Работа с DOM

Управление стилями

Есть несколько способов управления стилями при использовании JavaScript.

Можно получить список всех таблиц стилей, прикреплённых к документу, через **Document.stylesheets**, который возвращает массив объектов CSSStyleSheet. Стили добавляются или удаляются по желанию. Однако эти функции несколько архаичны и считаются трудным способом манипулирования стилями. Есть более простые способы.

Первый способ — добавить встроенные стили прямо на элементы, которые мы хотим динамически стилизовать. Для этого применяется свойство **HTMLElement.style**. Оно содержит встроенную информацию о стиле для каждого элемента документа. Можно установить свойства этого объекта для прямого обновления стилей элементов.

Рассмотрим пример:

```
const divElement = document.createElement('div')
const paragraphElement = document.createElement('p')
divElement.appendChild(paragraphElement)

paragraphElement.style.color = 'white'
paragraphElement.style.backgroundColor = 'black'
paragraphElement.style.padding = '10px'
paragraphElement.style.width = '250px'
paragraphElement.style.textAlign = 'center'
```

Перезагрузим страницу и увидим, что стили применяются к абзацу. Если посмотреть на этот параграф в инспекторе своего браузера, окажется, что эти строки действительно добавляют встроенные стили в документ:

```
<p style="color: white; background-color: black; padding: 10px; width: 250px;
text-align: center;">...</p>
```

Важно! *Версии свойств JavaScript стилей CSS пишутся в нижнем регистре верблюжьего стиля (lower camel case), в то время как версии свойств стилей CSS используют дефисы кебаб-стиля (kebab case). Примеры:* ***backgroundColor*** *и* ***background-color****. Их нельзя путать!*

Есть ещё один распространённый способ динамического управления стилями документа. Рассмотрим его.

1. Удалим предыдущие пять строк, добавленных в JavaScript.

2. Добавим в элемент <head> такое содержимое:

```
<style>
  .paragraph {
    color: white;
    background-color: black;
    padding: 10px;
    width: 250px;
    text-align: center;
  }
</style>
```

Теперь перейдём к очень полезному методу для общего манипулирования HTML — **Element.setAttribute()**. Этот метод принимает два аргумента: имя и значение атрибута, устанавливаемого для элемента.

Укажем в нашем абзаце имя класса выделения:

```
paragraphElement.setAttribute('class', 'paragraph')
```

Установка атрибута **class** через метод **setAttribute** — не единственный и, возможно, не самый идиоматичный способ. Современные браузеры поддерживают свойства **Element.className** и более мощный и функциональный **Element.classList.**

Обновим страницу и увидим следующие изменения: CSS по-прежнему применяется к абзацу, но на этот раз ему установлен класс, который выбран нашим правилом CSS, а не через встроенные (inline) стили CSS.

Можно выбрать любой понравившийся метод: оба имеют свои преимущества и недостатки. Первый метод принимает меньше настроек и хорош для простого использования, тогда как второй метод концептуально более чистый, то есть без смешивания CSS и JavaScript, без встроенных стилей, которые рассматриваются как плохая практика.

