REALFIVE
Учебно-практические курсы программирования

Практическое занятие №11 Функции

Задание №1

Убедитесь в том что у Вас запущен Open Server и папка с материалами урока размещена в директории /domains сервера. Заходим на http://js11.loc и начинаем урок.

Задание №2.

Функция – блок инструкций, который можно выполнить в любой момент, вызвав функцию. Этот блок инструкций может быть вызван многократно. Блок инструкций часто называют телом функции. Как правило, для функций так же, как и для переменных, используют имена.

Способы создания функций:

1. С помощью ключевого слова function (Function Declaration)

```
function f() {
 //тело функции
}
```

2. С помощью функционального литерала (Function Expression)

```
var f = function() {
 //тело функции
};
```

3. С помощью вызова класса конструктора (используется крайне редко)

```
var f = new Function('тело функции');
```

Последний способ нужно знать, но использовать не желательно по нескольким причинам. Первая – функция создается сразу в глобальной области видимости. Вторая – плохая читаемость кода в таком виде.

Именование функций. В качестве имени функции может использоваться любой допустимый идентификатор. Старайтесь выбирать функциям достаточно описательные, но не длинные имена. Искусство сохранения баланса между краткостью и информативностью приходит с опытом. Правильно подобранные имена функций могут существенно повысить удобочитаемость (а значит, и простоту сопровождения) ваших программ. Чаще всего в качестве имен функций выбираются глаголы или фразы, начинающиеся с глаголов. По общепринятому соглашению имена функций начинаются со строчной буквы. Если имя состоит из нескольких слов, в соответствии с одним из соглашений они отделяются друг от друга символом подчеркивания, примерно так: like_this(), по другому соглашению все слова, кроме первого, начинаются с прописной буквы, примерно так: likeThis(). Имена функций, которые, как предполагается, реализуют внутреннюю, скрытую от посторонних глаз функциональность, иногда начинаются с символа подчеркивания.

Задание №3.

Во всех трех случаях функция вызывается одинаково f();

Наберите следующий код и проверьте результат.

```
function f1() {
    console.log('f1');
}
f1();

var f2 = function () {
    console.log('f2');
};
f2();


var f3 = new Function("console.log('f3');");
f3();
```

Задание №4.

Также есть еще один вариант вызова. Функции в языке JavaScript являются объектами и подобно другим объектам имеют свои методы. В их числе есть два метода, call() и apply(), выполняющие косвенный вызов функции. Оба метода позволяют явно определить значение this (об этом дальнейших уроках).

```
function f1() {
    console.log('f1');
}
f1.call();
f1.apply();

var f2 = function () {
    console.log('f2');
};
f2.call();
f2.apply();


var f3 = new Function("console.log('f3');");
f3.call();
f3.apply();
```

Задание №5.

Function Declaration

Перед тем, как приступить к исполнению инструкций, интерпретатор JS сначала анализирует все декларации функций. Таким образом, при выполнении первой же инструкции все объявленные функции уже доступны.

```
f(); //-> f
function f(){
    console.log('f');
}
```

Function Expression

А вот инструкция, используемая для создания функции, выполняется в одном ряду с другими.

```
f1(); //ошибка: функция еще не существует
var f1 = function (){
    console.log('f1');
};
f1(); //-> f1
```

Тоже самое при создании функции через конструктор

```
f2(); //ошибка: функция не существует
var f2 = new Function('console.log("f2");');
```

Задание №6.

При вызове функции ей можно передать любое количество аргументов, что делает функции незаменимым инструментом программирования. Передача аргументов позволяет получать различные результаты в зависимости от переданных значений. Для передачи аргументов в момент определения функции в круглых скобках записывают имена. При вызове функции вместо этих имен подставляют значения.

```
function f(x, y) {
    console.log(x);
    console.log(y);
}
f(1, 2);
f(3, 4);
f('hello world', 33);
```

Метод call() позволяет передавать аргументы для вызываемой функции в своем собственном списке аргументов, а метод apply() принимает массив значений, которые будут использованы как аргументы. Первый аргумент этих функций позволяет определить контекст вызова – this (об этом дальнейших уроках).

```
function f2(x, y) {
    console.log(x);
    console.log(y);
}

var a = ['third', 'fourth']; //массив

f2.call(null, 'first', 'second');
f2.apply(null, a);
```

Задание №7.

Напишите функции:

1. Функция принимает 2 аргумента и выводит их сумму.
2. Функция принимает 1 аргумент и выводит его инвертированное значение.
3. Функция принимает 1 аргумент и выводит его абсолютное значение (модуль).
4. Функция принимает 2 аргумента и выводит наибольшее из них.
5. Функция принимает 2 аргумента и выводит наименьшее из них.

