# **JAVASCRIPT**

1. [Типы данных js](#Типы_данных_js)
2. [Сравнение (==, ===)](#Сравнение(двойное_и_тройное_равно))
3. [Контекст (this)](#Контекст_(this))
4. <#Методы_итерации_(и_еще_генераторы_yield)>
5. [#Отличие стрелочных функций от обычных](#Отличие_стрелочных_функций_от_обычных)
6. [#Виды функций](#Виды_функций)
7. <#Прототипы>
8. [#Class ООП (классы в js ооп)](#Class_ООП_(классы_в_js_ооп))
9. [#Что такое события в JavaScript и какие события знаете](#Что_такое_события_в_JavaScript_и_какие_с)
10. [#Как обрабатывать события](#Как_обрабатывать_события)
11. [# Какие методы строк знаете](#_Какие_методы_строк_знаете)
12. [#Что такое шаблонная строка](#Что_такое_шаблонная_строка)
13. [# Какие методы массива знаете](#_Какие_методы_массива_знаете)
14. [#Как объединить два массива](#Как_объединить_два_массива)
15. [#Как отсортировать массив](#Как_отсортировать_массив)
16. [#Что такое преобразование типов](#Что_такое_преобразование_типов)
17. [#Как работает обработка ошибок](#Как_работает_обработка_ошибок_)
18. [#Что такое hoisting (подъем)](#Что_такое_hoisting_(подъем))
19. [#Как вы дебажите свой код](#Как_вы_дебажите_свой_код)
20. [#Что такое асинхронность и как работает в JavaScript](#Что_такое_асинхронность_и_как_работает_в)
21. [#Что такое Promise и как им пользоваться](#Что_такое_Promise_и_как_им_пользоваться)
22. [#Какие методы Promise знаете](#Какие_методы_Promise_знаете)
23. [#Что такое async await](#Что_такое_async_await)
24. [# Разница promise и async await (promise vs async await)](#_Разница_promise_и_async_await_(promise_)
25. [#Что такое обратные вызовы(callback) и замыкание](#Что_такое_обратные_вызовы(callback)_и_за)
26. [#Host и native объекты (хост и нейтив)](#Host_и_native_объекты_(хост_и_нейтив))
27. [#Каррирование(функция карри)](#Каррирование(функция_карри))
28. [#Mutable и Immutable](#Mutable_и_Immutable)
29. <#Анаморфизмы>
30. <#Катаморфизмы>
31. [#Функции-контсрукторы и их отличие от классов](#Функции-контсрукторы_и_их_отличие_от_кла)
32. <#SOLID>
33. [#Области видимости(Какие есть области видимости)](#Области_видимости(Какие_есть_области_вид)
34. [#Что такое IIFE](#Что_такое_IIFE)
35. [#Что такое боксинг и анбоксинг](#Что_такое_боксинг_и_анбоксинг)
36. [#Что такое аксессор и что такое дескриптор](#Что_такое_аксессор_и_что_такое_дескрипто)
37. [#Есть ли length у объекта](#Есть_ли_length_у_объекта)
38. [#Какие методы массивов мутируют исходный массив](#Какие_методы_массивов_мутируют_исходный_)
39. [#Как можно скопировать объект](#Как_можно_скопировать_объект)
40. [#Как можно задать объекту свойство и параметры свойства. Какие бывают параметры свойства](#Как_можно_задать_объекту_свойство_и_пара)
41. [#В чем плюсы и минусы иммутабельности](#В_чем_плюсы_и_минусы_иммутабельности)
42. [#Зачем нужен useStrict](#Зачем_нужен_useStrict)
43. [#Как можно сравнить два объекта в js](#Как_можно_сравнить_два_объекта_в_js)
44. [#Возможно ли узнать сколько аргументов принимает функция](#Возможно_ли_узнать_сколько_аргументов_пр)
45. [#Как можно удалить произвольный элемент массива](#Как_можно_удалить_произвольный_элемент_м)
46. [#В чём различия между коллекциями Map и WeakMap Есть ли у Set какой-нибудь аналог](#В_чём_различия_между_коллекциями_Map_и_W)
47. [# Какие способы работы с асинхронным кодом ты знаешь](#_Какие_способы_работы_с_асинхронным_кодо)
48. [#В чём разница между promise.allsettled и promise.all](#В_чём_разница_между_promise.allsettled_и)
49. <#Ecmascript(ES)>
50. [#Что представляют из себя регулярные выражения и для чего они нужны](#Что_представляют_из_себя_регулярные_выра)
51. [#Почему два объекта с одинаковыми полями при сравнении дают false](#Почему_два_объекта_с_одинаковыми_полями_)
52. [#Как проверить из какого класса был создан объект](#Как_проверить_из_какого_класса_был_созда)
53. [#Декораторы в js](#Декораторы_в_js)

**Typescript**

1. <#Типы_данных_ts>
2. <#Generics(дженерики)>
3. <#Type_Utilities(Утилиты_типов)>
4. <#Type_class_interface(отличие_особенности>
5. <#Чем_отличается_type_от_interface>
6. <#Зачем_нужен_omit_never_unknown>
7. <#Что_такое_enum>
8. <#Чем_отличается_ts_и_Proptypes_TypeScript>
9. <#Влияет_ли_TS_на_исполнение_JS_в_рантайме>
10. <#Что_такое_утиная_типизация>
11. <#Type_guards_сужающие_и_расширяющие_типы>
12. <#Чем_отличается_any_от_unknown>

**Nest.js**

# **JAVASCRIPT**

1. Типы данных js

**Примитивные типы:**

* + number
  + string
  + boolean
  + Null - означает “ничего”, “пусто”
  + Undefined - значение не присвоено
  + Symbol - это уникальный и неизменяемый тип данных, который может быть использован как идентификатор для свойств объектов.

**Ссылочные типы:**

* + object
  + массивы, функции и объекты

Если мы инициализируем переменную, но не присваиваем ей значение, туда помещается специальный «маркер», который отображается при выводе на экран как undefined. Null присваиваем самостоятельно.

**Что такое ссылочные типы данных?** Ссылочные типы данных в JavaScript - это типы данных, значения которых хранятся и передаются по ссылке. При присваивании переменной ссылочного типа копируется ссылка на объект, а не сам объект.

**В чем разница ссылочных типов от примитивных?** Примитивные типы данных хранят свои значения непосредственно в переменной, в то время как ссылочные типы данных хранятся и передаются по ссылке.

**Что делает оператор typeof?** Оператор typeof используется для определения типа значения переменной или выражения в JavaScript. Он возвращает строку, указывающую тип данных.

2. Сравнение (==, ===)

**В чем разница между == и ===? Что лучше использовать?** Оператор == сравнивает значения с приведением типов, в то время как === сравнивает значения без приведения типов. Рекомендуется использовать ===, чтобы и збежать неожиданных результатов при сравнении значений разных типов.

3. **Контекст (this)**

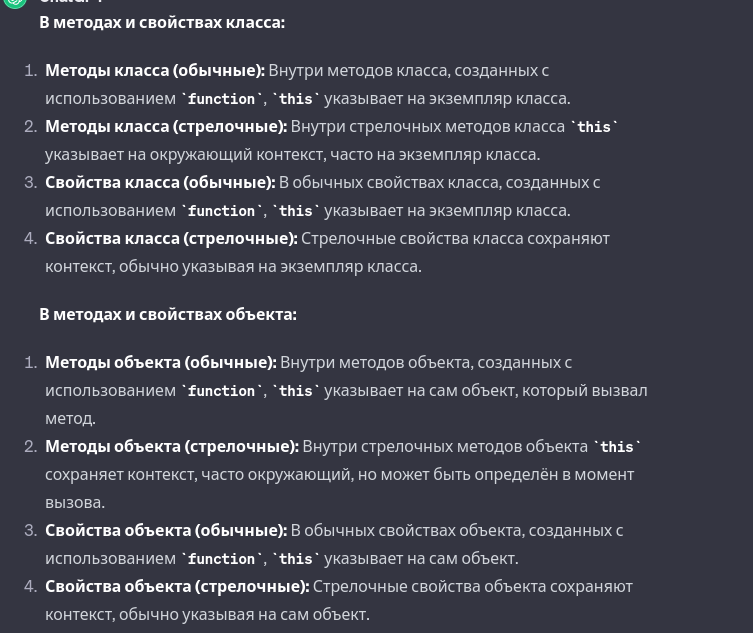
Контекст выполнения (execution context) — это, если говорить упрощённо, концепция, описывающая окружение, в котором производится выполнение кода на JavaScript. Код всегда выполняется внутри некого контекста. this ссылается на объект, в контексте которого выполняется текущий код.

**Три типа контекстов выполнения:**

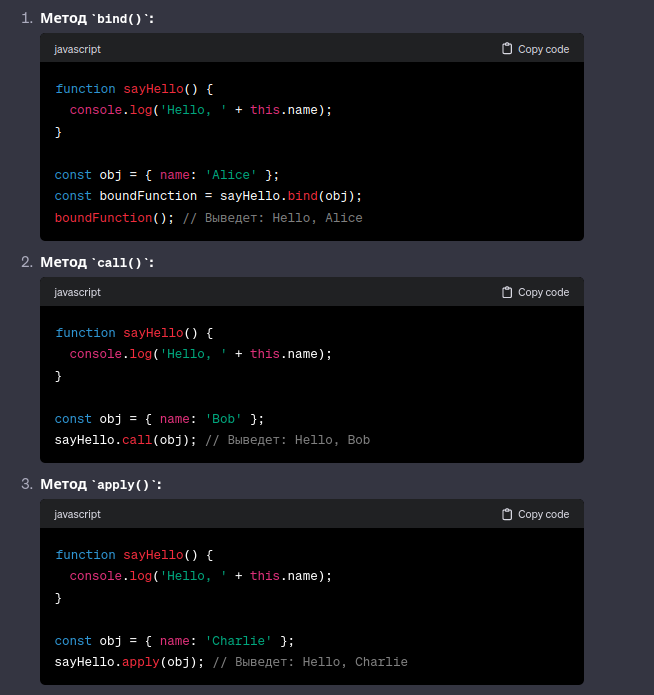
**Глобальный контекст выполнения:** Базовый контекст, используемый по умолчанию для кода вне функций. Глобальный объект, такой как window в браузере, служит контекстом, и ключевое слово this указывает на этот объект.

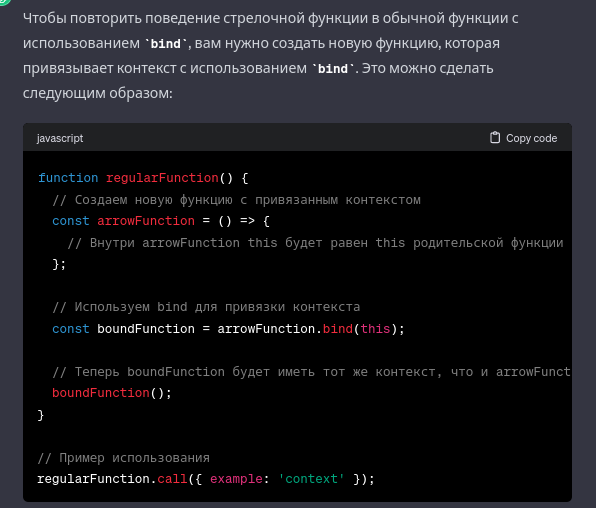
Программа имеет только один глобальный контекст.

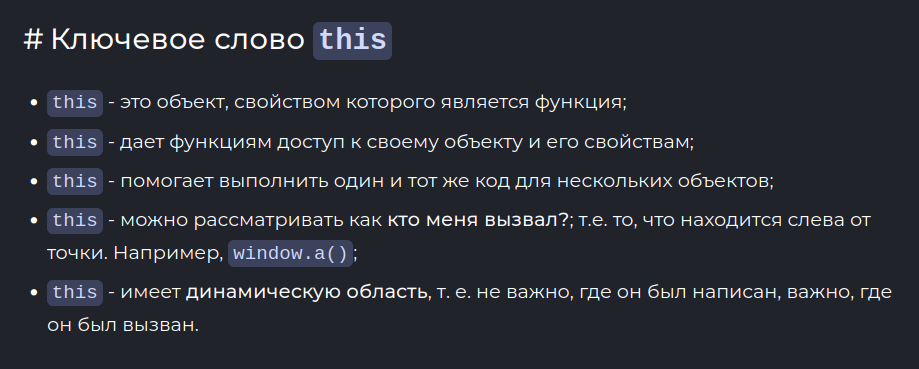
**Контекст выполнения функции:** Каждый раз при вызове функции создается новый контекст. У каждой функции свой контекст выполнения, и в программе может быть множество таких контекстов. При создании нового контекста выполнения функции выполняется определенная последовательность шагов. **Контекст выполнения функции eval:** Код, выполняемый внутри функции eval, также имеет свой контекст выполнения. Однако использование функции eval редко рекомендуется.



**Как поменять контекст функции:**







**4.** **Методы итерации (и еще генераторы, yield)**

**Итерация массивов:**

**forEach(callback)** (перебор элементов массива): Вызывает функцию обратного вызова для каждого элемента массива.

**map(callback)** (создание нового массива): Создает новый массив, в котором каждый элемент получен применением переданной функции обратного вызова.

**filter(callback)** (создание массива с фильтрацией): Возвращает новый массив, содержащий только те элементы, для которых функция обратного вызова возвращает true.

**find(callback)** (поиск первого подходящего элемента): Возвращает первый элемент, для которого функция обратного вызова возвращает true, или undefined, если ничего не найдено.

**some(callback)** (проверка наличия подходящего элемента): Возвращает true, если хотя бы один элемент соответствует условиям функции обратного вызова.

**every(callback)** (проверка всех элементов): Возвращает true, если все элементы соответствуют условиям функции обратного вызова.

**reduce(callback)** (аккумуляция значений): Применяет функцию обратного вызова к каждому элементу, сводя массив к одному значению.

**reduceRight(callback)** (аккумуляция значений в обратном порядке): Аналогичен reduce, но проходит элементы массива в обратном порядке.

**indexOf(element)** (поиск индекса элемента): Возвращает индекс первого вхождения указанного элемента в массиве.

**lastIndexOf(element)** (поиск последнего индекса элемента): Возвращает индекс последнего вхождения указанного элемента в массиве.

**includes(element)** (проверка наличия элемента): Возвращает true, если массив содержит указанный элемент, и false в противном случае.

**flat(depth)** (уплощение многомерного массива): Создает новый массив, в котором все вложенные массивы объединены в один. Глубина вложенности определяется параметром depth.

**flatMap(callback)** (преобразование и уплощение массива): Применяет функцию обратного вызова к каждому элементу массива и уплощает результат в одномерный массив.

**join(separator)** (создание строки из элементов массива): Создает строку, объединяя все элементы массива с использованием указанного разделителя.

**reverse()** (изменение порядка элементов на обратный): Изменяет порядок элементов в массиве на обратный.

**sort(compareFunction)** (сортировка элементов массива): Сортирует элементы массива с использованием указанной функции сравнения.

**splice(start, deleteCount, ...items)** (удаление и вставка элементов): Изменяет содержимое массива, удаляя существующие элементы и/или добавляя новые.

**copyWithin(target, start, end)** (копирование элементов внутри массива): Копирует часть массива в другое место в этом же массиве, не изменяя его длину.

**entries()** (получение итератора для пар ключ-значение): Возвращает новый объект итератора, который содержит пары ключ-значение для каждого элемента массива.

**keys()** (получение итератора для ключей): Возвращает новый объект итератора, который содержит ключи для каждого элемента массива.

**values()** (получение итератора для значений): Возвращает новый объект итератора, который содержит значения для каждого элемента массива.