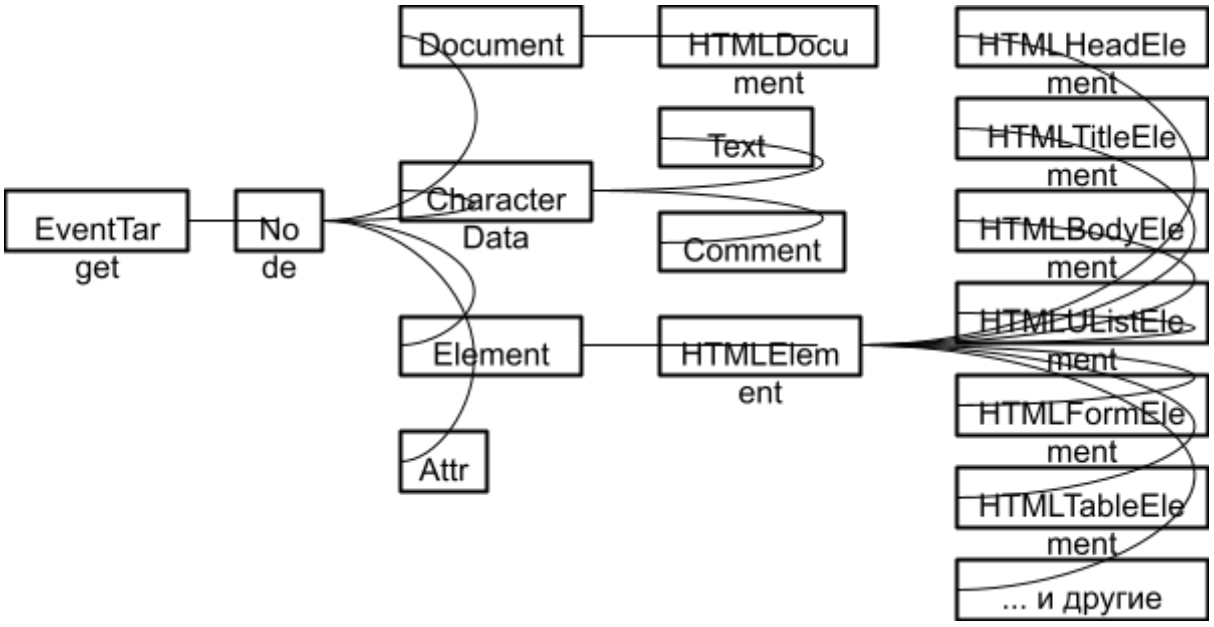
Навигация по элементам DOM-дерева

Для понимания возможностей навигации по элементам DOM-дерева рассмотрим типы элементов, а также, какие интерфейсы они реализуют.

Иерархия интерфейсов DOM

Стандарты DOM описывают интерфейсы для различных типов объектов.

Когда мы читаем документацию о DOM и его реализации в JavaScript, полезно немного узнать об иерархии классов DOM. Это понадобится для поиска по цепочке наследования, чтобы понять, какие атрибуты и методы наследует конкретный DOM-элемент.



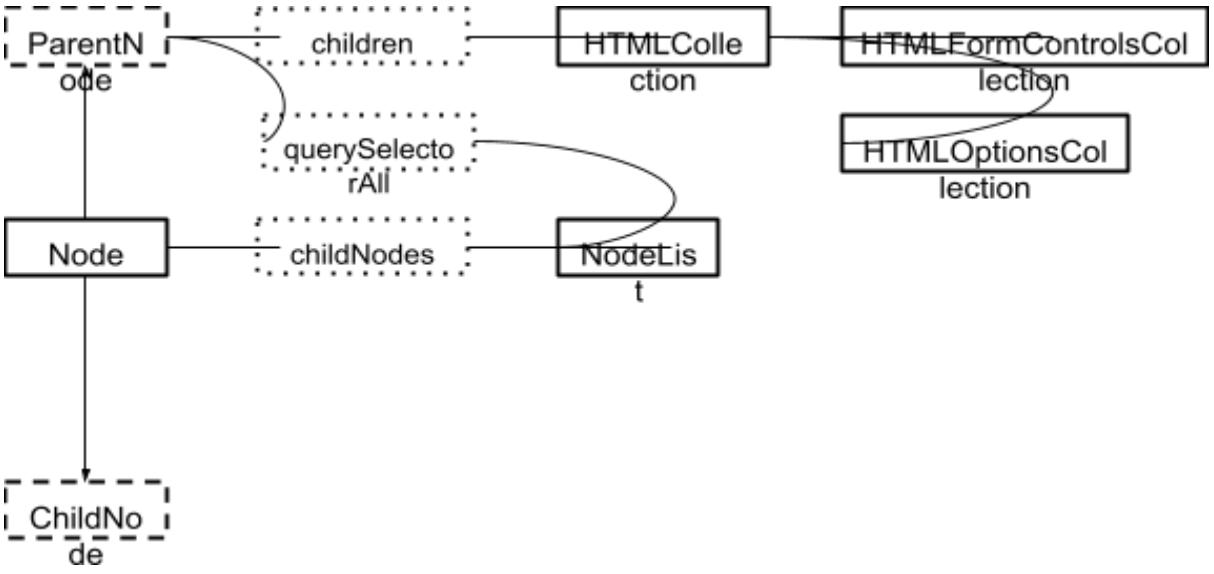
Как мы видим, все классы HTML-элементов (тегов) — это подклассы HTMLElement. В свою очередь, HTMLElement представляет собой подкласс Element.

