**Тема 2. Структуры данных**

1. Методы примитивов.
2. Числа.
3. Строки.
4. Массивы.
5. Методы массивов.
6. Объекты.
7. Тип данных Symbol.
8. Set, Map, WeakSet и WeakMap.
9. Деструктурирующее присваивание
10. Дата и время.
11. Формат JSON.
12. Intl: интернационализация в JavaScript.
13. Регулярные выражения.

Содержание данной темы включает материалы, доступные по адресу https://learn.javascript.ru.

1. **Методы примитивов.**

JavaScript позволяет работать с примитивными типами данных – строками, числами, и т.д., так, как если бы они были объектами. У них есть и методы. Рассмотрим ключевые различия между примитивами и объектами.

Примитив – это значение «примитивного» типа. Есть 6 примитивных типов: string, number, boolean, symbol, null и undefined.

Объект может хранить множество значений как свойства. Объявляется при помощи фигурных скобок {}, например: {name: "Рома", age: 30}. В JavaScript есть и другие виды объектов: например, функции тоже являются объектами.

Одна из лучших особенностей объектов – это то, что можно хранить функцию как одно из свойств объекта:

let roma = {

name: "Рома",

sayHi: function() {

alert("Привет, дружище!");

}

};

roma.sayHi(); // Привет, дружище!

Здесь создан объект roma с методом sayHi.

Существует множество встроенных объектов. Например, те, которые работают с датами, ошибками, HTML-элементами и т.д. Они имеют различные свойства и методы. Поэтому объекты «тяжелее» примитивов, они нуждаются в дополнительных ресурсах для поддержания внутренней структуры.

Язык JavaScript позволяет осуществлять доступ к методам и свойствам строк, чисел, булевых значений и символов. Чтобы это работало, при таком доступе создаётся специальный «объект-обёртка», который предоставляет нужный функционал, а после удаляется. Каждый примитив имеет свой собственный «объект-обёртку», которые называются: String, Number, Boolean и Symbol. Таким образом, они имеют разный набор методов. К примеру, существует метод [str.toUpperCase()](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/String/toUpperCase), который возвращает cтроку в верхнем регистре:

let str = "Привет";

alert( str.toUpperCase() );

Рассмотрим, как это работет:

1. Строка str – примитив. В момент обращения к его свойству, создаётся специальный объект, который содержит значение строки и метод toUpperCase().
2. Этот метод вызывается и возвращает новую строку (выводится в alert).
3. Специальный объект удаляется, оставляя только примитив str.

Таким образом, примитивы могут предоставлять методы, и в то же время оставаться «лёгкими». Движок JavaScript сильно оптимизирует этот процесс. Он даже может пропустить создание специального объекта.

Число имеет собственный набор методов. Например, [toFixed(n)](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/Number/toFixed) округляет число до *n* знаков после запятой.

let n = 1.23456;

alert( n.toFixed(2) ); // 1.23

В JavaScript также можно явное создавать «объекты-обёртки» с помощью конструкторов String/Number/Boolean (new Number(1) или new Boolean(false)), но это очень не рекомендуется, так как иногда приводит к необжиданным результатам:

alert( typeof 0 ); // "число"

alert( typeof new Number(0) ); // "object"

Объекты в if всегда дают true, поэтому в примере ниже будет показан alert:

let zero = new Number(0);

if (zero) {

// zero возвращает "true", так как является объектом

alert( "zero имеет «истинное» значение?!?" );

}

С другой стороны, использование функций String/Number/Boolean (без оператора new) полезно, так как они превращают значение в соответствующий примитивный тип: в строку, в число, в булевой тип. Например:

let num = Number("123"); // превращает строку в число

Особенные примитивы null and undefined являются исключениями – у них нет соответствующих «объектов-обёрток» и они не имеют никаких методов.

1. **Числа.**

Все числа в JavaScript хранятся в 64-битном формате [IEEE-754](http://en.wikipedia.org/wiki/IEEE_754-1985), который также называют «числа с плавающей точкой двойной точности» (double precision floating point numbers).

[**Способы записи числа**](https://learn.javascript.ru/number#sposoby-zapisi-chisla)

Допустим надо записать число 1 миллиард:

let billion = 1000000000;

Но в реальной жизни обычно множество нулей опускается, а укороченная запись может выглядеть как "1млрд" или "7.3млрд".  Такой принцип работает для всех больших чисел. В JavaScript можно использовать букву "e", чтобы укоротить запись числа. Она добавляется к числу и заменяет указанное количество нулей:

let billion = 1e9; // 1 миллиард

alert( 7.3e9 ); // 7.3 миллиардов

Другими словами, "e" производит операцию умножения числа на 1 с указанным количеством нулей.

1e3 = 1 \* 1000

1.23e6 = 1.23 \* 1000000

Допустим необходимо записать что-нибудь очень маленькое: 1 микросекунду (одна миллионная секунды):

let ms = 0.000001;

Записать микросекунду в укороченном виде также можно с помощью "e".

let ms = 1e-6; // шесть нулей, слева от 1

Т.е., отрицательное число после "e" подразумевает деление на 1 с указанным количеством нулей:

// -3 делится на 1 с 3 нулями

1e-3 = 1 / 1000 (=0.001)

// -6 делится на 1 с 6 нулями

1.23e-6 = 1.23 / 1000000 (=0.00000123)

[**Шестнадцатеричные, двоичные и восьмеричные числа**](https://learn.javascript.ru/number#shestnadtsaterichnye-dvoichnye-i-vosmerichnye-chisla)

[Шестнадцатеричные](https://ru.wikipedia.org/wiki/%D0%A8%D0%B5%D1%81%D1%82%D0%BD%D0%B0%D0%B4%D1%86%D0%B0%D1%82%D0%B5%D1%80%D0%B8%D1%87%D0%BD%D0%B0%D1%8F_%D1%81%D0%B8%D1%81%D1%82%D0%B5%D0%BC%D0%B0_%D1%81%D1%87%D0%B8%D1%81%D0%BB%D0%B5%D0%BD%D0%B8%D1%8F) числа широко используются в JavaScript для представления цветов, кодировки символов и многое другое. Естественно, есть короткий стиль записи: 0x, после которого указывается число. Например:

alert( 0xff ); // 255

alert( 0xFF ); // 255 (регистр не имеет значения)

Не так часто используются двоичные и восьмеричные числа, но они также поддерживаются 0b для двоичных и 0o для восьмеричных:

let a = 0b11111111; // бинарная форма записи числа 255

let b = 0o377; // восьмеричная форма записи числа 255

alert( a == b ); // true

Есть только 3 системы счисления с такой поддержкой. Для других систем счисления рекомендуется использовать функцию parseInt (рассмотрим позже).

[**toString(base)**](https://learn.javascript.ru/number#tostring-base)

Метод num.toString(base) возвращает строковое представление числа *num* в системе счисления base. Например:

let num = 255;

alert( num.toString(16) ); // ff

alert( num.toString(2) ); // 11111111

Значение *base* может варьироваться от 2 до 36 (по умолчанию 10):

* base = 16 – для шестнадцатеричного представления цвета, кодировки символов и т.д., цифры могут быть 0..9 или A..F.
* base = 2 – обычно используется для отладки побитовых операций, цифры 0 или 1.
* base = 36 – максимальное основание, цифры могут быть 0..9 или A..Z. То есть, используется весь латинский алфавит для представления числа.

Две точки в 123456..toString(36) используется, если надо вызвать метод toString непосредственно на числе. Если поставить одну точку: 123456.toString(36), тогда это будет ошибкой, поскольку синтаксис JavaScript предполагает, что после первой точки начинается десятичная часть. А если поставить две точки, то JavaScript понимает, что десятичная часть отсутствует, и начинается метод. Также можно записать как (123456).toString(36).

[**Округление**](https://learn.javascript.ru/number#okruglenie)

Одна из часто используемых операций при работе с числами – это округление. В JavaScript есть несколько встроенных функций для работы с округлением:

* Math.floor – округление в меньшую сторону: 3.1 становится 3, а -1.1 – -2.
* Math.ceil – округление в большую сторону: 3.1 становится 4, а -1.1 – -1.
* Math.round – округление до ближайшего целого: 3.1 становится 3, 3.6 – 4, а -1.1 – -1.
* Math.trunc (не поддерживается в Internet Explorer) – производит удаление дробной части без округления: 3.1 становится 3, а -1.1 – -1.

Таблица с различиями между функциями округления:

|  | **Math.floor** | **Math.ceil** | **Math.round** | **Math.trunc** |
| --- | --- | --- | --- | --- |
| 3.1 | 3 | 4 | 3 | 3 |
| 3.6 | 3 | 4 | 4 | 3 |
| -1.1 | -2 | -1 | -1 | -1 |
| -1.6 | -2 | -1 | -2 | -1 |

Эти функции охватывают все возможные способы обработки десятичной части. Если надо округлить число до *n*-ого количества цифр в дробной части, то это можно сделать одним из следующих способов:

1. Умножить и разделить.

Например, чтобы округлить число до второго знака после запятой, можно умножить число на 100, вызвать функцию округления и разделить обратно.

let num = 1.23456;

alert( Math.floor(num \* 100) / 100 ); // 1.23456 -> 123.456 -> 123 -> 1.23

1. Метод [toFixed(n)](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/Number/toFixed) округляет число до *n* знаков после запятой и возвращает строковое представление результата.

Если десятичная часть короче, чем необходима, будут добавлены нули в конец строки:

let num = 12.34;

alert( num.toFixed(5) ); // "12.34000", добавлены нули, чтобы получить 5 знаков после запятой

Можно преобразовать полученное значение в число, используя унарный оператор + или Number(), пример с унарным оператором: +num.toFixed(5).

[**Неточные вычисления**](https://learn.javascript.ru/number#netochnye-vychisleniya)

Внутри JavaScript число представлено в виде 64-битного формата [IEEE-754](https://ru.wikipedia.org/wiki/IEEE_754-1985). Для хранения числа используется 64 бита: 52 из них используется для хранения цифр, 11 из них для хранения положения десятичной точки (если число целое, то хранится 0), и один бит отведён на хранения знака. Если число слишком большое, оно переполнит 64-битное хранилище, JavaScript вернёт бесконечность:

alert( 1e500 ); // Infinity

Наиболее часто встречающаяся ошибка при работе с числами в JavaScript – это потеря точности. Например, сумма 0.1 и 0.2 не равна 0.3:

alert( 0.1 + 0.2 == 0.3 ); // false

alert( 0.1 + 0.2 ); // 0.30000000000000004

Так происходит потому, что число хранится в памяти в бинарной форме, как последовательность бит – единиц и нулей. Но дроби, такие как 0.1, 0.2, которые выглядят довольно просто в десятичной системе счисления, на самом деле являются бесконечной дробью в двоичной форме.

Число 0.1 – это единица, делённая на десять – 1/10, одна десятая. В десятичной системе счисления такие числа легко представимы, по сравнению с одной третьей: 1/3, которая становится бесконечной дробью 0.33333(3). Деление на 10 хорошо работает в десятичной системе, но деление на 3 – нет. По той же причине и в двоичной системе счисления, деление на 2 обязательно сработает, а 1/10 становится бесконечной дробью.

В JavaScript нет возможности для хранения точных значений 0.1 или 0.2, используя двоичную систему, точно также, как нет возможности хранить одну третью в десятичной системе счисления. Числовой формат IEEE-754 решает эту проблему путём округления до ближайшего возможного числа. Правила округления обычно не позволяют увидеть эту нехначительную потерю точности, но она существует:

alert( 0.1.toFixed(20) ); // 0.10000000000000000555

Когда складываются 2 числа, их неточности тоже суммируются, поэтому 0.1 + 0.2 – это не совсем 0.3.

Наиболее надёжный способ обойти эту проблему – это округлить результат используя метод [toFixed(n)](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/Number/toFixed):

let sum = 0.1 + 0.2;

alert( sum.toFixed(2) ); // 0.30

Метод toFixed всегда возвращает строку. Это гарантирует, что результат будет с заданным количеством цифр в десятичной части. В других случаях можно использовать унарный оператор +, чтобы преобразовать строку в число:

let sum = 0.1 + 0.2;

alert( +sum.toFixed(2) ); // 0.3

Также можно умножить число на 100 (или на большее), чтобы привести его к целому, выполнить математические действия, а после разделить обратно. Суммируя целые числа, мы уменьшаем погрешность, но она все равно появляется при финальном делении:

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

alert( (0.28 \* 100 + 0.14 \* 100) / 100); // 0.4200000000000001

Таким образом, метод умножения/деления уменьшает погрешность, но полностью её не решает.

Еще одно следствие внутреннего представления чисел – наличие двух нулей: 0 и -0. Это возможно потому, что знак представлен отдельным битом, так что, любое число может быть положительным и отрицательным, включая нуль. В большинстве случаев это поведение незаметно, так как операторы в JavaScript воспринимают их одинаковыми.

[**Проверка: isFinite и isNaN**](https://learn.javascript.ru/number#proverka-isfinite-i-isnan)

Специальные числовые значения Infinity ( -Infinity) и NaN  принадлежат типу number, но они не являются «обычными» числами, поэтому есть функции для их проверки:

* isNaN(value) преобразует значение в число и проверяет является ли оно NaN:

alert( isNaN(NaN) ); // true

alert( isNaN("str") ); // true

Нельзя просто сравнить === NaN, так как значение NaN уникально тем, что оно не является равным ни чему другому, даже самому себе:

alert( NaN === NaN ); // false

* isFinite(value) преобразует аргумент в число и возвращает true, если оно является обычным числом, т.е. не NaN/Infinity/-Infinity:

alert( isFinite("15") ); // true

alert( isFinite("str") ); // false

alert( isFinite(Infinity) ); // false

Иногда isFinite используется для проверки, содержится ли в строке число:

let num = +prompt("Enter a number", '');

alert( isFinite(num) );

Не стоит забывать, что пустая строка интерпретируется как 0 во всех числовых функциях, включая isFinite.

**Сравнение Object.is**

Существует специальный метод [Object.is](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/Object/is), который сравнивает значения примерно как ===, но более надёжен в двух особых ситуациях:

1. Сравнивает NaN: Object.is(NaN, NaN) === true.
2. Значения 0 и -0 разные: Object.is(0, -0) === false, это редко используется, но технически эти значения разные.

Во всех других случаях Object.is(a, b) идентичен a === b.

Этот способ сравнения часто используется в спецификации JavaScript. Когда алгоритму необходимо сравнить 2 значения на предмет точного совпадения, он использует Object.is.

[**parseInt и parseFloat**](https://learn.javascript.ru/number#parseint-i-parsefloat)

Для явного преобразования к числу можно использовать + или Number(). Если строка не является в точности числом, то результат будет NaN:

alert( +"100px" ); // NaN

Единственное исключение – это пробелы в начале строки и в конце, они игнорируются. На практике часто встречаются значения, у которых есть единица измерения, например, "100px" или "12pt" в CSS, во множестве стран символ валюты записывается после номинала "19€".

Чтобы получить числовое значение из таких строк есть методы parseInt и parseFloat. Они считывают число из строки. Если в процессе чтения возникает ошибка, они возвращают полученное до ошибки число. Функция parseInt возвращает целое число, а parseFloat возвращает число с плавающей точкой:

alert( parseInt('100px') ); // 100

alert( parseFloat('12.5em') ); // 12.5

alert( parseInt('12.3') ); // 12

alert( parseFloat('12.3.4') ); // 12.3

Функции parseInt/parseFloat вернут NaN, если не смогли прочитать ни одну цифру:

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

Функция parseInt(str, radix) имеет необязательный второй параметр. Он определяет систему счисления, таким образом parseInt может также читать строки с шестнадцатеричными числами, двоичными числами и т.д.:

alert( parseInt('0xff', 16) ); // 255

alert( parseInt('ff', 16) ); // 255

alert( parseInt('2n9c', 36) ); // 123456

[**Другие математические функции**](https://learn.javascript.ru/number#drugie-matematicheskie-funktsii)

В JavaScript встроен объект [Math](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/Math), который содержит различные математические функции и константы. Несколько примеров:

* Math.random() – возвращает псевдослучайное число в диапазоне [0, 1)

alert( Math.random() ); // 0.1234567894322

alert( Math.random() ); // 0.5435252343232

alert( Math.random() ); // ...

* Math.max(a, b, c...) / Math.min(a, b, c...) – возвращает наибольшее/наименьшее число из перечисленных аргументов.

alert( Math.max(3, 5, -10, 0, 1) ); // 5

alert( Math.min(1, 2) ); // 1

* Math.pow(n, power) – возвращает число n, возведённое в степень power

alert( Math.pow(2, 10) ); // 2 в степени 10 = 1024

В объекте Math есть множество других функций и констант, включая тригонометрические функции, с которыми подробнее можно ознакомиться в документации по объекту [Math](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/Math).

1. **Строки.**

В JavaScript любые текстовые данные являются строками. Не существует отдельного типа «символ», который есть в ряде других языков. Внутренний формат для строк – всегда [UTF-16](https://ru.wikipedia.org/wiki/UTF-16), вне зависимости от кодировки страницы.

[**Кавычки**](https://learn.javascript.ru/string#kavychki)

В JavaScript есть разные типы кавычек. Строку можно создать с помощью одинарных, двойных либо обратных кавычек:

let single = 'single-quoted';

let double = "double-quoted";

let backticks = `backticks`;

Одинарные и двойные кавычки работают, по сути, одинаково, а если использовать обратные кавычки, то в такую строку можно вставлять произвольные выражения, обернув их в ${…}:

function sum(a, b) {

return a + b;

}

alert(`1 + 2 = ${sum(1, 2)}.`); // 1 + 2 = 3.

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

let guestList = `Guests:

\* John

\* Pete

\* Mary

`;

alert(guestList);

Если использовать точно так же одинарные или двойные кавычки, то будет ошибка:

let guestList = "Guests: // Error: Unexpected token ILLEGAL

\* John";

Обратные кавычки также позволяют задавать «шаблонную функцию» перед первой обратной кавычкой. Используемый синтаксис: func`string`. Автоматически вызываемая функция func получает строку и встроенные в неё выражения и может их обработать. Подробнее об этом можно прочитать в [документации](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Template_literals#Tagged_template_literals). Если перед строкой есть выражение, то шаблонная строка называется «теговым шаблоном». Это позволяет использовать свою шаблонизацию для строк, но на практике теговые шаблоны применяются редко.

[**Спецсимволы**](https://learn.javascript.ru/string#spetssimvoly)

Многострочные строки также можно создавать с помощью одинарных и двойных кавычек, используя так называемый «символ перевода строки», который записывается как \n:

let guestList = "Guests:\n \* John\n \* Pete\n \* Mary";

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

В частности, эти две строки эквивалентны, просто записаны по-разному:

let str1 = "Hello\nWorld";

let str2 = `Hello

World`;

alert(str1 == str2); // true

Есть и другие, реже используемые спецсимволы:

| **Символ** | **Описание** |
| --- | --- |
| \n | Перевод строки |
| \r | Возврат каретки: самостоятельно не используется. В текстовых файлах Windows для перевода строки используется комбинация символов \r\n. |
| \', \" | Кавычки |
| \\ | Обратный слеш |
| \t | Знак табуляции |
| \b, \f, \v | Backspace, Form Feed и Vertical Tab – оставлены для обратной совместимости, сейчас не используются. |
| \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. |
| \u{X…XXXXXX} (от 1 до 6 шестнадцатеричных цифр) | Символ в кодировке UTF-32 с шестнадцатеричным кодом от U+0000 до U+10FFFF. Некоторые редкие символы кодируются двумя 16-битными словами и занимают 4 байта. Так можно вставлять символы с длинным кодом. |

Примеры с Юникодом:

// ©

alert( "\u00A9" );

// 佫, редкий китайский иероглиф

alert( "\u{20331}" );

// 😍

alert( "\u{1F60D}" );

Все спецсимволы начинаются с обратного слеша, \ – так называемого «символа экранирования». Он также используется, если необходимо вставить в строку кавычку. Например:

alert( 'I\'m the Walrus!' ); // I'm the Walrus!

Здесь перед входящей в строку кавычкой необходимо добавить обратный слеш  \, иначе она бы обозначала окончание строки. Требование экранировать относится только к таким же кавычкам, как те, в которые заключена строка. Можно использовать для этой строки двойные или обратные кавычки:

alert( `I'm the Walrus!` ); // I'm the Walrus!

