**JAVASCRIPT**

**StyleGuide:**

<https://github.com/rwaldron/idiomatic.js/tree/master/translations/ru_RU> - Idiomatic.js

<https://google-styleguide.googlecode.com/svn/trunk/javascriptguide.xml> - от гугла

**Учебники:** <http://www.w3schools.com/jsref/> ,

<https://developer.mozilla.org/ru/docs/Web/JavaScript>

<https://karmazzin.gitbooks.io/eloquentjavascript_ru/> - выразительный js

**АСИНХРОННОЕ ВЫПОЛНЕНИЕ СКРИПТОВ**

<script async src =”small.js”><script> // подключение

Если в script указан атрибут src, содержимое внутри script будет проигнорировано

Атрибуты для script работают только если скрипт указан через src

**Обычный** скрипт выполняется каскадно, и контент грузится только после скрипта

**async** выполняетсяасинхронно, независимо от положения в каскаде и других скриптов

**defer** как async, новыполняется в порядке очередности скриптов и только после того, как будет загружен DOM

Теги <script> добавленные с помощью Javascript, ведут себя как async. Если же нужно сохранить порядок выполнения, то используется свойство script.async = false.

**ШЕСТЬ ТИПОВ ДАННЫХ**

1. **"number" -** целые и дробные числа.

Существуют специальные числовые значения Infinity (бесконечность) и NaN (ошибка вычислений):

alert( 1 / 0 ); // Infinity

alert( "нечисло" \* 2 ); // NaN, ошибка

NaN **–** не число. Не равна самой себе: NaN == NaN // false. isNaN(num) возвращает falseесли внутри число

1. **"string"** Если хоть один аргумент — строка, то другой тоже приводится к строке
2. **"boolean"** – true \ false
3. **"null" –** ничего. Например, если надо присовать пустое значение переменной. При преобразовании становится = 0
4. **"undefined" –** значение не определено. Если переменная назначена, но не использовалась. Явно не присваивают, для этого лучше подойдет null. При преобразовании **= NaN. Null == undefined // true**
5. **"object" - единственный не-примитивный тип**

**typeof [arg** ]- возвращает тип аргумента, в строковом виде

**СИНТАКСИС, КОММЕНТАРИИ**

Точку с запятой после стройки ставить **НАДО, это стандарт**

Комментраии бывают трех типов:

***Что делает эта часть кода?*** Объяснение алгоритма, неочевидных приемов

***Почему я выбрал этот вариант решения задачи?*** Следует описывать то, почему не был выбран более очевидный вариант. Описать технологии, которые он поддерживает

***Вычисление или присвоения переменных неочевидным образом***. Например, переменные, которые используются в другой части кода

**ПЕРЕМЕННЫЕ**

Всегда обьявляются через **var.** Обьявление var внутри цикла или функции называется локальной переменной

Если обьявить переменную внутри функции **без var,** она становится глобальной

**Var name, cost, time; //**обьявление трех переменных

**var COLOR\_BLUE = "#00F"; /**/обьявление константы. КОНСТАНТА пишется большими буквами через подчервивание и не меняется

Правила выбора имени переменным:

Не использовать транслит. Называть только по-английски

Использовать короткие имена только для переменных местного значения (если они присутсвтуют в небольшом фрагменте кода и их применение очевидно)

Называть переменные так**: var camelCase;** //первое слово с маленькой, последующие – с Большой

Имя переменной — существительное (item, fruit, asshole)

Имя функции — глагол или начинается с глагола. Бывает, что имена для краткости делают существительными, но глаголы понятнее (power, disableElement, fuckThisShit).

**ЛОГИЧЕСКИЕ ОПЕРАТОРЫ СРАВНЕНИЯ**

**Логические операторы сравнения всегда преобразуют значение в скобках к логическому типу (true/false)**

* Число 0, пустая строка "", null и undefined, а также NaN являются false,
* Остальные значения – true.

**If \else** используется в конструкциях, где немного **переходов (>3)**

**Switch** используется, где есть точные значения **“string”,** без вариантов типа **(a <= b).** Обязательно надо прерывать **break**, иначе оно пойдет дальше. При сравнениях в конструкции**switch** используется оператор **===**.

**? тернарный оператор** используется для внутренних команд:

var text = x > 10 ? "x больше 10" : "х меньше либо равен 10"

**| | - OR** оператор.Вычисляет по очереди до **true.**

Короткий цикл вычислений: **е**сли все значения ложные, **возвращает последнее ложное значение**

console.log( undefined || '' || false || 0 ); // 0

**&&** - **AND** оператор, возвращает true если оба значения true

Короткий цикл вычислений: если левый аргумент – false, оператор AND возвращает его и заканчивает вычисления. Иначе – вычисляет и возвращает правый аргумент

alert( 1 && 0 ); // 0 - Первый аргумент - true, поэтому возвращается второй аргумент

alert( null && 5 ); // null - Первый аргумент - false, он и возвращается

**! - NOT.** Меняет значение **с true** на **false** и наоборот.

var x = 5; !(x == y) //true

**!!** используют для преобразования значений к логическому типу:

alert( !!"строка" ); // true

alert( !!null ); // false

**==** сравнивает тип данных с переводом в числа (0 == false // true)

**===** сравнивает тип данных без перевода. (смотрит на тип данныз) Рекомендуется использовать. Когда оба значения строки – не преобразуется Object

**ЧИСЛЕННОЕ ПРЕОБРАЗОВАНИЕ**

|  |  |
| --- | --- |
| **Значение** | **Преобразуется в ...** |
| undefined | NaN |
| null | 0 |
| true / false | 1/0 |
| Строка | Пробельные символы по краям обрезаются. Далее, если остаётся пустая строка, то 0, иначе из непустой строки "считывается" число, при ошибке результат NaN. |

**ЛОГИЧЕСКОЕ ПРЕОБРАЗОВАНИЕ**

|  |  |
| --- | --- |
| **Значение** | **Преобразуется в ...** |
| undefined, null | false |
| Числа | Все true, кроме 0, NaN - false. |
| Строки | Все true, кроме пустой строки "" - false |
| Обьекты | Всегда true |

**УНАРНЫЕ И БИНАРНЫЕ ОПЕРАТОРЫ**

Унарные операторы (работают с одной переменной) могут преобразовывать строки в числа

var a = "2" , b = "3"

alert( +a + +b); // 5, число, так как оба операнда преобразованы в число

**Бинарные операторы** – те, которые работают с двумя и больше переменными

Это **операторы арифметического вычисления,** деление, вычитание, умножение

**, (запятая)** вычисляет правый и левый операнд и возвращает значение правого

var text = (15,20);

console.log(test); // 20

**ИНКРЕМЕНТ**

i++ и ++i

Если хочется тут же использовать результат, то нужна префиксная форма:

var i = 0

alert(++i) // 1

Если нужно увеличить, но нужно значение переменной ***до увеличения*** — постфиксная форма:

var i = 0

alert(i++) // 0

**BREAK / CONTINUE**

**Break –** прерывает выполнение функции \цикла и выбрасывает из него

**Continue –** возвращает в начало цикла или служит как go to. Нельзя использовать после знака **?**

Поддерживаются директивы **break / continue** для выхода из цикла/перехода на следующую итерацию.

Для выхода одновременно из нескольких уровней цикла можно задать метку:

outer:

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

for (var x = 0; x < 10 ; x++) {

…code…

break outer; // выйдет из внешнего цикла тоже

}

}

**МОДАЛЬНЫЕ ОКНА**

**сonfirm(“I am ready”)** – выводит на экран сообщение, возвращает только **true \ false**

**alert(statement)**– выводит на экран сообщение

**prompt(title , default) –**юзер вводит значение. **Title** – вопрос, **default**– значение по умолчанию, отображается.Если пользователь нажал **“cancel”**возвращает **null.**  Даже если **default** нету, лучше написать **""** - пустое значение

**+prompt() –** передает не строки, а числа. При вводе строки вернет **NaN**

**ЦИКЛ И МАССИВ**

**for (инициализация, тест(условие), инкремент) { тело цикла }**

Одно повторение цикла назвается **итерацией**

Переменную можно объявлять прямо в цикле, но видна она будет и за его пределами.

for (var i = 0; i < 10; i++) { ... }

**while (выражение) инструкция**

var i = 10;

while (i--){ //цикл, который движется к нулю, выполняется быстрее

console.log(i);

}

**do *инструкция* while (выражение)** - когда нужно, что бы цикл выполнился хотя бы один раз

Переход на метку возможен только изнутри цикла, и только на внешний блок по отношению к данному циклу. В произвольное место программы перейти нельзя.

**ФУНКЦИЯ. ОСНОВЫ**

Схематично, функцию можно определить так:

**function name (аргументы){**

**инструкции**

**return выражение**

**}**

**name (аргументы) - вызов функции**

Функцию можно создавать в любом месте кода, она **«перемещается» наверх.**

**Не надо** помещать функции внутрь условных блоков или циклов. Только внутри функций или основной программы

Переменные внутри функции имеют локальную область видимости

Функцию можно вызвать с любым количеством аргументов. Если параметр не передан при вызове — он считается равным undefined.

Передаваемые значения копируются в параметры функции и становятся локальными переменными.

Параметры функции являются её локальными переменными.

Можно объявить новые локальные переменные при помощи **var.**

Значение возвращается оператором **return ....**

Вызов **return**тут же прекращает функцию. Можно писать перед строкой, в которой идут вычисления

Стараться выделять каждую функцию отдельно, не писать много функций в одной.

Называть **функции,** которые ничего не меняют и выводят **true / false** начиная с **is.., check.. или find...**

Если **return**  вызван без значения, или функция завершилась без **return,** то её результат равен **undefined:**

functiondoNothing() {

return;

}

alert(doNothing() == = undefined ); // true

Два способа сделать значение по умолчанию:

function showMessage(from, text) {

if (text === undefined) {

text = “текст не передан”;

}

function showMessage(from, text) {

text = true || “текст не передан”;

...

}

Необходимо выбирать правильные **имена** для функций:

**showMessage(..)** // префикс show, "показать" сообщение

**getAge(..)** // get, "получает" возраст

**calcD(..)** // calc, "вычисляет" дискриминант

**createForm(..)** // create, "создает" форму

**checkPermission(..)** // check, "проверяет" разрешение, возвращает true/false

**validateData** // validate, проверяет данные

Если количество подаваемых в функцию аргументов слишком много, лишние будут проигнорированы. Слишком мало – отсутствующим будет назначено **undefined:**

**Рекурсивная функция**– функция, которая высчитывает что-то вложенной функцией. Лучше пользоваться циклами, иначе работа будет замедленна в несколько раз:

function sumTo(n) { //арифметическая прогрессия

if (n === 1) return 1;

return n + sumTo(n-1); //вычисляет сумму sumTo(4) = 4 + 3 + 2 + 1 = 10

}

alert(sumTo(7) );

**ФУНКЦИИ**

Объявление создаёт функцию и дает **ссылку на нее в переменную**. Функция не просто значение, это **обьект**

Функцию можно скопировать. При этом, так как функция — это объект, то и копируется она **«по ссылке».**

То есть, сама функция лежит где-то в памяти, **а переменная содержит «адрес»,** где она находится. При присваивании **func = sayHi** копируется этот адрес, обе переменные начинают указывать на одно и то же «место в памяти», где находится функция.

function sayHi(person) {

alert('Привет, ' + person);

}

var func = sayHi;

func('Вася'); // выведет 'Привет, Вася'

sayHi('Маша'); // и так по-прежнему работает: 'Привет, Маша'

При обращении без скобок **console.log(func)** выведется ее код, а не она сама.

