**JAVA\_SCRIPT**

**js\_1. Какие типы данных есть в JS?**

В JavaScript существует 8 типов данных.

7 примитивов:

- Undefined (Неопределённый тип) - Boolean (Булев, Логический тип) - BigInt

- Number (Число) - String (Строка) - Symbol (в ECMAScript 6)

- Null - Специальный примитив, используемый не только для данных но и в качестве указателя на финальную точку в Цепочке Прототипов.

Ссылочный тип:

- Object (Объект) - структура, используемая не только для хранения данных, но и для создания других структур, где любая структура создаётся с использованием ключевого слова new: new Object, new Array, new Map, new Set, new WeakMap, new WeakSet, new Date и др.

**js\_2. Какие типы в JS изменяемые, а какие нет?**

Все типы данных, кроме объектов, являются иммутабельными (значения не могут быть модифицированы, а только перезаписаны новым полным значением).

**js\_3. Что значит ссылка на переменную?**

Примитивные значения хранятся в переменной именно как значение, а объекты — по ссылке на место в памяти. То, как будут храниться данные, жёстко связано с типом данных. Нельзя заставить значение примитивного типа храниться по ссылке, и наоборот.

Примитивные типы данных

Когда в переменную сохраняется примитивное значение, то в память записывается какое-то количество байт, которое описывает это значение. Таким образом можно сказать, что переменная уже сразу содержит эти байты.



Если присвоить какое-то значение переменной в другую, то мы просто скопируем это же количество байт в новое место.



Ссылочные типы данных

Объекты хранятся по-другому, т.к. неизвестно, какое количество памяти для них понадобится. Для этого компьютер отдаёт нам ссылку на место, где данные хранятся, и самостоятельно будет работать с ними по инструкциям, которые мы ему даём. Таким образом в переменную мы получаем лишь ссылку на данные. Такой тип данных называется ссылочным и таким образом хранятся объекты, массивы и функции.

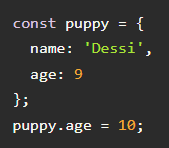
Аргументы функций

При передаче значений как аргументов в функцию все особенности его типа данных сохраняются:

* Примитивы, копируются как значение в аргумент.
* При передаче ссылочного типа копируется ссылка. Все изменения в объекте, который был передан в качестве аргумента, будут видны всем, кто владеет ссылкой.

**js\_3.1 Что такое мутирование?**

Это изменение объекта или массива без создания новой переменной и переприсваивания значения. Например:



Решением являются иммутабельные (неизменяемые) структуры данных. Эта концепция предусматривает создание нового объекта для каждого обновления.

Если у вас есть 9-месячный puppy, который внезапно подрос, придется записать его в новую переменную grownUpPuppy. К сожалению, иммутабельность из коробки в JavaScript не поддерживается.

Можно использовать метод Object.freeze, он защищает собственные свойства объекта от изменений.

Опасность мутаций заключается в том, что т.к. объекты передаются по ссылке, меняя что-то в коде, нечто находящиеся в где-то в другом месте, тоже может поменяться.

**js\_4. Что такое NaN? Как проверить, что переменная - NaN?**

Это специальное числовое значение (т.е. принадлежит к типу number) - Not-A-Number (не-число). **NaN** можно получить, например при:

- Математической операции c undefined (undefined + undefined = NaN)

- если результат математической операции не находится в поле действительных чисел (например, взятие корня от -1).

- арифметической операции со строкой, кроме сложения ("привет" \* 5)

- любая математическая операция с NaN возвращает NaN. Если где-то в математическом выражении есть NaN, то оно распространяется на весь результат (есть только одно исключение: NaN \*\* 0 равно 1).

- ошибка парсинга числа (например, при попытке превратить строку в число parseInt('привет'))

Т.к. NaN не является равным ничему, даже самому себе, нельзя просто сравнить NaN === NaN и NaN == NaN даст false. Для такой проверки существуют специальные методы:

- метод isNaN(value) преобразует значение в число и проверяет является ли оно isNaN(NaN) ); // true isNaN("str") ); // true

- метод Number.isNaN(value) определит, является ли переданное значение NaN, без принудительного преобразования в число.

- метод Object.is() - определяет, являются ли два значения одинаковыми, сравнивает примерно как === но, еще и работает с NaN - Object.is(NaN, NaN) будет true.

var isSame = Object.is(value1, value2);  
value1 - Первое сравниваемое значение.  
value2 - Второе сравниваемое значение.

- альтернативный вариант проверки, это сравнить, равна ли переменная самой себе x !== x:  
var x = NaN  
x != x // true  
x !== x // true

**js\_5. Что значит создавать переменную через конструктор или через литерал? Есть ли отличия?**

Литерал, или безымянная константа — это фиксированное значение, которое записывается непосредственно в коде программы.

В следующем примере 1 и Кот являются литералами, а number и pet это переменные:

int number = 1;

string pet = "Кот";

Создавать переменную через конструктор, это значит, что для создания используется конструктор new. Конструктор всегда возвращает объект.

Например:

Когда Number вызывается в качестве конструктора с ключевым словом new, он создаёт объект Number, который не является примитивом.

typeof new Number(42) === "object"

new Number(42) !== 42 // true

new Number(42) != 42 // false

new Number(42) == 42 // true

new Number(42) === 42 // false

При вызове конструктора (с помощью new) он приводит значение к соответствующему примитиву и возвращает объект-оболочку, который не является примитивом.

Если вызвать как функцию (без new) вернется соответствующее примитивное значение.

const a = new String("Hello world"); a === "Hello world" false

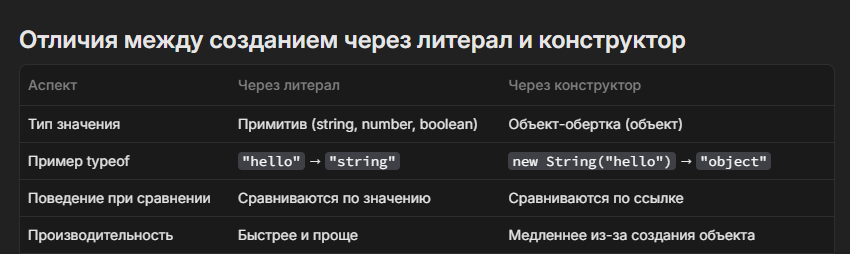
const b = String("Hello world"); // b === "Hello world" is true

a instanceof String; // is true

b instanceof String; // is false

typeof a; // "object"

typeof b; // "string"



**js\_6. Когда при обращении к свойству объекта стоит использовать точечную нотацию, а когда квадратные скобки?**

Dot Notation допускает только статические ключи, а Bracket Notation допускает динамические ключи, вычисляемые из выражения во время выполнения, а также ключи, состоящие из нескольких слов.

Точечную нотацию использовать, когда имя свойства является статическим т. е. известно заранее и вводятся напрямую, и также, когда имя свойства соответствует правилам именования переменных (например, не содержит пробелов, специальных символов и не начинается с цифры).

Точечная нотация менее громоздкая, быстрее пишется.

Нотацию скобок использовать, когдаимя свойства формируется динамически (ключ этого свойства может быть получен из выражений типа getKey(), "my" + "key", или keyVariable.), или имя свойства хранится в переменной, или имя свойства содержит специальные символы (например, пробелы, дефисы) или начинается с цифры.

**js\_7 Что произойдет, если попытаться получить несуществующее в объекте свойство?**

Ошибки не будет. Вернется undefined.

**js\_8 Что делает метод hasOwn/hasOwnPropery?**

Оба метода Object.hasOwn() и Object.hasOwnProperty() проверяют существование ключа в объекте. Они возвращают true, если указанный объект имеет собственное свойство, которое было передано. Если свойство унаследовано или не существует, метод возвращает false.

