9th December 2019

Numbers

In modern JavaScript, there are two types of numbers:

1. Regular numbers in JavaScript are stored in 64-bit format IEEE-754, also known as “double precision floating point numbers”. These are numbers that we’re using most of the time, and we’ll talk about them in this chapter.
2. BigInt numbers, to represent integers of arbitrary length. They are sometimes needed, because a regular number can’t exceed 2^{53} or be less than -2^{53}. As bigints are used in few special areas, we devote them a special chapter BigInt.

So here we’ll talk about regular numbers. Let’s expand our knowledge of them.

More ways to write a number

Imagine we need to write 1 billion. The obvious way is:

```
let billion = 1000000000;
```

But in real life, we usually avoid writing a long string of zeroes as it’s easy to mistype. Also, we are lazy. We will usually write something like `"1bn"` for a billion or `"7.3bn"` for 7 billion 300 million. The same is true for most large numbers.

In JavaScript, we shorten a number by appending the letter `"e"` to the number and specifying the zeroes count:

```
let billion = 1e9;  // 1 billion, literally: 1 and 9 zeroes

alert( 7.3e9 );  // 7.3 billions (7,300,000,000)
```

In other words, `"e"` multiplies the number by `1` with the given zeroes count.

```
1e3 = 1 * 1000
1.23e6 = 1.23 * 1000000
```

Now let’s write something very small. Say, 1 microsecond (one millionth of a second):

```
let ms = 0.000001;
```

Just like before, using `"e"` can help. If we'd like to avoid writing the zeroes explicitly, we could say the same as:

```
let ms = 1e-6; // six zeroes to the left from 1
```

If we count the zeroes in `0.000001`, there are 6 of them. So naturally it's `1e-6`.

In other words, a negative number after `"e"` means a division by 1 with the given number of zeroes:

```
// -3 divides by 1 with 3 zeroes
1e-3 = 1 / 1000 (=0.001)

// -6 divides by 1 with 6 zeroes
1.23e-6 = 1.23 / 1000000 (=0.00000123)
```

Hex, binary and octal numbers

Hexadecimal numbers are widely used in JavaScript to represent colors, encode characters, and for many other things. So naturally, there exists a shorter way to write them: `0x` and then the number.

For instance:

```
alert( 0xff ); // 255
alert( 0xFF ); // 255 (the same, case doesn't matter)
```

Binary and octal numeral systems are rarely used, but also supported using the `0b` and `0o` prefixes:

```
let a = 0b11111111; // binary form of 255
let b = 0o377; // octal form of 255

alert( a == b ); // true, the same number 255 at both sides
```

There are only 3 numeral systems with such support. For other numeral systems, we should use the function `parseInt` (which we will see later in this chapter).

toString(base)

The method `num.toString(base)` returns a string representation of `num` in the numeral system with the given `base`.

For example:

```
let num = 255;

alert( num.toString(16) );  // ff
alert( num.toString(2) );   // 11111111
```

The `base` can vary from `2` to `36`. By default it's `10`.

Common use cases for this are:

- **base=16** is used for hex colors, character encodings etc, digits can be `0..9` or `A..F`.
- **base=2** is mostly for debugging bitwise operations, digits can be `0` or `1`.
- **base=36** is the maximum, digits can be `0..9` or `A..Z`. The whole latin alphabet is used to represent a number. A funny, but useful case for `36` is when we need to turn a long numeric identifier into something shorter, for example to make a short url. Can simply represent it in the numeral system with base `36`:

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

Two dots to call a method

Please note that two dots in `123456..toString(36)` is not a typo. If we want to call a method directly on a number, like `toString` in the example above, then we need to place two dots `..` after it.

If we placed a single dot: `123456.toString(36)`, then there would be an error, because JavaScript syntax implies the decimal part after the first dot. And if we place one more dot, then JavaScript knows that the decimal part is empty and now goes the method.

Also could write `(123456).toString(36)`.

Rounding

One of the most used operations when working with numbers is rounding.

There are several built-in functions for rounding:

`Math.floor`

Rounds down: `3.1` becomes `3`, and `-1.1` becomes `-2`.

`Math.ceil`

Rounds up: `3.1` becomes `4`, and `-1.1` becomes `-1`.