**toLocaleString(locales, options)** (преобразование в строку с учетом локали): Возвращает строку, представляющую элементы массива, с учетом локали и опций

**Итерация объектов:**

**\*\*`Object.keys(obj)`\*\* (получение массива ключей): Возвращает массив строк, представляющих все перечисляемые свойства объекта.**

**\*\*`Object.values(obj)`\*\* (получение массива значений): Возвращает массив значений для всех перечисляемых свойств объекта.**

**\*\*`Object.entries(obj)`\*\* (получение массива пар ключ-значение): Возвращает массив, содержащий пары ключ-значение для каждого перечисляемого свойства объекта.**

**\*\*`Object.getOwnPropertyNames(obj)`\*\* (получение имен свойств): Возвращает массив строк, представляющих все свойства объекта (включая неперечисляемые).**

**\*\*`Object.getOwnPropertySymbols(obj)`\*\* (получение символьных свойств): Возвращает массив символьных ключей для всех свойств объекта (включая неперечисляемые).**

**\*\*`Object.getOwnPropertyDescriptors(obj)`\*\* (получение дескрипторов всех свойств): Возвращает объект, содержащий дескрипторы для всех свойств объекта.**

**\*\*`Reflect.ownKeys(obj)`\*\* (получение всех ключей): Возвращает массив всех ключей объекта, включая строки и символы (включая неперечисляемые).**

**\*\*`Object.hasOwnProperty(prop)`\*\* (проверка наличия свойства): Возвращает `true`, если объект содержит свойство с указанным именем.**

**\*\*`for...in`\*\* (перебор всех перечисляемых свойств): Итерация по всем перечисляемым свойствам объекта (включая свойства в прототипе).**

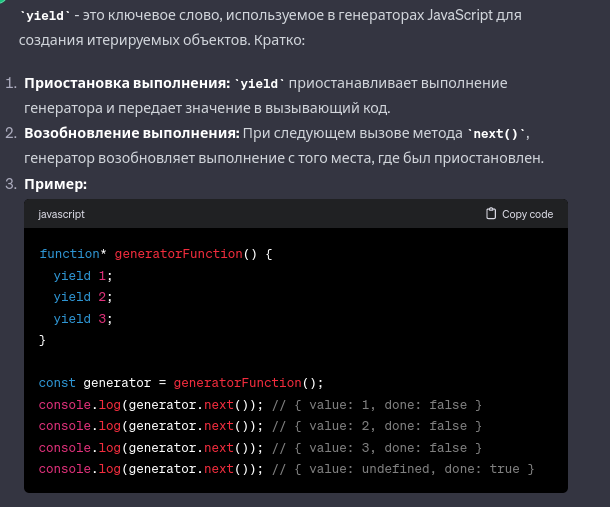
**\*\*`for...of`\*\* (перебор значений итерируемых свойств): Итерация по значениям перечисляемых свойств, таких как массивы или строки.**

**\*\*`Object.keys(obj).forEach(key => {...})`\*\* (итерация с использованием forEach): Вызывает предоставленную функцию обратного вызова для каждого элемента массива ключей объекта.**

**\*\*`Object.entries(obj).map(([key, value]) => {...})`\*\* (маппинг по парам ключ-значение): Применяет предоставленную функцию обратного вызова к каждой паре ключ-значение объекта и возвращает новый массив результатов.**

**\*\*`Object.fromEntries(entries)`\*\* (создание объекта из массива пар ключ-значение): Создает объект из массива пар ключ-значение, созданных, например, с помощью `Object.entries()`.**

**Yield(Что такое генератор?** Функция, которая когда вызвана, не выполняет свой код, а возвращает специальный объект, так называемый «генератор», для управления ее выполнением**):**



**5.** **Отличие стрелочных функций от обычных**

**Контекст this:**

**Стрелочные функции:** Не имеют своего собственного контекста this; они заимствуют его из родительского контекста, в котором были созданы.

**Обычные функции:** Имеют свой контекст this, который определяется во время вызова функции.

**Свойство arguments:**

**Стрелочные функции:** Не имеют собственного объекта arguments.

**Обычные функции:** Имеют объект arguments, представляющий переданные аргументы (arguments - это список всех переданных функции аргументов при её вызове. Объект arguments позволяет функции получить доступ к всем аргументам)

**Ключевое слово new:**

**Стрелочные функции:** Нельзя вызвать с использованием new и не имеют своего объекта prototype.

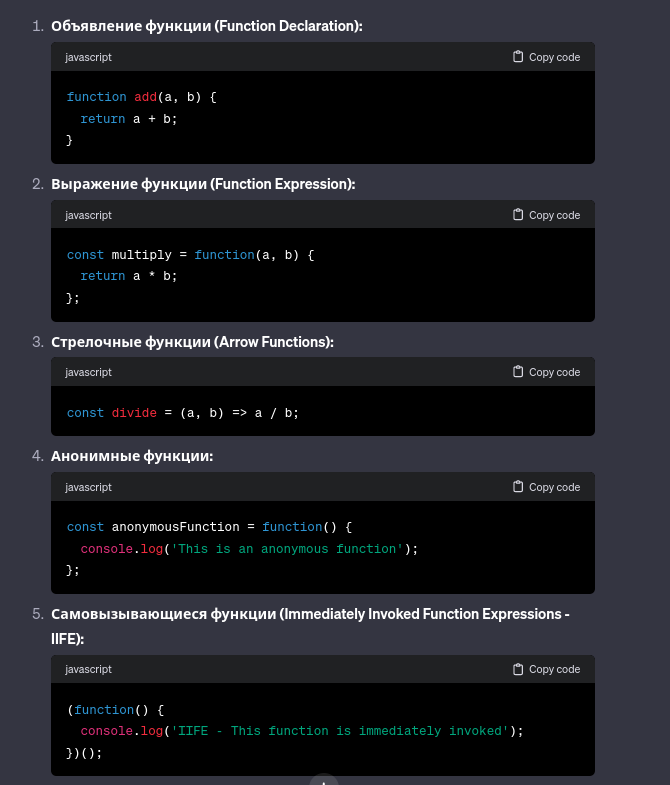
**Обычные функции:** Могут быть вызваны с использованием new для создания экземпляра объекта и имеют свой объект prototype.

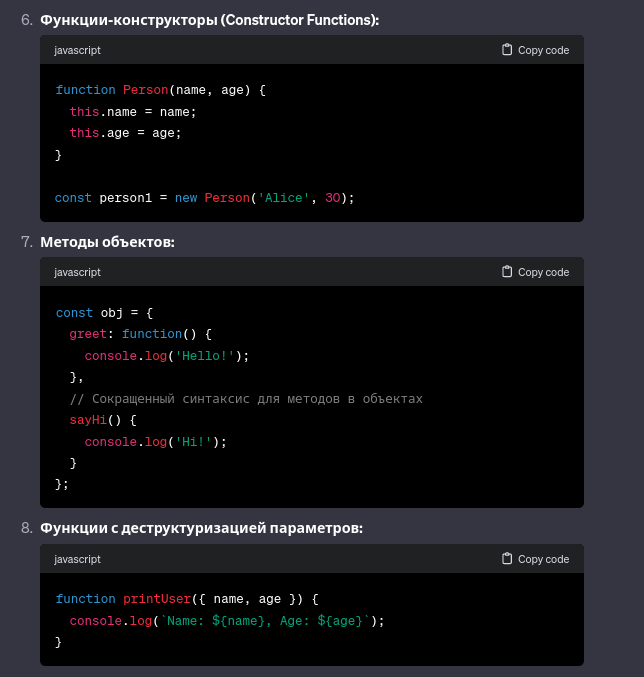
**Невозможность быть конструктором:**

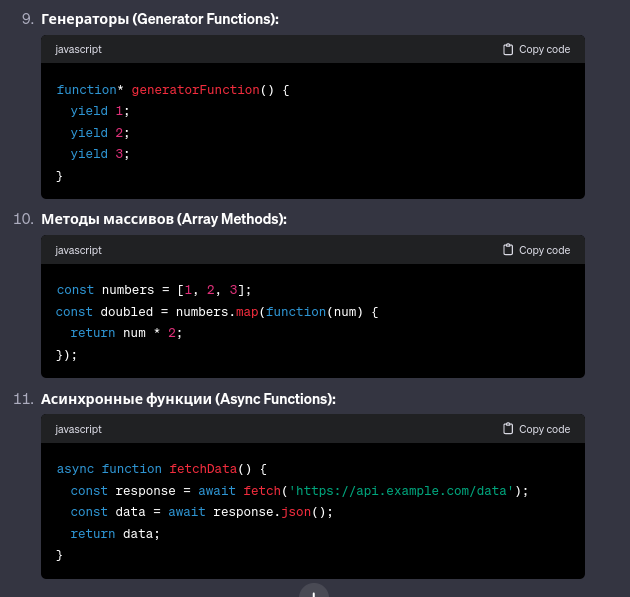
**Стрелочные функции:** Нельзя использовать для создания экземпляров объектов с использованием new.

**Обычные функции:** Могут быть использованы как конструкторы с использованием new для создания новых объектов.

6. Виды функций

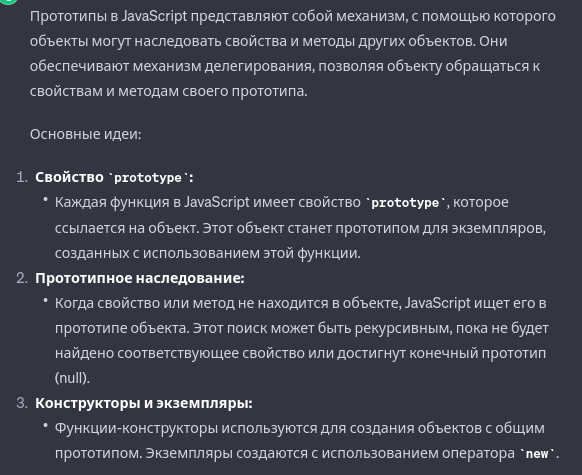




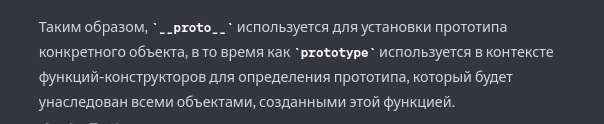


Функция высшего порядка - функция, которая получает функцию в качестве аргумента или возвращают функцию в качестве вывода. Например: Array.prototype.map, Array.prototype.filter и Array.prototype.reduce - это некоторые из функций высшего порядка, встроенных в язык.

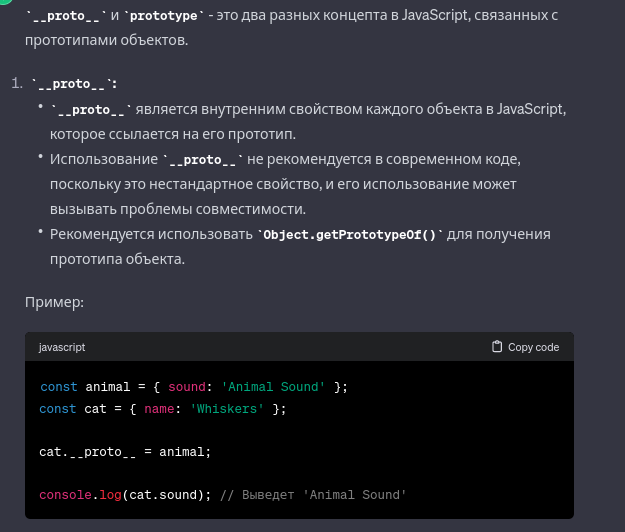
**7.** **Прототипы**



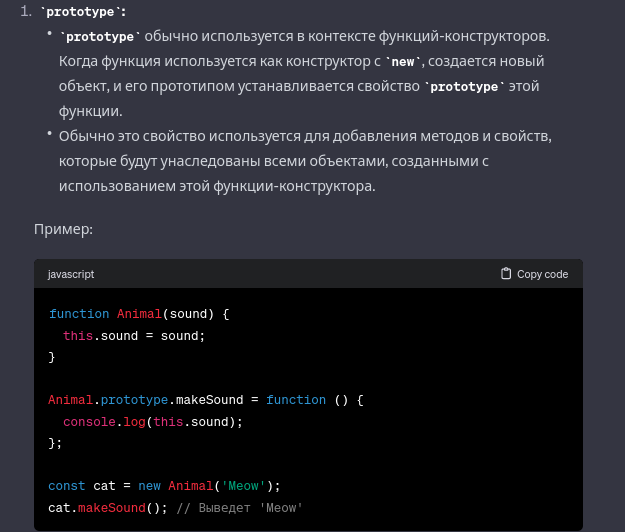


**Отличие \_\_proto\_\_ от prototype** 

**\_\_proto\_\_**



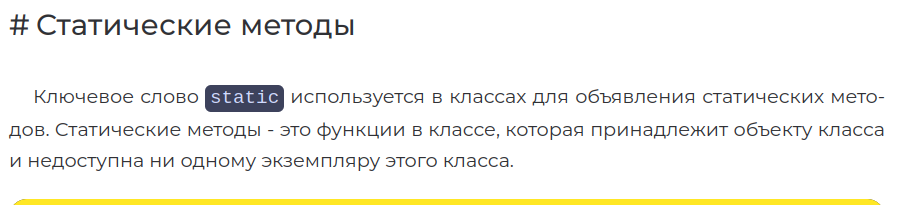
**prototype**



**8.** **Class ООП (классы в js, ооп)**

Объектно-ориентированное программирование (ООП) — это шаблон проектирования программного обеспечения, который позволяет решать задачи с точки зрения объектов и их взаимодействий.

Класс это специальная функция в js, которая создает и инициализирует объект и его свойства и вызывается с помощью new. Создание объекта - конструктор (вызывает сеттер), наследование от родительского класса - super. Как и функции, классы можно определять внутри другого выражения, передавать, возвращать, присваивать и т.д.



* Инкапсуляция - это механизм, который позволяет объединить данные и методы, работающие с этими данными, в единый объект. Внешний код может взаимодействовать с объектом через его публичный интерфейс, но не имеет доступа к его внутренней реализации. **Пример:**

class Car {

constructor(make, model) {

this.make = make; // публичное свойство

this.model = model; // публичное свойство

this.\_mileage = 0; // приватное свойство (инкапсулированное)

}

// публичный метод

drive(distance) {

this.\_mileage += distance;

console.log(`Driving ${distance} miles...`);

}

// публичный метод, который предоставляет доступ к приватному свойству \_mileage

get mileage() {

return this.\_mileage;

}

}

const myCar = new Car('Toyota', 'Corolla');

myCar.drive(100);

console.log(`Mileage: ${myCar.mileage}`); // Мы не имеем прямого доступа к \_mileage, но можем получить его значение через публичный метод get mileage()

* Полиморфизм - позволяет объектам одного класса проявлять разное поведение на основе их типа или контекста. Это достигается через переопределение методов в наследниках. **Пример:**

class Shape {

getArea() {

return 0;

}

}

class Circle extends Shape {

constructor(radius) {

super();

this.radius = radius;

}

getArea() {

return Math.PI \* this.radius \* this.radius;

}

}

class Square extends Shape {

constructor(sideLength) {

super();

this.sideLength = sideLength;

}

getArea() {

return this.sideLength \* this.sideLength;

}

}

function printArea(shape) {

console.log(`Area: ${shape.getArea()}`);

}

const circle = new Circle(5);

const square = new Square(4);

printArea(circle); // Выводит "Area: 78.53981633974483"

printArea(square); // Выводит "Area: 16"

* Наследование - позволяет создавать новые классы на основе существующих, наследуя их свойства и методы. Это способствует повторному использованию кода и обеспечивает иерархическую структуру классов. Наследование классов - с помощью слова extends, класс потомка наследует конструктор от родительского, но если нужно что то в нем изменить, то он вызывается с помощью super(). **Пример:**

class Animal {

constructor(name) {

this.name = name;

}

speak() {

console.log(`${this.name} makes a sound.`);

}

}

class Dog extends Animal {

constructor(name, breed) {

super(name);

this.breed = breed;

}

// Переопределение метода speak() для класса Dog

speak() {

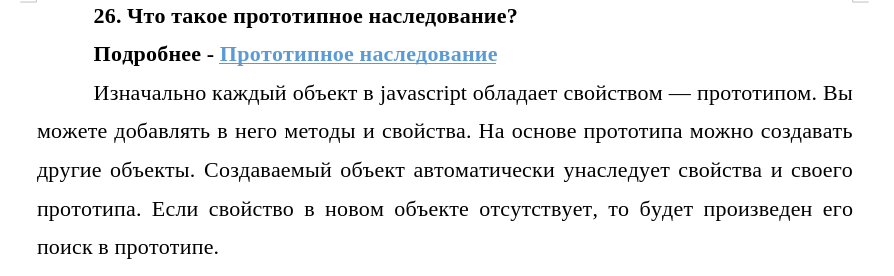
console.log(`${this.name} barks loudly.`);

}

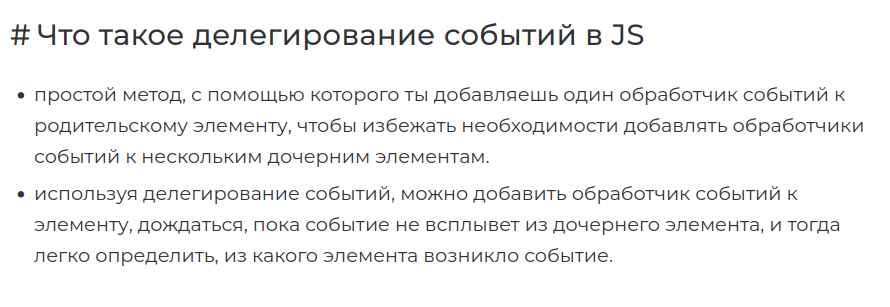
}

const dog = new Dog('Buddy', 'Golden Retriever');

dog.speak(); // Выводит "Buddy barks loudly."



