Working With Primitives

JavaScript

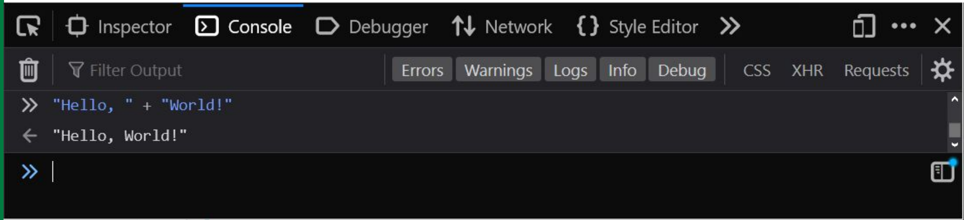
Dealing with Strings.

Concatenation

We can “glue” two pieces of text together into one string using concatenation.

+ is the concatenation operator. When it is found between strings, it will glue the text together!

Especially once variables get involved, this is extremely common.



```
>> "Hello, " + "World!"
← "Hello, World!"
>>
```

toUpperCase

A method of strings, capable of turning text into a new uppercase copy of the string.

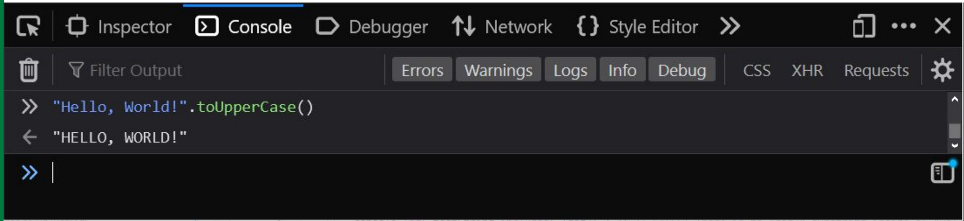
To use a string method, follow a string value or variable with a period, the name of the method, and finally a pair of parentheses.

See the pattern in action below with the toUpperCase string method!

“Hello, World!”.toUpperCase()

This example will return to you the text: “HELLO, WORLD!”

Don’t forget that these methods will all be capital-sensitive—it’s the JavaScript way!



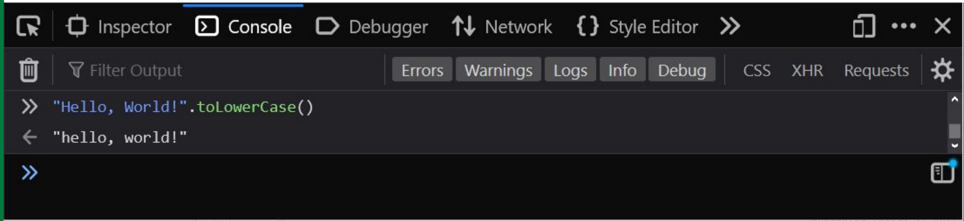
```
>> "Hello, World!".toUpperCase()
<- "HELLO, WORLD!"
>>
```

toLowerCase

A method of strings, capable of turning text into a new lowercase copy of the string.

As you can likely guess after trying out toUpperCase, the toLowerCase method does the opposite: a lower-cased version of the provided string will be returned to you.

“Hello, World!”.toLowerCase()



```
>> "Hello, World!".toLowerCase()
<- "hello, world!"
>>
```

includes

A method of strings, it checks if the string has a set of characters inside of it or not.

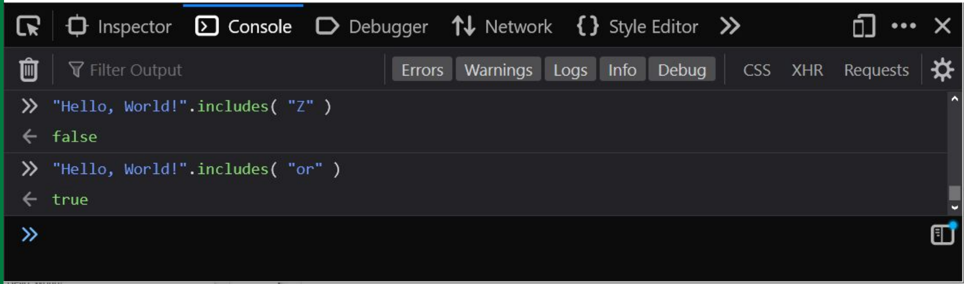
You enter a string as an argument to the includes method, and it will determine if that text exists inside of the string you're operating on.