`Math.round`

Rounds to the nearest integer: `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's the table to summarize the differences between them:

	`Math.floor`	`Math.ceil`	`Math.round`	`Math.trunc`
`3.1`	3	4	3	3

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

These functions cover all of the possible ways to deal with the decimal part of a number. But what if we'd like to round the number to `n-th` digit after the decimal?

For instance, we have `1.2345` and want to round it to 2 digits, getting only `1.23`.

There are two ways to do so:

1. Multiply-and-divide.

   For example, to round the number to the 2nd digit after the decimal, we can multiply the number by `100`, call the rounding function and then divide it back.

   ```
   1  let num = 1.23456;
   2
   3  alert( Math.floor(num * 100) / 100 ); // 1.23456 -> 123.456 -> 123 -> 1.23
   ```

2. The method toFixed(n) rounds the number to `n` digits after the point and returns a string representation of the result.

   ```
   1  let num = 12.34;
   2  alert( num.toFixed(1) ); // "12.3"
   ```

   This rounds up or down to the nearest value, similar to `Math.round`:

   ```
   1  let num = 12.36;
   2  alert( num.toFixed(1) ); // "12.4"
   ```

   Please note that result of `toFixed` is a string. If the decimal part is shorter than required, zeroes are appended to the end:

   ```
   1  let num = 12.34;
   2  alert( num.toFixed(5) ); // "12.34000", added zeroes to make exactly 5 digi
   ```

   We can convert it to a number using the unary plus or a `Number()` call: `+num.toFixed(5)`.

Imprecise calculations

Internally, a number is represented in 64-bit format IEEE-754, so there are exactly 64 bits to store a number: 52 of them are used to store the digits, 11 of them store the position of the decimal point (they are zero for integer numbers), and 1 bit is for the sign.

If a number is too big, it would overflow the 64-bit storage, potentially giving an infinity:

```
1  alert( 1e500 ); // Infinity
```

What may be a little less obvious, but happens quite often, is the loss of precision.

Consider this (falsy!) test:

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

That's right, if we check whether the sum of `0.1` and `0.2` is `0.3`, we get `false`.

Strange! What is it then if not `0.3`?

```
1  alert( 0.1 + 0.2 ); // 0.30000000000000004
```

Ouch! There are more consequences than an incorrect comparison here. Imagine you're making an e-shopping site and the visitor puts `$0.10` and `$0.20` goods into their cart. The order total will be `$0.30000000000000004`. That would surprise anyone.

But why does this happen?

A number is stored in memory in its binary form, a sequence of bits – ones and zeroes. But fractions like `0.1`, `0.2` that look simple in the decimal numeric system are actually unending fractions in their binary form.

In other words, what is `0.1`? It is one divided by ten `1/10`, one-tenth. In decimal numeral system such numbers are easily representable. Compare it to one-third: `1/3`. It becomes an endless fraction `0.33333(3)`.

So, division by powers `10` is guaranteed to work well in the decimal system, but division by `3` is not. For the same reason, in the binary numeral system, the division by powers of `2` is guaranteed to work, but `1/10` becomes an endless binary fraction.

There's just no way to store *exactly 0.1* or *exactly 0.2* using the binary system, just like there is no way to store one-third as a decimal fraction.

The numeric format IEEE-754 solves this by rounding to the nearest possible number. These rounding rules normally don't allow us to see that "tiny precision loss", but it exists.

We can see this in action:

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

And when we sum two numbers, their "precision losses" add up.

That's why `0.1 + 0.2` is not exactly `0.3`.

Not only JavaScript

The same issue exists in many other programming languages.

PHP, Java, C, Perl, Ruby give exactly the same result, because they are based on the same numeric format.

Can we work around the problem? Sure, the most reliable method is to round the result with the help of a method toFixed(n):

```
1  let sum = 0.1 + 0.2;
2  alert( sum.toFixed(2) ); // 0.30
```

Please note that `toFixed` always returns a string. It ensures that it has 2 digits after the decimal point. That's actually convenient if we have an e-shopping and need to show `$0.30` . For other cases, we can use the unary plus to coerce it into a number:

```
1  let sum = 0.1 + 0.2;
2  alert( +sum.toFixed(2) ); // 0.3
```

We also can temporarily multiply the numbers by 100 (or a bigger number) to turn them into integers, do the maths, and then divide back. Then, as we're doing maths with integers, the error somewhat decreases, but we still get it on division:

```
1  alert( (0.1 * 10 + 0.2 * 10) / 10 ); // 0.3
2  alert( (0.28 * 100 + 0.14 * 100) / 100); // 0.4200000000000001
```

So, multiply/divide approach reduces the error, but doesn't remove it totally.

