**20 Глобальны объект. Создание функции с помощью конструктора (new Function).**

**Глобальный объект** в JavaScript — это объект, который доступен во всей программе и содержит все глобальные переменные, функции, а также базовые методы языка. Этот объект отличается в зависимости от окружения, в котором выполняется код.

* В браузере **глобальным объектом** является window.
* В Node.js — global.

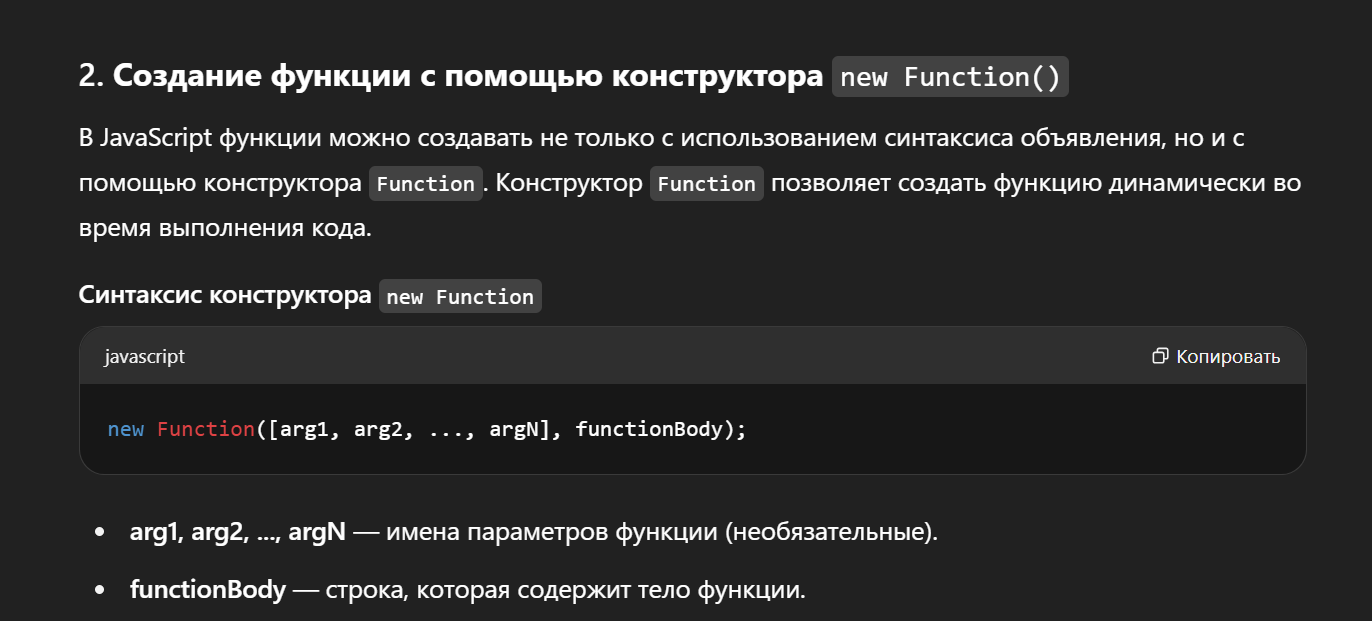
Когда вы создаете глобальные переменные или функции, они автоматически становятся свойствами глобального объекта.

**// Создание глобальной переменной**

**var x = 10;**

**// В глобальном контексте доступ к переменной x осуществляется через глобальный объект window**

**console.log(window.x); // 10**

****

 **Динамическое создание функций**: new Function позволяет создавать функции на лету. Это удобно, когда тело функции зависит от внешних данных или условий, и вы хотите создавать или изменять функции во время выполнения программы.

 **Использование строк**: тело функции в new Function передается как строка, что позволяет динамически генерировать и выполнять код. Однако это также делает код уязвимым для **инъекций** (например, если в строку передаются неподтвержденные данные).

 **Лексическая область видимости**: функции, созданные с помощью конструктора Function, не имеют доступа к переменным, определенным в внешнем контексте (локальной области видимости). Они работают только с глобальной областью видимости.

**21** [**Лексическое**](https://learn.javascript.ru/closure#leksicheskoe-okruzhenie) **окружение (LexicalEnvironment). Замыкание.**

**Лексическое окружение** — это концепция в JavaScript, которая описывает связь между **переменными и функциями**, которые доступны в процессе выполнения кода. Это структура, которая сохраняет все **переменные** и **функции**, объявленные в текущем контексте выполнения (например, в функции или глобальной области видимости).

Каждое **выполнение функции** или **создание нового блока кода** приводит к созданию нового лексического окружения.

**1.1. Структура лексического окружения**

Лексическое окружение состоит из двух частей:

1. **Объект переменных (Environment Record)**: здесь хранятся все переменные и функции, которые были объявлены в текущем контексте (внутри функции, блока и т.д.).
2. **Ссылка на внешнее лексическое окружение (Outer Lexical Environment Reference)**: это ссылка на окружение, в котором был создан данный контекст. Она позволяет создавать цепочку областей видимости.