Node — это базовый класс для всех DOM-интерфейсов, в том числе базовый класс для Document. У Node тоже есть родитель — **EventTarget**, интерфейс, реализуемый объектами, которые генерируют события и имеют подписчиков на эти события.

DOM-интерфейс разрабатывался как универсальный интерфейс не только для HTML, но и для любого XML-документа. **HTMLDocument** — это абстрактный интерфейс **DOM**, который обеспечивает доступ к специальным свойствам и методам, не представленным по умолчанию в регулярном XML-документе.

Интерфейсы коллекций в DOM

Коллекции в DOM представлены двумя базовыми интерфейсами. Рассмотрим их:



DOM-элементы, реализующие интерфейс Node, также реализуют два дополнительных интерфейса: ParentNode и ChildNode.

Интерфейс ParentNode содержит методы, относящиеся к Node-объектам, у которых могут быть потомки. А интерфейс ChildNode включает в себя методы, специфичные для объектов Node и имеющие родителя.

Таким образом, интерфейс Node может содержать поля childNodes и children, реализующие два разных интерфейса: NodeList и HTMLCollection соответственно. Эти поля используются **только для чтения** и представляют собой коллекции или псевдомассивы. На практике это означает, что они содержат поле **length** и поддерживают интерфейс получения элемента по индексу. Но мы не можем напрямую вызывать на них методы массивов типа **filter**, **map**, **reduce** и прочих.

Между этими коллекциями есть несколько различий:

1. **NodeList** включает в себя любые типы дочерних узлов, например, HTMLElement, Text, Comment.
2. **HTMLCollection** содержит только узлы типа HTMLElement, соответствующие HTML-тегам, например, и <p>, для которых поле **nodeType** равно **1**.
3. **NodeList** может быть как динамическим, так и статическим. Например, поле **childNodes** — это динамический NodeList, а NodeList, возвращаемый методом **Node.querySelectorAll,** считается статическим, то есть он не обновляет поле **length** при добавлении или удалении элемента из DOM-дерева.
4. **HTMLCollection** — это динамическая коллекция элементов.

Можно заметить, что в большинстве своём коллекции — это или массивы, или объекты. Нам нужен простой способ передать значения в массив, для этого существуют операторы spread и rest.

Spread, rest operator

Со стандартом ES2015 нам стали доступны очень полезные инструменты для работы с массивами: операторы spread и rest, а также деструктуризация.

