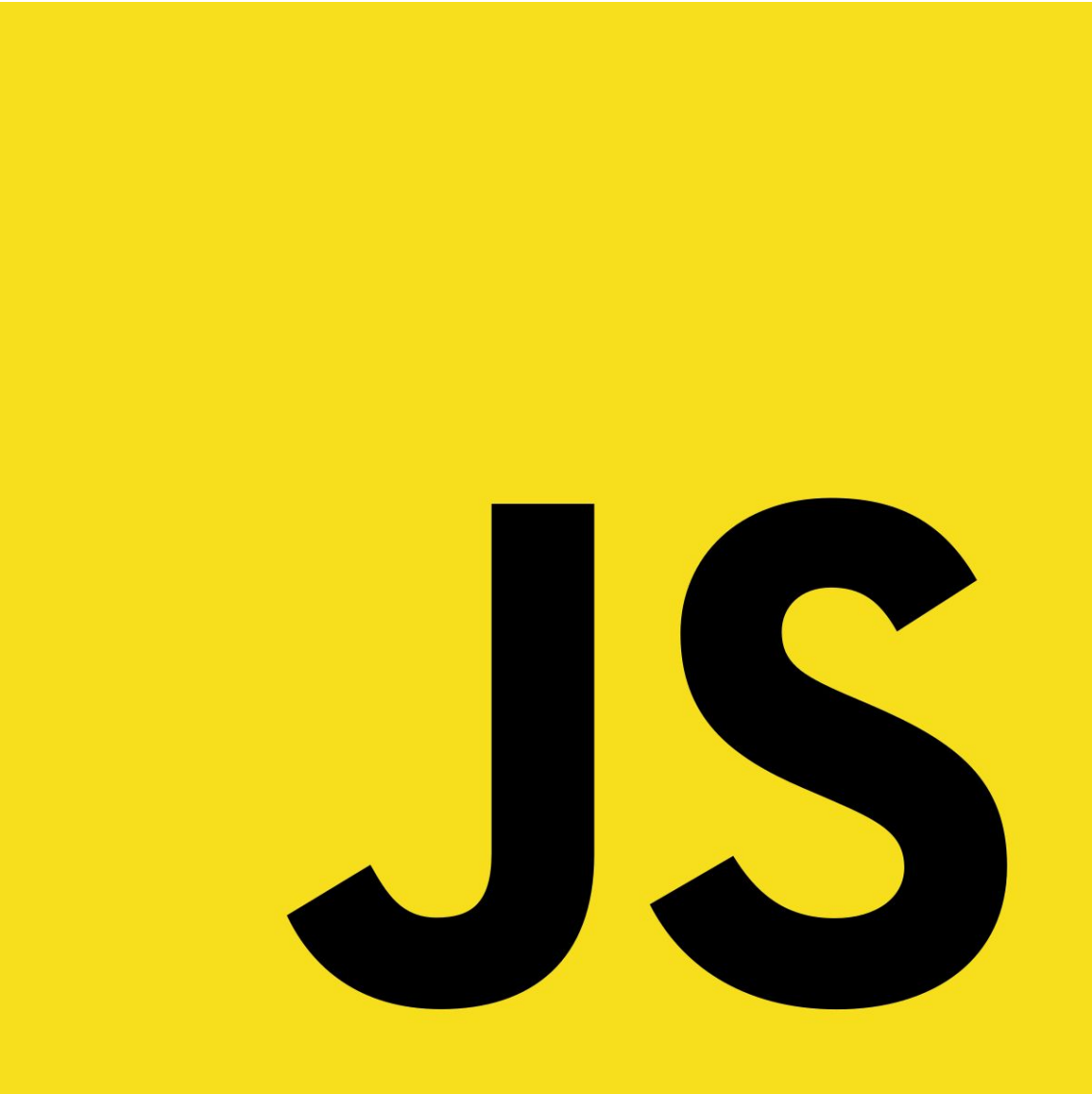


JavaScript

Функции – подробно

A large, bold, black 'JS' logo is centered on a solid yellow square background. The letters are thick and stylized, with the 'J' having a curved bottom and the 'S' being a simple, rounded shape.

Основные понятия

SINGLE TRAITING (обработка в один поток)

В один момент времени выполняется только одна команда.