Невозможно назвать функцию именем переменной, которая уже существует

При объявлении функции создаётся переменная со **значением-функцией.** На функцию можно дать другую ссылку, а первоначальное название удалить:

function func() { alert(1); }

var g = func; // скопировали

func = null; // поменяли значение

g(); // работает, теперь функция в g, а в func ничего нет

func(); // вызываем null()? ошибка

**Function Declaration**

Перед тем, как выполнять первую строку, интерпретатор сканирует код, ищет в нём **Function Declaration** (например **function func())** и обрабатывает их до загрузки основного кода. Потому функцию можно вызвать до обьявления

function greet (name){ // у функции есть имя (идентификатор)

retun "Hello " + name;

}

**Function Expression**

Функцию можно создать и присвоить переменной как самое обычное значение. Такая функция читается не в начале страницы, а по ходу выполнения кода

var greet = function(name){ // функция анонимна, но присвоена переменной

return "Hello " + name;

};

или например так:

var arr = [1, 2, function(a) { alert(a) }, 3, 4]; //объявлена внутри массива

var fun = arr[2]; //присваивается переменная, а ей – номер свойства массива

fun(1); // 1

Такое поведение позволяет объявлять функцию прямо внутри условных операторов, например **if else**

**Named Function Expression (NFE):**

**Именованная функция –** тот же самый **Function Expression**. Имя функционального выражения (**sayHi**) доступно только изнутри самой функции. Так же его невозможно перезаписать

var f = function sayHi(name) {

alert(sayHi); // изнутри функции - видно (выведет код функции)

};

alert(sayHi); // снаружи - не видно (ошибка: undefined variable 'sayHi')

**NFE** используется в первую очередь в тех ситуациях, когда функцию нужно передавать в другое место кода или перемещать из одной переменной в другую.

Внутреннее имя позволяет функции надёжно обращаться к самой себе, где бы она ни находилась.

/\*Для того, чтобы функция всегда надёжно работала, объявим её как NFE:\*/

var f = function factorial(n) {

return n ? n\*factorial(n-1) : 1;

};

var g = f; /\* скопировали ссылку на функцию-факториал в g\*/

f = null;

alert( g(5) ); /\* 120, работает!\*/

**Обьявление функции "на месте". Self-invoking anonymous function**

Это та же **Function Expression**, используется, что бы избежать **конфликтов**, если скрипт задаёт переменные a, b, а странице они уже используются. Скрипту задается отдельная область видимости, в которой он будет работать.

(function() { //без имени! Скобки создают локальную область видимости

var a = 1 , b = 2; // переменные для нашего скрипта

// код скрипта

})(); // скобки сразу же ее вызывают

Она никуда **не сохраняется,** и поэтому не может быть вызвана второй раз.

* Если браузер видит **function** в основном потоке кода - он считает, что это **Function Declaration.**
* Если же **function** идёт в составе более сложного выражения, то он считает, что это **Function Expression.**

Скобки вокруг функции и делают выражение сложным. Можно вызвать так:

+function() {

alert('Вызов на месте');

}();

Из соображений стиля и читаемости **скобки вокруг function** рекомендуется ставить:

var greeting = (function(name){ //нет никакого смысла давать имя самовызывающейся ф-ции

return "Hello " + name;

}("Sorax")); //функция вызывается на месте и сразу передаются внутрь аргументы

console.log(greeting) //вызов функции c выводом в консоль

**Function Expression** лучше использовать только когда это действительно нужно. Например, для объявления функции в зависимости от условий

**ТЕСТИРОВАНИЕ**

<http://plnkr.co/edit/5jBwLikmnjWelIFkJ8Qc?p=preview> - странича с примерами тестов

**МЕТОД И СВОЙСТВО**

У строки есть свойство **length,** содержащее длину:

alert( "Привет, мир!".length ); // 12

Еще у строк есть метод **toUpperCase(),** который возвращает строку в верхнем регистре (обязятельные круглые скобки в конце!):

var hello = "Привет, мир!";

alert( hello.toUpperCase() ); // "ПРИВЕТ, МИР!"

Если через точку вызывается функция **(toUpperCase())** — это называют «вызов метода», если просто читаем значение **(length)** — «получение свойства».

**ЧИСЛА**

При делении на ноль будет **infinity: 1 / 0**

**Infinity** больше любого числа

**isFinite(n) определяет число ли находится в скобках.** Возвращает **true** только тогда, когда **n** — обычное число, а не одно из этих значений:

alert( isFinite(1) ); // true

alert( isFinite(Infinity) ); // false

alert( isFinite(NaN) ); // false

**isNaN** пробует переобразоватьаргумент в число, а затем проверяет, число это или нет. Внимание! Пустая строка "" преобразуется в **0**, будет **false**

alert(isNaN("")) // false

**parseInt** и ее аналог **parseFloat** ищут целое число или дробное, пока не напорются на ошибку

alert( parseInt(“12px”) ) // 12, ошибка на символе 'p'

alert( parseFloat(“12.3.4”) ) // 12.3, ошибка на второй точке

Конечно, существуют ситуации, когда **parseInt / parseFloat** возвращают **NaN**. Это происходит при ошибке на первом же символе:

alert( parseInt('a123') ); // NaN

Если вам нужна действительно точная проверка на число, которая не считает числом строку из пробелов, логические и специальные значения — используйте следующую функцию **isNumeric:**

function isNumeric(n) {

return !isNaN(parseFloat(n)) && isFinite(n);

}

**toString(255,16) // FF**– метод, который переводит в 16-ричную систему, возвращает строку

**toPrecision()** – возвращает номер с указанной длинной:

var x = 9.656

x.toPrecision(6);  // returns 9.65600

**СТРОКИ**

Перенос строки - **\n**

Экранирование. Var str = " I'm a JavaScript \"programmer\" "; слеш экранирует символ

Строчные методы и свойства

var str = “Pilsner”

**str.length** // считает количество символов. Свойство строки

**str.charAt(0)** // возвращает первый символ. Если символа нет, показывает пустую строку

**str[0]** –возврат символа, только работает в IE8+. Если сивола нет, выдет undefined. Не рекомендуется

**str.charAt(0).toLowerCase() );** // “P” - изменение регистра строки или символа. Антоним – **toUpperCase**

**str.indexOf(“ls”)** // 2, сочетание букв найдено начиная со второй позиции. Вторым аргументом функция может принять начальную позицию, откуда начнет искать str.indexOf(“ls”, 2). Если не найдено, возвращает -1

**str.lastIndexOf()** – тоже самое, только ищет с конца. Возвращает только true!

**~** обычно используется для проверки совпадений:

if( ~str.indexOf("ner") ){ //если в строке есть сочетание «ner»

alert(“совпадение есть!”);

} //okay, what u`v gonna do?

**substring(start, end)** - возвращает подстроку с позиции start до, но не включая end. Если end отсутствует, идет до конца строки. При отрицательном значении, аргумент приравнивается «0».

**substr(start , length) –** работает так же, только второй аргумент содержит количество символов, которые надо возвратить. При минусе возвращает всю строку от индекса

**slice(start [, end]) -** Возвращает часть строки от позиции **start** до, но не включая, позиции **end**. Смысл параметров — такой же как в **substring.** Разница – отрицательные значения отсчитываются от конца строки:

"testme".slice(1, -1) ); // "estm", от первой позиции с начала до первой с конца.

**replace(value1, value2) –** заменяет **value1** на **value2.** Поскольку строку изменять нельзя, необходимо старую приравнять новой измененной.

**concat(str1, str2)** соединяет две или более строк и возвращает копию соединенных строк

**localCompare(str)** – сравнивает значения строк, возвращает 0 если идентичны, -1 если нет

**trim()** - удаляет пробелы с обоих концов строки

**Замена строки**

Строки нельзя изменять. Можно прочитать символ, но нельзя заменить его. Как только строка создана — она такая навсегда. Чтобы это обойти, создаётся новая строка и присваивается в переменную вместо старой:

var str = "строка";

str = str.charAt(3) + str.charAt(4) + str.charAt(5);

alert(str); // ока

**Округление чисел**

**Math.floor** округляет вниз

**Math.ceil** округляет вверх

**Math.round** округляет до ближайшего целого

Обычный трюк — это умножить и поделить на 10 с нужным количеством нулей. Например, округлим 3.456 до 2го знака после запятой:

var n = 3.456;

alert( Math.round( n \* 100 ) / 100 ); // 3.456 -> 345.6 -> 346 -> 3.46

**.toFixed(precision)** – может округлять до значения, возвращает строку, указанного в скобках

var n = 12.34;

alert( n.toFixed(5) ); // "12.34000", добавлены нули до 5 знаков после запятой

alert( +n.toFixed(5) ); // 12.34 – если хотим получить число, а не строку

Неточности вычисления есть в языке этом:

alert(0.1 + 0.2); // 0.30000000000000004

Методы решения :

alert( (0.1\*10 + 0.2\*10) / 10 ); // 0.3

var result = 0.1 + 0.2;

alert( +result.toFixed(10) ); // 0.3

**РАЗНЫЕ МАТЕМАТИЧЕСКИЕ КОМАНДЫ**

**Math.pow(x, exp)** - Возводит число в степень, возвращает x exp, например Math.pow(2,3) = 8. Работает в том числе с дробными и отрицательными степенями, например: Math.pow(4, -1/2) = 0.5.

**Math.max(a, b, c...)** Возвращает наибольший из списка аргументов

**Math.min(a, b, c...)** Возвращает наименьший из списка аргументов

**Math.round() -** округление чисел

**Math.random()** Возвращает псевдо-случайное число в интервале [0,1) - то есть между 0(включительно) и 1(не включая). Генератор случайных чисел инициализуется текущим временем.

var getRandom = function(min, max){

return Math.random() \* (max - min) + min;

};

getRandom(0,100) // произвольное число в любом задаваемом диапазоне

**ПРЕОБРАЗОВАНИЕ ТИПОВ**

Любые арифметические операции преобразуют типы. Сравнение преобразовывает к числам (кроме = = = ).

