**JavaScript**

**Вопросы**  
2) Как клонировать, копировать массив и потом работать с копией, при этом оригинал без изменения  
let fruits = [];  
let arr = fruits;  
Ob  
  
3)

4) не проходили **???.sort()** – сортирует массив «на месте», а потом возвращает его.

**???.reduce(func, initial)** – вычисляет одно значение на основе всего массива, вызывая func для каждого элемента и передавая промежуточный результат между вызовами.  
  
5) В каких методах массивов нужно прописывать return, а в каких нет???

6) typeof Number(1) === 'number'; // никогда не используйте эту запись! ??почему?? **Записи уроков по JS** <https://skilldesk.starta.university/lessons>  
19.10 – много практических задач было  
26.10 – разбор ДЗ и практические задачи  
  
JS. Работа с переменными, условный оператор JS.  
Циклы и массивы  
JS Работа с объектами JS.  
Функции, методы  
Практика (Работа с JS)  
JS. DOM (поиск элементов и изменение значений)  
JS. DOM (создание элементов)  
JS. DOM (Обработка событий)  
JS. DOM (Работа со стилями)  
JS Обработка формы.  
Объект event  
Практика (DOM)знакомство с REACT и создание проекта  
REACT работа с компонентами и props  
REACT useStateREACT useState  
Практика (верстка сайта с React)  
REACT useEffect  
REACT useContext  
REACT useMemo useCallback  
Практика (хуки,клиент-сервер)  
REACT useForm  
REACT Роутинг  
Redux знакомство с библиотекой  
Redux разбор практического применения  
Практика (реализация интерфейса с React и Redux)

**SyntaxError** – нарушение правил языка, синтаксическая ошибка (запятая, скобка и т.д.)  
**ReferenceError** – неправильное имя, ошибочное название функции, перемеренной и т.д.  
**TypeError** – перепутатть одно с другим, описали переменную, а вызываем ее как функцию.  
**LogicError** – перепутать

**“use strict”** — это модель разработки дает возможность разработчикам использовать только ограниченный синтаксис и не допускать лишних ошибок. Прописываем это в первую строчку кода, а в самом проекте должен быть файл настроек **.jshintrc**

&& - AND

|| - OR  
"Debug"

Shift+F9 - zapysk v regime otladki;

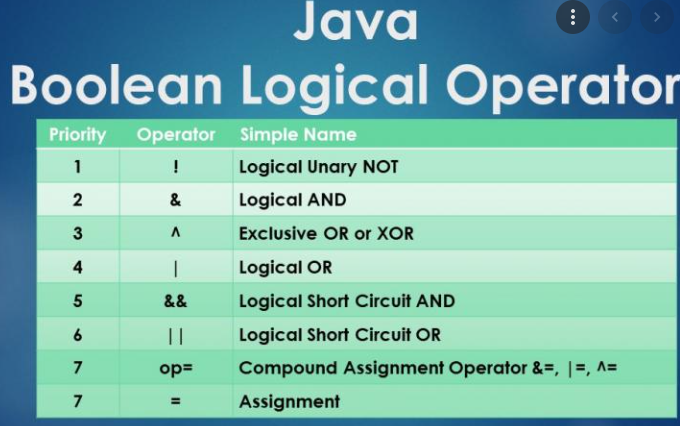
Ctrl+F8 - postavit 'break point';

F8 - one step;

F7 - one step(+metod);

F2 - pomenat znachenie vo vremia otladki;

Alt+F8 - vupolnit proizvolnii code;

|  |  |
| --- | --- |
|  | a &= b это a = a & b - операция Иa |= b это a = a | b - операция ИЛИ a ^= b это a = a ^ b - операция исключающее ИЛИ a += b это a = a + b a -= b это a = a - b |