The following will result in false, as "Z" is not inside of "Hello, World!".

"Hello, World!".includes("Z")

The following will result in true, as "or" does exist inside of "Hello, World!".

"Hello, World!".includes("or")



```
>> "Hello, World!".includes( "Z" )
<- false
>> "Hello, World!".includes( "or" )
<- true
>>
```

slice

A method of strings, it provides a cut version of the string containing only some of the text.

slice takes two different arguments: the position of the beginning letter and the position of the last letter you'd like to cut out.

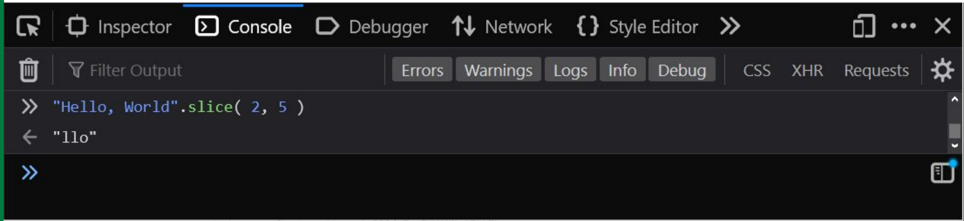
Count the letters in the string and you'll see!

"Hello, World!".slice(2, 5)

This cuts out and returns to you just "llo", from the original string "Hello, World!".

Note that letter position starts at zero, not one!

Try changing the numbers between the parentheses, what happens?



```
» "Hello, World".slice( 2, 5 )
← "llo"
»
```

replace

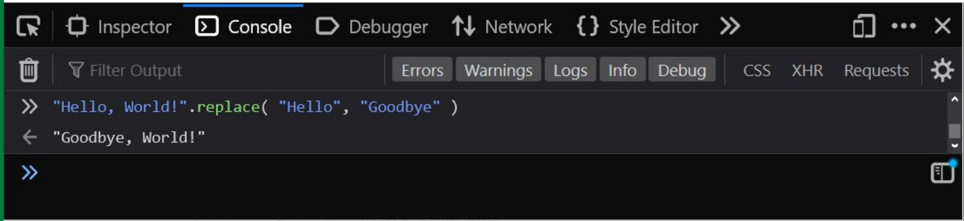
A method of strings, it replaces target text with new text.

replace takes two arguments: the text you'd like to target in the string, and the text you'd like to have take its place! It will return to you, a copy of the string with the replacement completed (if it found the target match, of course!)

Try this out…

“Hello, World!”.replace(“Hello”, “Goodbye”)

You'll get “Goodbye, World!” after the replacement has done its magic.



```
>> "Hello, World!".replace( "Hello", "Goodbye" )
<- "Goodbye, World!"
>>
```

Regular Expressions with replace

The power of pattern-matching within strings can be quickly multiplied with the use of regular expressions ("RegEx".)

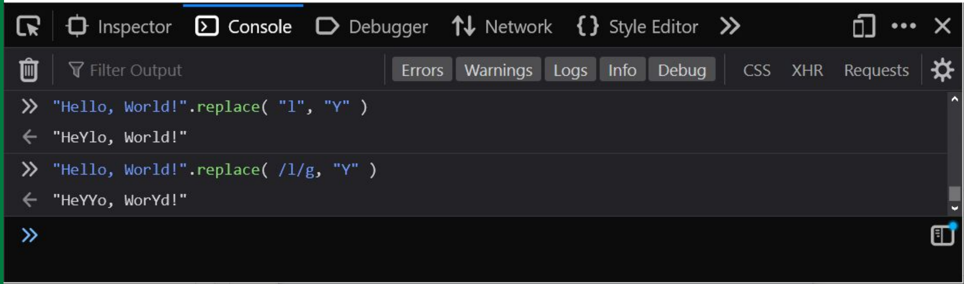
Note that one of the limitations (see figure below) of replace when passing a string in as your target, is that it will only replace the first match.

You can target more than one match using advanced pattern-matching via Regular Expressions, or, RegEx.

Try the following:

"Hello, World!".replace(/l/g, "Y")

This will return to you the updated string "HeYYo WorYd!"