Object.hasOwn() более современный метод, он пришел на замену Object.hasOwnProperty(). Object.hasOwn() работает для объектов без прототипа (например Object.create(null)) и для объектов, которые переопределили унаследованный hasOwnProperty() метод:

const person = Object.create(null);

person.age = 35;

Object.hasOwn(person, 'age')); // => true

person.hasOwnProperty('age')); // Error - person.hasOwnProperty is not a function

**js\_9 Когда использовать null, а когда undefined?**

В отличие от undefined, когда компьютер не знает, что лежит в переменной, null — это как раз одно из значений переменной.

Null — это специальное значение, которое означает, что переменная есть, но в переменной ничего нет.

Undefined используется для обозначения того, что переменная не была определена, это будет результатом неинициализированной переменной.

Обычно null присваивается явно, а undefined присваивается автоматически.

Можно использовать undefined, когда нужно проверить состояние переменной например, если она была объявлена, но еще не инициализирована.

Можно использовать null, когда нужно сбросить значение переменной (например, удалить объект).

**js\_10 Какие есть способы создания глобальных переменных?**

Глобальные переменные — это такие переменные, которые доступны из любого места в коде

1. присвоить значение через глобальный объект

globalThis.globalVar = "Привет";

2. объявить переменную в глобальной области видимости, то есть вне области видимости блока, функции или модуля.

В браузере глобальные функции и переменные, объявленные с помощью var (не let/const!), становятся свойствами глобального объекта.

**js\_11 Для чего нужна директива use strict?**

'use strict' включает строгий режим выполнения JavaScript. В 2009 году появился ECMAScript 5 (ES5), который добавил новые возможности в язык и изменил некоторые из существующих. Чтобы устаревший код работал, как и раньше, по умолчанию подобные изменения не применяются, и, чтобы явно их активировать необходимо использовать директиву "use strict". В строгом режиме интерпретатор будет явно выбрасывать ошибки на действия, которые ранее пропускал.

**js\_12 Что такое выражения и инструкции? В чем отличия между ними?**

Выражением (expression) является любой корректный блок кода, который возвращает значение.

Ниже 5 это expression, оно выражается в значение 5:

**const** x **=** 5;

Инструкция — это команда, которая выполняет действие. Инструкции не возвращают значения сами по себе. Они могут содержать одно или несколько выражений и управляют потоком выполнения программы.

if, while, for, const, import— примеры инструкций. Они производят или контролируют действия, но не превращаются в значения.

**js\_13 Чем отличаются var, let, const?**

*1. Обращение к переменной.*

К var можно обратиться до её фактического объявления, и значением будет undefined.



Это происходит потому, что переменная var поднимается (hoisting) в верхнюю часть своей области видимости. Переменная уже была объявлена (поднята вверх), но её значение ещё не было присвоено, поэтому выводится undefined. Механизм поднятия применяется только к переменным, объявленным через var.

К let и const нельзя обращаться до момента объявления – появиться ошибка: переменная не определена. У let и const есть так называемая *temporal dead zone* (TDZ) — момент выполнения скрипта до объявления переменной.

*2. Область видимости*

Var переменные, имеют функциональную область видимости. Это значит, что они доступны только в пределах текущей функции или глобального объекта, если функции нет. Объявление переменных var вне функций делает их глобальными переменными. Они доступны как свойства глобального объекта.

localVar доступна внутри функции example, поскольку var не имеет блочной области видимости. Здесь также используется механизм поднятия. Через поднятие переменная «всплывает» вверх в пределах текущей области видимости - в примере — до начала функции, и localVar становится видимой во всей функции, а не только внутри блока if, где была объявлена.



Let и const переменные имеют блочную область видимости и не становятся свойством глобального объекта (window в браузере, global в Node.js). Блочная область видимости не даёт получить значение переменной вне блока, где она была объявлена.

*3. Переназначение и повторное объявление*

Var можно заново объявлять сколь угодно раз – это будет считаться изменением значения уже существующей переменной. Получается, что значение, хранящееся в переменной var, можно изменить двумя способами -обратиться к имени переменной и присвоить новое значение или обратиться к имени переменной вместе с ключевым словом var.

Let переменную нельзя повторно объявлять в том же блоке кода. Повторное объявление вызовет ошибку. Переназначать можно.

*4. Инициализация.*

Для var инициализация не обязательна, для let и const обязательна.

**js\_14. Почему использование const может быть предпочтительнее?**

- Переназначения могут стать причиной багов – можно не заметить, что переменная переназначится, и ожидать старого значения.

- const защищает от случайного переназначения.

- С const код становится читабельнее – сразу понятно, что переменную нельзя изменять, и она в коде не изменится.

- использование const помогает интерпретатору JavaScript оптимизировать код, так как он знает, что значение переменной не будет изменяться. Это может привести к небольшому улучшению производительности.

**js\_15. Что такое тернарный оператор?**

Оператор в JavaScript, принимающий три операнда: условие, за которым следует знак вопроса (?), затем выражение, которое выполняется, если условие истинно, сопровождается двоеточием (:), и, наконец, выражение, которое выполняется, если условие ложно.

**js\_16. Что делает оператор for..in? Какие имеются особенности при использовании этого оператора с массивами?**

Цикл for...in перебирает ключи объекта, не только собственные, но и перечисляемые свойства, наследуемые от цепочки прототипов. Перечисляемые свойства — это свойства, у которых флаг enumerable установлен в true.

Особенности использования с массивами:

В цикл for...in попадут не только целочисленные свойства, но и строковые, а также наследуемые.

Переменная цикла на каждой итерации, соответствующая индексу текущего элемента, будет иметь тип строки, а не числа.

Порядок перебора свойств с помощью for...in не гарантируется.

Особенности метода:

Имена свойств в цикле не повторяются. Свойства объекта имеют наивысший приоритет, а свойства ближайших прототипов — больший приоритет над свойствами прототипов, находящихся дальше от объекта в его цепочке прототипов.



Если несколько объектов в цепочке прототипов имеют свойство с одинаковым именем, будет учитываться только первое из них, и оно будет участвовать в цикле только в том случае, если перечисляемое. Если оно неперечисляемое, никакие другие свойства с таким же именем дальше по цепочке прототипов не будут участвовать в цикле, даже если они являются перечислимыми.

**js\_17. Как безопасно проверить, что переменная существует (была объявлена), и не словить ReferenceError?**

1. Сравнить тип с undefined, typeof не вызывает ошибку, если переменная не была объявлена:

if(typeof dfshkhdkjfh !== "undefined")

2. оператор in, чтобы узнать, существует ли свойство в объекте.

**js\_18. Способы создания массивов (литерал, конструктор, фабричные методы Array.from() и Array.of());**

1. Литерал.

Просто указать элементы массива в квадратных скобках.

const array1 = [1, 2, 3, 4, 5];

2. Array.of()

Метод, который создаёт массив из переданных ему аргументов независимо от их количества.

3. Конструктор массива new Array()

Конструктор работает практически одинаково с методом Array.of(). Разница в том, что, если передать один числовой аргумент, new Array() будет считать этот аргумент длинной массива, и вернет пустой массив заданной длины, а Array.of() посчитает это первым элементом массива, и вернет массив с одним элементом.

4. Array.from()

Метод создаёт новый массив на основе переданного объекта. Объект должен быть либо псевдомассивом (как строка или объект arguments, т.е иметь числовые индексы и свойство length), либо итерируемым объектом (как Set или Map, т.е. объекты с методом Symbol.iterator).

Принимает три аргумента, последние два не обязательны:

-Объект, на основе которого создаётся массив.

