Брендан Эйх

API — это специальный интерфейс программы или приложения (библиотеки классов и процедур), с помощью которого одна программа/приложение может взаимодействовать с другой.

Аудио и видео API позволяет взаимодействовать с мультимедиа.

API Canvas и WebGL может создавать анимированные 2D и 3D-графики.

API-интерфейс DOM (Document Object Model) предназначено для манипулирования с HTML и CSS.

API геолокации позволяет запрашивать доступ к текущему местоположению пользователя.

В Netscape за 15 дней

Процесс преобразования исходного кода в машинный целиком

процесс немедленного, построчного считывания кода и возвращение результата таким же путем

/\* document.write('Hello!'); \*/

/\* document.body.innerHTML = '<h1>Hello from body!</h>'; \*/

const result = prompt('Как тебя зовут?');

debugger;

/\* const result = confirm('Ты человек?'); \*/

document.body.innerHTML = 'Привет, ' + result;

<p>

Привет из разметки!

</p>

**Взаимодействие с пользователем: alert, prompt, confirm**

Материал на этой странице устарел, поэтому скрыт из оглавления сайта.

Более новая информация по этой теме находится на странице <https://learn.javascript.ru/alert-prompt-confirm>.

В этом разделе мы рассмотрим базовые UI операции: alert, prompt и confirm, которые позволяют работать с данными, полученными от пользователя.

**[alert](https://learn.javascript.ru/uibasic" \l "alert)**

Синтаксис:

alert(сообщение)

alert выводит на экран окно с сообщением и приостанавливает выполнение скрипта, пока пользователь не нажмёт «ОК».

alert( "Привет" );

Окно сообщения, которое выводится, является *модальным окном*. Слово «модальное» означает, что посетитель не может взаимодействовать со страницей, нажимать другие кнопки и т.п., пока не разберётся с окном. В данном случае – пока не нажмёт на «OK».

**[prompt](https://learn.javascript.ru/uibasic" \l "prompt)**

Функция prompt принимает два аргумента:

result = prompt(title, default);

Она выводит модальное окно с заголовком title, полем для ввода текста, заполненным строкой по умолчанию default и кнопками OK/CANCEL.

Пользователь должен либо что-то ввести и нажать OK, либо отменить ввод кликом на CANCEL или нажатием Esc на клавиатуре.

**Вызов prompt возвращает то, что ввёл посетитель – строку или специальное значение null, если ввод отменён.**

**Safari 5.1+ не возвращает null**

Единственный браузер, который не возвращает null при отмене ввода – это Safari. При отсутствии ввода он возвращает пустую строку. Предположительно, это ошибка в браузере.

Если нам важен этот браузер, то пустую строку нужно обрабатывать точно так же, как и null, т.е. считать отменой ввода.

Как и в случае с alert, окно prompt модальное.

var years = prompt('Сколько вам лет?', 100);

alert('Вам ' + years + ' лет!')

**Всегда указывайте default**

Второй параметр может отсутствовать. Однако при этом IE вставит в диалог значение по умолчанию "undefined".

Запустите этот код в IE, чтобы понять о чём речь:

var test = prompt("Тест");

Поэтому рекомендуется *всегда* указывать второй аргумент:

var test = prompt("Тест", ''); // <-- так лучше

**[confirm](https://learn.javascript.ru/uibasic" \l "confirm)**

Синтаксис:

result = confirm(question);

confirm выводит окно с вопросом question с двумя кнопками: OK и CANCEL.

**Результатом будет true при нажатии OK и false – при CANCEL(Esc).**

Например:

var isAdmin = confirm("Вы - администратор?");

alert( isAdmin );

**[Особенности встроенных функций](https://learn.javascript.ru/uibasic" \l "osobennosti-vstroennyh-funktsiy)**

Конкретное место, где выводится модальное окно с вопросом – обычно это центр браузера, и внешний вид окна выбирает браузер. Разработчик не может на это влиять.

С одной стороны – это недостаток, так как нельзя вывести окно в своём, особо красивом, дизайне.

С другой стороны, преимущество этих функций по сравнению с другими, более сложными методами взаимодействия, которые мы изучим в дальнейшем – как раз в том, что они очень просты.

Это самый простой способ вывести сообщение или получить информацию от посетителя. Поэтому их используют в тех случаях, когда простота важна, а всякие «красивости» особой роли не играют.

**[Резюме](https://learn.javascript.ru/uibasic" \l "rezyume)**

* alert выводит сообщение.
* prompt выводит сообщение и ждёт, пока пользователь введёт текст, а затем возвращает введённое значение или null, если ввод отменён (CANCEL/Esc).
* confirm выводит сообщение и ждёт, пока пользователь нажмёт «OK» или «CANCEL» и возвращает true/false.

# Разбираемся с “поднятием” (hoisting) в JavaScript

[Stas Bagretsov](https://medium.com/@stasonmars?source=post_page-----7d2d27bc51f1--------------------------------)

[Stas Bagretsov](https://medium.com/@stasonmars?source=post_page-----7d2d27bc51f1--------------------------------)

[Feb 24, 2018·9 min read](https://medium.com/@stasonmars/%D1%80%D0%B0%D0%B7%D0%B1%D0%B8%D1%80%D0%B0%D0%B5%D0%BC%D1%81%D1%8F-%D1%81-%D0%BF%D0%BE%D0%B4%D0%BD%D1%8F%D1%82%D0%B8%D0%B5%D0%BC-hoisting-%D0%B2-javascript-7d2d27bc51f1?source=post_page-----7d2d27bc51f1--------------------------------)

Это перевод статьи [Understanding Hoisting in JavaScript](https://scotch.io/tutorials/understanding-hoisting-in-javascript)

https://miro.medium.com/max/368/1*fu2A46halqhXoHeVeVWjkw.jpeg

👉[**Мой Твиттер**](https://twitter.com/stassonmars)— там много из мира фронтенда, да и вообще поговорим🖖. Подписывайтесь, будет интересно: ) ✈️

https://miro.medium.com/max/60/1*WPv0CIP7s5G1XZqh8jkYuQ.jpeg?q=20

https://miro.medium.com/max/184/1*WPv0CIP7s5G1XZqh8jkYuQ.jpeg

В этом руководстве вы изучите то, как срабатывает всеми извезтный механизм “поднятия” в JavaScript. Ну или в оригинальном названии hoisting. Однако, перед тем как углубиться в детали, давайте узнаем что это вообще такое и как оно работает на самом деле.

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

Как следствие, это означает то, что совершенно неважно где были объявлены функция или переменные, все они передвигаются вверх своей области видимости, вне зависимости от того локальная она или же глобальная.

Стоит отметить то, что механизм “поднятия” передвигает только объявления функции или переменной. Назначения переменным остаются на своих местах.

В общем, если вы когда-либо удивлялись, почему вы могли вызывать функции перед тем, как они написаны в коде, то читайте дальше.

## Undefined vs ReferenceError

Перед тем, как мы начнем серьёзно углубляться в этот вопрос, давайте проясним несколько вещей.

console.log(typeof variable); **// Выводит: undefined**

Это приведет нас к первому заключению. В JavaScript, необъявленной переменной при выполнении кода назначается значение undefined , а так же и тип undefined.

Вторым заключением будет:

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

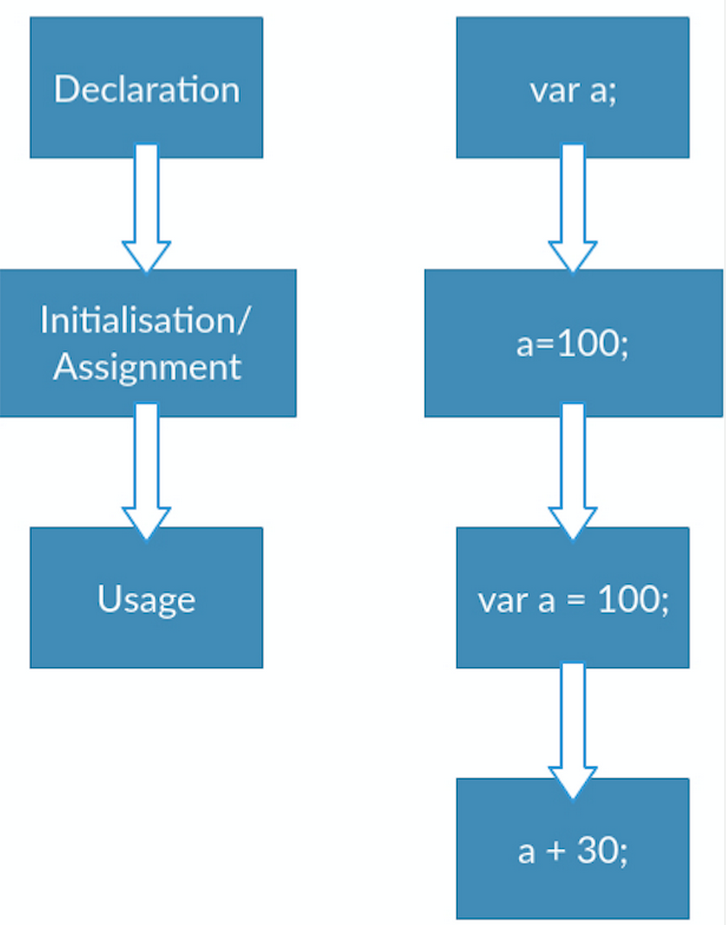
В JavaScript, ReferenceError появляется при попытке доступа к предварительно необъявленной переменной.

Поведение JavaScript при работе с переменными становится довольно утонченным делом из-за «поднятия». Мы увидим это более детально в следующих параграфах.

## ****«Поднятие» переменных****

Ниже вы видите цикл работы JavaScript, показывающий последовательность, в которой происходит объявление и инициализация переменных.

https://miro.medium.com/max/48/1*pUqWuCzYCP79tOm4gGzXIw.png?q=20



Однако, в JavaScript мы можем объявлять и инициализировать наши переменные одновременно, как в этом ну просто самом распространенном примере:

var a = 100;

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

Как упоминалось ранее, все переменные и объявления функций поднимаются вверх своей области видимости. Мне также стоит добавить, что объявление переменных происходит перед выполнением кода.

Но однако, необъявленные переменные не существуют до тех пор, пока код назначающий их не будет выполнен. Следовательно, указание значения для необъявленной переменной, тут же создаёт её как глобальную переменную, когда назначение будет выполнено. **Это говорит о том, что все необъявленные переменные это по факту глобальные переменные.**

Чтобы продемонстрировать это поведение, давайте посмотрим на следующий код.

function hoist() {  
 a = 20;  
 var b = 100;  
}  
  
hoist();  
  
console.log(a);**/\*Доступ как к глобальной переменной вне функции hoist()Выводит: 20\*/**console.log(b);**/\*Как только b была назначена, она заключена в рамки области видимости функции hoist(). Что означает то, что мы не можем вывести её за рамки функции.Вывод: ReferenceError: b is not defined\*/**

Так как это одна из причуд работы JavaScript с переменными, рекомендуется всегда объявлять их, вне зависимости от их положения в коде, в функции они или в глобальной области видимости.

Это ясно указывает на то, как движок JavaScript должен с ними работать во время выполнения кода.

## ****ES5****

**var**

Областью видимости переменной, объявленной через var, является её настоящий контекст выполнения. Это подходит как и для замыкания функции, так и для переменных объявленных вне любой функции, то есть глобальных. Давайте посмотрим на несколько примеров, чтобы понять, что это означает.

## ****Глобальные переменные****

console.log(hoist); **// Вывод: undefined**  
  
var hoist = 'The variable has been hoisted.';

Мы ожидали, что результат в логе будет: ReferenceError: hoist is not defined, но вместо этого нам вывело undefined.

Почему так произошло?

JavaScript **«поднял» объявление переменной**. Вот как это выглядит для движка JavaScript:

var hoist;  
  
console.log(hoist); **// Вывод: undefined**  
hoist = 'The variable has been hoisted.';

Поэтому мы можем использовать переменные перед тем как мы объявим их. Тем не менее, в этом вопросе нужно быть аккуратными, так как «поднятая» переменная инициализируется со значением undefined. Лучшим вариантом, как говорилось выше, было бы объявить и инициализировать нашу переменную перед её непосредственным использованием.

**Переменные в области видимости функции**

Как мы видели до этого, **переменные в глобальной области видимости поднимаются вверх**. Дальше давайте посмотрим на то, как ведут себя переменные при «поднятии» в функциях .

function hoist() {  
 console.log(message);  
 var message='Hoisting is all the rage!'  
}  
  
hoist();

Подумайте, что нам выдаст эта функция? Если вы предположили — undefined, то вы оказались правы. Если нет, то не переживайте, мы скоро доберемся до сути и объясним почему так.

Вот как движок видит код выше:

function hoist() {  
 **var message;**  
 console.log(message);  
 message='Hoisting is all the rage!'  
}  
  
hoist(); **// Вывод: undefined**

Объявление переменной var message, область видимости которой заканчивается в функции hoist(), «поднимается» наверх функции.

Чтобы избегать таких ловушек, нам всегда нужно убеждаться в том, что мы объявляем и инициализируем переменные перед их использованием.

function hoist() {  
 var message='Hoisting is all the rage!'  
 return (message);  
}  
  
hoist(); **// Вывод: Hoisting is all the rage!**

## ****Strict Mode или «Строгий режим»****

Спасибо такой полезной штуковине как, strict-mode в es5 версии JavaScript, с помощью которой вы можете быть внимательнее при объявлении переменных. Включая strict mode, мы включаем некий ограниченный вариант JavaScript, который не будет возиться с использованием переменных перед их объявлением.

**Запуск кода в strict mode:**

Устраняет некоторые скрытые ошибки в JavaScript, изменяя их на явную выдачу ошибок, которые будут в последствии выданы движком.

Устраняет ошибки, которые затрудняют движкам JavaScript выполнять оптимизацию.

Запрещает некоторый синтаксис, который с большой долей вероятности будет уже идти из коробки в будущих версиях JavaScript.

Мы включаем «**строгий режим**», заранее указывая в нашем файле или функции следующее:

'use strict';  
  
**// или**  
"use strict";

Давайте протестируем.

'use strict';  
  
console.log(hoist); **// Вывод: ReferenceError: hoist is not defined**  
hoist = 'Hoisted';

Мы видим, что вместо того, чтобы указать, то что мы пропустили объявление нашей переменной, use strict остановил нас на полпути, выдав Reference error. Попробуйте это без использования use strict и понаблюдайте что произойдет.

Тем не менее, use-strict ведет себя по разному в разных браузерах, так что будет вполне благоразумно провести тестирование функционала перед использованием в работе.

## ****ES6****

Тут нам представляет интерес то, как этот стандарт влияет на объявление и инициализацию JavaScript переменных.

**let**

Перед тем как идти дальше, стоит отметить то, что переменные объявленные через let заключены в область видимости блока, а не функции. Это очень важно, но это не должно нас сейчас волновать.

Вкратце, это просто говорит о том, что область видимости переменной привязана к блоку, в котором она объявлена, а не к функции в которой она объявлена.

Давайте посмотрим на поведение let.

console.log(hoist); **// Вывод: ReferenceError: hoist is not defined** ...  
let hoist = 'The variable has been hoisted.';

Как и с var, мы ожидаем, что выведется в лог undefined. Но так как es6 let недружелюбно относится к необъявленным переменным, интерпретатор выдает нам Reference ошибку.

Это еще раз доказывает то, что надо сначала объявлять наши переменные.

Однако, всё еще нужно быть аккуратными. Пример ниже выдаст undefined вместо Reference error.

let hoist;  
  
console.log(hoist); **// Вывод: undefined**  
hoist = 'Hoisted'

Следовательно, чтобы не напороться на предыдущие ошибки, нам нужно сначала объявить переменную, а потом назначить ей значение и только потом уже её использовать.