**Операторы**

**Конкатена́ция** - операция склеивания объектов, например «микро» **+** «мир» == «микромир».  
**Интерполяция** - операция склеивания объектов через **` …${переменная}…`** (кавычки где Ё и ~)  
**Присвоение –** знак **=  
Отрицание** – знак **!  
Сравнение** – знак **== ,** 2\*4 == ‘8’ – сравнение по значению (true) **Строгое сравнение** - знак **=== ,** 2\*4 === ‘8’ строгое сравнение (false), сравнивается значение и тип переменных

**Инкремент** – операция увеличения переменной на 1 (обычно в счетчиках и циклах  
**Декремент** – операция уменьшения переменной на 1 (обычно в счетчиках и циклах  
Знак +/- можно ставить перед переменной (префикс) или после (постфикс).  
**Тернарный оператор/аргумент** (num===50)?(выполнение1 при true):(выполнение при false)  
  
**Иттерация** – 1 шаг цикла, 1 элемент массива в цикле

**Взаимодейсвие с пользователем в браузере**  
 **alert (‘предупреждение’)** – всплывающее окно в браузере с каким-либо предупреждением   
**confirm (‘вопрос пользователю’)** – предполагается только ответ **yes (true) и no(false)**  
**promt (‘вопрос пользователю’, ‘placeholder’)** – предполагается ввод данных пользователем. По умолчанию результатом будет тип string. Если +promt(‘’,’’) – то тип результата будет number

**typeof** *operand* - возвращает строку, указывающую тип операнда.  
**document.write(разметка)** - пишет строку в поток документа, открытый с помощью **document.open()** и после записи потока в файл закрытый с помощью **document.close()**

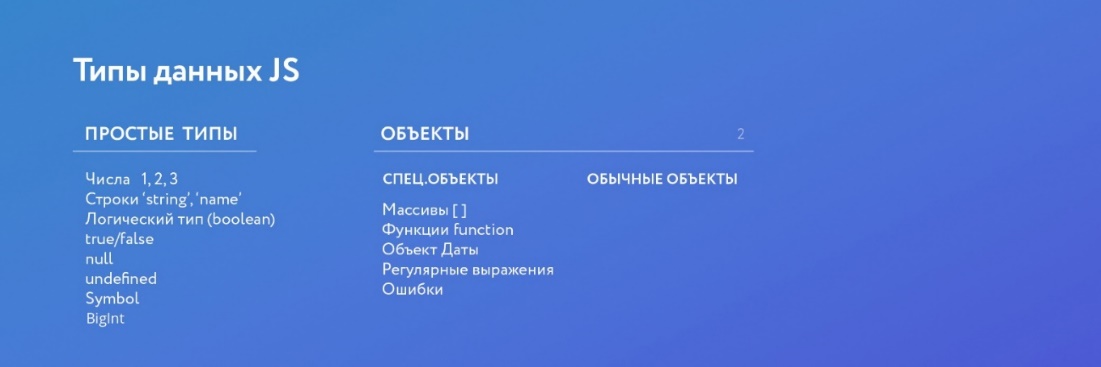
document.open(); - можно не использовать

document.write("<h1>Какой то заголовок</h1>");

document.close();

**Переменные**

**UpperCamelCase** – классы в Java (используется ли в JS?)  
**lowerCamelCase** – **переменные** с изменяемым значением  
**Upper\_Snake\_Case – константы** в JS  
**snake\_case** – для названия папок и файлов   
**PascalCase** – для названия **классов** в JS **Kebab-case** – для названия папок и файлов  
  
**Is not defined** – переменная не найдена (не объявлена)   
**undefined** – переменной не задано значение  
**Infinity** - бесконечноое значение (пример 4/0)

\ - экранирование переменной. \n - перевод строки \t – табуляция  
**Infinity** – бесконечное число, **true/false** – используется в циклах  
**NaN** является значением, представляющим не-число (Not-A-Number).  
**null** – пустое значение переменной, объявляется при объявлении переменной.   
**undefined** – -когда переменная не была инициализирована или не определилась  
**${b}** – вставка в текст значения переменной, текст заключается в обратные апострофы ` `  
let b = 10; console.log (‘Он сказал ${b} раз Привет’);  
 **false – это всегда: 0, null, ‘’, undefined, NaN  
true –** все остальные значения