Когда функция выполняется или создается, для неё создается новое лексическое окружение, которое хранит информацию о:

* Локальных переменных
* Аргументах функции
* Функциях, определённых внутри этой функции
* Ссылке на родительское окружение (если оно существует)

Эта цепочка лексических окружений образует **цепочку областей видимости** (scope chain).

**Замыкание** — это функция, которая **сохраняет доступ** к своему лексическому окружению, даже после того, как эта функция была вызвана и выполнена. Это позволяет функции продолжать ссылаться на переменные, которые были в её области видимости в момент создания. [Замыкание](https://ru.wikipedia.org/wiki/%D0%97%D0%B0%D0%BC%D1%8B%D0%BA%D0%B0%D0%BD%D0%B8%D0%B5_(%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5)) – это функция, которая запоминает свои внешние переменные и может получить к ним доступ.

#### 3.1. ****Как работает замыкание?****

Когда функция определена внутри другой функции, она может "замкнуть" доступ к переменным своей внешней функции, даже если эта внешняя функция завершила своё выполнение. Функция внутри сохраняет ссылку на переменные внешней функции через лексическое окружение.

**амыкания** позволяют функции запомнить и работать с переменными из своего лексического окружения, даже после завершения выполнения внешней функции.

* 1. **Объект функции. Именованное функциональное выражение (Named Function Expression).**

В JavaScript **функции** — это **объекты первого класса**, что означает, что они обладают всеми свойствами объектов. Функции могут быть присвоены переменным, переданы как аргументы и возвращены из других функций, а также имеют собственные свойства и методы.

#### Особенности объекта функции:

* **Функция как объект**: Функция в JavaScript — это не просто блок кода. Это объект, который можно назначить переменной, и этот объект будет содержать дополнительные свойства (например, name, length, prototype).
* **Методы функции**: Функции могут иметь свои методы, такие как call, apply, и другие.

**Именованное функциональное выражение** — это функция, которая присваивается переменной, но также имеет имя внутри тела функции. Это позволяет функции ссылаться на себя внутри её тела, что особенно полезно при **рекурсии**.

**Именованное функциональное выражение (Named Function Expression)** — это **функция**, которая присваивается переменной и имеет имя внутри своего тела. В отличие от обычных функциональных выражений, где функция анонимна, именованное выражение позволяет использовать имя функции внутри её самого тела, что может быть полезно для рекурсии или отладки.

## Особенности именованного функционального выражения:

1. **Имя функции** доступно только внутри самой функции.
2. Это выражение может быть присвоено переменной, а затем вызвано через эту переменную.
3. При рекурсивных вызовах функция может ссылаться на себя по имени.
   1. **Остаточные параметры и оператор расширения.**

**Остаточные параметры** (...) — это особый синтаксис в JavaScript, который позволяет функции принимать **неопределённое количество аргументов** и собирать их в массив. Остаточные параметры могут использоваться в определении функции для сбора всех оставшихся аргументов в массив.

#### Синтаксис:

javascript

Копировать

function functionName(...params) {

// тело функции

}

* **...params**: все переданные аргументы собираются в массив, который называется params.
* Остаточные параметры должны быть последними в списке параметров функции.

 **Остаточные параметры — это всегда массив**.

 **Остаточные параметры должны быть последними** в списке аргументов функции.

**Оператор расширения** (или **spread operator**) используется для **распаковки** элементов массива или объекта. Это противоположность остаточным параметрам, которые собирают элементы. Оператор расширения используется, когда нужно **распаковать** элементы из массива или объекта в новые структуры данных.

* Оператор расширения распаковывает элементы массива или объекта и добавляет их в новый массив или объект.
* В случае с объектами, если в обоих объектах есть одинаковые ключи, то значение из последнего объекта перезапишет значение из первого.

**24**

**Каррирование и частичное применение функции.**

**Каррирование** — это техника, которая преобразует функцию, принимающую несколько аргументов, в серию функций, каждая из которых принимает **один аргумент**. Каррирование позволяет создавать новые функции, частично применяя аргументы.

#### Суть каррирования:

Когда мы каррируем функцию, каждый её вызов принимает **один аргумент**, и результат этого вызова — это новая функция, которая ожидает следующий аргумент. Этот процесс продолжается до тех пор, пока все аргументы не будут переданы, и только тогда произойдёт выполнение основной логики функции.

#### Пример кар

**Частичное применение** — это техника, при которой мы создаём новую функцию, передав в неё часть аргументов исходной функции. Остальные аргументы остаются для дальнейшего использования.

Частичное применение позволяет **фиксировать часть аргументов функции**, создавая новую функцию, которая будет принимать оставшиеся аргументы.

