Типы данных

Числа(Number)

let n = 123;

n = 12.345;

**Infinity** – бесконечность, получатся либо через деления на ноль, либо через явное задавание alert( Infinity ); // Infinity, либо при операциях с бесконечностью

console.log(Infinity ); /\* Infinity \*/

console.log(Infinity + 1 ); /\* Infinity \*/

console.log(Math.pow(10, 1000)); /\* Infinity \*/

console.log(Math.log(0) ); /\* -Infinity \*/

Math.log(-1); // NaN, выход за пределы диапазона

Math.log(0); // -Infinity

Math.log(1); // 0

Math.log(10); // 2.302585092994046

console.log(1 / Infinity ); /\* 0 \*/

**NaN** – ошибка not a number, при делении строки на числа и тд alert( "не число" / 2 ); // NaN, такое деление является ошибкой

NaN является неравным (посредством сравнения через ==, !=, ===, and !==) любому другому значению, включая другое значение NaN. Используйте [Number.isNaN()](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/Number/isNaN) или [isNaN()](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/isNaN),

isNaN(NaN); // true

isNaN(undefined); // true

isNaN({}); // true

isNaN(true); // false

isNaN(null); // false

isNaN(37); // false

// strings

isNaN("37"); // false: "37" преобразуется в число 37 которое не NaN

isNaN("37.37"); // false: "37.37" преобразуется в число 37.37 которое не NaN

isNaN(""); // false: пустая строка преобразуется в 0 которое не NaN

isNaN(" "); // false: строка с пробелом преобразуется в 0 которое не NaN

isNaN("37,5"); // true

// Даты

isNaN(new Date()); // false

isNaN(new Date().toString()); // true

// Пример почему использование isNaN не всегда уместно

isNaN("blabla") // true: "blabla" преобразовано в число.

// При парсинге преобразуется в число при неудаче возвращает NaN

**BigInt**

В JavaScript тип «number» не может содержать числа больше, чем (253-1) (т. е. 9007199254740991), или меньше, чем -(253-1) для отрицательных чисел.

const bigInt = 1234567890123456789012345678901234567890n;

Строки

let str = "Привет";

let str2 = 'Одинарные кавычки тоже подойдут';

let phrase = `Обратные кавычки позволяют встраивать переменные ${str}`;

1. Двойные кавычки: "Привет".
2. Одинарные кавычки: 'Привет'.
3. Обратные кавычки: `Привет`.

alert( `Привет, ${name}!` ); // Привет, Иван!

// Вставим выражение

alert( `результат: ${1 + 2}` ); // результат: 3

Ещё одно преимущество обратных кавычек — они могут занимать более одной строки, вот так:

let guestList = `Guests:

\* John

\* Pete

\* Mary

`;

alert(guestList); // список гостей, состоящий из нескольких строк

Также есть спецсимволы, например:

|  |  |
| --- | --- |
| \n | Перевод строки |
| \t | Знак табуляции |
| \xXX | Символ с шестнадцатеричным юникодным кодом XX, например, '\x7A' — то же самое, что 'z'. |
| \uXXXX | Символ в кодировке UTF-16 с шестнадцатеричным кодом XXXX, например, \u00A9 — юникодное представление знака копирайта, ©. Код должен состоять ровно из 4 шестнадцатеричных цифр. |

BOOLEAN

Логический тип данных, со своими операторами

result = value1 || value2 || value3;

Оператор || выполняет следующие действия:

* Вычисляет операнды слева направо.
* Каждый операнд конвертирует в логическое значение. Если результат true, останавливается и возвращает исходное значение этого операнда.
* Если все операнды являются ложными (false), возвращает последний из них.

Оператор && выполняет следующие действия:

* Вычисляет операнды слева направо.
* Каждый операнд преобразует в логическое значение. Если результат false, останавливается и возвращает исходное значение этого операнда.
* Если все операнды были истинными, возвращается последний.

Ответ: null, потому что это первое «ложное» значение из списка.

alert( 1 && null && 2 );

NULL

Специальное значение null не относится ни к одному из типов, описанных выше.

Оно формирует отдельный тип, который содержит только значение null:

let age = null;

null является определённым значением отсутствия объекта, тогда как [undefined](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/undefined) обозначает неопределённость. Например:

typeof null // object (не "null" из соображений обратной совместимости)

typeof undefined // undefined

null === undefined // false

null == undefined // true

UNDEFINED

Оно означает, что «значение не было присвоено».

Если переменная объявлена, но ей не присвоено никакого значения, то её значением будет undefined:

let age;

alert(age); // выведет "undefined"

Технически мы можем присвоить значение undefined любой переменной:

let age = 123;

// изменяем значение на undefined

age = undefined;

alert(age); // "undefined"

…Но так делать не рекомендуется. Обычно null используется для присвоения переменной «пустого» или «неизвестного» значения, а undefined – для проверок, была ли переменная назначена.

OBJECT

Пустой объект («пустой ящик») можно создать, используя один из двух вариантов синтаксиса:

let user = new Object(); // синтаксис "конструктор объекта"

let user = {}; // синтаксис "литерал объекта"

let user = { // объект

name: "John", // под ключом "name" хранится значение "John"

age: 30 // под ключом "age" хранится значение 30

}

Для обращения к свойствам используется запись «через точку»:

// получаем свойства объекта:

alert( user.name ); // John

alert( user.age ); // 30

Для удаления свойства delete user.age;

"likes birds": true // имя свойства из нескольких слов должно быть в кавычках

};

Обращение к свойству объекта, написанного через пробел

let user = {};

// присваивание значения свойству

user["likes birds"] = true;

// получение значения свойства

alert(user["likes birds"]); // true

// удаление свойства

delete user["likes birds"];

Для вычисляемого свойства

let fruit = prompt("Какой фрукт купить?", "apple");

let bag = {

[fruit]: 5, // имя свойства будет взято из переменной fruit

};

alert( bag.apple ); // 5, если fruit="apple"