**Условные инструкции if else и тернарный оператор**

**Конструкция иначе если:  
if** (car.color == 'red') {

console.log('Автомобиль красный');

}**else if** (car.color == 'green') {

console.log('Автомобиль зеленый');

}**else**{

console.log('Цвет не определен');

}

}

**Оператор отрицания:**if(!true) {  
 console.log('Истина');  
}

**Тернарный оператор, присвоение значения переменной:**let color = car.color == 'red' **?** 'Красный' **:** 'Зеленый';  
console.log(color);

**||** - оператор «или» **&&** - оператор «и»

**Функции  
DRY – don’t repeat youself**

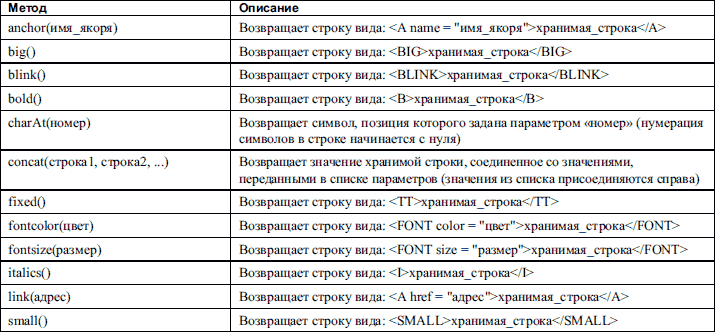
 **DRY – don’t repeat youself**  
function(item, index, array){} можно переделать в стрелочную (item, index, array)=>{}

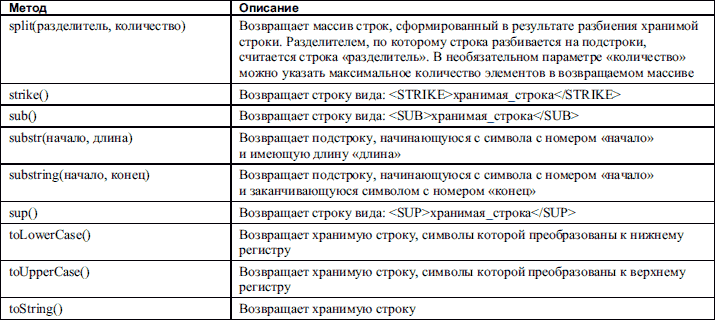
let car = {

maxSpeed: 250,

};    
speedChange(300);   
function speedChange(theSpeed) {  
 console.log('Скорость, которая была=', car.maxSpeed);  
 car.maxSpeed = theSpeed;     
console.log('Скорость, которая стала=', car.maxSpeed);   
};

**Строки. Методы в строках**





**repeat** - Принимает в качестве параметра число и повторяет строку указанное количество раз.

"proger ".repeat(3); // "proger proger proger "

**includes -** Проверяет, содержит ли строка указанную подстроку. Возвращает значение true или false. Вторым параметром можно указать позицию в строке, с которой следует начать поиск.

// "Hello proger".includes("proger"); // true

**indexOf** - Возвращает индекс первого найденного вхождения указанного значения. Поиск ведётся от начала до конца строки. Если совпадений нет, возвращает -1. Вторым параметром можно передать позицию, с которой следует начать поиск.

// "Hello World".indexOf("o"); // 4 "Hello World".indexOf("o", 5); // 7

**search** - Проверяет, есть ли в строке указанное значение и возвращает индекс начала совпадения.

// "hi, hello, hey".search("hello"); // 4

**slice** - Извлекает часть строки и возвращает новую строку. Обязательный параметр — начало извлечения. В которым параметром можно установить границу (по умолчанию — до конца строки).

