Курс JavaScript\_v2.0

Кусок кода HTML

<body>

<div id="out">1111</div>

    <h2 class="header">Hello</h2>

    <p id="one">Lorem ipsum dolor sit </p>

    <script src="js/script.js"></script>

</body>

JS\_код

Работа с консолью:

console.log('Hello!'); вывод строки в консоль

console.log(334); вывод числа

console.log('Hello' + ' ' + 'world'); конкатенация строк

console.info('Hello'); - аналог console.log

alert('Hello'); - вызов всплывающего окна

Обращение к элементам на странице:

document.getElementById('out').innerHTML = 'Hello'; - обращение к элементу по ID

document – говорит, что мы работаем в текущем документе, на текущей странице

.getElementById('id') – метод с помощью которого мы получаем элемент с html кода по его id

document.getElementById('id') – эта конструкция позволяет выбрать элемент со страницы

к которой был подключен javascript, по id элемента.

.innerHTML – специальное свойство, присвоив значение которому,

изменится содержимое выбранного элемента.

document.getElementById('out').innerHTML = '<b>2020</b>';

помимо присвоения строк, JS так же может записать новые теги на страницу

Новый вариант обращения к элементам через querySelector

Работает на основе синтаксиса CSS, т.е. обращаться к элементам можно аналогично

Обращению к ним в CSS. Т.е. обращение по тегам, классам, id или комбинаций.

querySelector получает первый элемент встреченный в html документе. Т.е. если

указать querySelector('p'), то в отличии от CSS будут выбраны не все элементы по типу тега “p”,

а только первый, и все изменения получит так же, только первый встреченный элемент.

document.querySelector('.header').innerHTML = 15;- обращение по классу

Разница в обращении по старому и новому методу.

document.querySelector('#one').innerHTML = 777; - есть #

document.getElementById('one').innerHTML = 888; - нет #

При многократном обращении к элементу, необходимо создать переменную и присвоить ей в значение

выбор необходимого элемента. И далее к этой переменной применять необходимые свойства.  
Это значительно облагораживает код и логику написания.

let a = document.querySelector('#one'); - задали переменной значение обращения к элементу

a.innerHTML = Новое значение для элемента с id=one; присвоение нового значение через переменную.

Добавление содержимого к элементу БЕЗ удаление существующего содержимого

реализуется с помощью .innerHTML +=

<h2 class="header">Hello</h2>

document.querySelector('.header').innerHTML += “ world”;

<h2 class="header">Hello world</h2>

Если в добавляемом элементе уже присутствуют кавычки, например при добавлении изображения

Необходимо использовать кавычки другого типа и чередовать их.

Т.е. “…‘…’…”

let b = document.querySelector('.out-13');

b.innerHTML='<img src="https://itgid.info/img/js20/js20\_unit\_01.png" alt="js">'

.textContent – свойство, присвоив значение которому, он добавит это значение к выбранному элементу

но в отличие от .innerHTML – если добавляемое значение будет иметь теги, он не учтёт их в разметку, а добавить как строчный элемент.

<h2 class="header">Hello</h2>

document.querySelector('.header').textContent += “<em>world</em>”;

<h2 class="header">Hello<em>world</em></h2>

Основы ввода данных

Кусок кода HTML



<input type="text" class="input-in" value="hello"> создаст поле ввода

<button>Go</button> создаст кнопку

<div class="out"></div> создаст пустой контейнер

JS\_код

Получаем в переменные элементы со страницы с помощью querySelecror

let inputIn = document.querySelector('.input-in'); // input

let button = document.querySelector('button'); // button

let div = document.querySelector('.out'); // пустой контейнер

Создаем событие onclick, т.е. по нажатию на button отработает функция

button.onclick = function () {

    // console.log(inputIn.value); // в консоль будет выведено значение value

value – это свойство input, обратившись к которому, мы получим значение введенное в поле

значение может быть прописано заранее или введено пользователем

    let b = +inputIn.value; // + перевести в число!!!!!!!

Знак “+” перед inputIn.value принудительно переведет введенное в input значение в number

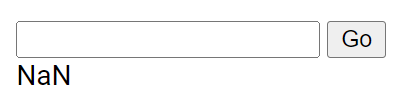
Без указания знака +, как со словами так и с числами будет производится конкатенация, а не математические действия. console.log(inputIn.value + 10); // '66'+10 = 6610

    console.log(b + 10); // 66+10 = 76

    div.innerHTML = b; - добавит в div с классом .out полученное из input значение

    inputIn.value = ''; - очистит input, после отработки функции, а следовательно и события.

}

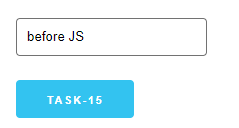


NaN – результат вывода строки (значения value) hello, JS сообщает нам,  
что это НЕ ЧИСЛО ( NaN ) и записав это в переменную b,

которую мы объявили числом ( с помощью “+” ), и добавляет в div.out

с помощью .innerHTML - NaN, если это будет число, то число отобразится корректно.

Добавление стилей при нажатии на кнопку



HTML\_код

<input type="text" class="i-15" value="before JS" />

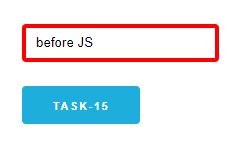
<!-- style="border: 4px solid red" -->

Закомментированная строчка кода, добавится к input после отработки JS кода ( по нажатию button )

<button class="button-primary b-15">Task-15</button>

JS код ниже добавит к данному input новый стиль при нажатии на кнопку

function t15() {



    let button\_15 = document.querySelector('.b-15');

    let input\_15 = document.querySelector('.i-15');

    button\_15.onclick = function () {

        input\_15.style.border = "4px solid red"

    }

}

document.querySelector('.b-15').onclick = t15;

Несколько встроенных функций и объектов в JS

Данные встроенные функции/объекты интерпретируют полученное значение value в зависимости от своего функционала.

let a = document.querySelector('откуда получаем значение').value;

document.querySelector('эл. вывода').innerHTML = +a;

Преобразует строку с числами в число.

Если аргумент не может быть преобразован в число, возвращается NaN.

document.querySelector('эл. вывода').innerHTML = parseInt(a, 10);

Функция parseInt() принимает строку в качестве аргумента и возвращает

целое число в соответствии с указанным основанием системы счисления.

Синтаксис

parseInt(string, radix);

Параметры

String

Значение, которое необходимо проинтерпретировать. Если значение параметра string не принадлежит строковому типу, оно преобразуется в него (с помощью абстрактной операции ToString). Пробелы в начале строки не учитываются.

Radix

Целое число в диапазоне между 2 и 36, представляющее собой основание системы счисления числовой строки string, описанной выше. В основном пользователи используют десятичную систему счисления и указывают 10. Всегда указывайте этот параметр, чтобы исключить ошибки считывания и гарантировать корректность исполнения и предсказуемость результата. Когда основание системы счисления не указано, разные реализации могут возвращать разные результаты.

Возвращаемое значение

Целое число, полученное парсингом (разбором и интерпретацией) переданной строки. Если первый символ не получилось сконвертировать в число, то возвращается NaN.

document.querySelector('эл. вывода').innerHTML = Number(a);

Преобразует строку с числами в число.

Если аргумент не может быть преобразован в число, возвращается NaN.

В отличии от “+”, имеет методы, возможные применить к значению

Number.isNaN()

Определяет, является ли переданное значение значением NaN.

Number.isFinite()

Определяет, является ли переданное значение конечным числом.

Number.isInteger()

Определяет, является ли тип переданного значения «числом», а само число — целым значением.

Number.isSafeInteger()

Определяет, является ли переданное значение безопасным целым числом

(числом в диапазоне от -(253 - 1) до 253 - 1).

Number.parseFloat()

Метод разбирает строковый аргумент и возвращает число с плавающей запятой.

Number.parseInt()

Метод разбирает строковый аргумент и возвращает целое число.

Элементарный счётчик на странице:

HTML

<button class="button-primary b-20">Task-20</button>

<div class="out-20">0</div>

JS

let count = 0;

function t20() {

    count++;

при каждом вызове функции переменная count будет прибавлять к своему значению в арифметической прогресс 1. Аналогичная запись count = count + 1;

Так как переменная объявлена глобально, и внутри функции аналогичной нет, а её значение только перезаписывается, то значение из локальной функции присвоится глобальной переменной let count.

Т.е. переменная count на 2-ом круге придёт уже со значением равному 1. На 3-ем, равному 2ум и тд.

    let out\_20 = document.querySelector('.out-20');

Схватили элемент вывода ( куда счётчик будет выводить значение )

    out\_20.innerHTML=count;

Присвоили свойство элементу, которое будет записывать значение полученное с помощью count++

}

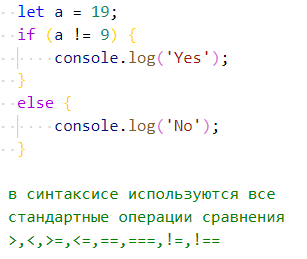
document.querySelector('.b-20').onclick = t20;

вызов функции

При каждом вызове функции, значение в out-20 будет перезаписываться согласно арифметической прогрессии.

**If…else**

Инструкция if выполняет инструкцию, если указанное условие выполняется (истинно). Если условие не выполняется (ложно), то может быть выполнена другая инструкция else.



Синтаксис

if (условие){

инструкция1

} else {

инструкция2

}

условие

Выражение, которое является либо истинным, либо ложным.

инструкция1

Инструкция, выполняемая в случае, если значение "условиe" истинно (true).

Может быть любой инструкцией в том числе и вложенным if.

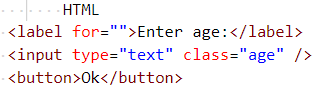
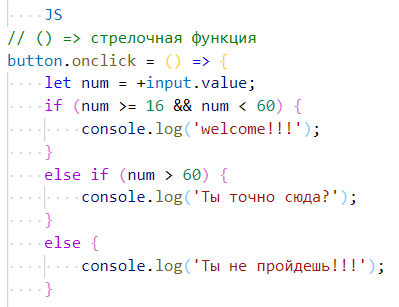
инструкция2

Инструкция, выполняемая в случае, если значение "условиe" ложно (false).

Может быть любой инструкцией, в том числе и вложенным if.



Пример:



В данном примере, значение из input,

после нажатия button пройдёт проверку в if…else;

И в зависимости от полученного value, выведет

в консоль один из трёх вариантов.

Помимо прямого условия типа a>b, можно создавать каскад

условий, т.е. if сработает если ( a>b && a<c ), в данном  
случае часть кода if отработает только при условии, что оба  
условия дадут true.

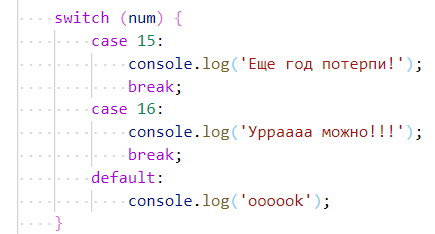
Операторы:

&& - оператор логическое И, т.е. ОБА условия должны дать true;

|| - оператор логическое ИЛИ, т.е. или ОДНО либо ДРУГОЕ условие должны дать true;

! - оператор логическое НЕ, возвращает false, если операнд может быть преобразован в true; в противном случае возвращает true.

Switch

Инструкция switch сравнивает выражение со случаями, перечисленными внутри неё, а затем выполняет соответствующие инструкции.

В данном примере, если выражение num, будет ранов 15,

то сработает соответствующая инструкция,

если же 16 то вторая, а если выражение не равно

ни 15 ни 16, то отработает default.

Сравнение в switch строгое ( === ).

Если не прописывать break, после каждой инструкции, то

после срабатывания инструкции которая была равна выражению num

– switch продолжит выполнять последующие инструкции,

и проверка на соответствие выражению не будет выполняться.

let foo = 0;

switch (foo) {

case -1:

console.log('negative 1');

break;

case 0:

// foo равно 0, случай соответствует выражению и эти инструкции будут выполнены

console.log(0)

// ПРИМЕЧАНИЕ: здесь могла находиться забытая инструкция break

case 1:

// В случае 'case 0:' не было break, инструкции данного случая также будут выполнены

console.log(1);

break;

// В конце расположен break, поэтому выполнение не перейдёт к случаю 'case 2:'

case 2:

console.log(2);

break;

default:

console.log('default');

}

Т.е. хоть foo и равно 0, и по идее должна была отработать только инструкция соответствующая выражению num, но так как мы не остановили инструкцию с помощью break, switch продолжит выводить безразборно инструкции до break или завершения switch

Цепочки case

Этот метод использует тот факт, что после case нет прерывания и продолжится выполнение следующего case независимо от того, соответствует ли case предоставленному условию.

Это пример case с одной операцией, где четыре разных значения отрабатывают одинаково.

let Animal = 'Giraffe';

switch (Animal) {

case 'Cow':

case 'Giraffe':

case 'Dog':

case 'Pig':

console.log('This animal is not extinct.');

break;

case 'Dinosaur':

default:

console.log('This animal is extinct.');

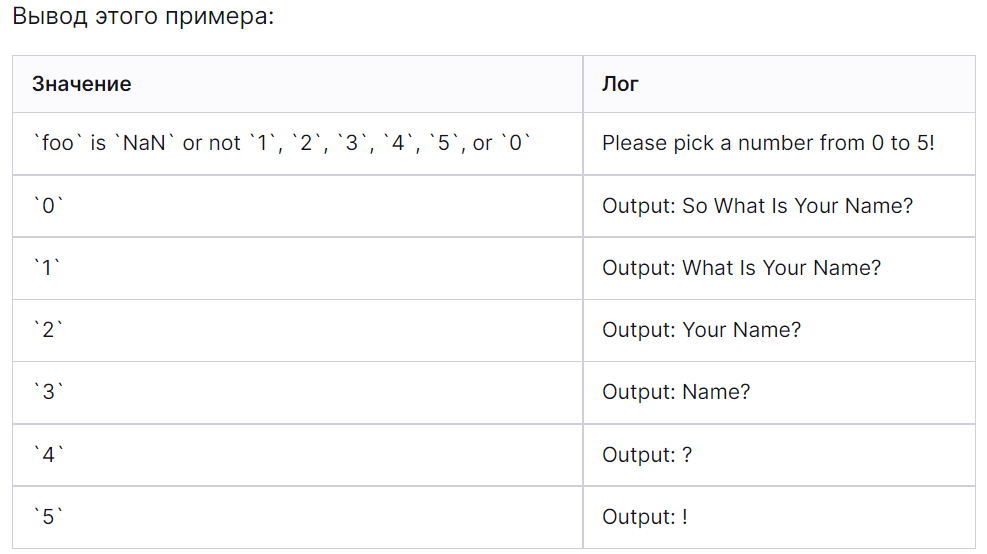
}

Какое бы из значений (Cow, Giraffe, Dog, Pig ) не приняла инструкция switch в качестве выражения,

вывод для них будет единым.