выведе 5 только если apple

Свойства из переменной – САМЫЙ РАСПРОСТАНЕННЫЙ СЛУЧАЙ ОБЯВЛЕНИЯ

function makeUser(name, age) {

return {

name: name,

age: age

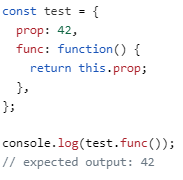
// ...другие свойства

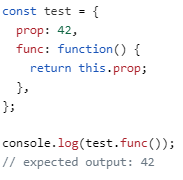
};

}

let user = makeUser("John", 30);

alert(user.name); // John





function User(name) {

this.name = name;

this.isAdmin = false;

}

let user = new User("Вася");

alert(user.name); // Вася

Операторы языка

Базовые операторы(есть унарные и бинарные операторы, работающие с операндами

Поддерживаются следующие математические операторы:

* Сложение +,
* Вычитание -,
* Умножение \*,
* Деление /,
* Взятие остатка от деления %, alert( 5 % 2 );// 1, остаток от деления 5 на 2
* Возведение в степень \*\*. alert( 2 \*\* 3 ); // 8

Работа со строками

let s = "моя" + "строка";

alert(s); // моястрока

Идет приведение типов

alert(2 + 2 + '1' ); // будет "41", а не "221"

Здесь операторы работают один за другим. Первый + складывает два числа и возвращает 4, затем следующий + объединяет результат со строкой, производя действие 4 + '1' = 41.

Унарный + преобразует строку в число

let apples = "2";

let oranges = "3";

// оба операнда предварительно преобразованы в числа

alert( +apples + +oranges ); // 5

// более длинный вариант

// alert( Number(apples) + Number(oranges) ); // 5

По поводу ПРИОРИТЕТА надо только знать чтоу унарный идет больший приоритет чем у бинарных

ПРИСВАИВАНИЕ – присваиваем переменной значение справо, также есть сокращенные формы  += и \*=: НЕТ РАБОТАЕТ ДЛЯ /=, -=

ИНКРЕМЕНТ ДЕКРЕМЕНТ

ПОСТФИКСАЛЬНАЯ ФОРМА(ПОСТИНКРЕМЕНТ) – ВОЗВРАЩАЕТ СТАРОЕ ЗНАЧЕНИЕ, ПРЕФИКСАЛЬНЯ ФОРМА – ВОЗВРАЩАЕТ УЖЕ НОВОЕ ЗНАЧЕНИЕ

1. вычитание всегда преобразует к числу, значит строка " -9 " становится числом -9 (пробелы по краям обрезаются).
2. null становится 0 после численного преобразования.
3. undefined становится NaN после численного преобразования.
4. Пробельные символы, такие как \t и \n, по краям строки игнорируются при преобразовании в число, так что строка " \t \n", аналогично пустой строке, становится 0 после численного преобразования.

ПОБИТОВЫЕ ОПЕРАТОРЫ

a & b И, ЕСЛИ СОВПОДАЮТ 1

a | b ИЛИ, ЕСЛИ ЕСТЬ ХОТЬ ОДНА 1

a ^ b ИСКЛ ИЛИ , 1 ЕСЛИ ХОТЯБЫ ОДНА 1, ВСЕ ОСТАЛЬНОЕ 0  
~ a НЕ , МЕНЯЕТ 0 НА 1

a << b СДВИГ ВЛЕВО, B БИТА В ЛЕВО + НУЛЕВЫЕ БИТЫ СПРАВО

9 (по осн.10)

= 00000000000000000000000000001001 (по осн.2)

--------------------------------

9 << 2 (по осн.10)

= 00000000000000000000000000100100 (по осн.2)

= 36 (по осн.10)

ПОЧТИ ВСЕГДА РАВНЯЕТЬСЯ УМНОЖЕНИЮ НА 2 ЧИСЛА СДВИГОВЫХ БИТОВ

a >> b СДВИГ ВПРАВО,B БИТА В ПРАВО + СЛЕВА ИДЕТ B КОПИИ САМОГО КРАЙНЕГО ЛЕВОГО БИТА

-9 (по осн.10)

= 11111111111111111111111111110111 (по осн.2)

--------------------------------

-9 >> 2 (по осн.10)

= 11111111111111111111111111111101 (по осн.2) = -3 (по осн.10)

a >>> b СДВИГ ВПРАВО С ЗАПОЛНЕНИЕМ НУЛЯМИ, С ЛЕВА БУДУТ НУЛИ

-9 (по осн.10)

= 11111111111111111111111111110111 (по осн.2)

--------------------------------

-9 >>> 2 (по осн.10)

= 00111111111111111111111111111101 (по осн.2)

= 1073741821 (по осн.10)

Числа отрицательные становятся положительными

В JavaScript побитовые операторы ^, &, | выполняются после сравнений ==.

Например, в сравнении a == b^0 будет сначала выполнено сравнение a == b, а потом уже операция ^0, как будто стоят скобки (a == b)^0.

Обычно это не то, чего мы хотим. Чтобы гарантировать желаемый порядок, нужно ставить скобки: a == (b^0).

ОПЕРАТОР ЗАПЯТАЯ, ГРУБО ГОВОРЯ КОГДА НЕСКОЛЬКО ОПЕРАЦИЙ ПОДРЯТ, И ОНА ВОЗВРАЩАЕТ РЕЗУЛЬТАТ ТОЛЬКО ПОСЛЕДНЕГО

let a = (1 + 2, 3 + 4);

alert( a ); // 7 (результат вычисления 3 + 4)

Пожалуйста, обратите внимание, что оператор , имеет очень низкий приоритет, ниже =, поэтому скобки важны в приведённом выше примере.

Без них в a = 1 + 2, 3 + 4 сначала выполнится +, суммируя числа в a = 3, 7, затем оператор присваивания = присвоит a = 3, а то, что идёт дальше, будет игнорировано. Всё так же, как в (a = 1 + 2), 3 + 4.

