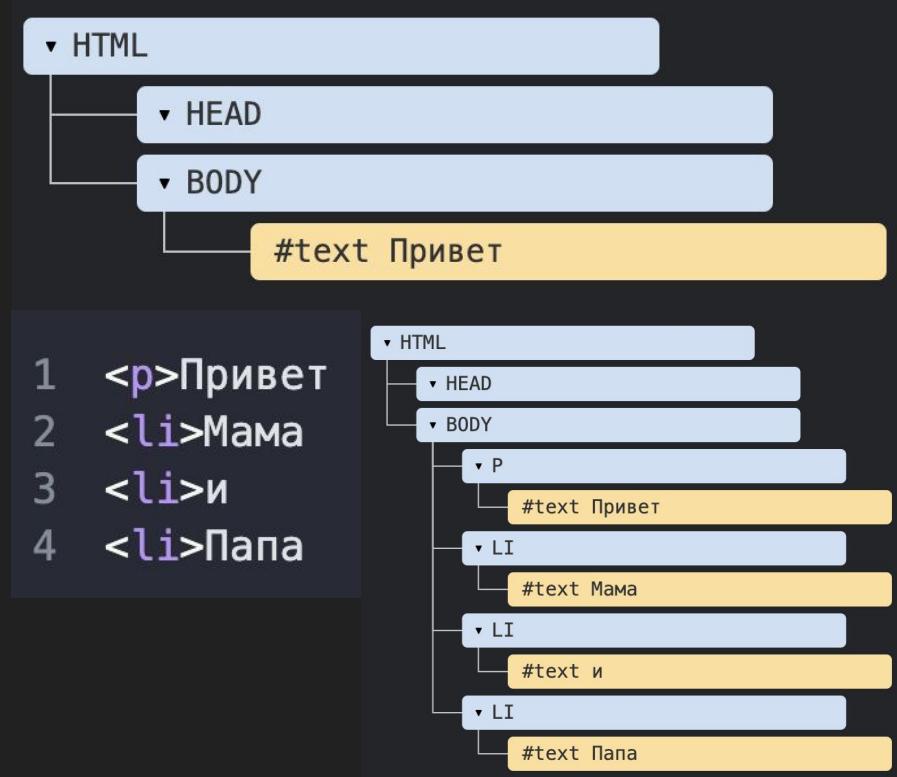
Например, в начале документа всегда должен быть тег `<html>`. Даже если его нет в документе – он будет в дереве DOM, браузер его создаст. То же самое касается и тега `<body>`.

При генерации DOM браузер самостоятельно обрабатывает ошибки в документе, закрывает теги и так далее.

# Примеры автоисправления

Например, если HTML-файл состоит из единственного слова "Привет", браузер обернул его в теги <html> и <body>, добавит необходимый тег <head>, и DOM будет выглядеть так:

DOM будет нормальным, потому что браузер сам закроет теги и восстановит отсутствующие детали:

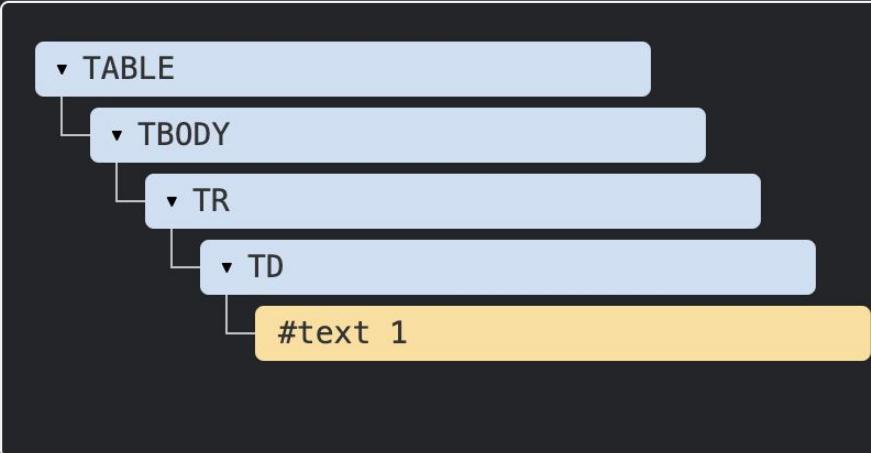


# Таблицы всегда содержат <tbody>

Важный «особый случай» – работа с таблицами. По стандарту DOM у них должен быть <tbody>, но в HTML их можно написать (официально) без него. В этом случае браузер добавляет <tbody> в DOM самостоятельно.

```
1 <table id="table"><tr><td>1</td></tr></table>
```

DOM-структура будет такой:



# Комментарии

Здесь мы видим узел нового типа – комментарий, обозначенный как `#comment`, между двумя текстовыми узлами.