SYNCHRONOUSLY EXECUTION (синхронное выполнение)

Команды выполняются последовательно, одна за другой, так как они написаны в программе.

Функции

У функций(методов) соответственно есть свой контекст

```
var a = 15;    // глобальная переменная a
function test(){
```

```
    var a = 30;
    alert(a);
```

```
}
```

Контекст функции

В этой точке программы
локальная переменная **a**
будет уничтожена

```
test();    // выводится локальная переменная a
alert(a);  // выводится глобальная переменная a
```

Execution content – контекст выполнения

Это обертка вокруг кода который выполняется в данный момент времени (например какая-либо функция). Эта обертка помогает выполнять код.

В коде много лексических областей. И текущая выполняется и управляется через **EXECUTION CONTEXT**, который в свою очередь может иметь сущности, которых нет в написанном коде.

Execution content – контекст выполнения

Есть глобальный контекст **[globalObject]**, в котором расположены все глобальные переменные и функции – для браузера это объект **window**. То есть эти переменные и функции являются свойствами и методами объекта **window**

Пример

```
var a = 10;           // создали свойство a объекта window
function test(){      // создали метод test объекта window
    alert(a);
}
```

`test();` // МОЖНО ВЫЗЫВАТЬ ТАК

`window.test();` // а МОЖНО ВЫЗЫВАТЬ И ТАК

GlobalObject

```
var name = "Mike";

console.log(name);
console.log(window.mike); // аналогично предыдущему

function getName(){
    console.log(window.name); // аналогично предыдущему
}

getName();
window.getName(); // аналогично предыдущему

// посмотрим объект window в консоли
console.log(window);
console.log(this); // аналогично предыдущему
```

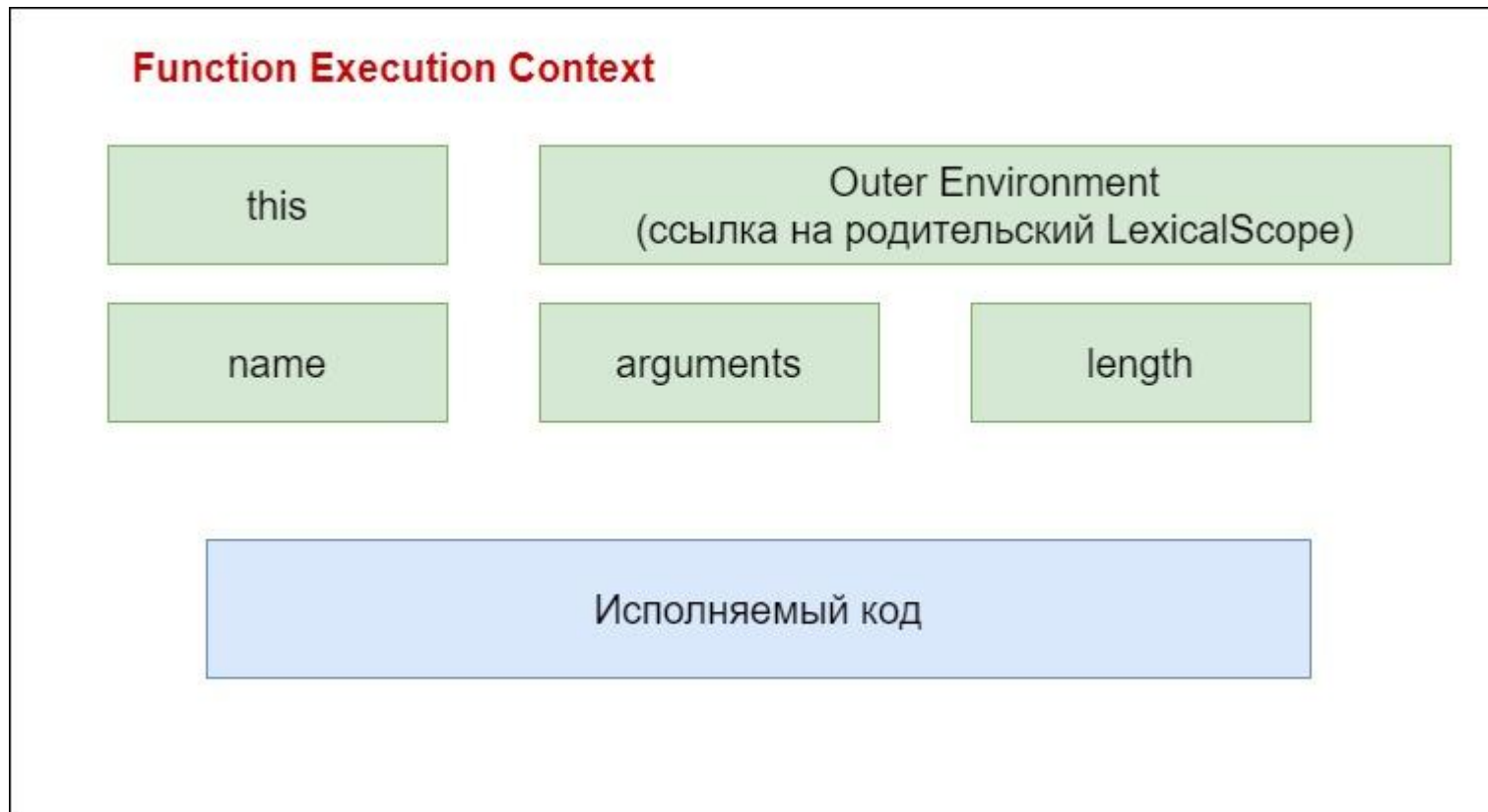
EXECUTION CONTEXT (ExC)

