Rounding to one decimal place javascript

Math.round() returns the rounded number value closest to an integer. Returns the value of a rounded number to the nearest number. Syntax Math.round(x) Parameters A number. Return value The value of the default number rounded at the nearest integer. Description If the fractional part of the number is 0.5 or greater, the argument is rounded to the next number above. If part of a number is less than 0.5, the argument is rounded to the next number below. Because a circle() is a static mathematical method, it should always be used as Math.round(), not as a method of the mathematical object you created. (Mathematics is not a builder) Examples using Math.round() // Returns a value of 20 x with Math.round (8.49pm); Returns 21 x with Math.round (20.5); Returns the value -20 x with Math.round(-20.5); Returns the value -21 x with Math.round(-20.51); Returns the value 1 (!) // Notice the rounding error due to the inaccuracies of the arithmetic point. Compare this with Math.round10(1,005, -2) from the example below. x - Math.round(1.005*100)/100; Decimal rounding // Conclusion (function)? /** * Decimal adjustment of the number. * * @param . String @param @returns @param . . . . if (typeof exp ? undefined ? + exp ? 0) ? return math [type](value); ? exp +exp; If the value is not a number or the ex is not an integer... if (isNaN(value) ? !(typeof exp ? number && exp % 1 with 0)) ? reimbursement of NaN; ? Shift value? value.toString().split('e'); value - Math[type](+(value[0] + e + (value[1]? (+value[1] - exp) : -exp)); Changing the value back? value.toString().split(s); return +(value[0] + e + (value[1]? (+value[1] + exp) : exp)); • // Decimal circle if (! Math.round10? Math.round10? function (value, ex)? return decimalAdjust('round', value, exp); ? ? ? / Decimal floor if (! Math.floor10? Math.floor10? function (value, exp) ? return decimalAdjust('floor', value, exp); ? ? ? / Decimal ceil if (! Math.ceil10) ? (); Round Math.round10(55.55, -1); 55.6 Mathematics.round10 (55,549, 55.5 Mathematics.okrugli10 (55, 1); 60 Mathematics.okrugli10(54,9, 1); 50 Mathematics.okrugli10(-55.55, -1); -55.5 Mathematics.okrugli10(-55551, -1); -55.6 Mathematics.okrugli10(-55, 1); -50 Mathematics.okrugli10(-55.1, 1); -60 Mathematics.okrugli10 (1,005, -2); 1.01 -- usporedite to s Math.round (1.005*100)/100 iznad // Kat Mathematics.floor10(55.59, -1); 55.5 Mathematics.floor10(59, 1); 50 Mathematics.floor10(-55.51, -1); -55.6 Mathematics.floor10(-51, 1); -60 // Ceil Math.ceil10(55.51, -1); 55.6 55.6 1); 60 Mathematics.ceil10(-55.59, -1); -55.5 Math.ceil10(-59, 1); -50 Especificaciones Característica Chrome Firefox (Gecko) Internet Explorer Opera Safari (WebKit) Soporte Básico (Yes) (Yes) (Yes) Característica Android Firefox Mobile (Gecko) IE Phone Opera Mobile Mobile Mobile Soporte Básico (Yes) (Yes) (Yes) (Yes) (Yes) (Yes) Yes) (Yes) Véase también Math.abs() Math.ceil() Math.floor() Math.sign() Math.trunc() There are two types of numbers in modern JavaScript: Regular numbers in JavaScript are stored in the 64-bit format of IEEE-754, also known as double precise floating point numbers. Those are the numbers we use most of the time, and we'll talk about them in this chapter. BigInt numbers, representing numbers of arbitrary length. Sometimes they are necessary, because the regular number can not exceed 253 or be less than -253. Since biginti are used in several special areas, we dedicate a special chapter of BigInt to them. So here we will talk about regular numbers. Let's expand our knowledge of them. There are more ways to write the number ofImagine we have to write a billion. The obvious way is: let the billion = 1000000000; But in real life we usually avoid writing a long string of zeros because it's easy to misspelle. Also, we're lazy. Normally we'll write something like $1 billion for a billion or 7.3 billion for 7 billion 300 million. The same goes for most large numbers. In JavaScript, we shorten the number by adding the letter e to the number and specifying the number zero: let the billion = 1e9; 1 billion, literally: 1 and 9 zero alerts (7,3e9); 7.3 billion (7,300,000,000) In other words, it multiplies the number by 1 with the given number zero. 1e3 = 1 * 1000 1.23e6 = 1.23 * 10000000 Now let's write something very small. Say, 1 microseconds (one millionth of a second): Just like before, using e can help. If we want to explicitly avoid writing zeros, we could say the same as: let ms = 1e-6; Six zeros left of 1 If we count