JS

Лабораторная работа 3. Функции в js.

Создадим папку lr1. Откроем ее в vsCode (файл – открыть папку…). Это будет проектом. Создадим в нем файлы index.html и script.js.

Index.html

<html>

<head>

<meta charset="UTF-8">

    <script src="script.js"></script>

<title>Лабораторная работа 3</title>

</head>

<body>

</body>

</html>

**Выполнение.**

1. Функции.
   1. Функция – Именованный блок программного кода. В JS функция является объектом класса функции.

Для создания функции используется ключевое слово function. В () могут быть указаны параметры функции.

function myFunction(myParam) {

    return myParam;

}

Для вызова функции нужно написать ее имя() и указать параметры, если они есть.

function consoleOut(str) {

    console.log(str);

}

consoleOut('Hello world');

Функцию можно вызвать и до ее объявления.

getSum(10, 11);

function getSum(a, b) {

    console.log(+a + +b);

}

Ключевое слово return обозначает инструкцию завершения выполнения функции с возвращением значения. Функции, которые ничего не возвращают – возвращают undefined.

function getSumm(a, b) {

    return +a + +b;

}

console.log(getSumm(10, 15));

Параметром функции можно передать массив.

function getArr(array) {

    if (!Array.isArray(array)) {

        return false;

    }

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

        document.write('В ячейке ' + i + ' значение ' + array[i] + '<br>');

    }

    return true;

}

let arr = [12, 32, 43];

getArr(arr);

Т.к. JS не строго типизирован, стоит делать проверки типов данных, которые попадают в параметры.

JS не проверяет количество параметров, переданных в функцию

function getStr(str1, str2) {

    document.write('<h1>' + str1 + '</h1><br>' + '<h1>' + str2 + '</h1>');

}

getStr('Hello');

Вместо пропущенных параметров будет undefined.

Следует делать проверки для передаваемых параметров.

Все параметры, переданные в функцию хранятся в массиве arguments, который виден внутри функции.

function getStr() {

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

        document.write('<h1>' + arguments[i] + '</h1><br>');

    }

}

getStr('Hello', 'world');

Т.о. можно обрабатывать в функции произвольное количество параметров.

1.2Контекст вызова – это тот объект, относительно которого вызывается (к которому она применяется) функция.

Контекст вызова можно получить с помощью ключевого слова this.

function func() {

    console.log(this);

}

func();

В данном примере функция вызывается относительно всего окна и в консоли мы получим объект window.

Создадим объект, добавим ему метод и посмотрим, что будет находится в контексте вызова

let someObj = {};

someObj.name = 'Это наш объект';

someObj.do = function () { console.log(this); }

someObj.do();

В этом случае мы получим объект someObj.

1.3Присваивание функции переменной.

Функцию можно присвоить переменной, но в этом случае ее нельзя будет вызвать до этого присвоения.

let summ = function (a, b) { console.log(+a + +b); }

summ(5, 6);

Обратите внимание, что переменной в данном случае присваивается анонимная функция т.е. без имени. Это не является обязательным и имя можно указать, если это требуется.

let f = function factorial(a) { return (a !== 1) ? a \* factorial(a - 1) : 1; }

console.log(f(4));

Функцию можно вызвать на исполнение сразу при объявлении. Как правило это используется для единоразовых вызовов функции или для изолирования части когда.

let summ = (function (a, b) { return +a + +b; }(10, 15));

console.log(summ);

Переменной можно присвоить ссылку на функцию. Рассмотрим пример

function func() {

    console.log('Функция вызвана');

    return 'строка из функции';

}

let a = func();

console.log(a);

В этом случае мы объявили функцию, вызвали ее и присвоили переменной a то значение, которое вернула функция. Уберем () после присваивания функции

function func() {

    console.log('Функция вызвана');

    return 'строка из функции';

}

let a = func;

console.log(a);

Теперь мы присвоили переменной a не значение, которое вернула функция, а ссылку на нее.

Чтобы вызвать эту функцию нужно написать

function func() {

    console.log('Функция вызвана');

    return 'строка из функции';

}

let a = func;

a();

Рассмотрим пример присваивания ссылки на функцию свойству onClick

Внутри html документа разместим

<body>

    <div class="item" id = "it"></div>

</body>

Зададим ему размер и цвет фона

.item {

    width: 150px;

    height: 150px;

    background-color: gray;

}

window.onload = function() {*// функция срабатывает, когда страница загружена*

    let item = document.querySelector('.item'); *// возвращает объект DOM с классом*

    item.onclick = func;

}

function func() {

    console.log('Функция вызвана');

}

1.4Замыкания (вложенные функции) – объявление функции внутри другой функции.

Внутренняя функция видит все переменные внутри внешней функции т.е. это в некотором роде область видимости.

function myFunc(a, b) {

    function summ () {

        console.log(a + b);

    }

    let some = summ();

}

1.5 Функции обратного вызова

В JS функции могут принимать в качестве аргументов другие функции, а также возвращать функции в качестве результата. Такие функции называются функциями «высшего порядка». Функции, которые передаются, как аргумент называют callback (или функция обратного вызова).

function showStatus(name, status, callback) {

    if (callback && typeof callback === 'function') {

        callback();

    } else {

        console.log(`Имя: ${name}, статус: ${status}`);

    }

}

showStatus('N123', 'Online');

showStatus('', '', function () {

    console.log('Нет данных');

});

1.6 Стрелочные функции

В es6 появился новый способ создания функция при помощи оператора =>. Рассмотрим способы ее использования и отличия от function.

Функция принимает несколько параметров:

*// function sum(a, b) {*

*//  return a + b;*

*// }*

let sum = (a, b) => a + b;

console.log(sum(1, 3));

Функция принимает 1 параметр:

*// function oneParam(a) {*

*//  return a + b;*

*// }*

let oneParam = a => a;

console.log(oneParam(10));

Функция не принимает параметров:

let some = () => 'что-то';

console.log(some());

Функция не возвращает значения

*// function msg() {*

*//  console.log('сообщение');*

*// }*

let msg = () => console.log('сообщение');

console.log(msg());

Тело функции состоит из нескольких строк:

*// function sum(a, b) {*

*//  let res = a + b;*

*//  return res;*

*// }*

let sum = (a, b) => {

    let res = a + b;

    return res;

}

console.log(sum(1, 2));

Функция само вызывающаяся:

*// (function () {*

*//  console.log('Выполнено немедленно');*

*// })();*

(() => console.log('Выполнено немедленно'))();

Важной особенностью стрелочных функция является то, что их контекстом является окружающая среда объекта, а не объект вызова.

let person = {

    name: 'имя',

    func: function () {

        console.log(this.name);

    },

    arrFunc: () => {

        console.log(this.name);

    }

}

person.func();

person.arrFunc();

Рассмотрим проблему, которую решают стрелочные функции

let person = {

    name: 'имя',

    func: function () {

        let \_this = this;

        setTimeout(function () {

            console.log(\_this.name);

        }, 500);

    },

    arrFunc: function () {

        setTimeout(() => {

            console.log(this.name);

        }, 500);

    }

}

person.func();

person.arrFunc();

Благодаря им при использовании классом каких-то глобальных методов мы можем удобно обращаться к контексту классов.