Заметим, что обратный слеш \ служит лишь для корректного прочтения строки интерпретатором, но он не записывается в строку после её прочтения. Когда строка сохраняется в оперативную память, в неё не добавляется символ \. Это можно увидеть в выводах alert в примерах выше. Но если надо добавить в строку сам обратный слеш \, то это можно сделать, добавив перед ним ещё один обратный слеш:

alert( `The backslash: \\` ); // The backslash: \

[**Длина строки**](https://learn.javascript.ru/string#dlina-stroki)

Свойство length содержит длину строки:

alert( `My\n`.length ); // 3

Обратите внимание, \n – это один спецсимвол, поэтому длина строки 3.

Так как str.length – это числовое свойство, а не функция, добавлять скобки не нужно.

[**Доступ к символам**](https://learn.javascript.ru/string#dostup-k-simvolam)

Получить символ, который занимает позицию pos, можно с помощью квадратных скобок: [pos]. Также можно использовать метод charAt: [str.charAt(pos)](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/String/charAt). Первый символ занимает нулевую позицию:

let str = `Hello`;

// первый символ

alert( str[0] ); // H

alert( str.charAt(0) ); // H

// последний символ

alert( str[str.length - 1] ); // o

Квадратные скобки – современный способ получить символ, в то время как charAt существует в основном по историческим причинам. Разница только в том, что если символ с такой позицией отсутствует, тогда [] вернёт undefined, а charAt – пустую строку:

let str = `Hello`;

alert( str[1000] ); // undefined

alert( str.charAt(1000) ); // ''

Также можно перебрать строку посимвольно, используя for..of:

for (let char of "Hello") {

alert(char); // H,e,l,l,o

}

Содержимое строки в JavaScript нельзя изменить. Нельзя взять символ посередине и заменить его. Как только строка создана – она такая навсегда:

let str = 'Hi';

str[0] = 'h'; // ошибка

alert( str[0] ); // не работает

Можно создать новую строку и записать её в ту же самую переменную вместо старой. Например:

let str = 'Hi';

str = 'h' + str[1]; // заменяем строку

alert( str ); // hi

[**Изменение регистра**](https://learn.javascript.ru/string#izmenenie-registra)

Методы [toLowerCase()](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/String/toLowerCase) и [toUpperCase()](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/String/toUpperCase) меняют регистр символов:

alert( 'Interface'.toUpperCase() ); // INTERFACE

alert( 'Interface'.toLowerCase() ); // interface

Если необходимо перевести в нижний регистр какой-то конкретный символ:

alert( 'Interface'[0].toLowerCase() ); // 'i'

[**Поиск подстроки**](https://learn.javascript.ru/string#poisk-podstroki)

Существует несколько способов поиска подстроки.

Первый метод – [str.indexOf(substr, pos)](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/String/indexOf). Он ищет подстроку substr в строке str, начиная с позиции pos, и возвращает позицию, на которой располагается совпадение, либо -1 при отсутствии совпадений. Например:

let str = 'Widget with id';

alert( str.indexOf('Widget') ); // 0, т.к. подстрока 'Widget' найдена в начале

alert( str.indexOf('widget') ); // -1, поиск чувствителен к регистру

alert( str.indexOf("id") ); // 1

Необязательный второй аргумент позволяет начать поиск с определённой позиции. Например, первое вхождение "id" – на позиции 1. Для того, чтобы найти следующее, необходимо начать поиск с позиции 2:

let str = 'Widget with id';

alert( str.indexOf('id', 2) ) // 12

Чтобы найти все вхождения подстроки, нужно запустить indexOf в цикле. Каждый раз, получив очередную позицию, начинаем новый поиск со следующей:

let str = 'Ослик Иа-Иа посмотрел на виадук';

let target = 'Иа'; // цель поиска

let pos = 0;

while (true) {

let foundPos = str.indexOf(target, pos);

if (foundPos == -1) break;

alert( `Найдено тут: ${foundPos}` );

pos = foundPos + 1; // продолжаем со следующей позиции

}

Также есть похожий метод [str.lastIndexOf(substr, position)](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/String/lastIndexOf), который ищет с конца строки к её началу. Он используется тогда, когда нужно получить самое последнее вхождение: перед концом строки или начинающееся до (включительно) определённой позиции.

**Методы** [**includes, startsWith, endsWith**](https://learn.javascript.ru/string#includes-startswith-endswith)

Более современный метод [str.includes(substr, pos)](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/String/includes) возвращает true, если в строке str есть подстрока substr, либо false, если нет. Стоит его использовать, если необходимо проверить, есть ли совпадение, но позиция не нужна:

alert( "Widget with id".includes("Widget") ); // true

alert( "Hello".includes("Bye") ); // false

Необязательный второй аргумент str.includes позволяет начать поиск с определённой позиции:

alert( "Midget".includes("id") ); // true

alert( "Midget".includes("id", 3) ); // false, поиск начат с позиции 3

Методы [str.startsWith](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/String/startsWith) и [str.endsWith](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/String/endsWith) проверяют, соответственно, начинается ли и заканчивается ли строка определённой строкой:

alert( "Widget".startsWith("Wid") ); // true, "Wid" – начало "Widget"

alert( "Widget".endsWith("get") ); // true, "get" – окончание "Widget"

[**Получение подстроки**](https://learn.javascript.ru/string#poluchenie-podstroki)

В JavaScript есть 3 метода для получения подстроки: substring, substr и slice.

* str.slice(start [, end]) – возвращает часть строки от start до (не включая) end.

Например:

let str = "stringify";

// 'strin', символы от 0 до 5 (не включая 5)

alert( str.slice(0, 5) );

// 's', от 0 до 1, не включая 1

alert( str.slice(0, 1) );

Если аргумент end отсутствует, slice возвращает символы до конца строки:

let str = "stringify";

alert( str.slice(2) ); // ringify, с позиции 2 и до конца

Также для start/end можно задавать отрицательные значения. Это означает, что позиция определена как заданное количество символов с конца строки:

let str = "stringify";

// начинаем с позиции 4 справа, заканчиваем на позиции 1 справа

alert( str.slice(-4, -1) ); // gif

* str.substring(start [, end]) – возвращает часть строки между start и end.

Это почти то же, что и slice, но можно задавать start больше end. Например:

let str = "stringify";

alert( str.substring(2, 6) ); // "ring"

alert( str.substring(6, 2) ); // "ring"

alert( str.slice(2, 6) ); // "ring"

alert( str.slice(6, 2) ); // ""

Отрицательные значения substring, в отличие от slice, не поддерживает, они интерпретируются как 0.

* str.substr(start [, length]) – возвращает часть строки от start длины length.

В противоположность предыдущим методам, этот позволяет указать длину вместо конечной позиции:

let str = "stringify";

alert( str.substr(2, 4) ); // ring

Значение первого аргумента может быть отрицательным, тогда позиция определяется с конца:

let str = "stringify";

alert( str.substr(-4, 2) ); // gi

Сравнительная таблица рассмотренных выше методов:

| **Метод** | **Диапазон** | **Отрицательные значения** |
| --- | --- | --- |
| slice(start, end) | от start до end (не включая end) | можно передавать отрицательные значения |
| substring(start, end) | между start и end | отрицательные значения равнозначны 0 |
| substr(start, length) | length символов, начиная от start | значение start может быть отрицательным |

Метод slice более гибок, он поддерживает отрицательные аргументы, и его короче писать, поэтому его используют наиболее часто.

[**Сравнение строк**](https://learn.javascript.ru/string#sravnenie-strok)

Строки сравниваются посимвольно в алфавитном порядке. Тем не менее, есть некоторые нюансы:

1. Строчные буквы больше заглавных:

alert( 'a' > 'Z' ); // true

1. Буквы, имеющие диакритические знаки, идут «не по порядку»:

alert( 'Österreich' > 'Zealand' ); // true

Это может привести к своеобразным результатам при сортировке названий стран: ожидается, что Zealand будет после Österreich в списке. Чтобы разобраться, что происходит на самом деле, необходимо ознакомимся с внутренним представлением строк в JavaScript.

Строки кодируются в [UTF-16](https://ru.wikipedia.org/wiki/UTF-16). Таким образом, у любого символа есть соответствующий код. Есть специальные методы, позволяющие получить символ по его коду и наоборот:

* str.codePointAt(pos) – возвращает код для символа, находящегося на позиции pos. Одна и та же буква в нижнем и верхнем регистре будет иметь разные коды:

alert( "z".codePointAt(0) ); // 122

alert( "Z".codePointAt(0) ); // 90

* String.fromCodePoint(code) – создаёт символ по его коду code

alert( String.fromCodePoint(90) ); // Z

Также можно добавлять юникодные символы по их кодам, используя \u с шестнадцатеричным кодом символа:

// 90 – 5a в шестнадцатеричной системе счисления

alert( '\u005a' ); // Z

Можно сформировать строку, содержащую символы с кодами от 65 до 220 – это латиница и ещё некоторые распространённые символы:

let str = '';

for (let i = 65; i <= 220; i++) {

str += String.fromCodePoint(i);

}

alert( str );

// ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^\_`abcdefghijklmnopqrstuvwxyz{|}~

// ¡¢£¤¥¦§¨©ª«¬­®¯°±²³´µ¶·¸¹º»¼½¾¿ÀÁÂÃÄÅÆÇÈÉÊËÌÍÎÏÐÑÒÓÔÕÖ×ØÙÚÛÜ

Как видино, сначала идут заглавные буквы, затем несколько спецсимволов, затем строчные и Ö ближе к концу вывода. Это объясняет, почему a > Z. Символы сравниваются по их кодам. Больший код – больший символ. Код a (97) больше кода Z (90):

* Все строчные буквы идут после заглавных, так как их коды больше.
* Некоторые буквы, такие как Ö, находятся вне основного алфавита. У этой буквы код больше, чем у любой буквы от a до z.

[**Правильное сравнение**](https://learn.javascript.ru/string#pravilnoe-sravnenie)

«Правильный» алгоритм сравнения строк сложнее, чем может показаться, так как разные языки используют разные алфавиты. Поэтому браузеру нужно знать, какой язык использовать для сравнения. Все современные браузеры (для IE10 нужна дополнительная библиотека [Intl.JS](https://github.com/andyearnshaw/Intl.js/)) поддерживают стандарт [ECMA 402](http://www.ecma-international.org/ecma-402/1.0/ECMA-402.pdf), обеспечивающий правильное сравние строк на разных языках с учётом их правил. Для этого есть соответствующий метод.

Вызов [str.localeCompare(str2)](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/String/localeCompare) возвращает число, которое показывает, какая строка больше в соответствии с правилами языка:

* Отрицательное число, если str меньше str2.
* Положительное число, если str больше str2.
* 0, если строки равны.

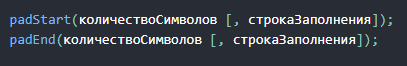
Например:

alert( 'Österreich'.localeCompare('Zealand') ); // -1

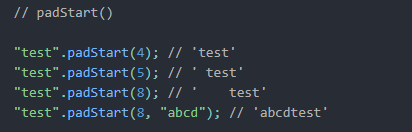
У этого метода есть два дополнительных аргумента, которые указаны в [документации](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/String/localeCompare). Первый позволяет указать язык (по умолчанию берётся из окружения) – от него зависит порядок букв. Второй – определить дополнительные правила, такие как чувствительность к регистру, а также следует ли учитывать различия между "a" и "á".

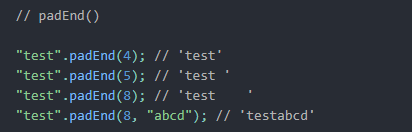
**Заполнение строки**

Целью заполнения строки является добавление символов в строку, чтобы она достигла определенной длины. ES2017 представляет два метода для строк padStart() и padEnd() - которые позволяют добавлять либо пустую строку, либо любую другую строку к началу или, концу исходной строки. Это оказывается удобным, если нужно выровнять текст, например, при выводе в консоль.



Пример использования:





1. **Массивы.**

Для хранения упорядоченных коллекций существует особая структура данных, которая называется массив, Array.

[**Объявление**](https://learn.javascript.ru/array#obyavlenie)

Существует два варианта синтаксиса для создания пустого массива:

let arr = new Array();

let arr = [];

Практически всегда используется второй вариант синтаксиса. В скобках можно указать начальные значения элементов:

let fruits = ["Яблоко", "Апельсин", "Слива"];

Элементы массива нумеруются, начиная с нуля. Можно получить элемент, указав его номер в квадратных скобках:

let fruits = ["Яблоко", "Апельсин", "Слива"];

alert( fruits[0] ); // Яблоко

alert( fruits[1] ); // Апельсин

alert( fruits[2] ); // Слива

Можно заменить элемент:

fruits[2] = 'Груша'; // ["Яблоко", "Апельсин", "Груша"]

Или добавить новый элемент к существующему массиву:

fruits[3] = 'Лимон'; // ["Яблоко", "Апельсин", "Груша", "Лимон"]

Общее число элементов массива содержится в его свойстве length:

let fruits = ["Яблоко", "Апельсин", "Слива"];

alert( fruits.length ); // 3

Вывести массив целиком можно при помощи alert:

let fruits = ["Яблоко", "Апельсин", "Слива"];

alert( fruits ); // Яблоко, Апельсин, Слива

В массиве могут храниться элементы любого типа:

let arr = [ 'Яблоко', { name: 'Джон' }, true, function() { alert('привет'); } ];

alert( arr[1].name ); // Джон

arr[3](); // привет

**«Висячая» запятая**

Массив может оканчиваться запятой:

let fruits = [

"Яблоко",

"Апельсин",

"Слива",

];

«Висячая» запятая упрощает процесс добавления/удаления элементов, так как все строки становятся идентичными.

[**Методы pop/push, shift/unshift**](https://learn.javascript.ru/array#metody-pop-push-shift-unshift)

[Очередь](https://ru.wikipedia.org/wiki/%D0%9E%D1%87%D0%B5%D1%80%D0%B5%D0%B4%D1%8C_(%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)) – один из самых распространённых вариантов применения массива. В области компьютерных наук так называется упорядоченная коллекция элементов, поддерживающая два вида операций:

* push добавляет элемент в конец.
* shift удаляет элемент в начале, сдвигая очередь, так что второй элемент становится первым.

Массивы поддерживают обе операции. На практике необходимость в этом возникает очень часто. Например, очередь сообщений, которые надо показать на экране. Существует и другой вариант применения для массивов – структура данных, называемая [стек](https://ru.wikipedia.org/wiki/%D0%A1%D1%82%D0%B5%D0%BA)). Она поддерживает два вида операций:

* push добавляет элемент в конец.
* pop удаляет последний элемент.

Таким образом, новые элементы всегда добавляются или удаляются из «конца». Примером стека обычно служит колода карт: новые карты кладутся наверх и берутся тоже сверху. Массивы в JavaScript могут работать и как очередь, и как стек. Можно добавлять/удалять элементы как в начало, так и в конец массива.

В компьютерных науках структура данных, делающая это возможным, называется [двусторонняя очередь](https://ru.wikipedia.org/wiki/%D0%94%D0%B2%D1%83%D1%85%D1%81%D1%82%D0%BE%D1%80%D0%BE%D0%BD%D0%BD%D1%8F%D1%8F_%D0%BE%D1%87%D0%B5%D1%80%D0%B5%D0%B4%D1%8C).

Методы, работающие с концом массива:

* pop – удаляет последний элемент из массива и возвращает его:

let fruits = ["Яблоко", "Апельсин", "Груша"];

alert( fruits.pop() ); // удаляем "Груша" и выводим его

alert( fruits ); // Яблоко, Апельсин

* push – добавляет элемент в конец массива:

let fruits = ["Яблоко", "Апельсин"];

fruits.push("Груша");

alert( fruits ); // Яблоко, Апельсин, Груша

Вызов fruits.push(...) равнозначен fruits[fruits.length] = ....

Методы, работающие с началом массива:

* shift – удаляет из массива первый элемент и возвращает его:

let fruits = ["Яблоко", "Апельсин", "Груша"];

alert( fruits.shift() ); // удаляем Яблоко и выводим его

alert( fruits ); // Апельсин, Груша

* unshift – добавляет элемент в начало массива:

let fruits = ["Апельсин", "Груша"];

fruits.unshift('Яблоко');

alert( fruits ); // Яблоко, Апельсин, Груша

Методы push и unshift могут добавлять сразу несколько элементов:

let fruits = ["Яблоко"];

fruits.push("Апельсин", "Груша");

fruits.unshift("Ананас", "Лимон");

alert( fruits ); // ["Ананас", "Лимон", "Яблоко", "Апельсин", "Груша"]

[**Внутреннее устройство массива**](https://learn.javascript.ru/array#vnutrennee-ustroystvo-massiva)

Массив – это особый подвид объектов. Квадратные скобки, используемые для того, чтобы получить доступ к свойству arr[0] – это обычный синтаксис доступа по ключу, как obj[key], где в роли obj выступает arr, а в качестве ключа – числовой индекс. Массивы расширяют объекты, так как предусматривают специальные методы для работы с упорядоченными коллекциями данных, а также свойство length. Но в основе все равно лежит объект.

Следует помнить, что в JavaScript существует всего 7 основных типов данных. Массив является объектом и, следовательно, ведёт себя как объект. Например, копируется по ссылке:

let fruits = ["Банан"]

let arr = fruits; // копируется по ссылке

alert( arr === fruits ); // true

arr.push("Груша"); // массив заполняется по ссылке

alert( fruits ); // Банан, Груша

Но то, что действительно делает массивы особенными – это их внутреннее представление. Движок JavaScript старается хранить элементы массива в непрерывной области памяти, один за другим. Существуют и другие способы оптимизации, благодаря которым массивы работают очень быстро. Но все они утратят эффективность, если перестать работать с массивом как с «упорядоченной коллекцией данных», и начать использовать его как обычный объект. Например, технически, можно сделать следующее:

let fruits = []; // создаём массив

fruits[99999] = 5; // создаём свойство с индексом, больше длины массива

fruits.age = 25; // создаём свойство с произвольным именем

Это возможно, потому что в основе массива лежит объект. Можно присвоить ему любые свойства. Но движок поймёт, что осуществляется работа с массивом, как с обычным объектом. Способы оптимизации, используемые для массивов, в этом случае не подходят и поэтому они будут отключены и никакой выгоды не принесут.

Варианты неправильного применения массива:

* Добавление нечислового свойства, например, arr.test = 5.
* Создание пустого пространства, например: добавление arr[0], затем arr[1000] (между ними ничего нет).
* Заполнение массива в обратном порядке, например: arr[1000], arr[999] и т.д.

Массив следует считать особой структурой, позволяющей работать с упорядоченными данными. Для этого массивы предоставляют специальные методы. Массивы тщательно настроены в движках JavaScript для работы с однотипными упорядоченными данными, поэтому, следует использовать их в таких случаях. Если нужны произвольные ключи, то лучше подойдёт обычный объект {}.

[**Эффективность**](https://learn.javascript.ru/array#effektivnost)

Методы push/pop выполняются быстро, а методы shift/unshift – медленно, т.е. у работать с концом массива быстрее, чем с его началом. Давайте посмотрим, что происходит во время выполнения:

fruits.shift(); // удаляем 1-ый элемент с начала

Просто взять и удалить элемент с номером 0 недостаточно. Нужно также заново пронумеровать остальные элементы. Операция shift должна выполнить 3 действия:

1. Удалить элемент с индексом 0.
2. Сдвинуть все элементы влево, заново пронумеровать их, заменив 1 на 0, 2 на 1 и т.д.
3. Обновить свойство length .

Чем больше элементов содержит массив, тем больше времени потребуется для того, чтобы их переместить, больше операций с памятью.

То же самое происходит с unshift: чтобы добавить элемент в начало массива, нужно сначала сдвинуть существующие элементы вправо, увеличивая их индексы.

В процессе работы методам push/pop не нужно ничего перемещать. Чтобы удалить элемент в конце массива, метод pop очищает индекс и уменьшает значение length. Действия при операции pop:

fruits.pop(); // удаляем 1 элемент с конца

Метод pop не требует перемещения, потому что остальные элементы остаются на тех же индексах. Именно поэтому он выполняется очень быстро.

Аналогично работает метод push.

[**Перебор элементов**](https://learn.javascript.ru/array#perebor-elementov)

Одним из самых старых способов перебора элементов массива является цикл for по цифровым индексам:

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

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

alert( arr[i] );

}

Но для массивов возможен и другой вариант цикла – for..of:

let fruits = ["Яблоко", "Апельсин", "Слива"];

// проходит по значениям

for (let fruit of fruits) {

alert( fruit );

}

Цикл for..of не предоставляет доступа к номеру текущего элемента, только к его значению, но в большинстве случаев этого достаточно, а также это короче.

Технически, так как массив является объектом, можно использовать и цикл for..in:

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

for (let key in arr) {

alert( arr[key] ); // Яблоко, Апельсин, Груша

}

Но так делать не рекомендуется. Существуют скрытые недостатки этого способа:

1. Цикл for..in выполняет перебор всех свойств объекта, а не только цифровых.

