```
let a = 'global';
function outer() {
    let b = 'outer';
    function inner() {
        let c = 'inner';
        let b = 'inner 2'
        console.log(c);    // выдаст 'inner'
        console.log(b);    // выдаст ''
        console.log(a);    // выдаст 'global'
    }
    console.log(a);        // выдаст 'global'
    console.log(b);        // выдаст 'outer'
    inner();
}
console.log(a);
console.log(b);
outer()
```

Объект “Arguments”

Одним из главных преимуществ функций является возможность многократного использования одного и того же кода для различных значений (аргументов), а также объединение этого кода под логичным и понятным именем.

Цель и задачи объекта

При вызове функции все переданные в нее аргументы попадают в специальный объект **«arguments»**, доступный для использования в теле функции.

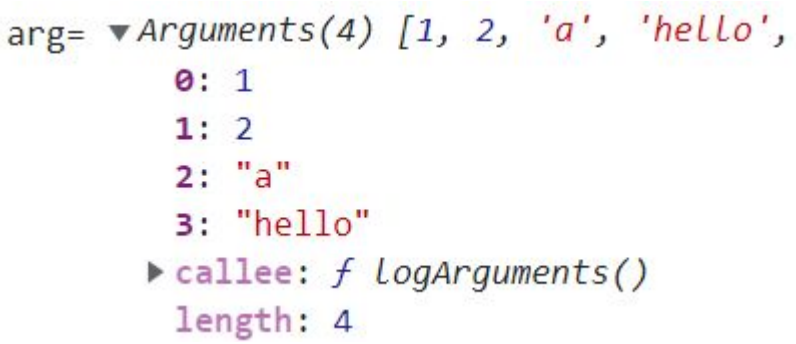
Объект **arguments** является локальной переменной, которая доступна во всех функциях, за исключением стрелочных, она позволяет ссылаться внутри функции на аргументы функции с помощью объекта **arguments**.

```
function logArguments(){
    console.log(arguments);
}
```

Вызов

```
function logArguments(){
    console.log(arguments);
}
```

```
logArguments(1, 2, 'a', 'hello')
```



length - Соответствует числу аргументов, ожидаемых функцией.

Обращение к аргументам

Этот объект содержит запись для каждого аргумента, переданного в функцию, индекс первой записи начинается с 0 и соответствует первому аргументу функции, индекс 1 соответствует второму аргументу функции и так далее.

```
function logArguments(a, b) {
  // соответствует первому аргументу функции
  console.log(arguments[0]);
    console.log(a);
    console.log(b);
}
```

```
logArguments(1, 2, 'a', 'hello')
```

```
1
1
2
```

```
logArguments() //?
```

Вывести все аргументы

```
function logArguments(x){
        console.log("x = "+x);
        for(i=0; i<arguments.length; i++){
        console.log("argument"+(i+1)+" = "+arguments[i])
         }
}
```

Вывести только числа. Тип number. Проверка через typeof

Подсчитать сумму чисел

```
function sum() {
    let n = arguments.length;
    let sum = 0;
    console.log('n=', n);
    for (let i = 0; i < n; i++) {
        console.log(typeof(arguments[i]));
        if (typeof(arguments[i]) === 'number') {
            sum = sum + arguments[i]}
    }
    return sum;
}
console.log(sum(1, 5, '3', -1));
```

//когда надо передать именованные параметры и какой-то перечень аргументов

```
function sum(a, b, ...args){
    let total = 0;
    for (i = 0; i < args.length; i++){
        total += args[i];
    }
    return total;
}
console.log( sum(1,2,2,3));//5
```

Рекурсия

Рекурсия — это процесс вызова функцией самой себя. Функция, которая в своем теле вызывает сама себя, называется *рекурсивной функцией*.

Рекурсия основана на возможности определения новых значений величин с использованием предыдущих значений величин

1*2*3*4*5*6*....*n

1! =1	**2! =1! * 2 = 1*2**	**3! =2! * 3 = 1*2*3**	**4! =3! * 4 = 1*2*3*4**

n! = (n-1)!·n

пример рекуррентной математической формулы

Код расчета факториала n

```
factorial = 1;

for(i=2; i<=n; i++) {

   factorial *= i;

}
```

Через рекурсию

правило

1) возврат определенного значения при некотором значении параметра и

2) повторный вызов себя с другим значением аргумента

```
function factorial(n){
        if(n==1) return 1;
        return n*factorial(n-1);
}
```