Как только мы запускаем js файл, интерпретатор создает для нас **GExC**, в котором устанавливает глобальный объект **GlobalObject**. В этом объекте хранятся сущности, которые для нас сформировал интерпретатор. К глобальному объекту можно обращаться через слово **window**. Глобальный объект доступен в любом месте нашей программы. Кроме того создается переменная **this** которая в **GExC** указывает на глобальный объект. То есть для **GExC** **window** и **this** совпадают

EXECUTION CONTEXT (ExC)



Function Execution Context



arguments – коллекция переданных аргументов

name – имя функции

length – количество объявленных параметров

Function invoke

```
function b(){  
    ...  
}  
  
function a(){  
    b();  
}  
  
a();
```

При вызове функций
создается
Execution Stack

b()

Для функции b создается свой Execution Context, то есть своя область памяти, свои переменные

a()

Для функции a создается свой Execution Context, то есть своя область памяти, свои переменные

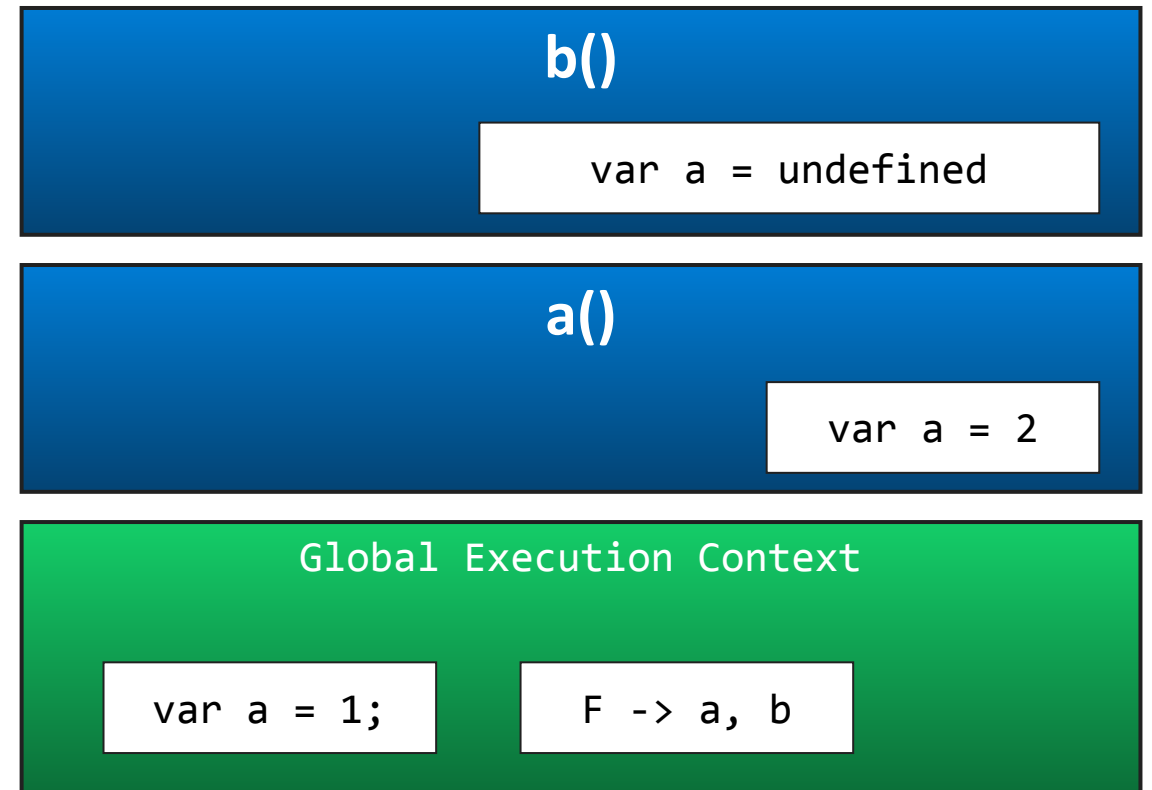
Global Execution Context

a, b

Function Execution context

```
function b(){  
  var a;  
}  
  
function a(){  
  var a = 2;  
  b();  
}  
  
var a = 1;  
a();
```

Для каждой функции создается **СВОЯ** область памяти, свои переменные.
После завершения работы функции эта область памяти очищается, и переменные соответственно уничтожаются.



Function Execution context

```
function b(){  
  console.log(a);  
}
```

```
function a(){  
  var a = 2;  
  console.log(a);  
  b();  
}
```

```
var a = 1;  
console.log(a);  
a();
```

Если обратиться к переменной, которая не определена в текущем Function Execution Context, то она будет искаться по цепочке в родительских LexicalScope

b()

a()

var a = 2

Global Execution Context

var a = 1;

F -> a, b

Инициализация локальных переменных

```
function sayHi(name){  
  
    // FExContext = { name : "Вася", say : undefined };  
  
    var say = "Hello" + name;  
  
    // FExContext = { name : "Вася", say:"Hello Вася"};  
  
    alert(say);  
}  
  
// FExContext уничтожается  
  
sayHi("Вася");
```

Еще раз про Lexical Scope

```
var b = 2;

function sum(a, b) {
  console.log(a + b);
}

sum(4)
```

В нашем случае в Function Execution Context (FExC) функции `sum` добавляется **ссылка на внешнее окружение** – OuterEnvironment (OE), а значение ее ставится по **[[Scope]]**, то есть

FExC = {a:4, OE:window }

И когда обращаемся к переменной **b**, то сначала она ищется в объекте FExC и если ее там нет, то просматривается родительский LexicalScope, ссылка на который хранится в OE.

Вариант 1

```
var b = 2; // window.b = 2

function sum(a){
  // FExContext = { a: 4, OE: window };

  alert(a + b);
  // a будет найдена в FExContext
  // b будет найдена в window
}

sum(4);
```


Вариант 2

```
var b = 2; // window.b = 2

function sum(a){
  // FExContext = { a: 4, b: undefined, OE: window };

  var b = 10;
  // FExContext = { a: 4, b: 10, OE: window };

  alert(a + b);
  // а и b будут найдены в FExContext
}

sum(4);
```