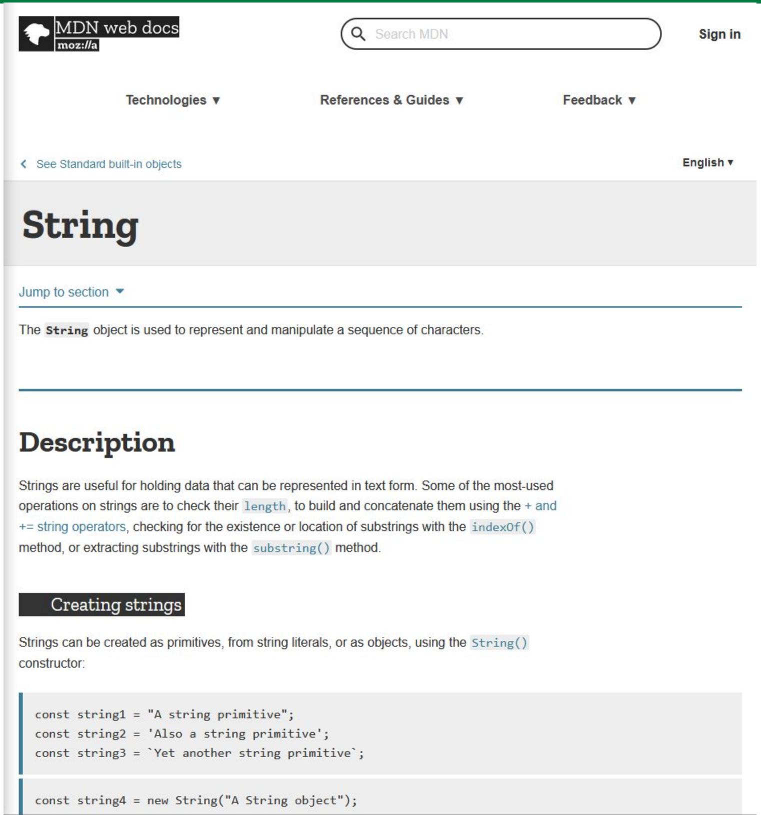


```
>> "Hello, World!".replace( "l", "Y" )
<- "HeYlo, World!"
>> "Hello, World!".replace( /l/g, "Y" )
<- "HeYYo, WorYd!"
>>
```

String Properties and Methods

There are plenty of more ways you can work with strings via their properties and methods.

Peruse the full list here.



String

The `String` object is used to represent and manipulate a sequence of characters.

Description

Strings are useful for holding data that can be represented in text form. Some of the most-used operations on strings are to check their `length`, to build and concatenate them using the + and += string operators, checking for the existence or location of substrings with the `indexOf()` method, or extracting substrings with the `substring()` method.

Creating strings

Strings can be created as primitives, from string literals, or as objects, using the `String()` constructor:

```
const string1 = "A string primitive";
const string2 = 'Also a string primitive';
const string3 = `Yet another string primitive`;
```

```
const string4 = new String("A String object");
```

Working with Numbers.

Mathematical Operators

We can do math with JavaScript! Much of it will look pretty familiar.

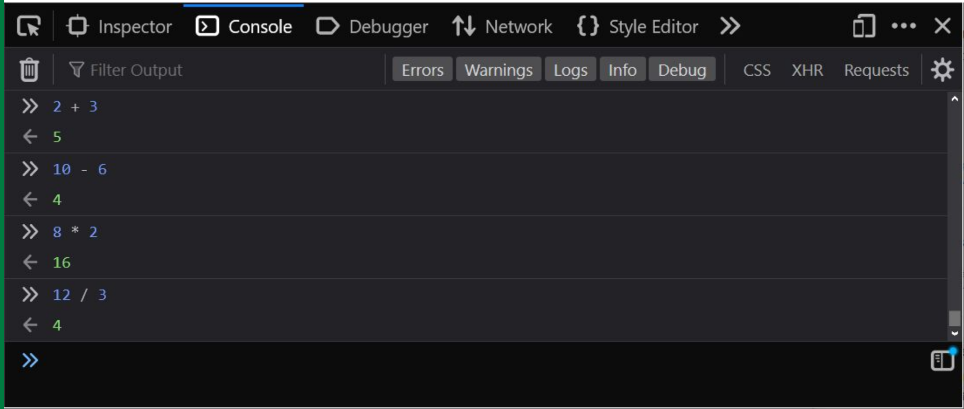
+ the addition operator.

- the subtraction operator.