Spread operator

Spread (англ. «расширять») — оператор расширения, или, по-другому, распространения данных из массива в атомарные элементы. Мы можем взять массив и вытащить все его элементы как отдельные переменные. Это бывает необходимо, когда мы хотим передать множество аргументов в функцию или перенести элементы одного массива в другой. Для этого перед массивом ставят многоточие (оператор spread). Давайте рассмотрим примеры:

```
const studentsGroup1PracticeTime = [
  {
    firstName: "Ivanov",
    practiceTime: 56
  },
  {
    firstName: "Petrov",
    practiceTime: 120
  },
  {
    firstName: "Sidorov",
    practiceTime: 148
  },
  {
    firstName: "Belkin",
    practiceTime: 20
  },
  {
    firstName: "Avdeev",
    practiceTime: 160
  }
];

const studentsGroup2PracticeTime = [
  {
    firstName: "Mankov",
    practiceTime: 87
  },
  {
```

```
    firstName: "Kistin",
    practiceTime: 133
  },
  {
    firstName: "Kotlyarov",
    practiceTime: 140
  },
  {
    firstName: "Peskov",
    practiceTime: 10
  },
];

// Напишем не очень удобную, но показательную функцию, которая умеет принимать неограниченное число аргументов и находить максимум среди них. Функция должна вызываться подобным образом: const maximum = findMax(4, 7, 10);
function findMax() {
  const values = arguments; // arguments — переменная, которая доступна внутри каждой функции и содержит в себе все аргументы, переданные в функцию. Является псевдомассивом.
  let maxValue = -Infinity;
  // Поскольку arguments является псевдомассивом, мы не можем применять к нему новые методы массивов, такие как forEach или reduce, а будем итерировать по старинке.
  for (let index = 0; index < values.length; index++) {
    if (values[index] > maxValue) maxValue = values[index];
  }
  return maxValue;
};
// Мы должны передавать в функции только числа, а в наших массивах содержатся объекты, поэтому сначала создадим массивы только со значениями времени, отработанного студентами.
const group1PracticeTime = studentsGroup1PracticeTime.map((student) => student.practiceTime);
const group2PracticeTime = studentsGroup2PracticeTime.map((student) => student.practiceTime);
// Теперь можем вызывать функцию поиска максимального значения. Она принимает множество числовых аргументов, а у нас есть только массив — тут нам и поможет оператор spread.

const maxTimeFromGroup1 = findMax(...group1PracticeTime); // ...group1PracticeTime вытянет из массива все элементы и передаст их в функцию как отдельные переменные.
// Это аналогично страшной и неудобной записи:
// findMax(group1PracticeTime[0], group1PracticeTime[1], group1PracticeTime[2], group1PracticeTime[3], group1PracticeTime[4])

console.log(maxTimeFromGroup1); // 160

const maxTimeFromGroup2 = findMax(...group2PracticeTime);
console.log(maxTimeFromGroup2); // 140
```

```
// Давайте также найдём максимально отработанное время среди двух групп. Мы можем сделать это, передав данные обоих массивов в функцию таким образом:
// findMax(...group1PracticeTime, ...group2PracticeTime);
// А можем объединить два массива в один — это очень частая операция, и оператор расширения (spread) очень в этом помогает.

const bothGroupsTime = [...group1PracticeTime, ...group2PracticeTime];
// Для объединения двух массивов нам нужно вытащить их элементы в один общий массив, поэтому мы объявляем новый массив, а в качестве его элементов делаем расширение элементов первого и второго массива. Также мы могли бы добавить в него и другие элементы.

const maxTimeBothGroups = findMax(...bothGroupsTime);
console.log(maxTimeBothGroups); // 160
```

Rest-operator

Оператор Rest (англ. «остальные», «оставшиеся») позволяет собрать оставшиеся аргументы функции в массив. Звучит немного странно, однако этот оператор позволяет не перечислять все аргументы функции как отдельные переменные, а получить их все одним массивом. Для его использования необходимо в функции, принимающей несколько аргументов, перечислить необходимые аргументы, а все оставшиеся, которые мы хотим собрать в один массив, — записать как …<имя массива>. Часто пишут *...rest*. Давайте перепишем наш предыдущий пример, используя оператор rest и тем самым избавившись от псевдомассива arguments.

```
const studentsGroup1PracticeTime = [
  {
    firstName: "Ivanov",
    practiceTime: 56
  },
  {
    firstName: "Petrov",
    practiceTime: 120
  },
  {
    firstName: "Sidorov",
    practiceTime: 148
  },
  {
    firstName: "Belkin",
    practiceTime: 20
  },
  {
    firstName: "Avdeev",
    practiceTime: 160
  }
];
```

```
const studentsGroup2PracticeTime = [
  {
    firstName: "Mankov",
    practiceTime: 87
  },
  {
    firstName: "Kistin",
    practiceTime: 133
  },
  {
    firstName: "Kotlyarov",
    practiceTime: 140
  },
  {
    firstName: "Peskov",
    practiceTime: 10
  },
];

// Напишем не очень удобную, но показательную функцию, которая умеет принимать неограниченное число аргументов и находить максимум среди них. Функция должна вызываться примерно следующим образом: const maximum = findMax(4, 7, 10);
function findMax(...values) { // тут мы принимаем все переданные аргументы и с помощью rest-оператора упаковываем их в массив values.
  // На этот раз values — уже настоящий массив, и мы можем использовать reduce для итерации по нему и для нахождения максимального числа.
  return values.reduce((acc, value) => {
    if (value > acc) return value;
    return acc;
  }, -Infinity);
};
// Создадим массивы только со значениями времени, отработанного студентами.
const group1PracticeTime = studentsGroup1PracticeTime.map((student) =>
student.practiceTime);
const group2PracticeTime = studentsGroup2PracticeTime.map((student) =>
student.practiceTime);

// Вызовем нашу функцию поиска максимума, используя оператор spread.
const maxTimeFromGroup1 = findMax(...group1PracticeTime);
console.log(maxTimeFromGroup1); // 160

const maxTimeFromGroup2 = findMax(...group2PracticeTime);
console.log(maxTimeFromGroup2); // 140

// Давайте также найдём максимально отработанное время среди двух групп.
const bothGroupsTime = [...group1PracticeTime, ...group2PracticeTime];

const maxTimeBothGroups = findMax(...bothGroupsTime);
console.log(maxTimeBothGroups); // 160
```