**9.** **Что такое события в JavaScript и какие события знаете?** События в JavaScript - это действия, которые происходят в браузере или веб-приложении и на которые можно реагировать. Некоторые примеры событий: клик мыши (click), загрузка страницы (load), отправка формы (submit), изменение значения (change) и другие.



**10.** **Как обрабатывать события?** События могут быть обработаны с помощью обработчиков событий, которые являются функциями, вызываемыми при наступлении события. В JavaScript это может быть достигнуто через свойства элемента DOM, методы addEventListener() или через атрибуты HTML-элементов.

**11.** **Какие методы строк знаете?** Некоторые методы строк в JavaScript: length (длина строки), charAt() (возвращает символ по указанному индексу), concat() (объединяет строки), toLowerCase() (преобразует строку в нижний регистр), toUpperCase() (преобразует строку в верхний регистр), slice() (извлекает часть строки) и другие.

**12.** **Что такое шаблонная строка?** Шаблонная строка (template string) в JavaScript - это строка, которая позволяет встраивать выражения и переменные, заключая их в фигурные скобки ${}. Также шаблонные строки поддерживают многострочный текст и использование выражений внутри строки.

**13.** **Какие методы массива знаете?** Некоторые методы массива в JavaScript: push() (добавляет элементы в конец массива), pop() (удаляет последний элемент массива), shift() (удаляет первый элемент массива), unshift() (добавляет элементы в начало массива), concat() (объединяет массивы), slice() (извлекает часть массива) и другие.

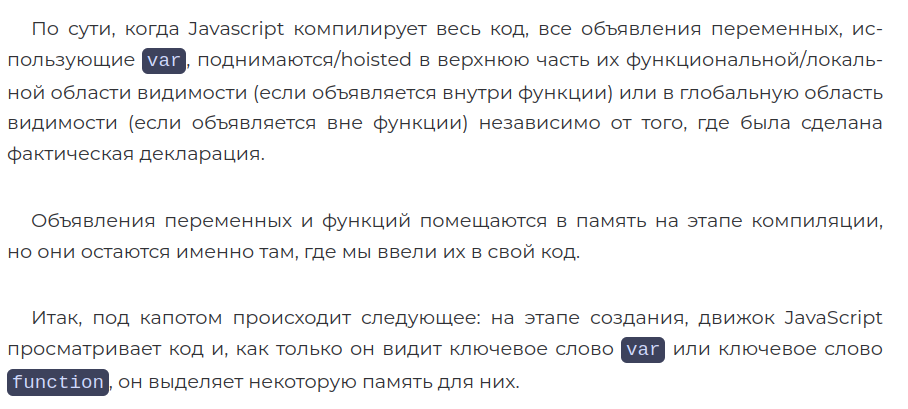
**14.** **Как объединить два массива?** Два массива могут быть объединены с помощью метода concat() или оператора расширения ([...array1, ...array2]).

**15.** **Как отсортировать массив?** Массив можно отсортировать с помощью метода sort(), который сортирует элементы массива по возрастанию. Можно также передать функцию сравнения в sort(), чтобы определить свой порядок сортировки.

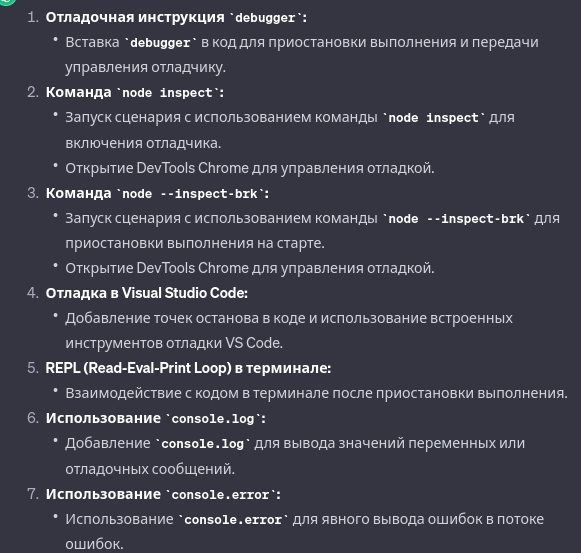
**16.** **Что такое преобразование типов?** Преобразование типов в JavaScript - это изменение типа значения из одного типа в другой. Преобразование может быть неявным (автоматическим) или явным (принудительным).

**17.** **Как работает обработка ошибок в JavaScript?** Обработка ошибок в JavaScript осуществляется с помощью блоков try, catch и finally. Код, который может вызвать ошибку, помещается в блок try, а обработчик ошибок размещается в блоке catch. Блок finally используется для выполнения кода, который должен быть выполнен независимо от наличия ошибки.

**18.** **Что такое hoisting (подъем)?** Hoisting (поднятие) в JavaScript - это механизм, который позволяет переменным и функциям быть доступными до момента фактического исполнения кода, называемого поднятием переменных и функций.



**19.** **Как вы дебажите свой код?**



**20.** **Что такое асинхронность и как работает в JavaScript?** Асинхронность в JavaScript позволяет выполнять операции, которые могут занимать время, без блокировки выполнения других операций. Это достигается с помощью асинхронных функций, колбэков, промисов и асинхронного кода.

**21.** **Что такое Promise и как им пользоваться?** Promise - это объект, представляющий результат асинхронной операции, которая может быть выполнена или отклонена. Можно использовать методы then() и catch() для обработки успешного выполнения или ошибки соответственно.

**22.** **Какие методы Promise знаете?**

1. \*\*`then`:\*\*

- Метод, вызываемый на промисе, который обрабатывает успешное завершение промиса.

2. \*\*`catch`:\*\*

- Метод, вызываемый на промисе, который обрабатывает ошибку в промисе.

3. \*\*`finally`:\*\*

- Метод, вызываемый на промисе, который выполняется независимо от успешного завершения или ошибки.

4. \*\*`Promise.resolve`:\*\*

- Создает успешно завершенный промис с переданным значением.

5. \*\*`Promise.reject`:\*\*

- Создает промис, завершающийся ошибкой с переданным значением.

6. \*\*`Promise.all`:\*\*

- Принимает массив промисов и возвращает новый промис, завершающийся успешно, если все переданные промисы завершены успешно.

7. \*\*`Promise.race`:\*\*

- Принимает массив промисов и возвращает новый промис, завершающийся результатом первого завершившегося промиса из массива.

8. \*\*`Promise.allSettled`:\*\*

- Принимает массив промисов и возвращает новый промис, завершающийся после завершения всех промисов в массиве, независимо от их успешного завершения или ошибки.

9. \*\*`Promise.any`:\*\*

- Принимает массив промисов и возвращает новый промис, завершающийся результатом первого успешно завершившегося промиса из массива.

10. \*\*`Promise.prototype.catch`:\*\*

- Альтернативная форма `catch` для обработки ошибок на экземпляре промиса.

11. \*\*`Promise.prototype.finally`:\*\*

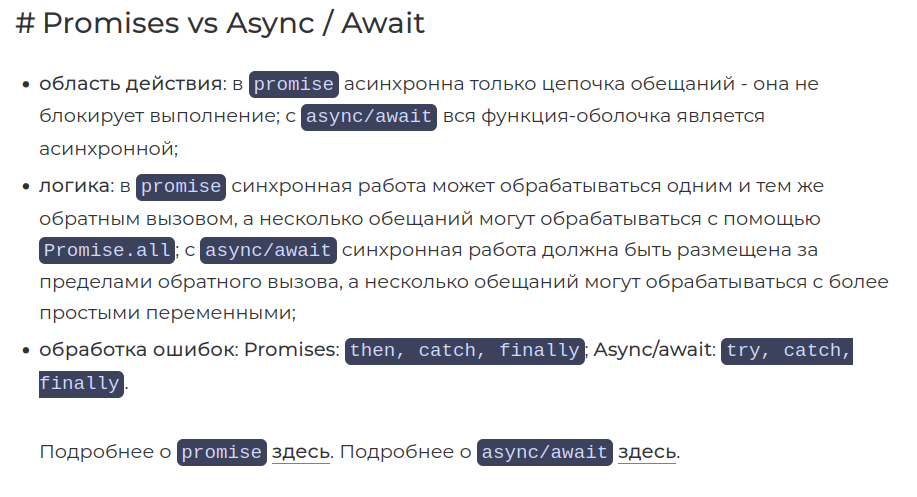
- Альтернативная форма `finally` для обработки завершения промиса на экземпляре.

12. \*\*`Promise.prototype.then`:\*\*

- Альтернативная форма `then` для обработки успешного завершения промиса на экземпляре.

**23.** **Что такое async/await?** async/await - это синтаксический сахар в JavaScript для работы с асинхронным кодом. Ключевое слово async используется для объявления асинхронной функции, а ключевое слово await используется внутри асинхронной функции для приостановки выполнения и ожидания результата промиса. Всегда возвращает промис. Даже если внутри такой функции есть явное возвращаемое значение, оно будет обернуто в промис.

**24.** **Разница promise и async/await (promise vs async/await)**



**25.** **Что такое обратные вызовы(callback) и замыкание**

* обратный вызов (callbacks): функция, которая доступная для другой функции и

вызываемая после первой функции - если эта первая функция завершилась.

* замыкание (closures): создаются всякий раз, когда к переменной, определенной вне текущей области, осуществляется доступ из внутренней области - это дает нам доступ к области внешней функции из внутренней функции;
* чтобы использовать замыкание, просто определи функцию внутри другой функции;
* Замыкание – это функция, которая запоминает свои внешние переменные и может получить к ним доступ. Оно позволяет функции сохранять доступ к переменным из своего внешнего лексического окружения, даже после того, как это окружение закрыто или функция вышла из своей области видимости. Когда функция в JavaScript создается, она сохраняет ссылку на свое лексическое окружение, включая все переменные и параметры функции, а также внешние функции и переменные, доступные в месте ее объявления. **Пример:**

function createCounter() {

let count = 0;

return function() {

count++;

console.log(count);

};

}

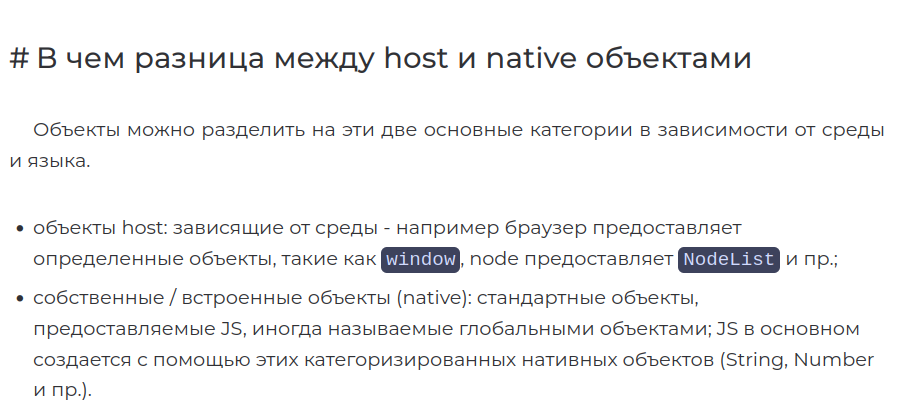
const counter = createCounter();

counter(); // 1

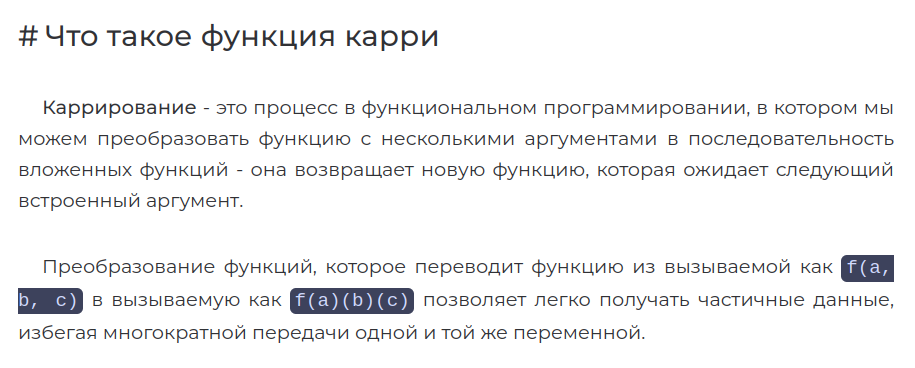
counter(); // 2

counter(); // 3

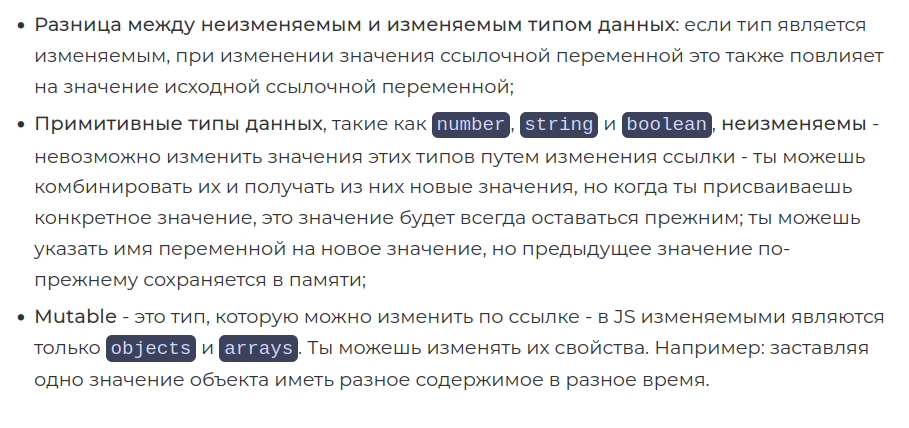
26. Host и native объекты (хост и нейтив)



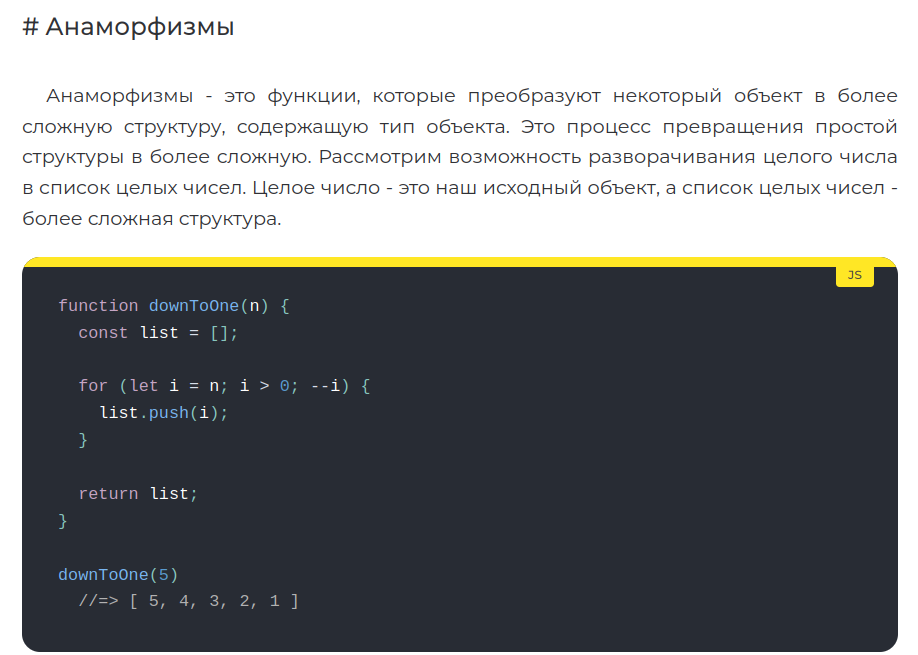
27. Каррирование(функция карри)