Вариант 3

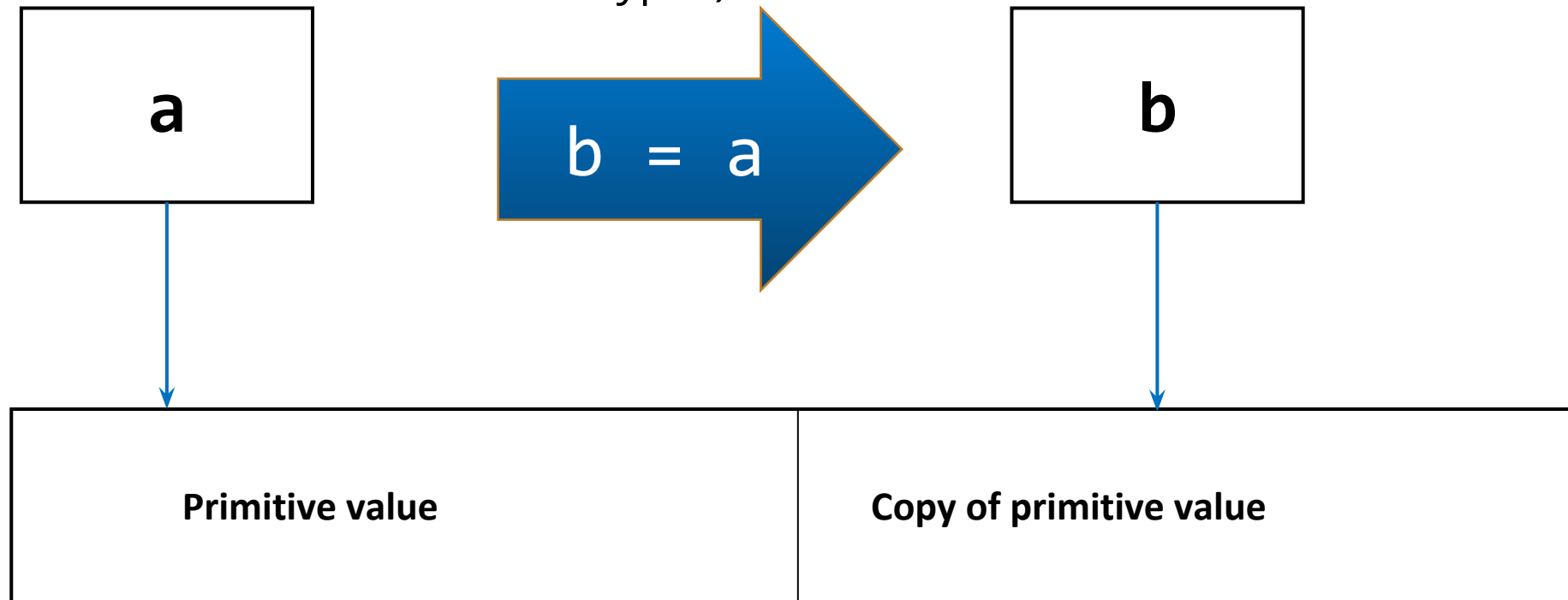
```
var b = 2; // window.b = 2
function f(a){
  // FExContext = { a: 2, OE: window };
  alert(a + b);
}

function g(){
  // FExContext = { b: undefined, OE: window };