**const**

const была представлена в es6 для того, чтобы можно было сделать неизменные переменные. Да, именно это, переменные значение которых не может быть изменено или переназначено.

С const, как и с let, переменные поднимаются вверх блока кода.

Давайте посмотрим, что происходит если мы попытаемся переназначить значение, прикрепленное к const переменной.

const PI = 3.142;  
  
PI = 22/7; **// Давайте изменим значение PI**  
console.log(PI); **// Вывод: TypeError: Assignment to constant variable.**

Как const изменяет объявление переменной? Давайте посмотрим.

console.log(hoist); **// Output: ReferenceError: hoist is not defined**  
const hoist = 'The variable has been hoisted.';

Как и с let, вместо тихого спокойного выхода с undefined, движок спасает нас, выдавая Reference error.

Тоже самое происходит при использовании const в функциях.

function getCircumference(radius) {  
 console.log(circumference)  
 circumference = PI\*radius\*2;  
 const PI = 22/7;  
}  
  
getCircumference(2) **// ReferenceError: circumference is not defined**

С const, es6 идет ещё дальше. Движок выдает ошибку, если мы используем константу перед её объявлением или инициализацией.

Наш линтер быстренько информирует нас об этом просчете:

PI was used before it was declared, which is illegal for const variables.

Глобально,

const PI;console.log(PI); **// Ouput: SyntaxError: Missing initializer in const declaration**PI=3.142;

Следовательно, константные переменные должны быть объявлены и инициализированы перед использованием.

Как пролог к этой секции, важно заметить, что JavaScript «поднимает» переменные let и const. Разница лишь в том, как он инициализирует их. Переменные объявленные с let и const остаются неинициализированными в начале выполнения, в то время как переменные объявленные с var инициализируются со значением undefined.

## ****Поднятие функций****

JavaScript функции могут классифицироваться как объявленные функции, так и как функциональные выражения. Далее мы узнаем как «поднятие» влияет на оба типа.

**Объявленные функции**

Такие функции полностью поднимаются вверх кода. Теперь понятно почему JavaScript позволяет нам вызывать функции прежде, чем мы их объявим по упоминанию в коде.

hoisted(); **// Вывод: "This function has been hoisted."**  
  
function hoisted() {  
 console.log('This function has been hoisted.');  
};

Функциональные выражения, однако, не поднимаются.

expression(); **//Вывод: "TypeError: expression is not a function**  
  
var expression = function() {  
 console.log('Will this work?');  
};

Как мы можем видеть выше, объявление переменной var expression поднимается, но его назначение как функции — нет. Следовательно, движок выдаст TypeError как увидит expression в виде переменой, а не функции.

## ****Порядок по приоритетам****

Очень важно помнить несколько вещей, объявляя JavaScript функции и переменные.

Назначение переменных имеет приоритет перед объявлением функции.

Объявление функции имеет приоритет перед объявлением переменной.

*Объявления функций «поднимаются» над объявлением переменных, но не над их назначениями.*

Давайте посмотрим как это работает.

**Назначение переменной над объявлением функции.**

var double = 22;  
  
function double(num) {  
 return (num\*2);  
}  
  
console.log(typeof double); **// Вывод: number**

**Объявление функции над объявлением переменной.**

var double;  
  
function double(num) {  
 return (num\*2);  
}  
  
console.log(typeof double); **// Вывод: function**

Даже если мы изменим позиции объявлений, JavaScript движок все равно бы взял double функцию.

## ****«Поднятие» классов****

Классы в JavaScript могут также классифицироваться как объявления и выражения.

**Объявления классов**

Так же как и свои коллеги функции, JavaScript классы при объявлении «поднимаются». Тем не менее, они остаются неинициализированными до определения. Это говорит о том, что вам надо объявить класс перед тем, как его использовать.

var Frodo = new Hobbit();  
Frodo.height = 100;  
Frodo.weight = 300;  
console.log(Frodo); **// Вывод: ReferenceError: Hobbit is not defined**  
  
class Hobbit {  
 constructor(height, weight) {  
 this.height = height;  
 this.weight = weight;  
 }  
}

Уверен, что вы заметили то, что вместо undefined мы получили Reference error. Это подвергает сомнению нашу позицию, что объявления классов «поднимаются».

Если мы взглянем на линтер, то там мы увидим полезный совет.

Hobbit was used before it is declared, which is illegal for class variables

Поэтому, чтобы получить доступ к классу, вам нужно сначала его объявить.

class Hobbit {  
 constructor(height, weight) {  
 this.height = height;  
 this.weight = weight;  
 }  
}  
  
var Frodo = new Hobbit();  
Frodo.height = 100;  
Frodo.weight = 300;  
console.log(Frodo); **// Вывод: { height: 100, weight: 300 }**

**Выражения классов**

Так же как и свои коллеги по функциям, выражения классов не «поднимаются».

Вот пример анонимного варианта выражения класса.

var Square = new Polygon();  
Square.height = 10;  
Square.width = 10;  
console.log(Square); **// Вывод: TypeError: Polygon is not a constructor**  
  
var Polygon = class {  
 constructor(height, width) {  
 this.height = height;  
 this.width = width;  
 }  
};

А вот пример с названным выражением класса.

var Square = new Polygon();  
Square.height = 10;  
Square.width = 10;  
console.log(Square); // Output: TypeError: Polygon is not a constructor  
  
  
var Polygon = class Polygon {  
 constructor(height, width) {  
 this.height = height;  
 this.width = width;  
 }  
};

А вот правильный порядок:

var Polygon = class Polygon {  
 constructor(height, width) {  
 this.height = height;  
 this.width = width;  
 }  
};  
  
var Square = new Polygon();  
Square.height = 10;  
Square.width = 10;  
console.log(Square);

## ****Предостережение****

Есть много споров о JavaScript es6 let, const переменных и классах, о том как они «поднимаются», приблизительно «поднимаются» или не «поднимаются» вообще. Некоторые утверждают, что они на самом деле «поднимаются», но не инициализируются, в то время как некоторые утверждают то, что они не «поднимаются» вообще.

## ****Заключение****

Давайте подведем итоги того, что мы изучили:

1. Используя es5 переменную var, попытки использования необъявленных переменных приведут к тому, что переменной будет назначено значение undefined при «поднятии».

2. Используя переменные es6 let и const, использование переменных приведет к Reference Error, потому что переменная останется неинициализированной при выполнении.

Тем не менее,

1. Нам нужно взять в привычку, объявлять и инициализировать JavaScript переменные перед использованием.

2. Использование strict mode в JavaScript es5 может помочь выявить необъявленные переменные.

# Тип данных Symbol

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

До сих пор мы видели только строки. Теперь давайте разберём символы, увидим, что хорошего они нам дают.

## [Символы](https://learn.javascript.ru/symbol" \l "simvoly)

«Символ» представляет собой уникальный идентификатор.

Создаются новые символы с помощью функции Symbol():

// Создаём новый символ - id

let id = Symbol();

При создании символу можно дать описание (также называемое имя), в основном использующееся для отладки кода:

// Создаём символ id с описанием (именем) "id"

let id = Symbol("id");

Символы гарантированно уникальны. Даже если мы создадим множество символов с одинаковым описанием, это всё равно будут разные символы. Описание – это просто метка, которая ни на что не влияет.

Например, вот два символа с одинаковым описанием – но они не равны:

let id1 = Symbol("id");

let id2 = Symbol("id");

alert(id1 == id2); // false

Если вы знаете Ruby или какой-то другой язык программирования, в котором есть своего рода «символы» – пожалуйста, будьте внимательны. Символы в JavaScript имеют свои особенности, и не стоит думать о них, как о символах в Ruby или в других языках.

**Символы не преобразуются автоматически в строки**

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

Это – языковая «защита» от путаницы, ведь строки и символы – принципиально разные типы данных и не должны неконтролируемо преобразовываться друг в друга.

Если же мы действительно хотим вывести символ с помощью alert, то необходимо явно преобразовать его с помощью метода .toString(), вот так:

let id = Symbol("id");

alert(id.toString()); // Symbol(id), теперь работает

Или мы можем обратиться к свойству symbol.description, чтобы вывести только описание:

let id = Symbol("id");

alert(id.description); // id

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

Символы позволяют создавать «скрытые» свойства объектов, к которым нельзя нечаянно обратиться и перезаписать их из других частей программы.

Например, мы работаем с объектами user, которые принадлежат стороннему коду. Мы хотим добавить к ним идентификаторы.

Используем для этого символьный ключ:

let user = {

name: "Вася"

};

let id = Symbol("id");

user[id] = 1;

alert( user[id] ); // мы можем получить доступ к данным по ключу-символу

Почему же лучше использовать Symbol("id"), а не строку "id"?

Так как объект user принадлежит стороннему коду, и этот код также работает с ним, то нам не следует добавлять к нему какие-либо поля. Это небезопасно. Но к символу сложно нечаянно обратиться, сторонний код вряд ли его вообще увидит, и, скорее всего, добавление поля к объекту не вызовет никаких проблем.

Кроме того, предположим, что другой скрипт для каких-то своих целей хочет записать собственный идентификатор в объект user. Этот скрипт может быть какой-то JavaScript-библиотекой, абсолютно не связанной с нашим скриптом.

Сторонний код может создать для этого свой символ Symbol("id"):

// ...

let id = Symbol("id");

user[id] = "Их идентификатор";

Конфликта между их и нашим идентификатором не будет, так как символы всегда уникальны, даже если их имена совпадают.

А вот если бы мы использовали строку "id" вместо символа, то тогда был бы конфликт:

let user = { name: "Вася" };

// Объявляем в нашем скрипте свойство "id"

user.id = "Наш идентификатор";

// ...другой скрипт тоже хочет свой идентификатор...

user.id = "Их идентификатор"

// Ой! Свойство перезаписано сторонней библиотекой!

### [Символы в литеральном объекте](https://learn.javascript.ru/symbol" \l "simvoly-v-literalnom-obekte)

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

Вот так:

let id = Symbol("id");

let user = {

name: "Вася",

[id]: 123 // просто "id: 123" не сработает

};

Это вызвано тем, что нам нужно использовать значение переменной id в качестве ключа, а не строку «id».

### [Символы игнорируются циклом for…in](https://learn.javascript.ru/symbol" \l "simvoly-ignoriruyutsya-tsiklom-for-in)

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

Например:

let id = Symbol("id");

let user = {

name: "Вася",

age: 30,

[id]: 123

};

for (let key in user) alert(key); // name, age (свойства с ключом-символом нет среди перечисленных)

// хотя прямой доступ по символу работает

alert( "Напрямую: " + user[id] );

Это – часть общего принципа «сокрытия символьных свойств». Если другая библиотека или скрипт будут работать с нашим объектом, то при переборе они не получат ненароком наше 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

Здесь нет никакого парадокса или противоречия. Так и задумано. Идея заключается в том, что, когда мы клонируем или объединяем объекты, мы обычно хотим скопировать все свойства (включая такие свойства с ключами-символами, как, например, id в примере выше).

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

Итак, как мы видели, обычно все символы уникальны, даже если их имена совпадают. Но иногда мы наоборот хотим, чтобы символы с одинаковыми именами были одной сущностью. Например, разные части нашего приложения хотят получить доступ к символу "id", подразумевая именно одно и то же свойство.

Для этого существует глобальный реестр символов. Мы можем создавать в нём символы и обращаться к ним позже, и при каждом обращении нам 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, при наличии в нём символа с именем key, возвращает его, иначе же создаётся новый символ Symbol(key) и записывается в реестр под ключом key.

Например:

// читаем символ из глобального реестра и записываем его в переменную

let id = Symbol.for("id"); // если символа не существует, он будет создан

// читаем его снова в другую переменную (возможно, из другого места кода)

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

// проверяем -- это один и тот же символ

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

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

**Похоже на Ruby**

В некоторых языках программирования, например, Ruby, на одно имя (описание) приходится один символ, и не могут существовать разные символы с одинаковым именем.

В JavaScript, как мы видим, это утверждение верно только для глобальных символов.

### [Symbol.keyFor](https://learn.javascript.ru/symbol" \l "symbol-keyfor)

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.

К примеру:

// 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.

Впрочем, для любых символов доступно свойство description.

Например:

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

let localSymbol = Symbol("name");

alert( Symbol.keyFor(globalSymbol) ); // name, глобальный символ

alert( Symbol.keyFor(localSymbol) ); // undefined для неглобального символа

alert( localSymbol.description ); // name

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

Существует множество «системных» символов, использующихся внутри самого JavaScript, и мы можем 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 так далее.

В частности, Symbol.toPrimitive позволяет описать правила для объекта, согласно которым он будет преобразовываться к примитиву. Мы скоро увидим его применение.

С другими системными символами мы тоже скоро познакомимся, когда будем изучать соответствующие возможности языка.

## [Итого](https://learn.javascript.ru/symbol" \l "itogo)

Символ (symbol) – примитивный тип данных, использующийся для создания уникальных идентификаторов.

Символы создаются вызовом функции Symbol(), в которую можно передать описание (имя) символа.

Даже если символы имеют одно и то же имя, это – разные символы. Если мы хотим, чтобы одноимённые символы были равны, то следует использовать глобальный реестр: вызов Symbol.for(key) возвращает (или создаёт) глобальный символ с key в качестве имени. Многократные вызовы команды Symbol.for с одним и тем же аргументом возвращают один и тот же символ.

Символы имеют два основных варианта использования:

1. «Скрытые» свойства объектов. Если мы хотим добавить свойство в объект, который «принадлежит» другому скрипту или библиотеке, мы можем создать символ и использовать его в качестве ключа. Символьное свойство не появится в for..in, так что оно не будет нечаянно обработано вместе с другими. Также оно не будет модифицировано прямым обращением, так как другой скрипт не знает о нашем символе. Таким образом, свойство будет защищено от случайной перезаписи или использования.

Так что, используя символьные свойства, мы можем спрятать что-то нужное нам, но что другие видеть не должны.

1. Существует множество системных символов, используемых внутри JavaScript, доступных как Symbol.\*. Мы можем использовать их, чтобы изменять встроенное поведение ряда объектов. Например, в дальнейших главах мы будем использовать Symbol.iterator для [итераторов](https://learn.javascript.ru/iterable), Symbol.toPrimitive для настройки [преобразования объектов в примитивы](https://learn.javascript.ru/object-toprimitive) и так далее.

Технически символы скрыты не на 100%. Существует встроенный метод [Object.getOwnPropertySymbols(obj)](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/Object/getOwnPropertySymbols) – с его помощью можно получить все свойства объекта с ключами-символами. Также существует метод [Reflect.ownKeys(obj)](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/Reflect/ownKeys), который возвращает все ключи объекта, включая символьные. Так что они не совсем спрятаны. Но большинство библиотек, встроенных методов и синтаксических конструкций не используют эти методы.

# Числа

В современном JavaScript существует два типа чисел:

1. Обычные 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). Это числа, которые мы будем использовать чаще всего. Мы поговорим о них в этой главе.
2. BigInt числа дают возможность работать с целыми числами произвольной длины. Они нужны достаточно редко и используются в случаях, когда необходимо работать со значениями более чем 253 или менее чем -253. Так как BigInt числа нужны достаточно редко, мы рассмотрим их в отдельной главе [BigInt](https://learn.javascript.ru/bigint).

В данной главе мы рассмотрим только первый тип чисел: числа типа number. Давайте глубже изучим, как с ними работать в JavaScript.