ОПЕРАТОР сравнения

alert( 2 > 1 ); // true (верно)

alert( 2 == 1 ); // false (неверно)

alert( 2 != 1 ); // true (верно)

СРАВНЕНИЕ СТРОК

alert( 'Я' > 'А' ); // true

alert( 'Коты' > 'Кода' ); // true

alert( 'Сонный' > 'Сон' ); // true

1. Сначала сравниваются первые символы строк.
2. Если первый символ первой строки больше (меньше), чем первый символ второй, то первая строка больше (меньше) второй. Сравнение завершено.
3. Если первые символы равны, то таким же образом сравниваются уже вторые символы строк.
4. Сравнение продолжается, пока не закончится одна из строк.
5. Если обе строки заканчиваются одновременно, то они равны. Иначе, большей считается более длинная строка.

Например, в JavaScript имеет значение регистр символов. Заглавная буква "A" не равна строчной "a". Какая же из них больше? Строчная "a". Почему? Потому что строчные буквы имеют больший код во внутренней таблице кодирования, которую использует JavaScript (Unicode)

При сравнении значений разных типов JavaScript приводит каждое из них к числу.

alert( '2' > 1 ); // true, строка '2' становится числом 2

alert( '01' == 1 ); // true, строка '01' становится числом 1

alert( true == 1 ); // true

alert( false == 0 ); // true

С ОПРАТОРОМ ==(НЕ СТРОГОЕ СРАВНЕНИЕ) ЕСТЬ ПРОБЛЕМЫ

alert( 0 == false ); // true

alert( '' == false ); // true

ОН НЕ МОЖЕТ ОТЛИЧИТЬ 0 И FALSE так как == приводит все к числам, тоже самое и со строкой

ДЛЯ ЭТОГО ЕСТЬ === КОТОРЫЙ ПРОВЕРЯЕТ ТИПЫ

alert( 0 === false ); // false, так как сравниваются разные типы

C NULL И UNDEFINED СВОИ ПРИКОЛЫ ПРИ СРАВНЕНИИ

С ===

alert( null === undefined ); // false, ТАК КАК РАЗНЫЕ ТИПЫ

С ==

alert( null == undefined ); // true, РАВНЫ ДРУГ ДРУГУ НО НЕ РАВНЫ ДРУГИМ

С < > <= >=

Значения null/undefined преобразуются к числам: null становится 0, а undefined – NaN.

alert( null > 0 ); // (1) false

alert( null == 0 ); // (2) false

alert( null >= 0 ); // (3) true

alert( undefined > 0 ); // false (1)

alert( undefined < 0 ); // false (2)

alert( undefined == 0 ); // false (3)

* Сравнения (1) и (2) возвращают false, потому что undefined преобразуется в NaN, а NaN – это специальное числовое значение, которое возвращает false при любых сравнениях.
* Нестрогое равенство (3) возвращает false, потому что undefined равно только null, undefined и ничему больше.
* ЛОГИЧЕСКИЕ ОПЕРАТОРЫ

|| (ИЛИ), && (И) и ! (НЕ).

|| (ИЛИ) НАХОДИТ ПЕРВОЕ ИСТИННОЕ ЗНАЧЕНИЕ, ИЛИ ПОСЛЕДНИЕ ЕСЛИ ИСТИННОЕ НЕ НАЙДЕНО

alert( null || 1 ); // 1 (первое истинное значение)

alert( null || 0 || 1 ); // 1 (первое истинное значение)

alert( undefined || null || 0 ); // 0 (поскольку все ложно, возвращается последнее значение)

&& (И) НАХОДИТ ПЕРВОЕ ЛОЖНОЕ ЗНАЧЕНИЕ, ИЛИ ПОСЛЕДНИЕ ЕСЛИ ЛОЖНОГО НЕ НАЙДЕНО, ЕЩЕ У && БОЛЬШИЙ ПРИОРИТЕТ ЧЕМ У ||

alert( null && 5 ); // null

alert( 0 && "no matter what" ); // 0

! (НЕ)

1. Сначала приводит аргумент к логическому типу true/false.
2. Затем возвращает противоположное значение.

alert( !true ); // false

alert( !0 ); // true

ЕСЛИ !! ТО ПРОСТО ПРЕОБРАЗУЕТ К ЛОГИЧЕСКОМУ ЗНАЧЕНИЮ(НЕ МЕНЯ ПРИ ЭТОМ ЕГО,САМО ЛОГ ЗНАЧЕНИЕ).

ТЕРНАРНЫЙ ОПЕРТОР ?

let result = условие ? значение1 : значение2;

Сначала вычисляется условие: если оно истинно, тогда возвращается значение1, в противном случае – значение2.

let age = prompt('Возраст?', 18);

let message = (age < 3) ? 'Здравствуй, малыш!' :

(age < 18) ? 'Привет!' :

(age < 100) ? 'Здравствуйте!' :

'Какой необычный возраст!';

alert( message );

?? оператор сравнения с null или undefined

То есть оператор ?? возвращает первый аргумент, если он не null/undefined, иначе второй.

let user = "Иван";

alert(user ?? "Аноним"); // Иван

возвращает первое определённое значение.(если несколько)

ЦИКЛЫ И УПРАВЛЯЮЩИЕ КОНСТРУКЦИИ

IF

if (year == 2015) {

alert( "Правильно!" );

alert( "Вы такой умный!" );

}

* Число 0, пустая строка "", null, undefined и NaN становятся false. Из-за этого их называют «ложными» («falsy») значениями.
* Остальные значения становятся true, поэтому их называют «правдивыми» («truthy»).

if (0) { // 0 is falsy

...

} НЕ ВЫПОЛНИТЬСЯ

if (1) { // 1 is truthy

...