zeros in 0.0000001, there are 6. So, of course it's 1e-6. In other words, a negative number after e means a division by 1 by a given number of zeros: // -3 is divided by 1 by 3 zeros 1e-3 = 1 / 1000 (= 0.001) // -6 is divided by 1 from 6 zeros 1.23e-6 = 1.23 / 100000000 (=0.00000123) Hex, binary and octon numbersHexadecimal numbers are widely used in JavaScript to represent colors, code characters for many other things as well. So, of course, there is a shorter way to write them: 0x, and then a number. For example: warning (0xff); 255 alerts (0xFF); 255 (also, the case does not matter) Binary and octacal numerical systems are rarely used, but also supported by prefixes of 0b and 0o: let a = 0b1111111; binary form of 255 flight b = 0o377; octave warning form of 255(a == b); True, the same number 255 on both sides There are only 3 numerical systems with such support. For other numbers we should use the parseInt function (which we will see later in this chapter). toString(base)The num.toString(base) method returns a series of representations of numbers in numeric systems with a specific base. For example: some number = 255; warning(num.toString(16)); ff Warning(num.toString(2)); 1111111111 Base can vary from 2 to 36. By default, 10. Common use cases for this are: base = 16 is used for hex colors, character encoding, etc., digits can be 0.9 or A.. F. base=2 is mainly for correcting documents in a substantial direction, digits can be 0 or 1. base=36 is the maximum, digits can be 0.9 or A.. Z. The entire Latin is used to represent the number. A funny but useful case for 36 is when we need to convert a long numerical identifier to something shorter, for example to make a short URL. It can be easily represented in base 36 numerical systems: warning (123456.toString(36)); 2n9c Keep hinting that two dots in 123456.toString(36) is not a typo. If we want to call the method directly to the number, such asstring it in the example above, then we need to set two dots .. after him. If we were to set one point: 123456.toString (36), then an error would occur, because JavaScript syntax implies the decimal part after the first point. And if we put another point, then JavaScript knows that the decimal part is empty and now goes the method. It can also write (123456).toString (36). RoundingThis one of the most commonly used operations when working with numbers is rounding. There are several built-in rounding functions: Math.floor Rounds Off: 3.1 becomes 3 and -1.1 becomes -2. Math.ceil Rounds Off: 3.1 becomes 4 and -1.1 becomes -1. Math.round Rounds to nearest number: 3.1 becomes 3, 3.6 becomes 4 and -1.1 becomes -1. Math.trunc (not supported by Internet Explorer) Removes anything after the decimal point without rounding: 3.1 becomes 3, -1.1 becomes -1. Here is the table to commence the difference between them: Math.floor Math.ceil Math.round Math.round Math.trunc 3.1 3 4 3 3.6 3 4 4 3 -1.1 -2 -1 -1 -1.6 -2 -1 -2 -1 These functions cover all possible ways to resolve the decimal part of the number. But what if we want to round the number to n-th digit after decimal? For example, we have 1.2345 and we want to round it to 2 digits, taking only 1.23. There are two ways to do this: Multiply and divide. For example, to round a number to another digit after decimal, we can multiply the number by 100 (or more than 10), call the rounding function, and then split it back. flight num = 1.23456; warning(Math.floor(num * 100) / 100); 1.23456 -> 123.456 -> 123 -> 1.23 The toFixed(n) method rounds the number to n digits after the point and returns the string to display the results. some number = 12.34; warning(num.toFixed(1)); 12.3 This rounds or drops to the nearest value, similar to Math.round: some number = 12.36; num.toFixed(1)); 12.4 Please Please that result toFixed is a series. If the decimal part is shorter than required, zeros are fully adapted: number = 12.34; warning(num.toFixed(5)); 12.34000, zeros added to make exactly 5 digits We can convert it to a number using an inconsiderate plus or number() of the call: +num.toFixed(5). Imprecise calculationsUse fully, the number is represented in the 64-bit format IEEE-754, so there are exactly 64 bits to store the number: 52 of them are used to store digits, 11 of them store the position of the decimal point (they are zero for numbers integer), and 1 bit is for the character. If the number is too large, it would pour over 64-bit storage space, potentially giving infinity: warning (1e500); Infinity What may be a little