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

Представьте, что нам надо записать число 1 миллиард. Самый очевидный путь:

let billion = 1000000000;

Но в реальной жизни мы обычно опускаем запись множества нулей, так как можно легко ошибиться. Укороченная запись может выглядеть как "1млрд" или "7.3млрд" для 7 миллиардов 300 миллионов. 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, буквально: 1 и 9 нулей

alert( 7.3e9 ); // 7.3 миллиардов (7,300,000,000)

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

1e3 = 1 \* 1000

1.23e6 = 1.23 \* 1000000

Сейчас давайте запишем что-нибудь очень маленькое. К примеру, 1 микросекунду (одна миллионная секунды):

let ms = 0.000001;

Записать микросекунду в укороченном виде нам поможет "e".

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

Если мы подсчитаем количество нулей 0.000001, их будет 6. Естественно, верная запись 1e-6.

Другими словами, отрицательное число после "e" подразумевает деление на 1 с указанным количеством нулей:

// 1 делится на 1 с 3 нулями

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

// 1.23 делится на 1 с 6 нулями

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

### [Шестнадцатеричные, двоичные и восьмеричные числа](https://learn.javascript.ru/number" \l "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 многого другого. Естественно, есть короткий стиль записи: 0x, после которого указывается число.

Например:

alert( 0xff ); // 255

alert( 0xFF ); // 255 (то же самое, 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, с двух сторон число 255

Есть только 3 системы счисления с такой поддержкой. Для других систем счисления мы рекомендуем использовать функцию parseInt (рассмотрим позже в этой главе).

## [toString(base)](https://learn.javascript.ru/number" \l "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

base может варьироваться от 2 до 36 (по умолчанию 10).

Часто используемые:

* **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. Забавно, но можно использовать 36-разрядную систему счисления для получения короткого представления большого числового идентификатора. К примеру, для создания короткой ссылки. Для этого просто преобразуем его в 36-разрядную систему счисления:

alert( 123456..toString(36) ); // 2n9c

**Две точки для вызова метода**

Внимание! Две точки в 123456..toString(36) это не опечатка. Если нам надо вызвать метод непосредственно на числе, как toString в примере выше, то нам надо поставить две точки .. после числа.

Если мы поставим 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" \l "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.

Ниже представлена 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. Что если нам надо округлить число до n-ого количества цифр в дробной части?

Например, у нас есть 1.2345 и мы хотим округлить число до 2-х знаков после запятой, оставить только 1.23.

Есть два пути решения:

1. Умножить и разделить.

Например, чтобы округлить число до второго знака после запятой, мы можем 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.
2. let num = 12.34;

alert( num.toFixed(1) ); // "12.3"

Округляет значение до ближайшего числа, как в большую, так и в меньшую сторону, аналогично методу Math.round:

let num = 12.36;

alert( num.toFixed(1) ); // "12.4"

Обратите внимание, что результатом toFixed является строка. 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

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

## [Неточные вычисления](https://learn.javascript.ru/number" \l "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 хранение 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.

Посмотрите на это (неверное!) сравнение:

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

Да-да, сумма 0.1 и 0.2 не равна 0.3.

Странно! Что тогда, если не 0.3?

alert( 0.1 + 0.2 ); // 0.30000000000000004

Ой! Здесь гораздо больше последствий, чем просто некорректное сравнение. Представьте, вы делаете интернет-магазин и посетители формируют заказ из 2-х позиций за $0.10 и $0.20. Итоговый заказ будет $0.30000000000000004. Это будет сюрпризом для всех.

Но почему это происходит?

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.

Другими словами, что такое 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 гарантированно 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 нам увидеть эту «крошечную потерю точности», но она существует.

Пример:

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

И когда мы суммируем 2 числа, их «неточности» тоже суммируются.

Вот почему 0.1 + 0.2 – это не совсем 0.3.

**Не только в JavaScript**

Справедливости ради заметим, что ошибка в точности вычислений для чисел с плавающей точкой сохраняется в любом другом языке, где используется формат IEEE 754, включая PHP, Java, C, Perl, Ruby.

Можно ли обойти проблему? Конечно, наиболее надёжный способ — 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

Помните, что метод toFixed всегда возвращает строку. Это гарантирует, что результат будет с заданным количеством цифр в десятичной части. Также это удобно для форматирования цен в интернет-магазине $0.30. 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 временно 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.

Иногда можно попробовать полностью отказаться от дробей. Например, если мы в нашем интернет-магазине начнём использовать центы вместо долларов. Но что будет, если мы применим скидку 30%? На практике у нас не получится полностью избавиться от дроби. Просто используйте округление, чтобы отрезать «хвосты», когда надо.

**Забавный пример**

Попробуйте выполнить его:

// Привет! Я – число, растущее само по себе!

alert( 9999999999999999 ); // покажет 10000000000000000

Причина та же – потеря точности. Из 64 бит, отведённых на число, сами цифры числа занимают до 52 бит, остальные 11 бит хранят позицию десятичной точки и один бит – знак. Так что если 52 бит не хватает на цифры, то при записи пропадут младшие разряды.

Интерпретатор не выдаст ошибку, но в результате получится «не совсем то число», что мы и видим в примере выше. Как говорится: «как смог, так записал».

**Два нуля**

Другим забавным следствием внутреннего представления чисел является наличие двух нулей: 0 и -0.

Все 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" \l "proverka-isfinite-i-isnan)

Помните эти специальные числовые значения?

* Infinity (и -Infinity) — особенное численное значение, которое ведёт себя в точности как математическая бесконечность ∞.
* NaN представляет ошибку.

Эти числовые значения 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

Нужна ли нам эта функция? Разве не можем ли мы просто сравнить === NaN? К сожалению, нет. 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, потому что специальное значение: NaN

alert( isFinite(Infinity) ); // false, потому что специальное значение: Infinity

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

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

// вернёт true всегда, кроме ситуаций, когда аргумент - Infinity/-Infinity или не число

alert( isFinite(num) );

Помните, что пустая строка интерпретируется как 0 во всех числовых функциях, включаяisFinite.

**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. Работает с NaN: Object.is(NaN, NaN) === true, здесь он хорош.
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 внутреннему алгоритму необходимо сравнить 2 значения на предмет точного совпадения, он использует Object.is (Определение [SameValue](https://tc39.github.io/ecma262/#sec-samevalue)).

## [parseInt и parseFloat](https://learn.javascript.ru/number" \l "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.

В реальной жизни мы часто сталкиваемся со значениями у которых есть единица измерения, например "100px" или "12pt" в CSS. Также во множестве стран символ валюты записывается после номинала "19€". Так как нам получить числовое значение из таких строк?

Для этого есть parseInt и parseFloat.

Они «читают» 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, вернётся только целая часть

alert( parseFloat('12.3.4') ); // 12.3, произойдёт остановка чтения на второй точке

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

alert( parseInt('a123') ); // NaN, на первом символе происходит остановка чтения

**Второй аргумент parseInt(str, radix)**

Функция parseInt() 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, без 0x тоже работает

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

## [Другие математические функции](https://learn.javascript.ru/number" \l "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 от 0 (включительно) до 1 (но не включая 1)

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

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

alert( Math.random() ); // ... (любое количество псевдослучайных чисел)

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

## [Итого](https://learn.javascript.ru/number#itogo)

Чтобы писать числа с большим количеством нулей:

* Используйте краткую форму записи чисел – "e", с указанным количеством нулей. Например: 123e6 это 123 с 6-ю нулями 123000000.
* Отрицательное число после "e" приводит к делению числа на 1 с указанным количеством нулей. Например: 123e-6 это 0.000123 (123 миллионных).

Для других систем счисления:

* Можно записывать числа сразу в шестнадцатеричной (0x), восьмеричной (0o) и бинарной (0b) системах счисления
* parseInt(str, base) преобразует строку в целое число в соответствии с указанной системой счисления: 2 ≤ base ≤ 36.
* num.toString(base) представляет число в строковом виде в указанной системе счисления base.

Для преобразования значений типа 12pt и 100px в число:

* Используйте parseInt/parseFloat для «мягкого» преобразования строки в число, данные функции по порядку считывают число из строки до тех пор пока не возникнет ошибка.

Для дробей:

* Используйте округления Math.floor, Math.ceil, Math.trunc, Math.round или num.toFixed(precision).
* Помните, что при работе с дробями происходит потеря точности.

Ещё больше математических функций:

* Документация по объекту [Math](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/Math). Библиотека маленькая, но содержит всё самое важное.

## [Задачи](https://learn.javascript.ru/number#tasks)

### [Сумма пользовательских чисел](https://learn.javascript.ru/number" \l "summa-polzovatelskih-chisel)

важность: 5

Создайте скрипт, который запрашивает ввод двух чисел (используйте prompt) и после показывает их сумму.

[Запустить демо](https://learn.javascript.ru/number)

P.S. Есть «подводный камень» при работе с типами.

решение

### [Почему 6.35.toFixed(1) == 6.3?](https://learn.javascript.ru/number" \l "pochemu-6-35-tofixed-1-6-3)

важность: 4

Методы Math.round и toFixed, согласно документации, округляют до ближайшего целого числа: 0..4 округляется в меньшую сторону, тогда как 5..9 в большую сторону.

Например:

alert( 1.35.toFixed(1) ); // 1.4

Но почему в примере ниже 6.35 округляется до 6.3?

alert( 6.35.toFixed(1) ); // 6.3

Как правильно округлить 6.35?

решение

### [Ввод числового значения](https://learn.javascript.ru/number" \l "vvod-chislovogo-znacheniya)

важность: 5

Создайте функцию readNumber, которая будет запрашивать ввод числового значения до тех пор, пока посетитель его не введёт.

Функция должна возвращать числовое значение.

Также надо разрешить пользователю остановить процесс ввода, отправив пустую строку или нажав «Отмена». В этом случае функция должна вернуть null.

[Запустить демо](https://learn.javascript.ru/number)

[Открыть песочницу с тестами для задачи.](https://plnkr.co/edit/dKRj7PBDpz58ed0t?p=preview)

решение

### [Бесконечный цикл по ошибке](https://learn.javascript.ru/number" \l "beskonechnyy-tsikl-po-oshibke)

важность: 4

Этот цикл – бесконечный. Он никогда не завершится, почему?

let i = 0;

while (i != 10) {

i += 0.2;

}

решение

### [Случайное число от min до max](https://learn.javascript.ru/number" \l "sluchaynoe-chislo-ot-min-do-max)

важность: 2

Встроенный метод Math.random() возвращает случайное число от 0 (включительно) до 1 (но не включая 1)

Напишите функцию random(min, max), которая генерирует случайное число с плавающей точкой от min до max (но не включая max).

Пример работы функции:

alert( random(1, 5) ); // 1.2345623452

alert( random(1, 5) ); // 3.7894332423

alert( random(1, 5) ); // 4.3435234525

решение

### [Случайное целое число от min до max](https://learn.javascript.ru/number" \l "sluchaynoe-tseloe-chislo-ot-min-do-max)

важность: 2

Напишите функцию randomInteger(min, max), которая генерирует случайное целое (integer) число от min до max (включительно).

Любое число из интервала min..max должно появляться с одинаковой вероятностью.

Пример работы функции:

alert( randomInteger(1, 5) ); // 1

alert( randomInteger(1, 5) ); // 3

alert( randomInteger(1, 5) ); // 5

Можно использовать решение из [предыдущей задачи](https://learn.javascript.ru/task/random-min-max).

решение

Математика

#### Введение

Наличие чисел и числовых данных позволяет вам каким-либо образом работать с языком. Но помимо работы с арифметическими операторами в JavaScript, математические конструкции могут быть сложной задачей для новичков. По этой причине, нужно концентрироваться не на синтаксисе, а на общих математических функциях, этот список начинается с таких вещей как сортировка, округление и генерация случайных значений, этого достаточно, прежде чем углубляться в детали. Чтобы [работать с математикой в JavaScript,](http://thenewcode.com/1131/Web-Developer-Reading-List-Maths-in-JavaScript) вам достаточно иметь понятия о таких вещах как функция, операнд и оператор.

#### От переводчиков

Всем привет, с вами [Максим Иванов](https://github.com/splincode) и [Дмитрий Сергиенков](https://habrahabr.ru/users/blackbaron/), и сегодня мы решили, что мы не будем говорить о модных и полезных штуках типа ReactJS, Angular, TypeScript и других. Сегодня мы уделим внимание математике в JavaScript. Если [вам нравится математика](http://itmozg.ru/news/1232/#.WBcNt3WLSkA), вы можете заниматься ей всё свободное время, но если вашей целью являются не научные изыскания, а работа программистом, [математика вряд ли станет лучшим объектом для изучения](https://professionali.ru/Soobschestva/it-specialisty/nuzhna-li-matematika-programmistu/).

1. [Работа со случайными числами](https://habr.com/ru/post/312880/#s1)
   * [Основной случай](https://habr.com/ru/post/312880/#s1_1)
   * [Случайное число в интервале [min, max)](https://habr.com/ru/post/312880/#s1_2)
   * [Случайное число в интервале [min, max]](https://habr.com/ru/post/312880/#s1_3)
   * [Булевские случайные величины (true/false)](https://habr.com/ru/post/312880/#s1_4)
   * [Случайные величины с исключениями](https://habr.com/ru/post/312880/#s1_5)
   * [Случайные величины без повторений](https://habr.com/ru/post/312880/#s1_6)
   * [Криптографические случайные величины](https://habr.com/ru/post/312880/#s1_7)
2. [Округления](https://habr.com/ru/post/312880/#s2)
   * [Зачем необходимо округление?](https://habr.com/ru/post/312880/#s2_1)
   * [Округление десятичных чисел](https://habr.com/ru/post/312880/#s2_2)
   * [Как избежать ошибок округления с десятичными числами](https://habr.com/ru/post/312880/#s2_3)
   * [Машинное эпсилон округление](https://habr.com/ru/post/312880/#s2_4)
   * [Отсечение дробной части](https://habr.com/ru/post/312880/#s2_5)
   * [Округление до ближайшего числа](https://habr.com/ru/post/312880/#s2_6)
   * [Округление к меньшему до ближайшего целого числа](https://habr.com/ru/post/312880/#s2_7)
   * [Округление к большему до ближайшего целого числа](https://habr.com/ru/post/312880/#s2_8)
   * [Округление до большего/меньшего необходимого числа](https://habr.com/ru/post/312880/#s2_9)
   * [Фиксирование числа в диапазоне](https://habr.com/ru/post/312880/#s2_10)
   * [Гауссово округление](https://habr.com/ru/post/312880/#s2_11)
3. [Сортировка](https://habr.com/ru/post/312880/#s3)
   * [Сортировка в алфавитном порядке](https://habr.com/ru/post/312880/#s3_1)
   * [Числовая сортировка](https://habr.com/ru/post/312880/#s3_2)
   * [Сортировка JSON-подобной структуры](https://habr.com/ru/post/312880/#s3_3)
4. [Работа со степенными функциями](https://habr.com/ru/post/312880/#s4)
   * [Возведение в степень](https://habr.com/ru/post/312880/#s4_1)
   * [Квадратный и кубический корень](https://habr.com/ru/post/312880/#s4_2)
5. [Математические константы](https://habr.com/ru/post/312880/#s5)
   * [Math.PI](https://habr.com/ru/post/312880/#s5_1)
   * [Math.SQRT2](https://habr.com/ru/post/312880/#s5_2)
   * [Math.SQRT1\_2](https://habr.com/ru/post/312880/#s5_3)
   * [Math.E](https://habr.com/ru/post/312880/#s5_4)
   * [Math.LN2](https://habr.com/ru/post/312880/#s5_5)
   * [Math.LN10](https://habr.com/ru/post/312880/#s5_6)
   * [Math.LOG2E](https://habr.com/ru/post/312880/#s5_7)
   * [Math.LOG10E](https://habr.com/ru/post/312880/#s5_8)
6. [Math.abs, parseInt, parseFloat](https://habr.com/ru/post/312880/#s6)
   * [Math.abs()](https://habr.com/ru/post/312880/#s6_1)
   * [parseInt()](https://habr.com/ru/post/312880/#s6_2)
   * [parseFloat()](https://habr.com/ru/post/312880/#s6_3)

### Работа со случайными числами

Случайные числа часто требуются в JavaScript, к примеру, для отрисовки [звезд, разбросанных по ночному небу.](http://thenewcode.com/81/Make-A-Starfield-Background-with-HTML5-Canvas) Но есть много различных видов случайностей, и в зависимости от логики и потребностей вашего приложения вам может понадобиться один из них.