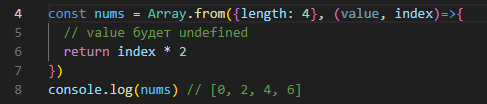
-Функция преобразования элемента перед его добавлением в массив. Работает как метод map().

-Значение, которое будет использоваться как this в функции из второго параметра.

Возвращает новый массив, составленный из элементов переданного объекта.

Можно использовать Array.from(), чтобы генерировать последовательности значений без использования классического цикла for.

Для этого нужно создать объект, который соответствует требованиям — имеет свойство length и индексы. Так как размер массива не всегда совпадает с количеством элементов внутри, мы можем создать объект со свойством length, но без индексированных элементов, и создавать такие элементы с помощью второго аргумента:



**js\_19. Удаление элемента из массива (какие есть способы и в чем особенности)?**

Для удаления первого элемента - shift()

Для удаления последнего элемента - pop()

Для удаления какого-либо элемента по условию, используется метод filter() - создаёт новый массив со всеми элементами, прошедшими проверку, задаваемую в передаваемой функции.

Для удаления элемента по определенному индексу, можно использовать метод – splice(), указав первым аргументом индекс с которого начинать удаление, и вторым аргументом передать число удаляемых элементов.

Можно изменить свойство length – и тем самым удалить последние элементы массива, или вообще все.

**js\_20. Свойство length у массива**

Значение свойства length — целое положительное число в диапазоне от 0 до 232. Представляет самый наибольший индекс в массиве + 1. Корректнее всего говорить, что свойство length хранит количество ячеек доступных для записи в массиве, а не количество элементов.

Cвойство length изменяемое. Можно установить свое значение, и, если новое значение меньше текущего, массив будет обрезан до новой длины, если больше — массив будет расширен, и новые элементы будут заполнены значением undefined.

**js\_20.1. Какое значение будет у свойства length массива a и почему:**

const a = [1, 2, 3];

a[10] = 4;

Значением будет 11. Тк. Это свойство не зависит от количества элементов в массиве, и его можно переписать вручную.

**js\_20.2. Что будет, если переприсвоить новое значение length?**

Если уменьшить length – то массив обрежется. Если увеличить, то реальное количество элементов в массиве увеличивается;

**js\_20.3. Влияет ли на length удаление элемента посередине массива? Какие способы удаления элементов влияют на длину, а какие нет?**

delete не изменит, splice изменит.

**js\_21. Как проверить, что в переменной лежит массив?**

* Метод Array.isArray()
* Оператор instanceof - fruits instanceof Array // true

 Позволяет проверить, является ли объект экземпляром массива.

* свойство constructor - fruits.constructor === Array // true

Позволяет проверить, является ли конструктор экземпляра массивом.

* Object.prototype.toString.call() // [object Array]

Этот метод более универсален и может быть использован для проверки типа объекта. Возвращает строку, представляющую тип объекта.

**js\_22. slice что делает, как и когда использовать?**

Возвращает новый массив, содержащий поверхностную копию части исходного массива, при этом исходный массив не изменяется.

arr.slice([begin[, end]])

Первым параметром принимает индекс, по которому начинать извлечение. Если индекс отрицательный, первый параметр указывает смещение от конца последовательности. Вызов slice(-2) извлечёт два последних элемента последовательности. Если параметр не определён, slice() начинает работать с индекса 0. Если параметр больше длины последовательности вернётся пустой массив.

Вторым параметром принимает индекс, по которому заканчивать извлечение, до, но не включая переданный индекс.

Если индекс отрицательный, end указывает смещение от конца последовательности. Вызов slice(2, -1) извлечёт из последовательности элементы начиная с третьего элемента с начала и заканчивая вторым с конца. Если end >= array.length или end опущен, slice() извлекает все элементы до конца последовательности.

Метод slice() может использоваться, когда необходимо скопировать часть элементов исходного массива в новый массив.

Одноименный метод есть для строк.

**js\_23. at что делает, как и когда использовать?**

Метод аргументом принимает значение в виде целого положительного и отрицательного числа и возвращает элемент массива с данным индексом. При отрицательном значении отсчёт происходит с конца массива.

Нельзя использовать array[-1], т.к. все значения внутри квадратных скобок трактуются буквально как строковые свойства. Из-за этого попытка обращения к -1 элементу будет прочитана как array["-1"], что является нормальным строковым значением, а не индексом массива.

Одноименный метод есть для строк.

**js\_24. Reverse что делает, как и когда использовать?**

Метод reverse() на месте переставляет элементы массива, на котором он был вызван, изменяет массив и возвращает ссылку на него. Первый элемент становится последним, а последний — первым.

**js\_25. toReversed что делает, как и когда использовать?**

Метод toReversed() возвращает новый массив с расположенными в обратном порядке элементами. Является копирующей версией метода reverse(). Может работать с псевдомассивами, в виде:

Array.prototype.toReversed.call(pseudoArr)

**js\_26. Splice что делает, как и когда использовать?**

Метод изменяет содержимое массива, удаляя существующие элементы и/или добавляя новые.

array.splice(start[, deleteCount[, item1[, item2[, ...]]]])

Первым параметром принимает индекс, по которому начинать изменять массив. Если больше длины массива, реальный индекс будет установлен на длину массива. Если отрицателен, указывает индекс элемента с конца.

Вторым параметром (необязательный) принимает целое число, указывающее количество удаляемых из массива элементов. Если передать 0, элементы не удаляются. Если deleteCount опущено, или больше количества элементов, оставшихся в массиве, начиная с индекса start, то будут удалены все элементы до конца массива. Т.е. если передать 1 аргумент – метод удалит все элементы до конца, начиная с указанного индекса, но не включая его.

Третий параметр (необязательный) - добавляемые к массиву элементы. Элементы вставляются перед индексом переданным первым параметром.

Возвращает массив из удаленных элементов. Если будет удалён только один элемент, вернётся массив из одного элемента. Если никакие элементы не будут удалены, вернётся пустой массив.

**js\_27. toSpliced что делает, как и когда использовать?**

Это копирующая версия метода splice(). Метод toSpliced() возвращает копию исходного массива после редактирования - удаления, добавления или замены. В отличие от метода splice(), toSpliced() не меняет исходный массив и не возвращает удаляемые элементы.

**js\_28. reduce что делает, как и когда использовать?**

Метод применяет функцию к каждому элементу массива, возвращая одно результирующее значение.

Первым аргументом принимает функцию, которая выполнится для каждого элемента массива. Функция принимает 4 аргумента:

- accumulator - аккумулятор, аккумулирующий значение, которое возвращает функция callback после посещения очередного элемента.

- currentValue - текущий обрабатываемый элемент массива.

- Index - индекс текущего обрабатываемого элемента массива.

- array - массив, для которого была вызвана функция reduce.

Вторым аргументом принимает начальное значение – используемое в качестве первого аргумента при первом вызове функции callback.

При первом вызове функции, параметры accumulator и currentValue могут принимать одно из двух значений. Если при вызове reduce() передан аргумент initialValue, то значение accumulator будет равным значению initialValue, а значение currentValue будет равным первому значению в массиве. Если аргумент initialValue не задан, то значение accumulator будет равным первому значению в массиве, а значение currentValue будет равным второму значению в массиве.

**js\_29. Sort что делает, как и когда использовать?**

Метод на месте сортирует элементы массива и возвращает отсортированный массив. Имеет один необязательный параметр – функцию-компоратор, определяющую порядок сортировки. Если параметр опущен, каждый элемент массива приводится к строке, и элементы сортируются в порядке следования символов Unicode.