В браузере и других программных средах также существуют так называемые «псевдомассивы» – объекты, которые выглядят, как массив. То есть, у них есть свойство length и индексы, но также они могут иметь дополнительные нечисловые свойства и методы, которые обычно не нужны. Тем не менее, цикл for..in выведет и их. Поэтому, если приходится иметь дело с объектами, похожими на массив, такие «лишние» свойства могут стать проблемой.

1. Цикл for..in оптимизирован под произвольные объекты, не массивы, и поэтому в 10-100 раз медленнее. Увеличение скорости выполнения может иметь значение только при возникновении узких мест.

Поэтому не следует использовать цикл for..in для массивов.

[**Свойство length**](https://learn.javascript.ru/array#nemnogo-o-length)

Свойство length автоматически обновляется при изменении массива. Если быть точными, это не количество элементов массива, а наибольший цифровой индекс плюс один. Например, единственный элемент, имеющий большой индекс, даёт большую длину:

let fruits = [];

fruits[123] = "Яблоко";

alert( fruits.length ); // 124

Свойство length можно перезаписать. Если вручную увеличить его, то ничего особенного не произойдет. Но если уменьшить – массив станет короче. Этот процесс необратим:

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

arr.length = 2; // укорачиваем до 2 элементов

alert( arr ); // [1, 2]

arr.length = 5; // возвращаем length как было

alert( arr[3] ); // undefined: значения не восстановились

Таким образом, самый простой способ очистить массив – это arr.length = 0;.

[**new Array()**](https://learn.javascript.ru/array#new-array)

Существует ещё один вариант синтаксиса для создания массива:

let arr = new Array("Яблоко", "Груша", "и тд");

Он редко применяется, так как квадратные скобки [] короче. Кроме того, у него есть одна особенность: если new Array вызывается с одним аргументом, который представляет собой число, он создаёт массив без элементов, но с заданной длиной:

let arr = new Array(2); // создаем массив [2]

alert( arr[0] ); // undefined! нет элементов.

alert( arr.length ); // length 2

Как видно из кода, представленного выше, в new Array(number) все элементы равны undefined. Чтобы избежать появления таких неожиданных ситуаций, обычно используются квадратные скобки, если конечно нет какой-то причины для использования именно Array.

[**Многомерные массивы**](https://learn.javascript.ru/array#mnogomernye-massivy)

Массивы могут содержать элементы, которые тоже являются массивами. Это можно использовать для создания многомерных массивов, например, для хранения матриц:

let matrix = [

[1, 2, 3],

[4, 5, 6],

[7, 8, 9]

];

alert( matrix[1][1] ); // 5, центральный элемент

**Метод** [**toString**](https://learn.javascript.ru/array#tostring)

Массивы по-своему реализуют метод toString, который возвращает список элементов, разделённых запятыми. Например:

let arr = [1, 2, 3];

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

alert( String(arr) === '1,2,3' ); // true

alert( [] + 1 ); // "1"

alert( [1] + 1 ); // "11"

alert( [1,2] + 1 ); // "1,21"

Массивы не имеют ни Symbol.toPrimitive, ни функционирующего valueOf, они реализуют только преобразование toString, таким образом здесь [] становится пустой строкой, [1] становится "1", а [1,2] становится "1,2". Когда бинарный оператор плюс "+" добавляет что-либо к строке, он тоже преобразует это в строку, таким образом:

alert( "" + 1 ); // "1"

alert( "1" + 1 ); // "11"

alert( "1,2" + 1 ); // "1,21"

1. **Методы массивов.**

**[Добавление/удаление элементов](https://learn.javascript.ru/array-methods" \l "dobavlenie-udalenie-elementov)**

Методы, которые добавляют и удаляют элементы из начала или конца, были рассмотрены ранее:

* arr.push(...items) – добавляет элементы в конец,
* arr.pop() – извлекает элемент из конца,
* arr.shift() – извлекает элемент из начала,
* arr.unshift(...items) – добавляет элементы в начало.

Есть и другие.

**Метод** [**splice**](https://learn.javascript.ru/array-methods#splice)

Метод [arr.splice(str)](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/Array/splice) – это универсальный метод для работы с массивами, который позволяет добавлять, удалять и заменять элементы.

Его синтаксис:

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

Он начинает с позиции *index*, удаляет *deleteCount* элементов и вставляет *elem1*, ..., *elemN* на их место. Возвращает массив из удалённых элементов.

Рассмотрим пример удаления элементов:

let arr = ["Я", "изучаю", "JavaScript"];

arr.splice(1, 1); // начиная с позиции 1, удалить 1 элемент

alert( arr ); // осталось ["Я", "JavaScript"]

Начиная с позиции 1, метод удалил 1 элемент.

Удалим 3 элемента и заменим их двумя другими:

let arr = ["Я", "изучаю", "JavaScript", "прямо", "сейчас"];

arr.splice(0, 3, "Давай", "танцевать");

alert( arr ) // теперь ["Давай", "танцевать", "прямо", "сейчас"]

Здесь видно, что splice возвращает массив из удалённых элементов:

let arr = ["Я", "изучаю", "JavaScript", "прямо", "сейчас"];

// удалить 2 первых элемента

let removed = arr.splice(0, 2);

alert( removed ); // "Я", "изучаю" – массив из удалённых элементов

Метод splice также может вставлять элементы без удаления, для этого достаточно установить deleteCount в 0:

let arr = ["Я", "изучаю", "JavaScript"];

// с позиции 2 удалить 0 элементов и вставить "сложный", "язык"

arr.splice(2, 0, "сложный", "язык");

alert( arr ); // "Я", "изучаю", "сложный", "язык", "JavaScript"

В этом и в других методах массива допускается использование отрицательного индекса. Он позволяет начать отсчёт элементов с конца:

let arr = [1, 2, 5];

// начиная с индекса -1 (перед последним элементом) удалить 0 элементов,

// затем вставить числа 3 и 4

arr.splice(-1, 0, 3, 4);

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

**Метод** [**slice**](https://learn.javascript.ru/array-methods#slice)

Метод [arr.slice](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/Array/slice) намного проще, чем похожий на него arr.splice. Его синтаксис:

arr.slice(start, end)

Он возвращает новый массив, в который копирует элементы, начиная с индекса start и до end (не включая end). Оба индекса start и end могут быть отрицательными. В таком случае отсчёт будет осуществляться с конца массива. Метод похож на строковый метод str.slice, но вместо подстрок возвращает подмассивы. Например:

let arr = ["t", "e", "s", "t"];

alert( arr.slice(1, 3) ); // e,s

alert( arr.slice(-2) ); // s,t

**Метод** [**concat**](https://learn.javascript.ru/array-methods#concat)

Метод [arr.concat](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/Array/concat) создаёт новый массив, в который копирует данные из других массивов и дополнительные значения. Его синтаксис:

arr.concat(arg1, arg2...)

Он принимает любое количество аргументов, которые могут быть как массивами, так и простыми значениями. В результате будет сформирован новый массив, включающий в себя элементы из *arr*, а также *arg1*, *arg2* и так далее.

Если аргумент *argN* – массив, то все его элементы копируются, иначе скопируется сам аргумент. Например:

let arr = [1, 2];

alert( arr.concat([3, 4])); // 1,2,3,4

alert( arr.concat([3, 4], [5, 6])); // 1,2,3,4,5,6

alert( arr.concat([3, 4], 5, 6)); // 1,2,3,4,5,6

Обычно он просто копирует элементы из массивов. Другие объекты, даже если они выглядят как массивы, добавляются как есть:

let arr = [1, 2];

let arrayLike = {

0: "что-то",

length: 1

};

alert( arr.concat(arrayLike) ); // 1,2,[object Object]

//[1, 2, arrayLike]

Но если объект имеет специальное свойство Symbol.isConcatSpreadable, то он обрабатывается concat как массив: вместо него добавляются его элементы:

let arr = [1, 2];

let arrayLike = {

0: "что-то",

1: "ещё",

[Symbol.isConcatSpreadable]: true,

length: 2

};

alert( arr.concat(arrayLike) ); // 1,2,что-то,ещё

[**Перебор массива: метод forEach**](https://learn.javascript.ru/array-methods#perebor-foreach)

Метод [arr.forEach](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/Array/forEach) позволяет запускать функцию для каждого элемента массива. Его синтаксис:

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

// вычисления с item

});

Например, этот код выведет на экран каждый элемент массива:

["Bilbo", "Gandalf", "Nazgul"].forEach(alert);

А этот выведет значение элемента и его позицию в массиве:

["Bilbo", "Gandalf", "Nazgul"].forEach((item, index, array) => {

alert(`${item} имеет позицию ${index} в ${array}`);

});

Результат функции (если она вообще что-то возвращает) отбрасывается и игнорируется.

**Методы** [**поиска**](https://learn.javascript.ru/array-methods#poisk-v-massive) **в массиве**

Методы [arr.indexOf](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/Array/indexOf), [arr.lastIndexOf](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/Array/lastIndexOf) и [arr.includes](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/Array/includes) имеют одинаковый синтаксис и делают по сути то же самое, что и их строковые аналоги, но работают с элементами вместо символов:

* arr.indexOf(item, from) ищет item, начиная с индекса from, и возвращает индекс, на котором был найден искомый элемент, в противном случае -1.
* arr.lastIndexOf(item, from) – то же самое, но ищет справа налево.
* arr.includes(item, from) – ищет item, начиная с индекса from, и возвращает true, если поиск успешен.

Например:

let arr = [1, 0, false];

alert( arr.indexOf(0) ); // 1

alert( arr.indexOf(false) ); // 2

alert( arr.indexOf(null) ); // -1

alert( arr.includes(1) ); // true

Обратите внимание, что методы используют строгое сравнение ===. Таким образом, если осуществляется поиск false, то он находит именно false, а не ноль.

Если надо проверить наличие элемента и нет необходимости знать его точный индекс, тогда предпочтительным является arr.includes.

Метод [arr.find](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/Array/find) позволяет найти элемент с определённым условием. Его синтаксис таков:

let result = arr.find(function(item, index, array) {

// если true - возвращается текущий элемент и перебор прерывается

// если все итерации оказались ложными возвращается undefined

});

Функция вызывается по очереди для каждого элемента массива:

* item – очередной элемент.
* index – его индекс.
* array – сам массив.

Если функция вернёт true, поиск прерывается и возвращается item. Если ничего не найдено, возвращается undefined. Например, имеется массив пользователей, каждый из которых имеет поля id и name. Найдем того, кто с id == 1:

let users = [

{id: 1, name: "Вася"},

{id: 2, name: "Петя"},

{id: 3, name: "Маша"}

];

let user = users.find(item => item.id == 1);

alert(user.name); // Вася

Метод [arr.findIndex](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/Array/findIndex) – в отличие от arr.find возвращает индекс, на котором был найден элемент, а не сам элемент, и -1, если ничего не найдено.

Метод find ищет один (первый попавшийся) элемент, на котором функция-колбэк вернёт true. Если найденных элементов может быть много, то стоит использовать метод [arr.filter(fn)](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/Array/filter). Синтаксис этого метода схож с find, но filter возвращает массив из всех подходящих элементов:

let results = arr.filter(function(item, index, array) {

// если true - элемент добавляется к результату и перебор продолжается

// возвращается пустой массив в случае, если ничего не найдено

});

Например:

let users = [

{id: 1, name: "Вася"},

{id: 2, name: "Петя"},

{id: 3, name: "Маша"}

];

// возвращает массив, состоящий из двух первых пользователей

let someUsers = users.filter(item => item.id < 3);

alert(someUsers.length); // 2

[**Преобразование массива**](https://learn.javascript.ru/array-methods#preobrazovanie-massiva)

Метод [arr.map](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/Array/map) является одним из наиболее полезных и часто используемых. Он вызывает функцию для каждого элемента массива и возвращает массив результатов выполнения этой функции. Синтаксис:

let result = arr.map(function(item, index, array) {

// возвращается новое значение вместо элемента

});

Например, в коде ниже осуществляется преобразование каждого элемента в его длину:

let lengths = ["Bilbo", "Gandalf", "Nazgul"].map(item => item.length);

alert(lengths); // 5,7,6

Вызов [arr.sort()](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/Array/sort) сортирует массив на месте, меняя в нём порядок элементов. Он возвращает отсортированный массив, но обычно возвращаемое значение игнорируется, так как изменяется сам arr. Например:

let arr = [ 1, 2, 15 ];

arr.sort();

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

Порядок стал 1, 15, 2 так как элементы преобразуются в строки при сравнении и по умолчанию сортируются как строки. Для сортировки строк применяется лексикографический порядок где "2" > "15". Чтобы задать свой порядок сортировки, нужно предоставить функцию в качестве аргумента arr.sort(). Функция может возвращать следующие значения:

function compare(a, b) {

if (a > b) return 1; // если первое значение больше второго

if (a == b) return 0; // если равны

if (a < b) return -1; // если первое значение меньше второго

}

Например, для сортировки чисел:

function compareNumeric(a, b) {

if (a > b) return 1;

if (a == b) return 0;

if (a < b) return -1;

}

let arr = [ 1, 2, 15 ];

arr.sort(compareNumeric);

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

Функция сравнения может возвращать любое положительное число, если первый сравниваемый элемент больше второго и отрицательное – если меньше. Это позволяет писать более короткие функции:

let arr = [ 1, 2, 15 ];

arr.sort(function(a, b) { return a - b; });

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

Можно использовать стрелочные функции, чтобы сортировка выглядела более компактной:

arr.sort( (a, b) => a - b );

Будет работать точно так же, как и более длинная версия выше.

Метод [arr.reverse](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/Array/reverse) меняет порядок элементов в arr на обратный. Например:

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

arr.reverse();

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

Он также возвращает массив arr с изменённым порядком элементов.

Метод [str.split(delim)](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/String/split) разбивает строку на массив по заданному разделителю delim. В примере ниже таким разделителем является строка из запятой и пробела.

let names = 'Вася, Петя, Маша';

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

for (let name of arr) {

alert( `Сообщение получат: ${name}.` );

}

У метода split есть необязательный второй числовой аргумент – ограничение на количество элементов в массиве. Если их больше, чем указано, то остаток массива будет отброшен. На практике это редко используется:

let arr = 'Вася, Петя, Маша, Саша'.split(', ', 2);

alert(arr); // Вася, Петя

Вызов split(s) с пустым аргументом s разделяет строку на массив букв:

let str = "тест";

alert( str.split('') ); // т,е,с,т

Вызов [arr.join(glue)](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/Array/join) выполняет действие противоположное split. Он создаёт строку из элементов arr, вставляя glue между ними. Например:

let arr = ['Вася', 'Петя', 'Маша'];

let str = arr.join(';'); // объединить массив в строку через ;

alert( str ); // Вася;Петя;Маша

Если надо перебрать массив – можно использовать forEach, for или for..of. Если надо перебрать массив и возвратить данные для каждого элемента – стоит используем map. Методы [arr.reduce](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/Array/reduce) и [arr.reduceRight](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/Array/reduceRight) похожи на методы выше, но они немного сложнее. Они используются для вычисления какого-нибудь единого значения на основе всего массива. Синтаксис:

let value = arr.reduce(function(previousValue, item, index, array) {

// ...

}, [initial]);

Функция применяется по очереди ко всем элементам массива и «переносит» свой результат на следующий вызов. Аргументы:

* previousValue – результат предыдущего вызова этой функции, равен initial при первом вызове (если передан initial),
* item – очередной элемент массива,
* index – индекс элемента,
* array – массив.

При вызове функции результат её вызова на предыдущем элементе массива передаётся как первый аргумент. Звучит сложновато, но всё становится проще, если думать о первом аргументе как «аккумулирующем» результат предыдущих вызовов функции. По окончании он становится результатом reduce. Этот метод проще всего понять на примере. Получим сумму всех элементов массива:

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

let result = arr.reduce((sum, current) => sum + current, 0);

alert(result); // 15

Здесь использован наиболее распространённый вариант reduce, который использует только 2 аргумента. Рассмотрим, как он работает:

1. При первом запуске, sum равен initial (последний аргумент reduce), то есть 0, а current – первый элемент массива, равнй 1. Таким образом, результат функции равен 1.
2. При втором запуске sum = 1, и к нему добавляется второй элемент массива (2).
3. На 3-м запуске sum = 3, к которому добавляется следующий элемент и так далее.

Ниже представлена таблица, где каждая строка – вызов функции на очередном элементе массива:

|  | **sum** | **current** | **result** |
| --- | --- | --- | --- |
| первый вызов | 0 | 1 | 1 |
| второй вызов | 1 | 2 | 3 |
| третий вызов | 3 | 3 | 6 |
| четвёртый вызов | 6 | 4 | 10 |
| пятый вызов | 10 | 5 | 15 |

Здесь отчётливо видно, как результат предыдущего вызова передаётся в первый аргумент следующего. Также можно опустить начальное значение:

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

let result = arr.reduce((sum, current) => sum + current);

alert( result ); // 15

Результат такой же потому, что при отсутствии initial в качестве первого значения берётся первый элемент массива, а перебор стартует со второго. Таблица вычислений будет такая же за вычетом первой строки. Но такое использование требует крайней осторожности. Если массив пуст, то вызов reduce без начального значения выдаст ошибку, поэтому рекомендуется всегда его указывать:

let arr = [];

// Error: Reduce of empty array with no initial value

arr.reduce((sum, current) => sum + current);

Метод [arr.reduceRight](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/Array/reduceRight) работает аналогично, но проходит по массиву справа налево.

Массивы не образуют отдельный тип языка. Они основаны на объектах. Поэтому typeof не может отличить простой объект от массива:

alert(typeof {}); // object

alert(typeof []); // object

Метод [Array.isArray(value)](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/Array/isArray) возвращает true, если value массив, и false, если нет:

alert(Array.isArray({})); // false

alert(Array.isArray([])); // true

Почти все методы массива, которые вызывают функции – такие как find, filter, map, за исключением метода sort, принимают необязательный параметр thisArg. Этот параметр очень редко используется. Полный синтаксис этих методов:

arr.find(func, thisArg);

arr.filter(func, thisArg);

arr.map(func, thisArg);

Значение параметра thisArg становится this для func. Например, если необходимо использовать метод объекта как фильтр, то thisArg с этим поможет:

let user = {

age: 18,

younger(otherUser) {

return otherUser.age < this.age;

}

};

let users = [

{age: 12},

{age: 16},

{age: 32}

];

// найти число пользователей моложе, чем заданный

let youngerUsers = users.filter(user.younger, user);

alert(youngerUsers.length); // 2

В вызове выше используется user.younger как фильтр, а user – в качестве контекста для него. Если бы не предоставлялся контекст, users.filter(user.younger) вызвал бы user.younger как ни к чему не привязанную функцию с this=undefined. Это привело бы кошибке.

1. **Объекты.**

Как известно, в JavaScript существует семь типов данных. Шесть из них называются примитивными, так как содержат только одно значение. Объекты же используются для хранения коллекций различных значений и более сложных сущностей. В JavaScript объекты используются очень часто, это одна из основ языка. Поэтому стоит их досконально изучить.

Объект может быть создан с помощью фигурных скобок {…} с необязательным списком свойств. Свойство – это пара «ключ: значение», где ключ – это строка (также называемая «именем свойства»), а значение может быть чем угодно.

Пустой объект можно создать, используя один из двух вариантов синтаксиса:

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

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

Обычно используют вариант с фигурными скобками {...}. Такое объявление называют литералом объекта или литеральной нотацией.

[**Литералы и свойства**](https://learn.javascript.ru/object#literaly-i-svoystva)

При использовании литерального синтаксиса {...} сразу можно поместить в объект несколько свойств в виде пар «ключ: значение»:

let user = { // объект

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

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

};

Свойства объекта также иногда называют полями объекта. У каждого свойства есть ключ (также называемый «имя» или «идентификатор»). После имени свойства следует двоеточие ":", и затем указывается значение свойства. Если в объекте несколько свойств, то они перечисляются через запятую.

В объекте user сейчас находятся два свойства:

1. Первое свойство с именем "name" и значением "John".
2. Второе свойство с именем "age" и значением 30.