Теперь посмотрим на ещё один пример:

```
const saveFullNameInDB = (firstName, lastName, ...additionals) => {
  saveFirstName(firstName);
  saveLastName(lastName);
  saveAdditionals(additionals); // Благодаря rest оператору мы смогли собрать все дополнительные данные, которые были переданы для сохранения в базе данных, и можем передать их одним массивом в функцию сохранения дополнительных данных.
}
```

Работа с коллекциями

Создадим HTML-файл со следующим содержимым и откроем его в браузере:

```
<!DOCTYPE html>
<html>
  <head>
    <meta name="viewport" content="width=device-width,initial-scale=1">
    <title>Пример использования DOM коллекций</title>
  </head>
  <body>
    <div>
      <p>Первый параграф</p>
      <p>Второй параграф</p>
      <p>Третий параграф</p>
    </div>
  </body>
</html>
```

Добавим следующий JavaScript-код на страницу или используем JavaScript-консоль браузера:

```
const divElement = document.querySelector('div')
console.log(divElement.childNodes.length) // 7
console.log(divElement.children.length) // 3
```

Коллекции **childNodes** и **children** имеют разную длину.

Посмотрим, какие элементы содержатся в каждой коллекции. Чтобы перебрать элементы, сначала преобразуем коллекции в массивы с помощью статического метода **Array.from** или оператора **spread**.

```
Array.from(divElement.childNodes).forEach((childNode) => {
 console.log('childNode "%s" типа "%d"', childNode.nodeName, childNode.nodeType)
})

[...divElement.children].forEach((child) => {
 console.log('child "%s" типа "%d"', child.nodeName, child.nodeType)
})
```

Коллекция **children** содержит только элементы P, в отличие от **childNodes**, где также есть текстовые ноды (переносы строк).