В функцию-компаратор передаются два аргумента a и b. Результат сравнения возвращаемого значения из функции с 0 определяет порядок элементов:

* значение меньше 0 означает, что элемент a меньше, чем элемент b;
* значение больше 0 означает, что элемент a больше, чем элемент b;
* значение равно 0 означает, что элементы равны.

**js\_30. toSorted что делает, как и когда использовать?**

Метод массива toSorted() это копия метода sort(), но, в отличие от него, не мутирует исходный массив, а возвращает новый отсортированный массив.

**js\_31. Filter что делает, как и когда использовать?**

Метод filter() вызывает переданную функцию-предикат(функции, которые возвращают true или false) один раз для каждого элемента массива, и создаёт новый массив со всеми значениями, для которых функция callback вернула значение, которое может быть приведено к true. Метод не изменяет массив, для которого он был вызван.

Первым аргументом принимает функцию callback, которая вызывается с тремя аргументами:

* Текущий обрабатываемый элемент массива.
* Индекс текущего обрабатываемого элемента в массиве.
* Массив, по которому осуществляется проход.

Вторым аргументом принимает thisArg – необязательный параметр, значение которого будет использоваться в качестве this при вызове колбэк-функции переданной в первый аргумент.

**js\_32. Map что делает, как и когда использовать?**

Метод позволяет трансформировать один массив в другой при помощи функций-колбэка. Переданная функция будет вызвана для каждого элемента массива по порядку. Она должна возвращать какое-то значение. Именно это значение попадёт в итоговый трансформированный массив.

Первым параметром принимает функцию, вызываемую для каждого элемента массива, которая принимает три аргумента:

* Текущий обрабатываемый элемент массива.
* Индекс текущего обрабатываемого элемента в массиве.
* Массив, по которому осуществляется проход.

Вторым аргументом принимает thisArg – необязательный параметр, значение которого будет использоваться в качестве this при вызове колбэк-функции переданной в первый аргумент.

Метод map не изменяет массив, для которого он был вызван (хотя функция callback может это делать).

**js\_33. forEach что делает, как и когда использовать?**

Метод forEach() вызывает переданную функцию один раз для каждого элемента в массиве. Можно использовать метод, когда необходимо совершить одну и ту же операцию над всеми элементами массива.

Первым параметром принимает функцию, вызываемую для каждого элемента массива arr, которая принимает три аргумента:

* Текущий обрабатываемый элемент массива.
* Индекс текущего обрабатываемого элемента в массиве.
* Массив, по которому осуществляется проход.

Вторым аргументом принимает thisArg – необязательный параметр, значение которого будет использоваться в качестве this при вызове колбэк-функции переданной в первый аргумент.

Метод используется как альтернатива цикла for. Есть ряд преимуществ:

* Использование метода forEach() является декларативным способом обозначить операцию. С точки зрения читабельности кода это больше приближено к естественному языку.
* Удобнее обращаться к элементам, без необходимости каждый раз обращаться к массиву по индексу.

И недостатков:

* В forEach() не работают return, break и continue, а следовательно, нет возможности прервать или пропустить итерацию. Потому, если нужен какой-то из этих операторов, нужно использовать обычный цикл for.
* forEach() обрабатывает элементы массива в прямом порядке, нельзя пройти по массиву с конца.

**js\_33.1 Объясните разницу между методами forEach() и map() ?**

Оба метода принимают колбэк, который вызывается для каждого элемента. Разница в том, что метод forEach() ничего не возвращает, а метод map() возвращает новый массив с результатами вызова колбэка на каждом исходном элементе. Если переданный колбэк ничего не возвращает в новом массиве появится undefined. Можно вернуть значение и из колбэка для forEach() но оно никак не будет использоваться дальше.

Используя map() можно создавать цепочки вызовов. С forEach() так не получится.

**js\_34. some что делает, как и когда использовать?**

Метод позволяет узнать, есть ли в массиве хотя бы один элемент, удовлетворяющий какому-либо условию.

Первым параметром принимает функцию, вызываемую для каждого элемента массива arr, которая принимает три аргумента:

* Текущий обрабатываемый элемент массива.
* Индекс текущего обрабатываемого элемента в массиве.
* Массив, по которому осуществляется проход.

Вторым аргументом принимает thisArg – необязательный параметр, значение которого будет использоваться в качестве this при вызове колбэк-функции переданной в первый аргумент.

Функция будет вызываться для каждого элемента массива до тех пор, пока не вернётся true, либо пока не закончатся элементы массива.

Результатом вызова метода будет true или false. Если ни один элемент в массиве не удовлетворит условию, то результат будет false. Метод возвращает false при любом условии для пустого массива.

**js\_35. Every что делает, как и когда использовать?**

Метод позволяет узнать, удовлетворяют ли все элементы в массиве условию из функции-колбэка. Результатом будет true или false. Если хотя бы один элемент не будет удовлетворять условию, то результат будет false.

Первым параметром принимает функцию, вызываемую для каждого элемента массива arr, которая принимает три аргумента:

* Текущий обрабатываемый элемент массива.
* Индекс текущего обрабатываемого элемента в массиве.
* Массив, по которому осуществляется проход.

Вторым аргументом принимает thisArg – необязательный параметр, значение которого будет использоваться в качестве this при вызове колбэк-функции переданной в первый аргумент.

Метод вызывает переданную функцию один раз для каждого элемента массива до тех пор, пока не найдёт такой, для которого callback вернёт ложное значение (значение, становящееся равным false при приведении его к типу Boolean). Если такой элемент найден, метод немедленно вернёт false. В противном случае, если callback вернёт true для всех элементов массива, метод every() вернёт true.

**js\_36. Какие есть 4 шаблона вызова функции, которые задают контекст выполнения этой функции?**

Cуществует четыре пути вызова функций:

Вызов метода — Method Invocation

Вызов функции — Function Invocation

Вызов конструктора — Constructor Invocation

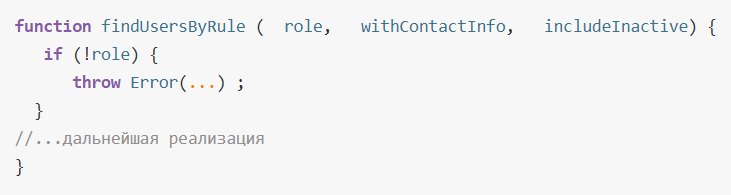
Вызов apply и call — Apply And Call Invocation

**js\_37. Какой наиболее простой паттерн, позволяющий облегчить читаемость функции, когда у нее огромное количество аргументов?**

1. Допустим в функции три параметра, если допустим чаще всего второй параметр не будет нужен, а будут нужны только первый и третий, тогда нам в любом случае придется всегда передавать все три.
2. Допустим в функции много параметров, и декларация функции находится где то в другом модуле, каждый раз придется искать модуль с декларацией функции чтобы понять для чего в неё передаются какие то значения - допустим три подряд булевых значения.

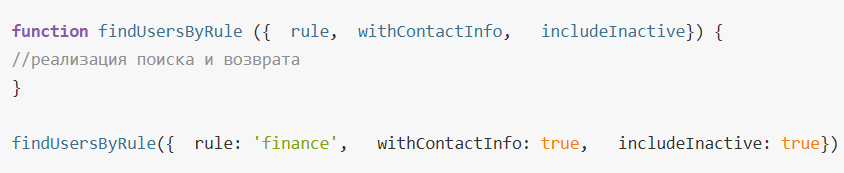
findUsersByRule( 'false', true, true)

1. Допустим, если нужно, чтобы часть параметров была обязательными, то придётся писать что-то вроде этого:



Получится, что функция, помимо своих обязанностей, будет выполнять ещё и валидацию. Конечно, функция может принимать в себя функции валидации, но тогда перечень входных параметров разрастётся. Это тоже минус такого паттерна кодирования.