Для обращения к свойствам используется запись «через точку»:

// получаем свойства объекта:

alert( user.name ); // John

alert( user.age ); // 30

Значение может быть любого типа. Добавим свойство с логическим значением:

user.isAdmin = true;

Для удаления свойства можно использовать оператор delete:

delete user.age;

Имя свойства может состоять из нескольких слов, но тогда оно должно быть заключено в кавычки. Последнее свойство объекта может заканчиваться висячей запятой:

let user = {

name: "John",

age: 30,

"likes birds": true,

};

[**Квадратные скобки**](https://learn.javascript.ru/object#kvadratnye-skobki)

Для свойств, имена которых состоят из нескольких слов, доступ к значению «через точку» не работает:

// приведет к синтаксической ошибкк

user.likes birds = true

Так происходит, потому что точка требует, чтобы ключ был именован по правилам именования переменных. То есть не имел пробелов, не начинался с цифры и не содержал специальные символы, кроме $ и \_. Для таких случаев существует альтернативный способ доступа к свойствам через квадратные скобки. Такой способ сработает с любым именем свойства:

let user = {};

// присваивание значения свойству

user["likes birds"] = true;

// получение значения свойства

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

// удаление свойства

delete user["likes birds"];

Квадратные скобки также позволяют обратиться к свойству, имя которого может быть результатом выражения. Например, имя свойства может храниться в переменной:

let key = "likes birds";

user[key] = true;

Здесь переменная key может быть вычислена во время выполнения кода или зависеть от пользовательского ввода. После этого можно использовать её для доступа к свойству. Пример:

let user = {

name: "John",

age: 30

};

let key = prompt("Что вы хотите узнать о пользователе?", "name");

alert( user[key] ); // John (если ввели "name")

Запись «через точку» такого не позволяет:

let user = {

name: "John",

age: 30

};

let key = "name";

user.key // undefined

[**Вычисляемые свойства**](https://learn.javascript.ru/object#vychislyaemye-svoystva)

Можно использовать квадратные скобки в литеральной нотации для создания вычисляемого свойства. Пример:

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"

Смысл вычисляемого свойства прост: запись [fruit] означает, что имя свойства необходимо взять из переменной fruit. И если посетитель введёт слово "apple", то в объекте bag теперь будет храниться свойство {apple: 5}.

Можно использовать и более сложные выражения в квадратных скобках:

let fruit = 'apple';

let bag = {

[fruit + 'Computers']: 5 // bag.appleComputers = 5

};

Таким образом, когда имена свойств известны и просты, используется запись через точку. Если же нужно что-то более сложное, то следует использовать квадратные скобки. Зарезервированные слова разрешено использовать как имена свойств:

let obj = {

for: 1,

let: 2,

return: 3

};

alert( obj.for + obj.let + obj.return ); // 6

[**Свойство из переменной**](https://learn.javascript.ru/object#svoystvo-iz-peremennoy)

В реальном коде часто необходимо использовать существующие переменные как значения для свойств с тем же именем. Например:

function makeUser(name, age) {

return {

name: name,

age: age

// ...другие свойства

};

}

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

alert(user.name); // John

В примере выше название свойств name и age совпадают с названиями переменных, которые указываются в качестве значений этих свойств. Такой подход настолько распространен, что существуют специальные короткие свойства для упрощения этой записи. Вместо name:name можно написать просто name. Можно использовать как обычные свойства, так и короткие в одном и том же объекте:

function makeUser(name, age) {

return {

name,

age

height: 170// ...

};

}

[**Проверка существования свойства**](https://learn.javascript.ru/object#proverka-suschestvovaniya-svoystva)

Особенность объектов в том, что можно получить доступ к любому свойству. Даже если свойства не существует – ошибки не будет. При обращении к свойству, которого нет, возвращается undefined. Это позволяет просто проверить существование свойства – сравнением его с undefined:

let user = {};

alert( user.noSuchProperty === undefined ); // true означает "свойства нет"

Также существует специальный оператор "in" для проверки существования свойства в объекте. Синтаксис оператора:

"key" in object

Пример:

let user = { name: "John", age: 30 };

alert( "age" in user ); // true, user.age существует

alert( "blabla" in user ); // false, user.blabla не существует

Обратите внимание, что слева от оператора in должно быть имя свойства. Обычно это строка в кавычках. Если кавычки опускаются, это значит, что используется переменная, в которой находится имя свойства. Например:

let user = { age: 30 };

let key = "age";

alert( key in user ); // true

Обычно строгого сравнения "=== undefined" достаточно для проверки наличия свойства. Но есть особый случай, когда оно не подходит, и нужно использовать "in". Это когда свойство существует, но содержит значение undefined:

let obj = {

test: undefined

};

alert( obj.test ); // выведет undefined

alert( "test" in obj ); // true, свойство существует

В примере выше свойство obj.test технически существует в объекте. Оператор in сработал правильно. Подобные ситуации случаются очень редко, так как undefined обычно явно не присваивается. Для «неизвестных» или «пустых» свойств используется значение null.

[**Цикл «for…in»**](https://learn.javascript.ru/object#tsikl-for-in)

Для перебора всех свойств объекта используется цикл for..in. Этот цикл отличается от изученного ранее цикла for(;;). Синтаксис:

for (key in object) {

// тело цикла выполняется для каждого свойства объекта

}

Например, выведем все свойства объекта user:

let user = {

name: "John",

age: 30,

isAdmin: true

};

.

for (let key in user) {

alert( key ); // name, age, isAdmin

alert( user[key] ); // John, 30, true

}

Обратите внимание, что все конструкции «for» позволяют объявлять переменную внутри цикла, как, например, let key здесь. Кроме того, можно использовать другое имя переменной. Например, часто используется вариант "for (let prop in obj)".

[**Упорядочение свойств объекта**](https://learn.javascript.ru/object#uporyadochenie-svoystv-obekta)

Свойства объекта упорядочены особым образом: свойства с целочисленными ключами сортируются по возрастанию, остальные располагаются в порядке создания. В качестве примера рассмотрим объект с телефонными кодами:

let codes = {

"49": "Германия",

"41": "Швейцария",

"44": "Великобритания",

// ..,

"1": "США"

};

for (let code in codes) {

alert(code); // 1, 41, 44, 49

}

Телефонные коды идут в порядке возрастания, потому что они являются целыми числами: 1, 41, 44, 49.

Термин «целочисленное свойство» означает строку, которая может быть преобразована в целое число и обратно без изменений. То есть, "49" – это целочисленное имя свойства, потому что если его преобразовать в целое число, а затем обратно в строку, то оно не изменится. А вот свойства "+49" или "1.2" таковыми не являются. Если ключи не целочисленные, то они перебираются в порядке создания, например:

let user = {

name: "John",

surname: "Smith"

};

user.age = 25;

for (let prop in user) {

alert( prop ); // name, surname, age

}

Таким образом, чтобы телефонные коды выводились, в том порядке в котром записаны, надо сделать коды не целочисленными свойствами. Для этого надо добавить знак "+" перед каждым кодом. Пример:

let codes = {

"+49": "Германия",

"+41": "Швейцария",

"+44": "Великобритания",

// ..,

"+1": "США"

};

for (let code in codes) {

alert( +code ); // 49, 41, 44, 1

}

**[Копирование по ссылке](https://learn.javascript.ru/object" \l "kopirovanie-po-ssylke)**

Одним из фундаментальных отличий объектов от примитивных типов данных является то, что они хранятся и копируются «по ссылке». Примитивные типы: строки, числа, логические значения – присваиваются и копируются «по значению». Например:

let message = "Hello!";

let phrase = message;

В результате имеются две независимые переменные, каждая из которых хранит строку "Hello!". Объекты ведут себя иначе. Переменная хранит не сам объект, а его «адрес в памяти», другими словами «ссылку» на него. Например:

let user = {

name: "John"

};

Сам объект хранится где-то в памяти. А в переменной user лежит «ссылка» на эту область памяти. Когда переменная объекта копируется – копируется ссылка, сам же объект не дублируется:

let user = { name: "John" };

let admin = user; // копируется ссылка

В результате есть две переменные, каждая из которых содержит ссылку на один и тот же объект. Можно использовать любую из переменных для доступа к объекту и изменения его содержимого:

let user = { name: 'John' };

let admin = user;

admin.name = 'Pete'; // изменено по ссылке из переменной "admin"

alert(user.name); // 'Pete', изменения видны по ссылке из переменной "user"

**[Сравнение объектов](https://learn.javascript.ru/object" \l "sravnenie-obektov)**

Операторы равенства == и строгого равенства === для объектов работают одинаково. Два объекта равны только в том случае, если это один и тот же объект. Например, две переменные ссылаются на один и тот же объект, они равны:

let a = {};

let b = a; // копирование по ссылке

alert( a == b ); // true

alert( a === b ); // true

В примере ниже два разных объекта не равны, хотя и оба пусты:

let a = {};

let b = {}; // два независимых объекта

alert( a == b ); // false

Для сравнений типа obj1 > obj2 или для сравнения с примитивом obj == 5 объекты преобразуются в примитивы. Такое сравнение используется очень редко и не рекомендуется.

**[Объекты-константы](https://learn.javascript.ru/object" \l "obekty-konstanty)**

Объект, объявленный через const, может быть изменен:

const user = {

name: "John"

};

user.age = 25; // (\*)

alert(user.age); // 25

Объявление const защищает от изменений только само значение user. В примере значение user – это ссылка на объект, и это значение не меняется. В строке (\*) вносятся изменения внутри объекта, а значение user не изменяется. Если же попытаться присвоить user другое значение, то const выдаст ошибку:

const user = {

name: "John"

};

// Ошибка (нельзя переопределять константу user)

user = {

name: "Pete"

};

Cделать константами свойства объекта тоже возможно с помощью флагов и дескрипторов свойств, кторые будут рассматриваться позже.

[**Клонирование и объединение объектов, Object.assign**](https://learn.javascript.ru/object#klonirovanie-i-obedinenie-obektov-object-assign)

При копировании переменной объекта создаётся ещё одна ссылка на тот же самый объект. Если надо создать независимую копию (клон), то необходимо создать новый объект и повторять структуру дублируемого объекта, перебирая его свойства и копируя их. Например так:

let user = {

name: "John",

age: 30

};

let clone = {};

for (let key in user) {

clone[key] = user[key];

}

clone.name = "Pete";

alert( user.name );

Кроме того, для этих целей можно использовать метод [Object.assign](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/Object/assign). Синтаксис:

Object.assign(dest, [src1, src2, src3...])

* Аргументы dest, и src1, ..., srcN (может быть столько, сколько нужно) являются объектами.
* Метод копирует свойства всех объектов src1, ..., srcN в объект dest. То есть, свойства всех перечисленных объектов, начиная со второго, копируются в первый объект. После копирования метод возвращает объект dest.

Например, объединим несколько объектов в один:

let user = { name: "John" };

let permissions1 = { canView: true };

let permissions2 = { canEdit: true };

// user = { name: "John", canView: true, canEdit: true }

Object.assign(user, permissions1, permissions2);

Если принимающий объект (user) уже имеет свойство с таким именем, оно будет перезаписано:

let user = { name: "John" };

// user = { name: "Pete", isAdmin: true }

Object.assign(user, { name: "Pete", isAdmin: true });

Также можно использовать Object.assign для простого клонирования:

let user = {

name: "John",

age: 30

};

let clone = Object.assign({}, user);

Все свойства объекта user будут скопированы в пустой объект, и ссылка на этот объект будет в переменной clone. Такое клонирование работает так же, как и через цикл, но короче.

Если свойство не примитивно, а явлется ссылкой на другой объект, то при клонировании недостаточно просто скопировать clone.sizes = user.sizes, поскольку user.sizes – это объект, он будет скопирован по ссылке. А значит объекты clone и user в своих свойствах sizes будут ссылаться на один и тот же объект:

let user = {

name: "John",

sizes: {

height: 182,

width: 50

}

};

let clone = Object.assign({}, user);

alert( user.sizes === clone.sizes );

user.sizes.width++;

alert(clone.sizes.width); // 51

Чтобы исправить это, необходимо в цикле клонирования делать проверку, не является ли значение user[key] объектом, и, если это так, – копировать и его структуру тоже. Это называется «глубокое клонирование». Существует стандартный алгоритм глубокого клонирования, [Structured cloning algorithm](http://w3c.github.io/html/infrastructure.html#safe-passing-of-structured-data). Он решает описанную выше задачу, а также более сложные задачи.

1. **Тип данных Symbol**

По спецификации, в качестве ключей для свойств объекта могут использоваться только строки либо символы. Ни числа, ни логические значения не подходят, разрешены только эти два типа данных.

Символ представляет собой уникальный идентификатор. Создаются новые символы с помощью функции Symbol(). Создание нового символа – id:

let id = Symbol();

При создании символу можно дать описание (также называемое имя), в основном использующееся для отладки кода. Например, создадим символ id с описанием (именем) "id":

let id = Symbol("id");

Символы гарантированно уникальны. Если создать множество символов с одинаковым описанием, это всё равно будут разные символы. Описание – это просто метка, которая ни на что не влияет. Например, два символа с одинаковым описанием, но они не равны:

let id1 = Symbol("id");

let id2 = Symbol("id");

alert(id1 == id2); // false

Большинство типов данных в JavaScript могут быть неявно преобразованы в строку. Например, функция alert принимает практически любое значение, автоматически преобразовывает его в строку, а затем выводит это значение, не сообщая об ошибке. Символы же особенные и не преобразуются автоматически. К примеру, alert ниже выдаст ошибку:

let id = Symbol("id");

alert(id); // TypeError: Cannot convert a Symbol value to a string

Если нужно вывести символ с помощью alert, то необходимо явно преобразовать его с помощью метода .toString():

let id = Symbol("id");

alert(id.toString()); // Symbol(id)

Также можно обратиться к свойству symbol.description, чтобы вывести только описание:

let id = Symbol("id");

alert(id.description); // id

[**«Скрытые» свойства**](https://learn.javascript.ru/symbol#skrytye-svoystva) **объекта**

Символы позволяют создавать «скрытые» свойства объектов, к которым нельзя нечаянно обратиться и перезаписать их из других частей программы. Например, выполняются действия с объектами user, которые принадлежат стороннему коду, и в которых нет поля "id". Надо добавить к ним идентификаторы. Можно использовать для этого символьный ключ:

let user = { name: "Егор" };

let id = Symbol("id");

user[id] = "ID Value";

alert( user[id] );

Лучше использовать Symbol("id"), а не строку "id", так как объект user принадлежит стороннему коду, и этот код также работает с ним, то не следует добавлять к нему какие-либо поля. Это небезопасно. Но к символу нечаянно обратиться сложно, сторонний код вряд ли его вообще увидит. Сторонний код может создать для этого свой символ Symbol("id"):

let id = Symbol("id");

user[id] = "Их идентификатор";

Конфликта между двумя идентификатором не будет, так как символы всегда уникальны, даже если их имена совпадают. А вот если использовать строку "id" вместо символа, то тогда будет конфликт:

let user = { name: "Егор" };

user.id = "Первый идентификатор";

// свойство перезаписано

user.id = "Второй идентификатор"

Если необходимо использовать символ при литеральном объявлении объекта {...}, его необходимо заключить в квадратные скобки:

let id = Symbol("id");

let user = {

name: "Егор",

[id]: 123

};

Это связано с необходимостью использовать значение переменной id в качестве ключа, а не строку «id».

Свойства, чьи ключи – символы, не перебираются циклом for..in. Например:

let id = Symbol("id");

let user = {

name: "Егор",

age: 30,

[id]: 123

};

for (let key in user) alert(key); // name, age

alert( "Напрямую: " + user[id] );

Это невозможно с целью скрытия символьных свойств. Если другая библиотека или скрипт будут работать с текущим объектом, то при переборе они не получат символьное свойство. Object.keys(user) также игнорирует символы. А вот [Object.assign](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/Object/assign), в отличие от цикла for..in, копирует и строковые, и символьные свойства:

let id = Symbol("id");

let user = {

[id]: 123

};

let clone = Object.assign({}, user);

alert( clone[id] ); // 123

Так происходит потому, что, когда осуществляется клонирование или объединение объектов, обычно необходимо скопировать все свойства (включая такие свойства с ключами-символами, как, например, id в примере выше).

[**Глобальные символы**](https://learn.javascript.ru/symbol#globalnye-simvoly)

Иногда необходимо, чтобы символы с одинаковыми именами были одной сущностью. Например, разные части приложения хотят получить доступ к символу "id", подразумевая именно одно и то же свойство. Для этого существует глобальный реестр символов. Можно создавать в нём символы и обращаться к ним позже, и при каждом обращении гарантированно будет возвращаться один и тот же символ.

Для чтения (или, при отсутствии, создания) символа из реестра используется вызов Symbol.for(key). Он проверяет глобальный реестр, и если в нём есть символ с именем key, то возвращает его, иначе создает новый символ Symbol(key) и записывает его в реестр под ключом key. Например:

let id = Symbol.for("id");

let idAgain = Symbol.for("id");

alert( id === idAgain ); // true

Символы, содержащиеся в реестре, называются глобальными символами. Если нужен символ, доступный везде в коде – используйте глобальные символы.

Для глобальных символов, кроме Symbol.for(key), который ищет символ по имени, существует обратный метод: Symbol.keyFor(sym), который, наоборот, принимает глобальный символ и возвращает его имя. Например:

// получаем символ по имени

let sym = Symbol.for("name");

let sym2 = Symbol.for("id");

// получаем имя по символу

alert( Symbol.keyFor(sym) ); // name

alert( Symbol.keyFor(sym2) ); // id

Внутри метода Symbol.keyFor используется глобальный реестр символов для нахождения имени символа. Так что этот метод не будет работать для неглобальных символов. Если символ неглобальный, метод не сможет его найти и вернёт undefined. Не стоит забывать, что для любых символов доступно свойство description.

**[Системные символы](https://learn.javascript.ru/symbol" \l "sistemnye-simvoly)**

Существует множество «системных» символов, использующихся внутри самого JavaScript, и можно использовать их, чтобы настраивать различные аспекты поведения объектов. Эти символы перечислены в спецификации в таблице [Well-known symbols](https://tc39.github.io/ecma262/#sec-well-known-symbols):

* Symbol.hasInstance
* Symbol.isConcatSpreadable
* Symbol.iterator
* Symbol.toPrimitive и др.

В частности, Symbol.toPrimitive позволяет описать правила для объекта, согласно которым он будет преобразовываться к примитиву.

1. **Set, Map, WeakSet и WeakMap**

Ранее рассматривались следующие сложные структуры данных: объекты – для хранения именованных коллекций, массивы – для хранения упорядоченных коллекций. Существуют и другие более сложные коллекции.

[**Map**](https://learn.javascript.ru/map-set#map)

[Map](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/Map) – это коллекция ключ/значение, как и Object. Но основное отличие в том, что Map позволяет использовать ключи любого типа. Методы и свойства:

* new Map() – создаёт коллекцию.
* map.set(key, value) – записывает по ключу key значение value.
* map.get(key) – возвращает значение по ключу или undefined, если ключ key отсутствует.
* map.has(key) – возвращает true, если ключ key присутствует в коллекции, иначе false.
* map.delete(key) – удаляет элемент по ключу key.
* map.clear() – очищает коллекцию от всех элементов.
* map.size – возвращает текущее количество элементов.

Например:

let map = new Map();

map.set("1", "str1"); // строка в качестве ключа

map.set(1, "num1"); // цифра как ключ

map.set(true, "bool1"); // булево значение как ключ

alert(map.get(1)); // "num1"

alert(map.get("1")); // "str1"

alert(map.size); // 3

Как мы видим, в отличие от объектов, ключи не были приведены к строкам. Можно использовать любые типы данных для ключей, даже объекты. Например:

let john = { name: "John" };

let visitsCountMap = new Map();

visitsCountMap.set(john, 123);

alert(visitsCountMap.get(john)); // 123

Использование объектов в качестве ключей – это одна из известных и часто применяемых возможностей объекта Map. При строковых ключах обычный объект Object может подойти, но для ключей-объектов – уже нет.

Чтобы сравнивать ключи, объект Map использует алгоритм [SameValueZero](https://tc39.github.io/ecma262/#sec-samevaluezero). Это почти такое же сравнение, что и ===, с той лишь разницей, что NaN считается равным NaN. Так что NaN также может использоваться в качестве ключа. Этот алгоритм не может быть заменён или модифицирован.

Каждый вызов map.set возвращает объект map, так что можно объединить вызовы в цепочку:

map.set("1", "str1")

.set(1, "num1")

.set(true, "bool1");

Для перебора коллекциии Map есть 3 метода:

* map.keys() – возвращает итерируемый объект по ключам,
* map.values() – возвращает итерируемый объект по значениям,
* map.entries() – возвращает итерируемый объект по парам вида [ключ, значение], этот вариант используется по умолчанию в for..of.