// "Методы строк - как извлечь подстроку".slice(16); // как извлечь подстроку

// "Методы строк - как извлечь подстроку".slice(16, 21); // как из

// "Методы строк - как извлечь подстроку".slice(-7); // дстроку

**substr** - Извлекает часть строки указанной длины. Первым параметром принимает стартовую позицию, вторым — длину. Значение первого параметра может быть отрицательным, тогда позиция определяется с конца строки.

// let str = "Методы строк - как извлечь подстроку".substr(7, 5);

// let str = "Методы строк - как извлечь подстроку".substr(-7, 5);

// console.log(str)

**replace** - Ищет в строке указанное значение и возвращает новую строку, в которой выполнена замена на второй параметр. Можно заменить найденные значения другой строкой или передать функцию для работы над совпадениями.

// "hi, hello, hi".replace("hi", "hey"); // "hey, hello, hi"

**replaceAll** - Даёт такой же результат, как метод replace() с глобальным флагом g. Заменяет все найденные совпадения другой строкой или переданной функцией.

// "hi, hello, hi".replaceAll("hi", "hey"); // "hey, hello, hey"

let stroka = ‘Hello World’  
stroka.**toLowerCase()** – переводит все слово в нижний регистр.  
stroka.**toUpperCase()** - переводит все слово в верхний регистр.  
Можно перевести отдельную букву, например stroka**[4].toUpperCase()  
  
Массивы. Методы в объектах и массивы**Методы перезаписывают исходный массив. Если мы хотим первоначальный оставить без изменений, то перед выполнением методов нужно создать переменную и скопировать туда массив.

**Добавление/удаление в массиве**

**.push( ) -** добавляет элемент в конец массива, т.е. на позицию arr[arr.length]. Через return возвращает новую length массива.  
 **.unshift( )** - добавляет элемент в начало массива т.е. на позицию arr[0].

**.pop( ) –** удаляет элемент в конце массива. Удаленный элемент можно сохранить в переменную и вернуть через return.   
 **.shift( ) -** удаляет элемент в начале массива, сдвигая очередь, так что второй элемент становится первым. Удаленный элемент можно сохранить в переменную и вернуть через return.

**.splice()** – удаление элемента из массива // arr.splice(index[, deleteCount, elem1, ..., elemN])

Начиная с позиции index, удаляет deleteCount элементов и вставляет elem1, ..., elemN на их место  
 **.slice** - возвращает новый массив, в который копирует элементы, начиная с индекса start и до end (не включая end) - arr.slice([start], [end])

let arr = ["t", "e", "s", "t"];

let arr2 = arr.slice(1, 3); // e, s (копирует с 1 до 3)  
 let arrClone = arr.slice() // создаёт копию массива arr

**.concat()** – конкатенация (склеивание) - объединение массивов в новый   
 let newArr = arr.concat(arr2, arr3)

**Разбивка и соединение массива**

**.split()** и **.join()** - разбивка и соединение строк в массивы

У метода split есть необязательный второй числовой аргумент - ограничение на количество элементов в массиве. Если их больше, чем указано, то остаток массива будет отброшен.

// let arr = 'Вася, Петя, Маша, Саша'.split(', ', 2); // Вася, Петя

**Поиск и перебор элементов в массиве.**

**.includes()** - Поиск в массиве.   
 array.includes(item, from) – ищет item, начиная с индекса from, и возвращает true, если поиск  
 успешен, где from – не обязательный элемент и тогда поиск по всему массиву

let arr = [1, 0, 5, 8]; arr.includes(1) // true

**.find( ) -** углубленный поиск в массиве. Если true - возвращается найденный элемент и перебор прерывается, если все итерации оказались ложными, возвращается undefined

let result = arr.find(function(item, index, array) {… делать что-то с item или index…});

let user = users.find(

 (item, i ) => i == 1 // if true return item (

)[0]; // возвращает 0 элемент из item, если он представляет собой массив

**.findIndex()** похож на find, но возвращает индекс вместо значения.

**.indexOf/lastIndexOf**(item, pos) – ищет item, начиная с позиции pos, и возвращает его индекс или -1, если ничего не найдено. Старые методы, почти не используются