}

ELSE

if (year == 2015) {

alert( 'Да вы знаток!' );

} else {

alert( 'А вот и неправильно!' ); // любое значение, кроме 2015

}

if (year < 2015) {

alert( 'Это слишком рано...' );

} else if (year > 2015) {

alert( 'Это поздновато' );

} else {

alert( 'Верно!' );

}

ЦИКЛЫ

let i = 0;

while (i < 3) { // выводит 0, затем 1, затем 2

alert( i );

i++;

}

let i = 0;

do {

alert( i );

i++;

} while (i < 3);

for (let i = 0; i < 3; i++) { // выведет 0, затем 1, затем 2

alert(i);

}

for (начало; условие; шаг)

ТАКЖЕ МОЖНО БЕЗ НАЧАЛА( ЕСЛИ ПЕРЕМЕННАЯ ЗАДАНАЯ ЗАРАНЕЕ)

let i = 0; // мы уже имеем объявленную i с присвоенным значением

for (; i < 3; i++) { // нет необходимости в "начале"

alert( i ); // 0, 1, 2

}

И БЕЗ ШАГА

let i = 0;

for (; i < 3;) {

alert( i++ );

}

BREAK

let sum = 0;

while (true) {

let value = +prompt("Введите число", '');

if (!value) break; // (\*)

sum += value;

}

alert( 'Сумма: ' + sum );

CONTINUE

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

// если true, пропустить оставшуюся часть тела цикла

if (i % 2 == 0) continue;

alert(i); // 1, затем 3, 5, 7, 9

}

SWITCH

let a = 2 + 2;

switch (a) {

case 3:

alert( 'Маловато' );

break;

case 4:

alert( 'В точку!' );

break;

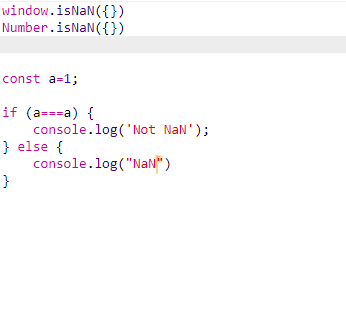
case 5:

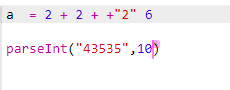
alert( 'Перебор' );

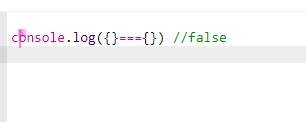
break;

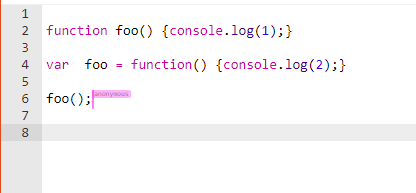
default:

alert( "Нет таких значений" );

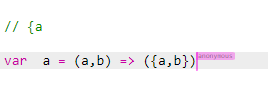








Строгий и нестрогий режим



ОбЪявления и использование переменных

[Переменная](https://ru.wikipedia.org/wiki/%D0%9F%D0%B5%D1%80%D0%B5%D0%BC%D0%B5%D0%BD%D0%BD%D0%B0%D1%8F_(%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5)) – это «именованное хранилище» для данных. Мы можем использовать переменные для хранения товаров, посетителей и других данных.

LET

переменная let всегда видна именно в том блоке, где объявлена, и не более.

* Видны только после объявления и только в текущем блоке.
* Нельзя переобъявлять (в том же блоке).
* При объявлении переменной в цикле for(let …) – она видна только в этом цикле. Причём каждой итерации соответствует своя переменная let.

При вызове через var у нас будет показываться undefined? Поэтому при вызове несущ переменной у нас будет ошибка

Если в блоке кода дважды объявить одну и ту же переменную let, будет ошибка:

let user;

let user; // SyntaxError: 'user' has already been declared

VAR

Область видимости переменных var ограничивается либо функцией, либо, если переменная глобальная, то скриптом.

Используя var, можно переобъявлять переменную сколько угодно раз. Повторные var игнорируются:

var user = "Pete";

var user; // ничего не делает, переменная объявлена раньше

// ...нет ошибки

Объявления переменных var обрабатываются в начале выполнения функции (или запуска скрипта, если переменная является глобальной). переменные var считаются объявленными с самого начала исполнения функции вне зависимости от того, в каком месте функции реально находятся их объявления (при условии, что они не находятся во вложенной функции).

МОЖЕТ ПРОСХОДИТЬ ВСПЛЫВАНИЕ, грубо говоря

function sayHi() {

alert(phrase);

var phrase = "Привет";

}

sayHi();

function sayHi() {

var phrase; // объявление переменной срабатывает вначале...

alert(phrase); // undefined

phrase = "Привет"; // ...присвоение - в момент, когда исполнится данная строка кода.

}

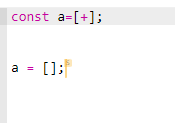
sayHi();

CONST

Анлогичен LET

Значение нельзя изменить, кроме как работу со свойствами объекта

Обычно мы используем буквы в верхнем регистре для констант, которые «жёстко закодированы».сслка



Объявление и использование функций

ГРУБО ГОВОРЯ СУЩЕСТВУЮТ ДЛЯ ПОВТОРНОГО ИСПОЛЬЗОВАНИЯ КОДА

function showMessage() {

alert( 'Всем привет!' );

}

1 ЛОКАЛЬНЫЕ ПЕРЕМЕННЫЕ(LET, CONST) ВИДНЫ ТОЛЬКО В ТЕЛЕ ФУНКЦИИ

2 ГЛОБАЛЬНЫЕ ПЕРЕМЕННЫЕ(VAR) МОЖНО ИСПОЛЬЗОВАТЬ В ТЕЛЕ, И ДАЖЕ ИЗМЕНЯТЬ ИХ ЗНАЧЕНИЕ

Внешняя переменная используется, только если внутри функции нет такой локальной.

3 МОЖЕМ ПЕРЕДОВАТЬ ПАРАМЕТРЫ(АРГУМЕНТЫ)

function showMessage(from, text) { // аргументы: from, text

alert(from + ': ' + text);

}

showMessage('Аня', 'Привет!'); // Аня: Привет! (\*)

showMessage('Аня', "Как дела?"); // Аня: Как дела? (\*\*)

если параметр не указан, то его значением становится undefined.

function showMessage(from, text = "текст не добавлен") {

alert( from + ": " + text );

}

showMessage("Аня"); // Аня: текст не добавлен

В ЭТОМ МЫ ЗАДАЕМ ГРУБО ГОВОРЯ ДЕФОЛТНОЕ ЗНАЧЕНИЕ

ДИРЕКТИВА RETURN

вернуть результат

function sum(a, b) {

return a + b;

}

let result = sum(1, 2);

alert( result ); // 3

Возможно использовать return и без значения. Это приведёт к немедленному выходу из функции.

function showMovie(age) {

if ( !checkAge(age) ) {

return;

}

alert( "Вам показывается кино" ); // (\*)

// ...