Например:

let recipeMap = new Map([

["огурец", 500],

["помидор", 350],

["лук", 50]

]);

// перебор по ключам (овощи)

for (let vegetable of recipeMap.keys()) {

alert(vegetable); // огурец, помидор, лук

}

// перебор по значениям (числа)

for (let amount of recipeMap.values()) {

alert(amount); // 500, 350, 50

}

// перебор по элементам в формате [ключ, значение]

for (let entry of recipeMap) {

alert(entry); // огурец,500 (и так далее)

}

В отличие от обычных объектов Object, в Map перебор происходит в том же порядке, в каком происходило добавление элементов. Кроме этого, Map имеет встроенный метод forEach, схожий со встроенным методом массивов Array:

// выполняем функцию для каждой пары (ключ, значение)

recipeMap.forEach((value, key, map) => {

alert(`${key}: ${value}`); // огурец: 500 и так далее

});

[**Создание Map из Object**](https://learn.javascript.ru/map-set#object-entries-map-iz-object)

Если есть обычный объект, и необходимо создать Map из него, то поможет встроенный метод [Object.entries(obj)](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/Object/entries), который получает объект и возвращает массив пар ключ-значение для него, как раз в этом формате. Поэтому можно создать Map из обычного объекта следующим образом:

let obj = {

name: "John",

age: 30

};

let map = new Map(Object.entries(obj));

Здесь Object.entries возвращает массив пар ключ-значение: [ ["name","John"], ["age", 30] ]. Это именно то, что нужно для создания Map.

**[Создание Object из Map](https://learn.javascript.ru/map-set" \l "object-fromentries-object-iz-map)**

Метод Object.fromEntries делает противоположное: получив массив пар вида [ключ, значение], он создаёт из них объект:

let prices = Object.fromEntries([

['banana', 1],

['orange', 2],

['meat', 4]

]);

// prices = { banana: 1, orange: 2, meat: 4 }

alert(prices.orange); // 2

Можно использовать Object.fromEntries, чтобы получить обычный объект из Map. Например, есть данные в Map, но их нужно передать в сторонний код, который ожидает обычный объект. Вот как это сделать:

let map = new Map();

map.set('banana', 1);

map.set('orange', 2);

map.set('meat', 4);

let obj = Object.fromEntries(map);

// obj = { banana: 1, orange: 2, meat: 4 }

alert(obj.orange); // 2

Вызов map.entries() возвращает массив пар ключ/значение, как раз в нужном формате для Object.fromEntries.

**[Set](https://learn.javascript.ru/map-set" \l "set)**

Объект Set – это особый вид коллекции: «множество» значений (без ключей), где каждое значение может появляться только один раз. Его основные методы:

* new Set(iterable) – создаёт Set, и если в качестве аргумента был предоставлен итерируемый объект (обычно это массив), то копирует его значения в новый Set.
* set.add(value) – добавляет значение (если оно уже есть, то ничего не делает), возвращает тот же объект set.
* set.delete(value) – удаляет значение, возвращает true если value было в множестве на момент вызова, иначе false.
* set.has(value) – возвращает true, если значение присутствует в множестве, иначе false.
* set.clear() – удаляет все имеющиеся значения.
* set.size – возвращает количество элементов в множестве.

Суть в том, что при повторных вызовах set.add() с одним и тем же значением ничего не происходит, за счёт этого как раз и получается, что каждое значение появляется один раз. Например, список посетителей:

let set = new Set();

let john = { name: "John" };

let pete = { name: "Pete" };

let mary = { name: "Mary" };

// считаем гостей, некоторые приходят несколько раз

set.add(john);

set.add(pete);

set.add(mary);

set.add(john);

set.add(mary);

// set хранит только 3 уникальных значения

alert(set.size); // 3

for (let user of set) {

alert(user.name); // John (потом Pete и Mary)

}

Альтернативой множеству Set может выступать массив для хранения гостей и дополнительный код для проверки уже имеющегося элемента с помощью [arr.find](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/Array/find). Но в этом случае будет хуже производительность, потому что arr.find проходит весь массив для проверки наличия элемента. Множество Set лучше оптимизировано для добавлений, оно автоматически проверяет на уникальность.

Можно перебрать содержимое объекта set как с помощью метода for..of, так и используя forEach:

let set = new Set(["апельсин", "яблоко", "банан"]);

for (let value of set) alert(value);

set.forEach((value, valueAgain, set) => {

alert(value);

});

Функция в forEach у Set имеет 3 аргумента: значение value, потом снова то же самое значение valueAgain, и целевой объект. Значение появляется в списке аргументов дважды. Это сделано для совместимости с объектом Map, в котором колбэк forEach имеет 3 аргумента. Выглядит странно, но в некоторых случаях может помочь легко заменить Map на Set и наоборот. Set имеет те же встроенные методы, что и Map:

* set.keys() – возвращает перебираемый объект для значений,
* set.values() – то же самое, что и set.keys(), присутствует для обратной совместимости с Map,
* set.entries() – возвращает перебираемый объект для пар вида [значение, значение], присутствует для обратной совместимости с Map.

**WeakMap и WeakSet**

Как извстно, движок JavaScript хранит значения в памяти до тех пор, пока они достижимы (то есть, эти значения могут быть использованы). Например:

let john = { name: "John" };

// объект доступен, переменная john -- это ссылка на него

// перепишем ссылку

john = null;

// объект будет удалён из памяти

Обычно свойства объекта, элементы массива или другой структуры данных считаются достижимыми и сохраняются в памяти до тех пор, пока эта структура данных содержится в памяти. Например, если поместить объект в массив, то до тех пор, пока массив существует, объект также будет существовать в памяти, несмотря на то, что других ссылок на него нет. Например:

let john = { name: "John" };

let array = [ john ];

john = null; // перезаписываем ссылку на объект

// объект john хранится в массиве, поэтому он не будет удалён

Аналогично, если используется объект как ключ в Map, то до тех пор, пока существует Map, также будет существовать и этот объект. Он занимает место в памяти и не может быть удалён сборщиком мусора. Например:

let john = { name: "John" };

let map = new Map();

map.set(john, "...");

john = null; // перезаписываем ссылку на объект

// объект john сохранён внутри объекта `Map`

3

WeakMap – принципиально другая структура в этом аспекте. Она не предотвращает удаление объектов сборщиком мусора, когда эти объекты выступают в качестве ключей. Первое его отличие от Map в том, что ключи в WeakMap должны быть объектами, а не примитивными значениями:

let weakMap = new WeakMap();

let obj = {};

weakMap.set(obj, "ok");

weakMap.set("test", "Whoops"); // Ошибка, потому что "test" не объект

Теперь, если использовать объект в качестве ключа и если больше нет ссылок на этот объект, то он будет удалён из памяти (и из объекта WeakMap) автоматически.

let john = { name: "John" };

let weakMap = new WeakMap();

weakMap.set(john, "...");

john = null;

// объект john удалён из памяти

Теперь john существует только как ключ в WeakMap и может быть удалён оттуда автоматически. WeakMap не поддерживает перебор и методы keys(), values(), entries(), так что нет способа взять все ключи или значения из неё. В WeakMap присутствуют только следующие методы:

* weakMap.get(key)
* weakMap.set(key, value)
* weakMap.delete(key)
* weakMap.has(key)

Такие ограничения связаны с особенностью технической реализации. Если объект станет недостижим (как объект john в примере выше), то он будет автоматически удалён сборщиком мусора. Но нет информации, в какой момент произойдет эта очистка. Решение о том, когда делать сборку мусора, принимает движок JavaScript. Он может посчитать необходимым как удалить объект прямо сейчас, так и отложить эту операцию, чтобы удалить большее количество объектов за раз позже. Так что технически количество элементов в коллекции WeakMap неизвестно. Движок может произвести очистку сразу или потом, или сделать это частично. По этой причине методы для доступа ко всем сразу ключам/значениям недоступны.

В основном, WeakMap используется в качестве дополнительного хранилища данных или кеширования, когда результат вызова функции должен где-то запоминаться («кешироваться») для того, чтобы дальнейшие её вызовы на том же объекте могли просто брать уже готовый результат, повторно используя его. Для хранения результатов можно использовать Map:

// cache.js

let cache = new Map();

// вычисляем и запоминаем результат

function process(obj) {

if (!cache.has(obj)) {

let result = /\* какие-то вычисления \*/ obj;

cache.set(obj, result);

}

return cache.get(obj);

}

// main.js

let obj = {/\* какой-то объект \*/};

let result1 = process(obj);

let result2 = process(obj); // ранее вычисленный результат взят из кеша

obj = null;

alert(cache.size); // 1, объект всё ещё в кеше и занимает память

Многократные вызовы process(obj) с тем же самым объектом в качестве аргумента ведут к тому, что результат вычисляется только в первый раз, а затем последующие вызовы берут его из кеша. Недостатком является то, что необходимо вручную очищать cache от ставших ненужными объектов. Но если использовать WeakMap вместо Map, то эта проблема исчезнет: закешированные результаты будут автоматически удалены из памяти сборщиком мусора.

// cache.js

let cache = new WeakMap();

function process(obj) {

if (!cache.has(obj)) {

let result = /\* вычисляем результат для объекта \*/ obj;

cache.set(obj, result);

}

return cache.get(obj);

}

// main.js

let obj = {/\* какой-то объект \*/};

let result1 = process(obj);

let result2 = process(obj);

obj = null;

// Нет возможности получить cache.size, так как это WeakMap,

// но он равен 0 или скоро будет равен 0

// Когда сборщик мусора удаляет obj, связанные с ним данные из кеша тоже удаляются

[**WeakSet**](https://learn.javascript.ru/weakmap-weakset#weakset)

Коллекция WeakSet ведёт себя похоже:

* Она аналогична Set, но можно добавлять в WeakSet только объекты (не примитивные значения).
* Объект присутствует в множестве только до тех пор, пока доступен где-то ещё.
* Как и Set, она поддерживает add, has и delete, но не size, keys() и не является перебираемой.

WeakSet тоже служит в качестве дополнительного хранилища. Но не для произвольных данных, а скорее для значений типа «да/нет». Присутствие во множестве WeakSet может что-то сказать об объекте. Например, можно добавлять пользователей в WeakSet для учёта тех, кто посещал наш сайт:

let visitedSet = new WeakSet();

let john = { name: "John" };

let pete = { name: "Pete" };

let mary = { name: "Mary" };

visitedSet.add(john); // John

visitedSet.add(pete); // Pete

visitedSet.add(john); // John

// visitedSet сейчас содержит двух пользователей

alert(visitedSet.has(john)); // true

alert(visitedSet.has(mary)); // false

john = null;

// структура данных visitedSet будет очищена автоматически

Наиболее значительным ограничением WeakMap и WeakSet является то, что их нельзя перебрать или взять всё содержимое. Это может доставлять неудобства, но не мешает WeakMap/WeakSet выполнять их главную задачу – быть дополнительным хранилищем данных для объектов, управляемых из каких-то других мест в коде.

1. **Деструктурирующее присваивание**

В JavaScript есть две чаще всего используемые структуры данных – это Object и Array. Объекты позволяют создавать одну сущность, которая хранит элементы данных по ключам, а массивы – хранить упорядоченные коллекции данных.

Но когда они передаются в функцию, то ей может понадобится не объект/массив целиком, а элементы по отдельности.

*Деструктурирующее присваивание* – это специальный синтаксис, который позволяет преобразовать массивы или объекты в кучу переменных, так как иногда они более удобны. Деструктуризация также прекрасно работает со сложными функциями, которые имеют много параметров, значений по умолчанию, и так далее.

[**Деструктуризация массива**](https://learn.javascript.ru/destructuring-assignment#destrukturizatsiya-massiva)

Пример:

let arr = ["Steve", "Jobs"]

// firstName=arr[0], surname=arr[1]

let [firstName, surname] = arr;

alert(firstName); // Steve

alert(surname); // Jobs

Теперь можно использовать переменные вместо элементов массива. Удобно использовать в сочетании со split или другими методами, возвращающими массив:

let [firstName, surname] = "Steve Jobs".split(' ');

Деструктурирующее присваивание не уничтожает массив. Оно вообще ничего не делает с правой частью присваивания, его задача – только скопировать нужные значения в переменные.

Ненужные элементы массива также могут быть отброшены через запятую:

// второй элемент не нужен

let [firstName, , title] = ["Julius", "Caesar", "Consul", "of the Roman Republic"];

alert( title ); // Consul

В примере выше второй элемент массива пропускается, а третий присваивается переменной title, оставшиеся элементы массива также пропускаются (так как для них нет переменных).

Можно использовать любой перебираемый объект, не только массивы:

let [a, b, c] = "abc"; // ["a", "b", "c"]

let [one, two, three] = new Set([1, 2, 3]);

Можно использовать что угодно с левой стороны. Например, можно присвоить свойству объекта:

let user = {};

[user.name, user.surname] = "Ilya Kantor".split(' ');

alert(user.name); // Ilya

Ранее рассматривался метод [Object.entries(obj)](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/Object/entries). Можем использовать его с деструктуризацией для цикличного перебора ключей и значений объекта:

let user = {

name: "John",

age: 30

};

for (let [key, value] of Object.entries(user)) {

alert(`${key}:${value}`); // name:John, then age:30

}

То же самое для map:

let user = new Map();

user.set("name", "John");

user.set("age", "30");

for (let [key, value] of user) {

alert(`${key}:${value}`); // name:John, then age:30

}

Если надо не просто получить первые значения, но и собрать все остальные – мы можем добавить ещё один параметр, который получает остальные значения, используя троеточие "...":

let [name1, name2, ...rest] = ["Julius", "Caesar", "Consul", "of the Roman Republic"];

alert(name1); // Julius

alert(name2); // Caesar

alert(rest[0]); // Consul

alert(rest[1]); // of the Roman Republic

alert(rest.length); // 2

Переменная rest является массивом из оставшихся элементов. Вместо rest можно использовать любое другое название переменной, и она должна находится на последнем месте в деструктурирующем присваивании.

Если в массиве меньше значений, чем в присваивании, то ошибки не будет. Отсутствующие значения считаются неопределёнными:

let [firstName, surname] = [];

alert(firstName); // undefined

alert(surname); // undefined

Если необходимо указать значения по умолчанию, то можно использовать =:

let [name = "Guest", surname = "Anonymous"] = ["Julius"];

alert(name); // Julius (из массива)

alert(surname); // Anonymous (значение по умолчанию)

Значения по умолчанию могут быть гораздо более сложными выражениями или даже функциями. Они выполняются, только если значения отсутствуют. Например, в в примере ниже используется функция prompt для указания двух значений по умолчанию. Но она будет запущена только для отсутствующего значения:

let [name = prompt('name?'), surname = prompt('surname?')] = ["Julius"];

alert(name); // Julius (из массива)

alert(surname); // результат prompt

[**Деструктуризация объекта**](https://learn.javascript.ru/destructuring-assignment#destrukturizatsiya-obekta)

Деструктурирующее присваивание также работает с объектами. Имена переменных и ключи должны совпадать. Синтаксис:

let {var1, var2} = {var1:…, var2:…}

У нас есть существующий объект с правой стороны, который необходимо разделить на переменные. Левая сторона содержит «шаблон» для соответствующих свойств. В простом случае это список названий переменных в {...}. Например:

let options = {

title: "Menu",

width: 100,

height: 200

};

let {title, width, height} = options;

alert(title); // Menu

alert(width); // 100

alert(height); // 200

Свойства options.title, options.width и options.height присваиваются соответствующим переменным. Порядок не имеет значения. Вот так тоже работает:

// изменён порядок в let {...}

let {height, width, title} = { title: "Menu", height: 200, width: 100 }

Шаблон с левой стороны может быть более сложным и определять соответствие между свойствами и переменными.

Если нужно присвоить свойство объекта переменной с другим названием, например, свойство options.width присвоить переменной *w*, то можно использовать двоеточие:

let options = {

title: "Menu",

width: 100,

height: 200

};

// { sourceProperty: targetVariable }

let {width: w, height: h, title} = options;

// width -> w, height -> h, title -> title

alert(title); // Menu

alert(w); // 100

alert(h); // 200

Двоеточие показывает «что : куда идёт». В примере выше свойство width сохраняется в переменную *w*, свойство height сохраняется в *h*, а title присваивается одноимённой переменной.

Для потенциально отсутствующих свойств можно установить значения по умолчанию, используя "=":

let options = {

title: "Menu"

};

let {width = 100, height = 200, title} = options;

alert(title); // Menu

alert(width); // 100

alert(height); // 200

Как и в случае с массивами, значениями по умолчанию могут быть любые выражения или даже функции. Они выполнятся, если значения отсутствуют.

[**Остаток объекта «…»**](https://learn.javascript.ru/destructuring-assignment#ostatok-obekta)

Если в объекте больше свойств, чем переменных, то можно использовать троеточие, так же как для массивов. В некоторых старых браузерах (IE) это не поддерживается, необходимо использовать полифилы. Например:

let options = {

title: "Menu",

height: 200,

width: 100

};

let {title, ...rest} = options;

// title="Menu", rest={height: 200, width: 100}

alert(rest.height); // 200

alert(rest.width); // 100

В примерах выше переменные были объявлены в присваивании: let {…} = {…}. Если использовать существующие переменные и не указывать let, то это не будет работать. Проблема в том, что JavaScript обрабатывает {...} в основном потоке кода (не внутри другого выражения) как блок кода. Такие блоки кода могут быть использованы для группировки операторов, например:

{

// блок кода

let message = "Hello";

// ...

alert( message );

}

Так что здесь JavaScript считает, что видит блок кода, отсюда и ошибка. На самом-то деле здесь деструктуризация. Чтобы показать JavaScript, что это не блок кода, можно заключить выражение в скобки (...):

let title, width, height;

({title, width, height} = {title: "Menu", width: 200, height: 100});

alert( title ); // Menu

[**Вложенная деструктуризация**](https://learn.javascript.ru/destructuring-assignment#vlozhennaya-destrukturizatsiya)

Если объект или массив содержит другие вложенные объекты или массивы, то можно использовать более сложные шаблоны с левой стороны, чтобы извлечь более глубокие свойства. В приведённом ниже коде options хранит другой объект в свойстве size и массив в свойстве items. Шаблон в левой части присваивания имеет такую же структуру, чтобы извлечь данные из них:

let options = {

size: {

width: 100,

height: 200

},

items: ["Cake", "Donut"],

extra: true

};

let {

size: {

width,

height

},

items: [item1, item2],

title = "Menu"

} = options;

alert(title); // Menu

alert(width); // 100

alert(height); // 200

alert(item1); // Cake

alert(item2); // Donut

Весь объект options, кроме свойства extra, присваивается в соответствующие переменные. В итоге есть width, height, item1, item2 и title со значением по умолчанию. Заметьте, что переменные для size и items отсутствуют, так как сразу использовали их содержимое.

[**Умные параметры функций**](https://learn.javascript.ru/destructuring-assignment#umnye-parametry-funktsiy)

Есть ситуации, когда функция имеет много параметров, большинство из которых не обязательны. Это особенно верно для пользовательских интерфейсов. Представьте себе функцию, которая создаёт меню. Она может иметь ширину, высоту, заголовок, список элементов и так далее. Вот так – плохой способ писать подобные функции:

function showMenu(title = "Untitled", width = 200, height = 100, items = []) {

// ...

}

В реальной жизни проблема заключается в том, как запомнить порядок всех аргументов. Обычно IDE помогают в этом, особенно если код хорошо документирован. Но есть и другая проблема, которая заключается в том, как вызвать функцию, когда большинство параметров передавать не надо, и значения по умолчанию вполне подходят. Разве что вот так:

// undefined там, где подходят значения по умолчанию

showMenu("My Menu", undefined, undefined, ["Item1", "Item2"])

Это выглядит плохо. И становится нечитаемым, когда имеется большое количество параметров. Решить эту проблему помогает деструктуризация. Можно передать параметры как объект, и функция немедленно деструктурирует его в переменные:

let options = {

title: "My menu",

items: ["Item1", "Item2"]

};

function showMenu({title = "Untitled", width = 200, height = 100, items = []}) {

alert( `${title} ${width} ${height}` ); // My Menu 200 100

alert( items ); // Item1, Item2

}

showMenu(options);

Также можно использовать более сложное деструктурирование с вложенными объектами и двоеточием:

let options = {

title: "My menu",

items: ["Item1", "Item2"]

};

function showMenu({

title = "Untitled",

width: w = 100,

height: h = 200,

items: [item1, item2]

}) {

alert( `${title} ${w} ${h}` ); // My Menu 100 200

alert( item1 ); // Item1

alert( item2 ); // Item2

}

showMenu(options);

Полный синтаксис – такой же, как для деструктурирующего присваивания:

function({

incomingProperty: varName = defaultValue

...