### Основной случай

Самая простая форма случайности — это функция Math.random(), встроенная в JavaScript.

> Math.random()

0.19401081069372594

Math.random() всегда возвращает число с плавающей точкой между 0 и 1. С технической точки зрения число, возвращаемое при помощи Math.random() может быть 0, но никогда не будет равно 1.  
  
Если вы часто используйте Math.random(), используйте свою собственную функцию в сценариях:

function getRandom() {

return Math.random();

}

Проблема, конечно, в том, что данная функция всегда будет создавать случайное число в пределах весьма ограниченного диапазона, далее мы постараемся рассмотреть некоторые рекомендации предназначенные для решения этой проблемы.

### Случайное число в интервале [min, max)

Расширение такой функциональности требует немного математики:  
  
Случайное число с плавающей точкой:

function getRandomFloat(min, max) {

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

}

getRandomFloat(11, 101)

> 75.31898734299466

Целочисленное случайное число:

function getRandomInt(min, max) {

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

}

getRandomInt(10, 20)

> 12

### Случайное число в интервале [min, max]

function getRandomInRange(min, max) {

return Math.floor(Math.random() \* (max - min + 1)) + min;

}

getRandomInRange(1, 10)

> 7

### Булевские случайные величины (true/false)

Если вы хотите простую случайность 0 или 1, представляющую подбрасывание монеты:

function coinToss() {

return Math.floor(Math.random() \* 2);

}

coinToss();

> 0

Если вам нужно получить true или false:

function coinToss() {

return (Math.floor(Math.random() \* 2) === 0);

}

coinToss();

> true

Или же:

function coinToss() {

return Math.random()<.5;

}

coinToss();

> true

Если вы хотите связать конкретные слова со сторонами монеты (да / нет, верх / низ и т.д.):

function coinFlip() {

return (Math.floor(Math.random() \* 2) === 0) ? "up" : "down";

}

coinFlip();

> up

### Случайные величины с исключениями

Для ограниченного диапазона целых чисел необходимо создать массив чисел, которые вы хотели бы извлечь и в дальнейшем выбрать случайным образом из этого массива:

let numPool = [ 1, 3, 5, 7, 9, 10 ],

rand = numPool[Math.floor(Math.random() \* numPool.length)];

Можно также использовать массив чисел, которые вы хотите исключить, и приготовить пустой массив, в котором будет содержаться результат фильтрации из первого массива во второй:

let numPool = [ 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 ];

let excludePool = [ 3, 4 ];

let filteredPool = [];

Затем в цикле прогоняем массив numPool, если выпавшее число существует и находится в массиве excludePool, помещаем результат в filteredPool:

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

if (excludePool.indexOf(numPool[i]) === -1) {

filteredPool.push(numPool[i]);

}

}

И, наконец, получаем случайные числа из массива filteredPool:

let rand = filteredPool[Math.floor(Math.random() \* filteredPool.length)];

### Случайные величины без повторений

Для небольшого набора чисел необходимо создать массив, заполненный элементами, далее перетасуйте их в случайном порядке, поместите результат в новый массив, а затем вытаскивайте по одному:

let numPool = [ 13, 21, 36, 14, 27, 10 ];

function shuffle(numPool) {

for(

let j, x, i = numPool.length; i;

j = parseInt(Math.random() \* i),

x = numPool[--i],

numPool[i] = numPool[j],

numPool[j] = x

);

return numPool;

};

let randomResult = shuffle(numPool);

while( randomResult.length > 0 ) {

console.log( randomResult.pop() );

}

Для получения более широкого диапазона чисел, создайте и заполните массив случайными числами, исключая любые, которые были ранее сгенерированы:

let numReserve = []

while (numReserve.length < 12) {

let randomNumber = Math.ceil(Math.random() \* 1000);

let found = false;

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

if (numReserve[i] === randomNumber){

found = true;

break;

}

}

if (!found) { numReserve[numReserve.length]=randomNumber; }

}

В приведенном выше коде numReserve заполняется 12ю случайными числами от 0 до 1000.

### Криптографические случайные величины

К сожалению, ни один из выше методов не создает число с достаточной хаотичностью для криптографически защищенных функций (Math.random() не является достаточной функцией, которая генерирует случайные числа). Поэтому, мы можем использовать Web Cryptography API путем создания typedArray:

let cryptoStor = new Uint16Array(8);

В этом случае мы создаем массив с восемью различными слотами, каждый из которых содержит беззнаковое 16-разрядное целое число. Существуют и другие варианты Int8Array, Uint8Array, int16Array, Int32Array и Uint32Array.  
  
Затем, необходимо заполнить массив случайными числами определенного типа:

window.crypto.getRandomValues(cryptoStor);

> [43484, 57947, 46691, 49849, 24272, 11827, 28203, 17423]

Ко всему прочему, Web Cryptography API имеет хорошую поддержку в [современных браузерах](http://caniuse.com/#search=Web%20Cryptography).

### **К прочтению:**

1. [Случайное простое число](https://ru.wikipedia.org/wiki/%D0%A1%D0%BB%D1%83%D1%87%D0%B0%D0%B9%D0%BD%D0%BE%D0%B5_%D0%BF%D1%80%D0%BE%D1%81%D1%82%D0%BE%D0%B5_%D1%87%D0%B8%D1%81%D0%BB%D0%BE)  
2. [Генерация случайных целых чисел в JavaScript в определенном диапазоне](http://stackoverflow.com/questions/1527803/generating-random-whole-numbers-in-javascript-in-a-specific-range)  
3. [Введение в криптографию](http://window.edu.ru/resource/256/20256/files/rsu572.pdf)  
4. [Криптографически стойкий генератор псевдослучайных чисел](https://ru.wikipedia.org/wiki/%D0%9A%D1%80%D0%B8%D0%BF%D1%82%D0%BE%D0%B3%D1%80%D0%B0%D1%84%D0%B8%D1%87%D0%B5%D1%81%D0%BA%D0%B8_%D1%81%D1%82%D0%BE%D0%B9%D0%BA%D0%B8%D0%B9_%D0%B3%D0%B5%D0%BD%D0%B5%D1%80%D0%B0%D1%82%D0%BE%D1%80_%D0%BF%D1%81%D0%B5%D0%B2%D0%B4%D0%BE%D1%81%D0%BB%D1%83%D1%87%D0%B0%D0%B9%D0%BD%D1%8B%D1%85_%D1%87%D0%B8%D1%81%D0%B5%D0%BB)  
5. [Генератор псевдослучайных чисел](https://ru.wikipedia.org/wiki/%D0%93%D0%B5%D0%BD%D0%B5%D1%80%D0%B0%D1%82%D0%BE%D1%80_%D0%BF%D1%81%D0%B5%D0%B2%D0%B4%D0%BE%D1%81%D0%BB%D1%83%D1%87%D0%B0%D0%B9%D0%BD%D1%8B%D1%85_%D1%87%D0%B8%D1%81%D0%B5%D0%BB)

### Округления

Очень часто вычисления в JavaScript дают не совсем те результаты, которые мы хотим. Разумеется, мы можем делать с числами что угодно — округлять в большую или меньшую сторону, устанавливать диапазоны, отсекать ненужные числа до определенного количества знаков после запятой, все зависит от того, что вы хотите сделать в дальнейшем с этим числом.

### Зачем необходимо округление?

Одним из любопытных аспектов JavaScript является то, что он на самом деле не хранит целые числа, мы сразу же работаем с числами с плавающей точкой. Это, в сочетании с тем фактом, что многие дробные значения не могут быть выражены конечным числом знаков после запятой, в JavaScript мы можем получить такие результаты:

0.1 \* 0.2;

> 0.020000000000000004

0.3 - 0.1

> 0.19999999999999998

Для практических целей эта неточность не имеет никакого значения, в нашем случае мы говорим об ошибке в квинтиллионных долях, однако, кого-то это может разочаровать. Мы можем получить несколько странный результат и при работе с числами, которые представляют собой значения валют, процентов или размеров файла. Для того, чтобы исправить эти неточности, нам как раз и необходимо уметь округлять результаты, при этом достаточно установить десятичную точность.  
  
Округление чисел имеет практическое применение, мы можем манипулировать числом в некотором диапазоне, например, хотим округлить значение до ближайшего целого числа, а не работать только с десятичной частью.

### Округление десятичных чисел

Для того, чтобы отсечь десятичное число, используйте [toFixed](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Number/toFixed) или метод [toPrecision](https://developer.mozilla.org/en/docs/Web/JavaScript/Reference/Global_objects/Number/toPrecision). Оба они принимают единственный аргумент, который определяет, соответственно, сколько значащих цифр (т.е. общее количество цифр, используемых в числе) или знаков после запятой (количество после десятичной точки) должен включать в себя результат:

1. Если аргумент не определен для toFixed(), то по умолчанию он будет равен нулю, что означает 0 знаков после запятой, аргумент имеет максимальное значение, равное 20.
2. Если аргумент не задан для toPrecision, число остается нетронутым

let randNum = 6.25;

randNum.toFixed();

> "6"

Math.PI.toPrecision(1);

> "3"

randNum = 87.335;

randNum.toFixed(2);

> "87.33"

randNum = 87.337;

randNum.toPrecision(3);

> "87.3"

Оба метода toFixed() и toPrecision() возвращают строковое представление результата, а не число. Это означает, что при суммировании округленного значения с randNum будет произведена конкатенация строк, а не сумма чисел:

let randNum = 6.25;

let rounded = randNum.toFixed(); *// "6"*

console.log(randNum + rounded);

> "6.256"

Если вы хотите, чтобы результат имел числовой тип данных, то вам необходимо будет применить parseFloat:

let randNum = 6.25;

let rounded = parseFloat(randNum.toFixed(1));

console.log(rounded);

> 6.3

Обратите внимание, что значения 5 округлены, за исключением редких случаев.  
  
Методы toFixed() и toPrecision() являются полезными, ибо они могут не только отсекать дробную часть, но и дополнять знаки после запятой, что удобно при работе с валютой:

let wholeNum = 1

let dollarsCents = wholeNum.toFixed(2);

console.log(dollarsCents);

> "1.00"

Стоить обратите внимание, что toPrecision будет давать результат в экспоненциальной записи, если число целых чисел больше, чем сам сама точность:

let num = 123.435

num.toPrecision(2);

> "1.2e+2"

### Как избежать ошибок округления с десятичными числами

В некоторых случаях, toFixed и toPrecision округляет значение 5 в меньшую сторону, а в большую:

let numTest = 1.005;

numTest.toFixed(2);

> "1.00"

Результат расчета выше должен был быть 1.01, а не 1. Если вы хотите избежать подобную ошибку, мы можем использовать решение, предложенное [Jack L Moore](http://www.jacklmoore.com/notes/rounding-in-javascript/), которое использует экспоненциальные числа для расчета:

function round(value, decimals) {

return Number(Math.round(value+'e'+decimals)+'e-'+decimals);

}

Теперь:

round(1.005,2);

> 1.01

Если вы хотите более надежное решение, чем решение показанное выше, вы можете перейти на [MDN](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Math/round).

### Машинное эпсилон округление

Альтернативный метод округления десятичных чисел был введен в ES6. [Машинное эпсилон](https://ru.wikipedia.org/wiki/%D0%9C%D0%B0%D1%88%D0%B8%D0%BD%D0%BD%D1%8B%D0%B9_%D0%BD%D0%BE%D0%BB%D1%8C) округление обеспечивает разумный предел погрешности при сравнении двух чисел с плавающей точкой. Без округления, сравнения могут дать результаты, подобные следующим:

0.1 + 0.2 === 0.3

> false

Мы используем [Math.EPSILON](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Number/EPSILON) в нашей функции для получения корректного сравнения:

function epsEqu(x, y) {

return Math.abs(x - y) < Number.EPSILON \* Math.max(Math.abs(x), Math.abs(y));

}

Функция принимает два аргумента: первый — текущий расчет, второй — ожидаемый результат. Она возвращает сравнение двух:

epsEqu(0.1 + 0.2, 0.3)

> true

Все современные браузеры уже поддерживают ES6 математические функции, но если вы хотите получить поддержку в таких браузерах, как IE 11, используйте [polyfills](https://github.com/MaxArt2501/es6-math).

### Отсечение дробной части

Все методы, представленные выше умеют округлять до десятичных чисел. Для того, чтобы просто отсечь число до двух знаков после запятой, необходимо сначала умножить его на 100, а затем полученный результат уже разделить на 100:

function truncated(num) {

return Math.trunc(num \* 100) / 100;

}

truncated(3.1416)

> 3.14

Если вы хотите приспособить метод под любое количество знаков после запятой, вы можете воспользоваться [двойным побитовым отрицанием](http://rocha.la/JavaScript-bitwise-operators-in-practice):

function truncated(num, decimalPlaces) {

let numPowerConverter = Math.pow(10, decimalPlaces);

return ~~(num \* numPowerConverter)/numPowerConverter;

}

Теперь:

let randInt = 35.874993;

truncated(randInt,3);

> 35.874

### Округление до ближайшего числа

Для того, чтобы округлить десятичное число до ближайшего числа в большую или в меньшую сторону, в зависимости от того, к чему мы ближе всего, используйте Math.round():

Math.round(4.3)

> 4

Math.round(4.5)

> 5

Обратите внимание, что «половина значения», 0.5 округляется в большую сторону [по правилам математики](https://ru.wikipedia.org/wiki/%D0%9E%D0%BA%D1%80%D1%83%D0%B3%D0%BB%D0%B5%D0%BD%D0%B8%D0%B5).

### Округление к меньшему до ближайшего целого числа

Если вы хотите всегда округлять в меньшую сторону, используйте Math.floor:

Math.floor(42.23);

> 42

Math.floor(36.93);

> 36

Обратите внимание, что округление в меньшую сторону работает для всех чисел, в том числе и для отрицательных. Представьте небоскреб с бесконечным количеством этажей, в том числе с этажами нижнего уровня (представляющий отрицательные числа). Если вы находитесь в лифте на нижним уровнем между 2 и 3 (что представляет собой значение -2.5), Math.floor доставит вас до -3:

Math.floor(-2.5);

> -3

Но если вы хотите избежать подобной ситуации, используйте [Math.trunc](https://developer.mozilla.org/en/docs/Web/JavaScript/Reference/Global_Objects/Math/trunc), поддерживаемый во всех современных браузерах (кроме IE / Edge):

Math.trunc(-41.43);

> -41

На MDN вы найдете polyfill, который обеспечит поддержку Math.trunc в браузерах и IE / Edge.