Казалось бы – зачем комментарий в DOM? Он никак не влияет на визуальное отображение. Но есть важное правило: если что-то есть в HTML, то оно должно быть в DOM-дереве.

Все, что есть в HTML, даже комментарии, является частью DOM.

Даже директива `<!DOCTYPE...>`, которую мы ставим в начале HTML, тоже является DOM-узлом. Она находится в дереве DOM прямо перед `<html>`. Мы не будем рассматривать этот узел, мы даже не рисуем его на наших диаграммах, но он существует.

Даже объект `document`, представляющий весь документ, формально является DOM-узлом.

Существует 12 типов узлов. Но на практике мы в основном работаем с 4 из них:

- `document` – «входная точка» в DOM.
- узлы-элементы – HTML-теги, основные строительные блоки.
- текстовые узлы – содержат текст.
- комментарии – иногда в них можно включить информацию, которая не будет показана, но доступна в DOM для чтения JS.

# Навигация по DOM дереву

DOM позволяет нам делать что угодно с элементами и их содержимым, но для на

Все операции с DOM начинаются с объекта `document`. Это главная «точка входа»

Самые верхние элементы дерева доступны как свойства объекта `document`:

`<html>` = `document.documentElement`

Самый верхний узел документа: `document.documentElement`. В DOM он соответст

`<body>` = `document.body`

Другой часто используемый DOM-узел – узел тега `<body>`: `document.body`.

`<head>` = `document.head`

Тег `<head>` доступен как `document.head`.

В DOM значение `null` значит «не существует» или «нет такого узла».



# document.body может быть равен null

Нельзя получить доступ к элементу, которого еще не существует в момент выполнения скрипта.

В частности, если скрипт находится в `<head>`, `document.body` в нем недоступен, потому что браузер его ещё не прочитал.

Поэтому, в примере первый `alert` выведет `null`

```
↳ index.html > ⚒ html
1  <html>
2
3  <head>
4  |  <script>
5  |    alert( "Из HEAD: " + document.body ); // null, <body> ещё нет
6  |  </script>
7  </head>
8
9  <body>
10 |  <script>
11 |    alert( "Из BODY: " + document.body ); // HTMLBodyElement, теперь он есть
12 |  </script>
13
14 </body>
15
16 </html>
```

## Дети: childNodes, firstChild, lastChild

Дочерние узлы (или дети) – элементы, которые являются непосредственными детьми узла. Другими словами, элементы, которые лежат непосредственно внутри данного. Например, `<head>` и `<body>` являются детьми элемента `<html>`.

Потомки – все элементы, которые лежат внутри данного, включая детей, их детей и т.д.

Свойства `firstChild` и `lastChild` обеспечивают быстрый доступ к первому и последнему дочернему элементу.

Для проверки наличия дочерних узлов существует также специальная функция `elem.childNodes()`.

# DOM-коллекции

`childNodes` похож на массив. На самом деле это не массив, а коллекция – особый перебираемый объект-псевдомассив.

И есть два важных следствия из этого:

1. Для перебора коллекции мы можем использовать `for..of`.
2. Методы массивов не будут работать, потому что коллекция – это не массив.

Первый пункт – это хорошо для нас. Второй – бывает неудобен, но можно пережить. Если нам хочется использовать именно методы массива, то мы можем создать настоящий массив из коллекции, используя `Array.from`.

# Работа с DOM коллекциями

## **DOM-коллекции – только для чтения**

DOM-коллекции, и даже более – все навигационные свойства, перечисленные в этой главе, доступны только для чтения.

Мы не можем заменить один дочерний узел на другой, просто написав `childNodes[i] = ...`.

## **DOM-коллекции живые**

Почти все DOM-коллекции, за небольшим исключением, живые. Другими словами, они отражают текущее состояние DOM.

Если мы сохраним ссылку на `elem.childNodes` и добавим/удалим узлы в DOM, то они появятся в сохранённой коллекции автоматически.

## **Не используйте цикл `for..in` для перебора коллекций**

Коллекции перебираются циклом `for..of`. Некоторые начинающие разработчики пытаются использовать для этого цикл `for..in`.