less obvious, but happens very often, is a loss of precision. Consider this (falsy!) test: warning (0.1 + 0.2 == 0.3); That's false, if we check that the sum is 0.1 and 0.2 0.3, we get fake. Strange! Then what if not 0.3? warning(0.1 + 0.2); 0.30000000000000000004 Joj! There are more consequences here than inaccurate comparisons. Imagine you're doing an e-shopping website and a visitor puts $0.10 and $0.20 merchandise in their shopping cart. The total order will be $0.30000000000000000000000. That would surprise anyone. But why is this happening? The number is stored in memory in binary form, sequence of bits – one and zero. But fractions like 0.1, 0.2 that look simple in the decimal numerical system are actually endless fractions in their binary form. In other words, what is 0.1? That's one divided by ten 1/10, one tenth. In a decimal numeric system, such numbers are easily interchangeable. Compare it to one third: 1/3. It becomes an infinite fraction of 0.33333(3). Thus, the division by powers 10 is guaranteed to work well in the decimaal system, but the division by 3 is not. For the same reason, in the binary numerical system, division by powers 2 is guaranteed to work, but 1/10 becomes an endless binary fraction. There is simply no way to store exactly 0.1 or exactly 0.2 using a binary system, just as there is no way to store one third as a decimal fraction. The IEEE-754 numeric format resolves this by rounding it to the nearest number possible. These rounding rules don't usually allow us to see that little loss of precision, but there is. We can see this in action: warning(0.1.toFixed(20)); 0.10000000000000000055 And when we add up the two numbers, their precise losses add up. That's why 0.1 + 0.2 is not exactly 0.3. The same problem exists in many other programming languages. PHP, Java, C, Perl, Ruby give exactly the same result, because they are based on the same numerical format. Can we get around the problem? Of course, the most reliable method is to round off the result with the help of the toFixed(n) method: let the sum = 0.1 + 0.2; warning(sum.toFixed(2)); 0.30 Keep hinting that toFixed always returns the string. It ensures that has 2 digits after the decimal point. It's actually handy if we have an e-shopping and need to show $0.30. For other cases, we can use an unconfirmed plus to force it into a number: let the sum = 0.1 + 0.2; warning(+sum.toFixed(2)); 0.3 We can also temporarily multiply the numbers by 100 (or greater) to convert them into numbers, calculate them, and then divide them back together. Then, as we do the math with an integer, the error decreases somewhat, but we still get it on the division: warning (0.1 * 10 + 0.2 * 10) / 10); 0.3 warning((0.28 * 100 + 0.14 * 100) / 100); 0.42000000000000001 Therefore, the multiply/divide approach reduces the error, but does not completely remove it. Sometimes we might even try to avoid fractions. It's like we're dealing with trade, then we can store prices in cents instead of dollars. But what if we apply a 30% discount? In practice, complete avoidance of fractions is rarely possible. Just round them cut their tails when needed. Try running this: // Hello! I'm a self-employed number! warning(9999999999999999999999999999999); Shows 100000000000000000 This suffers from the same problem: loss of precision. There are 64 bits for a number, 52 of them can be used to store digits, but this is not enough. So the least significant digits disappear. JavaScript does not start an error in such events. It does its best to fit the number into the desired format, but unfortunately, this format is not large enough. Another ridiculous consequence of the internal representation of numbers is the existence of two zeros: 0 and -0. This is because the character is represented by a single bit, so it can be set or not set for any number, including zero. In most cases, the difference is imperceptible because operators are suitable for treating them. Tests: isFinite and isNaNRemember these two special numerical values? Infinity (i -Infinity) is a special numerical value that is greater (less) than anything else. NaN represents an error. They belong to a typical number, but they are not normal numbers, so there are special functions to check for them: isNaN (value) converts its argument to a number, and then tests it on nan: warning (isNaN (NaN)); real alert (isNaN(p)); True, but do we need this function? Can't we use the comparison === NaN? I'm sorry, but the answer is no. The NaN value