Sometimes we could try to evade fractions at all. Like if we're dealing with a shop, then we can store prices in cents instead of dollars. But what if we apply a discount of 30%? In practice, totally evading fractions is rarely possible. Just round them to cut "tails" when needed.

The funny thing

Try running this:

```
1  // Hello! I'm a self-increasing number!
2  alert( 9999999999999999 ); // shows 10000000000000000
```

This suffers from the same issue: a loss of precision. There are 64 bits for the number, 52 of them can be used to store digits, but that's not enough. So the least significant digits disappear.

JavaScript doesn't trigger an error in such events. It does its best to fit the number into the desired format, but unfortunately, this format is not big enough.

Two zeroes

Another funny consequence of the internal representation of numbers is the existence of two zeroes: `0` and `-0`.

That's because a sign is represented by a single bit, so it can be set or not set for any number including a zero.

In most cases the distinction is unnoticeable, because operators are suited to treat them as the same.

Tests: isFinite and isNaN

Remember these two special numeric values?

- `Infinity` (and `-Infinity`) is a special numeric value that is greater (less) than anything.
- `NaN` represents an error.

They belong to the type `number`, but 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 for being `NaN`:

```
1 alert( isNaN(NaN) ); // true
2 alert( isNaN("str") ); // true
```

But do we need this function? Can't we just use the comparison `=== NaN`? Sorry, but the answer is no. The value `NaN` is unique in that it does not equal anything, including itself:

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

- `isFinite(value)` converts its argument to a number and returns `true` if it's a regular number, not `NaN/Infinity/-Infinity`:

```
1 alert( isFinite("15") ); // true
2 alert( isFinite("str") ); // false, because a special value: NaN
3 alert( isFinite(Infinity) ); // false, because a special value: Infinity
```

Sometimes `isFinite` is used to validate whether a string value is a regular number:

```
1 let num = +prompt("Enter a number", '');
2
3 // will be true unless you enter Infinity, -Infinity or not a number
4 alert( isFinite(num) );
```

Please note that an empty or a space-only string is treated as `0` in all numeric functions including `isFinite`.

Compare with `Object.is`

There is a special built-in method Object.is that compares values like `===`, but is more reliable for two edge cases:

1. It works with `NaN`: `Object.is(NaN, NaN) === true`, that's a good thing.
2. Values `0` and `-0` are different: `Object.is(0, -0) === false`, technically that's true, because internally the number has a sign bit that may be different even if all other bits are zeroes.

In all other cases, `Object.is(a, b)` is the same as `a === b`.

This way of comparison is often used in JavaScript specification. When an internal algorithm needs to compare two values for being exactly the same, it uses `Object.is` (internally called SameValue).

parseInt and parseFloat

Numeric conversion using a plus `+` or `Number()` is strict. If a value is not exactly a number, it fails:

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

The sole exception is spaces at the beginning or at the end of the string, as they are ignored.

But in real life we often have values in units, like `"100px"` or `"12pt"` in CSS. Also in many countries the currency symbol goes after the amount, so we have `"19€"` and would like to extract a numeric value out of that.

That's what `parseInt` and `parseFloat` are for.

They "read" a number from a string until they can't. In case of an error, the gathered number is returned. The function `parseInt` returns an integer, whilst `parseFloat` will return a floating-point number:

```
1  alert( parseInt('100px') ); // 100
2  alert( parseFloat('12.5em') ); // 12.5
3
4  alert( parseInt('12.3') ); // 12, only the integer part is returned
5  alert( parseFloat('12.3.4') ); // 12.3, the second point stops the reading
```

There are situations when `parseInt/parseFloat` will return `NaN`. It happens when no digits could be read:

```
1  alert( parseInt('a123') ); // NaN, the first symbol stops the process
```