}

Именование функций:

* Имя функции должно понятно и чётко отражать, что она делает. Увидев её вызов в коде, вы должны тут же понимать, что она делает, и что возвращает.
* Функция – это действие, поэтому её имя обычно является глаголом.
* Есть много общепринятых префиксов, таких как: create…, show…, get…, check… и т.д. Пользуйтесь ими как подсказками, поясняющими, что делает функция.

let func = new Function([arg1, arg2, ...argN], functionBody);

let sum = new Function('a', 'b', 'return a + b');

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

Главное отличие от других способов объявления функции, которые были рассмотрены ранее, заключается в том, что функция создаётся полностью «на лету» из строки, переданной во время выполнения.

Но new Function позволяет превратить любую строку в функцию. Например, можно получить новую функцию с сервера и затем выполнить её:

ЗАМЫКАНИЯ

function getFunc() {

let value = "test";

let func = new Function('alert(value)');

return func;

}

getFunc()();

function getFunc() {

let value = "test";

let func = function() { alert(value); };

return func;

}

getFunc()(); // "test", из лексического окружения функции getFunc

Область видимости. Что такое «поднятие» (hoisting)

ДОСТУП К ПЕРЕМЕННОЙ

**if** (true) {

**const** message = 'Hello'

}

console.log(message) // ReferenceError: message is not defined

**ТАКЖЕ С FOR WHILE**

С ФУНКЦИЯМИ ПРИМЕРНО ТОЖЕ САМОЕ

**function** **run**() {

// область видимости функции run()

**var** message = 'Беги, Форрест, беги!'

console.log(message)

}

run() // 'Беги, Форрест, беги!'

console.log(message) // ReferenceError

ОБЛАСТИ ВИДИМОСТИ МОГУТ БЫТЬ ВЛОЖЕННЫМИ

**function** **run**() {

// область видимости функции run()

**const** message = 'Беги, Форрест, беги!'

**if** (true) {

// область видимости блока if

**const** friend = 'Бубба'

console.log(message) // 'Беги, Форрест, беги!'

}

console.log(friend) // ReferenceError

}

run()

НО НЕ РАБОТАЕТ ДЛЯ ФУНКЦИЙ

**function** **outer**() {

// область видимости функции outer()

**let** v = 'Я из области видимости функции outer()!'

**function** **inner**() {

// область видимости функции inner()

console.log(v) // 'Я из области видимости функции outer()!'

}

**return** inner

}

**const** f = outer()

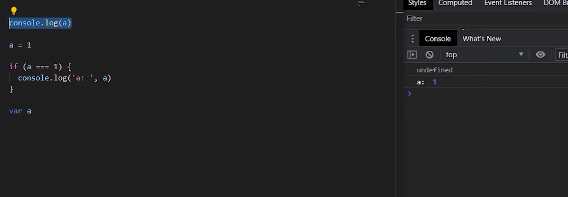
f()

Поднятие или hoisting — это механизм в JavaScript, в котором переменные и объявления функций, передвигаются вверх своей области видимости перед тем, как код будет выполнен.

console.log(variable); // **Выводит: ReferenceError: variable is not defined**

В JavaScript, ReferenceError появляется при попытке доступа к предварительно необъявленной переменной.

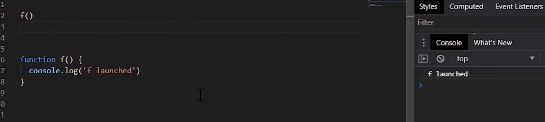
Запомните и держите в уме одну важную деталь, **JavaScript непреклонно сначала объявляет, а уже затем инициализирует наши переменные.**



ГРУБО ГОВОРЯ МЫ ИСПОЛЬЗУЕМ ПЕРМЕННУЮ ЖО ЕЕ ОБЯВЛЕНИЯ,

С LET НЕЛЬЗЯ

С ФУНКЦИЯМИ НАДО СНАЧАЛО ОБЯВИТЬ И ТОЛЬКО ПОТОМ ВЫЗЫВАТЬ



НО ЕСЛИ ПРОСТО ВЫЗЫВАЕМ ТО РАБОТАЕТ, ЧЕРЕЗ ФУНК ВЫРАЖЕНИЕ ТО РАБОТАТ КАК С ПЕРЕМЕННЫМИ

ФУНКЦИИ СТРЕЛКИ

Короткая форма обычных функций

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

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

или для сложных

let sum = (a, b) => { // фигурная скобка, открывающая тело многострочной функции

let result = a + b;

return result; // при фигурных скобках для возврата значения нужно явно вызвать return

};

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

TDZ Временная мертвая зона (Temporal Dead Zone)

Это связано с тем, что, когда механизм JavaScript сканирует код и находит объявления переменных, он либо поднимает их до вершины области (при обнаружении объявлений var), либо помещает объявления в TDZ (при обнаружении объявлений let и const). Доступ к переменным в TDZ вызывает ошибку времени выполнения. Только после выполнения оператора объявления переменной переменная будет удалена из TDZ и затем станет доступной.

var value = 'global'

// Пример 1

(function() {

console.log(value)

let value = 'local'

}())

// Пример 2

{

console.log(value)

cosnt value = 'local'

}

В обоих случаях результаты не будут распечатаны.**"gobal"** Но ошибка**ReferenceError: value is not defined**, Это из-за ТДЗ.

