

NaN

"NaN" stands for:

Not a Number

What kinds of things give us NaN?

Fuzzy math

```
console.log(  
  0 / 0,  
  Infinity / Infinity,  
  0 * Infinity,  
  Infinity - Infinity  
);
```

```
> NaN NaN NaN NaN
```

Complex Numbers

```
console.log(  
    Math.sqrt(-1),  
    Math.log(-1),  
    Math.acos(2),  
    Math.asin(2)  
)
```

```
> NaN NaN NaN NaN
```

Turning things into Numbers

```
console.log(  
    parseInt('hello'), parseFloat('world'),  
    Number(undefined), Number({}),  
    +undefined, +{},  
    +new Date('hello'));
```

```
> NaN NaN NaN NaN NaN NaN NaN
```

What is NaN? (in JavaScript)

"Not a Number" is...

```
console.log(NaN);
```

> NaN

... a particular JavaScript value.

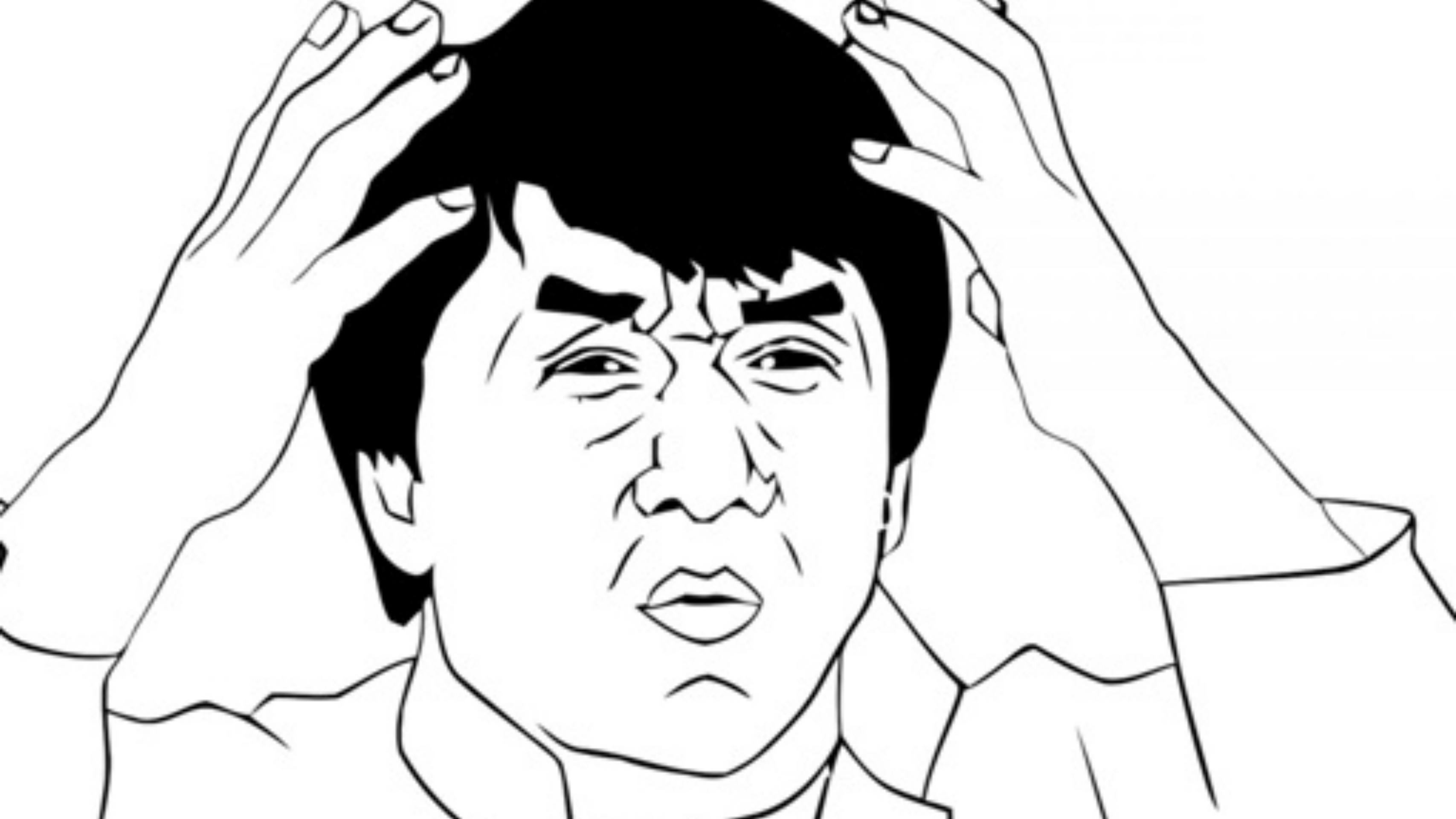
(very particular)

"Not a Number" is...

```
console.log(typeof NaN);
```

```
> number
```

...a Number.

