**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 множество других структур.

**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. Во время работы с такой структурой компьютеру необходимо следить за тем, сколько памяти уже есть, сколько понадобится, и выделять новую. Работать с такими данными сложнее. 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?**

Прямое изменение данных объекта.

**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** является результатом неправильной математической операции, например если разделить строку на число результатом будет **NaN.**

**NaN** «прилипчиво» - 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).

Т.к. NaN не является равным ничему, даже самому себе нельзя просто сравнить NaN === NaN. 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?**

Литералы используются для представления значений в JavaScript. Литерал в программировании — это прямое указание значения в коде, например, число 42 или текст "привет". Эти значения не меняются во время работы программы. Они как фиксированные константы, помогающие сделать код понятнее.

Создавать переменную через конструктор, это значит что для создания исполльзуется конструктор new. Конструктор всегда вернет объект.

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

Например, typeof new Number(42) === "object" и new Number(42) !== 42 (при этом new Number(42) == 42).

При String()вызове в качестве конструктора (с помощью new) он приводится value к строковому примитиву и возвращает String объект-оболочку, который не является примитивом.

const a = new String("Hello world"); // a === "Hello world" is 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?**

Если статический ключ - использовать точечную, если ключ динамический, вычисляемый из выражения во время выполнения - использовать скобочную.

Т.к. Dot Notation допускает только статические ключи, а Bracket Notation допускает динамические ключи. Статический ключ здесь означает, что ключ вводится напрямую, тогда как Динамический ключ здесь означает, что ключ вычисляется из выражения.

Точечная нотация полезна, когда свойство известно зараннее, и она менее громоздкая, быстрее пишется.

Нотация скобок полезна, нужно динамически получить доступ к свойству. Ключ этого свойства может быть получен из выражений типа getKey(), "my" + "key", или keyVariable.

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

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(). 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?**

Использовать null для обозначения отсутствия значения, тогда как undefined следует рассматривать как результат неинициализированной переменной. 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 означает, что переменная не определена.

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

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;

JavaScript различает **выражения** **(expressions)** и **инструкции** **(statements)**. Инструкция — это (грубо говоря) команда, действие.

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 текущей 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.

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

- Переназначения могут стать причиной багов – можно не заметить, если переменная переназначится, и это может оказаться причиной багов. Особенно в замыканиях. Const дает вам уверенность в том, что вы всегда будете видеть одно и то же значение.

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

**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,и он будет перебирать не только собственные свойства объекта, но и перечисляемые свойства, наследуемые от цепочки прототипов. Перечисляемые свойства объекта — это свойства, которые явно помечены такими.

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.

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:

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());**

Array.of() — метод, который создаёт массив из переданных ему аргументов независимо от их количества.

Конструктор массива Array() и Array.of() работают практически одинаково. Разница возникает, если передавать этим функциям один аргумент, причём этот аргумент должен быть числом. В этом случае для конструктора Array() аргумент будет считаться длинной массива, а для Array.of() — первым элементом массива.

Array.from() - метод создаёт новый массив на основе переданного объекта. Объект должен быть либо массивоподобным (как строка или объект arguments), либо итерируемым (как Set или Map). Принимает три аргумента, последние два из которых 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:

const nums = Array.from({length: 4}, function(value, index) {

// value будет undefined

return index \* 2

})

console.log(nums)

// [0, 2, 4, 6]

**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. Возвращает массив из удалённых элементов. Допускается использование отрицательных индексов. Они определяют позицию с конца массива.

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.

**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.

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

Если уменьшить length – то массив обрежется. Когда вы расширяете массив, изменяя его свойство length, реальное количество элементов в массиве увеличивается; например, если установить свойство length в 3, когда оно равно 2, массив будет из 3 элементов, где значение третьего элемента будет равно undefined .

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

Смотря как удалить. Splice, shift, pop изменит длину массива, delete не изменит.

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

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

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

Возвращает новый массив, содержащий копию части исходного массива

**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.

**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.

**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\_--. 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\_ 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\_ Что такое мемоизация?**

Мемоизация — сохранение результатов выполнения функций для предотвращения повторных вычислений. Это один из способов оптимизации, который применяеется 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\_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:

- возможность динамически изменять прототипы объектов во время выполнения программы.

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

- объекты могут наследовать свойства и методы непосредственно от других объектов, а не от классов и экземпляров. В классическом подходе создается иерархия классов, где подклассы наследуют свойства и методы от своих родительских классов. В прототипном наследовании объекты могут наследовать свойства и методы от любого другого объекта, независимо от того, какой у него происхождение.

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

У каждой функции (за исключением стрелочных) по умолчанию есть свойство "prototype". По умолчанию это объект с единственным свойством constructor, которое ссылается на саму функцию.

При вызове функции с оператором new, в момент вызова new установит объект из prototype в качестве прототипа для нового объекта.

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

**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.

**ss\_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.