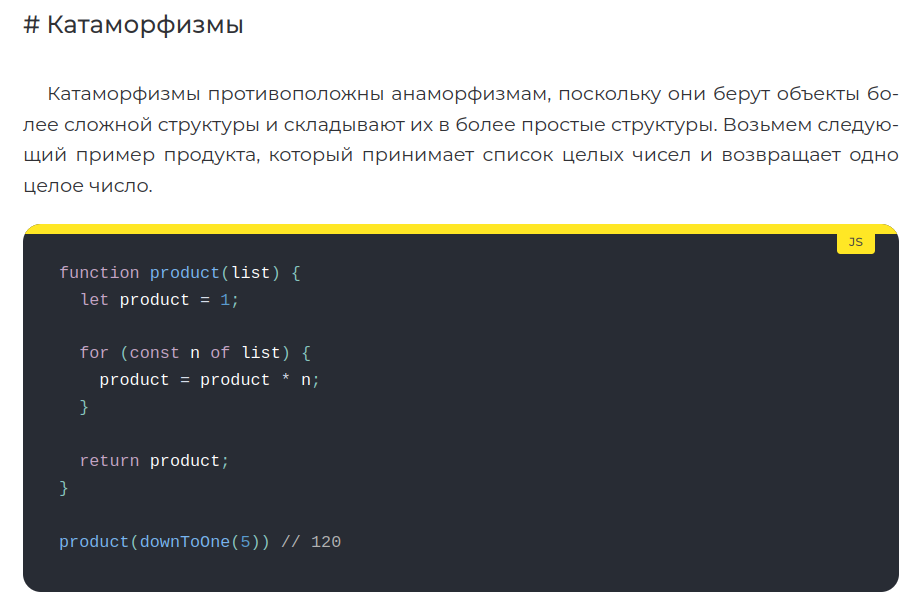
28. Mutable и Immutable



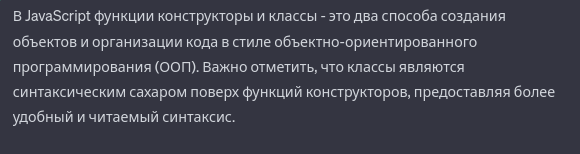
29. Анаморфизмы

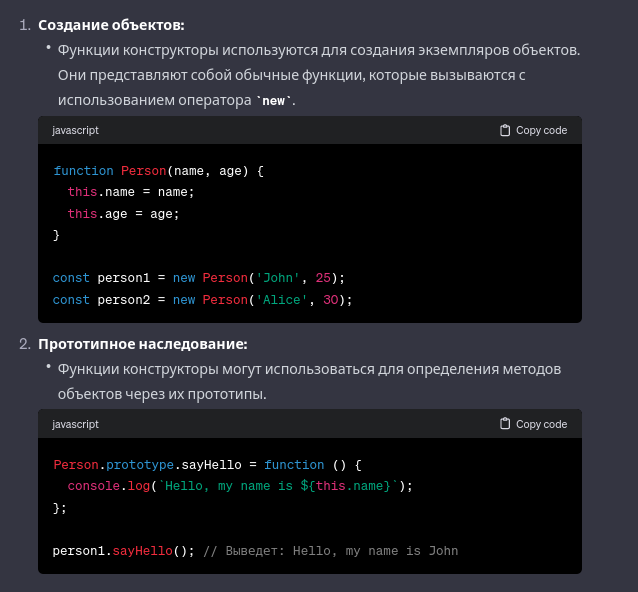


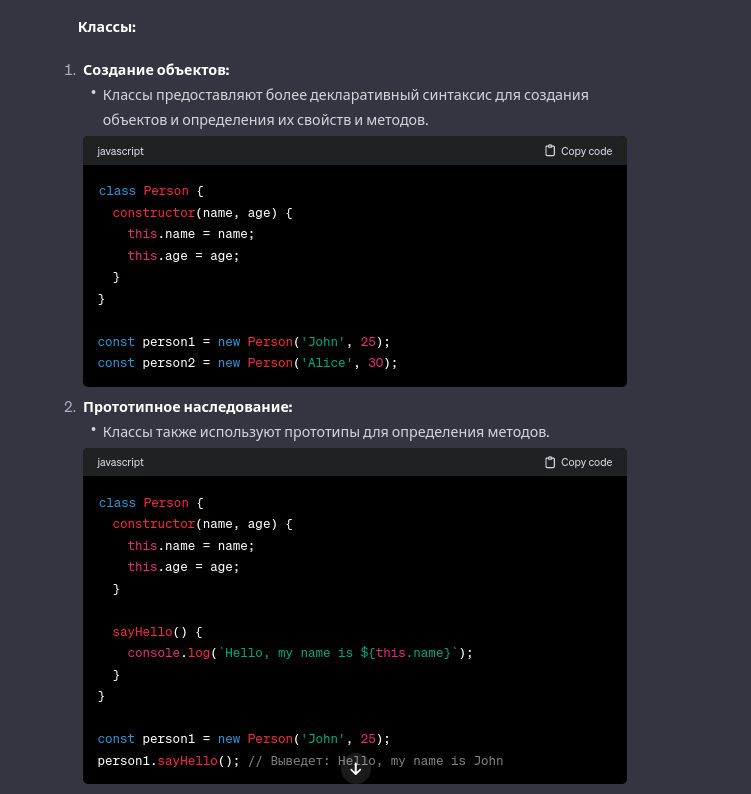
30. Катаморфизмы

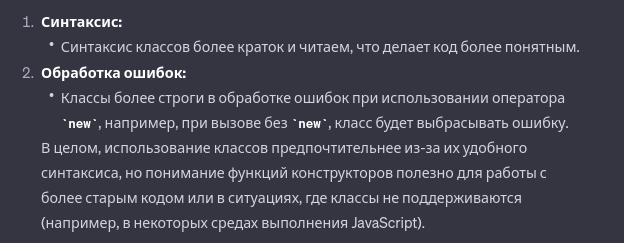


**31.** **Функции-контсрукторы и их отличие от классов**

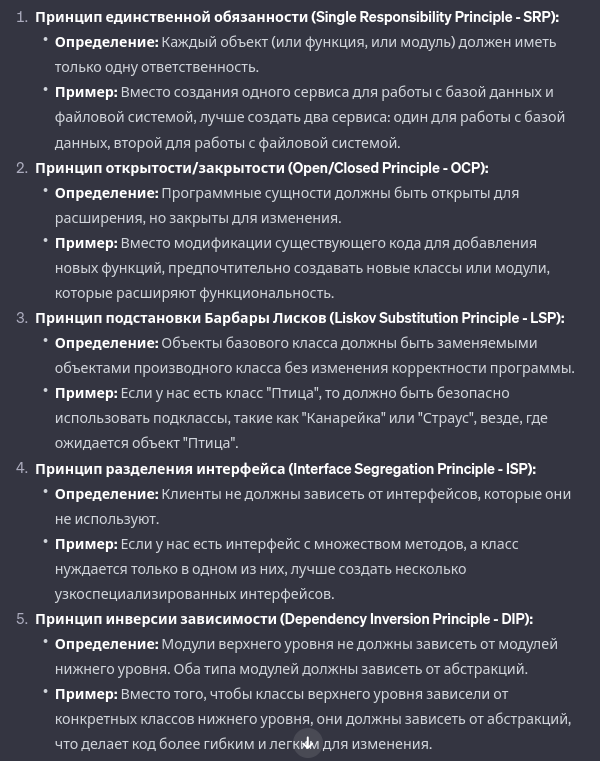








**32.** **SOLID**



**DRY (Don't Repeat Yourself)** Этот принцип подразумевает, что в программе не должно быть повторяющегося кода. Лучше использовать абстракции, функции или классы для устранения дублирования кода. Это способствует улучшению читаемости, поддерживаемости и повторного использования кода.

**YAGNI (You Ain't Gonna Need It)** Этот принцип заключается в том, что не следует включать функциональность в программу, которая в данный момент не требуется. Избегайте добавления ненужного кода, который может усложнить разработку и поддержку программы. Разрабатывайте только то, что вам действительно нужно сейчас.

**KISS (Keep It Simple, Stupid)** Этот принцип призывает к созданию простого и понятного кода. Избегайте излишней сложности, используйте простые решения, которые легко понять и поддерживать. Простота кода способствует его читаемости, отладке и расширению.

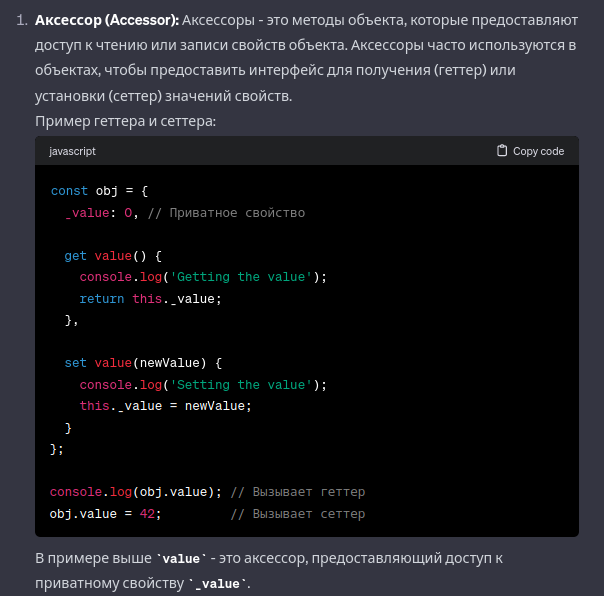
**33.** **Области видимости(Какие есть области видимости?)**

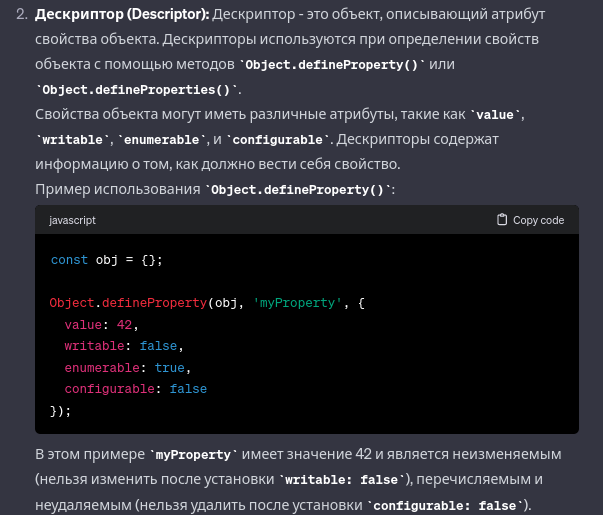
* Глобальная область видимости: Переменные, объявленные в глобальной области видимости, доступны из любой части кода. Они являются глобальными и видимыми для всех функций и блоков.
* Локальная область видимости функции: Переменные, объявленные внутри функции, имеют локальную область видимости. Они доступны только внутри этой функции и не видимы вне ее.
* Блочная область видимости (с введением let и const): Переменные, объявленные с помощью let и const, имеют блочную область видимости. Они видны только внутри блока, в котором они были объявлены (например, внутри условий if или циклов), и не видимы за пределами этого блока.
* Лексическая область видимости: Лексическая область видимости означает, что область видимости переменных определяется структурой кода и его физическим расположением внутри иерархии функций и блоков. Вложенные функции имеют доступ к переменным внешних функций.

**34.** **Что такое IIFE?** Функция, которая выполняется сразу же после того, как она была определена.

**35.** **Что такое боксинг и анбоксинг?** Оборачивание примитивного типа в объект при применении к нему каких либо действий, и возврат обратно к примитивному типу после завершения операции.

36. **Что такое аксессор и что такое дескриптор?**





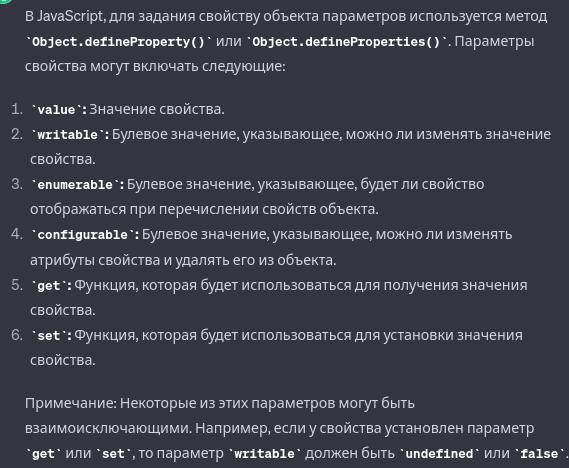
**37.** **Есть ли length у объекта?** Нет.

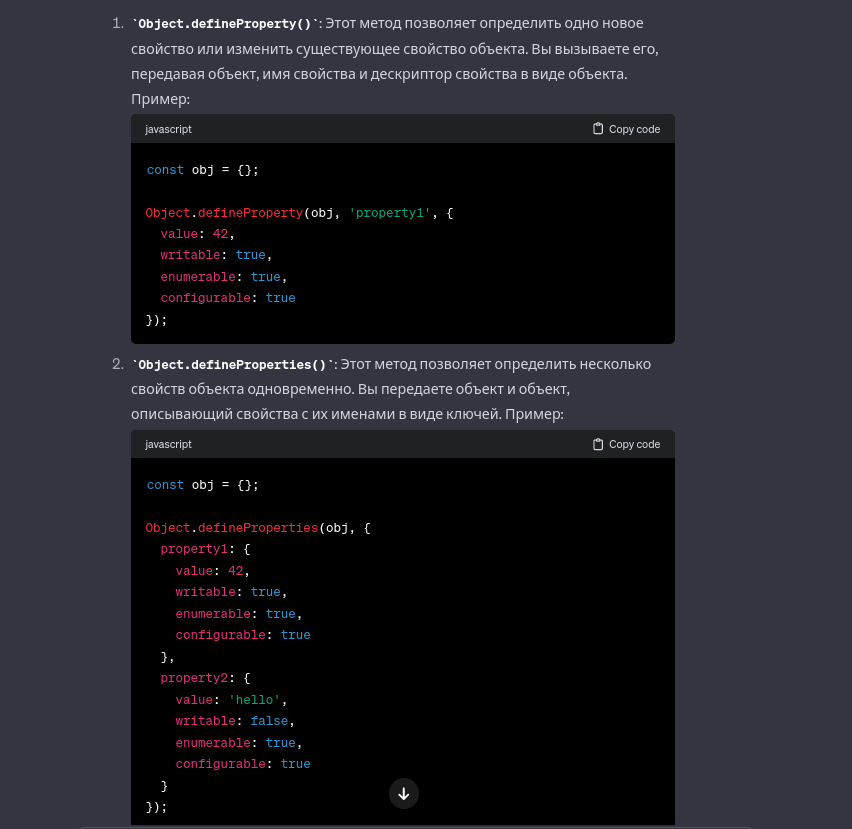
**38.** **Какие методы массивов мутируют исходный массив?** splice, pop, push, shift, unshift, sort

**39.** **Как можно скопировать объект?**

* Object.assign(), spread оператор
* JSON.stringify -> JSON.parse
* deepClone (Lodash)

40. **Как можно задать объекту свойство и параметры свойства. Какие бывают параметры свойства?**





41. **В чем плюсы и минусы иммутабельности?** Неизменяемым (англ. immutable) называется объект, состояние которого не может быть изменено после создания. Результатом любой модификации такого объекта всегда будет новый объект, при этом старый объект не изменится.

**Плюсы:**

* + Простое и быстрое отслеживание изменений (можно сравнивать не поля объектов, а ссылки на них).
  + Безопаснее использовать и легче тестировать.

**Минусы:**

* + Использование доп. памяти.
  + При использовании сторонних библиотек для иммутабельности возможно ухудшение производительности.

42. **Зачем нужен useStrict?**

Более строгий режим, по умолчанию в классах и модулях. вводит ограничения:

* + Неопределенной переменной не присваивается значение.
  + Нельзя применить инструкцию «with».
  + Нет возможности добавить повторяющиеся свойства в литерале объекта.
  + Нет возможности добавить дополнительные параметры формальной функции.
  + Когда изменяется объект «arguments», не изменяются аргументы.
  + Когда аргумент является неизменным свойством объекта, то «delete» выдаст ошибку.
  + Нет возможности преобразовать «this» в объект.
  + Увеличенное количество слов, которые зарезервировали для будущего применения.
  + Нет возможности применять конструкции кода, которые могут затруднить оптимизацию самого кода.
  + Нельзя удалить обычные переменные.

**44.** **Как можно сравнить два объекта в js?**

* + Строгое равенство (===): Оператор строгого равенства === сравнивает объекты на идентичность, то есть проверяет, ссылаются ли они на один и тот же объект в памяти.
  + Глубокое сравнение: Глубокое сравнение (deep comparison) позволяет сравнивать значения свойств объектов рекурсивно, проверяя их равенство. Это может быть реализовано с помощью цикла, рекурсии или встроенных функций, таких как JSON.stringify() и JSON.parse().
  + Также существуют различные библиотеки, такие как Lodash или DeepEqual.

