В современном JavaScript существует два типа чисел:

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

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 (253-1) или менее чем -(253-1).

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

1. Можно использовать символ нижнего подчёркивания \_ в качестве разделителя для больших чисел:

let billion = 1\_000\_000\_000;

Символ нижнего подчёркивания \_ – это «[синтаксический сахар](https://ru.wikipedia.org/wiki/%D0%A1%D0%B8%D0%BD%D1%82%D0%B0%D0%BA%D1%81%D0%B8%D1%87%D0%B5%D1%81%D0%BA%D0%B8%D0%B9_%D1%81%D0%B0%D1%85%D0%B0%D1%80)», он делает число более читабельным. Движок JavaScript игнорирует \_ между цифрами, поэтому в примере выше получается просто миллиард.

2. Укороченая запись с использованием e.

Чтобы укоротить запись числа, мы можем добавить к нему букву "e" и указать необходимое количество нулей:

let billion = 1e9; // 1 миллиард, буквально: 1 и 9 нулей

Другими словами, "e" умножает число на 1 с указанным количеством нулей.

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

1e3 === 1 \* 1000 // e3 означает \*1000

1.23e6 === 1.23 \* 1000000 // e6 означает \*1000000

Можно записать что-нибудь очень маленькое. К примеру, 1 микросекунду (одна миллионная секунды):

let mcs = 0.000001;

или:

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

Если мы подсчитаем количество нулей в 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)

3. Шестнадцатеричные, двоичные и восьмеричные числа

Шестнадцатеричные - 0x, после которого указывается число.

Например:

alert( 0xff ); // 255

alert( 0xFF ); // 255 (то же самое, регистр не имеет значения)

Восьмеричные- 0o, после которого указывается число.

Например:

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

Двоичные - 0b, после которого указывается число.

Например:

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

alert( a == b ); // true, с двух сторон число 255

Есть только 3 системы счисления с такой поддержкой.

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

num.toString(base)

возвращает строковое представление числа num в системе счисления base.

let num = 255;

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

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

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

**Две точки для вызова метода**

Например

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

Если нужно 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**

**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, 3.5 — 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.

**округлить число до n-ого количества цифр в дробной части**

Есть два пути решения:

1. Умножить и разделить.

Например, чтобы округлить число до первого знака после запятой, мы можем умножить число на 10, вызвать функцию округления, чтобы срезать десятичную часть и разделить обратно.

let num = 1.21456;

* console.log(Math.trunc(num\*10) / 10) 12.145600000000002 -> 12 -> 1.2

**Как умножать десятичные дроби на 10, 100, 1000**

Чтобы умножить десятичную дробь на 10, 100, 1000, нужно просто перенести запятую в дроби вправо на столько знаков, сколько нулей стоит во втором множителе. Лишние нули слева можно отбросить. А если цифр не хватает — дописываем нули.

Примеры:

1,15 ∗ 10 = 11,5;

22,345 ∗ 100 = 2 234,5;

8,99 ∗ 1 000 = 8 990;

0,54678 ∗ 10 000 = 5467,8;

0,07 ∗ 1 000 = 70;

0,00033 ∗ 100 = 0,033.

2. 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. Округляет значение до ближайшего числа.

let num = 12.34;

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

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"

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

Для хранения числа используется 64 бита: 52 из них используется для хранения цифр, 11 для хранения положения десятичной точки и один бит отведён на хранение знака.

Если число слишком большое, оно переполнит 64-битное хранилище, JavaScript вернёт бесконечность:

alert( 1e500 ); // Infinity

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.

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 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 всё 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. Просто используйте округление, чтобы отрезать «хвосты», когда надо.

**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.

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

Nan и Infinity принадлежат типу number, но они не являются «обычными» числами, поэтому есть функции для их проверки:

* isNaN(value) преобразует значение в число и проверяет является ли оно NaN:
* alert( isNaN(NaN) ); // true
* alert( isNaN("str") ); // true

Нельзя просто сравнить NaN === NaN Т.к NaN не является равным ничему 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

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

**Методы - Number.isNaN и Number.isFinite**

Методы [Number.isNaN](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/Number/isNaN) и [Number.isFinite](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/Number/isNaN) – это более «строгие» версии функций isNaN и isFinite. Они не преобразуют аргумент в число, а наоборот – первым делом проверяют, является ли аргумент числом (принадлежит ли он к типу number).

**Number.isNaN(value)** возвращает true только в том случае, если аргумент принадлежит к типу number и является NaN. Во всех остальных случаях возвращает false.

alert( Number.isNaN(NaN) ); // true

alert( Number.isNaN("str" / 2) ); // true

// Обратите внимание на разный результат:

alert( Number.isNaN("str") ); // false, так как "str" является строкой, а не числом

alert( isNaN("str") ); // true, так как isNaN сначала преобразует строку "str" в число и в результате преобразования получает NaN

**Number.isFinite(value)** возвращает true только в том случае, если аргумент принадлежит к типу number и не является NaN/Infinity/-Infinity. Во всех остальных случаях возвращает false.

alert( Number.isFinite(123) ); // true

alert( Number.isFinite(Infinity) ); // false

alert( Number.isFinite(2 / 0) ); // false

/ Обратите внимание на разный результат:

alert( Number.isFinite("123") ); // false, так как "123" является строкой, а не числом

alert( isFinite("123") ); // true, так как isFinite сначала преобразует строку "123" в число 123

Не стоит считать Number.isNaN и Number.isFinite более «корректными» версиями функций isNaN и isFinite. Это дополняющие друг-друга инструменты для разных задач.

**Сравнение Object.is**

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/" \l "sec-samevalue)).

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

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€".

+ или Number() не вытащат число из этих строк - дадут NAN.

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

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

Второй параметр принимает значения от 2х до 36 включительно.

[Math](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/Math)

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

**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