Цепочка операций

Это пример множественных операций внутри case, где в зависимости от предоставленного числа можно увидеть разный вывод. Здесь показывается, что операции отрабатывают в том порядке, в котором расположены case. При вводе строки, выведется инструкция default.



var foo = 1;

var output = 'Output: ';

switch (foo) {

case 0:

output += 'So ';

case 1:

output += 'What ';

output += 'Is ';

case 2:

output += 'Your ';

case 3:

output += 'Name';

case 4:

output += '?';

console.log(output);

break;

case 5:

output += '!';

console.log(output);

break;

default:

console.log('Please pick a number from 0 to 5!');

Проверка числа на чётность:

function f6(){

   let input = document.querySelector(".i-6").value;

   if (input % 2 === 0) {

      document.querySelector(".out-6").textContent = 'even';

   }

   else {

      document.querySelector(".out-6").textContent = 'odd';

   }

}

Отработка события при смене фокуса с элемента:

function f11(){

   let select = document.querySelector(".s-110").value;

   select.onchange = document.querySelector(".out-11").textContent = select;

}

Определение типа данных:

function f12(){

   let i120 = document.querySelector('.i-120');

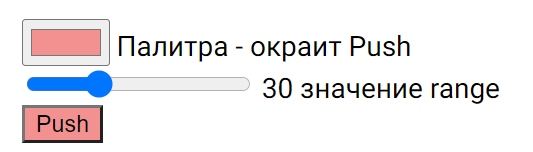
   let v = i120.value;

   document.querySelector('.out-12').innerHTML = (typeof v);

}

Работа с INPUT, RANGE, TEXTAREA, CHECKBOX и FORM

HTML



 <input type="color" id="one-1">

 <span class="color">Палитра - окрасит Push</span>

 <br>

 <input type="range" id="one-2" min="0" max="100" value="30">

 <span><span class="range">30</span> значение range</span><br>

 <button id="btn">Push</button>

JS

document.querySelector('#btn').onclick = () => {

    let a = document.querySelector('#one-1')

    console.log(a.value);

    document.querySelector('#btn').style.backgroundColor = a.value

}

let b = document.querySelector('#one-2')

b.oninput = () => {

    console.log(b.value);

    document.querySelector('span.range').innerHTML = b.value

}

Указано 2 input один для выбора цвета, второй имеет шкалу range.  
при нажатии на button, выбранный в палитре цвет окрашивает кнопку и выводит rgb код этого цвета в консоль.   
Второй input работает при изменение его шкалы range, выводит в span и консоль текущее значение range.

HTML



<input type="checkbox" id="i2" value="I'm\_checkbox\_value">

<label for="i2">Click me</label>

<button id="btn-1">Push</button>

JS

document.querySelector('#btn-1').onclick = () => {

    let myCheckBox = document.querySelector('#i2');

    console.log(myCheckBox.value);

    console.log(myCheckBox.checked);

    if (myCheckBox.checked) {

        console.log('Нажат');

    }

    else {

        console.log('Не Нажат');

    }

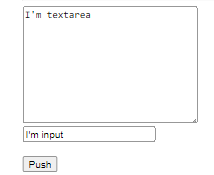
}

Данный input типа checkbox создает область выбора типа Вкл/Выкл.

JS отслеживает его состояние и выводит по нажатию button:

1. Значение value данного checkbox
2. Логическое состояние Вкл/Выкл ( true/false ) в консоль.
3. Выводит сообщение в зависимости от состояния в консоль.

HTML



<form action="">

<textarea name="on\_name" cols="30" rows="10"></textarea>

<br>

<input type="text" id="on\_id">

<br><br>

<button id="btn-2">Push</button>

</form>

Тег form – служебный тег, обычно им оборачивают область кода для ввода информации от пользователя. Атрибут action – это адрес URL куда будет отправлена форма, после её отправки пользователем. ( Если не указывать путь, то страница будет перезагружаться при нажатии на button. )

JS

document.querySelector('#btn-2').onclick = (event) => {

   event.preventDefault();

   let form = document.querySelector('form');

   console.log(form.elements.on\_name.value);

   console.log(form.elements.on\_id.value);

}

Что бы получать в консоль value элементов, нам для начала нужно оставить принудительную перезагрузку страницы, это возможно с помощью специального метода.  
На событии .onclick необходимо добавить параметр event ( объект события ) и добавить соответствующий метод event.preventDefault();

К элементам form, можно обращаться на прямую ( не выбирая их через querySelector )

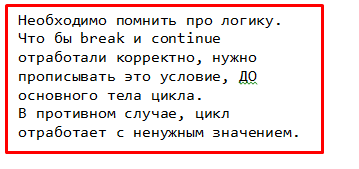
Для этого, у всех элементов form должно быть либо name либо id. Обращаться к элементам, можно и так и так. К textarea обратились через name. К input через id.

Функция выше, выведет в консоль value обоих элементов формы.

**Цикл For**

Цикл будет повторяться, пока условие true;

for ( условие ) {  
тело цикла, которое будет  
выполняться пока true;



}

let out = '';

for (let i = 0; i < 10; i++) {

    if (i == 6) continue;



    out += i + ' ';

    // if (i == 6) break;

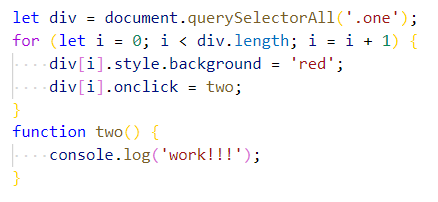
}

document.querySelector('#out').innerHTML = out;

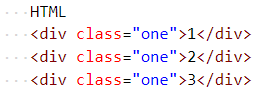
переменная out изначально пустая, туда будет выводиться результат цикла.

continue – пропустить какое-то значение из выполнения в цикле, т.е. в данном случае, когда i==6, тело цикла не выполнится, а перескочит к следующему значению.

break – выход из цикла, при достижении какого либо значения переменной из условия цикла.



Циклы тесно работают с массивами.



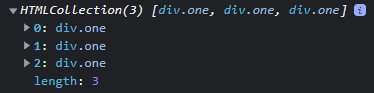
Функция div выберет все элементы страницы



с классом .one и передаст их в тело функции

в виде массива. Каждый элемент будет помещён

в массив ( HTMLCollection(3) ) и им будут



присвоены индексы, начиная с 0. К каждому

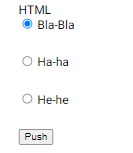
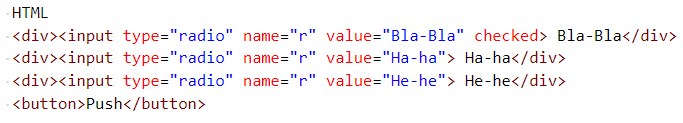
элементу массива можно как обратиться

отдельно так и создать цикл, который будет

перебирать их в последовательности.

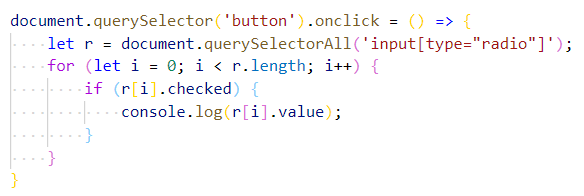
Данный код присвоит каждому элементу массива

background-color: red; и при нажатии на любой элемент ( div.one ) в консоль будет выведена надпись “work!!!”



Цикл срабатывает по нажатию на button.

в переменную r выбраны все элементы



типа радио кнопка

input[type=“radio”]

И по нажатию на button в зависимости

какой пункт выбран checked [0,1,2] –

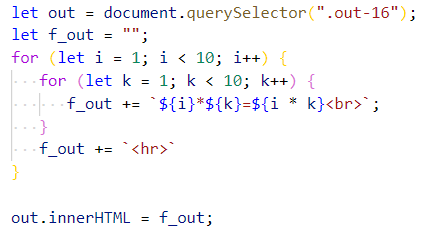
условие if отработает и выведет в

консоль value того элемента массива

которое сейчас checked.

**Цикл в цикле**

Цикл for можно вкладывать в другой цикл for. Что позволяет выполнять вложенный цикл, столько раз, сколько итераций заложено в родительском условии.



Данный код, выведет в подготовленный div.out-16

таблицу умножения.

Схема цикла:

Проверка условия i=1 > 10; true -> переходим к

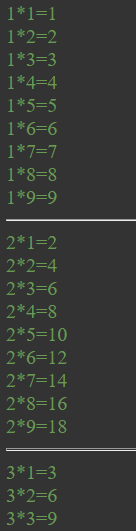
телу цикла. В теле дочерний ( вложенный ) цикл

for. Заходим в него и начинаем выполнять ДО

тех пор пока k<10 даёт true; Как только

получаем false, вложенный цикл завершается и мы

выходим из него, попадая на строку f\_out += `<hr>`



добавляя горизонтальную отсечку,после добавления которой

цикл возвращается к условию прибавляя к переменной i единицу

и производит проверку, i=2 < 10; true ->

прогоняем тело цикло еще раз, а вложенный цикл for,

отработает аналогичным образом, как и в первый раз, так же

после завершения вложенного он из него выйдет,

добавит черту и продолжит проверять значения в родительском

цикле и выполнять его до получения false.

Интерполяция

Так же, во вложенном цикле for был применен способ соединения

строк – интерполяция. Что является упрощённым вариантом

конкатенации.  
Синтаксис:

для обозначения интерполяции необходим

обратный апостроф `...`;

внутри которого записывается строка. Если необходимо

вывести какие либо переменные, то их необходимо поместить

в следующую структуру ${...}

Где вместо многоточия добавляется текст или переменные.

**Цикл While**

Оператор while создаёт цикл, выполняющий заданную инструкцию, пока истинно проверяемое условие. Логическое значение условия вычисляется перед исполнением тела цикла.

**while (условие) {**

**инструкция**

**}**

Пока условие даёт true будет выполняться инструкция.

let sum = 0;

let p = 0;

while (p <= 3) {

    sum = sum + p;

    p++;

}

console.log('sum = ' + sum);

summ – придёт результат выполнения цикла.

p – переменная для задания условия и выполнения инструкции цикла

p++ - добавление значения к переменной p после каждой итерации инструкции.

let out = document.querySelector('.out'); - ловим элемент со страницы, для вывода

p = 0;// переписываем переменную p ( так можно, она была задана глобально )

let outStr = ''; // для результата цикла

let flag = 3; // вспомогательная переменная для отрисовки 0;



while (p < 4) {

    let p1 = 0; // объявление вспомогательной переменной, для вложенного цикла

//переменная p1 ВНУТРИ РОДИТЕЛЬСКОГО ЦИКЛА!!!

    while (p1 < 4) { // вложенный цикл while

        if (p1 < flag) // условие для добавления 0

            outStr += '0'; // добавление 0 при true в if

        }

        else {

            outStr += '\*'; // добавляем \* при false в условии выше

        }

        p1++; // увеличение значения p для отработки if else вложенного цикла.

    }

    flag--; // уменьшение значения переменной flag после выполнения вложенной инструкции

    outStr += '<br>'; // добавление переноса строки после выполнения вложенной инструкции

    p++; // добавление значения переменной p после выполнения вложенной инструкции

}

out.innerHTML = outStr; //вывод на страницу результата цикла

**Цикл do...while**

Выражение do...while создаёт цикл, который выполняет указанное выражение до тех пор, пока условие не станет ложным. Условие проверяется после выполнения выражения, то есть выражение выполнится как минимум один раз.

let result = '';

let i = 0;

do {

   i += 1;

   result += i + ' ';

} while (i<0); // false

console.log(result); // в консоль выведется 1;

Не смотря на то, что в условии while сразу будет false, одна итерация цикла выполнится, потому что условие проверки идёт ПОСЛЕ, отработки условия. Т.е. сначала инструкция, потом условие.

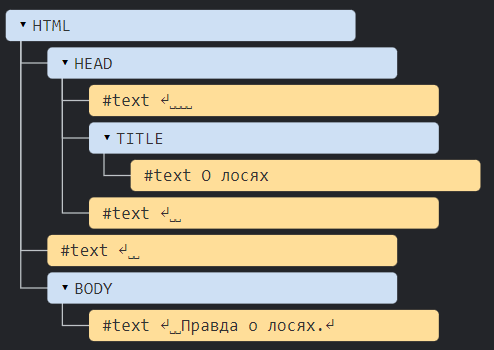
**Работа с DOM**

(«**Document Object Model**»)

В соответствии с объектной моделью документа(DOM), каждый HTML-тег является объектом. Вложенные теги являются «детьми» родительского элемента. Текст, который находится внутри тега, также является объектом.

Все эти объекты доступны при помощи JavaScript, мы можем использовать их для изменения страницы.

Пример DOM



<!DOCTYPE HTML>

<html>

<head>

<title>О лосях</title>

</head>

<body>

Правда о лосях.

</body>

</html>

На рисунке выше отображены узлы-элементы, связь родительских и дочерних узлов-элементов.

Каждый узел этого дерева – это объект.

Теги являются узлами-элементами (или просто элементами). Они образуют структуру дерева: <html> – это корневой узел, <head> и <body> его дочерние узлы и т.д.

Текст внутри элементов образует текстовые узлы, обозначенные как #text. Текстовый узел содержит в себе только строку текста. У него не может быть потомков, т.е. он находится всегда на самом нижнем уровне.

**Итого**

HTML/XML документы представлены в браузере в виде DOM-дерева.

Теги становятся узлами-элементами и формируют структуру документа.

Текст становится текстовыми узлами.

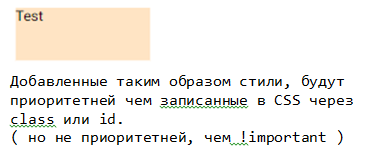
… и т.д. Всё, что записано в HTML, есть и в DOM-дереве, даже комментарии.

( читать больше: <https://learn.javascript.ru/dom-nodes> )

Манипуляции со стилями и классами через JS

С помощью JS можно добавлять стили к элементам HTML разметки.

HTML



<p class="one">Test</p>

JS

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

one.style.width = '150px';

one.style.paddingBottom = '40px';

one.style.background = "bisque";

Обращение к элементу на прямую, через конструкцию element.style.property – считается не лучшим решением и по возможности избегается, в пользу использования классов.

Одним из вариантов работы с классами в JS, представляется с помощью element.classList.

При вызове подобной конструкции console.log(element.classList); выведет в консоль динамическую DOMTokenList коллекцию class атрибутов элемента.

( читать больше: <https://developer.mozilla.org/en-US/docs/Web/API/Element/classList> )

( читать о DOMTokenList <https://developer.mozilla.org/en-US/docs/Web/API/DOMTokenList> )

Хотя само classList свойство доступно только для чтения, можно изменить его – DOMTokenList,

используя методы .add(), .remove(), .replace(), .toggle() и .contains().

**Методы для classList**

.add() – добавляет новый class к элементу. // element.classList.add(“class”);

.remove() - удаляет существующей class у элемента // element.classList.remove(“class”);

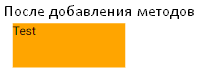
.replace() – заменят сущ. old\_class на new\_class у элемента.

Если первого не существует, возвращается false. //element.classList.add(“old\_class”,”new\_class”);

.toggle() – переключает class у элемента .**Вкл/Выкл** // element.classList.toggle(“class”);

.contains(). – производит проверку наличия class у элемента // element.classList.contains(“class”);

**Пример с add и remove**



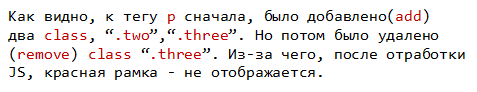
HTML



<p class="one">Test</p>

CSS

.one {background: red;}



.two {background: orange;}

.three {border: 2px solid red;}

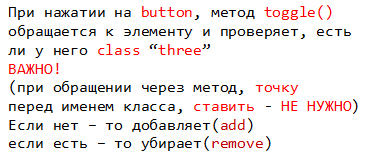
JS

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

one.classList.add('two', 'three');

one.classList.remove('three');

**Пример с toggle**



HTML

<button class="b-1">Push</button>



JS

button\_1.onclick = function () {

    this.classList.toggle('three');

}

**Пример с replace**



HTML

<span class="one three">Меняем class='one' на class='two'</span>

JS



const span = document.querySelector('span.one');

document.querySelector("span").onclick = () => {

    span.classList.replace("one", "two");

}

До нажатия на span, ему были присвоены классы “one”,“three” по нажатию на span класс “one” был заменен на класс “two”. ( Применение стрелочной функции и события .onclick не обазятельный для метода replace )

**Пример с contains**

Если бы код ниже, проверил наличие class=“one”

у span из предыдущего примера, то

до метода replace, contains вернул бы true, после false.

span.classList.contains("one");

**Работа с атрибутами**

**HTML-атрибуты** это специальные слова, которые управляют поведением HTML-элемента. Они добавляют дополнительную функциональность, либо меняют поведение элемента по умолчанию. Атрибуты элемента выражаются внутри начального тега элемента. Атрибут имеет имя и значение.

Некоторые атрибуты не требуют значение, потому что у них есть только одна опция. Они называются логическими атрибутами.