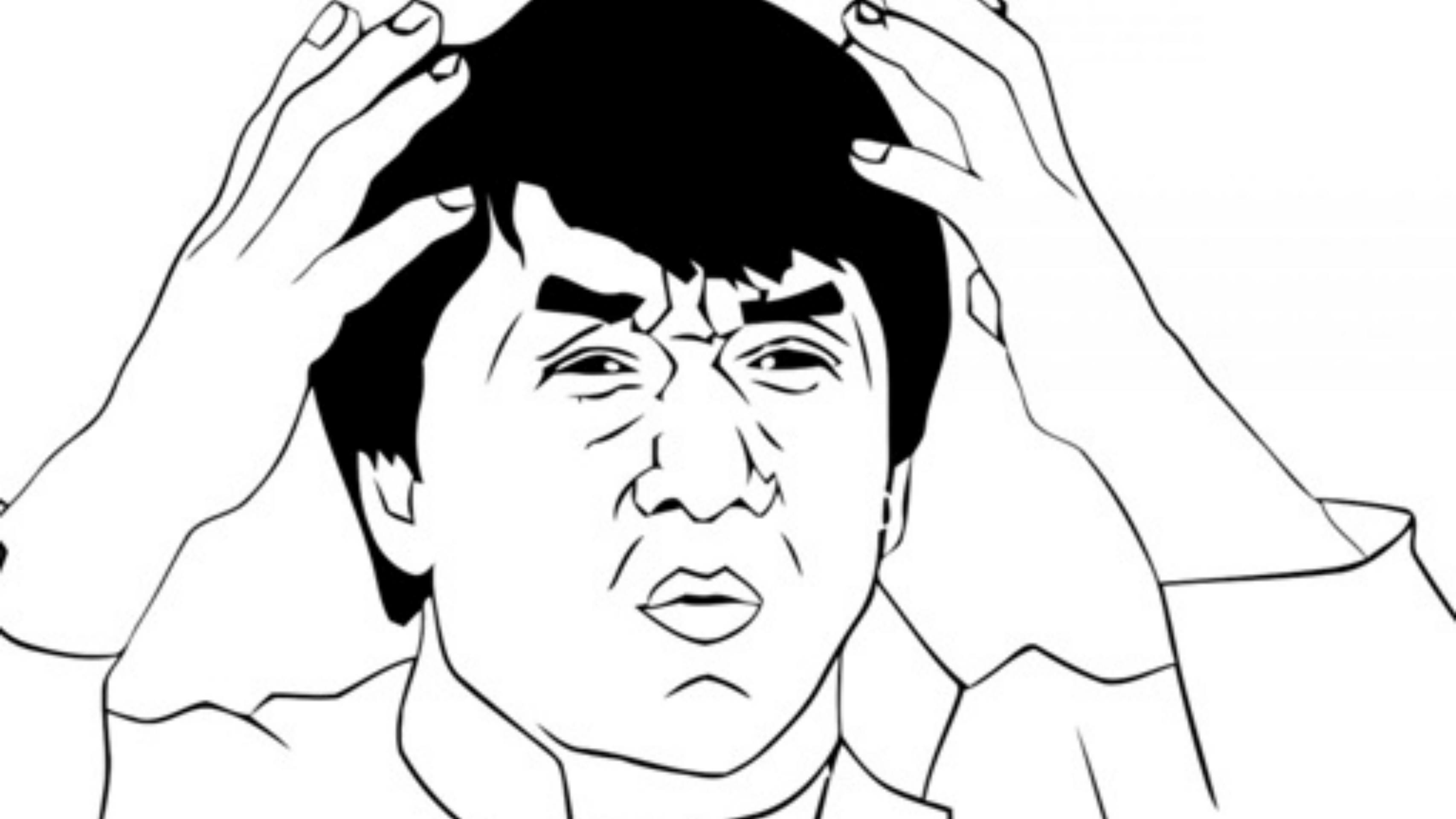


"Not a Number" is...

```
console.log(NaN === NaN);
```

```
> false
```

...not "Not a Number".



"Not a Number" is...

```
var assert = require('assert');
assert.equal(NaN, NaN);
```

```
> AssertionError: NaN == NaN
```

...tricky to test.

"NaN" actually stands for:

Not a NaN



Ariya Hidayat

@AriyaHidayat



Following

This is your annual reminder that NaN stands for
"Not a NaN".

RETWEETS

106

LIKES

48



8:22 AM - 23 Oct 2013



...

So how can we tell if something is
NaN?