**Необходимо всегда использовать оператор сравнения ===**

При преобразовании, операция идет **слева на право**. Т.е. сначала либо вычисляется результат, а потом приравнивается к строке, которая в конце, либо если строка в начале – все что дальше, переводится в строку

**Number("444")** - преобразует в числовой тип

**String(44433)** - преобразует в строчный тип

**Boolean(1)** - преобразует в булевый тип

**console.log(!!5)** - быстрое преобразование в логический тип

**+"343"** - быстро преобразует строку в число

**343 + ""** - быстрое преобразование в строку. Так же можно использовать **toString**

**parseInt("45 px") / parseInt("45 px") -** преобразует строку в число так же

**+""** - становится 0

**+true -** становится 1

**+false -** становится 0

**Преобразование в обьект**

var n = 5;

n.value // undefined, преобразование прошло потому, что попытались использовать метод

Для явного преобразования используется двойное логическое отрицание **!!value** или вызов **Boolean(value**):

alert( !!" " ); // любые непустые строки, даже из пробелов - true!

Все значения, которые интуитивно **«пусты»,** становятся **false**. Ихвсего 5:

**!!0**

**!!NaN**

**!!""**

**!!null**

**!!undefined**

Также, в отличие от ряда других языков программирования, **пустой** **обьект {}** или **массив []** являются **true**

**Логическое преобразование (while(obj), if(obj)**

if (true) console.log("Its true") //произошло преобразование в булевый тип

var n = 0;

while (n < 10){ //идет проверка, преобразование в булевое значение

console.log("Vasya lol")

n++

}

**ОБЬЕКТ**

**Обьект - набор свойств, которые представленны парой имя: значение**

var obj = { name: "Alex", age: 28 }

Доступ к элементам осуществляется:

* **obj.name**
* **obj["age"]**

Добавить новое свойство можно так же двумя способами:

* **obj.prop = 5**
* **obj["key"] = 5 - можем подставлять любое выражение, поскольку оно задается через кавычки ""**

На имя свойства при доступе «через точку» наложены синтаксические ограничения — примерно те же, что и на имя переменной. Оно не может начинаться с числа, содержать пробелы и т.п. При обращении через квадратные скобки можно использовать любую строку:

person['день рождения!'] = '18.04.1982'; // запись через точку была бы невозможна

В обоих случаях, имя свойства обязано быть строкой. Если использовано значение другого типа — JavaScript приведет его к строке автоматически.

**Второй способ создать обьект - через конструктор:**

var obj = new Object();

obj.property = "value"

**Третий способ создать обьект:**

var obj = Object.create(null) //если не хотим, что бы обьект наследовал свойства

var obj = Object.create({x: 10, y: 20});

Свойства, которые передаются в обьект выше, являются свойства прототипа

**Удалить свойства обьекта можно через унарный оператор delete:**

delete obj.name /\*можно удалить только родные свойства обьекта, свойства прототипа можно удалить только у прототипа\*/

**Проверка наличия свойства у обьекта**

obj.hasOwnProperty("age") //выдает true or false, проверяет собственные свойства обьекта

"x" in obj //проверяет наличие свойств как собственных, так и наследуемых

\*\*\*хинт: если у обьекта свойство задано вручную **undefined**, бинарный оператор оператор **in** покажет его:

obj.z = undefined;

console.log(z in obj) //true

**В обьект может быть вложен еще один обьект:**

var user = {

name: "Таня",

age: 25,

size: {

top: 90,

middle: 60,

bottom: 90

}

}

**ПЕРЕБОР СВОЙСТВ И ЗНАЧЕНИЙ ОБЬЕКТА**

Перебор работает точно так же, как обычный цикл. Только инкремент заложен автоматом:

for (var prop in obj) { // var prop обычно задают прямо тут

/\* ... делать что-то с obj[key] ... \*/

}

Сортируются по порядку введения только те свойства, которые являются строчными. Для того, что бы сделать их численными, в строке добавляется + перед числом:

var codes = {

"+255": "Танзания", // передаём 255 в виде строки, чтобы браузер сохранил порядок

"+7": "Россия",

"+38": "Украина",

"+1": "США"

};

for (var key in codes ) {

var value = codes[key];

var id = +key; // ..если нам нужно именно число, преобразуем: "+255" -> 255

alert( id + ": " + value ); // 255, 7, 38, 1 во всех браузерах

}

**ССЫЛКА НА ОБЬЕКТ**

Обьект является независмой переменной в памяти. Обьект невозможно удалить, можно только обрубить ссылки на него. Именно потому, если две переменные ссылаются на один обьект, при изменении свойств обьекта в одной переменной, свойства изменятся и во второй:

var user = { name: 'Вася' };

var admin = user;

admin.name = 'Петя'; // поменяли данные через admin

alert(user.name); // 'Петя', изменения видны в user

**КЛОНИРОВНИЕ ОБЬЕКТА**

Для клонирования обьектов, что бы они были независимы друг от друга, необходимо переписать всех их свойства:

var user = {

name: "Вася",

age: 30

};

var clone = {}; // новый пустой объект

// скопируем в него все свойства user

for (var key in user) {

clone[key] = user[key];

} // теперь clone - полностью независимая копия

**МАССИВЫ**

Массивы существуют для работы с упорядоченным набором элементов.

Объявление:

var arr = [ элемент1, элемент2... ]; // предпочтительно

var arr = new Array( элемент1, элемент2...); // new Array

При этом **new Array(число)** создаёт массив заданной длины, без элементов. Чтобы избежать ошибок, предпочтителен первый синтаксис.

Свойство**length** — длина массива. Если точнее, то последний индекс массива плюс 1. Если её уменьшить вручную, то массив укоротится. Если **length** больше реального количества элементов, то отсутствующие элементы равны **undefined**.

arr.length = 2; // укоротить массив до 2 элементов

Массив можно использовать как очередь или стек. В массиве можно хранить любые обьекты:

var arr = [ 1, 'Имя', { name: 'Петя' }, true ]; // микс значений

alert( arr[2].name ); // Петя (получить обьект из массива и тут же его свойство)

После последнего свойства массива **не надо ставить запятую**! Могут быть **ошибки**

Операции с концом массива:

arr.push(элемент1, элемент2...) добавляет элементы в конец. Аналог fruits[fruits.length] = ....

**var elem = arr.pop()** удаляет и возвращает последний элемент.

Операции с началом массива:

arr.unshift(элемент1, элемент2...) добавляет элементы в начало.

**var elem = arr.shift()** удаляет и возвращает первый элемент.

Эти операции перенумеровывают все элементы, поэтому работают медленно.

Не стоит использовать **for…in** для массива, выведет все свойства обьекта, а не только цифровые. Циклfor(var i=0; i<arr.length...)**надёжнее и быстрее.**

Многомерный массив, матрица:

var matrix = [

[1, 2, 3],

[4, 5, 6],

[7, 8, 9]

];

alert(matrix[1][1]); // центральный элемент

Самый простой способ **очистить** массив — это **arr.length=0**

**МАССИВЫ: МЕТОДЫ**

Если в **массив** прописывать **именованный индекс** (название вместо цифры), джаваскрипт преобразует массив в обьект и **методы массива работать не будут**!

**Object.keys(obj)**  возвращает ключи обьекта в виде массива. Он поддерживается везде, кроме IE<9:

var user = { name: "Петя", age: 30}

var n = Object.keys(user);

alert(n); // name, age

//В более старых браузерах аналогом будет цикл:

var n = [];

for(var key in user) n.push(key);

При отборе **уникальных(не повторяющихся)** значений, можно приводить значения к массиву, а потом записывать их как **свойства** обьекта. Название свойства может быть только одно, потому обьект перезаписывает повторы:

function unique(arr) {

var obj = {};

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

var str = arr[i];

obj[str] = true; // запомнить строку в виде свойства обьекта

}

return Object.keys(obj); // выводятся свойства обьекта как массив}

Метод **split(s)** позволяет превратить строку в массив, разбив ее по разделителю s. В примере ниже таким разделителем является строка из запятой и пробела:

var names = 'Маша, Петя, Марина, Василий';

var arr = names.split(', ');

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

alert('Вам сообщение ' + arr[i]); //используется в почтовых рассылках

}

У метода **split** есть необязательный второй аргумент — ограничение на количество элементов в массиве. Если их больше, чем указано — остаток массива будет отброшен

Пустая строка str.split('') разобьёт строку на буквы

**arr.join(str)**делает в точности противоположное **split**. Он берет **массив и склеивает его в строку**, используя **str** как разделитель:

var arr = ['Маша', 'Петя', 'Марина', 'Василий'];

var str = arr.join(';');

alert(str); // Маша;Петя;Марина;Василий

**splice** это универсальный метод для работы с массивами. Умеет все: удалять элементы, вставлять элементы, заменять элементы - по очереди и одновременно.

arr.splice(index, deleteCount, elem1, ..., elemN)

Удалить **deleteCount** элементов, начиная с номера **index**, а затем вставить **elem1**, ..., **elemN** на их место.Сдвигает элементы на место удаленных самостоятельно.Возвращает массив из удаленных элементов; также может вставлять элементы без удаления, для этого достаточно установить deleteCount в 0**.**

При отрицательного номера позиции **arr.splice(-1, 0, you, me)** отсчитывает позицию с конца.

**slice(begin, end)** копирует участок массива от **begin** до **end**, не включая **end**. **Исходный массив при этом не меняется**. Работает так же, как и строчный. Если не указать аргументов – скопируется весь массив

**splice(**[индекс, с которого нужно начать удаление], [количество элементов], [что нужно вставить на их место] **)** - используется для одновременного удаления и вставки элементов в массив. Изменяет оригинальный массив. Возвращает удаленные элементы

array,splice(1,1,"Sorax", "WebDev")

**sort()** сортирует массив на месте, преобразовывая элементы к строке. Внутренняя реализация метода **arr.sort(fn)** умеет сортировать любые массивы, если указать функцию **fn** от двух элементов, которая умеет сравнивать их. **Если эту функцию не указать, то элементы сортируются как строки:**

function compareNumeric(a, b) {

return a - b

}

var arr = [ 1, 2, 15 ];

arr.sort(compareNumeric); //указываем функцию без скобок () !!

alert(arr); // 1, 2, 15

**arr.reverse()** меняет порядок элементов в массиве на обратный. **Изменяет оригинальный массив!**

var arr = [1,2,3];

arr.reverse();

alert(arr); // 3,2,1

**arr.concat(value1, value2, … valueN)** **возвращает новый масссив**, в который копируются элементы из arr, а также value1, value2, ... valueN. Если value — массив, то concat добавляет его элементы:

var arr = [1,2];

var newArr = arr.concat( [3,4], 5); // 3,4,5)

alert(newArr); // 1,2,3,4,5

**arr.indexOf(searchElement, fromIndex)** возвращает номер элемента searchElement в массиве arr или -1, если его нет. **indexOf/lastIndexOf** методы не поддерживаются в IE<9. Для их поддержки подключить библиотеку **ES5-shim.** Поиск начинается с номера **fromIndex**, если он указан. Если нет — с начала массива. Для поиска используется строгое сравнение ===**.** **Метод полностью аналогичен методу для строк**

**toString = valueOf** – приводит к строке

Object.key(array)