Помимо стандартных атрибутов, типа class,style, id, link, src..., можно задавать произвольные атрибуты, значение и функционал которых будет зависеть от ситуации. Синтаксис data-\*=value ( где \* - это произвольное имя, а value – это числовое, либо строковое выражение, которое будет назначено данному атрибуту )

(Читать об атрибутах больше: <https://developer.mozilla.org/ru/docs/Web/HTML/Attributes>)

С помощью JS можно получить значение существующего атрибута с помощью метода getAttribute() :

element.getAttribute(“class”) – получит все значения, присвоенные атрибуту типа class, данного элемента.

Если у выбранного элемента, несколько атрибутов одного типа ( названия ), то обратиться к конкретному можно через индекс массива:

element[i]. getAttribute(“class”) – где i – это порядковый индекс необходимого элемента массива.

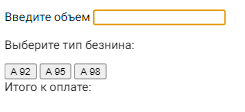
С помощью JS мы можем как манипулировать существующими, так и создавать новые атрибуты с помощью

метода setAttribute() :

element.setAttribute(“data-name”, 10 ) – добавит элементу (element) атрибут с именем “data-name” и значением 10.

В параметрах, сначала идёт имя нового атрибута, потом его значение.

HTML



<p>Введите объем:</p>

<input type="text" class="gallons">

<p>Выберите тип безнина:</p>

<div>

 <button class="gas" data-A="1.2">A 92</button>

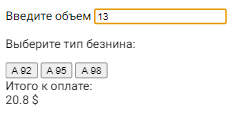
 <button class="gas" data-A="1.6">A 95</button>

 <button class="gas" data-A="1.9">A 98</button>

</div>

<label for="pay">Итого к оплате:</label>

<div class="out" name="pay"></div>



JS

let out\_1 = document.querySelector(".out");

let gas = document.querySelectorAll('.gas');

for (let i = 0; i < gas.length; i++) {

    gas[i].onclick = function () {

        let gallons = document.querySelector('.gallons').value;

        let amount = this.getAttribute('data-A');

        out\_1.textContent = gallons \* amount + " $";

    }

}

Введя число, и нажав на одну из кнопок, введенное число умножится на коэффициент указанный в data-A и результат выведется в поле out.

Переменная gas получает массив кнопок с class=“gas”. Далее с помощью цикла for, выбирается элемент gas[i] по порядковому индексу выбранной кнопки ( методом перебора от 0 до 2-ух, переменная i существует только внутри цикла, и условно имеет одновременно все возможные значения удовлетворяющие условию i<gas.length), который в свою очередь по событию .onclick запускает функцию, получающую значение из input и

значение атрибута data-A этого элемента массива ( this ), далее выводит произведение в div.out.

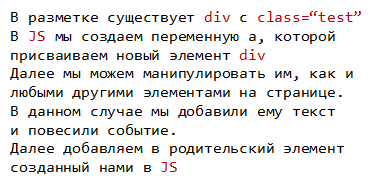
Создание новых элементов

В JS есть возможность создавать и добавлять на страницу новые теги.

Создавать новые элементы, можно с помощью метода document.createElement(“имя тега”)

Что-бы разместить элемент на странице, необходимо выбрать родительский элемент

и прописать метод .appendChild(element), где element это размещаемый тег/элемент



HTML

<div class="test">Test</div>

JS

let a = document.createElement('div');

a.innerHTML = 'Hello!';

a.classList.add('one');

a.onclick = function () {

    alert('hello');



}

document.querySelector('.test').appendChild(a);

Element.append() и его отличия от appendChild()

(.before . after .replaceWith )

Метод Element.append() вставляет узлы или строки с текстом в конец Element. Строки с текстом вставляются как текстовое содержимое.

Отличия от метода Node.appendChild():

1)Метод Element.append() позволяет вставлять строки с текстом,

в то время как Node.appendChild() работает только с узлами.

2)При вызове метод Element.append() ничего не возвращает,

в то время как Node.appendChild() возвращает вставленный узел.

3)С помощью Element.append() можно вставить сразу несколько узлов или строк текста,

в то время как Node.appendChild() умеет вставлять по одной сущности за раз.

let a = document.createElement("div");

let b = document.createElement("div");

function f11() {

    let out = document.querySelector(".out-11");

    a.textContent = 25;

    b.textContent = "REPKA";

    out.append(a, b, "ZABIYAKA");

}

Аналогичным образом работают следующие методы:

.before – добавляет элементы непосредственно перед родительским элементом

document.querySelector(element\_1).before(element\_2)

element\_2 будет расположен перед element\_1

.after – добавляет элементы непосредственно после родительского элемента

document.querySelector(element\_1).after(element\_2)

element\_2 будет расположен после element\_1

.replaceWith - заменяет существующий элемент на новый

document.querySelector(element\_1).replaceWith(element\_2)

element\_2 будет размещён вместо element\_1

Данные методы, работают как с одним, так и несколькими узлами, по аналогии с .append()

( Больше о элементе и его методах <https://developer.mozilla.org/en-US/docs/Web/API/Element#methods> )

Массивы

Массив – это способ представление данных в “связанном”(объединенном) виде. Массив в JS объявляется [].

В массиве могут храниться разные типы данных.

let a = [“Sergey”, 27, “Kolomna”]; - переменная a, это массив имеющий 3 значения.

let b = []; - переменная b, это массив, которому еще не присвоены значения.

Что бы обратиться к элементу массива, используется индекс массива. У каждого массива, значения его индексов начинается с 0;

console.log(a[2]) – в консоль выведется значение “Kolomna” из массива a.

Если обратиться, к еще не созданному элементу массива ( a[5] ) – то вернётся undefined.

a.length – выведет длину массива ( в данном случае length: 3 )

Длинна массива – это индекс последнего элемента + 1;

Изменение местами значений массива

let a = [1, 2, 3, 4] // создаем стандартный массив

console.log(a); // вывод в консоли (4)[1,2,3,4]

let b = a[0]; // создаём переменную-хранилище, для записи изначального значение 0-го элемента массива

a[0] = a[a.length - 1]; // присваиваем 0-му значению – значение последнего элемента массива.

a[a.length - 1] = b; // присваиваем последнему значению, значение записанное во временную переменную

а значит, изначально 0-ое значение массива

console.log(a); // вывод в консоли (4)[4,2,3,1]

Вывод цикла на страницу

На странице создан div с class=out-1. Выведем в него элементы массива.

( В данном примере, было добавлено условие вывода if...else оно не обязательно )

let arro = [1, 2, 3, 4] // создаём массив

let out = ""; // создаем пустую переменную в которую будем добавлять элементы массива

for (let i = 0; i < arro.length; i++) { // создаем цикл, который переберёт все индексы массива

    if (arro[i] % 2 == 0) { // создадим условие, которое будет добавлять все чётные числа

        out += arro[i] + "\_\_"; // если число чётное в переменную out добавится значение + “\_\_”

    } else {

        out += arro[i] + '\_'; // если число не чётное в переменную out добавится значение + “\_”

    }

}

document.querySelector(".out-1").innerHTML = out; // вывод на страницу

( Результат на странице 1\_2\_\_3\_4\_\_ )

Нахождение максимального/минимального значение в массиве

let arro\_2 = [12, 22, 18, 9, 65, 1, 85, 0]; //создаём массив

let max = arro\_2[0]; //создаём переменную и присваиваем ей первое значение в массиве

for (let i = 0; i < arro\_2.length; i++) { //создаем цикл, который будет перебирать все значения массива

    if (arro\_2[i] > max) { //делаем проверку, каждого элемента массива с предыдущим, если true (изменив знак на <, в условии выше, будет

выводится меньшие число из массива)

        max = arro\_2[i]; // то присваиваем переменной max новое значение ( которое оказалось больше )

    }

}

document.querySelector(".out-2").innerHTML = `max: ${max}`; // вывод на страницу

( Результат на странице max: 85 )

Нахождение суммы

let arro\_3 = [12, 22, 18, 9, 65, 1, 85, 0]; // создаем массив

let sum = 0; //создаем переменную “корзинку” куда будут складываться значения из массива

for (let i = 0; i < arro\_3.length; i++) { //создаем цикл, который будет перебирать все значения массива

    sum = sum + arro\_3[i]; //при каждой итерации цикла, прибавляет к sum значение[i]

}

document.querySelector(".out-3").innerHTML = `sum: ${sum}`; // вывод на страницу

( Результат на странице sum: 212 )

**Разделение исходного массива на чётный и не чётный массивы с выводом на страницу**

HTML

<button class="button-primary b-16">Task-16</button>

            <div class="out-16-even"></div>

            <div class="out-16-odd"></div>

JS

Созданы 3 переменные:исходный массив ar16 и два для разделения ar16\_odd и ar16\_even – изначально пустые

let ar16 = [0, 2, 5, -4, 6, 22, -9, -12, 8, 12, 13, 78, 16, 25, 91, 1, 99, 1952];

let ar16\_odd = [];

let ar16\_even = [];

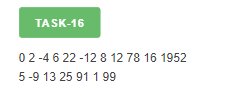
function f16() {

    let a16 = 0;

    let b16 = 0;

    let out\_1 = "";

    let out\_2 = "";



    for (let i = 0; i < ar16.length; i++) {

        if (ar16[i] % 2 != 0) {

            ar16\_odd[a16] = ar16[i];

            out\_1 += `${ar16\_odd[a16]} `;

            a16++

        } else {

            ar16\_even[b16] = ar16[i];

            out\_2 += `${ar16\_even[b16]} `;

            b16++;

        }

    }

    document.querySelector('.out-16-odd').innerHTML = out\_1;

    document.querySelector('.out-16-even').innerHTML = out\_2;

}

document.querySelector('.b-16').onclick = f16;

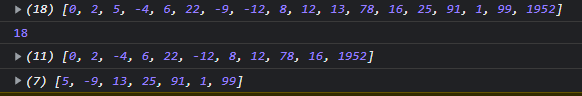
Внутри функции, по нажатию на которую будет происходит выборка и разделение, создано 4 переменные. Две переменные счётчики a16 и b16 и две переменные-контейнеры для вывода на страницу out\_1 и out\_2.Создаем цикл for для работы с заданным массивом. В условии задано количество итераций через переменную i и условия, что пока i меньше длинный цикла ar16 ( i<ar16.length ) будет исполняться тело цикла. Тело цикла в свою очередь разделено на if...else, где первое условие производит проверку деления на 2 без остатка значений массива ar16, если остаток есть,то к массиву ar16\_odd с индексом a16 ( ar16\_odd[a16] ) добавится значение изначального массива ar16 с индексом i ( ar16[i] ), после чего происходит добавление значения массива ar16\_odd[a16] в переменную для вывода out\_1, после чего увеличиваем значение переменной a16 на 1 ( a16++ ). Else отрабатывает аналогичным образом с чётными числами.

Переменные a16 и b16 нужны для выставления правильных индексов в новые массивы, т.е. при добавлении первого значения в массив ar16\_odd/ ar16\_even, переменные a16 и b16 равны 0 ( нулю ), соответственно и индекс при добавлении значения будет 0 (нулевой). После добавления первого значения в массив, нам нужно, что бы индекс для последующего добавленного значения был уже не 0 ( нуль ), а 1 ( единица ), для этого, перед завершением отработки if...else и перехода к новой итерации цикла for, мы добавляем к переменной a16 или b16 единицу ( a16++/b16++ ), увеличивая индекс для последующих добавлений в цикл соответственно.

**Для проверки корректного разделения массивов, лучше вывести их в консоль**

JS

f16();



console.log(ar16);

console.log(ar16.length);

console.log(ar16\_even);

console.log(ar16\_odd);

Методы push(), pop(), shift(), unshift(), splice(),

reverse(), indexOf(), concat(), includes(),join() и оператор delete

const a = [1, 2, 3, 4, 5, 6, 7, 8, 9];

const b = ["a", "b", "c", "d", "e", "f", "w", "t", "q",];

//метод push() - позволяет добавлять элементы в массив. Может принимать несколько элементов

//arr.push(element1,...elementN);

console.log(a.length);

console.log(a.push(10, 11, 71, 91));//выведет новую длину массива в консоль

console.log(a);

//(13) [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 71, 91]

b.push("j", "x", "o");

console.log(b);

//(12) ['a', 'b', 'c', 'd', 'e', 'f', 'w', 't', 'q', 'j', 'x', 'o']

//метод pop() - удаляет элемент в конце массива и возвращает его значение или undefined.

b.pop(); // просто удалили последний элемент массива b

console.log(b.pop()); // удалили последний элемент и вывели его значение (х)

console.log(b); //вывели массив b после метода pop();

// (10) ['a', 'b', 'c', 'd', 'e', 'f', 'w', 't', 'q', 'j']

//оператор delete a[3]; - оператор удалит элемент массива a с индексом 3;

// заменит значение в массиве на пустой (empty) элемент, не изменяя длину массива.

delete a[3];

console.log(a);

// (13) [1, 2, 3, empty, 5, 6, 7, 8, 9, 10, 11, 71, 91]

//метод splice() - изменяет содержимое массива, удаляя существующие элементы и/или добавляя новые.

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

//где

//start - начальный индекс элемента,

//deleteCount - количество удаляемых после start элементов.

//item1 - добавляемые элементы

a.splice(3, 3, 'hi');

console.log(a);

//(11) [1, 2, 3, 'hi', 7, 8, 9, 10, 11, 71, 91]

Методы shift() и unshift() работают аналогично методам pop() и push().

Метод shift() удаляет первый элемент из массива и возвращает его значение. Этот метод изменяет длину массива.

Метод unshift() добавляет один или более элементов в начало массива и возвращает новую длину массива.

Метод reverse() на месте обращает порядок следования элементов массива. Первый элемент массива становится последним, а последний — первым. Возвращает перевернутый массив

console.log(a);

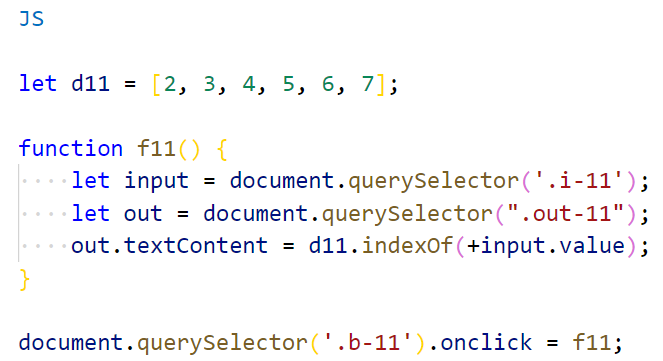
//(9) [1, 2, 3, 4, 5, 6, 7, 8, 9]

a.reverse();

console.log(a);

//(9) [9, 8, 7, 6, 5, 4, 3, 2, 1]

Метод indexOf() возвращает первый индекс, по которому данный



элемент может быть найден в массиве или -1, если такого индекса нет.

HTML

<div class="form-control">indexOf <input type="text" class="i-11"></div>

<button class="button-primary b-11">Task-11</button>

<div class="out-11"></div>

При вводе в input значения, проверит массив d11 на введенное

значение, если оно есть ( например ввели 3 ), то выведет индекс

этого значения ( т.е. 1 ), при отсутствии такого значения,

вернёт -1.

Метод concat() возвращает новый массив, состоящий из массива, на котором он был вызван, соединённого с другими массивами и/или значениями, переданными в качестве аргументов.

const array1 = ['a', 'b', 'c'];

const array2 = ['d', 'e', 'f'];

const array3 = array1.concat(array2);

console.log(array3);

// expected output: Array ["a", "b", "c", "d", "e", "f"]

Метод includes() определяет, содержит ли массив определённый элемент, возвращая в зависимости от этого true или false.

const array1 = [1, 2, 3];

console.log(array1.includes(2));

// expected output: true

console.log(array1.includes(9));

// expected output: false

Метод join() объединяет все элементы массива (или массивоподобного объекта) в строку.

const elements = ['Fire', 'Air', 'Water'];

console.log(elements.join());

// expected output: "Fire,Air,Water"

console.log(elements.join('-'));

// expected output: "Fire-Air-Water"

Пример комбинации методов:

let d15 = [0, 2, 5, -4, 6, 22, -9, -12, 8, 12, 13, 78];

function f15() {

    let input = document.querySelector('.i-15');

    for (let i = 0; i < d15.length; i++) {

        if (d15.indexOf(+input.value) == -1) {

            d15.push(+input.value);

        }

    }

    showArr('.out-15', d15); // внешняя, вспомогательная функция для вывода на страницу.