Под выводом понимается console.log();

Задание №8.

В отличие от многих других языков программирования JS позволяет передавать произвольное число аргументов, а не только заявленное в объявлении функции.

```
function f(x, y) {
    console.log(x + ', ' + y);

}

f(1, 2); //->1, 2
f(1); //1, undefined
f(1, 2, 3);//->1, 2
```

Если число аргументов в вызове функции превышает число имен параметров, функция лишается возможности напрямую обращаться к неименованным значениям. Для этого у каждой функции есть локальная переменная – объект arguments, предоставляющий доступ ко всем переданным аргументам. Для работы с ними мы можем использовать методы работы с массивами.

```
function outMax()
{
    if (arguments.length > 0) {

        for (var i = 1, max = arguments[0]; i < arguments.length; i++) {
            max = arguments[i] > max ? arguments[i] : max;
        }

        console.log(max);
    }

}


outMax(1, 2);
outMax(1);
outMax();
outMax(1, 2, 3, 4, 6, 7, 3, 8, 10, 15, 9);
outMax(20, 2, 3, 4, 6, 7, 3, 8, 10, 15, 9);
```

Изменение значения аргумента через массив arguments[] меняет значение, извлекаемое по имени аргумента.

```
function f2(x) {
    console.log(x);
    arguments[0] = null;
    console.log(x);
}
f2(5);
```

Задание №9.

С функциями связано понятие и области видимости переменных.

До применения функций мы создавали переменные в глобальном контексте. Эти переменные называются глобальными.

Переменная, создаваемая внутри тела функции с помощью оператора var, является локальной. Локальные переменные видны только внутри тела функции. Это означает следующее:

1. Невозможно обратиться к локальной переменной вне тела функции, в котором эта переменная была определена
2. Для названия локальных переменных можно использовать те же имена, что уже были использованы для глобальных переменных или внутри других функций

Переменная, объявленная без оператора var, автоматически является глобальной. Присвоив такой переменной некоторое значения, мы присвоим это значение глобальной переменной. Если глобальной переменной с таким именем не было, она будет создана и ее значение будет равно новому значению.

Ни в коем случае не следует определять локальные переменные без оператора var, так как это может привести к непредсказуемым последствиям.

Если внутри функции мы обращаемся к переменной с некоторым именем, эта переменная сначала ищется среди локальных переменных. Если такая переменная среди локальных переменных этой функции (ее контексте) не найдена, следует переход в контекст внешней функции и поиск среди переменных этого контекста. Эта операция происходит, пока программа не доходит до глобального объекта.

Все аргументы функции являются локальными переменными этой функции.

```
var g = 5;
function outer_function(x) {
    var y = 3;

    function inner_function() {
        var z = 8;
        console.log('inner_function x: ' + x);
        console.log('inner_function z: ' + z);
        console.log('inner_function y: ' + y);
        console.log('inner_function g: ' + g);
    }
    inner_function();
    //z не существует
    console.log('outer_function x: ' + x);
    console.log('outer_function y: ' + y);
    console.log('outer_function g: ' + g);
}
outer_function(6);

//z,x и y не существует
console.log('global g: ' + g);
```

Задание №10.

Поднятие переменных.

Интерпретатор JavaScript всегда незаметно для нас перемещает («поднимает») объявления функций и переменных в начало области видимости. Формальные параметры функций и встроенные переменные языка, очевидно, изначально уже находятся в начале. Это значит, что этот код:

```
function f() {
    console.log(x);
    var x = 1;
}
```

на самом деле интерпретируется так:

```
function f() {
    var x;
    console.log(x);
    x = 1;
}
```

Задание №11.

Функции как пространства имен

Предположим, например, что имеется модуль JS, который можно использовать в различных JS-программах (или, если говорить на языке клиентского JS, в различных веб-страницах). Допустим, что в программном коде этого модуля, как в практически любом другом программном коде, объявляются переменные для хранения промежуточных результатов вычислений. Проблема состоит в том, что модуль будет использоваться множеством различных программ, и поэтому заранее неизвестно, будут ли возникать конфликты между переменными, создаваемыми модулем, и переменными, используемыми в программах, импортирующих этот модуль. Решение состоит в том, чтобы поместить программный код модуля в функцию и затем вызывать эту функцию. При таком подходе переменные модуля из глобальных превратятся в локальные переменные функции

```
var a = 2;

function f2(){
    console.log(a);
}
f2(a);


(function mymodule() {

        var a = 1, b = 3;//существует только в модуле

        function f1(){ //существует только в модуле
            console.log(a);
        }

        function f2(){
            console.log(b);
        }

        f1();
        f2();


})();
```

Функция mymodule обернута в скобки и после ее объявления она сразу вызывается. Это называется немедленно вызываемой функцией. Немедленно вызываемая функция в JS – это синтаксическая конструкция, позволяющая вызвать функцию сразу же в месте ее определения.

```
//в данном случае функция анонимна,
//после ее выполнения вызвать ее еще раз невозможно
(function () {
      //тело
})();
```

Задание №12.