The second argument of `parseInt(str, radix)`

The `parseInt()` function has an optional second parameter. It specifies the base of the numeral system, so `parseInt` can also parse strings of hex numbers, binary numbers and so on:

```
alert( parseInt('0xff', 16) ); // 255
alert( parseInt('ff', 16) ); // 255, without 0x also works

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

Other math functions

JavaScript has a built-in Math object which contains a small library of mathematical functions and constants.

A few examples:

`Math.random()`

Returns a random number from 0 to 1 (not including 1)

```
alert( Math.random() ); // 0.1234567894322
alert( Math.random() ); // 0.5435252343232
alert( Math.random() ); // ... (any random numbers)
```

`Math.max(a, b, c...)` / `Math.min(a, b, c...)`

Returns the greatest/smallest from the arbitrary number of arguments.

```
alert( Math.max(3, 5, -10, 0, 1) ); // 5
alert( Math.min(1, 2) ); // 1
```

`Math.pow(n, power)`

Returns `n` raised the given power

```
alert( Math.pow(2, 10) ); // 2 in power 10 = 1024
```

There are more functions and constants in `Math` object, including trigonometry, which you can find in the docs for the Math object.

Summary

To write numbers with many zeroes:

- Append `"e"` with the zeroes count to the number. Like: `123e6` is the same as `123` with 6 zeroes `123000000`.
- A negative number after `"e"` causes the number to be divided by 1 with given zeroes. E.g. `123e-6` means `0.000123` (`123` millionths).

For different numeral systems:

- Can write numbers directly in hex (`0x`), octal (`0o`) and binary (`0b`) systems
- `parseInt(str, base)` parses the string `str` into an integer in numeral system with given `base`, $2 \le$ `base` ≤ 36.
- `num.toString(base)` converts a number to a string in the numeral system with the given `base`.

For converting values like `12pt` and `100px` to a number:

- Use `parseInt/parseFloat` for the "soft" conversion, which reads a number from a string and then returns the value they could read before the error.

For fractions:

- Round using `Math.floor`, `Math.ceil`, `Math.trunc`, `Math.round` or `num.toFixed(precision)`.
- Make sure to remember there's a loss of precision when working with fractions.

More mathematical functions:

- See the Math object when you need them. The library is very small, but can cover basic needs.

Tasks

Sum numbers from the visitor

importance: 5

Create a script that prompts the visitor to enter two numbers and then shows their sum.

Run the demo

P.S. There is a gotcha with types.

solution

Why 6.35.toFixed(1) == 6.3?

importance: 4

According to the documentation `Math.round` and `toFixed` both round to the nearest number: `0..4` lead down while `5..9` lead up.

For instance:

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

In the similar example below, why is `6.35` rounded to `6.3`, not `6.4`?

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

How to round `6.35` the right way?

solution

Repeat until the input is a number

importance: 5

Create a function `readNumber` which prompts for a number until the visitor enters a valid numeric value.

The resulting value must be returned as a number.

The visitor can also stop the process by entering an empty line or pressing “CANCEL”. In that case, the function should return `null`.

Run the demo

Open a sandbox with tests.

solution

An occasional infinite loop

importance: 4

This loop is infinite. It never ends. Why?

```
let i = 0;
while (i != 10) {
  i += 0.2;
}
```

solution

A random number from min to max

importance: 2

The built-in function `Math.random()` creates a random value from `0` to `1` (not including `1`).

Write the function `random(min, max)` to generate a random floating-point number from `min` to `max` (not including `max`).

Examples of its work:

```
alert( random(1, 5) ); // 1.2345623452
alert( random(1, 5) ); // 3.7894332423
alert( random(1, 5) ); // 4.3435234525
```

solution

A random integer from min to max

importance: 2

Create a function `randomInteger(min, max)` that generates a random *integer* number from `min` to `max` including both `min` and `max` as possible values.

Any number from the interval `min..max` must appear with the same probability.

Examples of its work:

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

You can use the solution of the previous task as the base.

solution

Previous lesson Next lesson

Share

Tutorial map

Comments

- If you have suggestions what to improve - please submit a GitHub issue or a pull request instead of commenting.
- If you can't understand something in the article – please elaborate.
- To insert a few words of code, use the `<code>` tag, for several lines – use `<pre>`, for more than 10 lines – use a sandbox (plnkr, JSBin, codepen…)