**push()** - толкает новый элемент в конец массива, изменяет исходный массив

**unshift()** - добавляет элементы в начало массива

**pop()** - удаляет элемент с конца массива и возвращает его

**shift()** - удаляет элемент с начала массива и возвращает его

**ПЕРЕБИРАЮЩИЕ МЕТОДЫ МАССИВА**

IE8- просто подключите библиотеку **ES5-shim**

**arr.forEach()** используется для перебора массива. Метод **forEach** ничего не возвращает, его используют только для перебора. Он для каждого элемента массива вызывает функцию **callback**. Этой функции он передаёт три параметра **callback(element, index, array):**

**element -** очередной элемент массива.

**index -** его номер.

**array -**  массив, который перебирается

var arr = ["Яблоко", "Апельсин", "Груша"];

arr.forEach(function(element, index, array) {

array[index] = element.toUpperCase();

});

console.log(arr); //мы сами задани в callback изменение исходного массива

**arr.map()** используется для трансформации массива. Возвращает новый массив, каждый элемент которого формируется из значений, которые возвращаются из функции, которую мы передаем в качестве первого аргумента. Таким образом, нам нен нужны даже элементы с индексом и самим массивом:

var arr = ["большой", "огромный", "гигантский"]

arr.map(function(e){return e.toUpperCase()}) // все приведет к верхнему регистру

**arr.filter()** позволяет отсеять элементы из массива по какому-либо критерию. Он принимает функцию, которая должна вернуть истинное или ложное значение. Если функция возвращает истину для элемента, **элемент добавляется в возвращаемый массив**

**var arr = [1, 2, 3, 4, 5];**

**var num = arr.filter(function(number) {**

**return number > 3;**

**});**

**console.log(num); // 4,5**

**arr.every()** возвращает булевое значение. Вернет **true**, если для всех элементов будет выполнено какое-то условие, и false если найдется хотя бы один элемент, для которого условие не будет выполнено

**arr.some()** возвращает **true**, если вызов **callback** вернёт **true** хотя бы для одного элемента

var arr = [1, 2, 3, 4, 5];

function num(number) {

return number > 3;

}

console.log( arr.every(num) ); // false, не все больше 3

console.log( arr.some(num) ); // true, есть хоть одно больше 3.

**arr.reduce(callback[, initialValue])** возвращает какое-то одно значение, которое получается в результате выполнения функции для каждого элемента массива с возможностью сохранения промежуточного результата

Метод **reduce** используется для вычисления на основе массива какого-либо единого значения, иначе говорят «для свёртки массива».Он применяет функцию **callback** по очереди к каждому элементу массива слева направо, сохраняя при этом промежуточный результат.

Аргументы функции callback(previousValue, currentElement, index, array):

**previousValue** — последний результат вызова функции, он же «промежуточный результат».

**currentElement**— текущий элемент массива, элементы перебираются по очереди слева-направо.

**index** — номер текущего элемента.

**array** — обрабатываемый массив.

Кроме **callback**, методу можно передать «начальное значение» — аргумент **initialValue**. Если он есть, то на первом вызове значение **previousValue** будет равно **initialValue**, а если у reduce нет второго аргумента, то оно равно первому элементу массива, а перебор начинается со второго

var arr = [1, 2, 3, 4, 5] //например мы хотим получить сумму всех элементов

// Для каждого элемента массива запустить функцию,

// промежуточный результат передавать первым аргументом далее

var result = arr.reduce(function(a, b) {

return a + b; //запоминает промежуточное значение, начинает с 0: 0+1=1, 1+2=3, 3+3=6

}, 0);

console.log( result ); // 15

При первом запуске **a** — исходное значение, с которого начинаются вычисления, равно .

Сначала анонимная функция вызывается с этим начальным значением и первым элементом массива, результат запоминается и передаётся в следующий вызов, уже со вторым аргументом массива, затем новое значение участвует в вычислениях с третьим аргументом и так далее.

**ДАТА И ВРЕМЯ**

new Date – создает обьект Date с текущей датой и временем

**new Date(milliseconds) –** можно передавать миллисекунды

new Date(datestring) - если единственный аргумент - строка, используется вызов **Date.parse()** для ее разбора

**new Date(year, month, date, hours, minutes, seconds, ms)** – дату можно создать, используя компоненты в местной временной зоне. Для этого формата обязательны только первые два аргумента. Отсутствующие параметры, начиная с **hours** считаются равными нулю, а **date** — единице.

Год year должен быть из 4 цифр, а отсчет месяцев month начинается с нуля.

var now = new Date(2012,0,20); //20 января 2012

**getFullYear()** Получить год(из 4 цифр)

**getMonth()** Получить месяц, от 0 до 11.

**getDate()** Получить число месяца, от 1 до 31.

**getDay()** Получить номер дня в неделе. Неделя в JavaScript начинается с воскресенья, так что результат будет числом **от 0(воскресенье) до 6(суббота)**.

getHours(), getMinutes(), getSeconds(), getMilliseconds() – получить

getUTCDate – показывает время по 0 Гринвича

**getTime()** Возвращает число миллисекунд, прошедших с 01.01.1970 00:00:00 UTC.

**getTimezoneOffset()** Возвращает разницу между местным и UTC-временем, в **минутах**

**Методы установки даты и времени:**

При установке времени, работает автоисправление, например при установке setDay(2014.0.-1) метод автоматически откатывает на один день на прошлый месяц. Так же и прибавляет нужное кол-во дней

setFullYear(year [, month, date])

setMonth(month [, date])

setDate(date)

setHours(hour [, min, sec, ms])

setTime(milliseconds) (устанавливает всю дату по миллисекундам с 01.01.1970 UTC)

//Какое число месяца было 100 дней назад? Какой день недели?

var d = new Date; //берет текущую дату

d.setDate( d.getDate() - 100 ); **//отнимает количество дней**

alert(d.getDate()) //выводит с автоисправлением

var dayNames = ['вс','пн','вт','ср','чт','пт','сб']; **//первый день [0] - вс**

alert( dayNames[d.getDay()] ); //выводит нужный день недели

**toString()**,**toDateString()**,**toTimeString() –** возвращают строку целиком или дату. Созданы только для строчного вывода даты

**Date.parse(str)**

Все современные браузеры, включая IE9+, понимают даты в упрощённом формате ISO 8601 Extended.

Этот формат выглядит так: YYYY-MM-DDTHH:mm:ss.sssZ

Метод Date.parse(str) разбирает строку str в таком формате и возвращает соответствующее ей количество миллисекунд. Если это невозможно, Date.parse возвращает NaN.

var msUTC = Date.parse('2012-01-26T13:51:50.417Z'); // зона UTC

console.log( msUTC ); // 1327571510417 (число миллисекунд)

**Date.now()** возвращает дату сразу в виде миллисекунд.

Технически, он аналогичен вызову +new Date(), но в отличие от него не создаёт промежуточный объект даты, а поэтому – во много раз быстрее.

**Форматирование и вывод дат**

Вызов **date.toLocaleString(локаль, опции)** позволяет указать, какие параметры даты нужно вывести, и ряд настроек вывода, после чего интерпретатор сам сформирует строку.

var date = new Date(2014, 11, 31, 12, 30, 0);

var options = {

era: 'long',

year: 'numeric',

month: 'long',

day: 'numeric',

weekday: 'long',

timezone: 'UTC',

hour: 'numeric',

minute: 'numeric',

second: 'numeric'

};

alert( date.toLocaleString("ru", options) ); // среда, 31 декабря 2014 г. н.э. 12:30:00

**БЕНЧМАРКИНГ**

Самый простой бенчмаркинг:

var start = new Date; // засекли время

doSomething();

var end = new Date; // конец измерения

console.log("doSomething заняла " + (end - start) + " ms" );

**performance.now()**

В современных браузерах (кроме IE9-) вызов **performance.now()** возвращает количество миллисекунд, прошедшее с начала загрузки страницы. Причём именно с самого начала, до того, как загрузился HTML-файл, если точнее – с момента выгрузки предыдущей страницы из памяти.

Его можно посмотреть в любом месте страницы, даже в <head>, чтобы узнать, сколько времени потребовалось браузеру, чтобы до него добраться, включая загрузку HTML.

var t0 = performance.now();

doSomething();

var t1 = performance.now();

console.log("Call to doSomething took " + (t1 - t0) + " milliseconds.")

**console.time(метка) и console.timeEnd(метка)**

console.time('doSomething') // включить хронометр

console.timeEnd('doSomething') // выключить хронометр и вывести результат

**CALLBACK (Оповещение)**

**Callback** - это передача функции в функцию. Обычно используется для оповещения, например о том, что загрузка завершена.

Пример 1 (Дима):

function download(callbackFunc) {

downloading... // по завершеню закачки вызывает callback

callbackFunc();

}

function callbackFunc(){

console.log("Downloading complete")

}

download(callbackFunc) // вызов функции, с передачей функции, которая будет вызвана позже

Пример 2 (Sorax):

var func = function(callback){

var name = "Alex";

callback(name);

};

func(function(n){

console.log("Hello " + n);

});

**Функция как возвращаемое значение**

Функция может возвращать функцию, которую тоже необходимо вызвать:

var func = function () {

return function () { //еще одна анонимная функция

console.log("Hi");

}

};

func()();

/\*первый вызов: в точку вызова возвращается анонимная функция, содержимое не читается\*/

/\*Второй вызов: вызывается инструкция внутри анонимной функции\*/

Переписать пример,что бы было понятнее, можно так:

var func = function () {

function hi () { //дали название функции

console.log("Hi");

}

hi(); // тут функция вызывается сразу внутри

};

func(); //потому двойной вызов не нужен

**LexicalEnvironment – ФУНКЦИЯ ИЗНУТРИ**

При запуске функция создает объект **LexicalEnvironment**, записывает туда аргументы, функции и переменные. Процесс инициализации выполняется в том же порядке, что и для глобального объекта, который, вообще говоря, является частным случаем лексического окружения.

В отличие от **window**, объект **LexicalEnvironment** является внутренним, он скрыт от прямого доступа.

* Каждая функция при создании получает ссылку **[[Scope]]** на объект с переменными, в контексте которого была создана.
* При запуске функции создается новый объект с переменными. В него копируется ссылка на внешний объект из **[[Scope]].**
* При поиске переменных он осуществляется **сначала в текущем объекте переменных, а потом — по этой ссылке.** Благодаря этому в функции доступны внешние переменные. Если поиск не находит значения в текущем окружении, **он ищет на окружение выше, потом еще выше и т.д.** вплоть до **window.**

**Вложенные функции (функции внутри функций)** обрабатываются в точности так же, как и глобальные. Единственная разница — они создаются в объекте переменных внешней функции, а не в **window**..

**Память**

Объект переменных внешней функции существует в памяти до тех пор, пока существует хоть одна внутренняя функция, ссылающаяся на него через свойство **[[Scope]].** Например, если функция возвращает что-то, ссылка есть. Если функция ничего не возвращает, она удаляется после использования

**Исключение из правил:**

При создании функции с использованием **new Function**, её свойство **[[Scope]]** ссылается не на текущий **LexicalEnvironment, а на window.**

var a = 1; //внешнаяя переменная в window

function getFunc() {

var a = 2; //создана в локальном LexicalEnvironment

var func = new Function('', 'alert(a)'); //функция берет данные из window

return func;

}

getFunc()(); // 1, из window

**Hoisting или подьем**

Интерпретатор **поднимает** все обьявления в начало функции, потому не имеет значения, в каком месте была обьявлена переменная, поэтому в языке javascript лучше обьявлять переменные в начале функции.