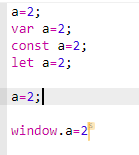
//console.log(aLet) // would throw ReferenceError

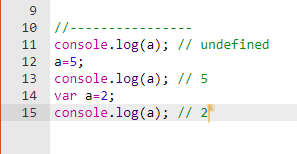
let aLet;

console.log(aLet); // undefined

aLet = 10;

console.log(aLet); // 10

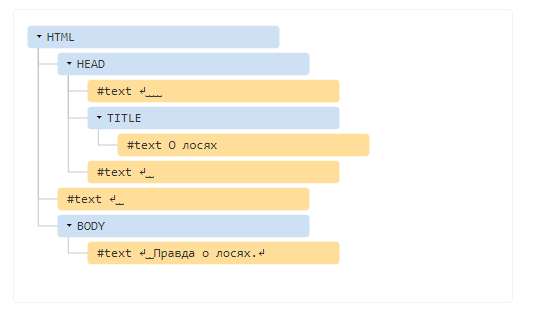




Что такое DOM? Из чего состоит DOM, Узлы и их виды.

В соответствии с объектной моделью документа («Document Object Model», коротко DOM), каждый HTML-тег является объектом. Вложенные теги являются «детьми» родительского элемента. Текст, который находится внутри тега, также является объектом.

DOM – это представление HTML-документа в виде дерева тегов. Вот как оно выглядит:



Теги являются узлами-элементами (или просто элементами). Они образуют структуру дерева: <html> – это корневой узел, <head> и <body> его дочерние узлы и т.д.

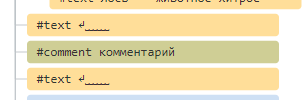
Текст внутри элементов образует текстовые узлы, обозначенные как #text. Текстовый узел содержит в себе только строку текста. У него не может быть потомков, т.е. он находится всегда на самом нижнем уровне.

Например, в теге <title> есть текстовый узел "О лосях".

Пробелы и переводы строки – это полноправные символы, как буквы и цифры. Они образуют текстовые узлы и становятся частью дерева DOM. Так, в примере выше в теге <head> есть несколько пробелов перед <title>, которые образуют текстовый узел #text (он содержит в себе только перенос строки и несколько пробелов).

ТАКЖЕ ЕСТЬ ПОНЯТИЕ КАК АВТОИСПРАВЛЕНИЕ, КОГДА БРАУЗЕР ДОБАВЛЯЕТ НЕОБХОДИМЫЕ ЭЛЕМЕНТЫ(НАПРИМЕР САММ ЗЫКРЫВАЕЬ ТЕГ, ИЛИ ДОБАВЛЯЕТ НЕОБХОДИМЫЕ ТЕГИ(HTML <tbody>))

УЗЕЛ КОМЕНТАРИЯ



1. document – «входная точка» в DOM.
2. узлы-элементы – HTML-теги, основные строительные блоки.
3. текстовые узлы – содержат текст.
4. комментарии – иногда в них можно включить информацию, которая не будет показана, но доступна в DOM для чтения JS.

Что такое BOM? Что в себя включает BOM?

Объектная модель браузера (BOM от англ. Browser Object Model) позволяет JavaScript "общаться" с браузером.

Объект **window** поддерживается всеми браузерами. Он представляет окно браузера.

Объект **window.screen** содержит информацию об экране пользователя.

Объект **window.screen** можно использовать без префикса **window**.

**screen.width** возвращает ширину экрана пользователя в пикселях:

document.getElementById("demo").innerHTML =

"Ширина экрана: " + screen.width;

**screen.height** возвращает высоту экрана пользователя в пикселях:

document.getElementById("demo").innerHTML =

"Высота экрана: " + screen.height;

Объект **window.location** может использоваться для получения адреса (URL) текущей страницы и перенаправления браузера на новую страницу.

**window.location.href** возвращает URL текущей страницы.

document.getElementById("demo").innerHTML =

"Адрес страницы: " + window.location.href;

**window.location.hostname** возвращает имя интернет хоста (текущей страницы). (не совсем понял)

document.getElementById("demo").innerHTML =

"Имя хоста страницы: " + window.location.hostname;

**window.location.pathname** возвращает путь текущей страницы. (через /)

document.getElementById("demo").innerHTML =

"Путь страницы: " + window.location.pathname;

**window.location.protocol** возвращает веб-протокол страницы.

document.getElementById("demo").innerHTML =

"Протокол страницы: " + window.location.protocol;

document.getElementById("demo").innerHTML =

"Номер порта: " + window.location.port;

Большинство браузеров не покажут номер порта по умолчанию (80 для http и 443 для https

window.location.assign("https://msiter.ru") на новую страницу

Объект **window.history** содержит историю посещенных браузером страниц.

Метод **history.back()** загружает предыдущий URL в списке посещенных страниц.

**history.forward()** загружает следующий URL в списке посещенных страниц.  
 function goForward() {

window.history.forward()

}

<input type="button" value="Вперед" onclick="goForward()">

Объект **window.navigator** содержит информацию о браузере посетителя страницы.

Свойство **appName** возвращает имя браузера, как приложения:

Довольно странно, но именем приложения для браузеров IE11, Chrome, Firefox и Safari является "Netscape".

**appCodeName** возвращает кодовое имя браузера:

Кодовым именем для браузеров Chrome, Firefox, IE, Safari и Opera является "Mozilla".

**language** возвращает язык браузера:

navigator.language;

Alert prompt confirm

Объект **window** позволяет выполнять код через заранее определенный периодвремени или временные интервалы.

**setTimeout(*функция, миллисекунды*)** — Выполняет функцию по истечению заданного числа миллисекунд.

<button onclick="setTimeout(myFunction, 3000)">Try it</button>

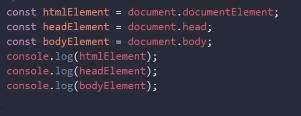
**clearTimeout()** останавливает выполнение функции, вызванной в методе **setTimeout()**.