****

* 1. **Генераторы. Функции-генераторы. Перебор объектов-генераторов.**

**Генераторы** в JavaScript — это функции, которые могут **приостанавливать** свое выполнение, а затем **возобновлять его**. Генераторы возвращают **итератор**, с помощью которого можно поочередно извлекать значения, генерируемые функцией, по мере необходимости.

### 1.1. **Что такое функция-генератор?**

Функция-генератор в JavaScript определяется с помощью ключевого слова function\*. Эти функции используют ключевое слово yield для приостановки выполнения, и они возвращают объект-итератор, с помощью которого можно перебирать значения, возвращаемые функцией.

### **Как работает генератор?**

Генераторы работают с помощью ключевого слова yield, которое приостанавливает выполнение функции и возвращает значение. Когда вызывается метод next() итератора, выполнение функции возобновляется с места, где был приостановлен предыдущий вызов yield.

* 1. **Методы объектов, this. Оператор опциональной последовательности.**

Методы объектов — это **функции**, которые являются свойствами объекта. Внутри метода объекта можно использовать ключевое слово this, чтобы ссылаться на сам объект. Это позволяет методам работать с данными, хранящимися в объекте.

#### Пример:

const person = {

firstName: 'John',

lastName: 'Doe',

fullName: function() {

return `${this.firstName} ${this.lastName}`;

}

};

console.log(person.fullName()); // "John Doe"

Оператор опциональной последовательности (?.) в JavaScript используется для **проверки наличия свойства** в объекте, прежде чем обращаться к этому свойству, что позволяет избежать ошибок в случае, если свойство не существует.

Оператор ?. выполняет **безопасное обращение** к свойствам и методам объекта, и если объект не существует (или равен null или undefined), то выражение сразу возвращает undefined, не вызывая ошибку.

* 1. **Преобразование объектов.**
  2. **Создание объектов через "new".**

### **Создание объектов с использованием конструктора** new

В JavaScript можно создавать объекты не только с помощью литералов объектов {}, но и с помощью оператора new и конструктора. Конструкторы позволяют создавать экземпляры объектов с определёнными свойствами и методами. Это важная часть объектно-ориентированного программирования в JavaScript.

### 2. **Как работает** new**?**

Когда используется оператор new с функцией-конструктором, создается **новый экземпляр объекта**, связанный с этой функцией. Процесс создания объекта через new включает следующие шаги:

1. **Создание нового пустого объекта.**
2. **Привязка контекста this** к новому объекту внутри конструктора.
3. **Выполнение кода конструктора.**
4. Если конструктор не возвращает явный объект, то по умолчанию возвращается новый объект, созданный с помощью new.
   1. **Флаги и дескрипторы свойств.**

В JavaScript каждый объект имеет свойства, которые могут быть настроены с помощью **дескрипторов свойств**. Эти дескрипторы определяют, как поведение и доступ к свойствам объекта. С помощью дескрипторов можно управлять, можно ли изменять, удалять или перечислять свойства объекта, а также определять их значение.

#### ****Дескриптор свойства**** представляет собой объект, который может содержать следующие параметры:

* **value**: значение свойства.
* **writable**: флаг, который указывает, может ли значение свойства быть изменено.
* **enumerable**: флаг, который указывает, будет ли свойство показываться при переборе объекта (например, в цикле for...in или методах Object.keys()).
* **configurable**: флаг, который указывает, может ли свойство быть удалено или изменены его атрибуты (например, writable или enumerable).

Эти параметры могут быть установлены или изменены с помощью метода **Object.defineProperty()** и **Object.defineProperties()**.

* 1. **Геттеры и сеттеры.**

**Геттеры** и **сеттеры** — это специальные методы объектов, которые позволяют управлять доступом к свойствам объекта. Они используются для **получения** (геттеры) и **установки** (сеттеры) значений свойств объекта, но работают через специальные методы, а не прямое обращение к свойствам.

#### В чем разница между геттерами и сеттерами?

* **Геттер** — это метод, который позволяет читать значение свойства, но, по сути, позволяет это делать через доступ к свойству объекта, как если бы это было обычное свойство.
* **Сеттер** — это метод, который позволяет **устанавливать** значение свойства, но через вызов функции, а не прямое присваивание.

**31Декораторы. Методы call(), apply(), bind().**

**екораторы** — это особые функции, которые модифицируют поведение других функций или классов. Они могут быть применены к методам, свойствам или классам для изменения их поведения без изменения исходного кода.

Декораторы предоставляют функциональность, такую как **логирование**, **кеширование** или **проверка доступа**, и позволяют добавлять новые поведения к объектам и функциям. В JavaScript декораторы активно используются с **классами** и **методами классов**.