Можно использовать деструктуризацию, и передавать в функцию вместо нескольких различных параметров объект, с одноименными свойствами.



1. Благодаря этому паттерну параметры выглядят гораздо менее двусмысленно, и их намного легче читать и понимать.
2. Пропуск или иной порядок параметров больше не является проблемой, т.к. они являются именованными свойствами объекта.
3. Т.к. деструктуризация копирует переданное значение, то его изменения в функции не повлияют на оригинал.

**js\_38. Как получить все аргументы функции (включая те, что не объявлены, но все-таки были переданы)?**

1. Rest parameters - остаточные параметры. Синтаксис остаточных параметров функции позволяет представлять неограниченное множество аргументов в виде массива.
2. Объект arguments. Это подобный массиву объект, который содержит аргументы, переданные в функцию.  Arguments доступная внутри любой (нестрелочной) функции.

Различия:

1. остаточные параметры включают только те, которым не задано отдельное имя, в то время как объект arguments содержит все аргументы, передаваемые в функцию;
2. объект arguments не массив, а остаточные это массив и ему доступны все методы массива.
3. у объекта arguments есть дополнительная функциональность, специфичную только для него (например, свойство callee).
4. Arguments есть только в нормальных функциях а rest параметры можно использовать в любой функции.

**js\_39. Что такое рекурсия? Когда удобно её использовать?**

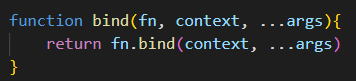
Ответ в отдельном файле.

**js\_40. Что такое замыкания, и в каких случаях они могут быть полезны?** **Как сохранить состояние с помощью замыкания, и для чего это состояние может быть использовано?**

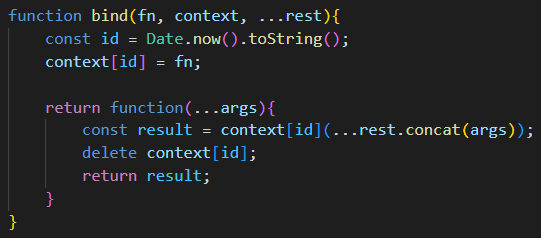
Ответ в отдельном файле.

**js\_41. Как реализовать функцию bind?**

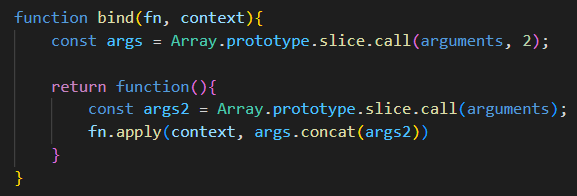
1. Простой способ, используя встроенный метод bind:



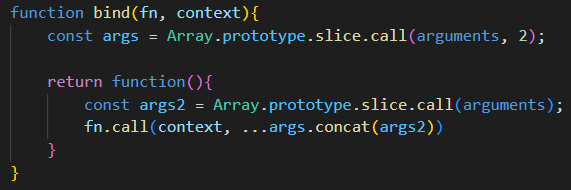
2. С использованием rest оператора:



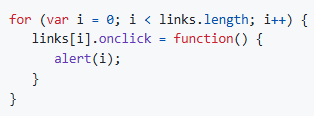
3. Используя apply:



4. Используя call:

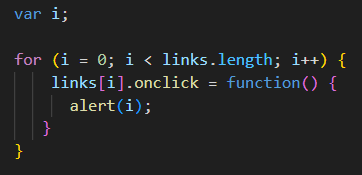


**js\_42. Решить такую вот проблему: пускай у нас есть массив ссылок, и наша задача — сделать так, чтобы при клике на каждую выводился alertом ее порядковый номер. Первое решение, что приходит в голову, выглядит так?**



**На деле же оказывается, что при клике на любую ссылку выводится одно и то же число — значение links.length. Почему так происходит и как эту гадость исправить?**

Проблема в том, что запрос переменной i происходит после завершения цикла. Так как объявления переменных var поднимаются в верхнюю часть области видимости, вышеприведённый код аналогичен следующему:

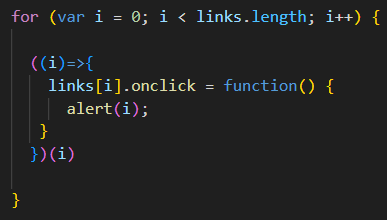


В результате, при запросе i, она уже равна links.length -1, в результате, во всех alert мы видим одно значение.

Самое простое решение использовать ключевое слово let. Теперь на каждой итерации цикла каждый alert, получает собственную копию i, из собственного лексического окружения, создаваемого на каждой итерации.

Ещё один способ решения этой проблемы, который часто применялся до появления стандарта ES6, когда ключевого слова let ещё не было, заключается в использовании IIFE.

При таком подходе значение i копируется в замыкании, а в алерт попадает i из собственного лексического окружения.



**js\_43. Что такое callback (функция обратного вызова)? Когда они обычно применяются?**

Функция, которая передается как аргумент, называется callback-функцией.

В JavaScript функции — это объекты. Поэтому функции могут принимать другие функции в качестве аргументов, а также возвращать функции в качестве результата. Функции, которые это умеют, называются функциями высшего порядка.

Часто используются для обработки результатов асинхронных операций. Например, обработка событий, загрузки данных с сервера и таймеры. В этих случаях callback функции позволяют задать, что именно должно произойти после завершения асинхронной операции.

Современные альтернативы — это Promises и async/await, которые предлагают более чистый и удобный синтаксис для работы с асинхронными операциями. Они позволяют избежать callback hell и делают код более читаемым.

**js\_44. Что такое каррирование?**

Каррирование или карринг (currying) в функциональном программирование — это преобразование функции с множеством аргументов в набор вложенных функций с одним аргументом. Можно сказать, что это процесс превращения функции с несколькими аргументами в функцию с меньшей арностью. Этот метод преобразования функции реализуется в JS благодаря замыканиям и благодаря тому, что функции в JS являются объектами первого класса (их можно передавать в качестве аргументов другим функциям, возвращать из них, присваивать переменным).

Арность — это количество аргументов функции. Если функция принимает два аргумента — это бинарная или 2-арная функция, три аргумента - тернарная, 3-арная функция.

При вызове каррированной функции с передачей ей одного аргумента, она возвращает новую функцию, которая ожидает поступления следующего аргумента. Новые функции, ожидающие следующего аргумента, возвращаются при каждом вызове каррированной функции — до тех пор, пока функция не получит все необходимые ей аргументы. Ранее полученные аргументы, благодаря замыканию, ждут того момента, когда функция получит всё, что ей нужно для выполнения вычислений. После получения последнего аргумента функция выполняет вычисления и возвращает результат.

Каррирование и частичное применение функций очень похожи друг на друга, но концепции это разные. При частичном применении функцию преобразуют в другую функцию, обладающую меньшим числом аргументов (меньшей арностью). Некоторые аргументы такой функции оказываются зафиксированными (для них задаются значения по умолчанию). При каррировании число вложенных функций равно числу аргументов исходной функции. Каждая из этих функций ожидает собственный аргумент. При этом понятно, что если функция аргументов не принимает, или принимает лишь один аргумент, то каррировать её нельзя.

В ситуации, когда функция имеет два аргумента, результаты её каррирования и частичного применения, можно сказать, совпадают.

Продвинутое каррирование:



Продвинутое каррирование с bind:



**js\_45 Что такое частичное применение?**

Частичное применение — возможность зафиксировать часть аргументов многоместной функции и создать другую функцию, меньшей арности. Этот метод преобразования функции реализуется в JS благодаря замыканиям и благодаря тому, что функции в JS являются объектами первого класса (их можно передавать в качестве аргументов другим функциям, возвращать из них, присваивать переменным).