### Округление к большему до ближайшего целого числа

С другой стороны, если вам нужно всегда округлять в большую сторону, используйте Math.ceil. Опять же, вспоминаем бесконечный лифт: Math.ceil всегда будет идти «вверх», независимо от того, является ли число отрицательное или нет:

Math.ceil(42.23);

> 43

Math.ceil(36.93);

> 37

Math.ceil(-36.93);

> -36

### Округление до большего/меньшего необходимого числа

Если мы хотим, чтобы округлить до ближайшего числа, кратного 5, самый простой способ создать функцию, которая делит число на 5, округляет его, а затем умножает его на ту же сумму:

function roundTo5(num) {

return Math.round(num/5)\*5;

}

Теперь:

roundTo5(11);

> 10

Если вы хотите округлять до кратных своему значению, мы использовать более общую функцию, передавая в нее начальное значение и кратное:

function roundToMultiple(num, multiple) {

return Math.round(num/multiple)\*multiple;

}

Теперь:

let initialNumber = 11;

let multiple = 10;

roundToMultiple(initialNumber, multiple);

> 10;

### Фиксирование числа в диапазоне

Есть много случаев, когда мы хотим получить значение х, лежащее в пределах диапазона. Например, нам может понадобиться значение от 1 до 100, но при этом мы получили значение 123. Для того, чтобы исправить это, мы можем использовать минимальное (возвращает наименьшее из набора чисел) и максимальное (возвращает наибольшее из любого множества чисел). В нашем примере, диапазон от 1 до 100:

let lowBound = 1;

let highBound = 100;

let numInput = 123;

let clamped = Math.max(lowBound, Math.min(numInput, highBound));

console.log(clamped);

> 100;

Опять же, мы можем переиспользовать операцию и обернуть все это в функцию, воспользуемся решением предложенное [Daniel X. Moore](http://strd6.com/author/yahivin/):

Number.prototype.clamp = function(min, max) {

return Math.min(Math.max(this, min), max);

};

Теперь:

numInput.clamp(lowBound, highBound);

> 100;

### Гауссово округление

Гауссово округление, также известное как банковское округлением, заключается в том, что округление для этого случая происходит к ближайшему чётному. Этот метод округления работает без статистической погрешности. Лучшее решение было предложено [Tim Down](https://github.com/timdown):

function gaussRound(num, decimalPlaces) {

let d = decimalPlaces || 0,

m = Math.pow(10, d),

n = +(d ? num \* m : num).toFixed(8),

i = Math.floor(n), f = n - i,

e = 1e-8,

r = (f > 0.5 - e && f < 0.5 + e) ?

((i % 2 == 0) ? i : i + 1) : Math.round(n);

return d ? r / m : r;

}

Теперь:

gaussRound(2.5)

> 2

gaussRound(3.5)

> 4

gaussRound(2.57,1)

> 2.6

Десятичный знак в CSS:  
  
Так как JavaScript часто используется для создания позиционного преобразования HTML-элементов, вы можете задаться вопросом, что произойдет, если мы cгенерируем десятичные значения для наших элементов:

#box { width: 63.667731993px; }

Хорошая новость заключается в том, что современные браузеры будут учитывать десятичные значения в блочной модели, в том числе в процентных или пиксельных единицах измерения.

### **К прочтению:**

1. [Что каждый ученый должен знать об арифметике при работе с плавающей запятой](http://docs.oracle.com/cd/E19957-01/806-3568/ncg_goldberg.html)  
2. [Быстрая JavaScript-библиотека, работающая с десятичной арифметикой произвольной точности](https://github.com/MikeMcl/big.js/)  
3. [Расширенный тип Decimal произвольной точности для JavaScript](https://github.com/MikeMcl/decimal.js/)  
4. [JavaScript-библиотека произвольной точности для десятичной и произвольной арифметики](https://github.com/MikeMcl/bignumber.js/)  
5. [Что должен знать каждый JavaScript-разработчик о плавающей запятой](https://modernweb.com/2014/02/17/what-every-javascript-developer-should-know-about-floating-points/)  
6. [Отчего такие проблемы с плавающей запятой в JS?](http://stackoverflow.com/questions/588004/is-floating-point-math-broken)

### Сортировка

Очень часто нам приходится сортировать какие-либо элементы, например, у нас есть массив игровых рекордов, при этом они должны быть организованы по убыванию ранга игроков. К сожалению, стандартный метод [sort()](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/Array/sort) имеет некоторые удивительные ограничения: он хорошо работает с часто употребляемыми английскими словами, но сразу же ломается при встрече с числами, уникальными символами или словами в верхнем регистре.

### Сортировка в алфавитном порядке

Казалось бы, сортировки массива по алфавиту должна быть простейшей задачей:

let fruit = ["butternut squash", "apricot", "cantaloupe"];

fruit.sort();

> "apricot", "butternut squash", "cantaloupe"]

Тем не менее мы сталкиваемся с проблемой, как только один из элементов находится в верхнем регистре:

let fruit = ["butternut squash", "apricot", "Cantalope"];

fruit.sort();

> "Cantaloupe", "apricot", "butternut squash"]

Это связано с тем, что, по умолчанию, сортировщик сравнивает первый символ представленный в [Unicode](https://ru.wikipedia.org/wiki/%D0%AE%D0%BD%D0%B8%D0%BA%D0%BE%D0%B4). Unicode — это уникальный код для любого символа, независимо от платформы, независимо от программы, независимо от языка. Например, если смотреть [по кодовой таблице](https://en.wikipedia.org/wiki/Basic_Latin_(Unicode_block)) символ «a» имеет значение U+0061 (в шестнадцатеричной системе [0x61](https://unicodelookup.com/#latin/1)), в то время как символ «C» имеет код U+0043 (0x43), который идет раньше в Unicode-таблице, чем символ «a».  
  
Чтобы отсортировать массив, который может содержать смешанные регистры первых букв, нам необходимо либо преобразовать все элементы временно в нижний регистру, или определить свой порядок сортировки при помощи метода [localeCompare()](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/String/localeCompare) c некоторыми аргументами. Как правило, для такого случая, лучше сразу создать функцию для многократного использования:

function alphaSort(arr) {

arr.sort(function (a, b) {

return a.localeCompare(b, 'en', {'sensitivity': 'base'});

});

}

let fruit = ["butternut squash", "apricot", "Cantaloupe"];

alphaSort(fruit)

> ["apricot", "butternut squash", "Cantaloupe"]

Если вы хотите получить массив отсортированный в обратный алфавитном порядке, просто поменяйте позициями а и b в функции:

function alphaSort(arr) {

arr.sort(function (a, b) {

return b.localeCompare(a, 'en', {'sensitivity': 'base'});

});

}

let fruit = ["butternut squash", "apricot", "Cantaloupe"];

alphaSort(fruit)

> ["Cantaloupe", "butternut squash", "apricot"]

Тут стоит обратить внимание, что localeCompare используется с аргументами, еще надо помнить, что он поддерживается IE11+, для более старых версий IE, мы можем использовать его без аргументов, и в нижнем регистре:

function caseSort(arr) {

arr.sort(function (a, b) {

return a.toLowerCase().localeCompare(b.toLowerCase());

});

}

let fruit = ["butternut squash", "apricot", "Cantaloupe"];

caseSort(fruit)

> ["apricot", "butternut squash", "Cantaloupe"]

### Числовая сортировка

Все это не относится к тому примеру, о котором мы говорили выше про массив игровых рекордов. С некоторыми числовыми массивами сортировка работает просто идеально, но в какой-то момент результат может быть непредсказуемым:

let highScores = [11, 57, 10, 16, 32, 100];

highScores.sort();

> [10, 100, 11, 16, 32, 57]

Дело в том, что метод sort() производит [лексикографическую сравнение](https://ru.wikipedia.org/wiki/%D0%9B%D0%B5%D0%BA%D1%81%D0%B8%D0%BA%D0%BE%D0%B3%D1%80%D0%B0%D1%84%D0%B8%D1%87%D0%B5%D1%81%D0%BA%D0%B8%D0%B9_%D0%BF%D0%BE%D1%80%D1%8F%D0%B4%D0%BE%D0%BA): а это означает, что числа будут преобразованы в строку и сравнения будут снова проводиться путем сопоставления первого символа этой строки в порядке символов Unicode-таблицы. Поэтому нам снова необходимо определить свой порядок сортировки:

let highScores = [11, 57, 10, 16, 32, 100];

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

> [10, 11, 16, 32, 57, 100]

Опять же, для сортировки чисел в обратном порядке, поменяйте позициями a и b в функции.

### Сортировка JSON-подобной структуры

И наконец, если у нас есть [JSON-подобная структура данных](http://thenewcode.com/339/JavaScript-Fundamentals-JSON-Explained), представленная как массив игровых рекордов:

let scores = [

{

"name": "Daniel",

"score": 21768

},

{

"name": "Michael",

"score": 33579

},

{

"name": "Alison",

"score": 38395

}

];

В ES6+, вы можете использовать стрелочные функции:

scores.sort((a, b) => b.score - a.score));

Для старых браузеров, не имеющих такую поддержку:

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

Как видите, сортировка в JavaScript это довольно не очевидная вещь, я надеюсь, что эти примеры облегчат как-нибудь жизнь.

### **К прочтению:**

1. [Сортировка](https://ru.wikipedia.org/wiki/%D0%A1%D0%BE%D1%80%D1%82%D0%B8%D1%80%D0%BE%D0%B2%D0%BA%D0%B0)  
2. [Алгоритм сортировки](https://ru.wikipedia.org/wiki/%D0%90%D0%BB%D0%B3%D0%BE%D1%80%D0%B8%D1%82%D0%BC_%D1%81%D0%BE%D1%80%D1%82%D0%B8%D1%80%D0%BE%D0%B2%D0%BA%D0%B8)  
3. [Пузырьковая сортировка и все-все-все](https://habrahabr.ru/post/204600/)  
4. [Естественная сортировка строк на JavaScript](https://habrahabr.ru/post/127943/)  
5. [Утонченная сортировка в JavaScript](https://www.sitepoint.com/sophisticated-sorting-in-javascript/)  
6. [Javascript Sort Benchmark](https://jsperf.com/javascript-sort)  
7. [Looking for performance? Probably you should NOT use [].sort (V8)](http://blog.mgechev.com/2012/11/24/javascript-sorting-performance-quicksort-v8/)

### Работа со степенными функциями

Возведение в степень — операция, первоначально определяемая как результат многократного умножения натурального числа на себя, квадратный корень из числа a — число, дающее a при возведении в квадрат. Этими функциями мы могли пользоваться постоянно в повседневной жизни на уроках математики, в том числе при вычислении площадей, объемов или даже при физическом моделировании.  
  
В JavaScript степенная функция представлена как Math.pow(), в новом стандарте ES7 был представлен новый оператор возведения в степень — " \* \* ".

### Возведение в степень

Для того, чтобы возвести число в n-ую степень, используйте функцию Math.pow(), где первый аргумент это число, которое будет возведено в степень, второй аргумент это показатель степени:

Math.pow(3,2)

> 9

Такая форма записи означает 3 в квадрате, или 3 × 3, что приводит к результату 9. Можно привести еще пример, конечно:

Math.pow(5,3);

> 125

То есть, 5 в кубе, или 5 × 5 × 5, равно 125.  
  
ECMAScript 7 — это следующая версия JavaScript, в принципе, мы можем использовать новый предложенный оператор возведения в степень — \* \*, такая форма записи может быть более наглядной:

3 \*\* 2

> 9

На данный момент [поддержка этого оператора довольно ограниченная](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Arithmetic_Operators#Exponentiation_(**)), поэтому его не рекомендуется использовать.  
  
Степенная функция может пригодиться в самых разных ситуациях. Простой пример, вычисление количества секунд в часе: Math.pow (60,2).

### Квадратный и кубический корень

Math.sqrt() и Math.cbrt() противоположны функции Math.pow(). Как мы помним, квадратный корень из числа a — число, дающее a при возведении в квадрат.

Math.sqrt(9)

> 3

В тоже время кубический корень из числа a — число, дающее a при возведении в куб.

Math.cbrt(125)

> 5

Math.cbrt() был введен в спецификацию JavaScript совсем недавно, и поэтому поддерживается только в современных браузерах: Chrome 38+, Firefox и Opera 25+ и Safari 7.1+. Вы заметите, что Internet Explorer отсутствует в этом списке, однако на MDN вы найдете [полифилл](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Math/cbrt).

### **Примеры**

Конечно, мы можем использовать и не целые значения в одной из этих функций:

Math.pow(1.25, 2);

> 1.5625

Math.cbrt(56.57)

> 3.8387991760286138

Обратите внимание, что это вполне себе работает и при использовании отрицательных значениях аргументов:

Math.pow(-5,2)

> 25

Math.pow(10,-2)

> 0.01

Тем не менее, для квадратного корня это не будет работать:

Math.sqrt(-9)

> NaN

Из математического анализа мы знаем, что под мнимым числом понимают квадратные корни из отрицательных чисел. И это может привести нас к еще одной технике работы с комплексными числами, но это уже другая история.  
  
Вы можете использовать дробные значения в Math.pow(), чтобы найти квадратные и кубические корни чисел. Квадратный корень использует показатель 0.5:

Math.pow(5, 0.5); *// = Math.sqrt(5) = 5 \*\* (1/2)*

> 2.23606797749979

Однако, из-за капризов с плавающей точкой, вы не можете точно предположить правильный результат:

Math.pow(2.23606797749979,2)

> 5.000000000000001

В таких ситуациях, вы вам придется прибегать к отсечению знаков у числа или округление до какого-либо значения.  
  
Некоторые, по непонятным причинам в JavaScript путают функцию Math.pow() с [Math.exp()](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Math/exp), которая является экспоненциальной функцией для чисел, в целом. Примечание: в английском языке «показатель степени» переводится как «exponent», поэтому это скорее относится к англоговорящим, хотя существуют и альтернативные [названия показателя степени](https://www.mathsisfun.com/images/8-squared.gif), такие как index, power.

### **К прочтению:**

1. [Math.pow(num, 2) vs (self = num) \* self](https://jsperf.com/math-pow-num-2-vs-self-num-self)  
2. [Арифметический квадратный корень из отрицательного числа](http://images.myshared.ru/20/1247251/slide_14.jpg)  
3. [Пример решения квадратного уравнения с отрицательным дискриминантом](http://mat.1september.ru/2001/10/no10_12.gif)  
4. [Обширная математическая библиотека для JavaScript и Node.js](http://mathjs.org/)  
5. [MathML](https://ru.wikipedia.org/wiki/MathML) и [polyfill](https://www.mathjax.org/)

### Математические константы

Работа с математикой в JavaScript облегчается за счет ряда встроенных констант. Эти константы являются свойствами объекта Math. Стоит обратить внимание, что константы пишутся в верхнем регистре, а не [CamelCase](https://ru.wikipedia.org/wiki/CamelCase) нотации.

### Math.PI

[Число Пи](https://ru.wikipedia.org/wiki/%D0%9F%D0%B8_(%D1%87%D0%B8%D1%81%D0%BB%D0%BE)) — математическая константа, равная отношению длины окружности к длине её диаметра. Старое название — лудольфово число. Пи — иррациональное число, то есть его значение не может быть точно выражено в виде дроби m/n, где m и n — целые числа. Следовательно, его десятичное представление никогда не заканчивается и не является периодическим. На бумаге популярно использовать его короткую форму записи — 3.14159.  
  
Ваш браузер не может хранить бесконечное число, так что JavaScript округляет число Пи до такого значения 3,141592653589793, что более чем достаточно для большинства задач.  
  
Хотя это число наиболее распространено в задачах на нахождении длин, площади окружности, число Пи также используется в вероятности, статистики, инженерных и естественных науках: это как универсальная константа.

### Math.SQRT2

[Квадратный корень из числа 2](https://upload.wikimedia.org/wikipedia/commons/thumb/a/a6/Square_root_of_2_triangle.svg/174px-Square_root_of_2_triangle.svg.png) — положительное вещественное число, которое при умножении само на себя даёт число 2. Геометрически корень из 2 можно представить как длину диагонали квадрата со стороной 1 (это следует из теоремы Пифагора). Это было первое известное в истории математики иррациональное число. JavaScript округляет это число до такого значения 1.4142135623730951. (Из-за ошибок округления в JavaScript: Math.SQRT2 \* Math.SQRT2 не равно 2).

### Math.SQRT1\_2

Квадратный корень из 0.5 — это единица, деленная на корень квадратный из 2. И опять же, это иррациональное число.  
  