}

document.querySelector('.b-15').onclick = f15;

Данный код, проверит введённое в input значение, если такого нет в массиве d15, то добавит его (значение) к данному массиву d15.

Пример эмуляции метода indexOf();

let d12 = [6, 62, 60, 70, 1, 5];

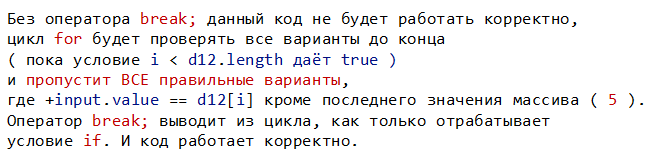
function f12() {

  let input = document.querySelector('.i-12');

  let out = document.querySelector(".out-12");

  for (let i = 0; i < d12.length; i++) {

      if (+input.value == d12[i]) {



          out.textContent = i;

          break; // ВАЖНЫЙ МОМЕНТ !

     } else {

          out.textContent = -1;

       }

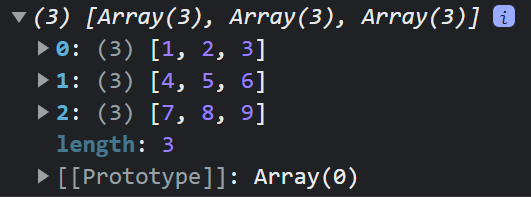
   }

}

document.querySelector('.b-12').onclick = f12;

**Многомерные массивы**

Массивы могут содержать элементы, которые тоже являются массивами. Это можно использовать для создания многомерных массивов, например, для хранения матриц:



let b = [

    [1, 2, 3],

    [4, 5, 6],

    [7, 8, 9],

];

console.log(b); // выведет весь массив

Работа с многомерными(двумерными) массивами аналогична обычным, за исключением, что используется два параметра при обращении.

console.log(b[1][1]); // выведет 5

Так же работа аналогична и с циклами.

let out = '';



for (let i = 0; i < b.length; i++) { // i – является индексом для значений основного массива

    for (let k = 0; k < b[i].length; k++) { // k – индекс для значений дочерних элементов

        out += b[i][k] + ' '; // ротация i и k позволяют перебирать массивы

    }

    out += '<br>';

}

document.querySelector('.out').innerHTML = out;

Данный цикл, вывел на страницу двумерный массив.

out = ''; // обнуляем out

for (let i = 0; i < b.length; i++) {

    for (let k = 0; k < b[i].length; k++) {

        if (b[i][k] > 4) { // добавляем условие if, которое будет отсекать все значения меньше 4.

            out += b[i][k] + ' ';

        }



    }

    out += '<br>';

}

document.querySelector('.out').innerHTML = out;

Итоговый вывод, похож на предыдущий, отсеклись все значения не удовлетворяющие if

Бонусная задача на перемещение единицы, с помощью массива.



let d = [1, 0, 0, 0, 0]; // исходный массив

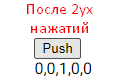
document.querySelector('.out2').innerHTML = d; //вывод массива на страницу

let k = 0; // вспомогательная переменная

document.querySelector('button').onclick = () => { //функция

    if (k + 1 < 5) { // условие ограничивает 1 в пределах длинны массива

        d[k] = 0; //0-ому элементу массива присваивается нуль



        d[k + 1] = 1; // элементу k(0)+1 присваивается единица ( 1 )

        k++; // увеличиваем k на единицу.

    }

    document.querySelector('.out2').innerHTML = d; // вывод массива после перемещения 1.

По нажатию на button, запустится функция, которая будет перемещать 1 на следующее место ( вместо нуля ), а вместо единицы образуется 0 ( нуль ).

Пример “шахматная доска”

let a13 = []; массив, в которой будет помещены дочерние массивы.

function f13() {

    let out = document.querySelector(".out-13")

    let s = 0; // создание вспомогательной переменной с первым значением для первого дочернего массива

    for (let i = 0; i < 8; i++) { // запуск цикла, отвечающий за родительский массив ( строки )

let temp = []; //создаем массив, он будет вложен в массив a13, как один из дочерних.

При каждой итерации цикла, его значения будут добавляться к родительскому, в виде

вложенного массива, а при последующей он будет обнуляться в исходное значение temp = [].

for (let k = 0; k < 8; k++) {

            temp.push(s);

            if (s == 0) {

                s = 1;

            } else {

                s = 0;

            }

        }

        if (s == 0) {

            s = 1;

        } else {

            s = 0;

        }

        a13.push(temp);

    }

    for (let i = 0; i < a13.length; i++) {

        out.innerHTML += a13[i].join(" ") + "<br>"

    }

    console.log(a13);

}

document.querySelector('.b-13').onclick = f13;

Объекты

(ассоциативные массивы)

Объекты используются для хранения коллекций различных значений и более сложных сущностей.

Объект может быть создан с помощью фигурных скобок {…} с необязательным списком свойств. Свойство – это пара «ключ: значение», где ключ – это строка (также называемая «именем свойства»), а значение может быть чем угодно.

Пустой объект («пустой ящик») можно создать, используя один из двух вариантов синтаксиса:

let user = new Object(); // синтаксис "конструктор объекта"

let user = {}; // синтаксис "литерал объекта"

Обычно используют вариант с фигурными скобками {...}. Такое объявление называют *литералом объект*а или *литеральной нотацией*.

Литералы и свойства

При использовании литерального синтаксиса {...} мы сразу можем поместить в объект несколько свойств в виде пар «ключ: значение»:

let user = { // объект

name: "John", // под ключом "name" хранится значение "John"

age: 30 // под ключом "age" хранится значение 30

};

У каждого свойства есть ключ (также называемый «имя» или «идентификатор»). После имени свойства следует двоеточие ":", и затем указывается значение свойства. Если в объекте несколько свойств, то они перечисляются через запятую.

В объекте user сейчас находятся два свойства:

Первое свойство с именем(ключом) "name" и значением "John".

Второе свойство с именем(ключом) "age" и значением 30.

Мы можем в любой момент добавить в него новые папки, удалить папки или прочитать содержимое любой папки.

Для обращения к свойствам используется запись «через точку»:

console.log(user.name) // выведет в консоль значение ключа name – John

Значение может быть любого типа. Давайте добавим свойство с логическим значением:

user.isAdmin = true; //в объект user добавлен ключ isAdmin со значением логики true

Для удаления свойства мы можем использовать оператор delete:

delete user.age; //из объекта user удалён ключ age со значением 30

Имя свойства (ключ) может состоять из нескольких слов, но тогда оно должно быть заключено в кавычки:

let user = {

name: "John",

age: 30,

"likes birds": true // имя свойства из нескольких слов должно быть в кавычках

};

К ключам типа “like birds” необходимо обращаться через квадратные скобки [].

// получение значения свойства

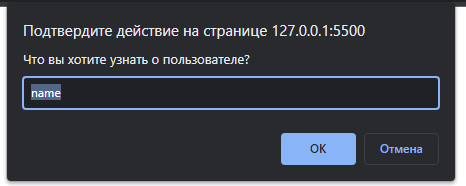
alert(user["likes birds"]); // true

// удаление свойства

delete user["likes birds"];

Так же с помощью [] можно обращаться к свойствам, ключ которой( имя ) было результатом выражение и имя свойства хранится в переменной.

let key = "likes birds";



// то же самое, что и user["likes birds"] = true;

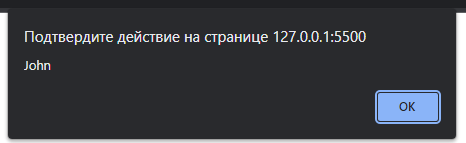
user[key] = true;

let user = {

    name: "John",

    age: 30

};



let key = prompt("Что вы хотите узнать о пользователе?", "name");

// доступ к свойству через переменную

alert(user[key]); // John (если ввели "name")

Последнее свойство объекта может заканчиваться запятой. Это называется «висячая запятая».

Значения ключей объекта, объявленного через const, может быть изменено.

const user = {

name: "John"

};

user.name = "Pete"; // значение ключа name было перезаписано.

Примеры

вспомогательная функция, в которую мы сможем подставлять разные параметры и она будет отрабатывать.

function f5(arr, block) {

    let out = '';

    for (let key in arr) { //проверка ключей в объекте

        out += `${key} : ${arr[key]} <br>`;

    }

    document.querySelector(block).innerHTML = out;

}

При нажатии на кнопку, массив a5 подставится в функцию выше и функция f5 выведет результат тела в .out-5

document.querySelector('.b-5').onclick = () => {

    let a5 = {

        "one": 1,

        "two": 2,

    }

    f5(a5, '.out-5'); // задаем функции f5 параметры

}

Вспомогательная функция, которая будет проверять условие if, если оно даст true вернёт true, если условие в if не будет давать true, то выйдет из цикла for и вернёт false.   
**ВАЖНО!** if...else не отработает корректно при работе с объектом, циклом мы должны перебрать все условия дающие true, и если такого не нашлось, то после него выводим false.

function f10(arr, val) {

    for (let key in arr) {

        if (arr[key] == val) {

            return true

        }

    }

    return false

}

document.querySelector('.b-10').onclick = () => {

    let a10 = {

        "k": 22,

        "d": 54,

        "m": 22,

    }

    document.querySelector('.out-10').innerHTML = f10(a10, 54);

};

// Task 20

// При нажатии b-20 выполняете функцию f20. Функция должна в out-20 вывести название станции которые содержат переход на другую ветку. Такие станции маркируются 2. Вывод через пробел

let a20 = {

    "red": [['Akademmistechko', 1], ['Nyvky', 0], ['Universytet', 3], ['Lisova', 1]],

    "blue": [['Minska', 1], ['Obolon', 0], ['Pochaina', 2], ['Holosiivska', 0]],

    "green": [['Syrets', 1], ['Zoloti Vorota', 2], ['Klovska', 0], ['Vidubichi', 1]],

}

function f20() {

    let out = document.querySelector(".out-20");

    let out\_1 = "";

    for (let key in a20) {

        for (let i = 0; i < a20[key].length; i++) {

            if (a20[key][i][1] == 2) {

                out\_1 += a20[key][i][0] + " ";

            }

        }

    }

    out.innerHTML = out\_1;

}

document.querySelector('.b-20').onclick = f20

API погода

Сайт ( сервис ) предоставляющий данные о погоде - <https://openweathermap.org/>

Обращение к сервису происходит через API, который является протоколом взаимодействия

В JS используем метод fetch(), который формирует запрос на сервере и подгрузит ответ на страницу.

Для начала рассмотрим строку запроса, помещённую в fetch().

[https://api.openweathermap.org/data/2.5/weather?lat=55.0794&lon=38.7783& appid=257bbb36fe65249c5e2821a206c56d88](https://api.openweathermap.org/data/2.5/weather?lat=55.0794&lon=38.7783&limit=5&appid=257bbb36fe65249c5e2821a206c56d88)

где:

https://api.openweathermap.org – адрес сервера который предоставляет API

/data – адрес отвечающий за API

/2.5 – версия API

/weather – часть URL адреса, указывающая, что обращаемся к погоде.

Всё что ниже – это GET запрос ( начинается со знака вопроса (?) )

с параметрами разделенные амперсандом ( & )

? – начало get запроса

lat=55.0794 – параметр широты

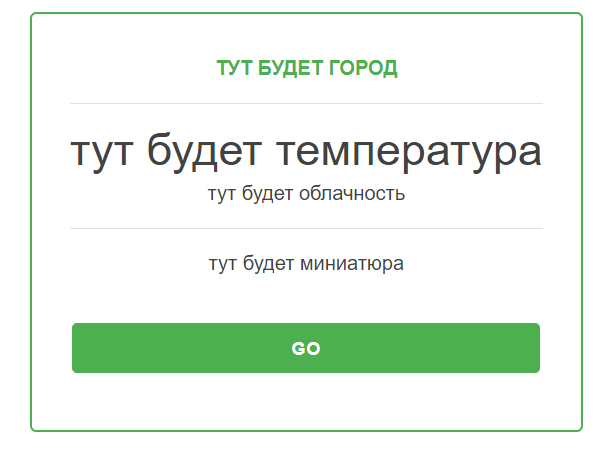
& - амперсанд

lon=38.7783 – параметр долготы

& - амперсанд

appid=257bbb36fe65249c5e2821a206c56d88 – уникальный ключ

**HTML**



<div class="pricing-table row">

            <div class="package featured">

                <p class="package-name">тут будет город</p>

                <hr>

                <p class="price">тут будет температура</p>

                <p class="disclaimer">тут будет облачность</p>

                <hr>

                <ul class="features">

                    <li>тут будет миниатюра</li>

                </ul>

                <button class="button-primary">Go</button>

            </div>

        </div>

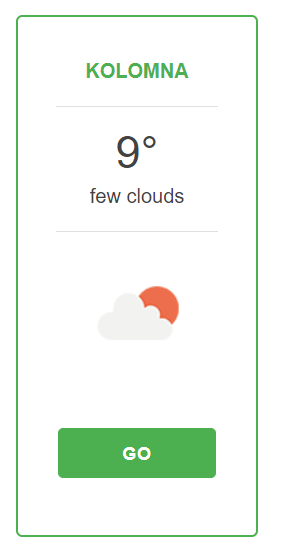
**JS**

fetch('https://api.openweathermap.org/data/2.5/weather?lat=55.0794&lon=38.7783& appid=257bbb36fe65249c5e2821a206c56d88')

    .then(function (resp) { return resp.json() })

    .then(function (data) {

        console.log(data); ***// (см. прим)***



        document.querySelector('.package-name').textContent = data.name;

        document.querySelector('.price').innerHTML = Math.round(data.main.temp - 273) + '&deg;';

        document.querySelector('.disclaimer').textContent = data.weather[0]['description'];

        document.querySelector('.features li').innerHTML =

`<img src="https://openweathermap.org/img/wn/${data.weather[0]['icon']}@2x.png">`;

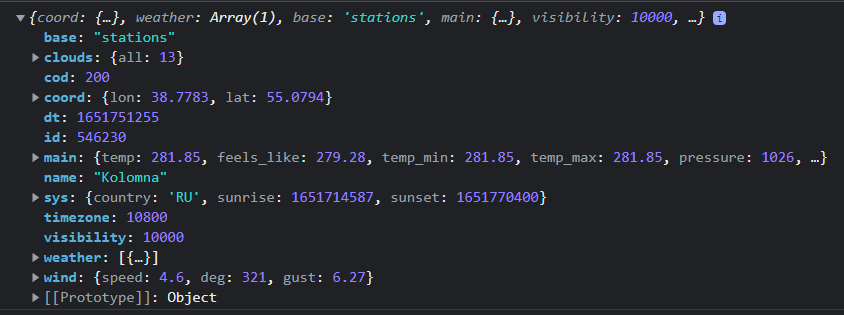
    })

***Прим.***

Принимаем пока метод fetch() и идущий до console.log как магию !

Далее, console.log(data) – выведет нам все возможные параметры

для вывода в виде массива ( объекта )



Далее, мы просто находим интересующий нас ключ. Например ключ main содержит информацию о температуре

( в кельвинах ) далее в коде мы используем этот параметр и переводит его в градусы Цельсия.

И добавляем его в нужный тег в HTML разметке.

***Все обращение к элементам объекта идёт по типу – data.KEY***

где, key – интересующий нас ключ.