**.filter()** - Синтаксис этого метода схож с find, но filter возвращает массив из всех подходящих элементов. Если true - элемент добавляется к результату в массив, и перебор продолжается и возвращается пустой массив в случае, если ничего не найдено

let results = arr.filter(function(item, index, array){ … делать что-то с item или index…});

**Перебор массива**

**.forEach() -** перебор элементов, позволяет запускать функцию для каждого элемента массива. Перебирает массив, но не создает новый массив и ничего не возвращает  
 array.forEach(function(item, index, array){… делать что-то с item или index…});  
 **Преобразование массива.   
  
.map()** – **самый популярный**, вызывает функцию для каждого элемента массива и возвращает новый созданный массив результатов выполнения этой функции.

let result = arr.map(function(item, index, array) {…} **return item** );

let lengths = ["Bilbo", "Gandalf", "Nazgul"].map(item => item.length); // преобразуем каждый элемент в его длину:

**???.sort()** – сортирует массив «на месте», а потом возвращает его.

**.reverse()** – «на месте» меняет порядок следования элементов на противоположный и возвращает изменённый массив.

**???.reduce(func, initial)** – вычисляет одно значение на основе всего массива, вызывая func для каждого элемента и передавая промежуточный результат между вызовами.

**Дополнительно:**

**Array.isArray**(arrBlablabla); // проверяет, является ли arr массивом.

**Остальные методы**

**.at()** - последний элемент в массиве arr.at(-1), т.е. как бы arr.length() - 1

.constructor() -

.copyWithin() -

.entries() -

.every() -

.fill() -

.findIndex() -

.findLast() -  
.findLastIndex() -

.flat() - ƒ flat()

.flatMap() -

.includes() -

.indexOf() -

.keys() - ƒ keys()

.reduceRight( ) -

.some() - ƒ some()

.sort() - ƒ sort()

.toLocaleString() - ƒ toLocaleString()

.toString() - ƒ toString()

.values() - ƒ values()

.Symbol(Symbol.iterator): ƒ values()

.Symbol(Symbol.unscopables): {at: true, copyWithin: true, entries: true, fill: true, find: true, …}

[[Prototype]]: Object

constructor: ƒ Object()

hasOwnProperty: ƒ hasOwnProperty()

isPrototypeOf: ƒ isPrototypeOf()

propertyIsEnumerable: ƒ propertyIsEnumerable()

toLocaleString: ƒ toLocaleString()

toString: ƒ toString()

valueOf: ƒ valueOf()

\_\_defineGetter\_\_: ƒ \_\_defineGetter\_\_()

\_\_defineSetter\_\_: ƒ \_\_defineSetter\_\_()

\_\_lookupGetter\_\_: ƒ \_\_lookupGetter\_\_()

\_\_lookupSetter\_\_: ƒ \_\_lookupSetter\_\_()

\_\_proto\_\_: (...)

get \_\_proto\_\_: ƒ \_\_proto\_\_()

set \_\_proto\_\_: ƒ \_\_proto\_\_()