Частичное применение может быть полезным например — при разработке модулей, подходящих для повторного использования. Например, есть функция, которая используется для вычисления суммы к оплате с учётом скидки:



Есть определённая категория клиентов, которой мы даём скидку в 10%:



При таком подходе, постоянно придётся вызывать эту функцию с двумя аргументами:



Исходную функцию можно привести к такому виду, который позволял бы получать новые функции с заранее заданным уровнем скидки, при вызове которых им достаточно передавать сумму покупки:



Функция tenPercentDiscount() представляет собой результат частичного применения функции discount(). При вызове tenPercentDiscount() этой функции достаточно передать цену, а скидка в 10%, то есть — аргумент discount, уже будет задана:



Если в нашем магазине имеются покупатели, которым решено дать скидку размером в 20%, то получить соответствующую функцию для работы с ними можно так:



Теперь функцию twentyPercentDiscount() можно вызывать для расчёта стоимости товаров с учётом скидки в 20%:



**js\_46 Что такое мемоизация?**

Мемоизация — сохранение результатов выполнения функций для предотвращения повторных вычислений. Это один из способов оптимизации, который применяется для увеличения скорости выполнения компьютерных программ.

Проще говоря, мемоизация — это запоминание, сохранение чего-либо в памяти. Функции, в которых используется мемоизация, обычно работают быстрее, так как при их повторных вызовах с одними и теми же параметрами, они, вместо выполнения неких вычислений, просто считывают результаты из кэша и возвращают их.

В JS этот механизм возможен, т. к.:

1. функции в JavaScript — это объекты первого класса, что позволяет использовать их как функции высшего порядка и возвращать из них другие функции.

2. Закэшированные значения сохраняются в переменной и могут хранить данные между вызовами функции, так как переменная определяется в замыкании.

Мемоизация — это разновидность кэширования. Обычно под кэшированием понимают широкий набор способов сохранения чего-либо для последующего использования. Например, это может быть HTTP-кэширование. Мемоизация же обычно означает кэширование возвращаемых значений функций.

Мемоизация имеет ограниченное применение:

• Для того, чтобы функцию можно было подвергнуть мемоизации, она должна быть чистой, всегда возвращать одни и те же значения в ответ на одни и те же аргументы.

• Мемоизация — это компромисс между производительностью и потреблением памяти. Мемоизация хороша для функций, имеющих сравнительно небольшой диапазон входных значений, что позволяет достаточно часто, при повторных вызовах функций, задействовать значения, найденные ранее, не тратя на хранение данных слишком много памяти.

Кэширование — более общий термин, потому что кэшировать можно любые данные, а мемоизировать — только неизменяемые.



**js\_47. Что такое функция-конструктор? Как их создавать и как ими пользоваться?**

Технически это обычные функциии. Но есть два соглашения:

- Имя функции-конструктора должно начинаться с большой буквы.

- Функция-конструктор должна выполняться только с помощью оператора "new".

Когда функция вызывается как new User(...):

- Создаётся новый пустой объект, и он присваивается this.

- Выполняется тело функции. Обычно оно модифицирует this, добавляя туда новые свойства.

Возвращается значение this.

Может использоваться любая функция кроме стрелочных функций, поскольку у них нет this.

Обычно конструкторы не имеют оператора return. Их задача – записать все необходимое в this, и это автоматически становится результатом.

Но если return всё же есть, то:

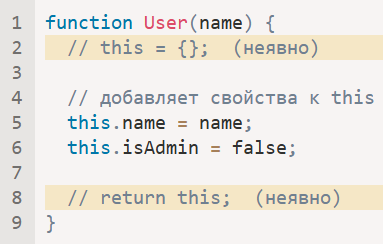
- При вызове return с объектом, вместо this вернётся объект.

- При вызове return с примитивным значением, оно проигнорируется.

Другими словами, return с объектом возвращает этот объект, во всех остальных случаях возвращается this.

Конструкторы используются для создания множества похожих объектов.

Вызов, new User(...) делает что-то вроде:



Таким образом, let user = new User("Jack") возвращает тот же результат, что и:



**js\_48. Что такое прототип? Какие возможности имеет/дает?**

Прототипы — это механизм, с помощью которого объекты JavaScript наследуют свойства друг от друга.

Прототип — это объект, который выступает как шаблон, от которого объект наследует методы и свойства. Объект-прототип так же может иметь свой прототип и наследовать его свойства и методы и так далее.

Возможности прототипов:

- прототип объекта можно изменить динамически во время выполнения программы.

- объекты могут наследовать свойства и методы от других объектов без необходимости копирования их данных, поэтому прототипное наследование обычно приводит к экономии памяти.

- прототипное наследование дает возможность вносить изменения в реальном времени: любые модификации прототипа немедленно применяются к уже существующим экземплярам.

**js\_49. Как работает свойство F.prorotype?**

"Prototype" это свойство функций. (Точно нет у стрелочных и еще каких-то, может async)

По умолчанию это простой объект с единственным свойством constructor, которое ссылается на саму функцию.

При вызове функции с помощью свойство конструктора new, объект из prototype станет прототипом созданного объекта.

Установка F.prototype = some\_object буквально говорит интерпретатору следующее: "При создании объекта через new F() запиши ему some\_object в [[Prototype]]".

**js\_50. Почему методы объекта лучше хранить в прототипе, а не в самом объекте?**

1. Хранение методов в прототипе позволяет всем экземплярам использовать один и тот же метод, что экономит память.
2. Если нужно изменить метод, хранящийся в прототипе, то это делается в одном месте, а если хранить методы в каждом экземпляре, то нужно обновлять их вручную для каждого объекта.
3. Хранение методов в прототипе позволяет отделить логику (методы) от данных (свойств), что улучшает структуру вашего кода.

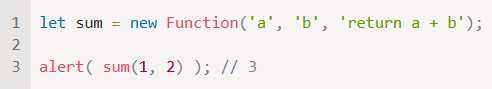
**js\_51. Можно ли создать инстанс функции через конструктора? И если да, то как, а если нет, то какой бы интерфейс вы реализовали бы для этой задачи?**

1. Можно только через конструктора new Function.

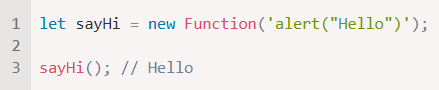
Синтаксис для объявления функции:



Функция создаётся с заданными аргументами arg1...argN и телом functionBody.



Можно создать функцию без аргументов, в этом случае достаточно указать только тело:



Обычно функция запоминает, где родилась, в специальном свойстве [[Environment]]. Это ссылка на лексическое окружение (Lexical Environment), в котором она создана.

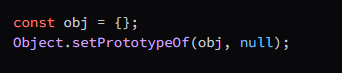
Но когда функция создаётся с использованием new Function, в её [[Environment]] записывается ссылка не на внешнее лексическое окружение, в котором она была создана, а на глобальное. Поэтому такая функция имеет доступ только к глобальным переменным.

2. Создать "инстанс" функции через обычный класс нельзя, т.к. класс может вернуть только объект.

**js\_52. Как создать объект, который ни от чего не наследуется?**

1. Object.create(null) – создаст новый объект без прототипа.

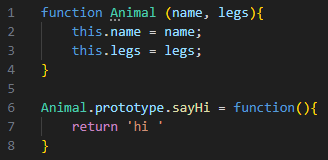
2. Создать объект через литерал, и затем удалить его прототип.