Простейшими преобразованиями на бумаге мы можем записать это так:

√(1/2) = √1 / √2 = 1 / √2 = √2 / 2

Но из-за проблем с плавающей точкой, мы можем получить такой результат:

Math.SQRT1\_2 === Math.sqrt(1/2)

> true

Math.sqrt(2) / 2 === Math.sqrt(1/2)

> true

1 / Math.sqrt(2) === Math.sqrt(1/2)

> false

### Math.E

Как ни странно, в [математике константа е](https://ru.wikipedia.org/wiki/E_(%D1%87%D0%B8%D1%81%D0%BB%D0%BE)) всегда записывалась в нижнем регистре, в JavaScript же это число используют в верхнем регистре. Число e — основание натурального логарифма, математическая константа, иррациональное и трансцендентное число. Иногда число e называют числом Эйлера или числом Непера. JavaScript округляет его как 2,718281828459045. Число e играет важную роль в дифференциальном и интегральном исчислении, а также во многих других разделах математики.

Math.pow(Math.E,1)

> 2.718281828459045

Math.pow(Math.E,2)

> 7.3890560989306495

Math.pow(Math.E,3)

> 20.085536923187664

### **Натуральный логарифм**

Натуральный логарифм — это логарифм по основанию e, где e — иррациональная константа, равная приблизительно 2,718281828. Натуральный логарифм числа x — это показатель степени, в которую нужно возвести число e, чтобы получить x. Math.log(х) — это натуральный из x по основанию e.

Math.log(-1); *// NaN, out of range*

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

Math.log(1); *// 0*

Math.log(10); *// 2.302585092994046*

Если вам нужно получить логарифм из y по основанию x:

function getBaseLog(x, y) {

return Math.log(y) / Math.log(x);

}

getBaseLog(1/5, 5)

> -1

Однако, из-за особенностей округления чисел с плавающей точкой, ответ получается не всегда точный и только близок к правильному значению:

getBaseLog(10, 1000)

> 2.9999999999999996

### Math.LN2

Свойство Math.LN2 представляет натуральный логарифм из 2 равный 0.6931471805599453.

### Math.LN10

Свойство Math.LN10 представляет натуральный логарифм из 10 равный 2.302585092994046

### Math.LOG2E

Свойство Math.LOG2E представляет двоичный логарифм из e равный 1.4426950408889634

### Math.LOG10E

Свойство Math.LOG10E представляет десятичный логарифм из e равный 0.4342944819032518

### **К прочтению:**

1. [Математическая константа](https://ru.wikipedia.org/wiki/%D0%9C%D0%B0%D1%82%D0%B5%D0%BC%D0%B0%D1%82%D0%B8%D1%87%D0%B5%D1%81%D0%BA%D0%B0%D1%8F_%D0%BA%D0%BE%D0%BD%D1%81%D1%82%D0%B0%D0%BD%D1%82%D0%B0)  
2. [10 чисел, на которых держится мир](http://expert.ru/russian_reporter/2008/19/cisla/)  
3. [Математическую константу положили на музыку](https://lenta.ru/news/2011/06/28/tau/)

### Math.abs, parseInt, parseFloat

Работа с числами в JavaScript может быть куда более сложной, чем кажется. Полученные значения не всегда попадают внутрь ожидаемых диапазонов, иногда результат может оказаться вовсе не тем, что мы ожидали.

### Math.abs()

Метод Math.abs() возвращает абсолютное значение числа, что напоминает нам аналогичную математическую функцию[модуля числа a](http://math-prosto.ru/?page=pages/modulus/modulus_of_number.php).

let newVal = -57.64;

Math.abs(newVal);

> 57.64

Math.abs(0) всегда возвращает нуль, но если поставить знак минус перед функцией -Math.abs(NUM) мы всегда будем получать отрицательное значение.

-Math.abs(0);

> -0

### parseInt()

Мы знаем, что JavaScript понимает, что «15» это строка, а не число и, например, при разборе CSS-свойств средствами JavaScript, или получив какое-либо значение из неподготовленного массива, наши результаты могут получиться непредсказуемыми. Мы могли получить на вход строку представленную как «17px», и для нас это не является редкостью. Вопрос заключается в том, как преобразовать эту строку в фактическое значение и использовать его в дальнейших расчетах.  
  
Синтаксис: parseInt(string, radix);  
  
Функция parseInt преобразует первый переданный ей аргумент в строковый тип, интерпретирует его и возвращает целое число или значение NaN. Результат (если не NaN) является целым числом и представляет собой первый аргумент (string), рассматривающийся как число в указанной системе счисления (radix). Например, основание 10 указывает на преобразование из десятичного числа, 8 — восьмеричного, 16 — шестнадцатеричного и так далее. Если основание больше 10, то для обозначения цифр больше 9 используются буквы. Например, для шестнадцатеричных чисел (основание 16) используются буквы от A до F.  
  
Рассмотрим пример работы с CSS-свойствами, где, условно говоря, мы можем получить такое значение:

let elem = document.body;

let centerPoint = window.getComputedStyle(elem).transformOrigin;

> "454px 2087.19px"

Мы можем разделить значения по пробелам:

let centers = centerPoint.split(" ");

> ["454px", "2087.19px"]

Однако, каждый элемент все еще есть строка, мы можем избавиться от этого применив нашу функцию:

let centerX = parseInt(centers[0], 10);

> 454

let centerY = parseInt(centers[1], 10);

> 2087

Как видите, вторым аргументом мы указываем систему счисления, в которую будет преобразовано число, этот параметр необязательный, но его рекомендуется использовать, в случае, если вы не знаете какая строка поступит на вход.

### parseFloat()

Из примера выше, вы наверное заметили, что parseInt отбрасывает дробную часть. В нашем случае, parseFloat умеет работать с числами с плавающей точкой. Опять же, это может быть полезным при разборе CSS и других задачах, особенно при работе с плавающей точкой в ​​процентах.  
  
Синтаксис: parseFloat(string)

let FP = "33.33333%";

console.log(parseFloat(FP));

> 33.33333

Обратите внимание, что в синтаксисе parseFloat нет второго аргумента.  
  
Мы понимаем, что parseInt() и parseFloat() являются чрезвычайно полезными функциями, важно учитывать, что и тут не обойтись без ошибок, поэтому необходимо проверять диапазон ожидаемых значений и в конечном счете анализировать результат, чтобы гарантировать, что полученные значения верны.

# Шаблонные строки

Шаблонными литералами называются строковые литералы, допускающие использование выражений внутри. С ними вы можете использовать многострочные литералы и строковую интерполяцию. В спецификациях до ES2015 они назывались "шаблонными строками".

## [Синтаксис](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Template_literals#syntax)

`строка текста`

`строка текста 1

строка текста 2`

`строка текста ${выражение} строка текста`

tag `строка текста ${выражение} строка текста`

## [Описание](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Template_literals#description)

Шаблонные литералы заключены в обратные кавычки (` `) вместо двойных или одинарных. Они могут содержать подстановки, обозначаемые знаком доллара и фигурными скобками (${выражение}). Выражения в подстановках и текст между ними передаются в функцию. По умолчанию функция просто объединяет все части в строку. Если перед строкой есть выражение (здесь это tag), то шаблонная строка называется "теговым шаблоном". В этом случае, теговое выражение (обычно функция) вызывается с обработанным шаблонным литералом, который вы можете изменить перед выводом. Для экранирования обратной кавычки в шаблонных литералах указывается обратный слеш **\**.

`\`` === '`' // --> true

### [**Многострочные литералы**](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Template_literals#%D0%BC%D0%BD%D0%BE%D0%B3%D0%BE%D1%81%D1%82%D1%80%D0%BE%D1%87%D0%BD%D1%8B%D0%B5_%D0%BB%D0%B8%D1%82%D0%B5%D1%80%D0%B0%D0%BB%D1%8B)

Символы новой строки являются частью шаблонных литералов. Используя обычные строки, вставка переноса потребовала бы следующего синтаксиса:

console.log('string text line 1\n' +

'string text line 2');

// "string text line 1

// string text line 2"

То же с использованием шаблонных литералов:

console.log(`string text line 1

string text line 2`);

// "string text line 1

// string text line 2"

### [**Интерполяция выражений**](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Template_literals#%D0%B8%D0%BD%D1%82%D0%B5%D1%80%D0%BF%D0%BE%D0%BB%D1%8F%D1%86%D0%B8%D1%8F_%D0%B2%D1%8B%D1%80%D0%B0%D0%B6%D0%B5%D0%BD%D0%B8%D0%B9)

Для вставки выражений в обычные строки вам пришлось бы использовать следующий синтаксис:

var a = 5;

var b = 10;

console.log('Fifteen is ' + (a + b) + ' and not ' + (2 \* a + b) + '.');

// "Fifteen is 15 and not 20."

Теперь, при помощи шаблонных литералов, вам доступен "синтаксический сахар", делающий подстановки вроде той более читабельными:

var a = 5;

var b = 10;

console.log(`Fifteen is ${a + b} and not ${2 \* a + b}.`);

// "Fifteen is 15 and not 20."

### [**Вложенные шаблоны**](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Template_literals#%D0%B2%D0%BB%D0%BE%D0%B6%D0%B5%D0%BD%D0%BD%D1%8B%D0%B5_%D1%88%D0%B0%D0%B1%D0%BB%D0%BE%D0%BD%D1%8B)

Временами, вложить шаблон — это кратчайший и, возможно, более читабельный способ составить строку. Просто поместите внутрь шаблона с обратными кавычками ещё одни, обернув их в подстановку ${ }. Например, если выражение истинно, можно вернуть шаблонный литерал.

В ES5:

var classes = 'header'

classes += (isLargeScreen() ?

'' : item.isCollapsed ?

' icon-expander' : ' icon-collapser');

В ES2015 с шаблонными литералами без вложения:

const classes = `header ${ isLargeScreen() ? '' :

(item.isCollapsed ? 'icon-expander' : 'icon-collapser') }`;

В ES2015 с вложенными шаблонными литералами:

const classes = `header ${ isLargeScreen() ? '' :

`icon-${item.isCollapsed ? 'expander' : 'collapser'}` }`;

### [**Теговые шаблоны**](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Template_literals#%D1%82%D0%B5%D0%B3%D0%BE%D0%B2%D1%8B%D0%B5_%D1%88%D0%B0%D0%B1%D0%BB%D0%BE%D0%BD%D1%8B)

Расширенной формой шаблонных литералов являются теговые шаблоны. Они позволяют разбирать шаблонные литералы с помощью функции. Первый аргумент такой функции содержит массив строковых значений, а остальные содержат выражения из подстановок. В итоге, функция должна вернуть собранную строку (или что-либо совсем иное, как будет показано далее). Имя функции может быть любым.

var person = 'Mike';

var age = 28;

function myTag(strings, personExp, ageExp) {

var str0 = strings[0]; // "That "

var str1 = strings[1]; // " is a "

// Технически, в конце итогового выражения

// (в нашем примере) есть ещё одна строка,

// но она пустая (""), так что пропустим её.

// var str2 = strings[2];

var ageStr;

if (ageExp > 99){

ageStr = 'centenarian';

} else {

ageStr = 'youngster';

}

// Мы даже можем вернуть строку, построенную другим шаблонным литералом

return `${str0}${personExp}${str1}${ageStr}`;

}

var output = myTag`That ${ person } is a ${ age }`;

console.log(output);

// That Mike is a youngster

Функция тега не обязана возвращать строку, как показано в примере ниже:

function template(strings, ...keys) {

return (function(...values) {

var dict = values[values.length - 1] || {};

var result = [strings[0]];

keys.forEach(function(key, i) {

var value = Number.isInteger(key) ? values[key] : dict[key];

result.push(value, strings[i + 1]);

});

return result.join('');

});

}

var t1Closure = template`${0}${1}${0}!`;

t1Closure('Y', 'A'); // "YAY!"

var t2Closure = template`${0} ${'foo'}!`;

t2Closure('Hello', {foo: 'World'}); // "Hello World!"

### [**Сырые строки**](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Template_literals#%D1%81%D1%8B%D1%80%D1%8B%D0%B5_%D1%81%D1%82%D1%80%D0%BE%D0%BA%D0%B8)

Специальное свойство raw, доступное для первого аргумента тегового шаблона, позволяет получить строку в том виде, в каком она была введена, без [экранирования](https://developer.mozilla.org/ru/docs/Web/JavaScript/Guide/Grammar_and_types#%D0%B8%D1%81%D0%BF%D0%BE%D0%BB%D1%8C%D0%B7%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5_%D1%81%D0%BF%D0%B5%D1%86%D0%B8%D0%B0%D0%BB%D1%8C%D0%BD%D1%8B%D1%85_%D1%81%D0%B8%D0%BC%D0%B2%D0%BE%D0%BB%D0%BE%D0%B2_%D0%B2_%D1%81%D1%82%D1%80%D0%BE%D0%BA%D0%B0%D1%85).

function tag(strings) {

return strings.raw[0];

}

tag`string text line 1 \\n string text line 2`;

// выводит "string text line 1 \\n string text line 2",

// включая 'n' и два символа '\'

Вдобавок, существует метод [String.raw()](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/String/raw), возвращающий точно такую же исходную строку, какую вернула бы функция шаблона по умолчанию и строковая конкатенация вместе.

var str = String.raw`Hi\n${2+3}!`;

// "Hi\n5!"

str.length;

// 6

str.split('').join(',');

// "H,i,\,n,5,!"

### [**Теговые шаблоны и экранирование символов**](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Template_literals#%D1%82%D0%B5%D0%B3%D0%BE%D0%B2%D1%8B%D0%B5_%D1%88%D0%B0%D0%B1%D0%BB%D0%BE%D0%BD%D1%8B_%D0%B8_%D1%8D%D0%BA%D1%80%D0%B0%D0%BD%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5_%D1%81%D0%B8%D0%BC%D0%B2%D0%BE%D0%BB%D0%BE%D0%B2)

#### **Поведение в ES2016**

В ECMAScript 2016 теговые шаблоны следуют правилам экранирования следующих символов:

* символы Unicode, начинающиеся с "\u", например, \u00A9
* точки кода Unicode, начинающиеся с "\u{}", например, \u{2F804}
* шестнадцатеричные представления символов, начинающиеся с "\x", например, \xA9
* восьмеричные представления символов, начинающиеся с "\", например, \251​​​​​​

Отсюда вытекает проблема теговых шаблонов: следуя грамматике ECMAScript, анализатор кода, найдя символ \, будет искать корректное представление символа Unicode, но может не найти его вовсе. Пример ниже показывает это:

latex`\unicode`

// В старых версиях ECMAScript (ES2016 и раньше) выкинет исключение:

// SyntaxError: malformed Unicode character escape sequence

#### **Поведение в ES2018**

Теговые шаблоны должны позволять встраивать языки (например, [DSLs](https://ru.wikipedia.org/wiki/%D0%9F%D1%80%D0%B5%D0%B4%D0%BC%D0%B5%D1%82%D0%BD%D0%BE-%D0%BE%D1%80%D0%B8%D0%B5%D0%BD%D1%82%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%BD%D1%8B%D0%B9_%D1%8F%D0%B7%D1%8B%D0%BA) или [LaTeX](https://ru.wikipedia.org/wiki/LaTeX)), в которых широко используются многие другие экранирования. Предложение [Редакция шаблонных литералов](https://tc39.github.io/proposal-template-literal-revision/) (уровень 4, одобренный к добавлению в стандарт ECMAScript 2018) устраняет синтаксические ограничения экранирования теговых шаблонов в ECMAScript.