***** the multiplication operator.

/ the division operator.

These work the way you'd expect! Give them a try in your Web Console.



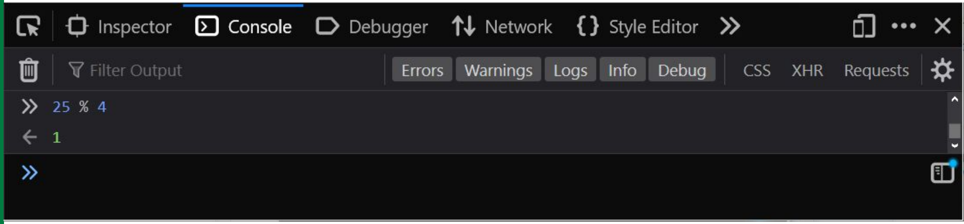
```
>> 2 + 3
<- 5
>> 10 - 6
<- 4
>> 8 * 2
<- 16
>> 12 / 3
<- 4
>>
```

Modulus Operator

This symbol may feel a little less familiar—it is just a division remainder!

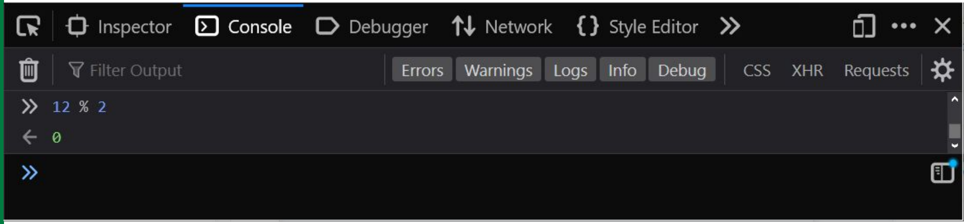
% is the modulus operator.

You can get the remainder of what would-be a division operation using this.



```
>> 25 % 4
<- 1
```

As an example, is commonly used for checking if a number is even or odd.



```
>> 12 % 2
<- 0
```

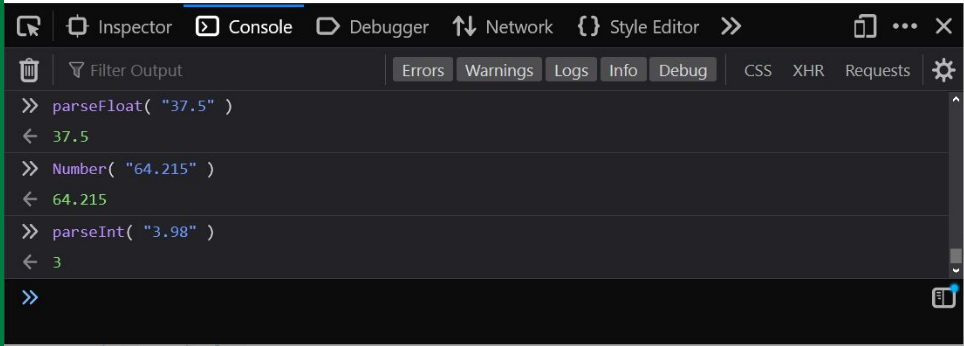
parseInt, parseFloat, and Number

We can transform strings composed of numeral characters into number data-type values.

The parseFloat and Number can be used to ensure a value is returned as a number (if possible.) If there are decimal values, they will be maintained.

parseInt works much the same, but will chop off any decimal points. Note that parseInt doesn't round up or down.

You can pass strings with numeral characters, or full-on numbers into these functions.



```
» parseFloat( "37.5" )
← 37.5
» Number( "64.215" )
← 64.215
» parseInt( "3.98" )
← 3
»
```