**Синтаксический сахар - деструктуризация массива**

// let [item1 = default, item2, ...rest] = array

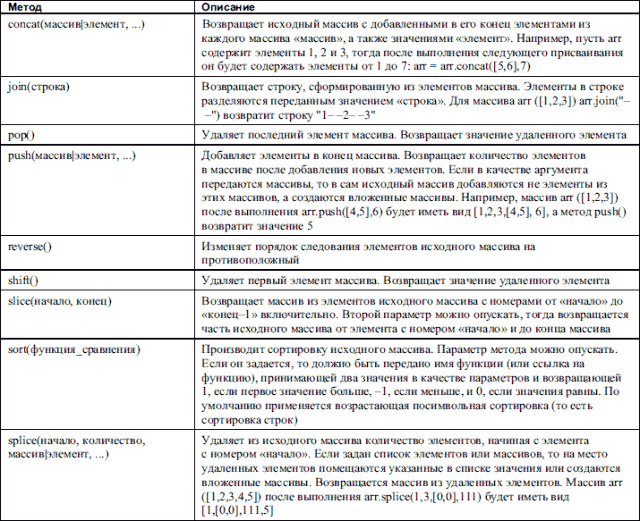
Первый элемент отправляется в item1, второй отправляется в item2, все остальные элементы попадают в массив rest. // «остаточные параметры» – троеточие ("..."):

let [name1, name2, ...rest] = ["Julius", "Caesar", "Consul", "of the Roman Republic"];

// console.log(name1); // Julius

// console.log(name2); // Caesar

// console.log(rest); // ["Consul", "of the Roman Republic"]



### **1. some()**

Этот метод проверяет, удовлетворяет ли какой-либо элемент массива условию, заданному в передаваемой функции. Он вернет значение true, если хотя бы один элемент совпадет с проверяемой функцией, и значение false — если нет.

**const** myAwesomeArray = ["a", "b", "c", "d", "e"]

myAwesomeArray.some(test => test === "d")

//-------> Output : true

**2. reduce()**Этот метод принимает функцию, которая имеет в качестве аргумента аккумулятор и значение. Он применяет функцию к аккумулятору и каждому значению массива, чтобы в результате вернуть только одно значение.

**const** myAwesomeArray = [1, 2, 3, 4, 5]

myAwesomeArray.reduce((total, value) => total \* value)

// 1 \* 2 \* 3 \* 4 \* 5

//-------> Output = 120

**3. every()**Этот метод проверяет, удовлетворяют ли все элементы массива условию, заданному в передаваемой функции. Он вернет значение true, если каждый элемент совпадет с проверяемой функцией, и значение false — если нет.

**const** myAwesomeArray = ["a", "b", "c", "d", "e"]

myAwesomeArray.every(test => test === "d")

//-------> Output : false

**const** myAwesomeArray2 = ["a", "a", "a", "a", "a"]

myAwesomeArray2.every(test => test === "a")

//-------> Output : true

### **4. map()**

Этот метод принимает функцию в качестве параметра и создает новый массив с результатом вызова указанной функции для каждого элемента массива. Он всегда будет возвращать одинаковое количество элементов.

**const** myAwesomeArray = [5, 4, 3, 2, 1]

myAwesomeArray.map(x => x \* x)

//-------> Output : 25

// 16

// 9

// 4

// 1

**5. flat()**

Этот метод принимает в качестве аргумента массив массивов и сглаживает вложенные массивы в массив верхнего уровня. Обратите внимание, что этот метод работает только для одного уровня.

**const** myAwesomeArray = [[1, 2], [3, 4], 5]

myAwesomeArray.flat()

//-------> Output : [1, 2, 3, 4, 5]

### **6. filter()**

Этот метод принимает функцию в качестве параметра и возвращает новый массив, содержащий все элементы массива, для которого функция фильтрации передавалась в качестве аргумента, и возвращает ее со значением true.

**const** myAwesomeArray = [

{ id: 1, name: "john" },

{ id: 2, name: "Ali" },

{ id: 3, name: "Mass" },

{ id: 4, name: "Mass" },

]

myAwesomeArray.filter(element => element.name === "Mass")

//-------> Output : 0:{id: 3, name: "Mass"},

// 1:{id: 4, name: "Mass"}

**7. forEach()**  
Этот метод применяет функцию к каждому элементу массива.

**const** myAwesomeArray = [

{ id: 1, name: "john" },

{ id: 2, name: "Ali" },

{ id: 3, name: "Mass" },

]

myAwesomeArray.forEach(element => console.log(element.name))

//-------> Output : john

// Ali

// Mass

**8. findIndex()**  
Этот метод принимает функцию в качестве параметра и в дальнейшем применяет ее к массиву. Он возвращает индекс найденного элемента, если элемент удовлетворяет условию проверяющей функции, переданной в качестве аргумента. Если не удовлетворяет, возвращается –1.