**45.** **Возможно ли узнать сколько аргументов принимает функция?** Свойство length определяет количество аргументов, ожидаемых функцией.

**46.** **Как можно удалить произвольный элемент массива?**

let array = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10];

let random = Math.floor(Math.random() \* array.length);

array.splice(random, 1);

**47.** **В чём различия между коллекциями Map и WeakMap? Есть ли у Set какой-нибудь аналог?**

Map и WeakMap - это две разные коллекции в JavaScript с различными свойствами и использованием:

* + **Map:**
    1. Map - это упорядоченная коллекция, которая хранит данные в виде пар ключ-значение.
    2. Ключи в Map могут быть любого типа данных, включая объекты или примитивы.
    3. Map поддерживает итерацию и предоставляет методы для добавления, получения, обновления и удаления элементов.
    4. Map сохраняет порядок вставки элементов, что означает, что они могут быть итерированы в том же порядке, в котором были добавлены.
    5. Map обеспечивает возможность быстрого поиска элементов по ключу и предоставляет методы для работы с коллекцией.
  + **WeakMap:**
    1. WeakMap - это специальный тип Map, предназначенный для использования только с объектами в качестве ключей.
    2. WeakMap не поддерживает итерацию, методы size или методы для получения всех ключей или значений.
  + **Set:**
    1. Set - это коллекция уникальных значений любого типа данных, которые могут быть итерируемыми.
    2. В Set каждое значение может появляться только один раз. Дубликаты не разрешены.
    3. Set сохраняет порядок вставки элементов, поэтому элементы могут быть итерированы в том же порядке, в котором были добавлены.
    4. Set предоставляет методы для добавления, удаления, проверки наличия и итерации элементов.

Аналогом Set для ключей является WeakSet, который аналогичен WeakMap по своей природе. WeakSet также хранит только объекты и имеет ограниченные возможности, но не сохраняет порядок вставки элементов.

**48.** **Какие способы работы с асинхронным кодом ты знаешь?**

* Callbacks - это функции, которые передаются в асинхронные операции в качестве аргументов и вызываются по завершении операции. Callbacks позволяют выполнить код после завершения асинхронной операции и получить результаты. Однако, использование множества колбэков может привести к появлению так называемого "callback hell" и усложнению понимания и поддержки кода.
* Promises представляют асинхронные операции, которые могут быть в состояниях ожидания (pending), выполнения (fulfilled) или отклонения (rejected).
* Async/await: Асинхронные функции и оператор await предоставляют более декларативный синтаксис для работы с асинхронным кодом. async перед объявлением функции указывает, что функция содержит асинхронный код, а await используется для ожидания выполнения асинхронной операции. Async/await делает код более линейным и упрощает чтение и написание асинхронного кода.

**49.** **В чём разница между promise.allsettled и promise.all?**

* Promise.all отклонит, как только будет отклонено одно из обещаний в массиве.
* Promise.allSettled никогда не отклонит - он разрешится, как только все обещания в массиве будут либо отклонены, либо разрешены. У них разные значения решимости.

**50.** **Ecmascript(ES)**

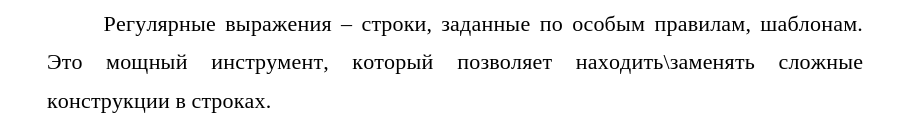
**ECMAScript (ES) — это стандарт, на основе которого создаются языки программирования JavaScript, JScript и ActionScript. В основном, когда мы говорим о новых возможностях и обновлениях в JavaScript, мы имеем в виду изменения в стандарте ECMAScript.**

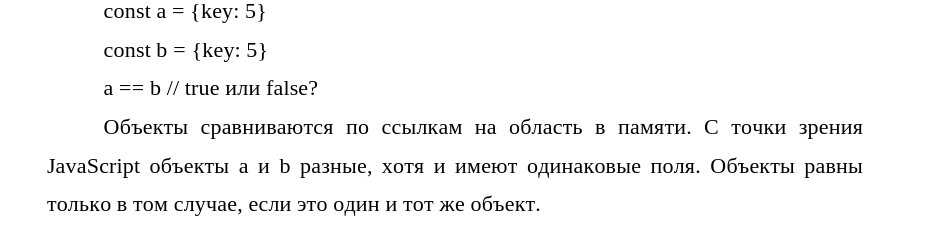
Вот несколько версий ECMAScript и их ключевые особенности:

1. **ES5 (ECMAScript 5):**
   * Год выпуска: 2009.
   * Важные функции: Внедрение "строгого режима" ("use strict"), методы массивов (forEach, map, filter, reduce, и т. д.).
2. **ES6 (ECMAScript 2015):**
   * Год выпуска: 2015.
   * Важные функции: Лямбда-выражения (стрелочные функции), let и const для объявления переменных, деструктуризация, классы, строгое равенство (=== и !==), шаблонные строки, итераторы, генераторы и другие улучшения.
3. **ES2016 (ES7):**
   * Год выпуска: 2016.
   * Важные функции: Метод Array.prototype.includes, оператор возведения в степень (\*\*).
4. **ES2017 (ES8):**
   * Год выпуска: 2017.
   * Важные функции: async/await для работы с асинхронным кодом, объект Object.values и Object.entries, метод String.prototype.padStart и String.prototype.padEnd.
5. **ES2018 (ES9):**
   * Год выпуска: 2018.
   * Важные функции: Рест-параметры в объектах, обещания (Promises) получили метод finally, Object.entries, Object.values, и Object.getOwnPropertyDescriptors.
6. **ES2019 (ES10):**
   * Год выпуска: 2019.
   * Важные функции: Array.prototype.flat и Array.prototype.flatMap, динамический import(), оператор try {...} catch {...} finally {...} без переменной ошибки.
7. **ES2020 (ES11):**
   * Год выпуска: 2020.
   * Важные функции: BigInt для представления больших чисел, оператор nullish coalescing (??), оператор опциональной последовательности (?.), методы String.prototype.matchAll, import.meta.
8. **ES2021 (ES12):**
   * Год выпуска: 2021.
   * Важные функции: String.prototype.replaceAll, логическое присваивание (||= и &&=), метод Promise.any, числовые разделители (1\_000).

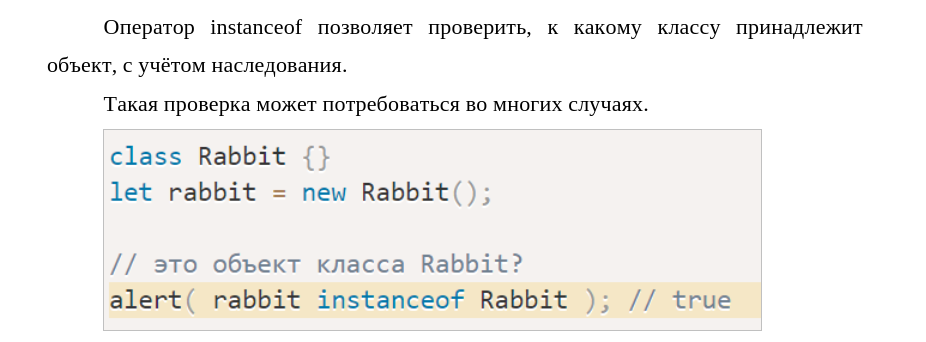
Эти обновления предоставляют новые возможности, улучшают синтаксис, упрощают разработку и делают язык более мощным и удобным для программистов.

**51.** **Что представляют из себя регулярные выражения и для чего они нужны?**

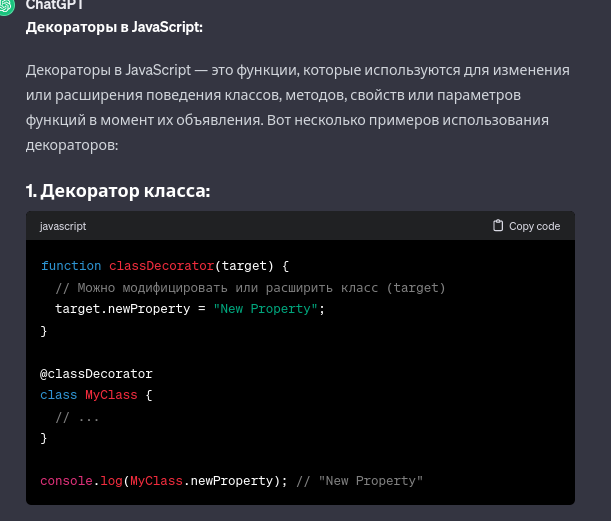
**52. Почему два объекта с одинаковыми полями при сравнении д****ают false****?**

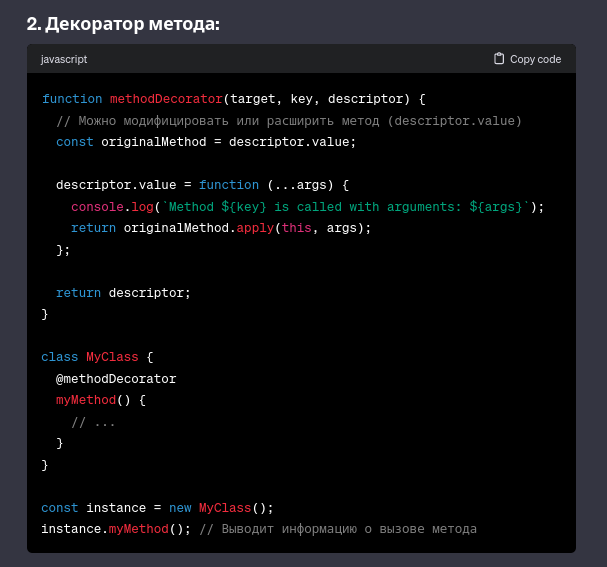
****

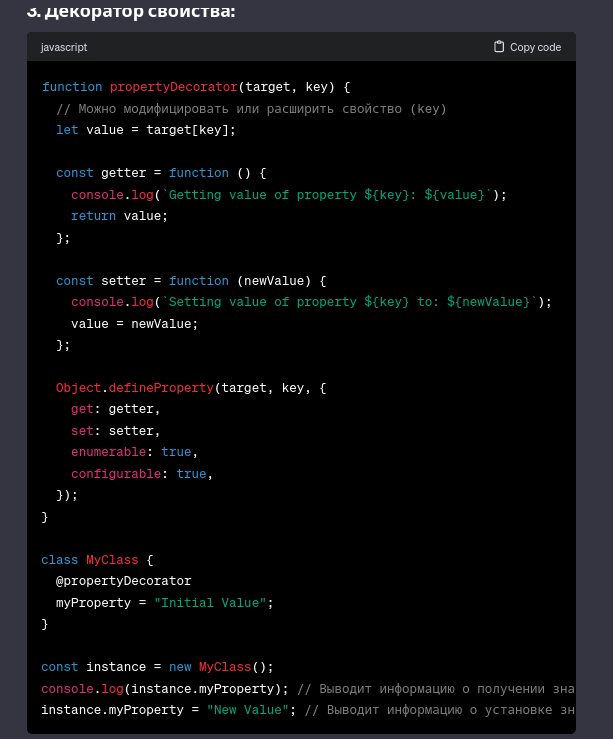
**53. Как проверить из какого класса был** **создан объект?**

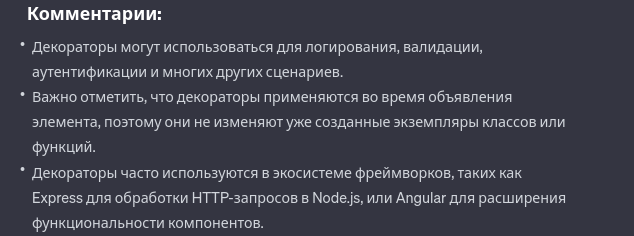
****

**54.** **Декораторы в js**

****

****

****

****

**Typescript**

**1.** **Типы данных ts**

В TypeScript существует несколько основных типов данных, каждый из которых имеет свои особенности. Вот краткое описание основных типов:

1. \*\*number:\*\* Числовой тип данных. Может быть целым или десятичным числом. (Пример: `let age: number = 25;`)

2. \*\*string:\*\* Строковый тип данных. Представляет текстовую информацию. (Пример: `let name: string = "John";`)

3. \*\*boolean:\*\* Логический тип данных. Принимает значения `true` или `false`. (Пример: `let isValid: boolean = true;`)

4. \*\*array:\*\* Массив. Позволяет хранить упорядоченный набор элементов одного типа или комбинацию типов. (Пример: `let numbers: number[] = [1, 2, 3];`)

5. \*\*tuple:\*\* Кортеж. Позволяет определить массив фиксированной длины с элементами разных типов. (Пример: `let person: [string, number] = ["John", 25];`)

6. \*\*enum:\*\* Перечисление. Позволяет создавать именованные наборы числовых значений. (Пример: `enum Color { Red, Green, Blue };`)

7. \*\*any:\*\* Динамический тип данных. Позволяет использовать переменные без явного определения их типа. (Пример: `let data: any = "Hello";`)

8. \*\*void:\*\* Тип, используемый для указания, что функция не возвращает значение. (Пример: `function logMessage(): void { console.log("Message"); }`)

9. \*\*null и undefined:\*\* Представляют отсутствие значения. Обычно используются как подтипы для других типов. (Пример: `let value: null = null;`)

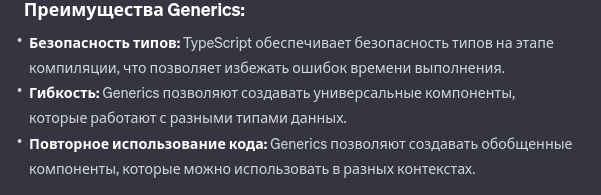
10. \*\*never:\*\* Тип, представляющий значения, которые никогда не произойдут. Обычно используется для функций, которые генерируют ошибки или возвращают бесконечный цикл. (Пример: `function throwError(message: string): never { throw new Error(message); }`)

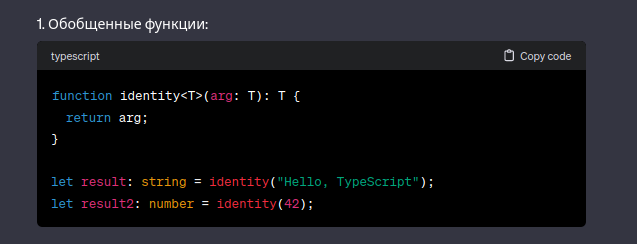
11. \*\*object:\*\* Тип данных для объектов (не примитивных типов). (Пример: `let user: { name: string, age: number } = { name: "John", age: 25 };`)

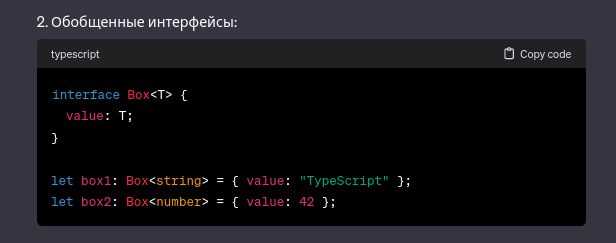
12. \*\*union:\*\* Объединение типов. Позволяет переменной принимать значения двух или более типов. (Пример: `let result: number | string = 10;`)

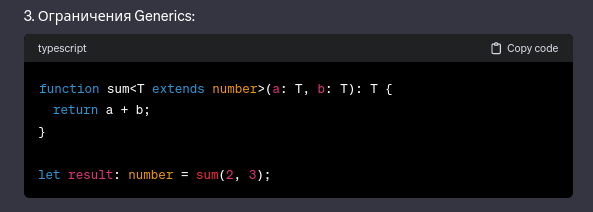
**2.** **Generics(дженерики)**

Generics в TypeScript представляют собой механизм для создания обобщенных типов и компонентов (функций, классов, интерфейсов), которые способны оперировать с различными типами данных. Это позволяет сохранять информацию о типах, делая код более гибким и повторно используемым. Вместо жестко заданных типов, мы создаем обобщенный каркас, который пользователь дженерика заполняет конкретными типами по необходимости. Generics позволяют описывать общую структуру, предоставляя пользователям возможность определения конкретных типов данных при использовании этих компонентов.