Приложение Погоды

Пример выполнения приложения – погоды. CSS условно не показан

HTML

            <div class="row">

                <div class="col col-auto">

                        <select id="city">

                            <option value="703448">Киев</option>

                            <option value="2673730">Стокгольм</option>

                            <option value="2988507">Париж</option>

                        </select>

              </div>

                <div class="col col-auto">

                    <div class="pricing-table">

                        <div class="package featured">

                            <p class="package-name format\_text">тут будет город</p>

                            <hr>

                            <p class="price format\_text">тут будет температура</p>

                            <p class="disclaimer format\_text">тут будет облачность</p>

                            <hr>

                            <div class="features">

                                тут будет миниатюра

                            </div>

                        </div>

                    </div>

                </div>

            </div>

JS

let menu = document.querySelector("#city")

function getWeather() {

   let cityId = menu.value;

   fetch(`https://api.openweathermap.org/data/2.5/weather?id=${cityId}&appid=257bbb36fe65249c5e2821a206c56d88`)

      .then(function (resp) { return resp.json() })

      .then(function (data) {

         document.querySelector('.package-name').textContent = data.name;

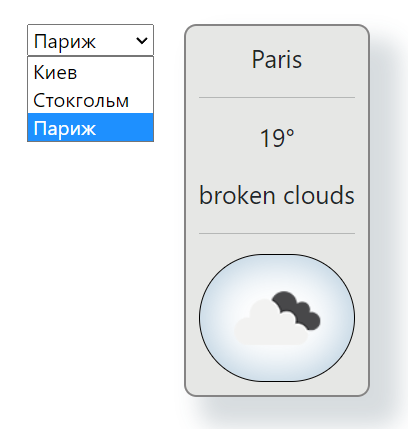
         document.querySelector('.price').innerHTML = Math.round(data.main.temp - 273) + '&deg;';

         document.querySelector('.disclaimer').textContent = data.weather[0]['description'];

         document.querySelector('.features').innerHTML = `<img src="https://openweathermap.org/img/wn/${data.weather[0]['icon']}@2x.png">`;

      })

}



menu.onchange = getWeather;

getWeather();

В menu мы ловим выбранный из списка город.

Далее создаем функцию которая будет отвечать за

логику приложения.

И после функции, создаем событие onchange,

которое отрабатывает при смене объекта в

списке. (menu.onchange = getWeather;)

Но что бы у нас изначально приложение было

заполнено данными, после события ( можно

и перед ) добавляем вызов функции

getWeather(); - которое заполнит приложение

данными из первого option – т.е Киев.

**Map и Set**

Map – это коллекция ключ/значение, как и Object. Но основное отличие в том, что Map позволяет использовать ключи любого типа.

Методы и свойства:

new Map() – создаёт коллекцию.

map.set(key, value) – записывает по ключу key значение value.

map.get(key) – возвращает значение по ключу или undefined, если ключ key отсутствует.

map.has(key) – возвращает true, если ключ key присутствует в коллекции, иначе false.

map.delete(key) – удаляет элемент по ключу key.

map.clear() – очищает коллекцию от всех элементов.

map.size – возвращает текущее количество элементов.

let map = new Map();

map.set("1", "str1"); // строка в качестве ключа

map.set(1, "num1"); // цифра как ключ

map.set(true, "bool1"); // булево значение как ключ

Перебор Map

Для перебора коллекции Map есть 3 метода:

map.keys() – возвращает итерируемый объект по ключам,

map.values() – возвращает итерируемый объект по значениям,

map.entries() – возвращает итерируемый объект по парам вида [ключ, значение], этот вариант используется по умолчанию в for..of.

Объект Set – это особый вид коллекции: «множество» значений (без ключей), где каждое значение может появляться только один раз.

Его основные методы это:

new Set(iterable) – создаёт Set, и если в качестве аргумента был предоставлен итерируемый объект (обычно это массив), то копирует его значения в новый Set.

set.add(value) – добавляет значение (если оно уже есть, то не добавляет), возвращает тот же объект set.

set.delete(value) – удаляет значение, возвращает true, если value было в множестве на момент вызова, иначе false.

set.has(value) – возвращает true, если значение присутствует в множестве, иначе false.

set.clear() – удаляет все имеющиеся значения.

set.size – возвращает количество элементов в множестве.

Перебор объекта Set

Мы можем перебрать содержимое объекта set как с помощью метода for..of, так и используя forEach:

let set = new Set(["апельсин", "яблоко", "банан"]);

for (let value of set) alert(value);

// то же самое с forEach:

set.forEach((value, value\_2, set) => {

alert(value);

});

Функция в forEach у Set имеет 3 аргумента: значение value, потом снова то же самое значение

value\_2 (не обязательный), и только потом целевой объект. Второй аргумент нужен для конвертации с объектом Map.

Set имеет те же встроенные методы, что и Map:

set.values() – возвращает перебираемый объект для значений,

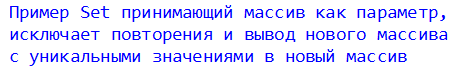
set.keys() – то же самое, что и set.values(), присутствует для обратной совместимости с Map,

set.entries() – возвращает перебираемый объект для пар вида [значение, значение], присутствует для обратной совместимости с Map.

Пример работы с Set

let arr = [1, 2, 3, 1, 4, 5, 'hello', 5, 1, 3];

let b = new Set(arr); // создаем set c параметром arr



console.log(b); // выведет set из 6 элем

console.log(b.size); //длинна 6

let bArr = Array.from(b); // создаем массив из set

console.log(bArr); // выведет как обычный массив

console.log(bArr[2]); // выведет 3 элемент массива

Способы перебора массивов

for(i), for(..in..), for(..of..)

HTML

<body>

    <p>1</p>

    <p>2</p>

    <p>3</p>

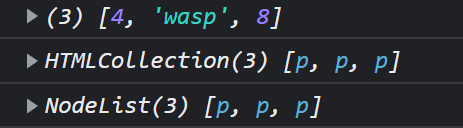
    <script src="script.js"></script>

</body>

JS

let arr = [4, "wasp", 8];

let allP = document.getElementsByTagName("p");



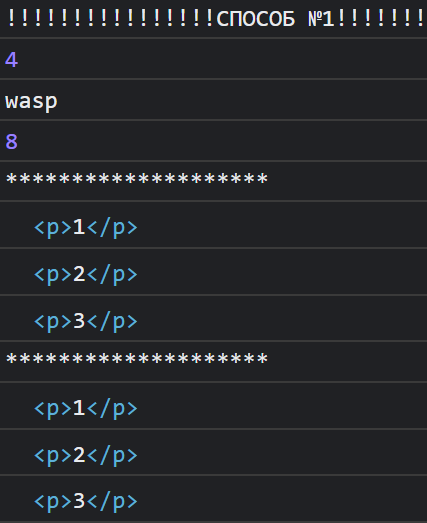
let qP = document.querySelectorAll("p");

console.log(arr);

console.log(allP)

console.log(qP);

//1способ



console.log("!!!!!!!!!!!!!!!!СПОСОБ №1!!!!!!!!!!!!!!!!!!!!!!")

for (let i = 0; i < arr.length; i++) {

    console.log(arr[i]);

}

console.log("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*")

for (let i = 0; i < allP.length; i++) {

    console.log(allP[i]);

}

console.log("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*")

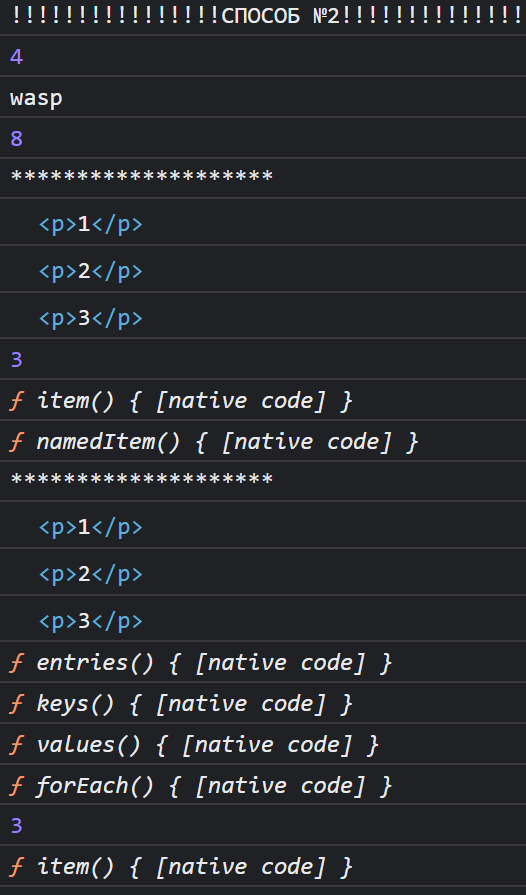
for (let i = 0; i < qP.length; i++) {

    console.log(qP[i]);

}

//2способ

console.log("!!!!!!!!!!!!!!!!СПОСОБ №2!!!!!!!!!!!!!!!!!!!!!!")



for (let key in arr) {

    console.log(arr[key]);

}

console.log("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*")

for (let key in allP) {

    console.log(allP[key]);

}

console.log("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*")

for (let key in qP) {

    console.log(qP[key]);

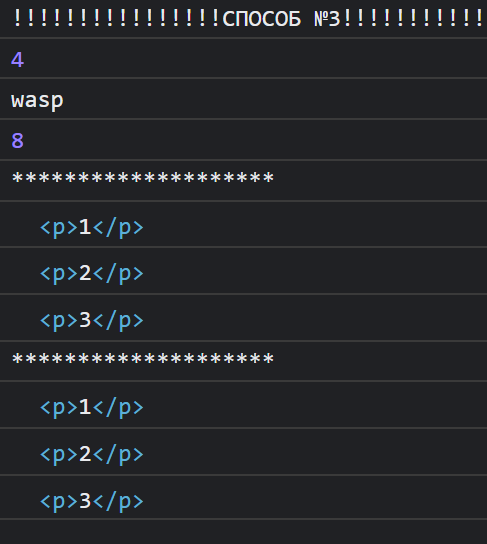
}

//3способ

console.log("!!!!!!!!!!!!!!!!СПОСОБ №3!!!!!!!!!!!!!!!!!!!!!!")

for (let item of arr) {

    console.log(item);



}

console.log("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*")

for (let item of allP) {

    console.log(item);

}

console.log("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*")

for (let item of qP) {

    console.log(item);

}

Методы массива map() и filter()

Map()

Метод map() создаёт новый массив с результатом вызова указанной функции для каждого элемента массива.

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

***Синтаксис***

const new\_array = arr.map(function callback( element[, index[, array]]) {

    // Возвращает элемент для new\_array

}

***Параметры***

**callback**

Функция, вызываемая для каждого элемента массива arr. Каждый раз, когда callback выполняется, возвращаемое значение добавляется в new\_array.

Функция callback, создающая элемент в новом массиве, принимает три аргумента:

element Текущий обрабатываемый элемент массива.

index (Необязательный) Индекс текущего обрабатываемого элемента в массиве.

array (Необязательный) Массив, по которому осуществляется проход.

***Возвращаемое значение***

Новый массив, где каждый элемент является результатом callback функции.

***Пример***

let a = [4, 5, 12, 200, 1, 0, -2]; // исходный массив

let b = a.map(function (item) { //метод map работает со значениями массива а. Функция может быть анонимной

    return item \* 5;// return вернёт новый массив b со значениями из a умноженные на 5

});

( let b = a.map(item => item \* 5); // короткая форма, когда параметр один и тело в одну строчку )

Filter()

Метод filter() создаёт новый массив со всеми элементами, прошедшими проверку, задаваемую в передаваемой функции.

Параметры и возвращаемое значение идентично методу map()

***Описание***

Метод filter() вызывает переданную функцию callback один раз для каждого элемента, присутствующего в массиве, и создаёт новый массив со всеми значениями, для которых функция callback вернула значение, которое может приведено к true. Функция callback вызывается только для индексов массива с уже определёнными значениями; она не вызывается для индексов, которые были удалены или которым значения никогда не присваивались. Элементы массива, не прошедшие проверку функцией callback, просто пропускаются и не включаются в новый массив.

let a = [4, 5, 12, 200, 1, 0, -2]; // исходный массив

let c = a.filter(function (item) {

    if (item > 0) { // проверка условия

        return true; //все что даст true в условии выше, попадёт в новый массив c.

    }

});

console.log(c);

Проверка на простое число

let foo = [-3, -2, -1, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13];

let bar = foo.filter(big100); // методу передана функция

function big100 (elem) { // параметр elem присваивается автоматически как каждый элемент массива

  if (elem % 2 !== 0  ) return elem > 1;

}

console.log(bar);// [3,5,7,9,11,13]

Проверка на тип данных и метод filter().

Определить тип данных можно с помощью typeof.

console.log(typeof 42);

// expected output: "number"

console.log(typeof 'blubber');

// expected output: "string"

console.log(typeof true);

// expected output: "boolean"

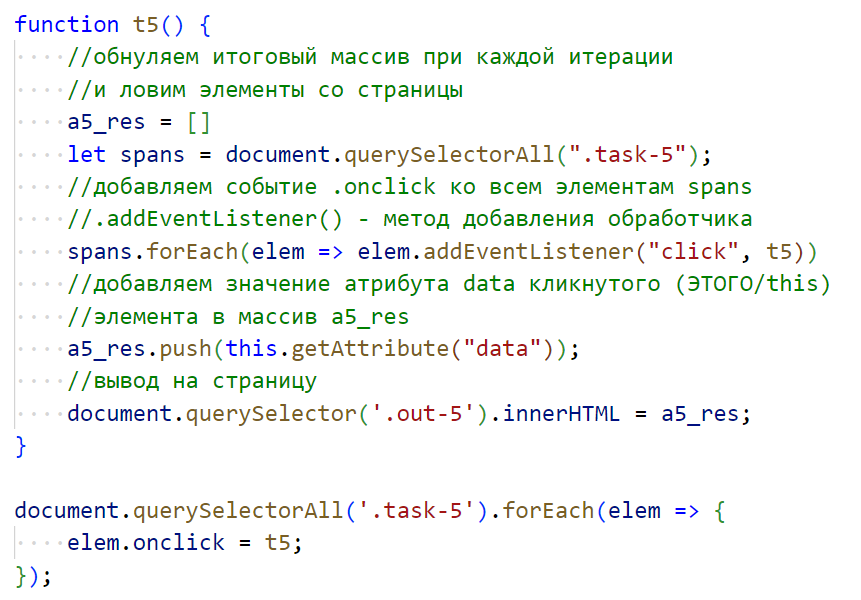
console.log(typeof undeclaredVariable);

// expected output: "undefined"

Соответственно и проверить тип данных и вывести определенный тип, так же можно используя typeof в синтаксисе if...else, или аналогичном.

**ЭТОТ ПРИМЕР К forEach()**

let a4 = [4, "3", 6, 7, "12", 34, "56", 78, 90, 11];



let a4\_res = [];

function t4() {

    a4\_res = a4.filter(function (item) {

        if (typeof item == "number") return true;

    });

    return a4\_res

};

document.querySelector('.b-4').onclick = () => {

    console.log(t4());

}

// (7) [4, 6, 7, 34, 78, 90, 11]

Если (typeof item == "number") даёт true, в данном случае, будут возвращены и добавлены в новый массив все значение исходного массива, удовлетворяющие данному условию, за фильтрацию исходного массива отвечает filter(), но делает он это с помощью typeof в данном случае.

Метод forEach()

Метод forEach() выполняет указанную функцию один раз для каждого элемента в массиве.

***Синтаксис – аналогичен методам map() и filter()***

arr.forEach(function callback(currentValue, index, array) {

//your iterator

});

***Описание***

Метод forEach() выполняет функцию callback один раз для каждого элемента, находящегося в массиве в порядке возрастания. Она не будет вызвана для удалённых или пропущенных элементов массива. Однако, она будет вызвана для элементов, которые присутствуют в массиве и имеют значение undefined.