**const** myAwesomeArray = [

{ id: 1, name: "john" },

{ id: 2, name: "Ali" },

{ id: 3, name: "Mass" },

]

myAwesomeArray.findIndex(element => element.id === 3)

//-------> Output : 2

myAwesomeArray.findIndex(element => element.id === 7)

//-------> Output : -1

**9. find()**

Этот метод принимает функцию в качестве аргумента и в дальнейшем применяет ее к массиву. Он возвращает значение элемента, найденного в массиве, если элемент удовлетворяет условию проверяющей функции. В противном случае оно возвращается со значением undefined.

**const** myAwesomeArray = [

{ id: 1, name: "john" },

{ id: 2, name: "Ali" },

{ id: 3, name: "Mass" },

]

myAwesomeArray.find(element => element.id === 3)

//-------> Output : {id: 3, name: "Mass"}

myAwesomeArray.find(element => element.id === 7)

//-------> Output : undefined

**10. sort()**

Этот метод принимает функцию в качестве параметра. Он сортирует элементы массива и возвращает их.

**const** myAwesomeArray = [5, 4, 3, 2, 1]

// Sort from smallest to largest

myAwesomeArray.sort((a, b) => a - b)

//-------> Output : [1, 2, 3, 4, 5]

// Sort from largest to smallest

myAwesomeArray.sort((a, b) => b - a)

//-------> Output : [5, 4, 3, 2, 1]

**11. concat()**

Этот метод объединяет два или более массива/значения и возвращает новый массив.

**const** myAwesomeArray = [1, 2, 3, 4, 5]

**const** myAwesomeArray2 = [10, 20, 30, 40, 50]

myAwesomeArray.concat(myAwesomeArray2)

//-------> Output : [1, 2, 3, 4, 5, 10, 20, 30, 40, 50]

**12. fill()**  
Этот метод заполняет все элементы массива одинаковым значением, от начального индекса (по умолчанию 0) до конечного индекса (по умолчанию array.length).

**const** myAwesomeArray = [1, 2, 3, 4, 5]

// The first argument (0) is the value

// The second argument (1) is the starting index

// The third argument (3) is the ending index

myAwesomeArray.fill(0, 1, 3)

//-------> Output : [1, 0, 0, 4, 5]

**13. includes()**

Этот метод возвращает значение true, если массив содержит определенный элемент, и значение false — если нет.

**const** myAwesomeArray = [1, 2, 3, 4, 5]

myAwesomeArray.includes(3)

//-------> Output : true

myAwesomeArray.includes(8)

//-------> Output : false

**14. reverse()**

Этот метод меняет порядок следования элементов в массиве на обратный. Первый элемент становится последним, а последний — первым.

**const** myAwesomeArray = ["e", "d", "c", "b", "a"]

myAwesomeArray.reverse()

//-------> Output : ['a', 'b', 'c', 'd', 'e']

**15. flatMap()**

Этот метод применяет функцию к каждому элементу массива, а затем сглаживает результат в новый массив. Он объединяет метод flat() и метод map() в одну функцию.