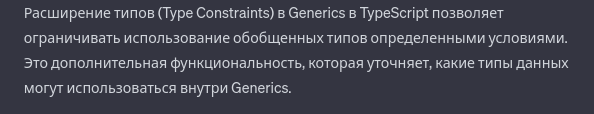
****

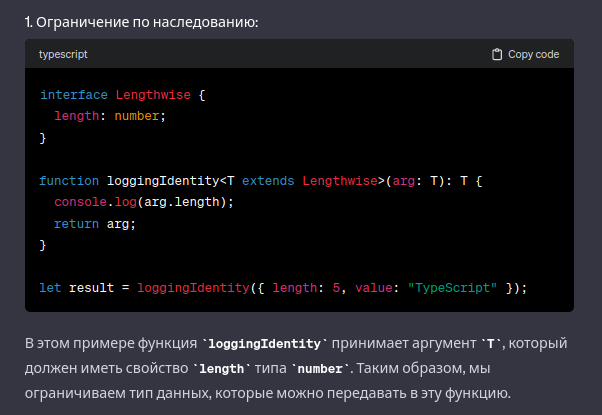
****

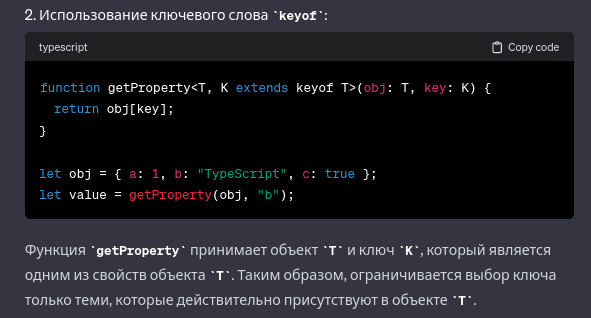
****

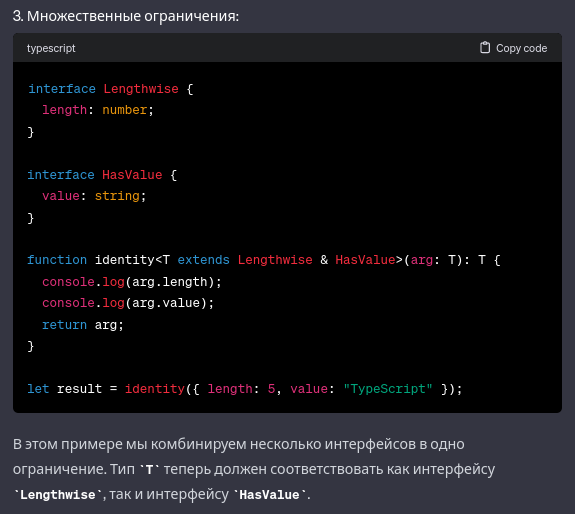
****

**Расширение типов**

****

****

****

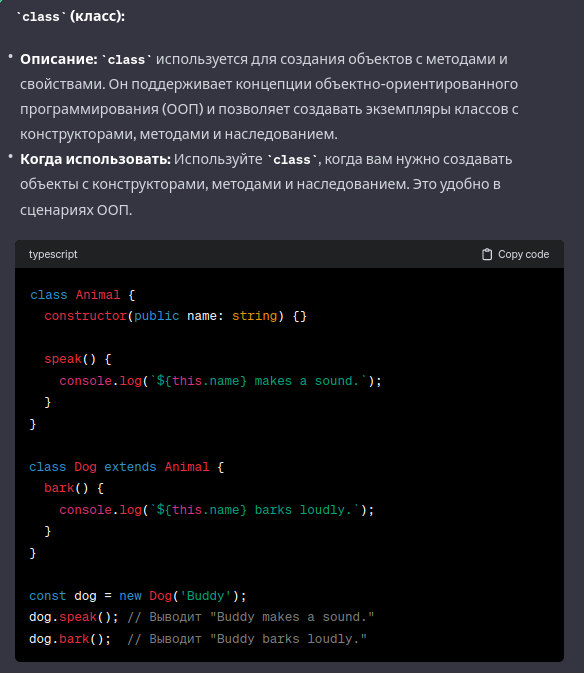
****

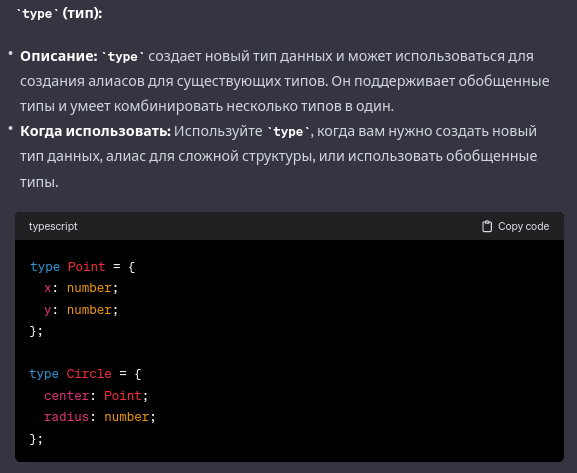
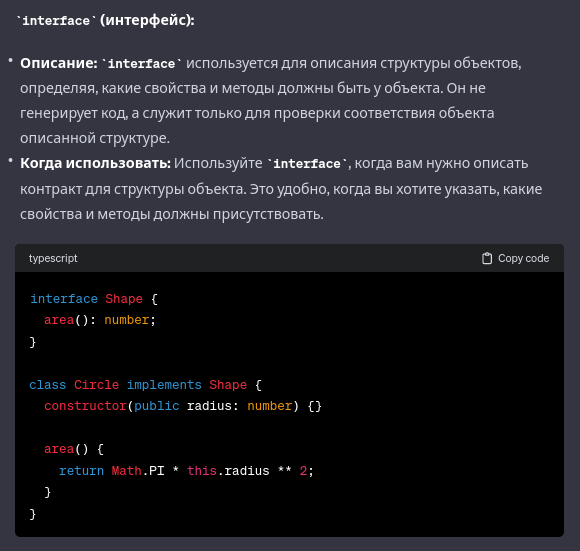
**3. Type Utilities(Утилиты ти****пов)**

Это предопределенные типы, предоставляемые самим TypeScript, которые позволяют выполнять операции на типах, такие как модификация, комбинирование или создание новых типов на основе существующих.

* Awaited<Type> - тип данных возвращаемых промисом
* Partial<Type> - все свойства становятся опциональными
* Required<Type> - все свойства обязательные
* Readonly<Type> - все свойства не могут быть перезаписаны
* Record<Keys, Type> - для коллекции одинаковых пар ключ-значение
* Pick<Type, Keys> - выбор отдельных свойств объекта
* Omit<Type, Keys> - убирает отдельные свойства
* Exclude<UnionType, ExcludedMembers> - убирает из типа указанные свойства
* Extract<Type, Union> - выбирает из типа указанные свойства
* NonNullable<Type> - оставляет в типе любые значения кроме null и undefined
* ReturnType<Type> - создает тип из типов return функции Type

4. Type, class, interface(отличие, особенности что такое)



****

**5. Чем отличает****ся type от interface?**

**type:**

* type позволяет создавать пользовательские типы данных путем комбинирования или алиасирования существующих типов.
* type может быть использован для создания объединения типов (union types), пересечения типов (intersection types), определения условных типов (conditional types) и других сложных конструкций типов.
* type может использоваться с оператором type или = для определения типа.
* type поддерживает инференцию типов и удобно использовать для создания типов, основанных на существующих типах.

**interface:**

* interface используется для определения структуры объекта или класса и обычно используется для определения контракта (сигнатуры методов и свойств) для классов.
* interface может определять свойства, методы, индексы и наследование других интерфейсов.
* interface поддерживает объединение интерфейсов при помощи оператора extends.
* interface может быть реализован классом с помощью ключевого слова implements

**6. Зачем нужен omit, neve****r, unknown?**

* omit - это функция-утилита в TypeScript, которая позволяет создать новый тип данных, исключив указанные свойства из существующего типа. Функция omit принимает два параметра: исходный тип и список имен свойств, которые необходимо исключить. Результатом будет новый тип, который содержит все свойства из исходного типа, за исключением указанных.
* never - это тип данных в TypeScript, который представляет недостижимое значение или ситуацию, которая не может произойти. Он используется, когда функция никогда не возвращает значение или когда переменная не может иметь никакого значения. never также используется в типизации ветвлений (switch) для обозначения непокрытых случаев.

unknown- это тип данных в TypeScript, который представляет значение, о котором ничего неизвестно. Он используется, когда тип значения может быть разным и неизвестным во время компиляции. Переменные типа unknown должны быть явно проверены или приведены к определенному типу перед использованием, чтобы избежать ошибок типизации.

**7.** **Что такое enum?** enum (перечисление) - это конструкция языка TypeScript, которая позволяет определить набор именованных константных значений. enum позволяет создавать типы данных, которые представляют набор допустимых значений, облегчая чтение и поддержку кода.

8. **Чем отличается ts и Proptypes?** **TypeScript:**

* TypeScript - это язык программирования, основанный на JavaScript, который добавляет статическую типизацию к языку.
* TypeScript использует синтаксис типов и деклараций типов для указания типов данных переменных, функций, классов и других элементов кода.
* TypeScript является отдельным языком, требующим компиляции в JavaScript перед запуском кода в браузере или на сервере.
* TypeScript обеспечивает статическую проверку типов во время компиляции, позволяя выявлять ошибки типизации и предоставлять подсказки и автодополнение на основе типов данных.

**PropTypes:**

* PropTypes - это библиотека, разработанная Facebook, для проверки типов данных в React-компонентах.
* PropTypes использует специальный синтаксис для указания ожидаемых типов данных для свойств (props) компонентов.
* PropTypes выполняет проверку типов во время выполнения, предупреждая об ошибках типизации и некорректных значениях свойств.
* PropTypes является дополнительной библиотекой, которую можно добавить в проект React, не требуя компиляции или изменения языка.

9. **Влияет ли TS на исполнение JS в рантайме?** Нет, так как TS компилируется в JS и только потом исполняется.

10. **Что такое утиная типизация?** Принцип, определяющий совместимость типов не основываясь на иерархии наследования или явной реализации интерфейсов, а основываясь на их описании, присуща лишь языкам с динамической типизацией.

**11.** **Type guards, сужающие и расширяющие типы:**

Защита типа позволяет уточнить тип объекта внутри блока условия, если у нас переменная может быть разных типов и к ней применяются

* какие либо действия, то надо проверить тип переменной до применения действий.
* Выражение typeof x === 'number' называется защитником или предохранителем типа (type guard). А процесс приведения определенного типа к более конкретной версии с помощью
* защитников типа и присвоений называется сужением типа (narrowing).
* Виды type guards: typeof, проверка на истинность, проверка на равенство, сужение типов с помощью оператора in (проверка что в объекте есть конкретное свойство), instanceof (является ли одна сущность "экземпляром" другой. 4. Например, выражение x instanceof Foo проверяет, содержится ли Foo.prototype в цепочке прототипов x. Данный оператор применяется к значениям, сконструированным с помощью ключевого слова new)

**12.** **Чем отличается any от unknown?**

* any представляет тип данных, который позволяет значению иметь любой тип. Переменные с типом any не проверяются компилятором TypeScript, и к ним можно применять любые операции без предупреждений о типах. Использование any отключает статическую типизацию TypeScript для соответствующих переменных или значений, что может привести к потере безопасности типов. Рекомендуется использовать any только в случаях, когда точный тип значения неизвестен или в ситуациях, где не требуется типовая безопасность.
* unknown также представляет тип данных, который может иметь любое значение. unknown отличается от any тем, что для значений типа unknown все операции и присваивания требуют явного приведения типов или проверки типов перед использованием. Это делает unknown более безопасным, чем any, поскольку обеспечивает большую типовую безопасность и предотвращает ошибки несоответствия типов. Рекомендуется использовать unknown, когда точный тип значения неизвестен, но вы хотите сохранить типовую безопасность и обеспечить явные проверки типов перед использованием.

13. Декораторы в ts

\*\***Декораторы в TypeScript**:\*\*

Декораторы - это экспериментальная функциональность в TypeScript, позволяющая применять метаданные и изменять поведение классов и их членов на этапе объявления.

\*\*Виды декораторов:\*\*

1. \*\*Декоратор класса:\*\*

- Применяется к объявлению класса.

- Принимает один аргумент - конструктор класса.

```typescript

@classDecorator

class MyClass {}

```

2. \*\*Декоратор метода:\*\*

- Применяется к объявлению метода внутри класса.

- Принимает три аргумента: `target` (экземпляр класса), `propertyKey` (имя метода) и `descriptor` (объект дескриптора метода).

```typescript

class MyClass {

@methodDecorator

myMethod() {}

}

```

3. \*\*Декоратор геттера/сеттера:\*\*

- Применяется к объявлению геттера или сеттера внутри класса.

- Принимает три аргумента: `target` (экземпляр класса), `propertyKey` (имя метода) и `descriptor` (объект дескриптора).

```typescript

class MyClass {

@getterSetterDecorator

get myProperty() {}

set myProperty(value) {}

}

```

4. \*\*Декоратор параметра:\*\*

- Применяется к объявлению параметра метода или конструктора.

- Принимает три аргумента: `target` (экземпляр класса), `propertyKey` (имя метода) и `parameterIndex` (индекс параметра).

```typescript

class MyClass {

myMethod(@parameterDecorator param: string) {}

}

```

\*\*Последовательность выполнения:\*\*

1. \*\*Декораторы класса:\*\*

- Выполняются сверху вниз.

2. \*\*Декораторы членов класса (методы, геттеры/сеттеры, параметры):\*\*

- Выполняются сверху вниз для каждого члена класса.

Порядок выполнения зависит от положения декораторов в коде.

\*\*Пример:\*\*

```typescript

@classDecorator

class Example {

@propertyDecorator

myProperty: string;

@methodDecorator

myMethod() {}

@getterSetterDecorator

get myGetterSetter() {

return this.myProperty;

}

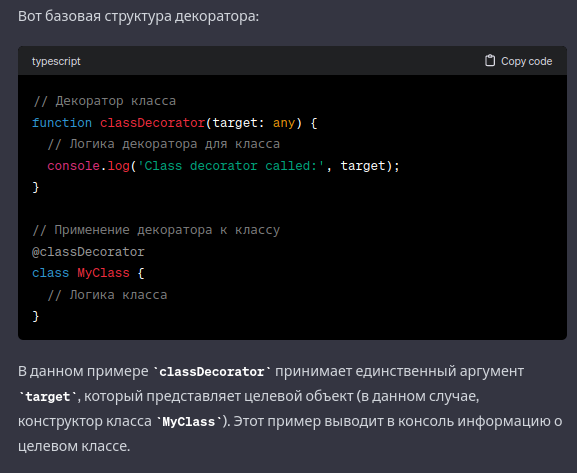
@parameterDecorator

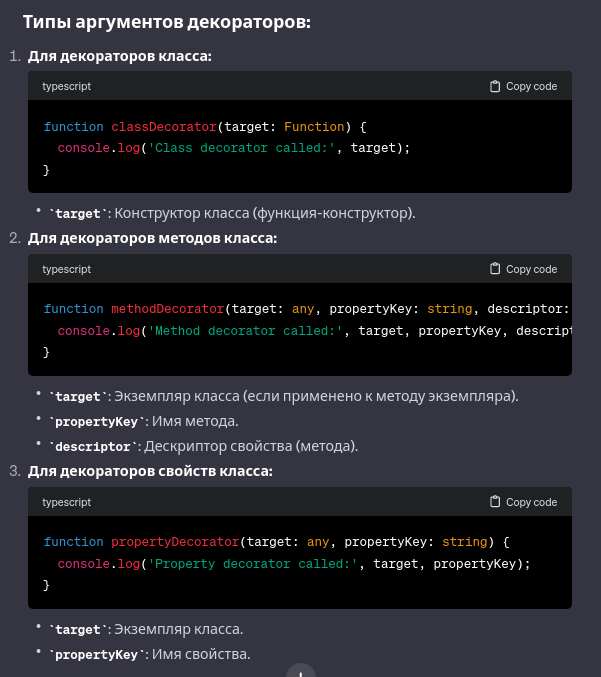
myParam(@parameterDecorator param: string) {}

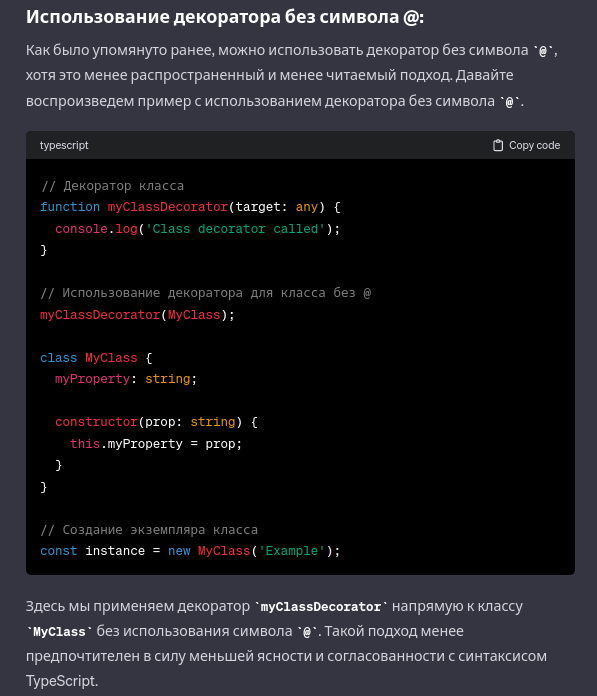
}

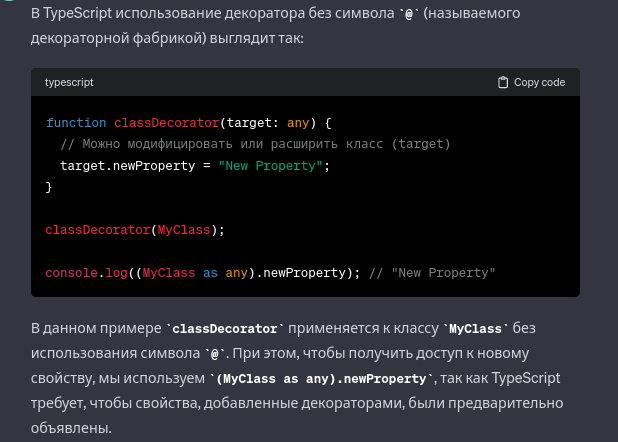
```

**Создание своего декоратора**









**Nest.js**

**1.** **Преимущества использования(в сравнении с express)**

1. \*\*Модульность:\*\*

- Nest.js акцентирует внимание на модульной архитектуре. Приложение строится из множества небольших и переиспользуемых модулей, что способствует поддержке чистого кода и улучшает тестируемость.