Addition Versus Concatenation

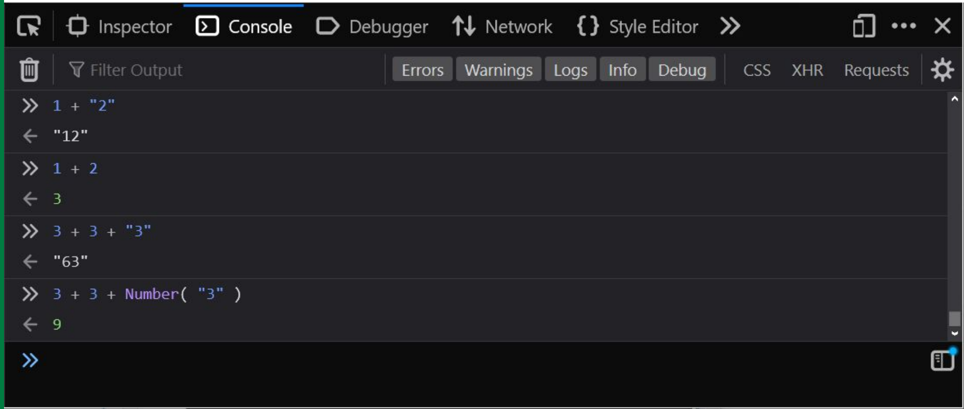
You may have noticed that JavaScript uses the same symbol for both addition and concatenation, we do have to be careful.

If a value on either side of a plus sign is a string, the operation will be concatenation.

If both values are of the number data-type, the operation will be addition.

Note you can leverage parseInt, parseFloat, and Number to convert strings if you need to ensure addition.

Try some of your own experiments!

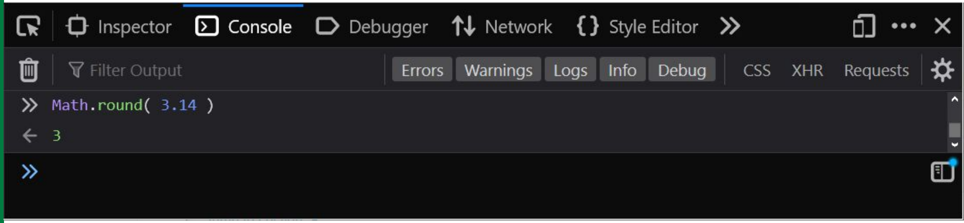


```
>> 1 + "2"
<- "12"
>> 1 + 2
<- 3
>> 3 + 3 + "3"
<- "63"
>> 3 + 3 + Number( "3" )
<- 9
>>
```

Math Round, Floor, and Ceiling

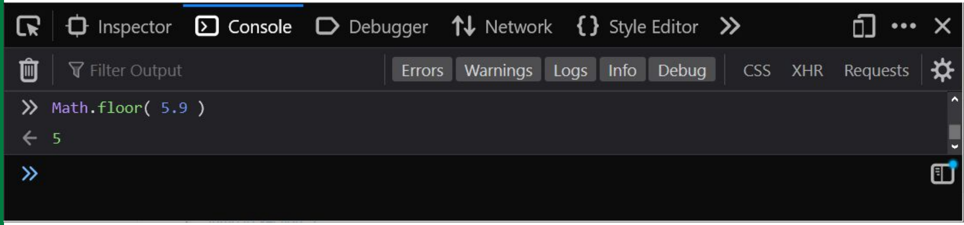
JavaScript's built-in Math class has some great methods.

Math.round is used for traditional rounding of a number to the nearest integer (whole number.)



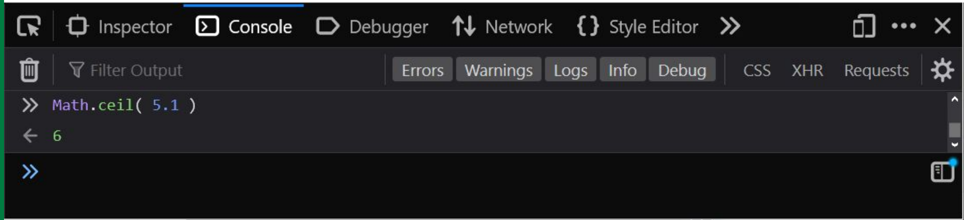
```
>> Math.round( 3.14 )
← 3
```

Math.floor will round down to the nearest integer.



```
>> Math.floor( 5.9 )
← 5
```

Math.ceil will round up to the nearest integer.