В javascript интерпретатор заранее знает о том, какие переменные обьявлены в какой области видимости (scope). Потому, если обрашение к переменной происходит до ее обьявления, интерпретатор берет значение не выше по цепочке видимости, как в других языках программирования, а даст ей значение **undefined**:

var fuck = function () {

console.log(i); //выводим переменную

var i = 15; //до ее обьявления

};

fuck(); // undefined

**Обьект счетчика + функция**

В данном примере создается обьект с функциями внутри, которые могут быть методами обьекта

function makeCounter() {

var currentCount = 0;

function counter() { // возвращаемся к функции

return ++currentCount;

}

counter.set = function(value) { // ...и добавляем ей методы!

currentCount = value;

};

counter.reset = function() {

currentCount = 0;

};

return counter;

}

var counter = makeCounter();

counter.set(5)

console.log(counter()); //6. Обьект вызывается как функция

**Замыкания функций**

**Замыкание - это внутренняя функция.**

Поскольку область видимости делится на глобальную и локальную, интерпретатор забывает о тех переменных, которые были обьявлены внутри функций, после заврешения работы этих функций. Удаляет из обьекта [[scope]].

function sum(x,y) {

... // неявно создался объект [[scope]]

var z // в [[scope]] записалось свойство z

z = x+y // нашли переменную в [[scope]], [[scope]].z = x+y

return z // нашли переменную в [[scope]], return [[scope]].z

// функция закончилась, [[scope]] никому больше не нужен и умирает вместе с z

}

Когда одна функция создается внутри другой, то ей передается ссылка на объект с локальными переменными **[[scope]]** внешней функции.

Благодаря существованию этой ссылки, из внутренней функции можно получить переменные внешней функции - через ссылку на ее [[scope]]. Сначала ищем у себя, затем - во внешнем [[scope]] - и так далее по цепочке до самого объекта window.

Замыкание - это когда объект локальных переменных [[scope]] внешней функции остается жить после ее завершения.

Внутренняя функция может обратиться к нему в любой момент и получить переменную внешней функции.

function addHideHandler(sourceId, targetId) {

// создан объект [[scope]] со свойствами sourceId, targetId

// записать в [[scope]] свойство sourceNode

var sourceNode = document.getElementById(sourceId)

// записать в [[scope]] свойство handler

var handler = function() {

var targetNode = document.getElementById(targetId)

targetNode.style.display = ‘none’

}

sourceNode.onclick = handler

// функция закончила выполнение, и тут - самое интересное!

}

При запуске функции все происходит стандартно:

1. создается [[scope]]
2. туда записываются локальные переменные
3. внутренняя функция получает ссылку на [[scope]]
4. Но в самом конце - внутренняя функция присваивается sourceNode.onclick. Внешняя функция закончила свою работу, но внутренняя - может запуститься когда-нибудь потом.

Интерпретатор javascript не проводит анализ - понадобятся ли внутренней функции переменные из внешней, и какие переменные могут быть нужны.

Вместо этого он просто оставляет весь **[[scope]]** внешней функции в живых.\

Но если **внутри функции есть еще одна функция (вложенная)**, которая ссылается на переменную в области видимости выше, эта **переменная записывается в обьект и не удаляется.**

Как сделать так, что бы счетчик был приватен (доступен только из функции), но мог в то же время сохранять промежуточное значение?

Выход: сделать самовызывающуюся функцию, внутри нее сделать счетчик и еще одну функцию, которую можно было бы вызывать дополнительно для изменения счетчика выше

var counter = (function(){ //функция должна быть самовызывающейся

var count = 0; // счетчик во внешней области видимости

return function(){ //возвращает функцию,

return count++; //которая может обратиться к внешнему счетчику

})(); //<== функция должна быть самовызывающейся

console.log(counter()); // 0

console.log(counter()); // 1, поскольку ссылка на [[scope]] жива

console.log(counter()); //2

console.log(counter()); //3

console.log(counter()); //4

**Прием проектирования МОДУЛЬ**

Создается для того, что бы пременные модуля были доступны только внутри области видимости модуля. Вызвать функцию сразу после создания можно только в том случае, если это Function Expression (var func = functiuon(){..}();)

1. **Библиотека lodash устроена примерно так:**

;(function() {

function lodash(value) { ... } // lodash - основная функция для библиотеки

var version = '2.4.1'; // вспомогательная переменная

// код функции size, пока что доступен только внутри

function size(collection) {

return Object.keys(collection).length;

}

// присвоим в lodash size и другие функции, которые нужно вынести из модуля

lodash.size = size

window.\_ = lodash; // "экспортировать" lodash наружу из модуля

}());

В начале кода выше находится точка с запятой - если получится, что несколько JS-файлов объединены в один и программист забыл поставить точку с запятой, то будет ошибка

1. **Вариант через return**

var lodash = (function() {

var version;

function assignDefaults() { ... }

return {

defaults: function() { }

}

})();

Здесь, кстати, скобки вокруг внешней function() { ... } не обязательны, ведь функция и так объявлена внутри выражения присваивания, а значит – является Function Expression.

Тем не менее, лучше их ставить, для улучшения читаемости кода, чтобы было сразу видно, что это не простое присвоение функции.

**Псевдомассив arguments**

Если в функцию подается больше аргументов, чем она может принять, можно использовать **arguments:**

function sayHi() { //нету передаваемых аргументов

for (var i=0; i<arguments.length; i++) {//пока i меньше длинны массива аргументов

alert("Привет, " + arguments[i]);

}

}

sayHi("Винни", "Пятачок"); // 'Привет, Винни', 'Привет, Пятачок'

Рекомендуется никогда не изменять arguments: arguments[0] = 5;

**arguments** – это не массив. Это обычный **объект**, просто ключи числовые и есть **length**. На этом сходство заканчивается. Никаких особых методов у него нет, и **методы массивов он тоже не поддерживает**. В массив можно превратить так:

var args = [];

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

args[i] = arguments[i];

}

arguments.callee и arguments.callee.caller устарели, вместо них используется Named Function Expression (NFE)

**Именнованные аргументы**

**Именованные аргументы** — альтернативная техника работы с аргументами, которая вообще не использует arguments.

Некоторые языки программирования позволяют передать параметры как-то так: **f(width=100, height=200),** то есть по именам, а что не передано, тех аргументов нет. Это очень удобно в тех случаях, когда аргументов много, сложно запомнить их порядок и большинство вообще не надо передавать, по умолчанию подойдёт.

Такая ситуация часто встречается в компонентах интерфейса. Например, у «меню» может быть масса настроек отображения, которые можно «**подкрутить**» но обычно нужно передать всего один-два главных параметра, а остальные возьмутся по **умолчанию**:

function showWarning(options) {

var width = options.width || 200; // по умолчанию

var height = options.height || 100;

var title = options.title || "Предупреждение";

//...

}

**ГЛОБАЛЬНЫЙ ОБЬЕКТ**

Все функции и переменные, которые являются глобальными, выполняются в браузере в обьекте **window. Это глобальный обьект.**

Выполнение глобального скрипта при инициализации :

* На первой фазе происходит инициализация, подготовка к запуску. Во время инициализации скрипт сканируется на предмет объявления функций вида **Function Declaration**, а затем — на предмет объявления переменных **var**. Каждое такое объявление добавляется в **window**.

Функции, объявленные как **Function Declaration**, создаются сразу работающими, а переменные — равными **undefined**.

* На второй фазе — собственно, выполнение. Присваивание (**=**) значений переменных происходит на второй фазе, когда поток выполнения доходит до соответствующей строчки кода.

Создать переменную можно присвоением:

window.a = 5;

alert(a); // 5

Браузер в начале определяет функции и переменные. Скобки условных операторов не считаются. В результате данный код будет работать:

if ("a" in window) { // в if видно что window.a уже есть

var a = 1; // поэтому эта строка сработает

}

alert(a); //1

**THIS ИСПОЛЬЗОВАНИЕ В ОБЬЕКТЕ**

Значением **this** является объект перед «точкой», в контексте которого вызван метод:

var user = { firstName: "Вася" };

var admin = { firstName: "Админ" };

function func() {

alert( this.firstName );

}

user.f = func;

admin.g = func;

// this равен объекту перед точкой:

user.f(); // Вася

admin.g(); // Админ

admin['g'](); // Админ (не важно, доступ к объекту через точку или квадратные скобки)

**Ссылочный тип (Reference type)**

Контекст this никак не привязан к функции, даже если она создана в объявлении объекта. Чтобы this передался, нужно вызвать функцию именно через точку (или квадратные скобки).

Любой более хитрый вызов приведёт к потере контекста, например:

var user = {

name: "Вася",

hi: function() { alert(this.name); },

bye: function() { alert("Пока"); }

};

user.hi(); // Вася (простой вызов работает)

// а теперь вызовем user.hi или user.bye в зависимости от имени

(user.name == "Вася" ? user.hi : user.bye)(); // undefined

Причина кроется в деталях работы вызова obj.method(). Он состоит из двух независимых операций: точка . – получение свойства и скобки () – его вызов (предполагается, что это функция).

Функция, как мы говорили раньше, сама по себе не запоминает контекст. Чтобы «донести его» до скобок, JavaScript применяет «финт ушами» – точка возвращает не функцию, а значение специального «ссылочного» типа Reference Type.

Этот тип представляет собой связку «base-name-strict», где:

base – как раз объект,

name – имя свойства,

strict – вспомогательный флаг для передачи use strict.

То есть, ссылочный тип (Reference Type) – это своеобразное «три-в-одном». Он существует исключительно для целей спецификации, мы его не видим, поскольку любой оператор тут же от него избавляется:

Скобки () получают из base значение свойства name и вызывают в контексте base.

Другие операторы получают из base значение свойства name и используют, а остальные компоненты игнорируют.

Поэтому любая операция над результатом операции получения свойства, кроме вызова, приводит к потере контекста.