})

Тогда для объекта с параметрами, будет создана переменная varName для свойства с именем incomingProperty, по умолчанию равная defaultValue. Обратите внимание, что такое деструктурирование подразумевает, что в showMenu() будет обязательно передан аргумент. Если нужны все значения по умолчанию, то следует определить пустой объект:

showMenu({}); // все значения по умолчанию

showMenu(); // ошибка

Можно исправить это, сделав {} значением по умолчанию для всего объекта параметров:

function showMenu({ title = "Menu", width = 100, height = 200 } = {}) {

alert( `${title} ${width} ${height}` );

}

showMenu(); // Menu 100 200

В приведённом коде выше весь объект аргументов по умолчанию равен {}, поэтому всегда есть что-то, что можно деструктурировать.

1. **Дата и Время.**

Встроенный объект [Date](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/Date) содержит дату и время, а также предоставляет методы управления ими. Например, его можно использовать для хранения времени создания/изменения, для измерения времени или просто для вывода текущей даты.

[**Создание**](https://learn.javascript.ru/date#sozdanie)

Для создания нового объекта Date нужно вызвать конструктор new Date() с одним из следующих аргументов:

* new Date() без аргументов – создать объект Date с текущими датой и временем:

let now = new Date();

alert( now ); // показывает текущие дату и время

* new Date(milliseconds) – создать объект Date с временем, равным количеству миллисекунд (тысячная доля секунды), прошедших с 1 января 1970 года UTC+0.

// 0 соответствует 01.01.1970 UTC+0

let Jan01\_1970 = new Date(0);

alert( Jan01\_1970 );

// добавим 24 часа и получим 02.01.1970 UTC+0

let Jan02\_1970 = new Date(24 \* 3600 \* 1000);

alert( Jan02\_1970 );

Целое число, представляющее собой количество миллисекунд, прошедших с начала 1970 года, называется *таймстамп* (англ. timestamp). Это легковесное численное представление даты. Из таймстампа всегда можно получить дату с помощью new Date(timestamp) и преобразовать существующий объект Date в таймстамп, используя метод date.getTime() (см. ниже).

* new Date(datestring) – если аргумент всего один, и это строка, то из неё считывается дата. В примере ниже время не указано, поэтому оно ставится в полночь по Гринвичу и меняется в соответствии с временной зоной места выполнения кода, так что в результате можно получить Thu Jan 26 2017 11:00:00 GMT+1100 (восточноавстралийское время) или Wed Jan 25 2017 16:00:00 GMT-0800 (тихоокеанское время):

let date = new Date("2017-01-26");

alert(date);

* new Date(year, month, date, hours, minutes, seconds, ms) – создать объект Date с заданными компонентами в местной временной зоне. Обязательны только первые два аргумента.
* year должен состоять из четырёх цифр: значение 2013 корректно, 98 – нет.
* month начинается с 0 (январь) по 11 (декабрь).
* Параметр date здесь представляет собой день месяца. Если параметр не задан, то принимается значение 1.
* Если параметры hours/minutes/seconds/ms отсутствуют, их значением становится 0.

Например:

new Date(2011, 0, 1, 0, 0, 0, 0); // // 1 Jan 2011, 00:00:00

new Date(2011, 0, 1); // то же самое, так как часы и проч. равны 0

Минимальная точность – 1 мс (1/1000 секунды):

let date = new Date(2011, 0, 1, 2, 3, 4, 567);

alert( date ); // 1.01.2011, 02:03:04.567

[**Получение компонентов даты**](https://learn.javascript.ru/date#poluchenie-komponentov-daty)

Существуют методы получения года, месяца и т.д. из объекта Date:

* [getFullYear()](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/Date/getFullYear) – получить год (4 цифры)
* [getMonth()](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/Date/getMonth) – получить месяц, от 0 до 11.
* [getDate()](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/Date/getDate) – получить день месяца, от 1 до 31, что несколько противоречит названию метода.
* [getHours()](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/Date/getHours), [getMinutes()](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/Date/getMinutes), [getSeconds()](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/Date/getSeconds), [getMilliseconds()](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/Date/getMilliseconds) – получить, соответственно, часы, минуты, секунды или миллисекунды.

Многие интерпретаторы JavaScript реализуют нестандартный и устаревший метод getYear(), который порой возвращает год в виде двух цифр. Не используйте его. Если нужно значение года, используйте getFullYear().

* [getDay()](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/Date/getDay) – вернуть день недели от 0 (воскресенье) до 6 (суббота). Несмотря на то, что в ряде стран за первый день недели принят понедельник, в JavaScript начало недели приходится на воскресенье.

Все вышеперечисленные методы возвращают значения в соответствии с местной временной зоной. Однако существуют и их UTC-варианты, возвращающие день, месяц, год для временной зоны UTC+0: [getUTCFullYear()](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/Date/getUTCFullYear), [getUTCMonth()](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/Date/getUTCMonth), [getUTCDay()](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/Date/getUTCDay). Для их использования требуется после "get" подставить "UTC". Если ваша местная временная зона смещена относительно UTC, то следующий код покажет разные часы:

// текущая дата

let date = new Date();

// час в вашей текущей временной зоне

alert( date.getHours() );

// час во временной зоне UTC+0 (лондонское время без перехода на летнее время)

alert( date.getUTCHours() );

Помимо вышеприведённых методов, существуют два особых метода без UTC-варианта:

* [getTime()](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/Date/getTime) – для заданной даты возвращает таймстамп – количество миллисекунд, прошедших с 1 января 1970 года UTC+0.
* [getTimezoneOffset()](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/Date/getTimezoneOffset) – возвращает разницу в минутах между местной временной зоной и UTC:

// если вы во временной зоне UTC-1, то выводится 60

// если вы во временной зоне UTC+3, выводится -180

alert( new Date().getTimezoneOffset() );

[**Установка компонентов даты**](https://learn.javascript.ru/date#ustanovka-komponentov-daty)

Следующие методы позволяют установить компоненты даты и времени:

* [setFullYear(year, [month], [date])](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/Date/setFullYear)
* [setMonth(month, [date])](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/Date/setMonth)
* [setDate(date)](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/Date/setDate)
* [setHours(hour, [min], [sec], [ms])](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/Date/setHours)
* [setMinutes(min, [sec], [ms])](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/Date/setMinutes)
* [setSeconds(sec, [ms])](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/Date/setSeconds)
* [setMilliseconds(ms)](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/Date/setMilliseconds)
* [setTime(milliseconds)](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/Date/setTime) (устанавливает дату в виде целого количества миллисекунд, прошедших с 01.01.1970 UTC)

У всех этих методов, кроме setTime(), есть UTC-вариант, например: setUTCHours(). Некоторые методы могут устанавливать сразу несколько компонентов даты, например: setHours. Если какая-то компонента не указана, она не меняется. Пример:

let today = new Date();

today.setHours(0);

alert(today); // выводится сегодняшняя дата, значение часа будет 0

today.setHours(0, 0, 0, 0);

alert(today); // выводится сегодняшняя дата, время будет ровно 00:00:00.

[**Автоисправление даты**](https://learn.javascript.ru/date#avtoispravlenie-daty)

Автоисправление – это очень полезная особенность объектов Date. Можно устанавливать компоненты даты вне обычного диапазона значений, а объект сам себя исправит. Пример:

let date = new Date(2013, 0, 32); // 32 Jan 2013

alert(date); // 1st Feb 2013

Неправильные компоненты даты автоматически распределяются по остальным. Предположим, требуется увеличить дату «28 февраля 2016» на два дня. В зависимости от того, високосный это год или нет, результатом будет «2 марта» или «1 марта». Объект Date вычислит это сам, надо просто прибавить два дня:

let date = new Date(2016, 1, 28);

date.setDate(date.getDate() + 2);

alert( date ); // 1 Mar 2016

Эту возможность часто используют, чтобы получить дату по прошествии заданного отрезка времени. Например, получим дату «спустя 70 секунд с текущего момента»:

let date = new Date();

date.setSeconds(date.getSeconds() + 70);

alert( date ); // выводит правильную дату

Также можно установить нулевые или даже отрицательные значения. Например:

let date = new Date(2016, 0, 2); // 2 Jan 2016

date.setDate(1); // задать первое число месяца

alert( date );

// первый день месяца - это 1, так что выводится последнее число предыдущего месяца

date.setDate(0);

alert( date ); // 31 Dec 2015

[**Преобразование к числу, разность дат**](https://learn.javascript.ru/date#preobrazovanie-k-chislu-raznost-dat)

Если объект Date преобразовать в число, то получим таймстамп, по аналогии с date.getTime():

let date = new Date();

alert(+date); // количество миллисекунд, то же самое, что date.getTime()

Важный побочный эффект: даты можно вычитать, в результате получим разность в миллисекундах. Этот приём можно использовать для измерения времени:

let start = new Date(); // начинаем отсчёт времени

for (let i = 0; i < 100000; i++) {

let doSomething = i \* i \* i;

}

let end = new Date(); // заканчиваем отсчёт времени

alert( `Цикл отработал за ${end - start} миллисекунд` );

[**Date.now()**](https://learn.javascript.ru/date#date-now)

Если нужно просто померить время, для этого существует особый метод Date.now(), возвращающий текущий таймстамп. Семантически он эквивалентен new Date().getTime(), однако метод не создаёт промежуточный объект Date. Так что этот способ работает быстрее и не нагружает сборщик мусора. Данный метод используется из соображений удобства или когда важно быстродействие, например, при разработке игр на JavaScript или других специализированных приложений. Поэтому, предыдущий пример лучше переписать так:

let start = Date.now(); // количество миллисекунд с 1 января 1970 года

for (let i = 0; i < 100000; i++) {

let doSomething = i \* i \* i;

}

let end = Date.now(); // заканчиваем отсчёт времени

alert( `Цикл отработал за ${end - start} миллисекунд` ); // вычитаются числа, а не даты

Метод [Date.parse(str)](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/Date/parse) считывает дату из строки. Формат строки должен быть следующим: YYYY-MM-DDTHH:mm:ss.sssZ, где:

* YYYY-MM-DD – это дата: год-месяц-день.
* Символ "T" используется в качестве разделителя.
* HH:mm:ss.sss – время: часы, минуты, секунды и миллисекунды.
* Необязательная часть 'Z' обозначает временную зону в формате +-hh:mm. Если указать просто букву Z, то получим UTC+0.

Возможны и более короткие варианты, например, YYYY-MM-DD или YYYY-MM или даже YYYY.

Вызов Date.parse(str) обрабатывает строку в заданном формате и возвращает таймстамп (количество миллисекунд с 1 января 1970 года UTC+0). Если формат неправильный, возвращается NaN. Например:

let ms = Date.parse('2012-01-26T13:51:50.417-07:00');

alert(ms); // 1327611110417 (таймстамп)

Можно тут же создать объект new Date из таймстампа:

let date = new Date( Date.parse('2012-01-26T13:51:50.417-07:00') );

alert(date);

1. **Формат JSON.**

Допустим, есть сложный объект, и необходимо преобразовать его в строку, чтобы отправить по сети или просто вывести для логирования. Естественно, такая строка должна включать в себя все важные свойства. Можно реализовать преобразование следующим образом:

let user = {

name: "John",

age: 30,

toString() {

return `{name: "${this.name}", age: ${this.age}}`;

}

};

alert(user); // {name: "John", age: 30}

Но в процессе разработки добавляются новые свойства, старые свойства переименовываются и удаляются. Обновление такого toString каждый раз может стать проблемой. Можно попытаться перебрать свойства в нем, но если объект является сложным и имеет вложенные объекты в свойствах, тогда надо также осуществить их преобразование. Решением данной проеблемы является формат JSON.

**[JSON.stringify](https://learn.javascript.ru/json" \l "json-stringify)**

[JSON](https://ru.wikipedia.org/wiki/JSON) (JavaScript Object Notation) – это общий формат для представления значений и объектов. Его описание задокументировано в стандарте [RFC 4627](http://tools.ietf.org/html/rfc4627). Первоначально он был создан для JavaScript, но многие другие языки также имеют библиотеки, которые могут работать с ним. Таким образом, JSON легко использовать для обмена данными, когда клиент использует JavaScript, а сервер написан на Ruby/PHP/Java/любой другой язык.

JavaScript предоставляет методы:

* JSON.stringify для преобразования объектов в JSON.
* JSON.parse для преобразования JSON обратно в объект.

Например, преобразуем через JSON.stringify данные студента:

let student = {

name: 'John',

age: 30,

isAdmin: false,

courses: ['html', 'css', 'js'],

wife: null

};

let json = JSON.stringify(student);

alert(typeof json); // получили строку

alert(json);

/\* выведет объект в формате JSON:

{

"name": "John",

"age": 30,

"isAdmin": false,

"courses": ["html", "css", "js"],

"wife": null

}

\*/

Метод JSON.stringify(student) берет объект и преобразует его в строку. Полученная строка json называется JSON-форматированным или сериализованным объектом. Можно отправить его по сети или поместить в обычное хранилище данных.

Объект в формате JSON имеет несколько важных отличий от объектного литерала:

* Строки используют двойные кавычки. Никаких одинарных кавычек или обратных кавычек в JSON. Так 'John' становится "John".
* Имена свойств объекта также заключаются в двойные кавычки. Это обязательно. Так age:30 становится "age":30.

JSON.stringify может быть применён и к примитивам.

JSON поддерживает следующие типы данных:

* объекты { ... },
* массивы [ ... ],
* примитивы: строки, числа, логические значения true/false, null.

Например:

alert( JSON.stringify(1) ) // 1

alert( JSON.stringify('test') ) // "test"

alert( JSON.stringify(true) ); // true

alert( JSON.stringify([1, 2, 3]) ); // [1,2,3]

JSON является независимой от языка спецификацией для данных, поэтому JSON.stringify пропускает некоторые специфические свойства объектов JavaScript. А именно:

* Свойства-функции (методы).
* Символьные свойства.
* Свойства, содержащие undefined.

let user = {

sayHi() { // пропущено

alert("Hello");

},

[Symbol("id")]: 123, // пропущено

something: undefined // пропущено

};

alert( JSON.stringify(user) ); // {} (пустой объект)

Вложенные объекты поддерживаются и конвертируются автоматически. Например:

let meetup = {

title: "Conference",

room: {

number: 23,

participants: ["john", "ann"]

}

};

alert( JSON.stringify(meetup) );

/\* вся структура преобразована в строку:

{

"title":"Conference",

"room":{"number":23,"participants":["john","ann"]},

}

\*/

Важное ограничение: не должно быть циклических ссылок. Например:

let room = {

number: 23

};

let meetup = {

title: "Conference",

participants: ["john", "ann"]

};

meetup.place = room;

room.occupiedBy = meetup;

JSON.stringify(meetup); // Ошибка: Преобразование цикличной структуры в JSON

Здесь преобразование завершается неудачно из-за циклической ссылки: room.occupiedBy ссылается на meetup, и meetup.place ссылается на room:

Полный синтаксис JSON.stringify:

let json = JSON.stringify(value[, replacer, space])

* value – значение для кодирования,
* replacer – массив свойств для кодирования или функция соответствия function(key, value),
* space – дополнительное пространство (отступы), используемое для форматирования.

В большинстве случаев JSON.stringify используется только с первым аргументом. Но если нужно настроить процесс замены, например, отфильтровать циклические ссылки, то можно использовать второй аргумент JSON.stringify. Если передать ему массив свойств, будут закодированы только эти свойства. Например:

let room = {

number: 23

};

let meetup = {

title: "Conference",

participants: [{name: "John"}, {name: "Alice"}],

place: room // meetup ссылается на room

};

room.occupiedBy = meetup; // room ссылается на meetup

alert( JSON.stringify(meetup, ['title', 'participants']) );

// {"title":"Conference","participants":[{},{}]}

Список свойств применяется ко всей структуре объекта. Так что свойство внутри participants объекты пустые, потому что name нет в списке. Включим в список все свойства, кроме room.occupiedBy, из-за которого появляется цикличная ссылка:

let room = {

number: 23

};

let meetup = {

title: "Conference",

participants: [{name: "John"}, {name: "Alice"}],

place: room // meetup ссылается на room

};

room.occupiedBy = meetup; // room ссылается на meetup

alert( JSON.stringify(meetup, ['title', 'participants', 'place', 'name', 'number']) );

/\*

{

"title":"Conference",

"participants":[{"name":"John"},{"name":"Alice"}],

"place":{"number":23}

}

\*/

Теперь все, кроме occupiedBy, сериализовано. Но список свойств довольно длинный. В качестве replacer можно использовать функцию, а не массив. Функция будет вызываться для каждой пары (key, value) и должна возвращать «заменённое» значение, которое будет использоваться вместо исходного. Или undefined, чтобы пропустить значение. В рассматриваемом примере можно вернуть value «как есть» для всего, кроме occupiedBy. Чтобы игнорировать occupiedBy, код ниже возвращает undefined:

let room = {

number: 23

};

let meetup = {

title: "Conference",

participants: [{name: "John"}, {name: "Alice"}],

place: room // meetup ссылается на room

};

room.occupiedBy = meetup; // room ссылается на meetup

alert( JSON.stringify(meetup, function replacer(key, value) {

alert(`${key}: ${value}`);

return (key == 'occupiedBy') ? undefined : value;

}));

/\* пары ключ:значение, которые приходят в replacer:

: [object Object]

title: Conference

participants: [object Object],[object Object]

0: [object Object]

name: John

1: [object Object]

name: Alice

place: [object Object]

number: 23

\*/

Функция replacer получает каждую пару ключ/значение, включая вложенные объекты и элементы массива. И она применяется рекурсивно. Значение this внутри replacer – это объект, который содержит текущее свойство. При первом вызове ему передаётся специальный «объект-обёртка»: {"": meetup}. Другими словами, первая (key, value) пара имеет пустой ключ, а значением является целевой объект. Вот почему первая строка из примера выше будет ":[object Object]".

Идея состоит в том, чтобы дать как можно больше возможностей replacer – у него есть возможность проанализировать и заменить/пропустить даже весь объект целиком, если это необходимо.

Третий аргумент в JSON.stringify(value, replacer, spaces) – это количество пробелов, используемых для удобного форматирования. Ранее все JSON-форматированные объекты не имели отступов и лишних пробелов. Аргумент spacer используется исключительно для вывода в удобочитаемом виде. Ниже spacer = 2 указывает JavaScript отображать вложенные объекты в несколько строк с отступом в 2 пробела внутри объекта:

let user = {

name: "John",

age: 25,

roles: {

isAdmin: false,

isEditor: true

}

};

alert(JSON.stringify(user, null, 2));

/\* отступ в 2 пробела:

{

"name": "John",

"age": 25,

"roles": {

"isAdmin": false,

"isEditor": true

}

}

\*/

/\* для JSON.stringify(user, null, 4) результат содержал больше отступов:

{

"name": "John",

"age": 25,

"roles": {

"isAdmin": false,

"isEditor": true

}

}

\*/

Параметр spaces применяется для логирования и красивого вывода.

**[JSON.parse](https://learn.javascript.ru/json" \l "json-parse)**

Чтобы декодировать JSON-строку, нужен другой метод с именем [JSON.parse](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/JSON/parse). Синтаксис:

let value = JSON.parse(str, [reviver]);

* str – JSON для преобразования в объект,
* reviver – необязательная функция, которая будет вызываться для каждой пары (ключ, значение) и может преобразовывать значение.

Например:

let numbers = "[0, 1, 2, 3]";

numbers = JSON.parse(numbers);

alert( numbers[1] ); // 1

Или для вложенных объектов:

let user = '{ "name": "John", "age": 35, "isAdmin": false, "friends": [0,1,2,3] }';

user = JSON.parse(user);

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

JSON может быть настолько сложным, насколько это необходимо, объекты и массивы могут включать другие объекты и массивы. Но они должны быть в том же JSON-формате. Типичные ошибки в написанном от руки JSON:

let json = `{

name: "John", // Ошибка: имя свойства без кавычек

"surname": 'Smith', // Ошибка: одинарные кавычки в значении (должны быть двойными)

'isAdmin': false // Ошибка: одинарные кавычки в ключе (должны быть двойными)

"birthday": new Date(2000, 2, 3), // Ошибка: не допускается конструктор "new", только значения.

"friends": [0,1,2,3] // Здесь все в порядке

}`;

Кроме того, JSON не поддерживает комментарии. Добавление комментария в JSON делает его недействительным.

Существует ещё один формат [JSON5](http://json5.org/), который поддерживает ключи без кавычек, комментарии и т.д. Но это самостоятельная библиотека, а не спецификация языка.