<button onclick="myVar = setTimeout(myFunction, 3000)">Запуск</button>

<button onclick="clearTimeout(myVar)">Остановить</button>

Навигация и поиск элементов по DOM-дереву

Document – обект, который предоставляет доступ к любому узлу,

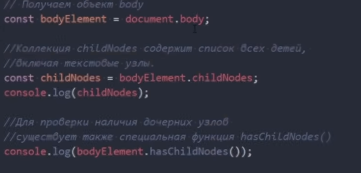


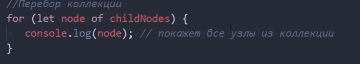
Document.body.firstChild/lastChild

Как первый узел будет перевод строки

Document.body.childNodes – коллекция(примерно как массив, можно использовать for of) всех элементов(узлов) body

hasChildNodes -дает true если есть дочерние и false если нет

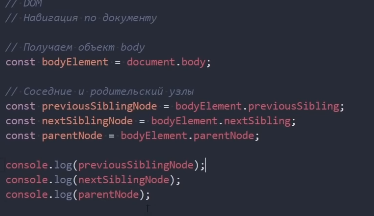




previousSiblingNode – предудщий обект на том же уровне(имено узел)

NextSiblingNode – следущий

parentNode – родитель



Для работы с тегами

children – колекция только с тегами

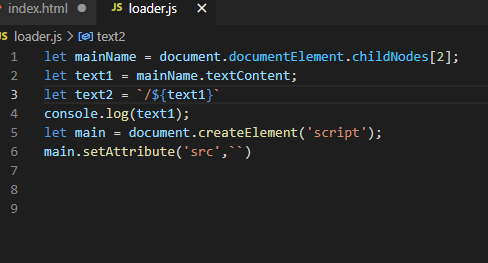
firstElementChild

lastElementChild



previousElementSibling

nextElementSibling



Для произвольного обекта

querySelectorAll(‘селктор(.класс, тег, id? По атрибуту[align=””])‘)

Статическая коллекция



3 элемент списка(при помощи ключа в

Все элементы(перебор)



querySelector(селктор) – первый обект, а не коллекция

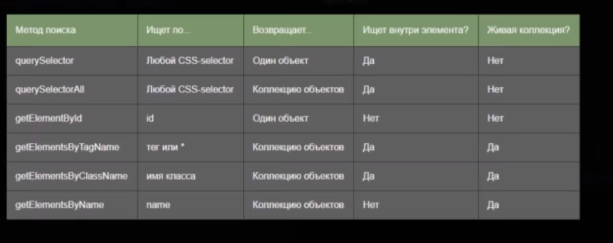
document.getElementByID(именно по id) стат

getElemetnsByTagName(“тег”) по имени тега

getElemetnsByClassName(“без точки”)

Живая коллекция – будет изменяться

Статическая – будет такая только на момент вызова



closest



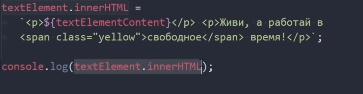
Ищет родителя по классу и тд

Менять

innerHTML(показывает содержимое вместе с тегами, дописываем) += перезаписывает

const a = document.querySelector(‘.auto’)

a.innerHTML = “ФОЛЬЦваген”



outerHTML – полностью меняет эллемнт(такде получаем сам обект) только не поддерживется

a. outerHTML = ‘<p> rtrturtueryery </p>’

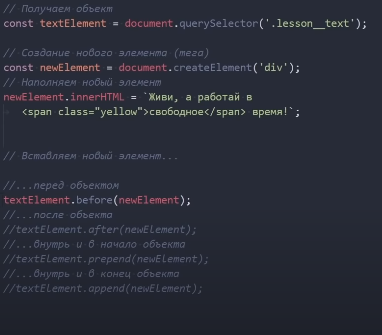
textContent - именно тектс меняет(напишем тег вссе равное будет текст)

a.textContent = ‘’

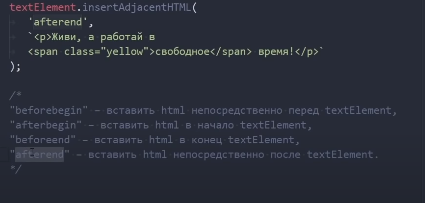
.data для комментариев

ЗОЗДАНИЕ

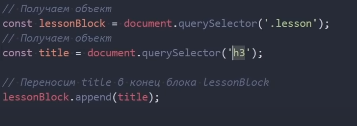
Document.createElement(‘div’ имя тега)



insertAdjacentHTML c двумя параметрами, более мощная штука создания новых элементов

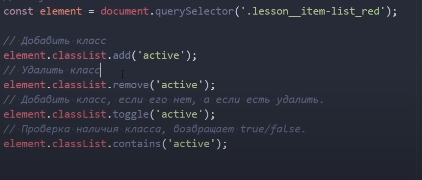


Append – перенос обекта



Remove -удаление

ДОБАВЛЕНИЕ КЛАСС



Управляем стилями

a.style.color = “red”

a.style.backgroudColor = “#aaa”

a.style.color = “” для сброса

cssText = “ – он полностью переписывает свойства

color:

width:

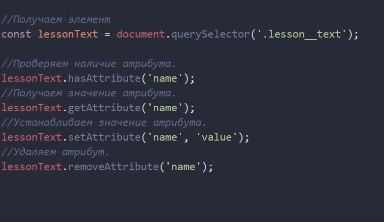
heith

”

getComputedStyle(“атрибут”) точно получить свойство

с атрибутами тож самое

если все вывод хотим то просто пишем console.dir(a)



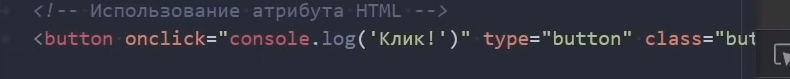
Назначение обработчиков событий. В чем разница каждого способа.