#### ****Метод**** call()

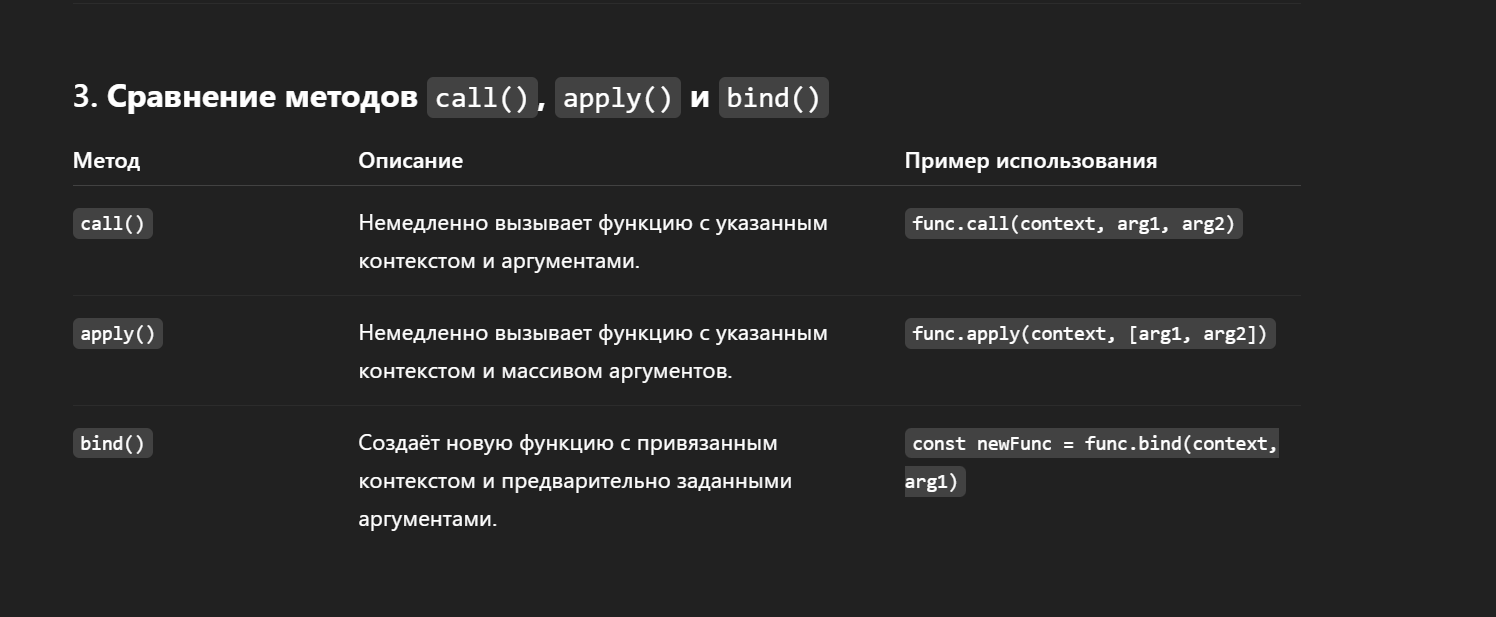
Метод call() позволяет вызвать функцию с заданным контекстом (this) и аргументами. Это полезно, если нужно явно указать, какой объект будет использоваться в качестве контекста внутри функции.

#### ****Метод**** apply()

Метод apply() работает так же, как и call, но принимает аргументы в виде массива. Это полезно, когда количество аргументов заранее неизвестно, или когда мы уже имеем массив аргументов.

##### Синтаксис:

Метод bind() создаёт **новую функцию**, которая привязывает контекст (this) и заранее задаёт некоторые аргументы. Это особенно полезно, если нужно использовать функцию позднее с уже определённым контекстом и аргументами.

****

 **Декораторы** позволяют изменять поведение функций или методов без изменения их исходного кода. Это полезно для добавления новых возможностей, таких как логирование, кэширование или проверка прав доступа.

 **Методы call(), apply() и bind()** являются важными инструментами для работы с контекстом (this):

* call() и apply() немедленно вызывают функцию с заданным контекстом и аргументами.
* bind() создаёт новую функцию с привязанным контекстом и аргументами, но не вызывает её немедленно.
  1. **Функции setTimeout and setInterval.**
* Функции **setTimeout** и **setInterval** в JavaScript позволяют выполнить код через определённое время. Эти функции являются частью **Web API**, предоставляемых браузером, и позволяют отложенно или периодически выполнять код. Они используются для асинхронных операций и управления временем в приложениях.
* **setTimeout()** — это функция, которая позволяет выполнить **однократное** действие после указанной задержки (в миллисекундах).

**setInterval()** — это функция, которая позволяет выполнять указанную функцию **непрерывно через определённые промежутки времени**.

* + 1. **Прототипное наследование. Собственные и унаследованные свойства. Свойство F.prototype.**
* **Прототипное наследование** — это механизм, с помощью которого объекты могут наследовать свойства и методы от других объектов. В JavaScript **каждый объект** имеет свойство prototype, которое указывает на другой объект, от которого он наследует методы и свойства.
* Когда вы пытаетесь получить доступ к свойству или методу объекта, и оно не найдено в самом объекте, JavaScript будет искать его в объекте, на который указывает его prototype. Этот процесс продолжается до тех пор, пока не будет найдено соответствующее свойство или не будет достигнут **null** (когда объект уже не имеет прототипа).