Однако, некорректное экранирование символов по-прежнему нужно отображать в "приготовленном" отображении. Оно показывается в виде [undefined](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/undefined) в "приготовленном" массиве:

function latex(str) {

return { "cooked": str[0], "raw": str.raw[0] }

}

latex`\unicode`

// { cooked: undefined, raw: "\unicode" }

Заметьте, что ограничение на экранирование символов проявляется лишь в теговых шаблонах, и не проявляется в нетеговых шаблонных литералах:

let bad = `bad escape sequence: \unicode`;

## [Спецификации](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Template_literals#%D1%81%D0%BF%D0%B5%D1%86%D0%B8%D1%84%D0%B8%D0%BA%D0%B0%D1%86%D0%B8%D0%B8)

|  |  |  |
| --- | --- | --- |
| **Спецификация** | **Статус** | **Комментарий** |
| [ECMAScript 2015 (6th Edition, ECMA-262) Определение 'Template Literals' в этой спецификации.](https://www.ecma-international.org/ecma-262/6.0/#sec-template-literals) | Стандарт | Изначальное определение. Определено в секциях [Template Literals](https://www.ecma-international.org/ecma-262/6.0/#sec-template-literals), [Tagged Templates](https://www.ecma-international.org/ecma-262/6.0/#sec-tagged-templates) |
| [ECMAScript (ECMA-262) Определение 'Template Literals' в этой спецификации.](https://tc39.es/ecma262/#sec-template-literals) | Живой стандарт | Определено в секциях [Template Literals](https://tc39.github.io/ecma262/#sec-template-literals), [Tagged Templates](https://tc39.github.io/ecma262/#sec-tagged-templates) |
| [Template Literal Revision](https://tc39.github.io/proposal-template-literal-revision/) | Черновик 4-го уровня | Устранено ограничение экранирования в теговых шаблонах |

# Методы массивов

Массивы предоставляют множество методов. Чтобы было проще, в этой главе они разбиты на группы.

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

Мы уже знаем методы, которые добавляют и удаляют элементы из начала или конца:

* 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.

### [splice](https://learn.javascript.ru/array-methods" \l "splice)

Как удалить элемент из массива?

Так как массивы – это объекты, то можно попробовать delete:

let arr = ["I", "go", "home"];

delete arr[1]; // удалить "go"

alert( arr[1] ); // undefined

// теперь arr = ["I", , "home"];

alert( arr.length ); // 3

Вроде бы, элемент и был удалён, но при проверке оказывается, что массив всё ещё имеет 3 элемента arr.length == 3.

Это нормально, потому что всё, что делает delete obj.key – это удаляет значение с данным ключом key. Это нормально для объектов, но для массивов мы обычно хотим, чтобы оставшиеся элементы сдвинулись и заняли освободившееся место. Мы ждём, что массив станет короче.

Поэтому для этого нужно использовать специальные методы.

Метод [arr.splice(str)](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/Array/splice) – это универсальный «швейцарский нож» для работы с массивами. Умеет всё: 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.

Этот метод проще всего понять, рассмотрев примеры.

Начнём с удаления:

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

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

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

Легко, правда? Начиная с позиции 1, он убрал 1 элемент.

В следующем примере мы удалим 3 элемента и заменим их двумя другими.

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

// удалить 3 первых элемента и заменить их другими

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

// вставить "сложный", "язык"

arr.splice(2, 0, "сложный", "язык");

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

**Отрицательные индексы разрешены**

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

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

### [slice](https://learn.javascript.ru/array-methods" \l "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.

Это похоже на строковый метод str.slice, но вместо подстрок возвращает подмассивы.

Например:

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

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

alert( arr.slice(-2) ); // s,t (копирует с -2 до конца)

Можно вызвать slice и вообще без аргументов: arr.slice() создаёт копию массива arr. Это часто используют, чтобы создать копию массива для дальнейших преобразований, которые не должны менять исходный массив.

### [concat](https://learn.javascript.ru/array-methods" \l "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 мы получаем 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];

// создать массив из: arr и [3,4]

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

// создать массив из: arr и [3,4] и [5,6]

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

// создать массив из: arr и [3,4], потом добавить значения 5 и 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]

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

Для корректной обработки в объекте должны быть числовые свойства и length:

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: forEach](https://learn.javascript.ru/array-methods" \l "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) {

// ... делать что-то с item

});

Например, этот код выведет на экран каждый элемент массива:

// Вызов alert для каждого элемента

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

А этот вдобавок расскажет и о своей позиции 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.

## [Поиск в массиве](https://learn.javascript.ru/array-methods" \l "poisk-v-massive)

Далее рассмотрим методы, которые помогут найти что-нибудь в массиве.

### [indexOf/lastIndexOf и includes](https://learn.javascript.ru/array-methods" \l "indexof-lastindexof-i-includes)

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 мы ищем false, он находит именно false, а не ноль.

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

Кроме того, очень незначительным отличием includes является то, что он правильно обрабатывает NaN в отличие от indexOf/lastIndexOf:

const arr = [NaN];

alert( arr.indexOf(NaN) ); // -1 (должен быть 0, но === проверка на равенство не работает для NaN)

alert( arr.includes(NaN) );// true (верно)

### [find и findIndex](https://learn.javascript.ru/array-methods" \l "find-i-findindex)

Представьте, что у нас есть массив объектов. Как нам найти объект с определённым условием?

Здесь пригодится метод [arr.find](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/Array/find).

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 возвращает true, поиск прерывается и возвращается item. Если ничего не найдено, возвращается undefined.

Например, у нас есть массив пользователей, каждый из которых имеет поля id и name. Попробуем найти 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

В реальной жизни массивы объектов – обычное дело, поэтому метод find крайне полезен.

Обратите внимание, что в данном примере мы передаём find функцию item => item.id == 1, с одним аргументом. Это типично, дополнительные аргументы этой функции используются редко.

Метод [arr.findIndex](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/Array/findIndex) – по сути, то же самое, но возвращает индекс, на котором был найден элемент, а не сам элемент, и -1, если ничего не найдено.

### [filter](https://learn.javascript.ru/array-methods" \l "filter)

Метод find ищет один (первый попавшийся) элемент, на котором функция-колбэк вернёт true.

На тот случай, если найденных элементов может быть много, предусмотрен 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" \l "preobrazovanie-massiva)

Перейдём к методам преобразования и упорядочения массива.

### [map](https://learn.javascript.ru/array-methods" \l "map)

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, здесь мы преобразуем каждый элемент в его длину:

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

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

### [sort(fn)](https://learn.javascript.ru/array-methods" \l "sort-fn)

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 ];

// метод сортирует содержимое arr

arr.sort();

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

Не заметили ничего странного в этом примере?

Порядок стал 1, 15, 2. Это неправильно! Но почему?

**По умолчанию элементы сортируются как строки.**

Буквально, элементы преобразуются в строки при сравнении. Для строк применяется лексикографический порядок, и действительно выходит, что "2" > "15".

Чтобы использовать наш собственный порядок сортировки, нам нужно предоставить функцию в качестве аргумента arr.sort().

Функция должна для пары значений возвращать:

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

Теперь всё работает как надо.

Давайте возьмём паузу и подумаем, что же происходит. Упомянутый ранее массив arr может быть массивом чего угодно, верно? Он может содержать числа, строки, объекты или что-то ещё. У нас есть набор каких-то элементов. Чтобы отсортировать его, нам нужна функция, определяющая порядок, которая знает, как сравнивать его элементы. По умолчанию элементы сортируются как строки.

Метод arr.sort(fn) реализует общий алгоритм сортировки. Нам не нужно заботиться о том, как он работает внутри (в большинстве случаев это оптимизированная [быстрая сортировка](https://ru.wikipedia.org/wiki/%D0%91%D1%8B%D1%81%D1%82%D1%80%D0%B0%D1%8F_%D1%81%D0%BE%D1%80%D1%82%D0%B8%D1%80%D0%BE%D0%B2%D0%BA%D0%B0)). Она проходится по массиву, сравнивает его элементы с помощью предоставленной функции и переупорядочивает их. Всё, что остаётся нам, это предоставить fn, которая делает это сравнение.

Кстати, если мы когда-нибудь захотим узнать, какие элементы сравниваются – ничто не мешает нам вывести их на экран:

[1, -2, 15, 2, 0, 8].sort(function(a, b) {

alert( a + " <> " + b );

});

В процессе работы алгоритм может сравнивать элемент с другими по нескольку раз, но он старается сделать как можно меньше сравнений.

**Функция сравнения может вернуть любое число**

На самом деле от функции сравнения требуется любое положительное число, чтобы сказать «больше», и отрицательное число, чтобы сказать «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

**Лучше использовать стрелочные функции**

Помните [стрелочные функции](https://learn.javascript.ru/arrow-functions-basics)? Можно использовать их здесь для того, чтобы сортировка выглядела более аккуратной:

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

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

### [reverse](https://learn.javascript.ru/array-methods" \l "reverse)

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.

### [split и join](https://learn.javascript.ru/array-methods" \l "split-i-join)

Ситуация из реальной жизни. Мы пишем приложение для обмена сообщениями, и посетитель вводит имена тех, кому его отправить, через запятую: Вася, Петя, Маша. Но нам-то гораздо удобнее работать с массивом имён, чем с одной строкой. Как его получить?

Метод [str.split(delim)](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/String/split) именно это и делает. Он 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}.` ); // Сообщение получат: Вася (и другие имена)

}

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

**Разбивка по буквам**

Вызов split(s) с пустым аргументом s разбил бы 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) делает в точности 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

### [reduce/reduceRight](https://learn.javascript.ru/array-methods" \l "reduce-reduceright)

Если нам нужно перебрать массив – мы можем использовать forEach, for или for..of.

Если нам нужно перебрать массив и вернуть данные для каждого элемента – мы 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 – его индекс,
* array – сам 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.

Тут мы получим сумму всех элементов массива всего одной строкой:

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

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

alert(result); // 15

Здесь мы использовали наиболее распространённый вариант reduce, который использует только 2 аргумента.

Давайте детальнее разберём, как он работает.

1. При первом запуске sum равен initial (последний аргумент reduce), то есть 0, а current – первый элемент массива, равный 1. Таким образом, результат функции равен 1.
2. При втором запуске sum = 1, и к нему мы добавляем второй элемент массива (2).
3. При третьем запуске sum = 3, к которому мы добавляем следующий элемент, и так далее…

Поток вычислений получается такой:

В виде таблицы, 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.

Мы также можем опустить начальное значение:

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

// убрано начальное значение (нет 0 в конце)

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

alert( result ); // 15

Результат – точно такой же! Это 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.

Вот пример:

let arr = [];

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

// если бы существовало начальное значение, reduce вернул бы его для пустого массива.

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

Поэтому рекомендуется всегда указывать начальное значение.

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

## [Array.isArray](https://learn.javascript.ru/array-methods" \l "array-isarray)

Массивы не образуют отдельный тип языка. Они основаны на объектах.

Поэтому typeof не может отличить простой объект от массива:

alert(typeof {}); // object

alert(typeof []); // тоже object

…Но массивы используются настолько часто, что для этого придумали специальный метод: [Array.isArray(value)](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/Array/isArray). Он возвращает true, если value массив, и false, если нет.

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

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

## [Большинство методов поддерживают «thisArg»](https://learn.javascript.ru/array-methods" \l "bolshinstvo-metodov-podderzhivayut-thisarg)

Почти все методы массива, которые вызывают функции – такие как find, filter, map, за исключением метода sort, принимают необязательный параметр thisArg.

Этот параметр не объяснялся выше, так как очень редко используется, но для наиболее полного понимания темы мы обязаны его рассмотреть.

Вот полный синтаксис этих методов:

arr.find(func, thisArg);

arr.filter(func, thisArg);

arr.map(func, thisArg);

// ...

// thisArg - это необязательный последний аргумент

Значение параметра thisArg становится this для func.

Например, вот тут мы используем метод объекта army как фильтр, и thisArg передаёт ему контекст:

let army = {

minAge: 18,

maxAge: 27,

canJoin(user) {

return user.age >= this.minAge && user.age < this.maxAge;

}

};

let users = [

{age: 16},

{age: 20},

{age: 23},

{age: 30}

];

// найти пользователей, для которых army.canJoin возвращает true

let soldiers = users.filter(army.canJoin, army);

alert(soldiers.length); // 2

alert(soldiers[0].age); // 20

alert(soldiers[1].age); // 23

Если бы мы в примере выше использовали просто users.filter(army.canJoin), то вызов army.canJoin был бы в режиме отдельной функции, с this=undefined. Это тут же привело бы к ошибке.

Вызов users.filter(army.canJoin, army) можно заменить на users.filter(user => army.canJoin(user)), который делает то же самое. Последняя запись используется даже чаще, так как функция-стрелка более наглядна.

## [Итого](https://learn.javascript.ru/array-methods#itogo)

Шпаргалка по методам массива:

* Для добавления/удаления элементов:
  + push (...items) – добавляет элементы в конец,
  + pop() – извлекает элемент с конца,
  + shift() – извлекает элемент с начала,
  + unshift(...items) – добавляет элементы в начало.
  + splice(pos, deleteCount, ...items) – начиная с индекса pos, удаляет deleteCount элементов и вставляет items.
  + slice(start, end) – создаёт новый массив, копируя в него элементы с позиции start до end (не включая end).
  + concat(...items) – возвращает новый массив: копирует все члены текущего массива и добавляет к нему items. Если какой-то из items является массивом, тогда берутся его элементы.
* Для поиска среди элементов:
  + indexOf/lastIndexOf(item, pos) – ищет item, начиная с позиции pos, и возвращает его индекс или -1, если ничего не найдено.
  + includes(value) – возвращает true, если в массиве имеется элемент value, в противном случае false.
  + find/filter(func) – фильтрует элементы через функцию и отдаёт первое/все значения, при прохождении которых через функцию возвращается true.
  + findIndex похож на find, но возвращает индекс вместо значения.
* Для перебора элементов:
  + forEach(func) – вызывает func для каждого элемента. Ничего не возвращает.
* Для преобразования массива:
  + map(func) – создаёт новый массив из результатов вызова func для каждого элемента.
  + sort(func) – сортирует массив «на месте», а потом возвращает его.
  + reverse() – «на месте» меняет порядок следования элементов на противоположный и возвращает изменённый массив.
  + split/join – преобразует строку в массив и обратно.
  + reduce(func, initial) – вычисляет одно значение на основе всего массива, вызывая func для каждого элемента и передавая промежуточный результат между вызовами.
* Дополнительно:
  + Array.isArray(arr) проверяет, является ли arr массивом.

Обратите внимание, что методы sort, reverse и splice изменяют исходный массив.

Изученных нами методов достаточно в 99% случаев, но существуют и другие.

* [arr.some(fn)](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/Array/some)/[arr.every(fn)](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/Array/every) проверяет массив.

Функция fn вызывается для каждого элемента массива аналогично map. Если какие-либо/все результаты вызовов являются true, то метод возвращает true, иначе false.

* [arr.fill(value, start, end)](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/Array/fill) – заполняет массив повторяющимися value, начиная с индекса start до end.
* [arr.copyWithin(target, start, end)](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/Array/copyWithin) – копирует свои элементы, начиная со start и заканчивая end, в собственную позицию target (перезаписывает существующие).

Полный список есть в [справочнике MDN](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/Array).

На первый взгляд может показаться, что существует очень много разных методов, которые довольно сложно запомнить. Но это гораздо проще, чем кажется.

Внимательно изучите шпаргалку, представленную выше, а затем, чтобы попрактиковаться, решите задачи, предложенные в данной главе. Так вы получите необходимый опыт в правильном использовании методов массива.

Всякий раз, когда вам будет необходимо что-то сделать с массивом, а вы не знаете, как это сделать – приходите сюда, смотрите на таблицу и ищите правильный метод. Примеры помогут вам всё сделать правильно, и вскоре вы быстро запомните методы без особых усилий.