  var b = 20;
  // FExContext = { b: 20, OE: window };
  f(2);
}

g(); // 4
```

Копирование переменных по значению (*by value*)

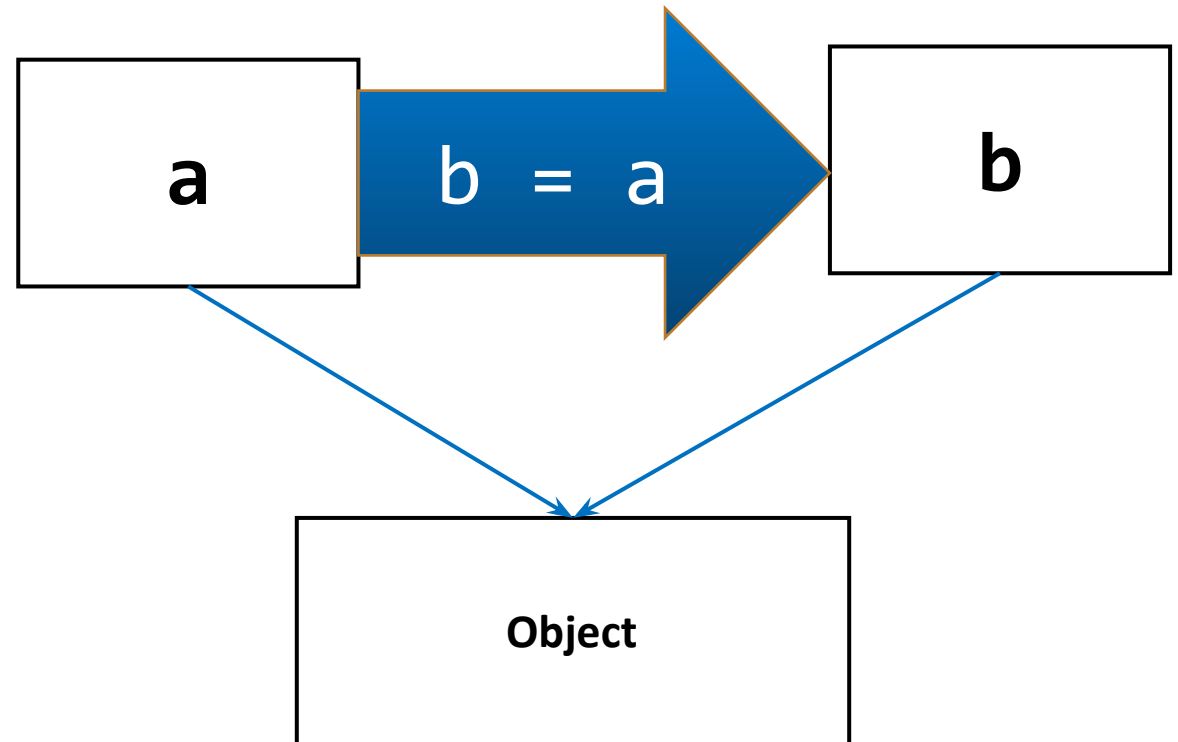
По значению копируются простые типы (primitive types)