### **Свойство** F.prototype

Каждая функция в JavaScript имеет свойство prototype. Это свойство используется для добавления методов и свойств, которые будут доступны всем экземплярам, созданным с помощью этой функции.

* 1. **Классы. Class Expression. Приватные и защищённые методы и свойства.**

В JavaScript классы были введены в ES6 и предоставляют более синтаксически удобный способ работы с объектами и их наследованием. Классы — это синтаксическая обертка над функциями-конструкторами и прототипами, позволяющая работать с объектно-ориентированным программированием (ООП) более интуитивно.

**Class Expression** — это синтаксис, при котором класс определяется как выражение. Это позволяет присваивать класс переменной или передавать его в качестве аргумента функции.

В JavaScript классы по умолчанию не поддерживают приватные или защищённые свойства, как это бывает в других языках (например, в Java или C++). Однако, с введением **приватных полей** в ES2022, появилась возможность создавать **приватные свойства и методы** с помощью синтаксиса #.

#### Приватные свойства и методы (с использованием #):

Приватные свойства и методы класса доступны только внутри самого класса и не могут быть доступны или изменены за пределами класса.

JavaScript **не имеет синтаксической поддержки защищённых свойств и методов**. Однако, защищённые свойства часто симулируются через соглашения, такие как использование префикса \_ для обозначения защищённых свойств. Это просто соглашение, а не настоящий механизм инкапсуляции. Тем не менее, такие свойства часто трактуются как защищённые, и разработчики не должны изменять их напрямую.

 **Классы** в JavaScript позволяют создавать объекты с определёнными свойствами и методами, а также поддерживают **прототипное наследование**.

 **Class Expression** предоставляет возможность динамически создавать классы через выражения.

 **Приватные свойства и методы** можно создать с помощью синтаксиса #, который доступен в последних версиях JavaScript (ES2022).

 **Защищённые свойства и методы** в JavaScript симулируются через соглашения (например, префикс \_), так как язык не поддерживает их напрямую.

 **Геттеры и сеттеры** предоставляют возможность контролировать доступ к свойствам и добавлять логику в процесс получения или установки значений.

* 1. **Наследование классов. Переопределение методов. Статические свойства и методы.** **Оператор instanceof**

Наследование в JavaScript позволяет создавать новый класс (подкласс), который наследует свойства и методы от другого класса (родительского класса). Это позволяет повторно использовать код и расширять функциональность классов.

В JavaScript наследование классов осуществляется с использованием ключевых слов extends и super.

 **extends** — ключевое слово, которое используется для создания подкласса, наследующего от родительского класса.

 **super()** — это метод, который вызывает конструктор родительского класса и позволяет передать аргументы родительскому классу.

Переопределение метода — это процесс, при котором метод в дочернем классе **замещает** метод с тем же именем в родительском классе.

**Статические методы и свойства** принадлежат самому классу, а не экземплярам этого класса. Они не могут быть вызваны через экземпляры объекта, но доступны через сам класс.

Оператор **instanceof** используется для проверки, является ли объект экземпляром определённого класса или его подкласса. Это позволяет проверить, принадлежит ли объект определённому типу.

1. **Наследование классов** позволяет создавать новые классы, которые наследуют свойства и методы других классов с помощью ключевого слова extends.
2. **Переопределение методов** позволяет дочерним классам изменять или расширять поведение методов родительского класса.
3. **Статические свойства и методы** принадлежат самому классу и не доступны через экземпляры этого класса, но могут быть полезны для операций, не связанных с экземплярами.
4. **Оператор instanceof** позволяет проверять, является ли объект экземпляром конкретного класса или его подкласса.

Если у вас есть дополнит

* 1. **Модули. Основные возможности модулей.**
* Модули в JavaScript позволяют организовать код в **отдельные файлы**, которые можно импортировать и экспортировать для удобства разработки. Модули помогают **структурировать** и **управлять зависимостями** в больших проектах, а также обеспечивают **инкапсуляцию** и **повторное использование** кода.
* В JavaScript модули поддерживаются с версии ES6, и они используют ключевые слова **import** и **export** для обмена кодом между различными файлами.

#### ****Экспорт и импорт****

**Экспорт** используется для **выведения** функциональности (например, функций, объектов или переменных) из одного модуля, чтобы другие модули могли её использовать. **Импорт** используется для **получения** функциональности из других модулей.

##### Пример экспорта и импорта:

Файл math.js (модуль для работы с числами)

#### ****Экспорт по умолчанию****

Иногда нужно экспортировать **один основной элемент** из модуля. Для этого используется **экспорт по умолчанию**.