is unique in that it does not equal anything, including itself: warning (NaN === NaN); false isFinite(value) converts its argument to a number and returns true if it is a regular number, not a NaN/Infinity/Infinity: alert(isFinite(15)); true warning (isFinite(p)); false, because special value: NaN alert (isFinite(Infinity)); false, because special value: Infinity Sometimes isFinite is used to check that the string value is an ordinary number: let the number = +prompt(Enter a number,); will be true unless you enter Infinity, warning(isFinite(num)); For example, an empty or spatial string only is treated as 0 in all numeric functions, including isFinite. There is a special built-in method Object.is that compares values such as ===, but is more reliable for two marginal cases: Works with NaN: Object.is(NaN, NaN) === true, that's a good thing. The values 0 and -0 are different: Object.is(0, -0) === false, technically this is true, because internally the number has a bit sign that may be different even if all other bits are zeros. In all other cases, the object.is(a, b) is the same as === b. This method of comparison is often used in javascript specification. When an internal algorithm needs to compare two values because it is exactly the same, it uses Object.is (internally called SameValue). parseInt and parseFloatNumeric conversion using plus + or number() is strict. If the value is not exactly a number, it fails: warning (+100px); NaN The only exceptions are spaces at the beginning or end of a series, as they are ignored. But in real life we often have values in units, such as 100px or 12pt in CSS. Also in many countries the currency symbol goes after the amount, so we have 19 € and we would like to draw a numerical value from this. For this are parseInt and parseFloat. They read the number from the wire until they can't. In the event of an error, the number collected is returned. ParseInt returns the integer, while parseFloat returns the floating point number: warning (parseInt('100px')); 100 alerts(parseFloat(12.5em)); 12.5 warning(parseInt('12.3')); 12, only the number part returns to alert(parseFloat(12.3.4)); 12.3, point two stops reading There are situations when parseInt/parseFloat will restore NaN. This occurs when digits cannot be read: warning(parse(a123); NaN, the first symbol stops the parseInt() function process has an optional second parameter. It specifies the base of the numeric system, so that the parsed strings of hex numbers, binary numbers and so on: warning(parsing('0xff', 16)); 255 warnings(parseInt('ff', 16)); 255, without 0x also works on alert(parseInt('2n9c', 36); 123456 Other mathematical functionsJavaScript has a built-in mathematical object that contains a small library of mathematical functions and constants. Several examples: Math.random() Returns a random number from 0 to 1 (not including 1) warning (Math.random()); 0.1234567894322 warning (Math.random()); 0.5435252343232 alert (Math.random()); // ... (any random numbers) Mathematics.max(a, b, c...) / Math.min(a, b, c...) Returns the largest/smallest of an arbitrary number of arguments. warning(mathematics.max(3, 5, -10, 0, 1)); 5 alerts(Math.min(1, 2)); 1 Math.pow(n, power) Returns n raised the given power warning (Math.pow(2, 10)); 2 in power 10 = 1024 There are multiple functions and constants in the mathematical object, trigonometry, which you can find in documents for a mathematical object. Summary Write numbers from many zeros: The e add-in with zeros is counted to a number. Like: 123e6 is the same as 123 with 6 zeros 1230000000. A negative number after e causes the number to be divided by 1 by zeros given. E.g. 123e-6 means 0.000123 (123 millionths). For different numerical systems: You can write numbers directly in hex (0x), octal (0o) and binary (0b) systems. parses a string into an integer in a numeral system with a specific base, 2 ≤ bases ≤ 36. num.toString(base) converts a number to a string in a numeral system with a specific base. To convert values such as 12pt and 100px to a number: Use parseInt/parseFloat for soft conversion, which reads a number from a string and then returns a value they could read before the error. For fractions: Round using Math.floor, Math.ceil, Math.trunc, Math.round or num.toFixed(precision). Be sure to remember that there has been a loss of precision when working with fractions. More mathematical functions: See a mathematical object when you need them. The library is very small, but it can cover basic needs. Need.