Working With the Number Type

In this lesson, we will cover the number type in JavaScript.
Let's begin!

WE'LL COVER THE FOLLOWING

JavaScript does not make a distinction among types of numbers, such as integers and floating-point numbers with different lengths.

JavaScript only has a single number type that represents both integers and floating point numbers using the IEEE-754

format.

You can check the `Number.MIN_VALUE` and the `Number.MAX_VALUE` constants to know this representation's limits:

```
MIN_VAULE: 5e-324
MAX_VALUE: 1.7976931348623157e+308
```

Despite the single storage format, JavaScript provides ***several literal formats.***

Integer literals

Integer numbers can be represented with **decimal**, **octal**, and **hexadecimal** literals. Most frequently, decimal literals are used:

```
var smallInt = 73;
var longInt = 12345678901234567;
var negInt = -162;
```

When the integer literal starts with 0, it is parsed as an octal literal, unless a digit out of the 0-7 range is detected, in this case, the number is interpreted as a decimal number:

```
var octal = 046;    // decimal 38
var invOctal = 085; // decimal 85!
```

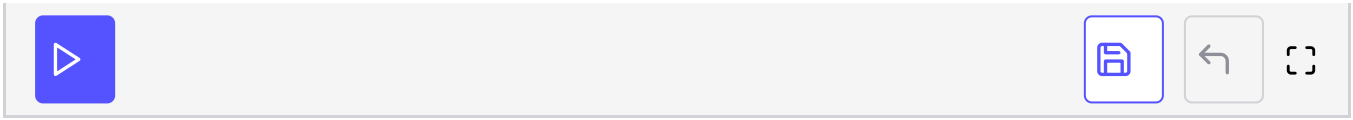
> 📜**NOTE:** In strict mode, decimal numbers with leading zeros are not allowed. To sign that a number is octal, use the 0o prefix (zero followed by a lowercase or uppercase letter O).

The second literal is not a valid octal number, so it is interpreted as decimal

85.

To define a hexadecimal integer literal, use the `0x` prefix and use hexadecimal digits either with lowercase or uppercase letters:

```
var hex1 = 0x3e;   // decimal 62
var hex2 = 0x2A4C; // decimal 10828
```



It does not matter which literal format you use to define an integer number; the JavaScript engine processes them the *same* way.

Floating-point literals

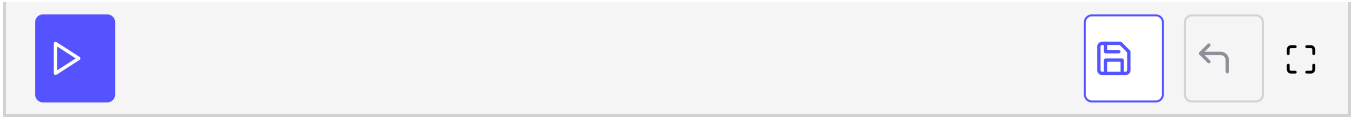
When you use a decimal point and at least one decimal digit after the decimal point, the number is interpreted as a floating-point number. You can even omit the integer part as an implicit 0 is assumed.

Alternatively, you can use the e-notation, and in this case you do not need to specify a decimal point.

Examples

Here are a few samples:

```
var fl1 = 12.34;
var fl2 = .467;
var fl3 = 1.527e13;
var fl4 = 1e5;
```

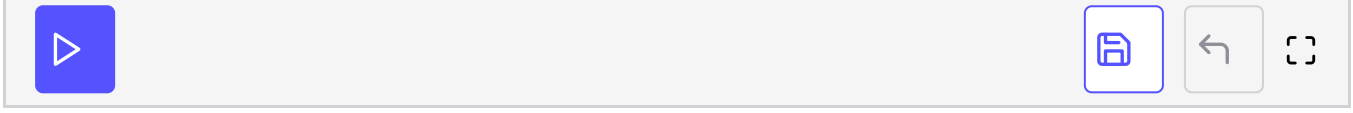


According to the **IEEE-754** format used to store the Number values, floating point numbers are accurate to **17 decimal places**, but far less accurate in arithmetic calculations than integer numbers.

As you know, these numbers use **binary format** internally. Because several decimal fractional numbers with final factional digits (such as 0.6) cannot be represented with final number of digits with binary base, it's not a good idea

to compare decimal numbers (numbers with fractional parts) directly:

```
console.log(0.3 + 0.3 == 0.6); // true
console.log(0.5 + 0.1 == 0.6); // true
console.log(0.2 + 0.4 == 0.6); // false
```



> 📜**NOTE:** As another consequence of the IEEE-754 format, it is possible in JavaScript to have positive zero (`+0`) and negative zero (`-0`) values. These are considered to be equivalent.