#### ****Импорт всего модуля****

Иногда нужно импортировать все элементы из модуля. Это можно сделать с помощью синтаксиса \* as.

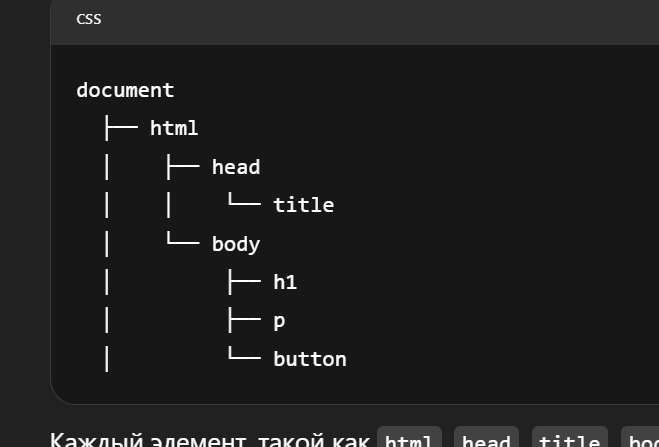
#### ****Реэкспорт****

В JavaScript можно **реэкспортировать** элементы из одного модуля в другой, чтобы агрегировать или передавать функциональность между модулями.

* 1. **Окружение: DOM, BOM. Дерево DOM.**

#### ****DOM (Document Object Model)****

**DOM** представляет собой **объектную модель документа**, которая позволяет программно изменять структуру, стиль и содержание веб-страницы. DOM описывает структуру HTML-документа как иерархическое дерево объектов, где каждый элемент, атрибут или текст на странице является объектом.

****

#### ****BOM (Browser Object Model)****

**BOM** представляет собой набор объектов, которые позволяют взаимодействовать с браузером, управлять окнами, вкладками и историей браузера. BOM предоставляет возможность управлять поведением браузера, например, изменять URL страницы, работать с окнами и вкладками, обрабатывать события браузера.

Примеры объектов BOM:

* **window** — объект, представляющий текущее окно браузера.
* **navigator** — объект для получения информации о браузере.
* **location** — объект для работы с адресом URL.
* **history** — объект для работы с историей браузера.

**Дерево DOM** — это **иерархическая структура** всех элементов документа, где каждый элемент является узлом дерева. Это позволяет JavaScript обращаться к элементам страницы и изменять их. Каждый узел может быть элементом, атрибутом или текстом.

* 1. **Навигация и методы поиска DOM-элементов.**

Навигация по **DOM** и **поиск DOM-элементов** — это основа работы с веб-страницами через JavaScript. С помощью различных методов и свойств мы можем **получать**, **изменять**, **удалять** и **перемещать** элементы в структуре документа.

#### Основные методы для навигации и поиска элементов в DOM:

1. **document.getElementById()**
2. **document.getElementsByClassName()**
3. **document.getElementsByTagName()**
4. **document.querySelector()**
5. **document.querySelectorAll()**
6. **parentNode и children**
7. **previousElementSibling и nextElementSibling**

#### document.getElementById()

Метод **getElementById()** возвращает элемент по его **ID**. Это самый быстрый метод поиска, так как ID должен быть уникальным на странице.

javascript

Копировать

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

console.log(element);

* В примере выше мы ищем элемент с ID myElement. Если элемент найден, он будет возвращён, иначе — null.

#### 2.2. document.getElementsByClassName()

Метод **getElementsByClassName()** возвращает **HTMLCollection** всех элементов, которые имеют указанный класс. Это "живой" список, что значит, что он будет автоматически обновляться при изменении DOM.

javascript

Копировать

const elements = document.getElementsByClassName('myClass');

console.log(elements); // HTMLCollection с элементами

* В отличие от метода getElementById(), этот метод возвращает коллекцию всех элементов с указанным классом, а не один элемент.

#### 2.3. document.getElementsByTagName()

Метод **getElementsByTagName()** возвращает **HTMLCollection** всех элементов с указанным тегом.

javascript

Копировать

const elements = document.getElementsByTagName('div');

console.log(elements); // HTMLCollection с элементами <div>

* Это метод может быть полезен, если нужно найти все элементы одного типа, например, все <p> или все <h1> на странице.

#### 2.4. document.querySelector()

Метод **querySelector()** позволяет искать **первый элемент**, который соответствует CSS-селектору. Это более гибкий метод, чем getElementById() или getElementsByClassName(), так как позволяет использовать любые CSS-селекторы.

javascript

Копировать

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

console.log(element); // Возвращает первый элемент с классом myClass

* Это один из самых популярных методов, так как он позволяет использовать любой селектор, включая **комбинированные селекторы** (например, #id .class, div > p).