Инструкция return. Иногда нужно, чтобы результатом выполнения функции было некоторое значение, и это значение можно было бы присвоить некоторой переменной. В этом случае нужно применить инструкцию return. Эта инструкция прерывает выполнение функции и возвращает то, что написано правее. Если ничего справа не указывать, произойдет просто прерывание выполнения функции. В любом случае, выполнение программы вернется в ту точку, откуда эта функция была вызвана. Вот несколько примеров использования return.

```
function multiply(x, y){
    return (x * y);
}
console.log(multiply(3, 5));

function f(x){

    if(x === 1){
        return;
    }

    return true;

}
console.log(f(1));
console.log(f(2));
```

Задание №13

Напишите функции:

1. Функция, которая принимает произвольное количество аргументов и возвращает результат их перемножения. Если какой-либо аргумент не число или равняется 0 его необходимо пропустить при перемножении.
2. Функция, которая выводит первые переданных аргументов.

Задание №14

Замыкания.

Каким же образом внутри функции мы имеем доступ к тем же глобальным переменным или переменным, объявленным во внешней функции?

Каждая функция момент создания образует так называемый «лексический контекст» – специальный объект scope, в котором хранятся все локальные переменные этой функции. В этот контекст попадают все переменные ближайшего внешнего контекста и так далее по цепочке вплоть до глобального контекста. В результате мы получаем так называемую scope chains – цепочку объектов scope.

```
function f() {
    var z = 7;
    return function () {
        alert(z);
    };
}

var x = f();
x();
```

В момент создания анонимной возвращаемой функции в ее контекст попало значение внешней переменной z. Теперь мы можем вызывать эту функцию в любой момент времени – значение переменной z будет доступно при каждом вызове.

Этот процесс, при котором переменные внешнего контекста попадают в контекст создаваемой функции, и называется замыканием (closure). По определению замыкание – это функция, находящаяся внутри создаваемой функции, которая и передает значения внешних переменных в ее лексический контекст. Мы замыкаем значения переменных из внешних контекстов на саму функцию. Эти значения определяются на момент завершения работы ближайшего внешнего контекста.

Задание №15

Замыкание – очень мощный инструмент в JS, который позволяет закрыть множество недостатков этого языка. Например, с их помощью можно создавать приватные методы и свойства у объектов (об этом в дальнейших уроках). Также с помощью замыканий можно создать переменные, которые будут доступны только функции, но при этом будут существовать и после ее выполнения. Например, можно создать счетчик:

```
function createCounter() {

    var count = 0;

    return function () {
        return ++count;
    };

}

var counter = createCounter();

console.log(counter());
console.log(counter());
console.log(counter());
```

Функцию, записанную в counter можно вызывать, например, при нажатии пользователем на кнопку для создания какого-либо счетчика на сайте. Немного модифицировав код можно сделать счетчик, которому можно присвоить начальное значение:

```
function createCounter(n) {

    var count = n;

    return function () {
        return ++count;
    };

}

var counter = createCounter(5);
```

Задание №16

Функции высшего порядка – это функции, которые оперируют функциями, принимая одну или более функций и возвращая новую функцию. Рассмотрим данный функционал на примере мемоизации (сохранение ранее вычисленных результатов):

```
function memoize(f) {

    var cache = {}; // Кэш значений сохраняется в замыкании.

    return function () {

        var n = arguments[0];
        if (n in cache) {
            console.log('from cache');
            return cache[n];
        } else {
            console.log('new calc');
            return cache[n] = f.call(null, n);
        }
    };


}
```

Напишем функцию, например, расчета факториала числа n:

```
function factorialN(n) {

    n = parseInt(n);

    if (n < 0) {
        return 0;
    }

    for (var i = 1, factorial = 1; i <= n; i++) {
        factorial *= i;
    }

    return factorial;

}
```

Далее создадим новую функцию расчета факториала, но уже с мемоизацией:

```
//создаем функцию с мемоизацией
var mFactorial = memoize(factorialN);

console.log(mFactorial(5));
console.log(mFactorial(5));
```

Как видно в консоли расчет значения происходит только 1 раз, если такое значение уже было посчитано, оно вернется из кеша.