**Методы обьекта**

Обьект, как и массив, даже пустой - всгда **true**

Если в объекте присутствует метод **toString**, который возвращает примитив, то он используется для преобразования.

var user = {

firstName: 'Василий',

toString: function() {

return 'Пользователь ' + this.firstName;

}

};

alert( user ); // Пользователь Василий

Для численного преобразования объекта используется метод **valueOf.** Если **valueOf** нету, вызывается **toString**

var room = {

number: 777,

valueOf: function() { return this.number; },

toString: function() { return this.number; }

};

alert( +room ); // 777, вызвался valueOf

delete room.valueOf; // valueOf удалён

alert( +room ); // 777, вызвался toString

**СОЗДАНИЕ ОБЬЕКТОВ ЧЕРЕЗ NEW**

Для создания множества обьектов используется конструктор **new.** Функцию с **new** обычно называют с большой буквы

* Автоматически создается новый пустой **объект**.
* Ключевое слово **this** получает ссылку на этот **объект**.
* Функция **выполняется**. Как правило, она модифицирует **this**, добавляет методы, свойства.
* Возвращается **this**.

function Animal(name) {

this.name = name;

this.canWalk = true;

}

var animal = new Animal("ёжик");

Получается обьект:

animal = {

name: "ёжик",

canWalk: true

}

**УКАЗАНИЕ THIS: “CALL”, “APPLY”**

Вызов func.call(context, a, b...) – то же, что обычный вызов func(a, b...), но первый аргумент call становится её this, а остальные передаются «как есть».

function showFullName() {

alert( this.firstName + " " + this.lastName );

}

var user = {

firstName: "Василий",

lastName: "Петров"

};

// функция вызовется с this=user

showFullName.call(user) // "Василий Петров"

**Копирование метода, одолжить метод (method borrowing)**

**call** вызывает методы работы с массивом из прототипа.

Можно добавить **arguments**, у которых нету своих методов, новый метод

Так как метод **join** находится в **Array.prototype,** то можно вызвать его оттуда напрямую.

Что бы не вносить путанницу в обьекты, которые уже имеют свои встроенные методы, и не перезаписывать существующие, можно добавить новый метод через **call(context, arg1, arg2, …):**

function printArgs() {

var join = [].join; // скопируем ссылку на функцию в переменную

var argStr = join.call(arguments, ':');

alert( argStr );

}

printArgs(1, 2, 3); // сработает и выведет 1:2:3

Вместо [].join можно написать:

var join = Array.prototype.join;

Это эффективнее, потому что не создаётся лишний объект массива [].

В JavaScript есть очень простой способ сделать из arguments настоящий массив. Для этого возьмём метод массива: **slice**. По стандарту вызов **arr.slice(start, end)** создаёт новый массив и копирует в него элементы массива arr от start до end. А если start и end не указаны, то копирует весь массив.

function printArgs() {

// вызов arr.slice() скопирует все элементы из this в новый массив

var args = [].slice.call(arguments);

alert( args.join(', ') ); // args - полноценный массив из аргументов

}

printArgs('Привет', 'мой', 'мир'); // Привет, мой, мир

Как и в случае с **join**, такой вызов технически возможен потому, что **slice** для работы требует только нумерованные свойства и **length**. Всё это в arguments есть.

**Метод Apply**

**apply аналогичен что call, только c массвом аргументов [arg1,arg2,arg3] вместо обычного списка**

Если нам неизвестно, с каким количеством аргументов понадобится вызвать функцию, можно использовать более мощный метод: **apply**.

showFullName.call(user, 'firstName', 'surname');

showFullName.apply(user, ['firstName', 'surname']); //работа идентична

Преимущество **apply** перед **call** отчётливо видно, когда мы формируем массив аргументов динамически. Встроенная функция **Math.max(a, b, c...),** является элегантным способом вычислить максимальное значение в массиве

var arr = [];

arr.push(1);

arr.push(5);

arr.push(2);

// получить максимум из элементов arr

alert( Math.max.apply(null, arr) ); // 5

В данном случае в качестве контекста можно передавать что угодно, поскольку в своей внутренней реализации метод **Math.max**не использует **this**, потому вместо **this** мы пишем **null.**

**ПРИВЯЗКА КОНТЕКСТА И КАРРИНГ: BIND**

При попытке вызвать метод обьекта через какую-то функцию, может теряться контекст this. Это часто бывает при вызове из setTimeOut:

var user = {

firstName: "Вася",

sayHi: function() {

alert( this.firstName );

}

};

setTimeout(user.sayHi, 1000); // undefined (не Вася!)

Для решения этой проблемы можно использовать обертку

setTimeout(function() {

user.sayHi(); // Вася

}, 1000);

Или использовать встроенный метод **bind:**

setTimeout(user.sayHi.bind(user), 1000); // аналог через встроенный метод

**Карринг**

Карринг - привязка аргументов

Например, есть функция умножения двух чисел **mul(a, b):**

function mul(a, b) {

return a \* b;

};

При помощи **bind** создадим функцию **double**, удваивающую значения. Это будет вариант функции mul с фиксированным первым аргументом:

// double умножает только на два

var double = mul.bind(null, 2); // контекст фиксируем null, он не используется

alert( double(3) ); // = mul(2, 3) = 6

alert( double(4) ); // = mul(2, 4) = 8

**ДЕКОРАТОРЫ И ОБЕРТКИ**

Декоратор - тот же самый bind, только самописный. Декораторы способны упростить рутинные, повторяющиеся задачи, вынести их из кода функции.

Например, создадим декоратор, который принимает функцию и массив, который описывает для какого аргумента какую проверку типа применять:

function checkPermissionDecorator(f) {

return function() {

if (isAdmin()) {

return f.apply(this, arguments);

}

alert( 'Недостаточно прав' );

}

}

function save() { ... }

// Теперь вызов функции save() проверяет права

save = checkPermissionDecorator(save);

**ДЕСКРИПТОРЫ, ГЕТТЕРЫ И СЕТТЕРЫ**

Object.defineProperty(obj, prop, {descriptor}) – основной метод управления свойствами

**obj -** объект, в котором объявляется свойство.

**prop -** имя свойства, которое нужно объявить или модифицировать.

**descriptor** объект, который описывает поведение свойства:

**value** — значение свойства, по умолчанию **undefined**

**writable** — значение свойства можно менять, если **true**. По умолчанию **false**.

**configurable** — если true, то свойство можно удалять, а также менять его в дальнейшем при помощи новых вызовов **defineProperty**. По умолчанию **false**.

**enumerable** — если **true**, то свойство будет участвовать в переборе **for..in**. По умолчанию **false**.

**get** — назначение нового свойство. Функция, которая возвращает значение свойства. По умолчанию **undefined**.

**set** — присвоение существующему свойству новое значение. Функция, которая записывает значение свойства. По умолчанию **undefined**.

Чтобы избежать конфликта, запрещено одновременно указывать значение **value** и функции **get/set.** Либо значение, либо функции для его чтения-записи, одно из двух. Также запрещено и не имеет смысла указывать **writable** при наличии **get/set** - функций.

var user = {

firstName: "Вася",

surname: "Петров"

}

Object.defineProperty(user, "fullName", {

get: function() {

return this.firstName + ' ' + this.surname;

},

set: function(value) {

var split = value.split(' ');

this.firstName = split[0];

this.surname = split[1];

}

});

user.fullName = "Петя Иванов";

alert( user.firstName ); // Петя

alert( user.surname ); // Иванов

**Object.defineProperties(obj, {descriptors})** позволяет обьявить несколько свойств сразу:

function User(fullName) {

this.fullName = fullName;

Object.defineProperties(this, {

firstName: {

get: function () {

return this.fullName.split(" ")[0];

}

},

lastName:{

get: function() {

return this.fullName.split(" ")[1];

},

set: function(value){

return this.fullName = this.firstName + " " + value;

}

}

})

}

var vasya = new User("Василий Попкин");

console.log( vasya.firstName ); // Василий

console.log( vasya.lastName ); // Попкин

vasya.lastName = 'Сидоров'; // запись в lastName

console.log( vasya.fullName ); // Василий Сидоров

**Object.keys** возвращает только **enumerable**-свойства (которые видны)

**Object.getOwnPropertyNames** — возвращает все, и видимые и невидимые

Object.getOwnPropertyDescriptor(obj, prop) - Возвращает дескриптор для свойства obj[prop].

**Используются очень редко:**

**Object.preventExtensions(obj)** Запрещает добавление свойств в объект.

**Object.seal(obj)** Запрещает добавление и удаление свойств, все текущие свойства делает configurable: false.

**Object.freeze(obj)** Запрещает добавление, удаление и изменение свойств, все текущие свойства делает configurable: false, writable: false.

**Object.isExtensible(obj), Object.isSealed(obj), Object.isFrozen(obj)** Возвращают true, если на объекте были вызваны методы **Object.preventExtensions/seal/freez**

**СТАТИЧЕСКИЕ И ФАБРИЧНЫЕ МЕТОДЫ**

Методы и свойства, которые не привязаны к конкретному экземпляру объекта, называют «**статическими**». Их записывают прямо в саму функцию-конструктор. Статический метод также можно использовать для функций, которые вообще не требуют наличия объекта.

Полиморфный конструктор:

function Journal() { /\*...\*/ }

//используем статический метод – идет не в теле функции-конструктора, отдельно!

Journal.formatDate = function(date) {

return date.getDate() + '.' + (date.getMonth()+1) + '.' + date.getFullYear();

}

// ни одного объекта Journal нет, просто форматируем дату

alert( Journal.formatDate(new Date) );

**Полиморфные конструкторы лучше использовать там, где нужна именно полиморфность**, т.е. когда непонятно, какого типа аргумент передадут, и хочется в одном конструкторе охватить все варианты.

А в остальных случаях отличная альтернатива — **фабричные** методы, позволяет лучше отлавливать ошибки.

Фабричный конструктор:

function User() {

this.sayHi = function() {

alert(this.name)

};

}

User.createAnonymous = function() {

var user = new User;

user.name = 'Аноним';

return user;

}

User.createFromData = function(userData) {

var user = new User;

user.name = userData.name;

user.age = userData.age;

return user;

}

var guest = User.createAnonymous(); // Использование

guest.sayHi(); // Аноним

var knownUser = User.createFromData({

name: 'Вася',

age: 25

});

knownUser.sayHi(); // Вася

Например, нужно добавить создание администратора, без аргументов. Фабричный метод сделать легко: **User.createAdmin = function() { ... }.** А для полиморфного конструктора вызов без аргумента создаст анонима, так что нужно добавить параметр – «тип посетителя» и усложнить этим код.

**ТИПЫ ДАННЫХ [[CLASS]], INSTANCEOF И УТКИ**

Во всех встроенных объектах есть специальное свойство **[[Class]],** в котором хранится информация о его типе или конструкторе. Это свойство — внутреннее. Явно получить его нельзя, но можно прочитать его «в обход», воспользовавшись методом **toString** стандартного объекта **Object**.

var toString = {}.toString;

var arr = [1, 2];

alert( toString.call(arr) ); // [object Array]

var date = new Date;

alert( toString.call(date) ); // [object Date]

var user = { name: "Вася" };

alert( toString.call(user) ); // [object Object]

Если выводить код вверху через **console.log(),** будет ошибка. Следует обернуть консоль в скобки:

(console.log( toString.call(user) ));

Для массивов есть встроенный метод **Array.isArray(arr),** возвращает **true** или **false:**

alert( Array.isArray([1,2,3]) ); // true

**instanceof** позволяет проверить, создан ли объект данной функцией, причём работает для любых функций — как встроенных, так и наших. Этот оператор учитывает наследование

function User() {...}

var user = new User();

alert( user instanceof User ); // true

**Утиная типизация**

Смысл утиной типизации — в проверке необходимых методов и свойств. Например, мы можем проверить, что объект — массив, не вызывая **Array.isArray**, а просто уточнив наличие важного для нас метода, например **splice**:

var something = [1, 2, 3];

if (something.splice) {

alert( «массив» );

}

**Пример полиморфной функции**

Функция **sayHi(who),** которая будет говорить «Привет» своему аргументу, причём если передан массив — то «Привет» каждому:

function sayHi(who) {

if (Array.isArray(who)) {

who.forEach(sayHi);

} else {

alert( 'Привет, ' + who );

}

}

sayHi("Вася"); // Привет, Вася - Вызов с примитивным аргументом

sayHi(["Саша", "Петя"]); // Привет, Саша... Петя - // Вызов с массивом

**ФОРМАТ JSON (JavaScript Object Notation)**

Данные в формате JSON (RFC 4627) представляют собой:

* JavaScript-объекты { ... }
* Массивы [ ... ]
* Значения одного из типов:
* строки в двойных кавычках,
* число,
* логическое значение true/false,
* null.