**const** myAwesomeArray = [[1], [2], [3], [4], [5]]

myAwesomeArray.flatMap(arr => arr \* 10)

//-------> Output : [10, 20, 30, 40, 50]

// With .flat() and .map()

myAwesomeArray.flat().map(arr => arr \* 10)

//-------> Output : [10, 20, 30, 40, 50]

**Объекты. Свойста**

Копирование  
Клонирование

**Объекты. Методы**

На основе методов из объекта user создаются массивы: Object.keys – массив ключей, Object.values – массив значений, Object.entries – массив ключ+значение

let userKeysArray = **Object.keys**(user); [ ключ, ключ2, ключ3 ]

let userValuesArray = **Object.values**(user); [ значение, значение2, значение3 ]

let userEntriesArray = **Object.entries**(user);  [ [ключ, значение], [ключ2, значение2] ]

**Массив является объектом**  
  
**for** (let fruit **of** fruits) - перебирает и отображает элементы **по порядку**  
обычно перебираем **массивы**, если нам не нужен шаг и индекс  
{console.log( fruit );}  
перебор массива  
  
**for** (let fruit **in** fruits) - перебирает и отображает элементы **без упорядочивания**   
обычно перебираем **объекты,** если нам не нужен шаг и индекс  
Обычно для понимания – есть ли вообще элемент в объекте  
{console.log( fruit );}  
перебор массива

**DOM (Document Object Model), DOM-дерево**

**HTML элементы - это объект.   
В свою очередь набор таких объектов – это массив с объектами**

let customWindow = {

 BOM: {

   location: {},

   screen: {},

   history: {},

 },

 JS: {

   Object: {},

   Array: {},

   String: {},

   Number: {},

 },

 document:{}

}

**document – объект входа.**

document.body.style.background = "red";

document.body.style.cssText = 'color: red; font-size: 40px;';

**Методы поиска (получить элемент)**

**querySelectorAll**

document.querySelectorAll(css) - возвращает все элементы по css селектору.

let elements = document.querySelectorAll('.todo-list li'); // возвращает массив

**querySelector**

document.querySelector(css) //возвращает первый элемент по css селектору.

let todoList = document.querySelector('.todo-list');

// Реакция на действия пользователя

// let todoList = document.querySelector('.todo-list');

// todoList.onclick = function(event) {

//  console.log(this)

//  console.log(event.target)

// };

// this VS event.target

// event.stopPropodation()

// addEventListener и removeEventListener // НУЖНО ЗАПОМНИТЬ

// element.addEventListener('eventName', functionName) - прослушка события

// let todoList = document.querySelector('.todo-list');

// todoList.addEventListener('click', showThisAndEventTarget);

// todoList.addEventListener('click', showHello);

// function showThisAndEventTarget(event) {

//  console.log(this);

//  console.log(event.target);

// };

// function showHello(event) {

//  console.log('Hello');

//  todoList.removeEventListener('click', showHello);

// };

// element.removeEventListener('eventName', functionName) - удаление прослушки события

// Для управления классами:

// className – строковое значение, удобно для управления всем набором классов.

// classList – объект с методами add/remove/toggle/contains,

// удобно для управления отдельными классами.

// element.classList.toggle('active'); // add, remove, toggle

// Можно запомнить className (все классы элемента как строка)

// Так же можно запомнить classList (все классы как массив)

// И очень удобен classList.toggle('someClass') очень удобен

// чтобы не мучится с флагами и доп.функции

// показать - спрятать

// 1) Получаем все элементы и сохраняем их в переменные

// let todoList = document.querySelector('.todo-list');

// let showHideBtn = document.querySelector('.button-show-hide');

// let testItem = document.querySelector('.test');

// // 2) Прикрепляем прослушку событий

// showHideBtn.addEventListener('click', showHideBlock); // что делает эта функция callback? эта прячет и показывает блок

// testItem.addEventListener('click', changeText); // это меняет текст например

// // 3) Описали функции для событий

// function showHideBlock(event) {

//  todoList.classList.toggle('hidden-block'); // скрыть-показать список

//  showHideBtn.classList.toggle('show-btn'); // изменить стиль кнопки

//  showHideBtn.classList.toggle('hide-btn'); // изменить стиль кнопки

// }