Копирование переменных по ссылке (*by reference*)

По ссылке копируются:

- если переменная, указывающая на объект присваивается другой переменной
- объекты, передаваемые как аргумент в функцию



Детальнее

1. Аргументы в функцию передаются **по значению** – то есть в аргумент функции передается копия глобальной переменной, и поэтому если внутри функции изменять локальную переменную который был передан аргумент, глобальная переменная остается без изменений.

2. Но если передать аргумент функции параметр ссылочного типа (объект) то все изменения свойств объекта внутри функции изменяют свойства объекта непосредственно.

Почему. Параметр передается как копия адреса, по которому в памяти расположен объект.

Копирование по значению и по ссылке

Когда мы пишем

```
var x = 5;
```

to JavaScript

- на этапе **Creation** (перед выполнением кода) выделяет место в оперативной памяти системы для переменной **x**. У этой ячейки памяти соответственно есть адрес.
- на этапе **Execution** записывает в эту ячейку памяти значение 5 и возвращает в переменную **x** адрес этой ячейки памяти
- когда в программе мы будем обращаться к переменной **x** то JavaScript "лезет" в память по адресу в этой переменной и возвращает нам ее значение

Простые типы данных копируются по значению

```
var a = 30;  
var b = a;
```

```
b = 100;
```

- JavaScript полез по адресу находящимся в ячейке **a**, достал оттуда ее значение 30;
- создал новую ячейку памяти, в которую, записал 30;
- вернул в переменную **b** адрес этой ячейки.

То значения переменных **a** и **b** равны, но хранятся в разных ячейках памяти и поэтому когда мы выполнили **b = 100;** то значение в переменной **a** не изменилось

Также при передаче их как аргументов в функции

```
var a = 30;

function test(a) {
  a = 100;
  console.log("Inside function a = ",
a);
}

test(a); // Inside function a = 100
console.log(a); // 30
```

То есть внутри функции для аргумента **a** JavaScript создал для нее ячейку памяти, взял значение глобальной **a** переменной **a** и записал ее в эту ячейку

Таким образом глобальная переменная **a** не изменилась

Объекты копируются по ссылке, а массивы – это объекты

```
var arr = ["Bill", "Tom", "John"];
```

```
var newArr = arr;
```

```
newArr[1] = "Greg";
```

```
console.log(newArr);
```

```
// ["Bill", "Greg", "John"]
```

```
console.log(arr);
```

```
// ["Bill", "Greg", "John"]
```

Что происходит когда мы копируем ссылочные типы?

JavaScript в переменную `newArr` скопировал адрес переменной `arr` и таким обе переменные сейчас указывают на одну и ту же ячейку памяти. То это одна и та же ячейка, но под разными именами.