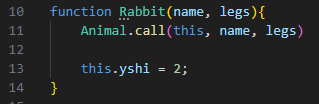
**js\_53. Какие 3 (как минимум) способа есть отнаследоваться в JavaScript-е? В чем отличия и нюансы.**

1. Наследование одной функции конструктора от другой.

Допустим есть функция конструктор:



Для того, чтобы Rabbit наследовался от Animal, для начала нам нужно вызвать конструктор Animal внутри Rabbit при помощи call, чтобы скопировать свойства из Animal

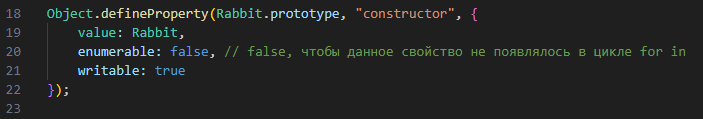


Сейчас мы скопировали только свойства, но не методы из Animal.prototype.

Чтобы отнаследовать методы, нужно скопировать Animal.prototype в Rabbit.prototype



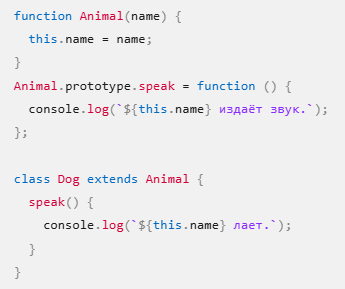
Теперь методы скопированы, но, теперь свойство Rabbit.prototype.constructor указывает на Animal, а должно указывать на сам Rabbit. По этому установим это свойство



Все, теперь все работает как положено.

2. Наследование классов

Классы могут наследоваться при помощи extends. Причем могут наследоваться как от классов, так и от конструкторов, но классы не могут расширять обычные объекты:



3. Object.setPrototypeOf(obj, prototype);

Метод Object.setPrototypeOf() устанавливает прототип (то есть, внутреннее свойство [[Prototype]]) указанного объекта в другой объект или null.

4. Object.create(proto[, descriptors])

Создаёт пустой объект со свойством [[Prototype]], указанным как proto (может быть null), и также в этом методе можно задать дескрипторы свойств вторым необязательным параметром.

5. Свойство объекта \_\_proto\_\_

Свойство \_\_proto\_\_ — исторически обусловленный геттер/сеттер для [[Prototype]]

Значение \_\_proto\_\_ может быть объектом или null. Другие типы игнорируются.

Свойство \_\_proto\_\_  устарело, и существует по историческим причинам

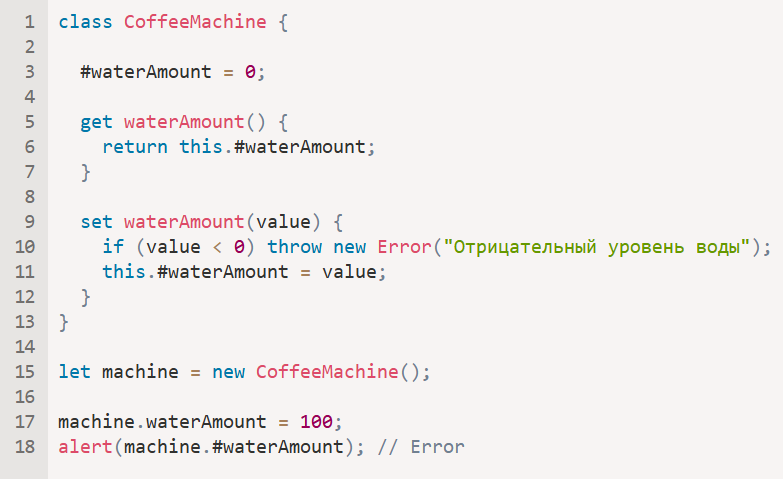
По спецификации \_\_proto\_\_ должен поддерживаться только браузерами, но по факту все среды, включая серверную, поддерживают его.

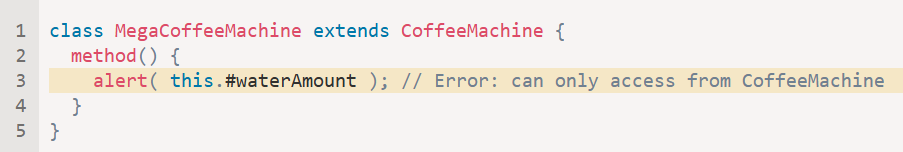
**js\_54. Как объявить приватное поле класса в JS?**

Использовать префикс хэша #.

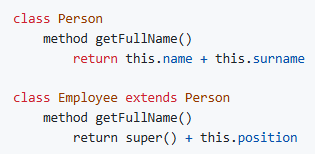
К приватному полю нельзя получить доступ из экземпляра, или наследуемого класса.

Приватные поля не конфликтуют с публичными. Может существовать два поля одновременно – приватное #name и публичное name.

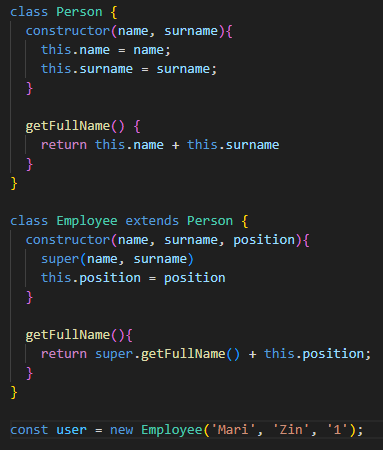




**js\_54. Как в переопределенном методе у наследующего класса вызвать переопределяемый метод родительского? Пример псевдокода:**



Через super:



**js\_54. Какие есть способы навсегда привязать метод класса к его инстансу (чтобы this всегда был текущим экземпляром класса)?**

|  |  |
| --- | --- |
| 1. Использовать стрелочную функцию для метода: | 2. Использовать обычную функцию, и явно привязать ее методом bind: |

**CSS**

**css\_4. Что такое padding?**

Свойство padding — это внутренний отступ — позволяет оттолкнуть контент от границ родительского элемента.

В качестве значения можно указать одно, два, три или четыре числа. Браузер будет читать это значение по-разному в зависимости от количества указанных значений.

* padding: 10px 15px 20px 30px — читается по часовой стрелке начиная с верхней границы. Сверху—> справа—> снизу—> слева.
* padding: 15px — для всех четырёх сторон 15 px.
* padding: 15px 20px — Первое значение для верхнего и нижнего отступов. Второе значение для боковых отступов.
* padding: 10px 5px 25px — первое значение для верхнего отступа, второе значение для боковых, последнее для нижнего отступа.

Вертикальные отступы не схлопываются так как margin.

**css\_14. Что делает border и какие значения может принимать?**

Свойство border отвечает за отрисовку рамки - видимой границы блока. Само свойство border является шорткатом (короткой записью) и объединяет в себе значение для нескольких полных свойств: border-width, border-style, border-color. Значения можно писать в любом порядке, интерпретатор браузера разберётся. Главное, разделять их пробелами. Но общепринятым является порядок: ширина → стиль → цвет рамки. Можно рисовать фигуры рамками.

**css\_14.1 border-style**

Задает стиль рамки элемента. Свойство border-style является шорткатом для свойств: border-bottom-style, border-left-style, border-right-style, border-top-style. Может принимать значения:

* none — отключает рамку (значение по умолчанию).
* solid — сплошная рамка. Этот стиль используется чаще всего.
* dotted — рамка состоит из точек с пробелами между ними.
* dashed — рамка состоит из коротких чёрточек с пробелами между ними.
* double — рамка состоит из двух сплошных линий с небольшим нерегулируемым отступом между ними.

Есть еще и другие варианты рамок объемные с тенями.

**css\_14.2 border-color**

Управляет цветом рамки. Можно писать одно, два, три или четыре значения, разделяя их пробелами. В зависимости от этого получится разный результат:

Одно значение - цвет рамки со всех сторон одинаковый.