Обращение к элементам в массиве, происходит не по аналогии с методами map() и filter(). Например, что бы перевести все элементы массива в нижний регистр.

let a9 = ['Quantum Break', 'Gears of War 4', 'Mass Effect: Andromeda', 'Far Cry Primal'];

function t9() {

    a9.forEach((elem, index) => { // обращение в методе аналогично map() и filter()

        a9[index] = elem.toLowerCase(); // что бы применить к каждому элементу массива метод

toLowerCase(), необходимо указать что мы ищем в массиве с

таким то индексов и присваиваем такое-то значение, элементу с

таким индексом. Смесь цикла for и методов

    });

    return a9;

}

document.querySelector('.b-9').onclick = () => {

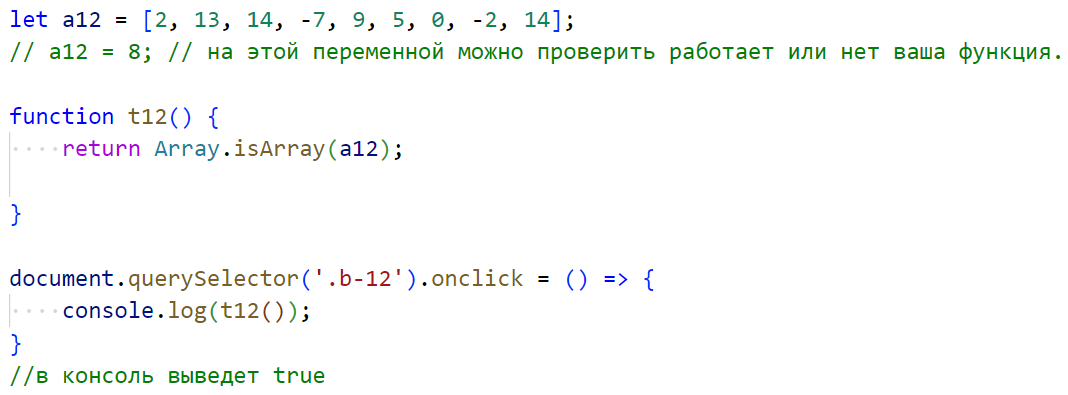
    console.log(t9());

}

результат

// (4) ['quantum break', 'gears of war 4', 'mass effect: andromeda', 'far cry primal']

Array.isArray()

Метод Array.isArray() возвращает true, если объект является массивом и false, если он массивом не является.

***Синтаксис***

Array.isArray(obj)

***Параметры***

obj - *Объект для проверки.*

***Возвращаемое значение***

true если объект является Array;

иначе, false.

includes()

Метод includes() определяет, содержит ли массив определённый элемент, возвращая в зависимости от этого true или false.

***Синтаксис***

arr.includes(searchElement[, fromIndex = 0])

***Параметры***

searchElement *Искомый элемент*.

fromIndex *Необязательный*

Позиция в массиве, с которой начинать поиск элемента searchElement. При отрицательных значениях поиск производится начиная с индекса array.length + fromIndex по возрастанию. Значение по умолчанию равно 0.

***Возвращаемое значение***

Boolean.

let a13 = [2, 13, 14, -7, 9, 5, 0, -2, 14]; // исходный массив в котором ищем

let a13\_num = 9; // что ищем

function t13() {

    return a13.includes(a13\_num); //тело функции которое вернет true или false

}

document.querySelector('.b-13').onclick = () => { //вызов функции

    console.log(t13());

}

//в консоль выведет true.

Метод split().

Метод split() разбивает объект String на массив строк путём разделения строки указанной подстрокой.

***Синтаксис***

str.split([separator[, limit]])

***Параметры***

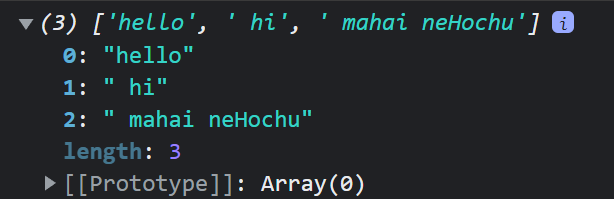
separator Необязательный параметр.Указывает символы, используемые в качестве разделителя внутри строки. Параметр separator может быть как строкой, так и регулярным выражением. Если параметр опущен, возвращённый массив будет содержать один элемент со всей строкой. Если параметр равен пустой строке, строка str будет преобразована в массив символов.

limit Необязательный параметр. Целое число, определяющее ограничение на количество найденных подстрок. Метод split() всё равно разделяет строку на каждом сопоставлении с разделителем separator, но обрезает возвращаемый массив так, чтобы он содержал не более limit элементов.

***Описание***

Метод split() возвращает новый массив.

Если разделитель separator найден, он удаляется из строки, а подстроки возвращаются в массиве. Если разделитель опущен, массив будет содержать только один элемент, состоящий из всей строки. Если разделитель является пустой строкой, строка str будет преобразована в массив символов.



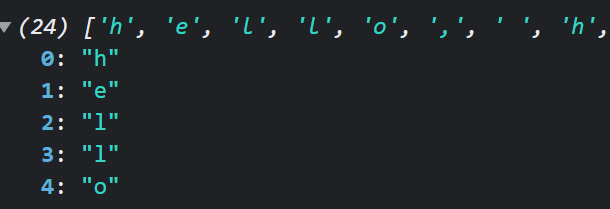
let a1 = "hello, hi, mahai neHochu"

console.log(a1.split([',']))

в выводе мы указали метод .split с разделителем “,”.

Без указания длины разбивки limit. ( разбил на массив по “,”)

let a1 = "hello, hi, mahai neHochu"



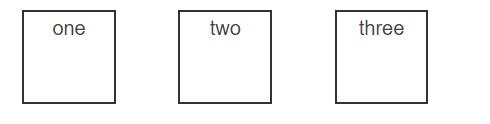
console.log(a1.split([' '[1]]))

здесь уже разделитель – пустая строка “ ”, а limit = 0.   
Подобный код разобьёт любую строку по 1 символу в массиве.

События мыши.

***.onclick, .ondblclick, .oncontextmenu, .onmousemove, .onmouseenter, .onmouseleave, .onmousedown, .onmouseup***

***HTML***



<div class="block one">one</div>

<div class="block two">two</div>

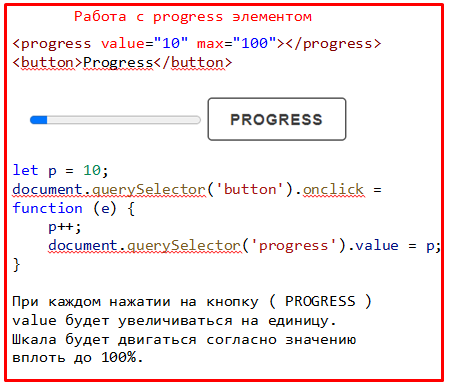
<div class="block three">three</div>

***.onclick***

Событие срабатывает по нажатию на элемент левой кнопкой мышки.  
В примере ниже событию .onclick присвоена функция с параметром event и ниже в консоль, этот параметр был вызван. В консоль выведется вся информация о элементе события .onclick, в виде объекта ключ:знач.

document.querySelector('.one').onclick = function (event) {

    console.log(event); // вывод объекта



    console.log('click'); //просто счётчик клика в консоли

}

***.ondblclick***

Событие сработает при двойном клике на эллемент

document.querySelector('.two').ondblclick = () => {

    console.log('double');

}

***.oncontextmenu***

Событие отработает при нажатии правой кнопки мыши.   
Так же тут добавлен return false; который предотвратит

всплытие “браузерного окна”.

document.querySelector('.two').oncontextmenu = () => {

    console.log('right button');

    return false;

}

***.onmousemove***

Событие отрабатывается, когда курсор попадает на элемент, и вызывается каждый раз, при движении мышки.

Событие, не отрабатывает каждый пиксель, и должно применяться с учётом этого. ( в зависимости от скорости движения мыши на объекте, чем медленней – тем больше событий отработается )

let w = 75; // исходная ширина блока

document.querySelector('.three').onmousemove = () => {

    document.querySelector('.three').style.width = w + 'px';

    w++; // прибавляем 1 к значению w

}

***.onmouseenter***

Событие срабатывает, когда курсор попадает на элемент ( похож на hover эффект в CSS )

document.querySelector('.three').onmouseenter = () => {

    document.querySelector('.three').style.background = 'red';

    console.log(1);

}

***.onmouseleave***

Событие срабатывает, когда курсор “уходит” с элемента ( похож на hover эффект в CSS )

document.querySelector('.three').onmouseleave = () => {

    document.querySelector('.three').style.background = 'green';

    console.log(2);

}

***.onmousedown***

Событие срабатывает, когда элемент нажат и еще не отпущен ( похож на active эффект в CSS )

document.querySelector('.three').onmousedown = () => {

    document.querySelector('.three').style.background = 'orange';

    console.log(2);

}

***.onmouseup***

Событие срабатывает, когда элемент нажат и уже отпущен ( похож на visited эффект в CSS )

document.querySelector('.three').onmouseup = () => {

    document.querySelector('.three').style.background = 'blue';

    console.log(3);

}

Unit 19\_задача 7, 8, 9, 16-17,

События клавиатуры

.onkeypress, .onkeydown, .onkeyup

.onkeypress

Событие keypress происходит когда нажимается символьная клавиша, то есть клавиша которая создаёт символ. Пример таких клавиш это буквы, цифры, знаки пунктуации и т.д. Примеры клавиш которые не создают символы, это клавиши модификаторы, такие как: Alt, Shift, Ctrl, или Meta , стрелки.

Событие onkeypress, как бы включает в себя сразу два события нажатую ( keydown ) и отпущенную (keyup) клавишу.

HTML

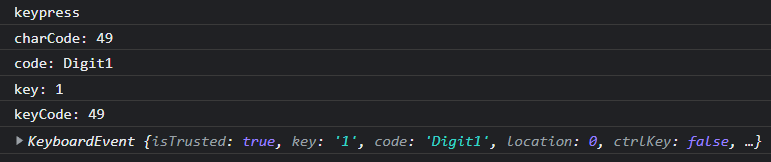


<input type="text" class="i-1">

JS

document.querySelector('.i-1').onkeypress = function (event) {

 console.log('keypress');



 console.log('charCode: ' + event.charCode);

 console.log('code: ' + event.code);

 console.log('key: ' + event.key);

 console.log('keyCode: ' + event.keyCode);

 console.log(event)

}

Вводим в input единицу ( 1 ). Получаем вывод в консоль. Где последняя строка, это общие сведенья о нажатой клавише. Все доступные ей свойства. Строки ***charCode*** и ***key***, так же ***code*** и ***keyCode*** по сути являются аналогами друг друга. Работают по разному в разных браузерах и так же результат вызова этих свойств иной при событиях .onkeydown и .onkeyup. Они являются основными идентификаторами этих клавиш, объявляют их номер и описывают клавишу.

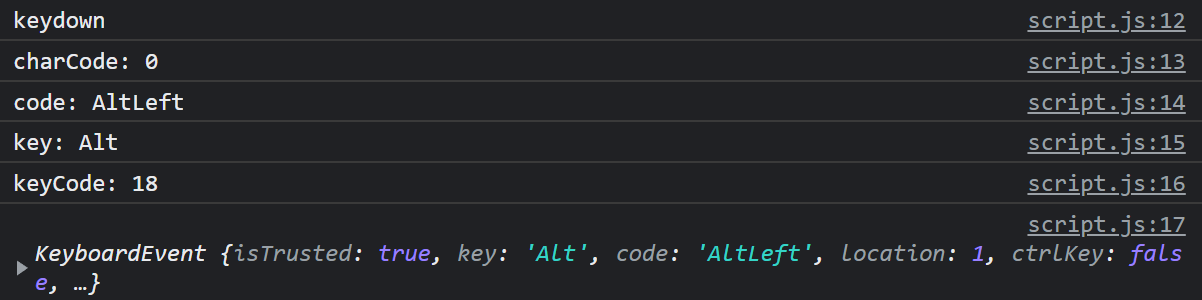
.onkeydown И .onkeyup

События .onkeydown срабатывает в момент нажатия, .onkeyup срабатывает в момент отпускания клавиши

данные события срабатывает и с Alt, Shift, Ctrl, или Meta , стрелки.

document.querySelector('.i-1').onkeydown = function (event) {

    console.log('keydown');



    console.log('charCode: ' + event.charCode);

    console.log('code: ' + event.code);

    console.log('key: ' + event.key);

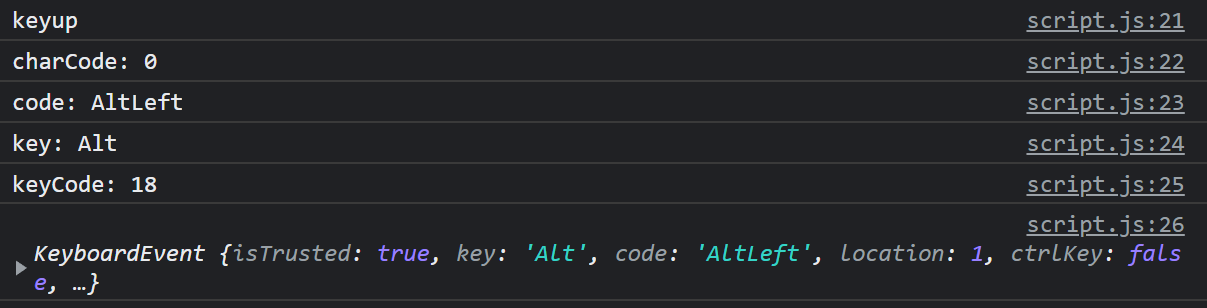
    console.log('keyCode: ' + event.keyCode);

    console.log(event);

}

document.querySelector('.i-1').onkeyup = function (event) {

    console.log('keyup');



    console.log('charCode: ' + event.charCode);

    console.log('code: ' + event.code);

    console.log('key: ' + event.key);

    console.log('keyCode: ' + event.keyCode);

    console.log(event);

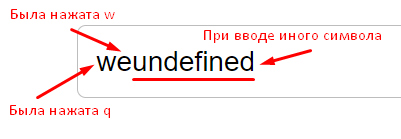
}

Первое событие отработало при нажатии, второе сразу после отжатия клавиши

Пример смещения символов.

document.querySelector('.i-1').onkeypress = function (event) {

    const a = {



        q: 'w',

        w: 'e'

    }

    document.querySelector('.i-1').value += a[event.key];

    return false;

}

При вводе в input символа ***q***, она будет заменена на ***w***, при вводе ***w*** она будет заменена на ***e***.

Любые другие введённые символы, дадут ***undefined***.

Метод EventTarget***.addEventListener()***

Метод EventTarget.addEventListener() регистрирует определённый обработчик события, вызванного на ***EventTarget***.

***EventTarget*** - может быть Element, Document, Window, или любым другим объектом, поддерживающим события (таким как XMLHttpRequest (en-US)).

***Методы***

EventTarget***.addEventListener()*** Регистрирует обработчик событий указанного типа на объекте.

EventTarget.removeEventListener() Удаляет обработчик события.

EventTarget.dispatchEvent() Генерирует событие на объекте EventTarget.

***Синтаксис*** ***.addEventListener()***

target.addEventListener(type, listener[, options]);

***Параметры***

***type*** - Чувствительная к регистру строка, представляющая тип обрабатываемого события.

***listener*** - Объект, который принимает уведомление, когда событие указанного типа произошло. Это должен быть объект, реализующий интерфейс EventListener или просто функция JavaScript.

***options*** – (Необязательный) Объект options, который определяет характеристики объекта, прослушивающего событие.

( читать больше... [https://developer.mozilla.org/ru/docs/Web/API/EventTarget/addEventListenerhttps://developer.mozilla.org/ru/docs/Web/API/EventTarget/addEventListener](https://developer.mozilla.org/ru/docs/Web/API/EventTarget/addEventListenerhttps:/developer.mozilla.org/ru/docs/Web/API/EventTarget/addEventListener) )

События touchstart, touchend, touchmove.