Infinity

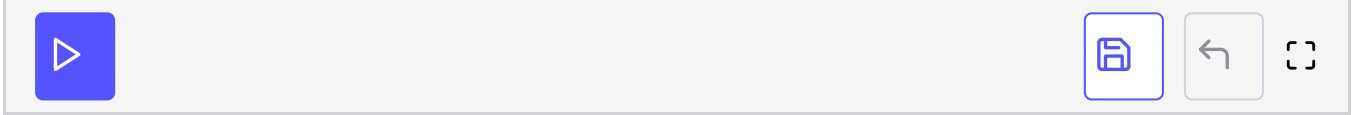
Calculations with numbers may result in a value that is out of the range the Number type that the IEEE-754 format utilized in the background can represent. In this case, the result automatically gets a special value, Infinity, or -Infinity, provided the calculation yields a negative number that cannot be represented in the allowed range.

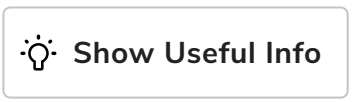
These short calculations show you examples:

Examples

```
// Results Infinity
var pInf = Number.MAX_VALUE * 2;

// Results -Infinity
var nInf = Number.MAX_VALUE * -1.2 - 100;
```

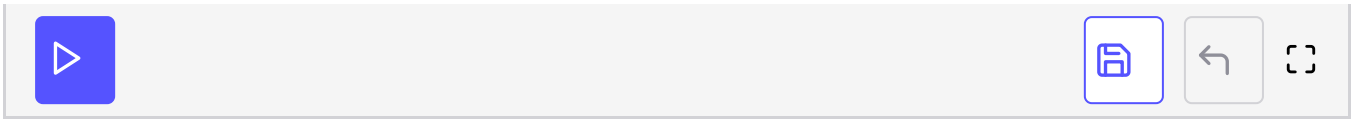




As it was mentioned earlier, dividing by 0 also results Infinity or –Infinity depending on the sign of the dividend:

```
console.log(1/0);  // Infinity
```

```
console.log(1/-0); // -Infinity
console.log(-1/0); // -Infinity
```

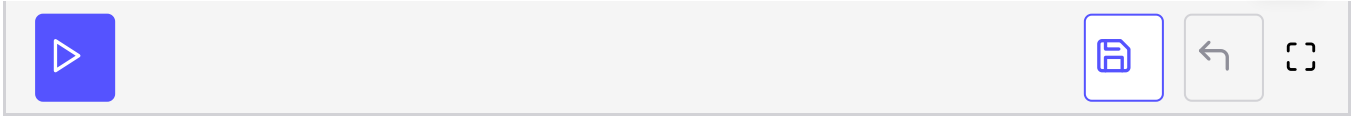


Not-A-Number

There are operations in JavaScript, where a numeric value is expected in the result. However, these operations may not provide a numeric result, such as this:

Examples

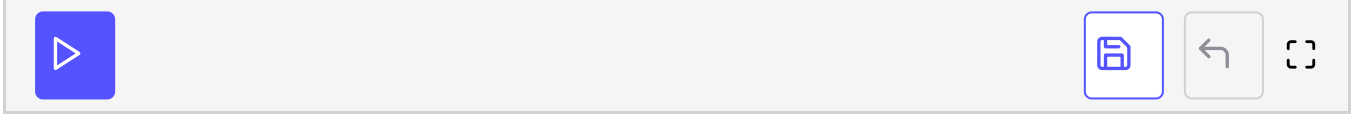
```
var int = parseInt("q123");
```



The `parseInt()` method cannot interpret `“q123”` as a valid integer. Instead of raising an error, it gives back NaN, a special value representing “not-a-number”. NaN shows unique behavior. Any calculations involving NaN always return NaN. Sounds odd, but NaN is not equal with any values, including NaN itself.

To check whether a certain operation results NaN, JavaScript provides the `isNaN()` function:

```
console.log(isNaN(NaN));       // true
console.log(isNaN("23.3e45")); // false
console.log(isNaN(1/0));       // false
console.log(isNaN("Linux"));   // true
```



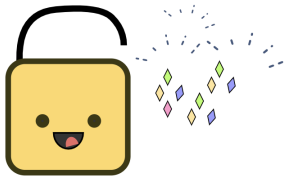
The `isNaN()` function can be applied upon objects as well, although, this is not a typical use:

```
console.log(isNaN(null));            // false
console.log(isNaN(new Number(123))); // false
console.log(isNaN(new String("3"))); // false
console.log(isNaN(new String("q"))); // true
```

If the argument of `isNaN()` is an object, it first checks if it can be converted to a number. If it does not work, it uses the `valueOf()` method of the object to check whether that can be converted to a number. Should this check fail, `isNaN()` uses the `toString()` method to check if the string representation of the object can be converted to a number. If all conversion steps fail, the object is `NaN`.

So `isNaN(null)` returns false, because null can be converted to a number (0, as you will learn soon). `IsNaN(new String("q"))` yields true, because all conversion attempts fail.

Achievement unlocked!



Congratulations! You've learned all about the number type in JavaScript.

Great work!

Give yourself a round of applause! :)

In the *next lesson*, we'll see how to convert numbers.

Stay tuned!