Не делайте так. Цикл `for..in` перебирает все перечисляемые свойства. А у коллекций есть некоторые «лишние», редко используемые свойства, которые обычно нам не нужны:

# Соседи и родитель

Соседи – это узлы, у которых один и тот же родитель.

Следующий узел того же родителя (следующий сосед) – в свойстве `nextSibling`, а предыдущий – в `previousSibling`.

Родитель доступен через `parentNode`.

```
js index.js
1 // родителем <body> является <html>
2 alert( document.body.parentNode === document.documentElement ); // выведет true
3
4 // после <head> идёт <body>
5 alert( document.head.nextSibling ); // HTMLBodyElement
6
7 // перед <body> находится <head>
8 alert( document.body.previousSibling ); // HTMLHeadElement
```

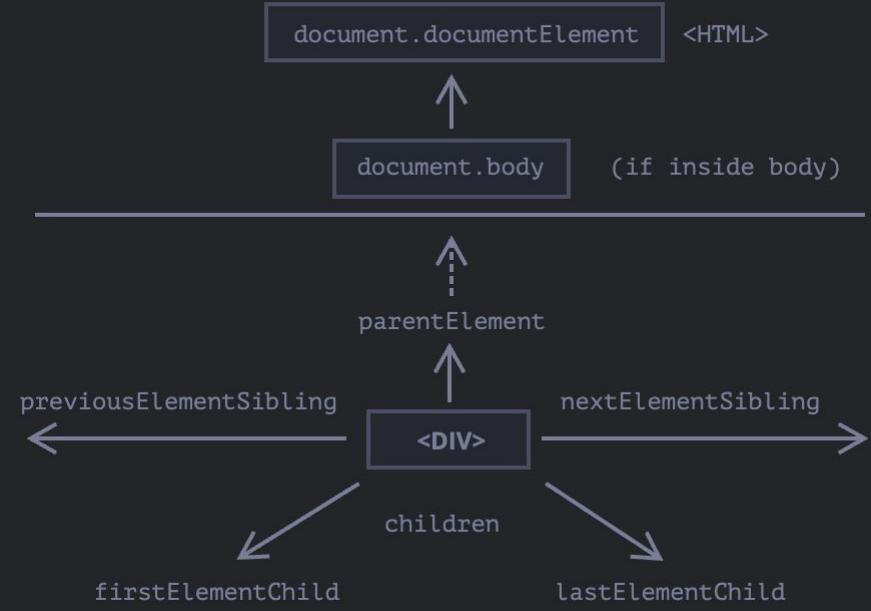
# Навигация только по элементам

Навигационные свойства, описанные выше, относятся ко всем узлам в документе. В частности, в `childNodes` находятся и текстовые узлы и узлы-элементы и узлы-комментарии, если они есть.

Но для большинства задач текстовые узлы и узлы-комментарии нам не нужны. Мы хотим манипулировать узлами-элементами, которые представляют собой теги и формируют структуру страницы.

Эти ссылки похожи на те, что раньше, только в ряде мест стоит слово `Element`:

- `children` – коллекция детей, которые являются элементами.
- `firstElementChild`, `lastElementChild` – первый и последний дочерний элемент.
- `previousElementSibling`, `nextElementSibling` – соседи-элементы.
- `parentElement` – родитель-элемент.



# Зачем нужен parentElement? Разве может родитель быть не элементом?

Свойство `parentElement` возвращает родитель-элемент, а `parentNode` возвращает «любого родителя». Обычно эти свойства одинаковы: они оба получают родителя.

Причина в том, что родителем корневого узла `document.documentElement (<html>)` является `document`. Но `document` – это не узел-элемент, так что `parentNode` вернёт его, а `parentElement` нет.

Эта деталь может быть полезна, если мы хотим пройти вверх по цепочке родителей от произвольного элемента `elem` к `<html>`, но не до `document`:

```
1 alert( document.documentElement.parentNode ); // выведет document
2 alert( document.documentElement.parentElement ); // выведет null

1 while(elem = elem.parentElement) { // идти наверх до <html>
2   alert( elem );
3 }
```

# Ресурсы

Руководство по DOM (MDN документация) - [ТЫК](#)

BOM (объект Window) документация MDN - [ТЫК](#)

Браузерное окружение - [ТЫК](#)

DOM дерево - [ТЫК](#)

Навигация по DOM дереву - [ТЫК](#)