Объекты копируются по ссылке, а массивы – это объекты

Чтобы скопировать массив не по ссылке можно поступить так:

```
var arr = ["Bill", "Tom", "John"];
```

```
var newArr = arr.slice();
```

```
newArr[1] = "Greg";
```

```
console.log(newArr); // ["Bill", "Greg", "John"]
```

```
console.log(arr); // ["Bill", "Greg", "John"]
```

Пример работы с объектом по ссылке

```
var ob = {  
  name: "Bill"  
}
```

```
function test(o) {  
  o.name = 'New Name';  
}
```

```
test(ob); // в аргумент функции  
копируется адрес объекта  
console.log(ob); // {name: "New  
Name", age: 25}
```

**Что происходит когда мы копируем сылочные
типы?**

При выполнении кода функции JavaScript создал для аргумента функции **o** новую ячейку памяти и скопировал туда адрес ячейки памяти объекта **ob**. Таким образом **ob** и **o** это две разные переменные, но в них хранится одно и то же значение – а именно адрес объекта **ob**.

Function declaration

```
sayHi();
```

```
function sayHi(name) {  
    assert(true, "Hi " + name);  
}
```

Когда объявляется именованная функция – это называется **function declaration**.

Интерпретатор JavaScript перед выполнением кода "пробегаёт" по коду и создает в памяти функции, которые декларированы как function declaration.

Поэтому такую функцию можно вызывать до ее объявления.

Function declaration

```
function test(){  
    alert("I am function test");  
}  
alert(test);
```

```
var newtest = test;  
newtest();
```

```
test = 33;  
alert(test);
```

Когда объявляется именованная функция, создается переменная с таким же именем в которую записан текст функции.

То есть функция является просто значением, и это значение можно записать в переменную, можно скопировать в другую переменную.

Function expression

```
var sayHi = function(){  
    alert("Hi");  
}
```

```
(function(){  
    ... // Здесь объявляемые  
переменные будут локальными  
})();
```

Функции, которые декларированы как **function expression** не создаются интерпретатором в памяти и выполняются при достижении кода. Поэтому такую функцию можно вызывать только после ее объявления.

function expression удобно использовать для создания локальных элементов (вызов функции "на месте").

The background image is a wide-angle landscape photograph of a mountain range. Two prominent, rugged mountain peaks are visible, their upper sections covered in patches of white snow. The lower slopes are dark and appear to be covered in low-lying vegetation or bare earth. The sky is filled with heavy, grey clouds, creating a somber and atmospheric mood. In the center of the image, the word 'Практика' is written in a clean, white, sans-serif font. A thin, solid red horizontal line is positioned directly beneath the text, extending slightly beyond its left and right edges.

Практика

Hosting

Global Execution Context

**Global Object
(window)**

this

**Outer Environment
(null)**

Выделение памяти под переменные и функции в глобальной области.

- всем переменным присваивается значение undefined
- код всех функций, созданных как **function declaration** запоминаются в памяти

Выполнение кода (Execution phase)

Пример

```
console.log(a); // undefined
```

```
var a;
```

```
test(); // выведет Hi, there  
console.log(a); // выведет undefined
```

```
function test(){  
    console.log("Hi, there");  
}
```

```
a = "Hello world";
```

```
console.log(a); // выведет Hello world
```



Практика

Параметры функции arguments и this

В функцию неявно передаются параметры

arguments – это коллекция передаваемых в функцию параметров. Работа с этим параметром происходит как с обычным массивом, но этот параметр массивом не является. Это специальный объект-коллекция

this – этот параметр зависит определяется от так называемого контекста функции относительно которого она вызывается. То есть значение параметра this зависит от способа вызова функции.

Способы вызова функций

1. Вызов как функции, подобно тому как это делается в других языках программирования.
2. Вызов как метода объекта. Объект становится контекстом функции и доступен в этой функции через `this`
3. Вызов как конструктора объекта.
4. Вызов с использованием методов `call()` и `apply()`

This

При вызове функции ей неявно передается параметр **this** который ссылается на объект связанный с контекстом функции.