#### 2.5. document.querySelectorAll()

Метод **querySelectorAll()** работает аналогично querySelector(), но возвращает **все элементы**, которые соответствуют CSS-селектору, в виде **NodeList** (который можно перебрать с помощью forEach).

javascript

Копировать

const elements = document.querySelectorAll('div.myClass');

console.log(elements); // NodeList с элементами <div> с классом myClass

* В отличие от getElementsByClassName() и getElementsByTagName(), возвращаемый **NodeList** не является "живым" и не обновляется автоматически при изменении DOM.

### 3. **Навигация по DOM-дереву**

После того как мы нашли нужный элемент, нам часто нужно перемещаться по соседним элементам или родительскому элементу.

#### 3.1. parentNode

С помощью **parentNode** можно получить **родительский элемент** текущего элемента.

javascript

Копировать

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

const parent = element.parentNode;

console.log(parent); // Родительский элемент

* Этот метод полезен, когда нужно перемещаться вверх по DOM-дереву.

#### 3.2. children

С помощью **children** можно получить **все дочерние элементы** текущего элемента (которые являются элементами, а не текстом или комментариями).

javascript

Копировать

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

const children = element.children;

console.log(children); // Все дочерние элементы в виде HTMLCollection

* Этот метод полезен, если нужно работать только с элементами (без текстовых узлов и комментариев).

#### 3.3. firstChild ****и**** lastChild

Методы **firstChild** и **lastChild** возвращают **первый и последний дочерний узлы** соответственно. Важно помнить, что эти методы могут вернуть не только элементы, но и текстовые узлы.

javascript

Копировать

const firstChild = element.firstChild; // Первый дочерний узел

const lastChild = element.lastChild; // Последний дочерний узел

* Если нужно работать именно с элементами, лучше использовать firstElementChild и lastElementChild, которые игнорируют текстовые узлы.

#### 3.4. firstElementChild ****и**** lastElementChild

Если важно, чтобы вернулись только **элементы**, используйте **firstElementChild** и **lastElementChild**.

javascript

Копировать

const firstElement = element.firstElementChild; // Первый дочерний элемент

const lastElement = element.lastElementChild; // Последний дочерний элемент

#### 3.5. previousElementSibling ****и**** nextElementSibling

С помощью **previousElementSibling** и **nextElementSibling** можно получить **соседей** текущего элемента.

javascript

Копировать

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

const prev = element.previousElementSibling; // Сосед слева

const next = element.nextElementSibling; // Сосед справа

* Эти методы позволяют перемещаться по элементам в пределах одного родителя.

### 4. **Пример: Поиск и манипуляция DOM-элементами**

html

Копировать

<!DOCTYPE html>

<html>

<head>

<title>DOM Navigation Example</title>

</head>

<body>

<div id="container">

<h1>Heading</h1>

<p class="text">This is a paragraph.</p>

<p class="text">Another paragraph.</p>

<button id="btn">Click Me</button>

</div>

<script>

// Поиск элемента по ID

const button = document.getElementById('btn');

// Поиск всех параграфов с классом 'text'

const paragraphs = document.querySelectorAll('.text');

// Изменение содержимого параграфов

paragraphs.forEach(paragraph => {

paragraph.textContent = 'Text updated!';

});

// Добавление обработчика события на кнопку

button.addEventListener('click', () => {

alert('Button clicked!');

});

</script>

</body>

</html>

### Пояснение:

* Мы находим элемент button по ID и добавляем обработчик события.
* Мы находим все элементы с классом .text и изменяем их содержимое.
* Эти операции выполняются с использованием различных методов поиска и навигации по DOM.

### Заключение

1. **Методы поиска DOM-элементов** позволяют находить элементы по ID, классу, тегу, селектору и т. д.
2. **Навигация по DOM-дереву** позволяет перемещаться по родительским и дочерним элементам, а также по соседям.
3. Использование методов **querySelector** и **querySelectorAll** даёт гибкость в работе с элементами, позволяя использовать любые CSS-селекторы.
4. Эти возможности позволяют эффективно работать с динамическим контентом на веб-странице и модифицировать её структуру и поведение.

Если у вас есть дополнительные вопросы или примеры, не стесняйтесь спрашивать!

* 1. **Свойства узлов: тип, тег и содержимое.**

### **Свойства узлов в DOM**

Когда мы работаем с DOM, каждый элемент документа представлен **узлом**. Узлы могут быть различных типов (элементы, текстовые узлы, комментарии и т. д.), и для работы с ними есть несколько полезных свойств. Основные свойства, которые позволяют нам работать с узлами, — это **тип узла**, **тег** и **содержимое**.

#### 1.1. ****Тип узла (Node Type)****

Каждый узел в DOM имеет свой тип, который можно получить через свойство **nodeType**. Тип узла описывает, какой именно тип информации он представляет. Например:

* **Элемент** — это узел, представляющий HTML-элемент.
* **Текст** — это узел, представляющий текст внутри элемента.
* **Комментарий** — это узел, представляющий комментарий в HTML.