Метод **JSON.parse(str)** превратит строку с данными в формате **JSON** в JavaScript-объект/массив/значение:

var user = '{ "name": "Vs", "age": 35, "isAdmin": false, "friends": [0,1,2,3] }'; //двойные кавычки у свойств, одинарные – у самого обьекта

user = JSON.parse(user);

alert( user.friends[1] ); // 1

Поскольку **JSON.parse()** работает со строками, внутренние свойства и функции могут быть недоступны. Для интеллектуального восстановления из строки у **JSON.parse(str, reviver)** есть второй параметр **reviver**, который является функцией **function(key, value**).

var str = '{"title":"Конференция","date":"2014-11-30T12:00:00.000Z"}';

var event = JSON.parse(str, function(key, value) {

if (key == 'date') return new Date(value);

return value;

});

alert( event.date.getDate() ); // сработает

Эта хрень так же работает для вложенных обьектов

Метод **JSON.stringify(value, replacer, space)** преобразует («сериализует») значение в JSON-строку.

var event = {

title: "Конференция",

date: "сегодня"

};

var str = JSON.stringify(event);

alert( str ); // {"title":"Конференция","date":"сегодня"}

event = JSON.parse(str); // Обратное преобразование.

Некоторые обьекты **JSON** преобразовать не может, например **window.** Будет ошибка.Потому стоит исключить его из потока любом доступным образом

Во втором параметре **JSON.stringify(value, replacer)** можно указать массив свойств, которые подлежат сериализации.

var user = {

name: "Вася",

age: 25,

window: window

};

alert( JSON.stringify(user, ["name", "age"]) ); // {"name":"Вася","age":25}

Для более сложных ситуаций вторым параметром можно передать функцию **function(key, value),** которая возвращает сериализованное value либо undefined, если его не нужно включать в результат:

var user = {

name: "Вася",

age: 25,

window: window

};

var str = JSON.stringify(user, function(key, value) {

if (key == 'window') return undefined;

return value;

});

alert( str ); // {"name":"Вася","age":25}

В методе JSON.stringify(value, replacer, space) есть ещё третий параметр space.

Если он является числом — то уровни вложенности в JSON оформляются указанным количеством пробелов, если строкой — вставляется эта строка.

var str = JSON.stringify(user, "", 4);

**Метод toJSON**

Когда метод **JSON.stringify** преобразует обьект в строку, он первым делом проверяет наличие метода **toJSON.**

toJSON передает только те свойства, которые перечисленны в нем. Используется, если например у нас есть обьект с ID пользователей, которые мы не хотим передавать:

var user = {

name: "Frank",

id: 42345,

lastvisit: Date.now(),

friends: [23432, 234324, 546546, 768678],

toJSON: function() {

return { // передает только то, что перечисленно в toJSON

name: this.name,

lastVisit: this.lastVisit

}

}

};

var userData = JSON.stringify(user) // {"name":"Frank", "lastVisit":12312321312541}

**ТАЙМЕРЫ - SETTIMEOUT И SETINTERVAL**

**setTimeout**

var timerId = setTimeout(func / code, delay[, arg1, arg2...])

Передача аргументов (не сработает в IE9-):

function func(phrase, who) {

alert( phrase + ', ' + who );

}

setTimeout(func, 1000, "Привет", "Вася"); // Привет, Вася

Можно передать функцию, если предварительно сделать ее **анонимной**:

setTimeout(function() { alert('Привет') }, 1000);

**clearTimeout**

clearTimeout(timerId); //отмена таймера

**setInterval**- для назначения интервалов повторного запуска кода. Что будет, если не нажать "ок" в алерте сразу? В разных браузерах обрабатывается по-разному

var timerId = setInterval(func / code, delay[, arg1, arg2...])

var timerId = setInterval(function() { alert( "тик" ); }, 2000); // начать повторы с интервалом 2 сек

**Рекурсивный setTimeout -** важная альтернатива **setInterval**. **Рекурсивный setTimeout** гарантирует паузу между вызовами, **setInterval** — не гарантирует (время, потраченное на само выполнение функции в рекурсивном таймере интервалов учитывается)

var timerId = setTimeout(function tick() {

alert( "тик" );

timerId = setTimeout(tick, 2000); //рекурсия

}, 2000);

setTimeout(function() { // через 5 сек остановить повторы

clearInterval(timerId);

alert( 'стоп' );

}, 5000);

Управление памятью для таймеров \ счетчиков

Сборщик мусора в JavaScript не чистит функции, назначенные в таймерах, пока таймеры актуальны.

При передаче функции в **setInterval/setTimeout** создаётся внутренняя ссылка на неё, через которую браузер её будет запускать, и которая препятствует удалению из памяти, даже если функция анонимна.

Для **setTimeout** — внутренняя ссылка исчезнет после исполнения функции.

Для **setInterval** — ссылка исчезнет при очистке таймера.

Так как функция также тянет за собой всё замыкание, то ставшие неактуальными, но не отменённые **setInterval** могут приводить к излишним тратам памяти.

**Eval() – запуск кода из строки.** Многие блокираторы строку **eval()** считают вредоносным кодом

**ПЕРЕХВАТ ОШИБОК, «TRY..CATCH...FINALLY»**

Конструкция **try..catch..finally** позволяет обработать произвольные ошибки в блоке кода.

Это удобно в тех случаях, когда проще сделать действие и потом разбираться с результатом, чем долго и нудно проверять, не упадёт ли чего. Ошибку, которая произойдёт в коде, запланированном «на будущее», например, в setTimeout, **try..catch** не поймает:

try {

setTimeout(function() {

throw new Error(); // вылетит в консоль

}, 1000);

} catch (e) {

alert( "не сработает" );

}

На момент запуска функции, назначенной через **setTimeout**, этот код уже завершится, интерпретатор выйдет из блока **try..catch.**

Чтобы поймать ошибку внутри функции из **setTimeout**, и **try..catch** должен быть в той же функции

try {

alert('Начало блока try');

lalala; // ошибка, переменная не определена!

alert('Конец блока try');

} catch(e) {

alert('Ошибка ' + e.name + ":" + e.message + "\n" + e.stack);

}

alert("Потом код продолжит выполнение...");.

Свойства:

**name** - Тип ошибки. Например, при обращении к несуществующей переменной: "ReferenceError".

**message -** Текстовое сообщение о деталях ошибки.

**stack -** Везде, кроме IE8-, есть также свойство stack, которое содержит строку с информацией о последовательности вызовов, которая привела к ошибке.

Оператор **throw** генерирует ошибку.

Технически, в качестве объекта ошибки можно передать что угодно, это может быть даже не объект, а число или строка, но всё же лучше, чтобы это был объект, желательно — совместимый со стандартным, то есть чтобы у него были как минимум свойства **name** и **message**.

В качестве конструктора ошибок можно использовать встроенный конструктор: **new Error(message)** или любой другой.

Есть ряд конструкторов: **SyntaxError, ReferenceError, RangeError**

var data = '{ "age": 30 }'; // данные неполны

try {

var user = JSON.parse(data); // <-- выполнится без ошибок

if (!user.name) {

throw new SyntaxError("Данные некорректны");

}

alert( user.name );

} catch (e) {

alert( "Извините, в данных ошибка" );

}

**Оборачивание исключений**

Важнейший общий подход к проектированию — каждый участок функционала должен получать информацию на том уровне, который ему необходим. Этот подход называют «оборачиванием» исключения, поскольку мы берём ошибки «более низкого уровня» и «заворачиваем» их в ReadError, которая соответствует текущей задаче.

function ReadError(message, cause) {

this.message = message;

this.cause = cause;

this.name = 'ReadError';

this.stack = cause.stack;

}

function readData() {

var data = '{ bad data }';

try {

// ...

JSON.parse(data);

// ...

} catch (e) {

// ...

if (e.name == 'URIError') {

throw new ReadError("Ошибка в URI", e);

} else if (e.name == 'SyntaxError') {

throw new ReadError("Синтаксическая ошибка в данных", e);

} else {

throw e; // пробрасываем

}

}

}

try {

readData();

} catch (e) {

if (e.name == 'ReadError') {

alert( e.message );

alert( e.cause ); // оригинальная ошибка-причина

} else {

throw e;

}

}

Секцию **finally** используют, чтобы завершить начатые операции при любом варианте развития событий. Блок **finally** срабатывает при любом выходе из **try..catch**, в том числе и **return**:

function func() {

try {

return 1; // сразу вернуть значение

} catch (e) {

/\* ... \*/

} finally {

alert( 'finally' );

}

}

alert( func() ); // сначала finally, потом 1

**window.onerror** – отлов ошибок в глобальном обьекте. Если в него записать функцию, то она выполнится и получит в аргументах сообщение ошибки, текущий URL и номер строки, откуда «выпала» ошибка. Часто используется для отправки отчета разработчикам:

<script>

window.onerror = function(message, url, lineNumber) {

alert("Поймана ошибка, выпавшая в глобальную область!\n" +

"Сообщение: " + message + "\n(" + url + ":" + lineNumber + ")");

};

function readData() {

error(); // ой, что-то не так

}

readData();

</script>

**ООП В JAVASCRIPT – ФУНКЦИОНАЛЬНЫЙ СТИЛЬ**

В терминологии ООП отделение и защита внутреннего интерфейса называется **инкапсуляция.**

В программировании мы будем разделять методы и свойства объекта на две группы:

**Внутренний интерфейс** — это свойства и методы, доступ к которым может быть осуществлен только из других методов объекта, их также называют **«приватными»** (есть и другие термины, встретим их далее).

**Внешний интерфейс** — это свойства и методы, доступные снаружи объекта, их называют **«публичными».**

Для изменения свойст обьекта обычно используют геттеры и сеттеры. Часто используется единный геттер\сеттер, если он принимает аргмуенты – становится сеттером, без аргументов – он геттер.

Используется наследование – создание новых классов на основе существующих. Наследник не имеет доступа к приватным свойствам родителя. Чтобы наследник имел доступ к свойству, оно должно быть записано в **this**:

function Machine(power) {

this.\_enabled = false; // вместо var enabled

this.enable = function() {

this.\_enabled = true;

};

this.disable = function() {

this.\_enabled = false;

};

}

function CoffeeMachine(power) {

Machine.call(this); //передаем базовому конструктору контекст

this.enable();

alert( this.\_enabled ); // true

}

var coffeeMachine = new CoffeeMachine(10000);

Подчёркивание в начале свойства — общепринятый знак, что свойство является внутренним, предназначенным лишь для доступа из самого объекта и его наследников. Такие свойства называют **защищёнными**.