## [Задачи](https://learn.javascript.ru/array-methods#tasks)

### [Переведите текст вида border-left-width в borderLeftWidth](https://learn.javascript.ru/array-methods" \l "perevedite-tekst-vida-border-left-width-v-borderleftwidth)

важность: 5

Напишите функцию camelize(str), которая преобразует строки вида «my-short-string» в «myShortString».

То есть дефисы удаляются, а все слова после них получают заглавную букву.

Примеры:

camelize("background-color") == 'backgroundColor';

camelize("list-style-image") == 'listStyleImage';

camelize("-webkit-transition") == 'WebkitTransition';

P.S. Подсказка: используйте split, чтобы разбить строку на массив символов, потом переделайте всё как нужно и методом join соедините обратно.

[Открыть песочницу с тестами для задачи.](https://plnkr.co/edit/1AmoAeEI1BVkl5mQ?p=preview)

решение

### [Фильтрация по диапазону](https://learn.javascript.ru/array-methods" \l "filtratsiya-po-diapazonu)

важность: 4

Напишите функцию filterRange(arr, a, b), которая принимает массив arr, ищет в нём элементы между a и b и отдаёт массив этих элементов.

Функция должна возвращать новый массив и не изменять исходный.

Например:

let arr = [5, 3, 8, 1];

let filtered = filterRange(arr, 1, 4);

alert( filtered ); // 3,1 (совпадающие значения)

alert( arr ); // 5,3,8,1 (без изменений)

[Открыть песочницу с тестами для задачи.](https://plnkr.co/edit/Tt8sRbALQYQ7YwxH?p=preview)

решение

### [Фильтрация по диапазону "на месте"](https://learn.javascript.ru/array-methods" \l "filtratsiya-po-diapazonu-na-meste)

важность: 4

Напишите функцию filterRangeInPlace(arr, a, b), которая принимает массив arr и удаляет из него все значения кроме тех, которые находятся между a и b. То есть, проверка имеет вид a ≤ arr[i] ≤ b.

Функция должна изменять принимаемый массив и ничего не возвращать.

Например:

let arr = [5, 3, 8, 1];

filterRangeInPlace(arr, 1, 4); // удалены числа вне диапазона 1..4

alert( arr ); // [3, 1]

[Открыть песочницу с тестами для задачи.](https://plnkr.co/edit/Tm408Cmi3OgRvQHo?p=preview)

решение

### [Сортировать в порядке по убыванию](https://learn.javascript.ru/array-methods" \l "sortirovat-v-poryadke-po-ubyvaniyu)

важность: 4

let arr = [5, 2, 1, -10, 8];

// ... ваш код для сортировки по убыванию

alert( arr ); // 8, 5, 2, 1, -10

решение

### [Скопировать и отсортировать массив](https://learn.javascript.ru/array-methods" \l "skopirovat-i-otsortirovat-massiv)

важность: 5

У нас есть массив строк arr. Нужно получить отсортированную копию, но оставить arr неизменённым.

Создайте функцию copySorted(arr), которая будет возвращать такую копию.

let arr = ["HTML", "JavaScript", "CSS"];

let sorted = copySorted(arr);

alert( sorted ); // CSS, HTML, JavaScript

alert( arr ); // HTML, JavaScript, CSS (без изменений)

решение

### [Создать расширяемый калькулятор](https://learn.javascript.ru/array-methods" \l "sozdat-rasshiryaemyy-kalkulyator)

важность: 5

Создайте функцию конструктор Calculator, которая создаёт «расширяемые» объекты калькулятора.

Задание состоит из двух частей.

1. Во-первых, реализуйте метод calculate(str), который принимает строку типа "1 + 2" в формате «ЧИСЛО оператор ЧИСЛО» (разделено пробелами) и возвращает результат. Метод должен понимать плюс + и минус -.

Пример использования:

let calc = new Calculator;

alert( calc.calculate("3 + 7") ); // 10

1. Затем добавьте метод addMethod(name, func), который добавляет в калькулятор новые операции. Он принимает оператор name и функцию с двумя аргументами func(a,b), которая описывает его.

Например, давайте добавим умножение \*, деление / и возведение в степень \*\*:

let powerCalc = new Calculator;

powerCalc.addMethod("\*", (a, b) => a \* b);

powerCalc.addMethod("/", (a, b) => a / b);

powerCalc.addMethod("\*\*", (a, b) => a \*\* b);

let result = powerCalc.calculate("2 \*\* 3");

alert( result ); // 8

* Для этой задачи не нужны скобки или сложные выражения.
* Числа и оператор разделены ровно одним пробелом.
* Не лишним будет добавить обработку ошибок.

[Открыть песочницу с тестами для задачи.](https://plnkr.co/edit/iFtIRenz8X4qjVkm?p=preview)

решение

### [Трансформировать в массив имён](https://learn.javascript.ru/array-methods" \l "transformirovat-v-massiv-imyon)

важность: 5

У вас есть массив объектов user, и в каждом из них есть user.name. Напишите код, который преобразует их в массив имён.

Например:

let vasya = { name: "Вася", age: 25 };

let petya = { name: "Петя", age: 30 };

let masha = { name: "Маша", age: 28 };

let users = [ vasya, petya, masha ];

let names = /\* ... ваш код \*/

alert( names ); // Вася, Петя, Маша

решение

### [Трансформировать в объекты](https://learn.javascript.ru/array-methods" \l "transformirovat-v-obekty)

важность: 5

У вас есть массив объектов user, и у каждого из объектов есть name, surname и id.

Напишите код, который создаст ещё один массив объектов с параметрами id и fullName, где fullName – состоит из name и surname.

Например:

let vasya = { name: "Вася", surname: "Пупкин", id: 1 };

let petya = { name: "Петя", surname: "Иванов", id: 2 };

let masha = { name: "Маша", surname: "Петрова", id: 3 };

let users = [ vasya, petya, masha ];

let usersMapped = /\* ... ваш код ... \*/

/\*

usersMapped = [

{ fullName: "Вася Пупкин", id: 1 },

{ fullName: "Петя Иванов", id: 2 },

{ fullName: "Маша Петрова", id: 3 }

]

\*/

alert( usersMapped[0].id ) // 1

alert( usersMapped[0].fullName ) // Вася Пупкин

Итак, на самом деле вам нужно трансформировать один массив объектов в другой. Попробуйте использовать =>. Это небольшая уловка.

решение

### [Отсортировать пользователей по возрасту](https://learn.javascript.ru/array-methods" \l "otsortirovat-polzovateley-po-vozrastu)

важность: 5

Напишите функцию sortByAge(users), которая принимает массив объектов со свойством age и сортирует их по нему.

Например:

let vasya = { name: "Вася", age: 25 };

let petya = { name: "Петя", age: 30 };

let masha = { name: "Маша", age: 28 };

let arr = [ vasya, petya, masha ];

sortByAge(arr);

// теперь: [vasya, masha, petya]

alert(arr[0].name); // Вася

alert(arr[1].name); // Маша

alert(arr[2].name); // Петя

решение

### [Перемешайте массив](https://learn.javascript.ru/array-methods" \l "peremeshayte-massiv)

важность: 3

Напишите функцию shuffle(array), которая перемешивает (переупорядочивает случайным образом) элементы массива.

Многократные прогоны через shuffle могут привести к разным последовательностям элементов. Например:

let arr = [1, 2, 3];

shuffle(arr);

// arr = [3, 2, 1]

shuffle(arr);

// arr = [2, 1, 3]

shuffle(arr);

// arr = [3, 1, 2]

// ...

Все последовательности элементов должны иметь одинаковую вероятность. Например, [1,2,3] может быть переупорядочено как [1,2,3] или [1,3,2], или [3,1,2] и т.д., с равной вероятностью каждого случая.

решение

### [Получить средний возраст](https://learn.javascript.ru/array-methods" \l "poluchit-sredniy-vozrast)

важность: 4

Напишите функцию getAverageAge(users), которая принимает массив объектов со свойством age и возвращает средний возраст.

Формула вычисления среднего арифметического значения: (age1 + age2 + ... + ageN) / N.

Например:

let vasya = { name: "Вася", age: 25 };

let petya = { name: "Петя", age: 30 };

let masha = { name: "Маша", age: 29 };

let arr = [ vasya, petya, masha ];

alert( getAverageAge(arr) ); // (25 + 30 + 29) / 3 = 28

решение

### [Оставить уникальные элементы массива](https://learn.javascript.ru/array-methods" \l "ostavit-unikalnye-elementy-massiva)

важность: 4

Пусть arr – массив строк.

Напишите функцию unique(arr), которая возвращает массив, содержащий только уникальные элементы arr.

Например:

function unique(arr) {

/\* ваш код \*/

}

let strings = ["кришна", "кришна", "харе", "харе",

"харе", "харе", "кришна", "кришна", ":-O"

];

alert( unique(strings) ); // кришна, харе, :-O

[Открыть песочницу с тестами для задачи.](https://plnkr.co/edit/JJc1HmaQ1tiqIUL6?p=preview)

решение

# Операторы сравнения

Многие операторы сравнения известны нам из математики.

В JavaScript они записываются так:

* Больше/меньше: a > b, a < b.
* Больше/меньше или равно: a >= b, a <= b.
* Равно: a == b. Обратите внимание, для сравнения используется двойной знак равенства ==. Один знак равенства a = b означал бы присваивание.
* Не равно. В математике обозначается символом ≠, но в JavaScript записывается как a != b.

В этом разделе мы больше узнаем про то, какие бывают сравнения, как язык с ними работает и к каким неожиданностям мы должны быть готовы.

В конце вы найдёте хороший рецепт того, как избегать «причуд» сравнения в JavaScript.

## [Результат сравнения имеет логический тип](https://learn.javascript.ru/comparison" \l "rezultat-sravneniya-imeet-logicheskiy-tip)

Все операторы сравнения возвращают значение логического типа:

* true – означает «да», «верно», «истина».
* false – означает «нет», «неверно», «ложь».

Например:

alert( 2 > 1 ); // true (верно)

alert( 2 == 1 ); // false (неверно)

alert( 2 != 1 ); // true (верно)

Результат сравнения можно присвоить переменной, как и любое значение:

let result = 5 > 4; // результат сравнения присваивается переменной result

alert( result ); // true

## [Сравнение строк](https://learn.javascript.ru/comparison" \l "sravnenie-strok)

Чтобы определить, что одна строка больше другой, JavaScript использует «алфавитный» или «лексикографический» порядок.

Другими словами, строки сравниваются посимвольно.

Например:

alert( 'Я' > 'А' ); // true

alert( 'Коты' > 'Кода' ); // true

alert( 'Сонный' > 'Сон' ); // true

Алгоритм сравнения двух строк довольно прост:

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

В примерах выше сравнение 'Я' > 'А' завершится на первом шаге, тогда как строки 'Коты' и 'Кода' будут сравниваться посимвольно:

1. К равна К.
2. о равна о.
3. т больше, чем д. На этом сравнение заканчивается. Первая строка больше.

**Используется кодировка Unicode, а не настоящий алфавит**

Приведённый выше алгоритм сравнения похож на алгоритм, используемый в словарях и телефонных книгах, но между ними есть и различия.

Например, в JavaScript имеет значение регистр символов. Заглавная буква "A" не равна строчной "a". Какая же из них больше? Строчная "a". Почему? Потому что строчные буквы имеют больший код во внутренней таблице кодирования, которую использует JavaScript (Unicode). Мы ещё поговорим о внутреннем представлении строк и его влиянии в главе [Строки](https://learn.javascript.ru/string).

## [Сравнение разных типов](https://learn.javascript.ru/comparison" \l "sravnenie-raznyh-tipov)

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

Например:

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

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

Логическое значение true становится 1, а false – 0.

Например:

alert( true == 1 ); // true

alert( false == 0 ); // true

**Забавное следствие**

Возможна следующая ситуация:

* Два значения равны.
* Одно из них true как логическое значение, другое – false.

Например:

let a = 0;

alert( Boolean(a) ); // false

let b = "0";

alert( Boolean(b) ); // true

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

С точки зрения JavaScript, результат ожидаем. Равенство преобразует значения, используя числовое преобразование, поэтому "0" становится 0. В то время как явное преобразование с помощью Boolean использует другой набор правил.

## [Строгое сравнение](https://learn.javascript.ru/comparison" \l "strogoe-sravnenie)

Использование обычного сравнения == может вызывать проблемы. Например, оно не отличает 0 от false:

alert( 0 == false ); // true

Та же проблема с пустой строкой:

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

Это происходит из-за того, что операнды разных типов преобразуются оператором == к числу. В итоге, и пустая строка, и false становятся нулём.

Как же тогда отличать 0 от false?

**Оператор строгого равенства === проверяет равенство без приведения типов.**

Другими словами, если a и b имеют разные типы, то проверка a === b немедленно возвращает false без попытки их преобразования.

Давайте проверим:

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

Ещё есть оператор строгого неравенства !==, аналогичный !=.

Оператор строгого равенства дольше писать, но он делает код более очевидным и оставляет меньше места для ошибок.

## [Сравнение с null и undefined](https://learn.javascript.ru/comparison" \l "sravnenie-s-null-i-undefined)

Поведение null и undefined при сравнении с другими значениями — особое:

**При строгом равенстве ===**

Эти значения различны, так как различны их типы.

alert( null === undefined ); // false

**При нестрогом равенстве ==**

Эти значения равны друг другу и не равны никаким другим значениям. Это специальное правило языка.

alert( null == undefined ); // true

**При использовании математических операторов и других операторов сравнения < > <= >=**

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

Посмотрим, какие забавные вещи случаются, когда мы применяем эти правила. И, что более важно, как избежать ошибок при их использовании.

### [Странный результат сравнения null и 0](https://learn.javascript.ru/comparison" \l "strannyy-rezultat-sravneniya-null-i-0)

Сравним null с нулём:

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

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

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

С точки зрения математики это странно. Результат последнего сравнения говорит о том, что "null больше или равно нулю", тогда результат одного из сравнений выше должен быть true, но они оба ложны.

Причина в том, что нестрогое равенство и сравнения > < >= <= работают по-разному. Сравнения преобразуют null в число, рассматривая его как 0. Поэтому выражение (3) null >= 0 истинно, а null > 0 ложно.

С другой стороны, для нестрогого равенства == значений undefined и null действует особое правило: эти значения ни к чему не приводятся, они равны друг другу и не равны ничему другому. Поэтому (2) null == 0 ложно.

### [Несравненное значение undefined](https://learn.javascript.ru/comparison" \l "nesravnennoe-znachenie-undefined)

Значение undefined несравнимо с другими значениями:

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

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

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

Почему же сравнение undefined с нулём всегда ложно?

На это есть следующие причины:

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

### [Как избежать проблем](https://learn.javascript.ru/comparison" \l "kak-izbezhat-problem)

Зачем мы рассмотрели все эти примеры? Должны ли мы постоянно помнить обо всех этих особенностях? Не обязательно. Со временем все они станут вам знакомы, но можно избежать проблем, если следовать надёжным правилам:

* Относитесь очень осторожно к любому сравнению с undefined/null, кроме случаев строгого равенства ===.
* Не используйте сравнения >= > < <= с переменными, которые могут принимать значения null/undefined, разве что вы полностью уверены в том, что делаете. Если переменная может принимать эти значения, то добавьте для них отдельные проверки.

## [Итого](https://learn.javascript.ru/comparison#itogo)

* Операторы сравнения возвращают значения логического типа.
* Строки сравниваются посимвольно в лексикографическом порядке.
* Значения разных типов при сравнении приводятся к числу. Исключением является сравнение с помощью операторов строгого равенства/неравенства.
* Значения null и undefined равны == друг другу и не равны любому другому значению.
* Будьте осторожны при использовании операторов сравнений вроде > и < с переменными, которые могут принимать значения null/undefined. Хорошей идеей будет сделать отдельную проверку на null/undefined.