( читать больше ...<https://developer.mozilla.org/ru/docs/Web/API/Touch_events> )

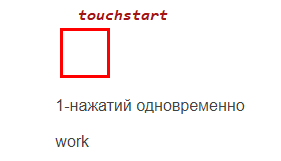
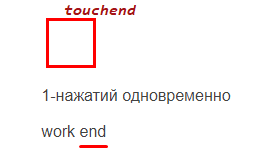
События касаний подобны событиям мыши за исключением того, что они поддерживают несколько одновременных касаний в разных местах сенсорной поверхности.

Базовые приёмы.

HTML

            <div class="row">

                <div class="block-1"></div>



                <div class="col">

                    <div class="out-1"></div>

                </div>

                <div class="col">

                    <div class="out-2"></div>

                </div>

            </div>

            <div class="row">

                <div class="block-2"></div>

                <div class="out-3"></div>

            </div>

JS

**Срабатывание при нажатии**

document.querySelector('.block-1').addEventListener("***touchstart***", myTouch);

function myTouch(event) {

    console.log(event);

    console.log('touch');

    document.querySelector('.out-1').innerHTML = event.touches.length;

    document.querySelector('.out-2').innerHTML += 'work ';

}

**Срабатывание при отжатии**

document.querySelector('.block-1').addEventListener("***touchend***", myTouchEnd);

function myTouchEnd() {

    document.querySelector('.out-2').innerHTML += 'end ';

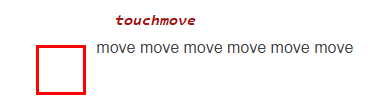
}

**Срабатывание при движении**

document.querySelector('.block-2').addEventListener("***touchmove***", myTouchMove);

function myTouchMove(event) {

    event.preventDefault();



    console.log(event);

    document.querySelector('.out-3').innerHTML += 'move ';

    return false;

}

Отлов ошибок

try{}, catch{}, finally{}

(читать тут... <https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Statements/try...catch>)

Конструкция try...catch пытается выполнить инструкции в блоке try, и, в случае ошибки, выполняет блок catch.

Конструкция try содержит блок try, в котором находится одна или несколько инструкций (Блок ({} ) обязательно должен присутствовать, даже если выполняется всего одна инструкция), и хотя бы один блок catch или finally. Таким образом, есть три основные формы конструкции try:

1. try {...} catch {...}
2. try {...} finally {...}
3. try {...} catch {...} finally {...}

Блок catch содержит инструкции, которые будут выполнены, если в блоке try произошла ошибка. Если любая инструкция в блоке try выбрасывает исключение, то управление сразу же переходит в блок catch. Если в блок try не было выброшено исключение, то блок catch не выполняется.

Блок finally выполнится после выполнения блоков try и catch, но перед инструкциями, следующими за конструкцией try...catch. Он выполняется всегда, в независимости от того, было исключение или нет.

Вы можете использовать вложенные конструкции try. Если внутренняя конструкция try не имеет блока catch (такое может быть при её использовании в виде try {...} finaly {...}, потому что try {...} не может быть без блоков catch или finally), будет вызван сatch внешней конструкции try.

Конструкция try также используется для обработки исключений JavaScript (то есть, выброшенных внутренними функциями языка или парсером).

***Условный блок catch***

"Условные блоки catch" можно создавать, используя try...catch с if...else if...else, как здесь:

***try {***

***myroutine();*** // может выбрасывать три вида исключений

***} catch (e) {***

***if (e instanceof TypeError) {***

// обработка исключения TypeError

***} else if (e instanceof RangeError) {***

// обработка исключения RangeError

***} else if (e instanceof EvalError) {***

// обработка исключения EvalError

***} else {***

// обработка остальных исключений

***logMyErrors(e);*** // передать обработчику ошибок

***}***

***}***

***HTML***

    <div class="test"></div>

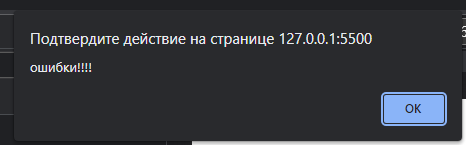
***JS***

let a = 5;

a = a \* 2;

try {

    document.querySelector('.test1').innerHTML = a; // !



}

catch (err) {

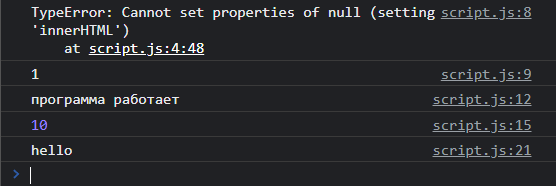
    alert('ошибки!!!!');

    console.log(err);

    console.log('1');

}

finally {



    console.log('программа работает');

}

console.log(a);

t1();

function t1() {

    console.log('hello');

}

LocalStorage

LocalStorage – хранилище данных внутри браузера. Сохранение происходит в формате string, но при использовании JSON-метода( JSON.stringify(x) + JSON.parse(x) ) - LocalStorage можно использовать для хранения массивов и объектов в их функциональном виде.

Следующий код показывает пример доступа к локальному объекту Storage для текущего домена и добавляет данные в него с помощью Storage.setItem().

***localStorage.setItem('myCat', 'Tom');***

Считывать данные из localStorage для определённого ключа, можно следующим образом:

***let cat = localStorage.getItem('myCat');***

Удалять данные можно так:

***localStorage.removeItem('myCat');*** // вернёт undefined

Для удаления всех записей, то есть полной очистки localStorage, используйте:

***localStorage.clear();***

Добавить в локальную память объект с: key = data; value = 5;

Данные будут в формате строка.

localStorage.setItem('data', 5);

(7)console.log(localStorage.getItem('data'));

Добавить данные в формате объекта с помощью JSON;

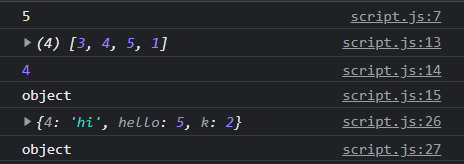
const a = [3, 4, 5, 1];

localStorage.setItem('a', JSON.stringify(a));

let b = localStorage.getItem('a');

b = JSON.parse(b);

(13)console.log(b);



(14)console.log(b[1]);

(15)console.log(typeof b);

const c = {

    hello: 5,

    k: 2,

    4: 'hi'

};

localStorage.setItem('c', JSON.stringify(c));

let d = localStorage.getItem('c');

d = JSON.parse(d);

(26)console.log(d);

(27)console.log(typeof d);

Связать две страницы друг с другом. Событие на одной – даст результат на второй.

HTML-1

<div class="out"></div>

JS-1

window.addEventListener('storage', function (e) {

    console.log('change');

    document.querySelector('.out').textContent = localStorage.getItem('b1');

});

HTML-2

<button class="button-primary ls">Go</button>



JS-2

let count = 0;

document.querySelector('.ls').onclick = () => {

    count++;

    localStorage.setItem('b1', count);



}

При нажатии на 2-ой странице, в 1-ую добавляется счётчик

нажатий. Так же в консоли отображается событие change

Выражения и операторы

В JS существует несколько типов операторов. Каждый из типов имеет приоритет друг над другом.

-Операторы присваивания

-Операторы сравнения

-Арифметические операторы

-Битовые(поразрядные) операторы

-Логические операторы

-Строковые операторы

-Условный (тернарный) оператор

-Оператор запятая

-Унарный оператор

-Операторы отношения

-Приоритет операторов

JS поддерживает бинарные и унарные операторы, а так же специальный – тернарный(условный) оператор.

Бинарная операция использует два операнда(operand), один ПЕРЕД оператором(operator) и ДРУГОЙ за ним:

operand1 operator operand2

Например: 3+4; x\*y; где 3,4,x,y – это operand, а +,\* - это operator;

Унарная операция использует один операнд, перед или после оператора:

operator operand или operand operator

Например: х++ ; ++х; - где x – это operand, а ++ - это operator;

**Приоритет операторов (operator precedence )**



Большинство **выражений**, таких как умножение и присвоение, являются **операторами-выражениями**. Таким образом, **выражение** умножения состоит из **оператора умножения**(звездочка) и двух **операндов** (умножаемых значений, которые сами могут быть выражениями )

Операторами не являются два выражения: определение идентификатора(т.е. имён переменных и констант) и

литеральные выражения

Функции (function)

-это блок программного кода на языке JS, который определяется один раз и может применяться многократно.

( Но функцию можно перезаписать, переназначить подобно переменной, но в отличии от переменной, функция имеет глобальное значение ВНЕ зависимости от расположение внутри кода. Т.е. до вызова или после. Функция всегда будет исполняться по ПОСЛЕДНЕЙ итерации. С перезаписью функций нужно быть аккуратным )

Создание функции. Метод 1

**function declaration**

Ниже, мы определяем(создаем) две функции с именами testFunc и hello

function – ключевое слово для деклорации функции.

testFunc/hello – имена функций, могут быть любыми(кроме зарезервированных имен )

() – круглые скобки, есть обязательная часть функций, существуют для передачи в функцию параметров. Если параметров нет, скобки остаются пустыми.

{...} – внутри фигурных скобок определяется ТЕЛО ФУНКЦИИ, которое будет отрабатывать при вызове функции.

function testFunc() {

   const x = 5;

   const y = 10;

   console.log(x + y);

}

function hello() {

   console.log("Hello");

}

Ниже, мы запускаем функции

Функцию, можно запускать в любом месте! До объявление этой функции, после и даже внутри другой функции.

testFunc(); //15

hello(); //Hello

Ниже, мы провоцируем вызов функции testFunc\_hello, ДО её фактического объявлении в коде.

(Движок V8, который интерполирует наш код, читает ВЕСЬ код JS до начала выполнения. И фиксирует все задекларированные имена функций. Что бы, если во время исполнения кода, появится вызов, еще не задекларированной по ходу кода функции, программа могла обратиться к ней, фактически еще не добравшись последовательно к её объявлению.)

testFunc\_hello();//вызов функции, которая в процессе отработки своего тела, запустит последовательно еще две функции.

function testFunc\_hello() {

   testFunc();

   hello();

   console.log("Выше, мы запустили две функции с помощью третьей(ЭТОЙ)")

}

//main.js:5 15

//main.js:8 Hello

//main.js:17 Выше, мы запустили две функции с помощью третьей(ЭТОЙ)

Можно определять функцию, внутри функции.

function insideFunc() {

   console.log("Внутри меня объявлена еще одна функция.");

   crazyInside();

   function crazyInside() {

      console.log("Я ОБъявлена ВНУТРИ!!!!");

   }

}

insideFunc();

//main.js:28 Внутри меня объявлена еще одна функция.

//main.js:31 Я ОБъявлена ВНУТРИ!!!!

Вызвать функцию, которая была определена(задекларированна) внутри другой функции, можно только в рамках этой функции. Глобальной видимости, такая функция не имеет. Т.е. её вызов приведёт к ошибке.

main.js:35 Uncaught ReferenceError: crazyInside is not defined

at main.js:35:1

Напишем функцию, выводящую время unix в консоль:

function unixTime() {

   let time = Math.floor(new Date().getTime());

   console.log(time);

}

unixTime();

//main.js:40 1667736587837

Напишем функцию генерирующую случайное число в заданном диапазоне:

function randomInt() {

   const min = 100;

   const max = 300;

   let rand = Math.floor(min + Math.random() \* (max + 1 - min));

   console.log(rand);

}

randomInt();

Если вызвать функцию, без круглых скобок (), то выведется листинг функции.

console.log(randomInt);

main.js:48 199 // вызов функции с () выводит результат.

main.js:51 ƒ randomInt() { // вызов без (), выводит листинг функции.

const min = 100;

const max = 300;

let rand = Math.floor(min + Math.random() \* (max + 1 - min));

console.log(rand);

}

Переопределение функции.

Ниже, мы переопределили тело функции randomInt. Назначив в её тело вывод строки в консоль.

Это приведет к тому, что все последующие и предыдущие вызовы данной функции, будут отрабатывать ПОСЛЕДНЮЮ итерацию тела функции.

function randomInt() {

   console.log("Я фейковый случайный генератор");

}

randomInt();

//main.js:54 Я фейковый случайный генератор

//main.js:51 ƒ randomInt() {

console.log("Я фейковый случайный генератор");

}

//main.js:54 Я фейковый случайный генератор

Как видно из логов консоли, вызов на 51 строке, вывел функцию с новой итерацией тела, а 48 строка заменилась на 54. Из чего можно сделать вывод, что предыдущий вывозов функции, определив, что существует несколько итераций данной функции, выбрал последнюю и вывел её в консоль.

Переменные и функции.

Можно объявлять переменные для функции снаружи.

let min = 100;

let max = 300;

function randFunc() {

   let rand = Math.floor(min + Math.random() \* (max + 1 - min));

   console.log(rand);

}

randFunc();

\*const был заменен на let, для дальнейших манипуляций

с телом функции, сейчас это не важно.

//main.js:64 172

Данная функция, работает аналогично функции randomInt(), за исключением того, что переменные объявлены не внутри тела функции, а снаружи функции. Логика функции такова, что не найдя заложенные в неё переменные внутри тела, она будет искать их снаружи.

Переопределив значения переменных ниже по коду, и вызвав функцию, то она отработает уже с новыми значениями переменных, что делает данный вариант более гибкий и юзабельный.  
**НО переменные должны быть объявлены ДО вызова функции!!!**

min = 10;

max = 50;

randFunc();

//main.js:64 48

Рассмотрим функцию счётчик ( ошибочную, с точки зрения задачи функции ):

function count\_x\_inside() {

   let xInside = 9;

   xInside++;

   console.log(xInside);

}

count\_x\_inside();

count\_x\_inside();

//main.js:75 10

//main.js:75 10

Как видно из выводов в консоль выше, мы дважды вызвали функцию count\_x\_inside(), но значение НЕ поменялось, это случилось от того, что при каждом вызове, переменная xInside имеет локальную область видимости и цикл жизни в один вызов функции. Т.е. при вызове функции count\_x\_inside(), начинается чтения тела этой функции, в которой обозначено создание переменной xInside со значением **9**. Далее мы прибавляем к ней единицу xInside**++** и выводим в консоль. Всё. Значение xInside нигде не сохраняется. Следовательно при последующих вызовах xInside по прежнему будет иметь начальное значение.

Для создания счётчика, необходимо, что бы переменная xInside была вынесена из тела функции.

Функция ниже, реализует функционал счётчика, за счёт вынесенной из тела функции переменной xOutside

let xOutside = 9;

function count\_x\_outside() {

   xOutside++;

   console.log(xOutside);

}

count\_x\_outside();

count\_x\_outside();

count\_x\_outside();

//main.js:85 10

//main.js:85 11

//main.js:85 12

Когда решается задача, необходимо понимать, что требуется:

Если требуется, что бы функция выполнила задачу(отработала тело) и не влияла на глобальные переменные, не имела “пост эффектов”, то всё определяется внутри тела функции.

Если же, требуется что бы результат предыдущего запуска сохранялся для следующего ( например счётчик ), то нужно вынести переменную за тело функции.

**Параметры и аргументы функции**

Параметры функции позволяют объявить функцию с формальными переменными, а затем в момент вызова функции, передать в эту функцию фактические аргументы.

**Параметры функции — это имена , перечисленные в определении функции.**

**Аргументы функции — это реальные значения , передаваемые (и получаемые) функцией.**

Вызов функции с аргументами, может быть произведён, ДО объявления этой функции последовательно по коду.

Количество параметров функции не ограничено. При вызове функции, эти аргументы передаются в функцию последовательно, в том порядке, котором они были объявлены в функции: (a,b) где при вызове а – нужно передать первым, b – вторым аргументом при вызове функции.