- Express предоставляет базовую структуру, и разработчикам часто приходится самостоятельно решать вопросы связанные с архитектурой.

2. \*\*Внедрение зависимостей (DI):\*\*

- Nest.js использует встроенный механизм внедрения зависимостей (DI), что упрощает управление зависимостями и делает код более тестируемым.

- Express не включает в себя механизм DI "из коробки", и разработчики обычно используют сторонние библиотеки для внедрения зависимостей.

3. \*\*Типизация (TypeScript):\*\*

- Nest.js построен с учетом использования TypeScript. Это обеспечивает статическую типизацию, что улучшает поддержку IDE, облегчает рефакторинг и предоставляет преимущества на этапе разработки.

- Express также может использовать TypeScript, но не предоставляет такой интеграции "из коробки".

4. \*\*Мидлвары и Пайпы:\*\*

- Nest.js вводит концепцию "пайпов", которая предоставляет более мощный и гибкий подход к обработке запросов перед тем, как они достигнут контроллера.

- Express использует мидлвары для обработки запросов, но их использование может стать сложным в случае сложной логики обработки.

5. \*\*Декораторы:\*\*

- Nest.js использует декораторы для определения роутов, контроллеров и других элементов приложения, что делает код более декларативным и легко читаемым.

- Express работает на основе более процедурного подхода без явного использования декораторов.

6. \*\*Генерация документации:\*\*

- Nest.js предоставляет интегрированное решение для генерации документации API (Swagger), что упрощает разработку и поддержку API.

- В Express разработчикам нужно использовать сторонние инструменты для генерации документации API.

Nest.js обеспечивает более структурированный подход к построению приложений, что может быть особенно полезно для средних и крупных проектов.

2. **Dependency Injection (DI) (депенденси инджекшен)**

**\*\*Dependency Injection (DI)\*\* в Nest.js предоставляет механизм, который позволяет внедрять зависимости (services, repositories и другие объекты) в компоненты вашего приложения. Вот основные плюсы и минусы DI в Nest.js:**

**### Плюсы DI в Nest.js:**

**1. \*\*Разделение ответственности (Separation of Concerns):\*\***

**- DI позволяет разделить создание и использование объектов. Это улучшает чистоту кода и делает его более поддерживаемым.**

**2. \*\*Тестируемость (Testability):\*\***

**- Внедрение зависимостей упрощает модульное тестирование, так как вы можете легко заменять реальные зависимости моками или заглушками в тестах.**

**3. \*\*Гибкость и расширяемость:\*\***

**- Вы можете легко заменять или изменять зависимости без изменения кода, что делает ваше приложение более гибким.**

**4. \*\*Избежание проблем циклических зависимостей:\*\***

**- Nest.js автоматически обрабатывает циклические зависимости, предотвращая проблемы, которые могут возникнуть в ручном режиме.**

**5. \*\*Конфигурирование внешних зависимостей:\*\***

**- DI упрощает конфигурирование и внедрение внешних зависимостей, таких как настройки приложения, подключение к базе данных и другие.**

**### Минусы DI в Nest.js:**

**3. \*\*Некоторые аспекты могут быть неявными:\*\***

**- Если не следить за структурой приложения, некоторые зависимости могут быть внедрены неявно, что может вызвать затруднения в отслеживании потока данных.**

**Не использование Dependency Injection (DI) может привести к нескольким негативным последствиям:**

1. **Сложность тестирования:**
   * Без DI тестирование может быть затруднено, так как объекты и зависимости создаются напрямую внутри методов, и вы не можете легко вмешиваться и заменять реальные зависимости моками в тестах.
2. **Сильная связанность (Tight Coupling):**
   * Вместо использования интерфейсов и абстракций, объекты могут прямо зависеть от конкретных реализаций. Это делает код менее гибким и труднее поддерживаемым, так как любые изменения в зависимостях могут затронуть множество мест в коде.
3. **Трудности в поддержке и расширении:**
   * Если ваши зависимости прямо встроены в код, изменение или добавление новых зависимостей может потребовать множества изменений в коде, что усложняет поддержку и расширение приложения.
4. **Проблемы с тестированием отдельных компонентов:**
   * Если объекты и зависимости не являются отдельными компонентами, их труднее изолировать для тестирования. Это может привести к необходимости тестировать большие блоки кода целиком, вместо отдельных частей.

**Пример использования DI в Nest.js:**

**```typescript**

**// Пример сервиса, который будет внедрен**

**@Injectable()**

**class MyService {**

**getHello(): string {**

**return 'Hello from MyService!';**

**}**

**}**

**// Пример контроллера, использующего внедренный сервис**

**@Controller('example')**

**class MyController {**

**constructor(private readonly myService: MyService) {}**

**@Get('hello')**

**getHello(): string {**

**return this.myService.getHello();**

**}**

**}**

**```**

**В этом примере `MyController` получает доступ к `MyService` через конструктор. Nest.js автоматически обеспечивает внедрение зависимостей.**

**3.** **Декораторы nest**

**Nest.js использует декораторы для определения различных аспектов приложения. Вот несколько основных декораторов Nest.js и их предназначение:**

**1. \*\*`@Module`\*\*:**

**- \*\*Описание\*\*: Используется для определения модуля.**

**- \*\*Пример\*\*:**

**```typescript**

**@Module({**

**controllers: [AppController],**

**providers: [AppService],**

**})**

**export class AppModule {}**

**```**

**2. \*\*`@Controller`\*\*:**

**- \*\*Описание\*\*: Определяет контроллер в рамках модуля.**

**- \*\*Пример\*\*:**

**```typescript**

**@Controller('cats')**

**export class CatsController {}**

**```**

**3. \*\*`@Injectable`\*\*:**

**- \*\*Описание\*\*: Помечает сервисы, которые могут быть внедрены в другие компоненты.**

**- \*\*Пример\*\*:**

**```typescript**

**@Injectable()**

**export class CatsService {}**

**```**

**4. \*\*`@Inject`\*\*:**

**- \*\*Описание\*\*: Используется для инъекции зависимостей в классы.**

**- \*\*Пример\*\*:**

**```typescript**

**@Inject(CatsService)**

**private readonly catsService: CatsService;**

**```**

**5. \*\*`@Get`, `@Post`, `@Put`, `@Delete`, и т.д.\*\*:**

**- \*\*Описание\*\*: Определяют обработчики маршрутов HTTP в контроллерах.**

**- \*\*Пример\*\*:**

**```typescript**

**@Get()**

**findAll(): string {**

**return 'This action returns all cats';**

**}**

**```**

**6. \*\*`@Param`\*\*:**

**- \*\*Описание\*\*: Извлекает параметры из URL-запроса.**

**- \*\*Пример\*\*:**

**```typescript**

**@Get(':id')**

**findOne(@Param('id') id: string): string {**

**return `This action returns a cat #${id}`;**

**}**

**```**

**7. \*\*`@Body`\*\*:**

**- \*\*Описание\*\*: Извлекает тело запроса (payload) в обработчиках маршрутов.**

**- \*\*Пример\*\*:**

**```typescript**

**@Post()**

**create(@Body() createCatDto: CreateCatDto): string {**

**return `This action creates a new cat with name: ${createCatDto.name}`;**

**}**

**```**

**8. \*\*`@Query`\*\*:**

**- \*\*Описание\*\*: Извлекает параметры запроса из URL.**

**- \*\*Пример\*\*:**

**```typescript**

**@Get()**

**findAll(@Query() query: ListAllEntities): string {**

**return `This action returns all cats (limit: ${query.limit} items)`;**

**}**

**```**

**9. \*\*`@UseGuards`, `@UseInterceptors`, `@UseFilters`, и т.д.\*\*:**

**- \*\*Описание\*\*: Применяет гварды, интерцепторы, фильтры и т.д. к контроллерам или обработчикам маршрутов.**

**- \*\*Пример\*\*:**

**```typescript**

**@UseGuards(AuthGuard)**

**@UseFilters(new HttpExceptionFilter())**

**@UseInterceptors(CacheInterceptor)**

**findAll(): string {**

**return 'This action returns all cats';**

**}**

**```**

**Последовательность выполнения декораторов обычно следует от более общих к более конкретным: `@Module`, `@Controller`, `@Injectable`, и т.д. Конкретные декораторы для обработки маршрутов (`@Get`, `@Post`, и др.) обычно размещаются после `@Controller`. Это позволяет Nest.js построить правильное дерево маршрутов и подготовить приложение к работе.**

****Pipes, Interceptors (пайпы и интерцепторы)****

**В NestJS интерсепторы (interceptors) и конвейеры (pipes) предоставляют механизмы для обработки запросов и ответов в приложении. Они позволяют выполнять дополнительные действия до или после обработки запроса контроллером. Вот более подробное описание каждого из этих элементов:**

**### Интерсепторы (Interceptors):**

**Интерсепторы предоставляют возможность изменять поток обработки запроса до или после выполнения обработчика (контроллера). Они могут быть использованы для внесения изменений в данные запроса, ответа или даже в сам процесс обработки. В NestJS интерсепторы могут быть глобальными (применяются ко всем запросам) или локальными (применяются к определенным контроллерам или действиям).**

**Пример использования интерсептора:**

**```typescript**

**import { Injectable, NestInterceptor, ExecutionContext, CallHandler } from '@nestjs/common';**

**import { Observable } from 'rxjs';**

**@Injectable()**

**export class LoggingInterceptor implements NestInterceptor {**

**intercept(context: ExecutionContext, next: CallHandler): Observable<any> {**

**console.log('Before...');**

**const now = Date.now();**

**return next.handle().pipe(**

**tap(() => console.log(`After... ${Date.now() - now}ms`)),**

**);**

**}**

**}**

**```**

**В данном примере `LoggingInterceptor` логгирует сообщения до и после выполнения обработчика. Он применяется с помощью декоратора `@UseInterceptors` в контроллере или действии.**

**### Конвейеры (Pipes):**

**Конвейеры в NestJS используются для валидации и преобразования данных перед их передачей в контроллер или после получения от него. Они могут быть применены к параметрам маршрута, параметрам запроса, телу запроса и другим частям запроса.**

**Пример использования конвейера:**

**```typescript**

**import { Injectable, PipeTransform, ArgumentMetadata, BadRequestException } from '@nestjs/common';**

**@Injectable()**

**export class ParseIntPipe implements PipeTransform<string, number> {**

**transform(value: string, metadata: ArgumentMetadata): number {**

**const intValue = parseInt(value, 10);**

**if (isNaN(intValue)) {**

**throw new BadRequestException('Validation failed: Invalid number');**

**}**

**return intValue;**

**}**

**}**

**```**

**В данном примере `ParseIntPipe` используется для преобразования строки в целое число. Если преобразование невозможно, генерируется исключение. Конвейер применяется с помощью декоратора `@UsePipes` в контроллере или действии.**

**### Порядок выполнения:**

**При выполнении запроса порядок выполнения интерсепторов и конвейеров может быть следующим:**

**1. \*\*Глобальные интерсепторы и конвейеры\*\*: Применяются ко всем запросам.**

**2. \*\*Интерсепторы и конвейеры контроллера\*\*: Применяются ко всем обработчикам внутри контроллера.**

**3. \*\*Интерсепторы и конвейеры действия (метода)\*\*: Применяются только к конкретному действию.**

**Этот порядок позволяет управлять обработкой запроса на разных уровнях приложения и применять различные обработчики и валидации в зависимости от конкретных требований.**

****Node.js****

**В Node.js термины "stream" и "thread" относятся к различным аспектам ввода/вывода и параллельного выполнения. Давайте рассмотрим каждый из них:**

**### Stream (Поток):**

**\*\*Описание:\*\***

**- \*\*Что это:\*\* Поток представляет собой объект Node.js, предназначенный для обработки данных в формате "поступает, обрабатывается и передается дальше", без необходимости ожидания полного завершения операции.**

**- \*\*Когда использовать:\*\* Используется при работе с большими объемами данных, например, при чтении/записи файлов, обработке HTTP-запросов, работы с сетевыми соединениями.**

**- \*\*Пример:\*\***

**```javascript**

**const fs = require('fs');**

**const readStream = fs.createReadStream('input.txt');**

**const writeStream = fs.createWriteStream('output.txt');**

**readStream.pipe(writeStream);**

**```**

**### Thread (Поток выполнения):**

**\*\*Описание:\*\***

**- \*\*Что это:\*\* В контексте Node.js, JavaScript выполняется в одном потоке (основном потоке событий). Однако Node.js поддерживает многозадачность через использование асинхронных операций и механизмов, таких как событийный цикл и колбэки.**

**- \*\*Когда использовать:\*\* Используется для асинхронного выполнения задач, чтобы избежать блокировки основного потока. Это может включать в себя выполнение долгих операций ввода/вывода, обращение к внешним API, работу с базой данных и т. д.**