Событие – это сигнал от браузера о том, что что-то произошло. Все DOM-узлы подают такие сигналы, клик мыши, закгрузка страницы, и все мы их можем отлавливать

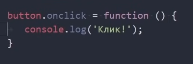
Это делаеться при помощи обработчика событий, т е функция которая сработает когда произойдет событие, благодаря этому мы можем реагировать на действия пользователя

Сопособы:

Использование атрибутов html



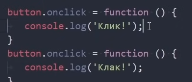
Создание внешних функций(через обращение к элементы путем DOM)



Такой же способ

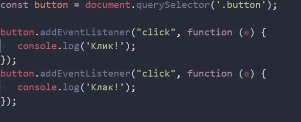


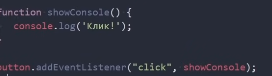
Из минусов в том что нельзя сразу несколько событий



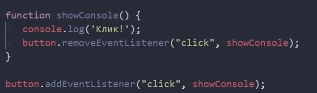
НО сразу несколько? Или некоторые события которые только при помощи его можно отловить

addEventListener(событие, имя функции, опции)

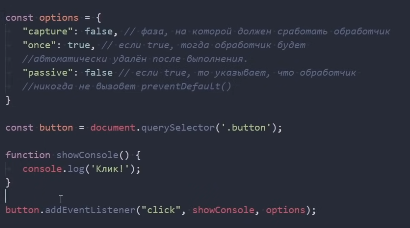




Если мы хотим чтобы выполнилась один раз



Свойства 



Capture

Once true – обработчик сразу будет после выполнения удален, как removeEventListner

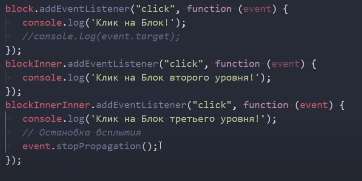
Capture : true с погружением, то есть выполниться он, затем выполниться нижний, потом всплывет, и выполниться самый верхний

False ,в момент погружения ничего не отлавливаеться

Всплытие и погружение событий(фокус не всплывает)

Если у нас идет вложенность элементов в HTML и мы каждому элементу, каждого уровня, назначаем событие, то при выполнении события на внутреннем элементе, он нам будет выдавать событие на нем на его родителе, и выше, произошло всплытие

Чтобы это убрать то мы делаем event.stopPropagation()



Погружение, то есть, сначало событие идут сверху вниз, с 1 блока до самого глубокого, и можем это отлавливать при помощи свойства capture

preventDefault – отментяет дефолтные действия браузера

// Находим на странице кнопку и попап

const button = document.querySelector('.click-button');

const popup = document.querySelector('.content');

// Навешиваем на кнопку обработчик клика button.onclick = function (evt) { // Отменяем переход по ссылке

evt.preventDefault();

// Добавляем попапу класс show, делая его видимым popup.classList.add('show');

};

**stopImmediatePropagation() – предотвращает всплытие**

Делегирование – механизм, который позволяет вместо того чтобы каждому элементу назначать свой обработчик)если они в одном блоке), мы просто создаем обработчик для их родителя

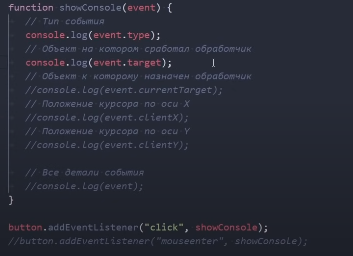


Closest – возвращает ближайшего родителя

Объект event и его свойства.

Чтобы хорошо обработать событие, недостаточно знать о том, что это – «клик» или «нажатие клавиши». Могут понадобиться детали: координаты курсора, введённый символ и другие, в зависимости от события.

Когда происходит, собыитие, браузер создает обект события и его можно использовать как аргумент в обработчике событий



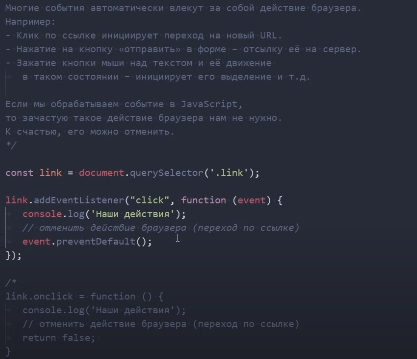
event.type -указывает тип события

event.target - показывает какой элемент, с которым мы что-то сделали

currentTarhet – конкретны обект к которому назнаен обработчик

clientX/Y – положение курсовра, оносительна окна бразуера

Дефолтные браузера



События мыши

Простые

Mousedown – кнопка мыши нажата, up – отпущена

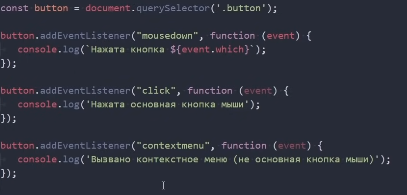
Mouseover – курсор над элементом, out- ушел с элемента

Mousemove – движение над обектом(при движениия будет генериться событие)

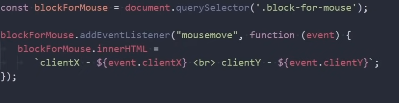
Комлексные

Click – когда mousedown + up

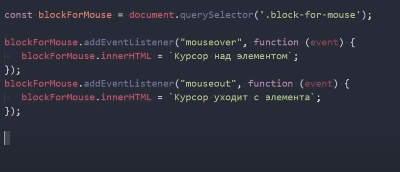
Dbclick – двойной клик



Эффект паралакса



При наведении курсора



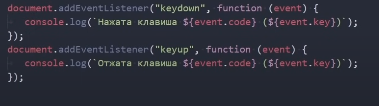
Собитии клавиатуры

Keydown – при нажатии клавиши

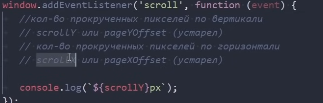
Keyup – при отпускании

Event.code и event.key(он еще и регистры считает, и также изменение языка) покажет какую клавишу вы нажали

Event.repeat – ТУПО ЗАЖАЛИ КЛАВИШУ



СОБЫТИЕ ПРИ СКРОЛЕ



СОБЫТИЕ ПРИ ЗАГРУЗКЕ СТРАНИЦЫ

Load – при полной загрузке страницы(для сложных структур)

DOMContentLoader – при конкретном обекте document