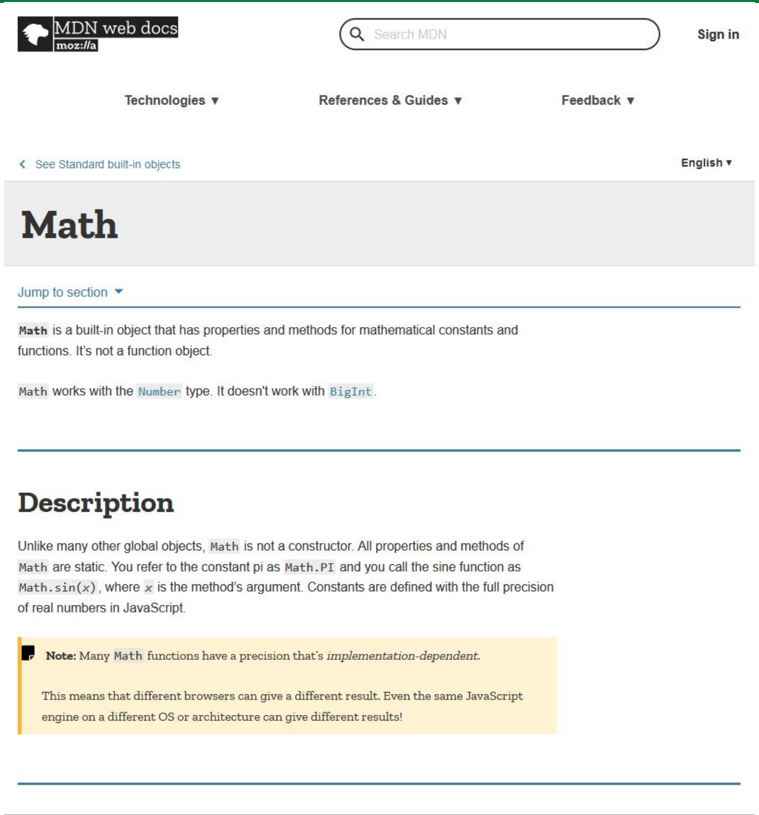


```
>> Math.ceil( 5.1 )
← 6
```

Math Object

JavaScript's Math class has a lot more built-in properties and methods.

Check out the full list to see what it is capable of!



Math

`Math` is a built-in object that has properties and methods for mathematical constants and functions. It's not a function object.

`Math` works with the `Number` type. It doesn't work with `BigInt`.

Description

Unlike many other global objects, `Math` is not a constructor. All properties and methods of `Math` are static. You refer to the constant pi as `Math.PI` and you call the sine function as `Math.sin(x)`, where `x` is the method's argument. Constants are defined with the full precision of real numbers in JavaScript.

Note: Many `Math` functions have a precision that's *implementation-dependent.*

This means that different browsers can give a different result. Even the same JavaScript engine on a different OS or architecture can give different results!

One More Thing! NaN and Infinity

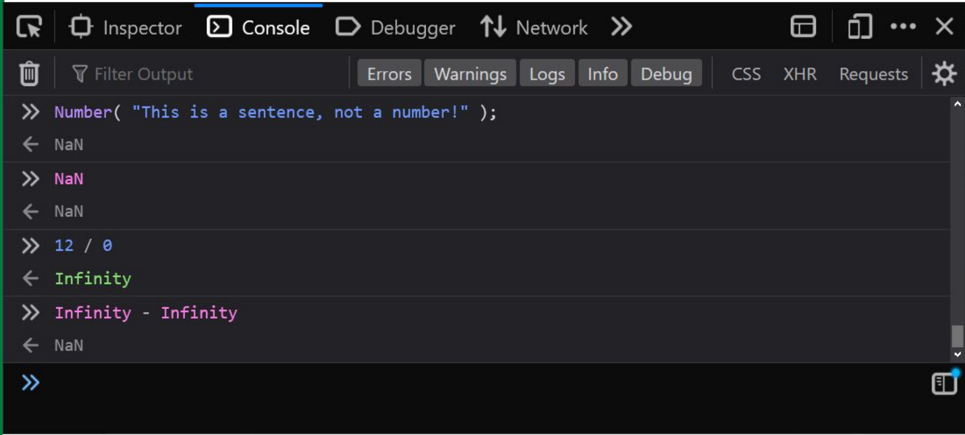
Another couple of results that may catch you off guard when dealing with numbers are the NaN and Infinity values.

NaN stands for “Not a Number.”

If a mathematical operation cannot be understood by JavaScript, or is impossible to express a result for within JavaScript’s ability, you’ll see this as the returned result.

JavaScript also has a representation for infinity (∞), intuitively named Infinity.

Remember: JavaScript is capital-sensitive!



```
>> Number( "This is a sentence, not a number!" );
<- NaN
>> NaN
<- NaN
>> 12 / 0
<- Infinity
>> Infinity - Infinity
<- NaN
>>
```