Объявление функции с формальными переменными:

function f1(a, b) {

   console.log(a + b);

}

//вызов функции с фактическими переменными

f1(5, 8); //main.js:2 13

f1(10, 20); //main.js:2 30

let z1 = 100;//объявили переменную и присвоили ей значение

f1(z1, 5); //main.js:2 105

f1(z1 \* 2, 2) //main.js:2 202

f1(5, 8, 9, 10); //main.js:2 13

f1(7); //main.js:2 NaN

Как видно из примеров выше, в качестве аргументов, может быть переданы, как значения так и переменные. Может быть передано аргументов больше, чем задано параметров у функции, но тогда функция отработает только с указанными по порядку аргументами ( 5 и 8 ). Так же, возможно передать и меньше параметров, но в нашем случае это привело к выводу NaN.

function f3(a, b, elem) {

   elem.textContent = a + b;

}

const out1 = document.querySelector(".out-1");

f3(5, 6, out1);

В качестве параметра, может указываться что угодно. В примере выше, мы передали переменную, которая ловит на странице div с классом out-1 и присвоили ему значение суммы a и b. (<div class="out-1">11</div>)

**Параметры по умолчанию**

Это механизм указания параметров, при котором значения параметров будут изначально заданы, но при вызове функции, можно передать собственные аргументы, которые заменят значения по умолчанию.

function f2(a = 0, b = 0) {

   console.log(a + b);

}

f2(7); //main.js:13 7

У функции **f2**, присутствует два параметра a и b, которые имеют значения по умолчанию 0. ( эти значения могут быть произвольными ). В случае, отсутствия значений по умолчанию у параметров, как в **f1**, вызов функции с одним аргументом привел бы к результату NaN, но у нас присутствуют значения по умолчанию. Это привело к тому, что а присвоился аргумент из вызова, b осталось со значением по умолчанию ( 0 ).

function f4(a = 0, b = 0, elem = ".out-2") {

   document.querySelector(elem).textContent = a + b;

}

f4(10, 5); // (<div class="out-2">15</div>)

При вызове функции **f4**, было передано два аргумента, которые заменили собой параметры a и b. Параметр elem не был задан, и принял значение по умолчанию.

**Arguments**

Объявим функцию **f5** с параметрами и вызовем её с аргументами.

function f5(a, b, elem) {

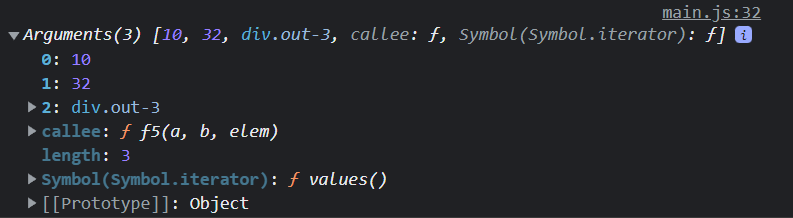
   elem.textContent = a + b;

   console.log(arguments);

}

f5(10, 32, out3); // (<div class="out-3">42</div>)

В теле функции, помимо присвоения элементу на странице, значения суммы параметров a и b. В консоль выводится arguments. В консоль вывелся набор значений похожих на массив.



Arguments - это специальная переменная (псевдомассив), которая позволяет получить доступ к любому аргументу. Псевдомассив — это объект, который структурно похож на массив. То есть у него есть числовые свойства (индексы) и свойство length.

Переменная arguments позволяет манипулировать аргументами, вне зависимости от их количества. Используя индекс данного аргумента. Данный синтаксис полезен, когда не известно количество аргументов или оно меняется в ходе работы.

Пример №1. Использование цикла и свойств arguments.

Данная функция, будет получать аргументы (параметры не заданы) и с помощью arguments сложит переданные аргументы.

function f6() {

   console.log(arguments);

   let sum = 0;

   for (let i = 0; i < arguments.length; i++) sum += arguments[i];

   console.log(sum);

}

f6(4, 6, 7, 10);

main.js:38 Arguments(4) [4, 6, 7, 10, callee: ƒ, Symbol(Symbol.iterator): ƒ] //первый вывод в консоль

main.js:41 27 //второй вывод в консоль

**Arguments и rest parameters**

Синтаксис **остаточных параметров (rest parameters)** функции позволяет представлять неограниченное множество аргументов в виде массива. И следовательно использовать методы массивов для манипуляции.

Пример №2. Использование **rest parameters** и метода массива.

Данная функция, будет получать аргументы, модифицировать полученный псеводомассив в массив, и далее манипулировать полученным массивом методами.

function f7(...args) {

   console.log(args);

   let sum = args.reduce((accum, item) => accum += item);

   console.log(sum);

}

f7(62, 6, 7, 10);

main.js:47 (4) [62, 6, 7, 10] //первый вывод в консоль

main.js:49 85 //второй вывод в консоль

**Функция в качестве аргумента.**

const out4 = document.querySelector(".out-4"); // переменная для вывода результата на страницу.

function likeArgumFunc(res) {

   out4.innerHTML = `<b>${res}<b>`

}

function mainFunc(num, drowFunc) {

   drowFunc(num);

}

mainFunc(45, likeArgumFunc); //(<div class="out-4"><b>45<b></div>)

Функция **mainFunc имеет два параметра num и** drowFunc. Тело функции состоит из:

вызова функции переданной в качестве аргумента - drowFunc с аргументом num.

Вызвав функцию **mainFunc** мы передали ей в качестве аргументов число 45 и функцию likeArgumFunc.

Получилась последовательность вызовов, где:

num = 45

drowFunc = likeArgumFunc

соответственно в теле функции **mainFunc**, при её вызове, образовалась следующая конструкция:

likeArgumFunc(45) – т.е. вызов другой функции (likeArgumFunc) с аргументом (45). А уже эта функция (**likeArgumFunc**), получив в параметр res, из функции **mainFunc** параметр num, который в свою очередь равен аргументу из вызова функции **mainFunc** - 45 и обрабатывает его согласно телу функции **likeArgumFunc**.

При использовании **rest parameters**, необходимо убедиться, что все аргументы, которые необходимо передать *ЯВНО* должны находиться *ПЕРЕД* **rest parameters (...num)**

function f8(drowFunc, ...num) {

   drowFunc(num);

}

f8(likeArgumFunc, 88, 11, 91, 100) //(<div class="out-4"><b>88,11,91,100<b></div>)

**Оператор return**

Этот оператор завершает выполнение функции и возвращает её значение. Все что будет ниже return не будет работать. Если нужно перенести строку с выражением после return нужно использовать ().

return (

выражение

);

Функция ниже возвращает значение 5. Оператор return позволяет использовать возвращаемое значение в математических выражениях:

function f1() {

   return 5

}

console.log(f1()); // main.js:5 5

console.log(4 \* f1()); // main.js:6 20

console.log(f1() / f1() + f1()); //main.js:7 6

Функция получает два параметра ( если нет, то отрабатывают по умолчанию ) суммирует их и возвращает значение:

function f2(a = 200, b = 100) {

   return a + b;

}

console.log(f2(2, 5) + f2(1, 1) \* f2(7)) // main.js:12 221

Функция ниже получает два числа и возвращает случайное:

function f3(min, max) {

   let rend = min + Math.random() \* (max + 1 - min);

   return Math.floor(rend);

}

console.log(f3(0, 255)); // main.js:18 128

Выводить значения с помощью return можно куда угодно:

document.querySelector(".out-1").textContent = f1(); - вывод на страницу

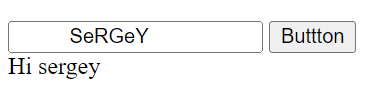
alert(f1()); - вывод в сплывающее окно

console.log(f1()); - вывод в консоль.

Код ниже ищет на странице элемент с классом “cube-1”. И обращается к его свойству background передавая цвет в rgb формате с помощью функции генерации случайных числе f3. В результате, элемент на странице будет получать случайный цвет при каждом обновлении страницы.

let cube1 = document.querySelector(".cube-1");

cube1.style.background = `rgb(${f3(0, 255)}, ${f3(0, 255)}, ${f3(0, 255)})`;



На странице существует форма ввода и вывода данных введенных

пользователем. Код ниже ищет кнопку по нажатию на которую

запускает функцию, которая берет значение из input

модернизирует его и выводит в вывод.

В анонимной функции присутствует два варианта вывода в консоль, с использованием вспомогательной функции и без.

document.querySelector(".b-2").addEventListener('click',

   function () {

      const input = document.querySelector(".i-2").value;

      document.querySelector(".out-2").innerHTML = "Hi " + clearText(input);

      console.log("Hi " + clearText(input));

      console.log("Hi " + input);

   })

function clearText(data) {

   return data.trim().toLowerCase()

}

//main.js:32 Hi sergey

//main.js:33 Hi SeRGeY

**Return прерывает функцию.**

Функция ниже, принимает два аргумента, массив и число и сравнивает. Если находит аналогичное число в массиве, то возвращает индекс этого массива, если нет то возвращает -1.

function indexOfEmul(arr, num) {

   for (let i = 0; i < arr.length; i++) {

      if (arr[i] === num) return `index = ${i}`;

   }

   return -1

}

console.log(indexOfEmul([22, 33, 44, 55], 33); // main.js:52 index = 1

console.log(indexOfEmul([22, 33, 44, 55], 66); // main.js:53 -1

Код ниже показывает год рождения пользователя в зависимости от введённых лет.

Ищем кнопку, создаем событие, получаем введенное значение, проводим проверку на строку и значения меньше нуля и больше ста, если пользователь ввел строку или число не в диапазоне 0=<year<100, то функция прерывается return в ином случае выводит год.

document.querySelector(".b-3").addEventListener('click',

   function () {

      let year = +document.querySelector(".i-3").value;

      if (isNaN(year)) return;

      if (year <= 0 || year > 100) return

      document.querySelector(".out-3").innerHTML = 2022 - year;

   })

Дан массив arr5, у него есть два вложенных массива. Код ниже суммирует значения вложенного массива.

const arr5 = [[3, 4, 5], [6, 7, 10]]

function t5() {

   let s = 0;

   for (let i = 0; i < arr5.length; i++) {

      s += sum(arr5[i]);

   }

   return s

}

function sum(arr) {

   let sum = 0;

   for (let i = 0; i < arr.length; i++) {

      sum += arr[i];

   }

   return sum

}

console.log(t5());

Код ниже, проверяет элементы вложенного массива - arr7, если сумма этих элементов четная, то добавляет их в новый массив – evenArr7. Тут была использована, уже ранее написанная функция sum().

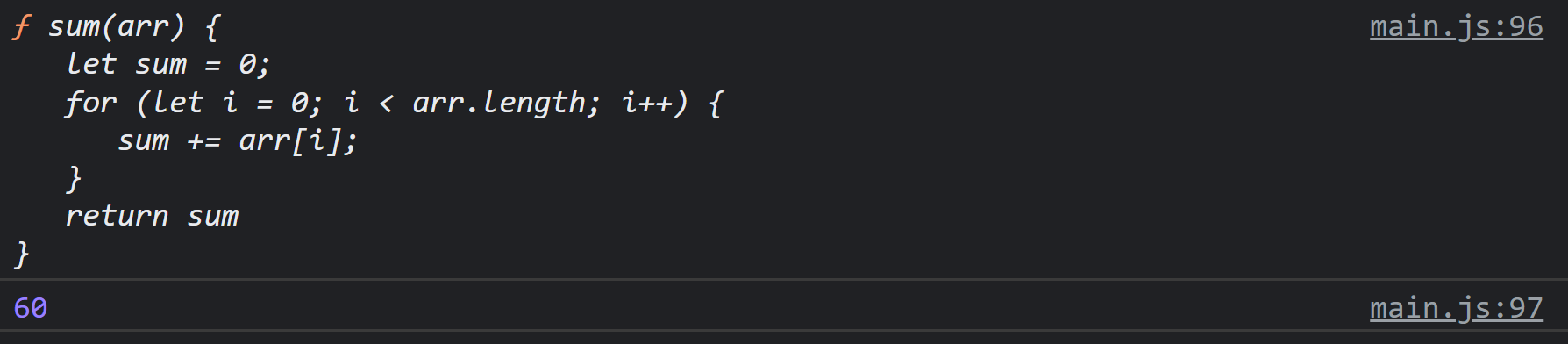
const arr7 = [[3, 4, 5], [5, 6, 2], [2, 1], [52, 14]];

const evenArr7 = arr7.filter(function (item) {

   return (sum(item) % 2 === 0);

});

console.log(evenArr7);



**Возвращение функции:**

function t8() {

   return sum;

}

let s = t8();

console.log(s); //main.js:96

console.log(s([10, 20, 30])); //main.js:97

Получается, что функция t8 вернула функцию sum. А ниже мы присвоили переменной s функцию t8().

Т.е s = sum. И мы можем передавать переменной s аргументы, аналогично, как мы действовали бы с функцией sum().

**Стрелочные функции**

Стрелочные функции – это аналог анонимных функций. Имеет более компактный синтаксис. И некие различия.

Объявим(задекларируем) классическую функцию суммы:

function sum(a, b) {

   return a + b;

};

Ниже представлены два варианта функции анонимная(classic) и анонимная(arrow).

*classic*

*arrow*

document.querySelector(".b-1").addEventListener("click", function () {

   let res = sum(23, 45);

   console.log(res);

});

document.querySelector(".b-1").addEventListener("click", () => {

   let res = sum(23, 45);

   console.log(res);

});

Обе они срабатывают по нажатию клавиши с классом b-1 и выводят в консоль результат суммы функции sum().

Ниже представлено два фрагмента кода, которые обрабатывают массивы **arr1** и **arr2** с помощью метода map().

В ***classic*** версии мы передаем в качестве параметра методу map() функцию, которую ранее задекларировали, а в версии arrow, передаем логику непосредственно как параметр методу map().

*classic*

*arrow*

const arr1 = [4, 5, 6, 7, 8];

console.log(arr1);

function pow1(a) {

   return a \*\* 2;

}

const res1 = arr1.map(pow1);

console.log(res1);

const arr2 = [4, 5, 6, 7, 8];

console.log(arr2);

const res2 = arr2.map((a) => {

   return a \*\* 2;

});

console.log(res2);

//если параметр(аргумент) один и строка тела тоже одна, то можно сократить до:

const res3 = arr2.map(a => a \*\* 2);

console.log(res3);

В варианте ***arrow***, есть возможность еще сократить синтаксис. При передаче одного параметра можно отказаться от круглых скобок (), и если тело функции состоит из одной строки ( или приводится к одной строке ), можно отказаться от фигурных скобок {...} и оператора return.

Ниже представлено два фрагмента кода, которые обрабатывают массивы **arr4** и **arr5** с помощью метода filter().

В ***classic*** версии в качестве параметра методу filter(), была передана анонимная функция.

В ***arrow*** в качестве параметра методу filter(), была передана стрелочная, причем сокращенная в части тела функции, так как логику можно привести к одной строке.

*classic*

*arrow*

const arr4 = [4, 5, 6, 7, 8];

const res4 = arr4.filter(function (item, index) {

   if (index % 2 === 0) {

      return true;

   }

})

console.log(res4);

const arr5 = [4, 5, 6, 7, 8];

const res5 = arr5.filter((item, index) => index % 2 === 0)

console.log(res5);

Ниже представлены различия между анонимной ***classic*** и arrow функциями с ***arguments*** и ***this***.

*classic*

*arrow*

// classic & this

document.querySelector(".b-2").addEventListener("click", function () {

   console.log(this);

});

// arrow & this

document.querySelector(".b-2").addEventListener("click", () => {

   console.log(this);

});

//стрелочная функция не имеет собственного this поэтому берёт this из глобальной области.

//classic & arguments

const t1 = function () {

   console.log(arguments)

console.log(arguments[1]) // 100

   console.log("work")

};

t1(99, 100); // выведется псевдомассив

//arrow & arguments

const t2\_1 = () => {

   console.log(arguments) – приведёт к ошибке

   console.log("work")

};

t2\_1(99, 100); // ОШИБКА!!!

const t2\_2 = (...rest) => {

   console.log(rest)

   console.log("work")

};

t2\_2(99, 100); //можно вывести через rest parametrs