Параметр **this** зависит от того каким способом вызывается функция

1. Для функции – `this` -> на объект `window` при "use strict" `undefined`
2. Для метода - `this` -> на собственный объект метода
3. Для конструктора - `this` -> на вновь созданный объект
4. Для вызова используются методы `call()` и `apply()`

`имя_функции.call(объект, параметр_1, параметр_2,);`

`имя_функции.apply(объект, массив_параметров);`

Использование call, apply

Если используется режим "use strict", то в качестве объекта, передаваемого в функции `call()` или `apply()` может быть какой угодно тип, в том числе `null` или `undefined`.

В старых браузерах, передача `null` или `undefined` приводит к тому, что `this` будет указывать на глобальный объект `window`.

Пример

```
var user = {  
  firstName : "Василий",  
  secondName : "Алибабаев",  
  patronymic : "Алибабаевич"  
};
```

```
function sayName(first, second){  
  assert(true,  this[first] + " " + this[second]);  
}
```

```
sayName.call(user, "firstName", "secondName");  
sayName.call(user, "firstName", "patronymic");
```


The background image is a wide-angle landscape photograph of a mountain range. Two prominent, rugged mountain peaks are visible, their upper sections covered in patches of white snow. The lower slopes are dark and appear to be covered in low-lying vegetation or bare earth. The sky is filled with heavy, grey clouds, creating a somber and atmospheric mood. In the center of the image, the word 'Практика' is written in a clean, white, sans-serif font. A thin, solid red horizontal line is positioned directly beneath the text, extending slightly beyond its left and right edges.

Практика

Анонимные функции

Такие функции не имеют имени. Они используются в тех случаях, когда имя функции необязательно, например:

- в качестве callback функции;
- при сохранении функций в переменной;
- как методы объектов;
- в качестве обработчиков событий

Концепция callback

Например есть код

```
function useless( callback ) {  
    return callback();  
}
```

В коде функции `useless` установлена функция по имени `callback` которая потом (во время вызова функции `useless`) будет возвращаться (термин `callback`) в точку вызова функции `useless`.

```
var text="Hello from callback";  
alert( unless( function (){ return text;} ) );
```

Сортировка элементов массива

```
let values = [ 213, 16, 2058, 54,  
10, 1965, 57, 9 ];
```

```
values.sort(function(var1, var2){  
    return var1 - var2;  
});
```

```
console.log(values);
```

Для сортировки массива будем применять функцию **sort** которая в качестве аргумента принимает пользовательскую функцию.

Пользовательская функция:

1. Принимает 2 параметра – например **a** и **b**
2. Должна возвращать:
 - 0 если $a == b$
 - 1 если $a < b$
 - 1 если $a > b$

A still from a film showing three men in a forest of birch trees. The man in the center, wearing a brown shirt and a black cap, has a shocked expression with wide eyes and an open mouth. To his left, a younger man in a dark jacket and a woven headband also looks shocked. To the right, a man in a light jacket, a patterned scarf, and a fedora-style hat looks equally surprised. The scene is set outdoors with many birch trees in the background.

Лабораторные

Рекурсивный вызов функций

Рекурсия означает вызов функцией саму себя в теле этой же функции

Пример – вычисление факториала $n! = 1 * 2 * \dots * n$

Например $5! = 1 * 2 * 3 * 4 * 5 = 120$

Это выражение можно записать и так:

$$5! = 4! * 5$$

$$4! = 3! * 4$$

$$3! = 2! * 3$$

$$2! = 1! * 2$$

$$1! = 1$$

$$0! = 1$$

```
function fact(n) {  
    var res;  
  
    if(n == 1) return 1;  
    res = fact (n-1) * n;  
    return res;  
}  
  
document.write( fact(5) );
```

fact(5) return 5* = 120

fact(4) return 4* = 24

fact(3) return 3* = 6

fact(2)

return 2* = 2

fact(1)
return 1

The background image is a wide-angle landscape photograph of a mountain range. Two prominent peaks are visible, their upper sections covered in patches of white snow. The lower slopes are a mix of brownish-tan earth and sparse, dark vegetation. The sky is filled with heavy, grey clouds, creating a somber and atmospheric mood. The word 'Практика' is centered in the middle of the frame in a clean, white, sans-serif font. A thin, solid red horizontal line is positioned directly beneath the text, extending across its width.

Практика

ВСЕМ СПАСИБО ЗА ВНИМАНИЕ!