1. **Intl: интернационализация в JavaScript.**

Общая проблема строк, дат, чисел в JavaScript заключается в том, что они не учитывают языуовые и временные особенностей стран, где находится посетитель. В частности:

* Строки: при сравнении сравниваются коды символов, а это неправильно, к примеру, в русском языке оказывается, что "ё" > "я" и "а" > "Я", хотя всем известно, что я – последняя буква алфавита и это она должна быть больше любой другой.
* Даты: в разных странах принята разная запись дат. Где-то пишут 31.12.2014 (Россия), а где-то 12/31/2014 (США), где-то иначе.
* Числа: в одних странах выводятся цифрами, в других – иероглифами, длинные числа разделяются где-то пробелом, где-то запятой.

Все современные браузеры, кроме IE10- (но есть библиотеки и для него) поддерживают стандарт [ECMA 402](http://www.ecma-international.org/ecma-402/1.0/ECMA-402.pdf), предназначенный решить эти проблемы навсегда.

[Основные объекты](https://learn.javascript.ru/intl" \l "osnovnye-obekty):

* Intl.Collator – умеет правильно сравнивать и сортировать строки.
* Intl.DateTimeFormat – умеет форматировать дату и время в соответствии с нужным языком.
* Intl.NumberFormat – умеет форматировать числа в соответствии с нужным языком.

[**Локаль**](https://learn.javascript.ru/intl#lokal)

Локаль – первый и самый важный аргумент всех методов, связанных с интернационализацией. Локаль описывается строкой из трёх компонентов, которые разделяются дефисом: код языка, код способа записи, код страны.

На практике не всегда указаны три, обычно меньше:

* ru – русский язык, без уточнений.
* en-GB – английский язык, используемый в Англии (GB).
* en-US – английский язык, используемый в США (US).
* zh-Hans-CN – китайский язык (zh), записываемый упрощённой иероглифической письменностью (Hans), используемый в Китае.

Также через суффикс -u-\* можно указать расширения локалей, например "th-TH-u-nu-thai" – тайский язык (th), используемый в Тайланде (TH), с записью чисел тайскими буквами (๐, ๑, ๒, ๓, ๔, ๕, ๖, ๗, ๘, ๙).

Стандарт, который описывает локали – [RFC 5464](http://tools.ietf.org/html/rfc5646), языки описаны в [IANA language registry](http://www.iana.org/assignments/language-subtag-registry/language-subtag-registry). Все методы принимают локаль в виде строки или массива, содержащего несколько локалей в порядке предпочтения. Если локаль не указана или undefined – берётся локаль по умолчанию, установленная в окружении (браузере).

[**Подбор локали localeMatcher**](https://learn.javascript.ru/intl#podbor-lokali-localematcher)

localeMatcher – вспомогательная настройка, которую тоже можно везде указать, она определяет способ подбора локали, если желаемая недоступна. У него два значения:

* "lookup" – означает простейший порядок поиска путём обрезания суффикса, например, zh-Hans-CN → zh-Hans → zh → локаль по умолчанию.
* "best fit" – использует встроенные алгоритмы и предпочтения браузера (или другого окружения) для выбора подходящей локали.

Если локалей несколько, например, ["zh-Hans-CN", "ru-RU"] то localeMatcher пытается подобрать наиболее подходящую локаль для первой из списка (китайская), если не получается – переходит ко второй (русской) и так далее. Если ни одной не нашёл, например, на компьютере не совсем поддерживается ни китайский ни русский, то используется локаль по умолчанию. Как правило, "best fit" является здесь наилучшим выбором.

[**Строки, Intl.Collator**](https://learn.javascript.ru/intl#intl-collator)

Синтаксис:

var collator = new Intl.Collator([locales, [options]])

Параметры:

* Locales – локаль, одна или массив в порядке предпочтения.
* Options – объект с дополнительными настройками:
* localeMatcher – алгоритм выбора подходящей локали.
* usage – цель сравнения: сортировка "sort" или поиск "search", по умолчанию "sort".
* sensitivity – чувствительность: какие различия в символах учитывать, а какие – нет, варианты:
* base – учитывать только разные символы, без диакритических знаков и регистра, например: а ≠ б, е = ё, а = А.
* accent – учитывать символы и диакритические знаки, например: а ≠ б, е ≠ ё, а = А.
* case – учитывать символы и регистр, например: а ≠ б, е = ё, а ≠ А.
* variant – учитывать всё: символ, диакритические знаки, регистр, например: а ≠ б, е ≠ ё, а ≠ А, используется по умолчанию.
* ignorePunctuation – игнорировать знаки пунктуации: true/false, по умолчанию false.
* numeric – использовать ли численное сравнение: true/false, если true, то будет 12 > 2, иначе 12 < 2.
* caseFirst – в сортировке должны идти первыми прописные или строчные буквы, варианты: "upper" (прописные), lower (строчные) или false (стандартное для локали, также является значением по умолчанию). Не поддерживается IE11-.

В подавляющем большинстве случаев подходят стандартные параметры, то есть options указывать не нужно. Использование:

var result = collator.compare(str1, str2);

Результат compare имеет значение 1 (больше), 0 (равно) или -1 (меньше). Например:

var collator = new Intl.Collator();

alert( "ёжик" > "яблоко" ); // true

alert( collator.compare("ёжик", "яблоко") ); // -1

Выше были использованы полностью стандартные настройки. Они различают регистр символа, но это различие можно убрать, если настроить чувствительность sensitivity:

var collator = new Intl.Collator();

alert( collator.compare("ЁжиК", "ёжик") ); // 1

var collator = new Intl.Collator(undefined, {

sensitivity: "accent"

});

alert( collator.compare("ЁжиК", "ёжик") ); // 0

[**Даты, Intl.DateTimeFormat**](https://learn.javascript.ru/intl#intl-datetimeformat)

Синтаксис:

var formatter = new Intl.DateTimeFormat([locales, [options]])

Первый аргумент – такой же, как и в Collator, а в объекте options можно определить, какие именно части даты показывать (часы, месяц, год…) и в каком формате. Полный список свойств options:

| **Свойство** | **Описание** | **Возможные значения** | **По умолчанию** |
| --- | --- | --- | --- |
| localeMatcher | Алгоритм подбора локали | lookup,best fit | best fit |
| formatMatcher | Алгоритм подбора формата | basic, best fit | best fit |
| hour12 | Включать ли время в 12-часовом формате | true -- 12-часовой формат, false -- 24-часовой |  |
| timeZone | Временная зона | Временная зона, например Europe/Moscow | UTC |
| weekday | День недели | narrow, short, long |  |
| era | Эра | narrow, short, long |  |
| year | Год | 2-digit, numeric | undefined или numeric |
| month | Месяц | 2-digit, numeric, narrow, short, long | undefined или numeric |
| day | День | 2-digit, numeric | undefined или numeric |
| hour | Час | 2-digit, numeric |  |
| minute | Минуты | 2-digit, numeric |  |
| second | Секунды | 2-digit, numeric |  |
| timeZoneName | Название таймзоны (нет в IE11) | short, long |  |

Все локали обязаны поддерживать следующие наборы настроек:

* weekday, year, month, day, hour, minute, second
* weekday, year, month, day
* year, month, day
* year, month
* month, day
* hour, minute, second

Если указанный формат не поддерживается, то настройка formatMatcher задаёт алгоритм подбора наиболее близкого формата: basic – по [стандартным правилам](http://www.ecma-international.org/ecma-402/1.0/#BasicFormatMatcher) и best fit – по умолчанию, на усмотрение окружения (браузера). Использование:

var dateString = formatter.format(date);

Например:

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

var formatter = new Intl.DateTimeFormat("ru");

alert( formatter.format(date) ); // 31.12.2014

var formatter = new Intl.DateTimeFormat("en-US");

alert( formatter.format(date) ); // 12/31/2014

Длинная дата, с настройками:

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

var formatter = new Intl.DateTimeFormat("ru", {

weekday: "long",

year: "numeric",

month: "long",

day: "numeric"

});

alert( formatter.format(date) ); // среда, 31 декабря 2014 г.

Только время:

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

var formatter = new Intl.DateTimeFormat("ru", {

hour: "numeric",

minute: "numeric",

second: "numeric"

});

alert( formatter.format(date) ); // 12:30:00

[**Числа: Intl.NumberFormat**](https://learn.javascript.ru/intl#chisla-intl-numberformat)

Форматтер Intl.NumberFormat умеет красиво форматировать не только числа, но и валюту, а также проценты. Синтаксис:

var formatter = new Intl.NumberFormat([locales[, options]]);

formatter.format(number); // форматирование

Параметры, как и раньше – локаль и опции. Список опций:

|  |  |  |  |
| --- | --- | --- | --- |
| **Свойство** | **Описание** | **Возможные значения** | **По умолчанию** |
| localeMatcher | Алгоритм подбора локали | lookup, best fit | best fit |
| style | Стиль форматирования | decimal, percent, currency | decimal |
| currency | Алфавитный код валюты | См. [Список кодов валюты](http://www.currency-iso.org/en/home/tables/table-a1.html), например USD |  |
| currencyDisplay | Показывать валюту в виде кода, локализованного символа или локализованного названия | code, symbol, name | symbol |
| useGrouping | Разделять ли цифры на группы | true, false | true |
| minimumIntegerDigits | Минимальное количество цифр целой части | от `1` до `21` | 21 |
| minimumFractionDigits | Минимальное количество десятичных цифр | от 0 до 20 | для чисел и процентов 0, для валюты зависит от кода. |
| maximumFractionDigits | Максимальное количество десятичных цифр | от minimumFractionDigits до 20. | для чисел max(minimumFractionDigits, 3), для процентов 0, для валюты зависит от кода. |
| minimumSignificantDigits | Минимальное количество значимых цифр | от 1 до 21 | 1 |
| maximumSignificantDigits | Максимальное количество значимых цифр | от minimumSignificantDigitsдо 21 | minimumSignificantDigits |

Пример без опций:

var formatter = new Intl.NumberFormat("ru");

alert( formatter.format(1234567890.123) ); // 1 234 567 890,123

С ограничением значимых цифр (важны только первые 3):

var formatter = new Intl.NumberFormat("ru", {

maximumSignificantDigits: 3

});

alert( formatter.format(1234567890.123) ); // 1 230 000 000

С опциями для валюты:

var formatter = new Intl.NumberFormat("ru", {

style: "currency",

currency: "GBP"

});

alert( formatter.format(1234.5) ); // 1 234,5 £

С двумя цифрами после запятой:

var formatter = new Intl.NumberFormat("ru", {

style: "currency",

currency: "GBP",

minimumFractionDigits: 2

});

alert( formatter.format(1234.5) ); // 1 234,50 £

[**Методы в Date, String, Number**](https://learn.javascript.ru/intl#metody-v-date-string-number)

Методы форматирования также поддерживаются в обычных строках, датах, числах:

String.prototype.localeCompare(that [, locales [, options]]) – сравнивает строку с другой, с учётом локали, например:

var str = "ёжик";

alert( str.localeCompare("яблоко", "ru") ); // -1

Date.prototype.toLocaleString([locales [, options]]) – форматирует дату в соответствии с локалью, например:

var date = new Date(2014, 11, 31, 12, 00);

alert( date.toLocaleString("ru", { year: 'numeric', month: 'long' }) ); // Декабрь 2014

Date.prototype.toLocaleDateString([locales [, options]]) – то же, что и выше, но опции по умолчанию включают в себя год, месяц, день.

Date.prototype.toLocaleTimeString([locales [, options]]) – то же, что и выше, но опции по умолчанию включают в себя часы, минуты, секунды.

Number.prototype.toLocaleString([locales [, options]]) – форматирует число, используя опции Intl.NumberFormat.

Все эти методы при запуске создают соответствующий объект Intl.\* и передают ему опции, можно рассматривать их как укороченные варианты вызова.

1. **Регулярные выражения.**

Регулярные выражения – это шаблоны используемые для сопоставления последовательностей символов в строках. В JavaScript, регулярные выражения также являются объектами. Эти шаблоны используются в методах [exec](https://developer.mozilla.org/en-US/docs/JavaScript/Reference/Global_Objects/RegExp/exec) и [test](https://developer.mozilla.org/en-US/docs/JavaScript/Reference/Global_Objects/RegExp/test) объекта[RegExp](https://developer.mozilla.org/en-US/docs/JavaScript/Reference/Global_Objects/RegExp), а также [match](https://developer.mozilla.org/en-US/docs/JavaScript/Reference/Global_Objects/String/match), [replace](https://developer.mozilla.org/en-US/docs/JavaScript/Reference/Global_Objects/String/replace),[search](https://developer.mozilla.org/en-US/docs/JavaScript/Reference/Global_Objects/String/search), и [split](https://developer.mozilla.org/en-US/docs/JavaScript/Reference/Global_Objects/String/split) объекта [String](https://developer.mozilla.org/en-US/docs/JavaScript/Reference/Global_Objects/String).

**Создание регулярного выражения**

Регулярное выражение можно создать двумя способами:

1. Используя литерал регулярного выражения, например:

let re = /ab+c/;

Литералы регулярных выражений вызывают предварительную компиляцию регулярного выражения при анализе скрипта. Если ваше регулярное выражение постоянно, то пользуйтесь им чтобы увеличить производительность.

1. Вызывая функцию конструктор объекта [RegExp](https://developer.mozilla.org/en-US/docs/JavaScript/Reference/Global_Objects/RegExp), например:

let re = new RegExp("ab+c");

Использование конструктора влечет за собой компиляцию регулярного выражения во время исполнения скрипта. Данный способ можно использовать, если известно, что выражение будет изменяться или не известен шаблон заранее. Например, при получении его из стороннего источника, при пользовательском вводе.

**Написание шаблона регулярного выражения**

Шаблон регулярного выражения состоит из обычных символов, например /abc/, или комбинаций обычных и специальных символов, например /ab\*c/ или /Chapter (\d+)\.\d\*/. Последний пример включает в себя скобки, которые используются как "запоминающий механизм".

Простые шаблоны используются для нахождения прямого соответствия в тексте. Например, шаблон /abc/ соответствует комбинации символов в строке только когда символы 'abc' встречаются вместе и в том же порядке. Такое сопоставление произойдет в строке "Hi, do you know your abc's?" и "The latest airplane designs evolved from slabcraft." В обоих случаях сопоставление произойдет с подстрокой 'abc'. Сопоставление не произойдет в строке "Grab crab", потому что она не содержит подстроку 'abc'.

В случае, когда поиск соответствия требует чего-то большего, чем прямое сопоставление, например, нахождение последовательности символов 'b' или нахождение пробела, шаблон включает в себя специальные символы. Например, шаблон /ab\*c/ соответствует любой комбинации символов, в которой за 'a' следует ноль или более символов 'b' (\* означает ноль или более вхождений предыдущего символа), за которыми сразу же следует символ 'c'. В строке "cbbabbbbcdebc," этому шаблону сопоставляется подстрока 'abbbbc'.

В следующей таблице приводится полный список специальных символов регулярных выражений с их описаниями.