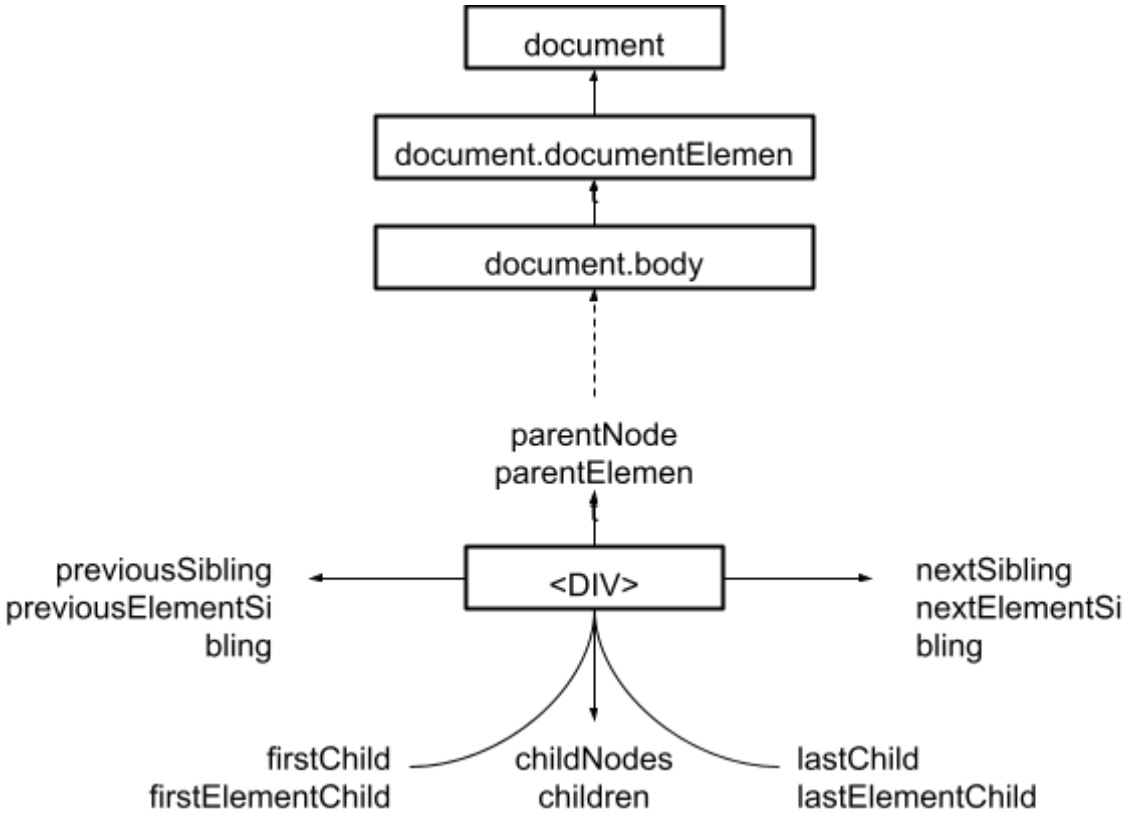
Рассмотрим разницу между динамическими и статическими коллекциями:

```
const allParagraphElements = divElement.querySelectorAll('p')

console.log('Static NodeList длина до: %d', allParagraphElements.length)
console.log('Dynamic NodeList длина до: %d', divElement.childNodes.length)
console.log('HTMLCollection длина до: %d', divElement.children.length)

const fourthParagraphElement = document.createElement('p')
fourthParagraphElement.textContent = 'Четвертый параграф'
divElement.appendChild(fourthParagraphElement)

console.log('Static NodeList длина после: %d', allParagraphElements.length)
console.log('Dynamic NodeList длина после: %d', divElement.childNodes.length)
console.log('HTMLCollection длина после: %d', divElement.children.length)
```

Статичный NodeList, возвращаемый из метода querySelectorAll, не меняет размер при добавлении новых нод в DOM, в отличие от динамических NodeList и HTMLCollection.

Методы для навигации по дереву DOM

По аналогии с полями **childNodes** и **children** интерфейсы Node и Element позволяют получить доступ к элементам в дереве, напрямую окружающих исходный элемент.



К таким элементам относятся:

1. **Родительский элемент** — Node.parentNode и Node.parentElement.
2. **Соседи исходного элемента** — Node.nextSibling / Node.previousSibling и Element.nextElementSibling / Element.previousElementSibling.
3. **Первый и последний дочерний элемент** — Node.firstChild / Node.lastChild и ParentNode.firstElementChild / ParentNode.lastElementChild.

Type	**Element**	**Node**
Parent	parentElement	parentNode
Children	children firstElementChild lastElementChild	childNodes firstChild lastChild
Siblings	nextElementSibling previousElementSibling	nextSibling previousSibling

Как и поля коллекций, эти поля используются только для чтения.

Для родительского элемента **parentNode** и **parentElement** практически всегда возвращают один и тот же элемент, за исключением случаев, когда **parentNode** элемента — не DOM Element. В таком случае **parentElement** возвращает **null**.

```
document.body.parentNode // Элемент <html>
document.body.parentElement // Элемент <html>
document.documentElement.parentNode // Нода document
document.documentElement.parentElement // null
(document.documentElement.parentNode === document) // true
(document.documentElement.parentElement === document) // false
```