**- \*\*Пример:\*\***

**```javascript**

**const fs = require('fs');**

**fs.readFile('file.txt', 'utf8', (err, data) => {**

**if (err) throw err;**

**console.log(data);**

**});**

**```**

**### Когда использовать:**

**- Используйте потоки, когда вам нужно работать с данными в потоке, чтобы уменьшить нагрузку на память.**

**- Используйте поток выполнения для выполнения асинхронных операций и предотвращения блокировки основного потока.**

**### Особенности:**

**- \*\*Stream:\*\***

**- Работает с потоками данных, такими как чтение/запись файлов, HTTP-запросы, потоки данных из сети.**

**- Уменьшает необходимость хранения больших объемов данных в памяти.**

**- \*\*Thread:\*\***

**- Работает с асинхронными операциями для предотвращения блокировки основного потока.**

**- Позволяет эффективно обрабатывать множество запросов, не замедляя работу приложения.**

**В целом, использование потоков и потоков выполнения в Node.js зависит от конкретных требований вашего приложения.**

****Распараллеливание процессов(веб воркеры, чайлд процесс(дочерние процессы), кластеризация)****

**1. \*\*Web Workers (Веб-воркеры):\*\***

**- \*\*Особенности:\*\***

**- Web Workers позволяют выполнять скрипты в фоновом потоке, не блокируя основной поток исполнения.**

**- Они работают в отдельном глобальном контексте, не имея доступа к DOM и основному потоку.**

**- \*\*Пример использования:\*\***

**- В главном файле:**

**```javascript**

**const worker = new Worker('worker.js');**

**worker.postMessage('Hello from main thread!');**

**worker.onmessage = (event) => {**

**console.log('Message from worker:', event.data);**

**};**

**```**

**- В файле `worker.js`:**

**```javascript**

**onmessage = (event) => {**

**console.log('Message from main thread:', event.data);**

**postMessage('Hello from worker!');**

**};**

**```**

**2. \*\*Child Processes (Дочерние процессы):\*\***

**- \*\*Особенности:\*\***

**- Child Processes позволяют создавать дополнительные процессы для выполнения кода в отдельном исполнении.**

**- Каждый дочерний процесс работает в отдельном потоке.**

**- \*\*Пример использования:\*\***

**```javascript**

**const { fork } = require('child\_process');**

**const child = fork('child.js');**

**child.on('message', (message) => {**

**console.log('Message from child process:', message);**

**});**

**child.send('Hello from main process!');**

**```**

**В файле `child.js`:**

**```javascript**

**process.on('message', (message) => {**

**console.log('Message from main process:', message);**

**process.send('Hello from child process!');**

**});**

**```**

**3. \*\*Кластеризация (Cluster):\*\***

**- \*\*Особенности:\*\***

**- Кластеризация в Node.js позволяет создавать несколько процессов, работающих в одном и том же приложении.**

**- Каждый процесс обрабатывает свои запросы, распределяя нагрузку.**

**- \*\*Пример использования:\*\***

**```javascript**

**const cluster = require('cluster');**

**const http = require('http');**

**const numCPUs = require('os').cpus().length;**

**if (cluster.isMaster) {**

**console.log(`Master ${process.pid} is running`);**

**for (let i = 0; i < numCPUs; i++) {**

**cluster.fork();**

**}**

**cluster.on('exit', (worker, code, signal) => {**

**console.log(`Worker ${worker.process.pid} died`);**

**});**

**} else {**

**// Worker process code**

**http.createServer((req, res) => {**

**res.writeHead(200);**

**res.end('Hello from worker!');**

**}).listen(3000);**

**console.log(`Worker ${process.pid} started`);**

**}**

**```**

**\*\*Отличия и преимущества:\*\***

**- \*\*Web Workers vs. Child Processes:\*\***

**- Web Workers предназначены для выполнения кода в браузере и имеют ограниченный доступ к ресурсам. Child Processes предоставляют более широкие возможности для взаимодействия с операционной системой и ресурсами компьютера.**

**- В Web Workers нельзя использовать некоторые API, такие как DOM, window, и document.**

**- \*\*Child Processes vs. Кластеризация:\*\***

**- Child Processes могут быть использованы для выполнения отдельных скриптов или задач в отдельных процессах. Кластеризация используется для распределения нагрузки между несколькими экземплярами приложения.**

**- Кластеризация включает в себя мастер-процесс и несколько рабочих процессов, которые могут принимать запросы независимо.**

**\*\*Когда использовать:\*\***

**- \*\*Web Workers:\*\***

**- Используйте в веб-приложениях для выполнения вычислительно интенсивных задач в фоновом режиме.**

**- \*\*Child Processes:\*\***

**- Используйте, когда требуется выполнение внешнего скрипта или исполнение кода в отдельном процессе.**

**- \*\*Кластеризация:\*\***

**- Используйте для масштабирования приложения на несколько ядер процессора для обработки большого количества одновременных запросов.**

**Различные подходы к распараллеливанию подходят для разных сценариев использования, и выбор зависит от конкретных требований вашего приложения.**

****Event loop****

****SQL****

**## БД Реляционной Модели:**

**### 1. \*\*Транзакции:\*\***

**- \*\*Описание:\*\* Транзакция представляет собой последовательность одного или нескольких SQL-запросов, образующих логическую единицу работы.**

**- \*\*Пример:\*\***

**```sql**

**BEGIN TRANSACTION;**

**UPDATE accounts SET balance = balance - 100 WHERE user\_id = 1;**

**UPDATE accounts SET balance = balance + 100 WHERE user\_id = 2;**

**COMMIT;**

**```**

**- \*\*Особенности:\*\***

**- ACID-свойства: Атомарность, Согласованность, Изолированность, Долговечность.**

**- В случае ошибки, транзакция откатывается (ROLLBACK), в противном случае - фиксируется (COMMIT).**

**### 2. \*\*Джойны:\*\***

**- \*\*Описание:\*\* Джойны используются для объединения данных из разных таблиц на основе определенного условия.**

**- \*\*Примеры:\*\***

**- INNER JOIN:**

**```sql**

**SELECT customers.name, orders.product**

**FROM customers**

**INNER JOIN orders ON customers.id = orders.customer\_id;**

**```**

**- LEFT JOIN:**

**```sql**

**SELECT customers.name, orders.product**

**FROM customers**

**LEFT JOIN orders ON customers.id = orders.customer\_id;**

**```**

**- \*\*Особенности:\*\***

**- INNER JOIN возвращает только строки, которые имеют совпадение в обеих таблицах.**

**- LEFT JOIN возвращает все строки из левой таблицы и соответствующие строки из правой таблицы.**

**### 3. \*\*Индексы:\*\***

**- \*\*Описание:\*\* Индексы используются для ускорения поиска, сортировки и фильтрации данных в таблицах.**

**- \*\*Пример:\*\***

**```sql**

**CREATE INDEX idx\_lastname ON employees(last\_name);**

**```**

**- \*\*Особенности:\*\***

**- Ускоряют выполнение запросов за счет уменьшения количества строк, которые необходимо просмотреть.**

**- Однако, они также увеличивают время выполнения операций вставки, обновления и удаления.**

**## БД Нереляционной Модели:**

**### 1. \*\*Транзакции:\*\***

**- \*\*Описание:\*\* В большинстве нереляционных баз данных отсутствуют транзакции в том виде, как они реализованы в реляционных базах данных.**

**- \*\*Примеры (MongoDB):\*\***

**```javascript**

**const session = await client.startSession();**

**session.startTransaction();**

**try {**

**await collection1.updateOne({ \_id: 1 }, { $set: { value: 42 } }, { session });**

**await collection2.insertOne({ \_id: 1, value: 42 }, { session });**

**await session.commitTransaction();**

**} catch (error) {**

**await session.abortTransaction();**

**throw error;**

**} finally {**

**session.endSession();**

**}**

**```**

**- \*\*Особенности:\*\***

**- MongoDB поддерживает транзакции с использованием `startSession`, `startTransaction`, `commitTransaction`, и `abortTransaction`.**

**### 2. \*\*Джойны:\*\***

**- \*\*Описание:\*\* Некоторые нереляционные базы данных предоставляют возможности для объединения данных, но они обычно реализованы не так, как в реляционных базах данных.**

**- \*\*Примеры (MongoDB):\*\***

**```javascript**

**db.orders.aggregate([**

**{**

**$lookup: {**

**from: "products",**

**localField: "product\_id",**

**foreignField: "\_id",**

**as: "orderDetails"**

**}**

**}**

**]);**

**```**

**- \*\*Особенности:\*\***

**- В MongoDB используется агрегационный фреймворк для объединения данных из разных коллекций.**

**### 3. \*\*Индексы:\*\***

**- \*\*Описание:\*\* Нереляционные базы данных также могут использовать индексы для улучшения производительности, но с различиями в сравнении с реляционными базами данных.**

**- \*\*Примеры (MongoDB):\*\***

**```javascript**

**db.collection.createIndex({ field: 1 });**

**```**

**- \*\*Особенности:\*\***

**- Индексы в нереляционных базах данных могут иметь свои особенности в сравнении с тем, как они реализованы в реляционных базах данных.**

**- В MongoDB индексы могут быть созданы для полей вложенных документов, массивов и т.д.**

**### Общие особенности:**

**- \*\*Гибкость схемы:\*\***

**- \*\*Реляционные БД:\*\* Требуют определенной схемы данных.**

**- \*\*Нереляционные БД:\*\* Обеспечивают гибкость схемы, что позволяет хранить различные типы данных в одной коллекции (или документе).**

**- \*\*Масштабируемость:\*\***

**- \*\*Реляционные БД:\*\* Вертикальное масштабирование (увеличение мощности сервера).**

**- \*\*Нереляционные БД:\*\* Горизонтальное масштабирование (добавление новых серверов).**

**- \*\*Сложность запросов:\*\***

**- \*\*Реляционные БД:\*\* Сложные запросы часто легче выражаются с использованием SQL.**

**- \*\*Нереляционные БД:\*\* Простые запросы могут быть легко выполнены, но сложные могут потребовать более глубокого понимания структуры данных и использования инструментов запросов.**

**Выбор между реляционными и нереляционными базами данных зависит от требований конкретного приложения**

**и характера данных, с которыми вы работаете.**

****Индексы в бд****

**Индексы в базах данных предназначены для оптимизации поиска, сортировки и фильтрации данных. Они значительно ускоряют выполнение запросов, уменьшая количество записей, которые необходимо просмотреть при выполнении операций. Вот некоторые основные виды индексов, которые используются в базах данных:**

**### 1. \*\*Кластерный (Clustered) и Некластерный (Non-Clustered) Индексы:\*\***

**- \*\*Кластерный индекс:\*\***

**- \*\*Описание:\*\* Кластерный индекс упорядочивает сами строки таблицы на основе значений ключа индекса. Таким образом, строки в таблице фактически упорядочены в том же порядке, что и значения индекса.**

**- \*\*Особенности:\*\***

**- В таблице может быть только один кластерный индекс.**

**- Использование кластерного индекса может привести к улучшению производительности при поиске, но может вызвать фрагментацию данных при обновлении.**

**- \*\*Некластерный индекс:\*\***

**- \*\*Описание:\*\* Некластерный индекс создает отдельную структуру данных, хранящую значения индекса и указатели на фактические строки таблицы.**

**- \*\*Особенности:\*\***

**- В таблице может быть несколько некластерных индексов.**

**- Облегчает поиск, но не изменяет физического порядка строк в таблице.**

**### 2. \*\*Уникальный Индекс:\*\***

**- \*\*Описание:\*\* Уникальный индекс гарантирует уникальность значений в индексируемом столбце или группе столбцов.**

**- \*\*Особенности:\*\***

**- Запрещает вставку или обновление строкы, если они создают дубликаты в уникальном индексе.**

**- Поддерживается как кластерными, так и некластерными индексами.**

**### 3. \*\*Составной Индекс:\*\***

**- \*\*Описание:\*\* Составной индекс создается для нескольких столбцов, что позволяет оптимизировать запросы, содержащие условия по нескольким столбцам.**

**- \*\*Особенности:\*\***

**- Индекс строится на комбинации значений из различных столбцов.**

**- Позволяет ускорить выполнение запросов, в которых используется фильтрация и сортировка по нескольким столбцам.**

**### 4. \*\*Полнотекстовый Индекс (Full-Text Index):\*\***

**- \*\*Описание:\*\* Полнотекстовый индекс создается для столбцов, содержащих текстовую информацию, и позволяет эффективно выполнять поиск по словам и фразам в тексте.**

**- \*\*Особенности:\*\***

**- Поддерживает операции поиска с учетом формы слов, стемминга и прочих лингвистических особенностей.**

**### 5. \*\*Геопространственный Индекс (Spatial Index):\*\***

**- \*\*Описание:\*\* Геопространственный индекс используется для ускорения поиска и анализа пространственных данных, таких как точки, линии или полигоны.**

**- \*\*Особенности:\*\***

**- Поддерживает операции, связанные с геометрическими объектами, такими как пересечение, ближайшие объекты и т.д.**

**- Часто используется в геоинформационных системах (ГИС).**

**### В БД Нереляционной Модели:**

**В нереляционных базах данных концепция индексов также может быть присутствует, но реализация может отличаться. Например, в MongoDB индексы создаются для полей коллекции и могут быть уникальными, текстовыми или составными.**

**### Общие Рекомендации по Использованию Индексов:**

**1. \*\*Анализируйте Запросы:\*\***

**- Создавайте индексы на основе запросов, которые часто выполняются в вашем приложении.**

**2. \*\*Осторожно Используйте Индексы при Модификации Данных:\*\***

**- Индексы улучшают производительность чтения, но могут замедлить операции записи. Внимательно оценивайте баланс.**

**3. \*\*Обслуживайте Индексы:\*\***

**- Периодически проводите обслуживание индексов, такое как перестройка, чтобы уменьшить фрагментацию.**

**4. \*\*Используйте Индексы Для Сортировки:\*\***

**- Индексы могут ускорить операции сортировки, используйте их в запросах, требующих упорядочивания.**

**5. \*\*Подбирайте Подходящий Тип Индекса:\*\***

**- Выбирайте тип индекса в зависимости от характера запросов (уникальный, полнотекстовый и т.д.).**

**Использование индексов требует внимательного анализа требований приложения и запросов, а также понимания того, как работают различные типы ин**

**дексов в конкретной СУБД.**

****Индексы в БД****

**Индексы в базах данных предназначены для ускорения выполнения запросов, уменьшения времени поиска и повышения общей производительности системы. В зависимости от требований и структуры данных, существуют различные виды индексов. Вот некоторые из них:**

**### 1. \*\*Кластерный и Некластерный Индексы:\*\***

**- \*\*Кластерный Индекс:\*\***

**- \*\*Описание:\*\* Упорядочивает фактические строки таблицы на основе значений ключа индекса. Фактически, строки в таблице упорядочиваются в том же порядке, что и значения в кластерном индексе.**

**- \*\*Применение:\*\* Ускорение поиска, но может вызвать фрагментацию данных при обновлении.**

**- \*\*Некластерный Индекс:\*\***

**- \*\*Описание:\*\* Создает отдельную структуру данных, хранящую значения индекса и указатели на фактические строки таблицы.**

**- \*\*Применение:\*\* Позволяет ускорить выполнение запросов, но не изменяет физического порядка строк в таблице.**

**### 2. \*\*Уникальный Индекс:\*\***

**- \*\*Описание:\*\* Гарантирует уникальность значений в индексируемом столбце или группе столбцов.**

**- \*\*Применение:\*\* Запрещает вставку или обновление строкы, если они создают дубликаты в уникальном индексе.**

**### 3. \*\*Составной Индекс:\*\***

**- \*\*Описание:\*\* Создается для нескольких столбцов, что позволяет оптимизировать запросы, содержащие условия по нескольким столбцам.**

**- \*\*Применение:\*\* Ускоряет выполнение запросов, где используется фильтрация и сортировка по нескольким столбцам.**

**### 4. \*\*Полнотекстовый Индекс (Full-Text Index):\*\***

**- \*\*Описание:\*\* Используется для эффективного выполнения поиска по словам и фразам в тексте.**

**- \*\*Применение:\*\* Поддерживает операции поиска с учетом формы слов, стемминга и других лингвистических особенностей.**

**### 5. \*\*Геопространственный Индекс (Spatial Index):\*\***

**- \*\*Описание:\*\* Используется для ускорения поиска и анализа пространственных данных, таких как точки, линии или полигоны.**

**- \*\*Применение:\*\* Поддерживает операции, связанные с геометрическими объектами, такими как пересечение, ближайшие объекты и т.д.**

**### 6. \*\*Хеш-Индекс (Hash Index):\*\***

**- \*\*Описание:\*\* Используется для создания хеш-таблицы, которая ускоряет поиск по значению ключа.**

**- \*\*Применение:\*\* Эффективен для точных поисков по равенству, но не поддерживает диапазонные запросы.**

**### 7. \*\*Битовый Индекс (Bitmap Index):\*\***

**- \*\*Описание:\*\* Используется для индексации битовых полей или колонок, где каждый бит представляет наличие или отсутствие значения.**

**- \*\*Применение:\*\* Эффективен для колонок с ограниченным набором возможных значений.**

**### 8. \*\*Покрывающий Индекс (Covering Index):\*\***

**- \*\*Описание:\*\* Включает в себя все столбцы, необходимые для выполнения запроса, избегая обращения к фактическим строкам таблицы.**

**- \*\*Применение:\*\* Уменьшает количество операций ввода-вывода и ускоряет выполнение запросов.**

**### 9. \*\*Индекс с Подсчетом Количества Уникальных Значений (Unique Count Index):\*\***

**- \*\*Описание:\*\* Используется для быстрого определения количества уникальных значений в столбце.**

**- \*\*Применение:\*\* Эффективен для оптимизации запросов, требующих подсчета уникальных значений.**

**Выбор подходящего типа индекса зависит от конкретных требований приложения, характера данных и типов запросов, которые выполняются в системе.**