**ПРОТОТИПЫ В ОБЬЕКТЕ**

Все обьекты можно назвать ссылочным типом. Когда мы присваиваем число или строку, эта переменная хранит значение - это касается всех простых типов данных:

var a = 10, b = 20;

a = b;

b = 15;

console.log(a) // 20, сохранила предыдущее значение

С обьектами работает иначе, поскольку в обьектах мы храним не значения, а ссылки:

var obj1 = {x: 10}, obj2 = {x: 20};

obj1 = obj2;

obj2.x = 15;

console.log(obj1.x); // 15, берет значение по ссылке

obj1.x = 45;

console.log(obj2.x); // 45, не важно, по какой ссылке выводить, они одинаковы

Любой обьект в джаваскрипт имет второй связаный с ним обьект **prototype**. Обьект наследует свойства **prototype**, или проще говоря, все свойства, которые есть у **prototype** будут доступны через дочерний обьект. Это **единственный тип наследования, который есть в javascript.**

Что бы «обучить» конструктор новому методу, можно задать свойство через **прототип**:

Пример 1.

function Dog (breed) { //есть конструктор Dog

this.breed = breed;

};

var buddy = new Dog("golden Retriever"); //создаем новый обьект с породой

Dog.prototype.bark = function() { //задаем новый метод для конструктора

console.log("Woof");

};

buddy.bark(); //"Woof"

var snoopy = new Dog("Beagle");

snoopy.bark(); //"Woof"

**Класс** в языке javascript - условное понятие и под классом понимают множество обьектов, которые наследуют свои свойства от одного прототипа. Обьекты в примере выше относятся к одному классу.

Пример наследования дочерних классов:

var WebDeveloper = Object.create(Person) // тоже самое, что new Person

//хотим добавить к классу массив skills, но мы не хотим переписывать его по-новой

WebDeveloper.properies = function(name, age, gender, skills){

Person.properies.apply(this, arguments); // пихаем вышесозданный класс

this.skills = skills || []; // что бы не было undefined, если аргумент не передан

return this;

};

var developer = Object.create(WebDeveloper).properies("Jack", 21, "male", ["html", "css"]);

console.log(developer.skills); // html, css

**ООП – ПРОТОТИПНЫЙ СТИЛЬ**

Объекты в JavaScript можно организовать в цепочки так, чтобы свойство, не найденное в одном объекте, автоматически искалось бы в другом.

Если один объект имеет специальную ссылку **\_\_proto\_\_** на другой объект, то при чтении свойства из него, если свойство отсутствует в самом объекте, оно ищется в объекте **\_\_proto\_\_**

var animal = {

eats: true

};

var rabbit = {

jumps: true

};

rabbit.\_\_proto\_\_ = animal;

// в rabbit можно найти оба свойства

console.log( rabbit.jumps ); // true

console.log( rabbit.eats ); // true

Объект, на который указывает ссылка **\_\_proto\_\_,** называется «прототипом». В данном случае получилось, что animal является прототипом для rabbit.

Также говорят, что **объект rabbit прототипно наследует» от animal**.

Обратим внимание — прототип используется исключительно при чтении. Запись значения, например, rabbit.eats = value или удаление delete rabbit.eats — работает напрямую с объектом.

Вызов **obj.hasOwnProperty(prop)** возвращает **true**, если свойство **prop** принадлежит самому объекту **obj** (не учитывая те свойства, которые обьект взял через прототипное наследование от родителя) иначе **false**.

Для того, чтобы перебрать свойства самого объекта, достаточно профильтровать key через **hasOwnProperty**:

var animal = {

eats: true

};

var rabbit = {

jumps: true,

\_\_proto\_\_: animal

};

for (var key in rabbit) {

if (!rabbit.hasOwnProperty(key)) continue; // пропустить "не свои" свойства

alert( key + " = " + rabbit[key] ); // выводит только "jumps"

}

**Object.create(null)**

Объект, создаваемый при помощи **Object.create(null)** не имеет прототипа, а значит в нём нет лишних свойств (типа встроенных по умолачнию методов toString).

var data = Object.create(null);

data.text = "Привет";

alert(data.text); // Привет

alert(data.toString); // undefined

Есть дополнительные методы работы с прототипами, но используются они редко, т.к. есть **\_\_proto\_\_:**

Object.getPrototypeOf(obj) (кроме IE8-)

Object.setPrototypeOf(obj, proto) (кроме IE10-)

Object.create(proto, descriptors) (кроме IE8-)

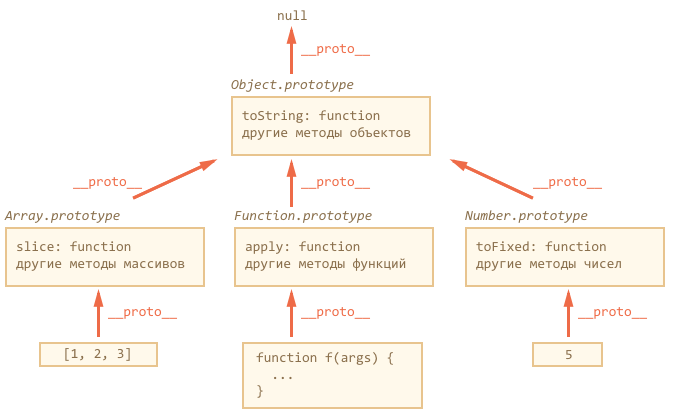
**СВОЙСТВО PROTOTYPE И СОЗДАНИЕ ОБЪЕКТОВ ЧЕРЕЗ NEW**

1. В JS все типы данных создаются через вызов **new**

**{} - короткая форма new Object**

**[] - короткая форма new Array**

1. При выполнении new Object, создаваемому объекту ставится \_\_proto\_\_ по prototype конструктора, который в данном случае равен встроенному Object.prototype.
2. В дальнейшем при обращении к obj.toString() – функция будет взята из Object.prototype.



**Примитивы**

Примитивы не являются объектами, но методы берут из соответствующих прототипов:

**Number.prototype, Boolean.prototype, String.prototype.**

var user = "Вася"; // создали строку (примитив)

alert( user.toUpperCase() ); // "Вася"

// создан временный объект new String

// вызван метод

// new String уничтожен, результат возвращён

Значения null и undefined не имеют свойств

**Свойство constructor**

Когда мы создаем новый конструктор через **new**, всем обьектам присваивается свойство **constructor.** Так же, это свойство есть и на прототипе:

console.log(worker\_1.constructor) // [Function] - this name = name

console.log(Person.prototype.constructor) // тоже самое

Свойство **constructor** легко потерять, т.к. интерпретатор не следит за ним. В частности, при перезаписи **Rabbit.prototype = { jumps: true }** свойства **constructor** больше не будет.

Если мы хотим, чтобы возможность получить конструктор, всё же, была, то можно при перезаписи гарантировать наличие **constructor** вручную: можно поступить аккуратно и добавить свойства к встроенному **prototype** без его замены:

Rabbit.prototype.jumps = true // сохранится встроенный constructor

**НАСЛЕДОВАНИЕ**

В JS нормально, когда обьекты, функции и массивы наследуют методы от родителей (см. картинку \_\_proto\_\_ выше). Такое наследование лучше функционального стиля, так как не дублирует методы в каждом объекте.

// --------- Класс-Родитель ------------

// Конструктор родителя пишет свойства конкретного объекта

function Animal(name) {

this.name = name;

this.speed = 0;

}

// Методы хранятся в прототипе

Animal.prototype.run = function() {

alert(this.name + " бежит!")

}

// --------- Класс-потомок -----------

// Конструктор потомка

function Rabbit(name) {

Animal.apply(this, arguments); // включаем все методы и свойства Animal

}

// Унаследовать

Rabbit.prototype = Object.create(Animal.prototype);

Rabbit.prototype.constructor = Rabbit; // \* Желательно и constructor сохранить

// Методы потомка

Rabbit.prototype.run = function() {

// Вызов метода родителя внутри своего

Animal.prototype.run.apply(this);

alert( this.name + " подпрыгивает!" );

};

// Готово, можно создавать объекты

var rabbit = new Rabbit('Кроль');

rabbit.run();

\*В prototype по умолчанию всегда находится свойство **constructor**, указывающее на функцию-конструктор. В частности, Rabbit.prototype.constructor == Rabbit. Если мы рассчитываем использовать это свойство, то при замене prototype через Object.create нужно его явно сохранить.

**РАЗНИЦА В ФУНКЦИОНАЛЬНОМ И ПРОТОТИПНОМ СТИЛЕ**

К примеру, есть у нас приватное свойство name и метод sayHi в функциональном стиле ООП:

function Animal(name) {

this.sayHi = function() {

alert( name );

};

}

var animal = new Animal("Зверь");

animal.sayHi(); // Зверь

При задании методов в прототипе мы не сможем её так оставить, ведь методы находятся вне конструктора, у них нет общей области видимости, поэтому приходится записывать name в сам объект, обозначив его как защищённое:

function Animal(name) {

this.\_name = name;

}

Animal.prototype.sayHi = function() {

alert( this.\_name );

}

var animal = new Animal("Зверь");

animal.sayHi(); // Зверь

Впрочем, недостаток этот — довольно условный. Ведь при наследовании в функциональном стиле также пришлось бы писать this.\_name, чтобы потомок получил доступ к этому значению.

**instanceof**

**instanceof** - бинарный оператор, с помощью которого можно сделать проверку на принадлежность к классу.

instanceof может проверить только те обьекты, родителем которых является функция, а не другой обьект:

var worker\_2 = {

\_\_proto\_\_: worker\_1,

};

console.log( worker\_2 instanceof worker\_1); //error: expecting Object

console.log( worker\_2 instanceof Person); // true

**instanceof** не будет работать, если изменен прототип:

function Rabbit() {}

var rabbit = new Rabbit();

Rabbit.prototype = {}; // изменили prototype...

alert( rabbit instanceof Rabbit ); // false, instanceof перестал работать!

**СВОИ ОШИБКИ, НАСЛЕДОВАНИЕ ОТ ERROR**

Разобрался плохо, сложно. Все плохо

**ПРИМЕСИ (MIXIN)**

В **JavaScript** невозможно унаследовать от двух и более объектов. Ссылка **\_\_proto\_\_** — только одна.

Самый простой вариант **примеси** — это объект с полезными методами, которые мы просто копируем в нужный прототип.

var sayHiMixin = { //примеси

sayHi: function() {

alert("Привет " + this.name);

},

sayBye: function() {

alert("Пока " + this.name);

}

};

// использование:

function User(name) {

this.name = name;

}

// передать методы примеси

for(var key in sayHiMixin) User.prototype[key] = sayHiMixin[key];

// User "умеет" sayHi

new User("Вася").sayHi(); // Привет Вася