Два значения - первое устанавливает цвет рамки сверху и снизу, второе — слева и справа.

Три значения - первое значение устанавливает цвет рамки сверху, второе — слева и справа, а третье — снизу.

Четыре значения — четыре значения, устанавливает для каждой стороны свой цвет, поочерёдно: для верхней, правой, нижней и левой рамки.

Значение по умолчанию соответствует значению color у элемента.

**css\_14.3 border-width**

Управляет толщиной рамки элемента. Значением может быть число с любыми единицами измерения, доступными в вебе, кроме процентов. Разрешается указывать одно, два, три или четыре значения. Для задания толщины можно использовать переменные - thin, medium или thick, значение переменных определяется браузерами.

**css\_14.4 border-radius**

Задаёт радиус закругления каждого из углов элемента. Значение в процентах будет высчитываться от размеров элемента: горизонтальные размеры будут высчитываться от ширины элемента, а вертикальные — соответственно, от высоты. Это свойство является шорткатом для четырёх свойств border-top-left-radius, border-top-right-radius, border-bottom-right-radius, border-bottom-left-radius.

Свойство border-radius может быть задано как:

одно, два, три или четыре значения - для задания обычного радиуса углов.

одна, две, три или четыре пары значений разделённые "/". Используется для задания эллиптического скругления.

**css\_14.5 Можно ли задать стили для каждой рамки отдельно (например, отдельно для левой и для правой рамок)?**

Можно! При помощи следующих свойств: border-top, border-right, border-bottom, border-left. Все четыре рамки могут быть разными по толщине, цвету и стилю.

**React**

**rct\_1. Что такое React?**

Это JavaScript-библиотека для создания пользовательских интерфейсов.

**rct\_2. Что такое декларативное UI программирование? Чем оно отличается от императивного подхода?** 

Императивное программирование — оно же «приказывающее» или «повелительное» — это парадигма (стиль написания исходного кода компьютерной программы), в которой описываются последовательные команды, которые должен совершить процессор, чтобы получить необходимый нам результат. Т.е. это такой стиль программирования, при котором описывается, как добиться желаемого результата.

Языки: C, C++;

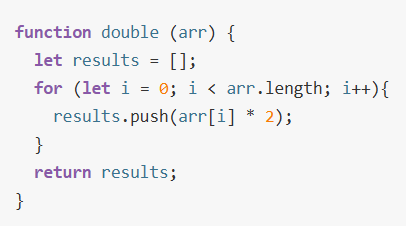
Декларативное программирование -  это парадигма(стиль написания исходного кода компьютерной программы), программирования, в которой описывается ожидаемый результат, а не способ его получения.

Языки: SQL, HTML.

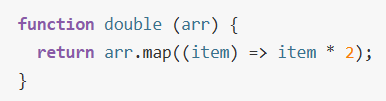
Неважно, как будут выбраны пользователи из Мексики и как браузер распарсит article. Важно, что мы получим мексиканских пользователей и новый header и paragraph на сайте не углубляясь в инструкции.

Например: Дан массив чисел, написать функцию, которая вернет массив, где каждое число -> [1, 2, 3] -> [2, 3, 6]

Императивный стиль:



Декларативный стиль:



В реализациях декларативных компонентов используется идея, в которой есть функция, зависящая от состояния. Результат такой функции — новый вариант верстки: **func render (context) → UI.**

**rct\_3. Что такое React Element? Как его можно создать? Из чего он состоит?** 

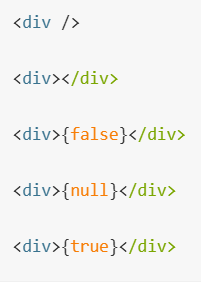
React-элемент описывает узел DOM в виде javascript-объекта. Объект состоит из свойств type(тип тега), children(дочерние узлы), props(атрибуты) и др. React-элемент — это не то, что будет на экране, а описание того, что будет на экране.

Создать можно:

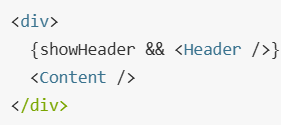
1. React.createElement(elementName, data, children)
2. Используя синтаксис JSX.

**rct\_3.1 Что будет, если передать булевые, null или undefined значения в children?**

Они игнорируются. Их можно использовать в качестве потомков, но данные значения не отображаются при рендеринге. Следующие JSX выражения будут отображены одинаково:



Вместе с условными выражениями это можно использовать для условного отображения элементов React:



**rct\_3.2 Что такое React Component?** 

React-компонент — это функция или класс, который принимает входные данные (опционально) и возвращает React-элемент — то есть, javascript-объект. Пользовательский интерфейс создается с помощью комонентов.

**rct\_4 Что такое JSX и зачем он нужен?**

JavaScript XML (JSX) — это синтаксическое расширение JavaScript предоставляющее «синтаксический сахар» для вызовов функций и построения объектов, особенно React.createElement() (иначе говоря, он упрощает процесс написания кода, сохраняя его функциональность). JSX ускоряет создание элементов React, делая их более компактными, а также упрощает их чтение. JSX создан только для разработчиков. Сам по себе он никак не ускоряет и не улучшает веб-приложения. JSX преобразуется в тот же код, который получается и без JSX.

Основные плюсы:

- Улучшенное взаимодействие с разработчиком (DX, Developer Experience) — код проще читать благодаря XML-подобному синтаксису, который хорошо подходит для написания вложенных декларативных структур.

- Улучшенные сообщения об ошибках — React по умолчанию предполагает использование JSX и выводит полезные сообщения об ошибках. Если JSX не используется, эти сообщения будут запутывать разработчика, так как в них будет упоминаться не тот синтаксис, который он реально использует.

- Ускорение кода — при преобразовании JSX в JavaScript транспилятор одновременно оптимизирует код, благодаря чему итоговый код JavaScript исполняется быстрее, чем если бы он был введен вручную.

- Повышение эффективности работы команд — неопытным разработчикам (например, веб-дизайнерам) проще менять код, потому что JSX напоминает разметку HTML, которая им уже знакома.

- Снижение количества синтаксических ошибок — разработчики набирают на клавиатуре меньше кода, а это значит, что они сделают меньше ошибок.

Процесс преобразования JSX в JavaScript, называется транспиляция. Транспиляция, а не компиляцией, потому что код транслируется с одного исходного языка (JSX) на другой исходный язык (JavaScript). В свою очередь, полученный код JavaScript интерпретируется «реальным» компилятором, который выполняет код. Транспиляция — всего лишь преобразование синтаксиса, а не интерпретация кода.

При выполнении приложения React браузер видит только команды React.createElement, необходимые для генерирования нужной структуры. JSX существует только в редакторе. Транспилятор преобразует файлы, содержащие JSX, в «чистый» JavaScript с многочисленными вызовами React.createElement().

Процесс транспиляции заключается в следующем: исходный код переписывается, чтобы получить такие же результаты, но уже с помощью синтаксиса, понимаемого устаревшими браузерами. Любой новый синтаксис для браузера, который его не поддерживает, просто недействителен и выдает ошибку разбора.

**rct\_x Что такое stateful и stateless компоненты?**

**rct\_x.x Что делает хук useState? Как обновить состояние?**

Позволяет добавлять состояние в функциональные компоненты. React хранит это состояние между рендерами. Вызов useState возвращает массив с двумя элементами - текущее значение состояния и функцию для его обновления. Единственный аргумент useState — это начальное состояние. Чтобы обновить состояние нужно вызвать функцию обновления состояния с новым значением или коллбэком. После этого React сделает повторный рендер, в котором использует уже новое значение.