Вот основные типы узлов:

* **1** — элемент (Element).
* **3** — текстовый узел (Text).
* **8** — комментарий (Comment).
* **9** — документ (Document).
* **11** — фрагмент документа (DocumentFragment).

#### ****Тег узла (Node Name)****

**nodeName** возвращает имя узла в виде строки. Для элементов это имя тега (например, "DIV", "P", "H1" и т. д.). Для текстовых узлов и комментариев это будет "TEXT" и "COMMENT" соответственно.

Пример:

javascript

Копировать

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

console.log(element.nodeName); // "P" (имя тега)

* Если мы ищем элемент с тегом <p>, то его nodeName будет равен "P", независимо от регистра (в DOM все теги приведены к верхнему регистру).

#### 1.3. ****Содержимое узла (Node Value)****

Свойство **nodeValue** позволяет получить или изменить содержимое узла. Для **текстовых узлов** это будет сам текст, а для других типов узлов это будет null.

* **Для текстовых узлов**: свойство nodeValue содержит сам текст.
* **Для других узлов (например, элементы и комментарии)**: свойство nodeValue равно null.

#### Пример:

html

Копировать

<p>This is a text node.</p>

javascript

Копировать

const paragraph = document.querySelector('p');

console.log(paragraph.firstChild.nodeValue); // "This is a text node."

* В этом примере мы получаем **первый дочерний узел** параграфа, который является текстом. С помощью nodeValue мы получаем сам текст "This is a text node.".

#### Пример изменения содержимого:

javascript

Копировать

const paragraph = document.querySelector('p');

paragraph.firstChild.nodeValue = 'New text content';

console.log(paragraph.firstChild.nodeValue); // "New text content"

* Мы изменяем текстовый узел с помощью nodeValue, и теперь текст в параграфе изменится.

### 2. **Типы узлов и их работа в DOM**

В DOM все узлы можно классифицировать по типу, и каждый тип узла предоставляет определённые свойства и методы. Рассмотрим несколько примеров работы с разными типами узлов.

#### 2.1. ****Элементы (Node Type 1)****

Элементы DOM представляют теги HTML (например, <div>, <p>, <a>). Для работы с элементами чаще всего используются свойства и методы, такие как nodeName, nodeValue, innerHTML, textContent, и другие.

Пример:

html

Копировать

<div id="myDiv">

<p>Some text inside paragraph.</p>

</div>

javascript

Копировать

const div = document.getElementById('myDiv');

console.log(div.nodeType); // 1 (Элемент)

console.log(div.nodeName); // "DIV" (Имя тега)

console.log(div.nodeValue); // null (Элементы не имеют nodeValue)

console.log(div.innerHTML); // "<p>Some text inside paragraph.</p>" (HTML содержимое)

#### 2.2. ****Текстовые узлы (Node Type 3)****

Текстовые узлы содержат текст внутри элементов и могут быть получены через childNodes или firstChild.

html

Копировать

<p>Example paragraph text.</p>

javascript

Копировать

const paragraph = document.querySelector('p');

const textNode = paragraph.firstChild;

console.log(textNode.nodeType); // 3 (Текстовый узел)

console.log(textNode.nodeValue); // "Example paragraph text."

* Текстовые узлы содержат **текст**, и мы можем работать с ним через nodeValue.

#### 2.3. ****Комментарии (Node Type 8)****

Комментарии в DOM представляют собой узлы с типом 8.

html

Копировать

<!-- This is a comment -->

javascript

Копировать

const commentNode = document.createComment('This is a comment');

console.log(commentNode.nodeType); // 8 (Комментарий)

console.log(commentNode.nodeValue); // "This is a comment"

* Мы можем создать комментарий с помощью document.createComment() и манипулировать его содержимым через nodeValue.

### 3. **Пример работы с деревом DOM**

Предположим, что у нас есть следующая структура HTML:

html

Копировать

<div>

<h1>Title</h1>

<p>This is a paragraph.</p>

<button id="btn">Click me</button>

</div>

Пример работы с узлами:

javascript

Копировать

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

// Получаем все дочерние узлы

console.log(div.childNodes); // Возвращает NodeList всех дочерних узлов (включая текстовые узлы)

// Получаем только элементы

console.log(div.children); // Возвращает HTMLCollection всех дочерних элементов (без текстовых узлов)

const firstChild = div.firstChild;

console.log(firstChild.nodeType); // Тип первого дочернего узла (например, текстовый узел или элемент)

const button = document.getElementById('btn');

console.log(button.nodeName); // "BUTTON"

### Заключение

1. **Тип узла** (nodeType) определяет, какой тип информации представляет узел: элемент, текст, комментарий и т.д.
2. **Тег узла** (nodeName) возвращает имя тега для элементов, а для других типов узлов возвращает строковые значения, такие как "TEXT" для текстовых узлов.
3. **Содержимое узла** (nodeValue) представляет данные узла. Для текстовых узлов это будет сам текст, для других узлов — null.