1! =1

n! = (n-1)!·n

```
function factorial(n){
    if(n==1) return 1;
    return n*factorial(n-1);
}
```

n=3 3*factorial(2) 3*2

n=2 2*factorial(1) 2*1

n=1 1

возвращается результат выражения «n*factorial(n-1);», что приведет к повторному вызову функции, но уже с аргументом n-1, умножению его на n и передаче в точку вызова. Каскад вызовов закончится уменьшением n до 1, после чего все возвраты со

Вывести числа от 1 до введенного пользоватлем

```
let num = +prompt("Введите число:");
console.log(displayNumber(num));

function displayNumber(n) {
    if (n == 1) return "1";
    return displayNumber(n - 1) + '  ' + n;
}
```

Вывести числа от 2 через одно до введенного пользователем

```
let num = +prompt("Введите число:");
console.log(displayNumber(num));

function displayNumber(n) {
    if (n == 2) return "2";
    return displayNumber(n - 2) + '  ' + n;
}
```

Вывести число в обратном порядке 123 ->321

```
let RevNumber = +displayRetNumber(12345);
console.log(RevNumber);

function displayRetNumber(n) {

    if (n < 10) return n
    else {
        return n % 10 + '' + displayRetNumber(Math.floor(n /
10));
    };
}
```

Сдвиг числа (12345 на 1 -> 23451)

```
//console.log(12345 % 10000 * 10 + Math.floor(12345 / 10000));
function displaysdvigNumber(n, s) {
    if (s <= 0) return n
    else {
        return displaysdvigNumber((n % 10000 * 10 + Math.floor(n / 
10000)), s - 1) + '    ' + n;
    }
}

let sdvigNumber = displaysdvigNumber(12345, 2);
console.log(sdvigNumber);
```

проверить разрядность числа

```
while (t > 1) {
        t = n / 10;
        k++
    }
```

Замыкание

Замыкание (**closure**) представляют собой конструкцию, когда функция, созданная в одной области видимости, запоминает свое лексическое окружение даже в том случае, когда она выполняет вне своей области видимости.

Замыкание технически включает три компонента:

- внешняя функция, которая определяет некоторую область видимости и в которой определены некоторые переменные - лексическое окружение
- переменные (лексическое окружение), которые определены во внешней функции
- вложенная функция, которая использует эти переменные

```
let a = 'global';
function outer() {
   let b = 'outer';
function inner() {
     let c = 'inner'
     console.log(c);    // выдаст 'inner'
     console.log(b);    // выдаст 'outer'
     console.log(a);    // выдаст 'global'
   }
   console.log(a);      // выдаст 'global'
   console.log(b);      // выдаст 'outer'
   inner();
 }
```

```
function person() {
 let name = 'Peter';

 return function displayName() {
   console.log(name);
 };
}


let peter = person();

peter(); // выведет 'Peter'
```

В этом примере мы вызываем функцию `person`, которая возвращает внутреннюю функцию `displayName` и сохраняет эту внутреннюю функцию в переменную `peter`. Когда мы вызываем функцию `peter` (которая на самом деле ссылается к функции `displayName`), имя “Peter” выводится в консоль

Но у нас же нет никакой переменной с именем `name` в `displayName`, так что эта функция как-то может получить доступ к переменной своей внешней функции `person`, даже после того, как та функция выполнится. Так что, функция `displayName` это ни что иное как замыкание.

```
function multiply(n){
    var x = n;
    return function(m){ return x * m;};
}
var fn1 = multiply(5);
var result1 = fn1(6); // 30
console.log(result1); // 30

var fn2= multiply(4);
var result2 = fn2(6); // 24
console.log(result2); // 24


var result = multiply(5)(6); // 30
console.log(result);
```

Как работают замыкания?

До этого момента мы обсуждали то, чем являются замыкания и их практические примеры.

Сейчас давайте поймём то, как замыкания на самом деле работают в JavaScript.

Чтобы реально это понять, нам надо разобраться в двумя самыми важными концепциями в JavaScript, а именно, **1) Контекст выполнения** и **2) Лексическое окружение**.

Контекст выполнения

Это абстрактная среда, в которой JavaScript код оценивается и выполняется. Когда выполняется “глобальный” код, он выполняется внутри глобального контекста выполнения, а код функции выполняется внутри контекста выполнения функции.

Лексическое окружение

Каждый раз, когда движок JavaScript создаёт контекст выполнения, чтобы выполнить функцию или глобальный код, он также создаёт новое лексическое окружение, чтобы хранить переменную определенную в этой функции во время её выполнения.