|  | |
| --- | --- |
| **Символ** | **Значение** |
| [\](https://developer.mozilla.org/ru/docs/Web/JavaScript/Guide/Regular_Expressions" \l "special-backslash) | Одно из следующего:   * Для символов обычно обрабатываемых буквально, означает что следующий символ является специальным и не должен интерпретироваться буквально. * Например, /b/ сопоставляется символу 'b'. Добавляя слэш перед b, т.е используя /\b/, символ становится специальным символом, означающим границу слова. * Для символов обычно обрабатываемых особым образом означает, что следующий символ не является специальным и должен интерпретироваться буквально. * Например, \* является специальным символом, сопоставляемым 0 или более повторений предыдущего символа; например, /a\*/ означает соответствие 0 или более символов а. Для буквальной интерпретации \*, поставьте перед ней обратный слэш; например, /a\\*/ соответствует 'a\*'. * Также не забудьте заэкранировать сам \ при его использовании в записи new RegExp("pattern") поскольку \ также является экранирующим символом в обычных строках. |
| [^](https://developer.mozilla.org/ru/docs/Web/JavaScript/Guide/Regular_Expressions" \l "special-caret) | Соответствует началу ввода. Если установлен флаг многострочности, также производит сопоставление непосредственно после переноса строки.  Например, /^A/ не соответствует 'A' в "an A", но соответствует 'A' в "An E".  Этот символ имеет другое значение при появлении в начале шаблона набора символов. Например, /[^a-z\s]/ соответствует 'I' в "I have 3 sisters". |
| [$](https://developer.mozilla.org/ru/docs/Web/JavaScript/Guide/Regular_Expressions" \l "special-dollar) | Соответствует концу ввода. Если установлен битовый флаг многострочности, также сопоставляется содержимому до переноса строки.  Например, /t$/ не соответствует 't' в строке "eater", но соответствует строке "eat". |
| [\*](https://developer.mozilla.org/ru/docs/Web/JavaScript/Guide/Regular_Expressions" \l "special-asterisk) | Соответствует предыдущему символу повторенному 0 или более раз. Эквивалентно {0,}.  Например, /bo\*/ соответствует 'boooo' в "A ghost booooed" и 'b' в "A bird warbled", но не в "A goat grunted". |
| [+](https://developer.mozilla.org/ru/docs/Web/JavaScript/Guide/Regular_Expressions" \l "special-plus) | Соответствует предыдущему символу повторенному 1 или более раз. Эквивалентно {1,}. Например, /a+/ соответствует 'a' в "candy" и всем символам 'a' в "caaaaaaandy". |
| [?](https://developer.mozilla.org/ru/docs/Web/JavaScript/Guide/Regular_Expressions" \l "special-questionmark) | Соответствует предыдущему символу повторенному  0 или 1 раз. Эквивалентно {0,1}.  Например, /e?le?/ соответствует 'el' в "angel" и 'le' в "angle" а также 'l' в "oslo".  Если использован сразу после квалификаторов \*, +, ?, or {}, делает квалификатор "нежадным" (соответствующим минимальному количеству символов), в отличие от режима по умолчанию, являющимся "жадным" (соответствующим максимальному числу символов). Например, используя /\d+/ не глобальное сопоставление "123abc" возвращает "123", если использовать /\d+?/, только "1" будет возвращена.  Также используется в упреждающих утверждениях (assertions), описанных в строках x(?=y) и x(?!y) данной таблицы. |
| [.](https://developer.mozilla.org/ru/docs/Web/JavaScript/Guide/Regular_Expressions" \l "special-dot) | (десятичная точка) соответствует любому символу кроме переноса строки.  Например, /.n/ соответствует 'an' и 'on' в "nay, an apple is on the tree", но не 'nay'. |
| [(x)](https://developer.mozilla.org/ru/docs/Web/JavaScript/Guide/Regular_Expressions" \l "special-capturing-parentheses) | Соответствует 'x' и запоминает это соответствие. Это называется захватывающие скобки.  Например, /(foo)/ соответствует 'foo' в "foo bar." Сопоставленная строка может быть получена из элементов результирующего массива [1], ..., [n]. |
| [(?:x)](https://developer.mozilla.org/ru/docs/Web/JavaScript/Guide/Regular_Expressions" \l "special-non-capturing-parentheses) | Соответствует 'x' но не запоминает соответствие. Это называется не-захватывающие скобки. Сопоставленная строка не может быть получена из элементов результирующего массива [1], ..., [n]. |
| [x(?=y)](https://developer.mozilla.org/ru/docs/Web/JavaScript/Guide/Regular_Expressions" \l "special-lookahead) | Соответствует 'x' только если за 'x' следует 'y'. Это называется упреждение.  Например, /Jack(?=Sprat)/ соответствует 'Jack' только если за ним следует 'Sprat'. /Jack(?=Sprat|Frost)/ соответствует 'Jack' только если за ним следует 'Sprat' или 'Frost'. Тем не менее, ни 'Sprat' ни 'Frost' не являются частью сопоставленного результата. |
| [x(?!y)](https://developer.mozilla.org/ru/docs/Web/JavaScript/Guide/Regular_Expressions" \l "special-negated-look-ahead) | Соответствует 'x' только если за 'x' не следует 'y'. Это называется отрицательное упреждение. Например, /\d+(?!\.)/ соответствует числу только если за ним не следует десятичная точка. Регулярное выражение /\d+(?!\.)/.exec("3.141") сопоставит '141' но не '3.141'. |
| [x|y](https://developer.mozilla.org/ru/docs/Web/JavaScript/Guide/Regular_Expressions" \l "special-or) | Соответствует либо 'x' либо 'y'.  Например, /green|red/ соответствует 'green' в "green apple" и 'red' в "red apple." |
| [{n}](https://developer.mozilla.org/ru/docs/Web/JavaScript/Guide/Regular_Expressions" \l "special-quantifier) | n - положительное целое. Соответствует ровно n вхождениям предыдущего символа. Например, /a{2}/ не соответствует 'a' в "candy," но соответствует всем а в "caandy," первым двум а в "caaandy." |
| [{n,m}](https://developer.mozilla.org/ru/docs/Web/JavaScript/Guide/Regular_Expressions" \l "special-quantifier-range) | m и n - положительные целые. Соответствует как минимум n и максимум m вхождениям предыдущего символа. При m=n=0 пропускается.  Например, /a{1,3}/ ничему не соответствует в строке "cndy", символу 'a' в "candy," двум а в "caandy," и трем первым а в "caaaaaaandy". Отметим, что при сопоставлении "caaaaaaandy", совпадает "aaa", хотя изначальная строка содержит больше а. |
| [[xyz]](https://developer.mozilla.org/ru/docs/Web/JavaScript/Guide/Regular_Expressions" \l "special-character-set) | Набор символов. Соответствует любому символу из перечисленных. Можно указать диапазон символов используя тире. Специальные символы (как точка (.) и звездочка (\*)) не имеют специального значения внутри такого набора.Их не надо экранировать. Экранирование работает так же.  Например, [abcd] эквивалентна [a-d]. Они соответствуют 'b' в "brisket" и 'c' в "city". /[a-z.]+/ и /[\w.]+/ обе соответствуют всему в "test.i.ng". |
| [[^xyz]](https://developer.mozilla.org/ru/docs/Web/JavaScript/Guide/Regular_Expressions" \l "special-negated-character-set) | Инвертированный или дополняющий набор символов. Это означает соответствие всему, что не в скобках. Можно указать диапазон символов с помощью тире. Все, что действует в обычном наборе символов, действует и здесь. Например, [^abc] эквивалентно [^a-c]. Они соответствуют изначально 'r' в "brisket" и 'h' в "chop." |
| [[\b]](https://developer.mozilla.org/ru/docs/Web/JavaScript/Guide/Regular_Expressions" \l "special-backspace) | Соответствует бэкспейсу (U+0008). (Не путать с \b.) |
| [\b](https://developer.mozilla.org/ru/docs/Web/JavaScript/Guide/Regular_Expressions" \l "special-word-boundary) | Соответствует границе слова. Граница слова соответствует позиции где за символом слова не следует другой символ слова или предшествует ему. Отметим, что граница слова не включается в соответствие. Другими словами, длина сопоставленной границы слова равна нулю. (Не путать с [\b].)  Примеры: /\bmoo/ соответствует 'moo' в слове "moon" ; /oo\b/ не соответствует 'oo' в слове "moon", поскольку за 'oo' следует символ 'n' , являющимся символом слова; /oon\b/ соответствует 'oon' в слове "moon", поскольку 'oon' является окончанием строки, и таким образом, за этими символами не следует другой символ слова; /\w\b\w/ никогда не будет ничему соответствовать, поскольку за символом слова никогда не может следовать и граница слова, и символ слова. |
| [\B](https://developer.mozilla.org/ru/docs/Web/JavaScript/Guide/Regular_Expressions" \l "special-non-word-boundary) | Соответствует несловообразующей границе. Несловообразующая граница соответствует позиции, в которой предыдущий и следующий символы являются символами одного типа: либо оба должны быть словообразующими символами, либо несловообразующими. Начало и конец строки считаются несловообразующими символами.  Например, /\B../ соответствует 'oo' в слове "noonday" (, а /y\B./ соответствует 'ye' в "possibly yesterday." |
| [\c](https://developer.mozilla.org/ru/docs/Web/JavaScript/Guide/Regular_Expressions" \l "special-control)*[X](https://developer.mozilla.org/ru/docs/Web/JavaScript/Guide/Regular_Expressions" \l "special-control)* | Где *X* является символом случайного выбора из последовательности от А до Я. Соответствует управляющему символу в строке.  Например, /\cM/ соответствует control-M (U+000D) в строке. |
| [\d](https://developer.mozilla.org/ru/docs/Web/JavaScript/Guide/Regular_Expressions" \l "special-digit) | Соответствует цифровому символу. Эквивалентно выражению [0-9].  Например, /\d/ or /[0-9]/ соотвествует  '2' в "B2 is the suite number." |
| [\D](https://developer.mozilla.org/ru/docs/Web/JavaScript/Guide/Regular_Expressions" \l "special-non-digit) | Соответствует любому нецифровому символу. Эквивалентно выражению [^0-9].  Например, /\D/ or /[^0-9]/ соответствует 'B' в предложении "B2 is the suite number." |
| [\f](https://developer.mozilla.org/ru/docs/Web/JavaScript/Guide/Regular_Expressions" \l "special-form-feed) | Соответствует символу прогона страницы (U+000C). Особый символ управления печатью. |
| [\n](https://developer.mozilla.org/ru/docs/Web/JavaScript/Guide/Regular_Expressions" \l "special-line-feed) | Соответствует символу перевода строки (U+000A). |
| [\r](https://developer.mozilla.org/ru/docs/Web/JavaScript/Guide/Regular_Expressions" \l "special-carriage-return) | Соответствует символу возврата каретки (U+000D). |
| [\s](https://developer.mozilla.org/ru/docs/Web/JavaScript/Guide/Regular_Expressions" \l "special-white-space) | Соответствует одиночному символу пустого пространства, включая пробел, табуляция, прогон страницы, перевод строки. Эквивалентен [ \f\n\r\t\v​\u00A0\u1680​\u180e\u2000​\u2001\u2002​\u2003\u2004​\u2005\u2006​\u2007\u2008​\u2009\u200a​\u2028\u2029​\u2028\u2029​\u202f\u205f​\u3000].  Например, /\s\w\*/ совпадает с ' bar' в "foo bar." |
| [\S](https://developer.mozilla.org/ru/docs/Web/JavaScript/Guide/Regular_Expressions" \l "special-non-white-space) | Соответствует одиночному символу непустого пространства. Эквивалентен [^ \f\n\r\t\v​\u00A0\u1680​\u180e\u2000-\u200a​​\u2028\u2029​\u202f\u205f​\u3000].  Например, /\S\w\*/ совпадает с 'foo' в "foo bar." |
| [\t](https://developer.mozilla.org/ru/docs/Web/JavaScript/Guide/Regular_Expressions" \l "special-tab) | Соответствует символу горизонтальной табуляции (U+0009). |
| [\v](https://developer.mozilla.org/ru/docs/Web/JavaScript/Guide/Regular_Expressions" \l "special-vertical-tab) | Соответствует символу вертикальной табуляции (U+000B). |
| [\w](https://developer.mozilla.org/ru/docs/Web/JavaScript/Guide/Regular_Expressions" \l "special-word) | Соответствует любому цифробуквенному символу включая нижнее подчеркивание. Эквивалентен [A-Za-z0-9\_].  Например, /\w/ совпадает с 'a' в "apple," '5' в "$5.28," и '3' в "3D." |
| [\W](https://developer.mozilla.org/ru/docs/Web/JavaScript/Guide/Regular_Expressions" \l "special-non-word) | Соответствует любому не цифробуквенному символу. Равносилен [^A-Za-z0-9\_].  Например, /\W/ or /[^A-Za-z0-9\_]/ совпадает с '%' в "50%." |
| [\](https://developer.mozilla.org/ru/docs/Web/JavaScript/Guide/Regular_Expressions" \l "special-backreference)*[n](https://developer.mozilla.org/ru/docs/Web/JavaScript/Guide/Regular_Expressions" \l "special-backreference)* | Где *n* это положительное целое, обратная ссылка на последнюю найденную подстроку, соответствующую *n* , заключенную в круглые скобки в регуляном выражении (считая левые скобки).  Например, /apple(,)\sorange\1/ соответствует 'apple, orange,' в "apple, orange, cherry, peach." |
| [\0](https://developer.mozilla.org/ru/docs/Web/JavaScript/Guide/Regular_Expressions" \l "special-null) | Соответствует символу NULL (U+0000). Не следует ставить за ним другой цифровой символ, поскольку \0<digits> является восьмеричной экранирующей последовательностью. |
| [\xhh](https://developer.mozilla.org/ru/docs/Web/JavaScript/Guide/Regular_Expressions" \l "special-hex-escape) | Соответствует символам кода hh (две шестнадцатиричные цифры) |
| [\uhhhh](https://developer.mozilla.org/ru/docs/Web/JavaScript/Guide/Regular_Expressions" \l "special-unicode-escape) | Соответствует символам кода hhhh (четыре шестнадцатиричные цифры). |

Экранирование пользовательского ввода, соответствующего буквенной строке внутри регулярного выражения, может быть достигнуто простой заменой:

function escapeRegExp(string){

return string.replace(/[.\*+?^${}()|[\]\\]/g, '\\$&');

}

**Использование скобок**

Скобки вокруг любой части регулярного выражения означают что эта часть совпадаемой подстроки будет запомнена. Например, паттерн /Chapter (\d+)\.\d\*/ включает в себя дополнительные экранирующие и специальные символы и указывает на то, что часть шаблона должна быть запомнена. Он точно соответствует символам слова 'Chapter ', за которыми следует один или более цифровых символов (\d означает любой цифровой символ, а '+' означает 1 или более раз), за которым следует десятичная точка (сама по себе являющаяся специальным символом; предшествующий десятичной точке слэш ' \' означает, что паттерн должен искать литеральный символ '.'), после которой следует любой цифровой символ 0 или более раз ('\d' обозначает цифровой символ, '\*' обозначает 0 или более раз). Кроме того, круглые скобки используются для запоминания первых же совпавших цифровых символов.

Этот шаблон будет найден во фразе "Open Chapter 4.3, paragraph 6" и цифра '4' будет запомнена. Но он не будет найден во фразе "Chapter 3 and 4", поскольку эта строка не имеет точки после цифры '3'.

Для того, чтобы сопоставить подстроку без вызова совпавшей части для запоминания, внутри круглых скобок необходимо предварить паттерн сочетанием символов ' ?:'. Например, шаблон (?:\d+) будет соответствовать одному или более цифровому символу, но не запомнит совпавших символов.

**Работа с регулярными выражениями**

Регулярные выражения используются в методах test и exec объекта RegExp и с методами match, replace, search, и split объекта String. Эти методы подробно объясняются в [Справочнике JavaScript](https://developer.mozilla.org/en-US/docs/JavaScript/Reference).

| Методы пользующиеся регулярными выражениями: | |
| --- | --- |
| **Метод** | **Описание** |
| [exec](https://developer.mozilla.org/en-US/docs/JavaScript/Reference/Global_Objects/RegExp/exec) | Метод RegExp, который выполняет поиск совпадения в  строке. Он возвращает массив данных. |
| [test](https://developer.mozilla.org/en-US/docs/JavaScript/Reference/Global_Objects/RegExp/test) | Метод RegExp, который тестирует совпадение в строке. Возвращет либо истину либо ложь. |
| [match](https://developer.mozilla.org/en-US/docs/JavaScript/Reference/Global_Objects/String/match) | Метод String, который выполняет поиск совпадения в строке. Он возвращет массив данных либо null если совпадения отсутствуют. |
| [match](https://developer.mozilla.org/en-US/docs/JavaScript/Reference/Global_Objects/String/match)All | Метод String, который выполняет поиск совпадений в строке и возвращает итератор, который в свою очередь возвращает все совпадающие группы одну за другой. |
| [search](https://developer.mozilla.org/en-US/docs/JavaScript/Reference/Global_Objects/String/search) | Метод String, который тестирует на совпадение в строке. Он возвращет индекс совпадения, или -1 если совпадений не будет найдено. |
| [replace](https://developer.mozilla.org/en-US/docs/JavaScript/Reference/Global_Objects/String/replace) | Метод String, который выполняет поиск совпадения в строке, и заменяет совпавшую подстроку другой подстрокой переданной как аргумент в этот метод. |
| [split](https://developer.mozilla.org/en-US/docs/JavaScript/Reference/Global_Objects/String/split) | Метод String, который использует регулярное выражение или фиксированую строку чтобы разбить строку на массив подстрок. |

Чтобы просто узнать есть ли в строке что либо соотвествующее шаблону, воспользуйтесь методами test или search; а чтобы получить больше информации пользуйтесь методами exec или match (хотя эти метода работают медленее), которые вернут массив и обновленные свойства объекта ассоцированного регулярного выражения, а также предопределенного объекта RegExp регулярного выражения. Если совпадений нет, метод exec вернет null (который сконвертируется в  false).

В следующем примере, скрипт использует метод exec чтобы найти совпадения в строке:

let myRe = /d(b+)d/g;

let myArray = myRe.exec("cdbbdbsbz");

Если не нужен доступ к свойствам регулярного выражения, то альтернативный способ получить myArray можно так:

let myArray = /d(b+)d/g.exec("cdbbdbsbz");

Если надо сконструировать регулярное выражение из строки, другой способ сделать это приведен ниже:

let myRe = new RegExp("d(b+)d", "g");

let myArray = myRe.exec("cdbbdbsbz");

С помощью этих скриптов, поиск совпадения завершается и возвращает массив и обновленные ствойства, показанные в следующей таблице:

|  | | | |
| --- | --- | --- | --- |
| **Объект** | **Свойство или индекс** | **Описание** | **В этом примере.** |
| myArray |  | Совпавшая строка и все запомненные подстроки. | ["dbbd", "bb"] |
| index | Индекс совпавшей подстроки (индекс начинается с нуля). | 1 |
| input | Исходная строка. | "cdbbdbsbz" |
| [0] | Последние совпавшие символы. | "dbbd" |
| myRe | lastIndex | Индекс с которого начнется след. поиск совпадения. (Это свойство определяется только если регулярное выражение использует параметр g, описанный в [Advanced Searching With Flags](https://developer.mozilla.org/ru/docs/Web/JavaScript/Guide/Regular_Expressions#Advanced_Searching_With_Flags).) | 5 |
| source | Текст шаблона. Обновляется в момент создания регулярного выражения, а не во время выполнения. | "d(b+)d" |

Как показано во втором варианте этого примера, можно использовать регулярное выражение, созданное при помощи инициализатора объекта, без присваивания его переменной. Таким образом, если используется данная форма записи без присваивания переменной, то в процессе дальнейшего использования нельзя получить доступ к свойствам данного регулярного выражения. Например:

var myRe = /d(b+)d/g;

var myArray = myRe.exec("cdbbdbsbz");

console.log("The value of lastIndex is " + myRe.lastIndex);

Этот скрипт выведет:

The value of lastIndex is 5

Однако, следующий скрипт:

var myArray = /d(b+)d/g.exec("cdbbdbsbz");

console.log("The value of lastIndex is " + /d(b+)d/g.lastIndex);

выведет

The value of lastIndex is 0

Совпадения /d(b+)d/g в двух случаях являются разными объектами регулярного выражения и, следовательно, имеют различные значения для свойства lastIndex. Если необходим доступ к свойствам объекта, созданного при помощи инициализатора, то надо сначала присвоить его переменной.

**Использование скобочных выражений для нахождения подстрок.**

Использование скобок в шаблоне регулярного выражения повлечет "запоминание" совпавшей подстроки. Для примера, /a(b)c/ вызовет совпадение 'abc' и запомнит 'b'. Чтобы получить совпадения скобочного выражения используйте Array elements [1], ..., [n].

Число возможных скобочных подстрок неограничено. Возвращаемый массив содержит все полученные совпадения, удовлетворяющие выражению в скобках. Следующий пример показывает, как использовать скобочные выражения для нахождения подстрок.

Следующий скрипт использует метод [replace()](https://developer.mozilla.org/en-US/docs/JavaScript/Reference/Global_Objects/String/replace), чтобы поменять местами слова (символы) в строке. Для замены текста скрипт использует обозначения $1 и $2 для обозначения первого и второго совпадения скобочного выражения.

let re = /(\w+)\s(\w+)/;

let str = "John Smith";

let newstr = str.replace(re, "$2, $1");

console.log(newstr);

Выведет "Smith, John".

**Расширенный поиск с флагами**

Регулярные выражения имеют четыре опциональных флага, которые делают возможным глобальный и регистронезависимый поиск. Флаги могут использоваться самостоятельно или вместе в любом порядке, а также могут являться частью регулярного выражения.

| **Флаги регулярных выражений** | |
| --- | --- |
| **Flag** | **Description** |
| g | Глобальный поиск. |
| i | Регистронезависимый поиск. |
| m | Многострочный поиск. |
| y | Выполняет поиск начиная с символа, который находится на позиции свойства lastindex текущего регулярного выражения. |

Чтобы использовать флаги в шаблоне регулярного выражения используйте следующий синтаксис:

var re = /pattern/flags;

или

var re = new RegExp("pattern", "flags");

Обратите внимание, что флаги являются неотъемлемой частью регулярного выражения. Флаги не могут быть добавлены или удалены позднее. Для примера, re = /\w+\s/g создаёт регулярное выражение, которое ищет один или более символов, после котрых следует пробел и ищет данное совпадение на протяжении всей строки.

var re = /\w+\s/g;

var str = "fee fi fo fum";

var myArray = str.match(re);

console.log(myArray);

Выведет ["fee ", "fi ", "fo "]. В этом примере можно заменить строку:

var re = /\w+\s/g;

на следующую:

var re = new RegExp("\\w+\\s", "g");

и получить тот же результат.

Флаг m используется, чтобы входная строка рассматривалась как многострочная. Если флаг m используется, то ^ и $ вызовет совпадение в начале или конце любой строки в строке ввода вместо начала или конца вводимой строки целиком.

Следующий пример иллюстрирует формирование регулярного выражения и использование string.split() и string.replace(). Он очищает неправильно сформатированную исходную строку, которая содержит имена в неправильном порядке (имя идет первым) разделенные пробелами, табуляцией и одной точкой с запятой. В конце, изменяется порядок следования имен (фамилия станет первой) и сортируется список.

var names = "Harry Trump ;Fred Barney; Helen Rigby ; Bill Abel ; Chris Hand ";

var output = ["---------- Original String\n", names + "\n"];

var pattern = /\s\*;\s\*/;

var nameList = names.split(pattern);

pattern = /(\w+)\s+(\w+)/;

var bySurnameList = [];

output.push("---------- After Split by Regular Expression");

var i, len;

for (i = 0, len = nameList.length; i < len; i++){

output.push(nameList[i]);

bySurnameList[i] = nameList[i].replace(pattern, "$2, $1");

}

output.push("---------- Names Reversed");

for (i = 0, len = bySurnameList.length; i < len; i++){

output.push(bySurnameList[i]);

}

bySurnameList.sort();

output.push("---------- Sorted");

for (i = 0, len = bySurnameList.length; i < len; i++){

output.push(bySurnameList[i]);

}

output.push("---------- End");

console.log(output.join("\n"));

В следующем примере, ожидается что пользователь введет телефонный номер и требуется проверить правильность символов, набранных пользователем. Когда пользователь нажмет кнопку "Check", скрипт проверит правильность введеного номера. Если номер правильный (совпадает с символами, определенными в регулярном выражении), то скрипт покажет сообщение благодарности для пользователя и подтвердит номер. Если нет, то скрипт проинформирует пользователя, что телефонный номер неправильный.

Внутри незахватывающих скобок (?:, регуляное выражение ищет три цифры \d{3} ИЛИ |  открывающую скобку \(, затем три цифры \d{3}, затем закрывающую скобку \), (закрывающая незахватывающая скобка )), затем тире, слэш, или десятичная точка, и когда это выражение найдено, запоминает символ ([-\/\.]),  следующие за ним и запомненные три цифры \d{3}, следующее соответствие тире, слэша или десятичной точки \1, и следующие четыре цифры \d{4}.

Регулярное выражение ищет сначала 0 или одну открывающую скобку \(?, затем три цифры \d{3}, затем 0 или одну закрывающую скобку \)?, потом одно тире, слеш или точка и когда найдет это, запомнит символ([-\/\.]), след. три цифры \d{3}, followed by the remembered match of a dash, forward slash, or decimal point \1, followed by four digits \d{4}.

Событие "Изменить" активируется, когда пользователь подтвердит ввод значения регулярного выражения, нажав клавишу "Enter".

<!DOCTYPE html>

<html>

<head>

<meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">

<meta http-equiv="Content-Script-Type" content="text/javascript">

<script type="text/javascript">

var re = /\(?\d{3}\)?([-\/\.])\d{3}\1\d{4}/;

function testInfo(phoneInput){

var OK = re.exec(phoneInput.value);

if (!OK)

window.alert(RegExp.input + " isn't a phone number with area code!");

else

window.alert("Thanks, your phone number is " + OK[0]);

}

</script>

</head>

<body>

<p>Enter your phone number (with area code) and then click "Check".

<br>The expected format is like ###-###-####.</p>

<form action="#">

<input id="phone"><button onclick="testInfo(document.getElementById('phone'));">Check</button>

</form>

</body>

</html>

autoPreviousNext("JSGChapters");