Задание №17 (домашнее задание)

В задании 16 была предоставлена функция мемоизации, которая поддерживает только функции с одним параметром. Перепишите ее так чтобы она поддерживала функции с произвольным количеством параметров и напишите функцию с мемоизацией, которая принимает произвольное количество параметров и находит их произведение.

Задание №18

Рекурсия. Простыми словами, рекурсия – определение части функции (метода) через саму себя, то есть это функция, которая вызывает саму себя, непосредственно (в своём теле) или косвенно (через другую функцию).

С помощью рекурсии можно решить множество простых и не очень простых задач:

Пример №1. Вывод чисел от 1 до N.

Дано натуральное число N (целое положительное). Вывести все числа от 1 до n.

Пример решения данной задачи:

```
function from1ToN(n) {

    if (n < 1) {
        return;
    }

    if (n > 1) {
        from1ToN(n - 1);
    }

    console.log(n);

}
```

Перенесите эту функцию файл.

По очереди допишите после функции следующий код:

```
from1ToN(2); //Проверьте консоль
```

Измените на код и перезагрузите страницу.

```
from1ToN(100); //Проверьте консоль
```

Пример №2. Числа Фибоначчи.

Вычисление чисел Фибоначчи – классический пример рекурсивного алгоритма.

Числа Фибоначчи - в которой первые два числа равны либо 1 и 1, либо 0 и 1, а каждое последующее число равно сумме двух предыдущих чисел.

Пример рекурсивного вычисления чисел Фибоначчи.

```
function fib(n) {

    if (n <= 0) {
        return 0;
    }

    if (n == 1) {
        return 1;
    }

    return fib(n - 1) + fib(n - 2);

}
```

По очереди допишите после функции следующий код:

```
fib(2); //Проверьте консоль
```

Измените на код и перезагрузите страницу.

```
fib(6); //Проверьте консоль
```

Измените на код и перезагрузите страницу.

```
fib(30); //Проверьте консоль
```

Следует помнить, что рекурсивные алгоритмы не всегда оптимальные.

Например, следующий алгоритм вычисления чисел Фибоначчи значительно более оптимальный.

```
function fibAlt(n) {

    if (n < 1) {
        return 0;
    }


    var current = 1 //тут хранится текущее значение
            , previous = 0 // тут хранится предыдущее значение
            , buffer //буффер
            ;


    for (var i = 1; i < n; i++) {
        //вычисляем следующее значение и записываем в буффер.
        buffer = current + previous;

        //записываем текущее значение как предыдущее.
        previous = current;

        //записываем вычисленное следующее значение как текущее.
        current = buffer;
    }

    return current;

}
```

Перенесите эту функцию рядом с предыдущей и проверьте скорость выполнения двух функций следующим образом:

```
var start, end;

start = Date.now(); //время начала вычислений в миллисекундах
console.log(fib(40)); //вывод 40-го числа Фибоначчи
end = Date.now(); //время окончания вычислений в миллисекундах
console.log((end - start) / 1000); //время вычисления числа Фибоначчи в секундах

start = Date.now(); //время начала вычислений в миллисекундах
console.log(fibAlt(40)); //вывод 40-го числа Фибоначчи
end = Date.now(); //время окончания вычислений в миллисекундах
console.log((end - start) / 1000); //время вычисления числа Фибоначчи в секундах
```

Как видите второй алгоритм работает значительно быстрее.

Дополнительные сведенья:

Сложность второго алгоритма куда ниже.

Вычислительная сложность — понятие в информатике и теории алгоритмов, обозначающее функцию зависимости объёма работы, которая выполняется некоторым алгоритмом, от размера входных данных.

В случае с рекурсивным алгоритмом сложность равняется $O(2^n)$ – экспоненциальная функция, а в случае со вторым $O(n)$ - линейная функция, где – n номер числа Фибоначчи.

Задание №19

Решите следующие задания с использованием рекурсивных функций:

1. Написать функцию вычисления факториала числа n.
   Факториал – произведение всех натуральных чисел от 1 до n включительно.
   Пример: 5 => 1 * 2 * 3 * 4 * 5 = 120.
   На вход подается число n > 0. Вернуть факториал числа n.
2. Написать функцию вычисления суммы цифр натурального числа n.

Пример: 123 => 1 + 2 + 3 = 6.
На вход подается число n > 0. Вернуть сумму его цифр.

3. Проверка числа на простоту.
   Простое число — натуральное (целое положительное) число, имеющее ровно два различных натуральных делителя — единицу и самого себя.
   На вход подается число больше 2. Определить является ли чисто простым с использованием рекурсии. Вернуть true если число простое, вернуть false если число не простое

Домашнее задание

1. Задание №17.
2. Реализовать все алгоритмы из задания №19, которые Вы не успели выполнить.