Easy! Just use the isNaN function:

```
console.log(isNaN(NaN));
```

```
> true
```

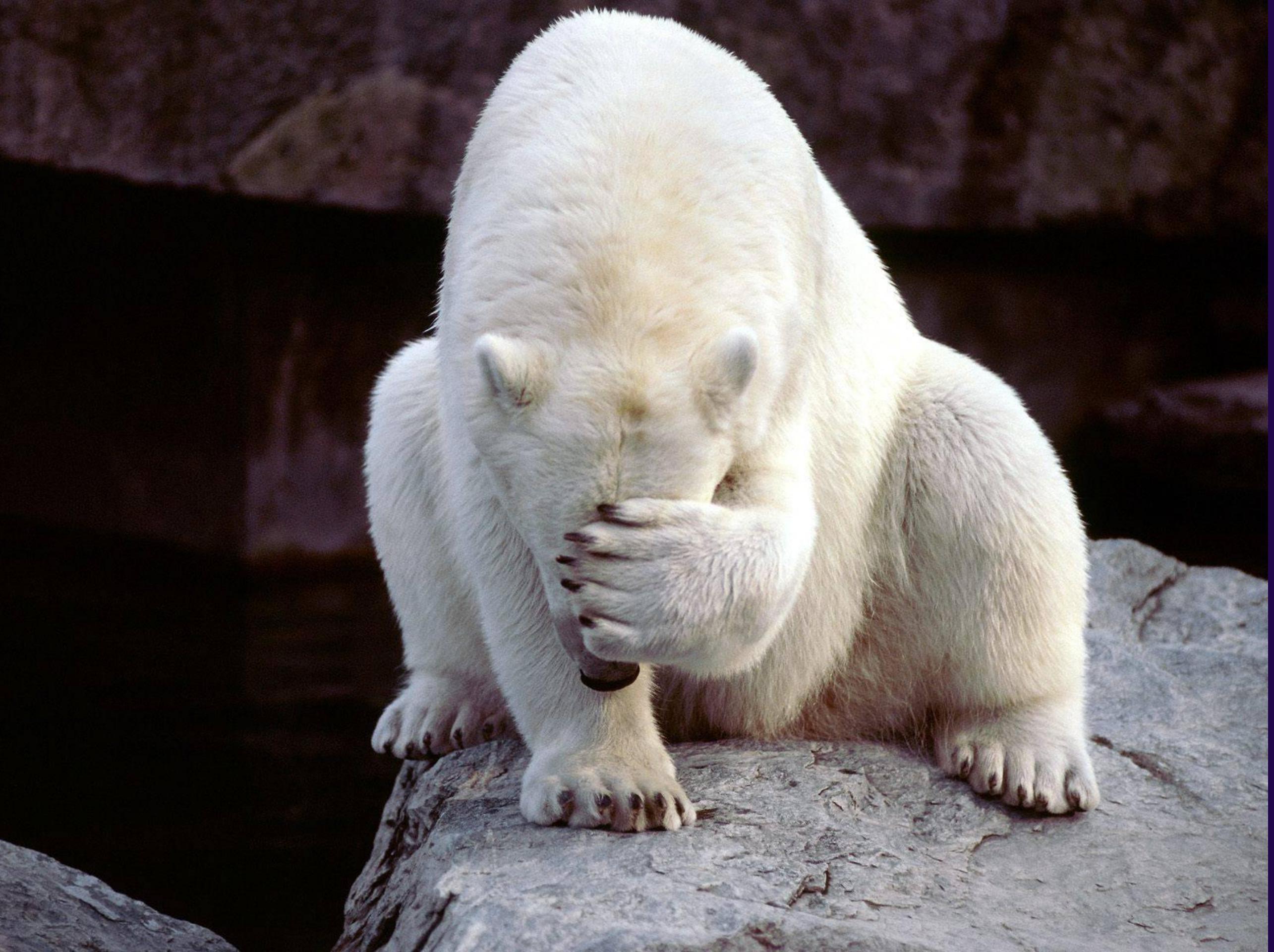
Or maybe not...

```
console.log(isNaN('foo'), isNaN(['bar']), isNaN({}));
```

```
> true true true
```

```
console.log(typeof 'foo', typeof ['bar'], typeof {});
```

```
> string object object
```



So let's just make our own:

```
function myIsNaN(x) {  
  return typeof x === 'number' && isNaN(x);  
  
}  
  
console.log([NaN, 'foo', ['bar'], {}].map(isNaN));  
console.log([NaN, 'foo', ['bar'], {}].map(myIsNaN));  
  
> true true true true  
> true false false false
```

Or we can recall "Not a NaN":

```
function myIsNaN(x) {  
  return x !== x;  
}  
  
console.log([NaN, 'foo', ['bar'], {}].map(isNaN));  
console.log([NaN, 'foo', ['bar'], {}].map(myIsNaN));  
  
> true true true true  
> true false false false
```

This works because NaN is the only value in JavaScript for which the equality operators are non-reflexive.

Fortunately, ES2015 adds Number.isNaN:

```
console.log([NaN, 'foo', ['bar'], {}].map(isNaN));  
console.log([NaN, 'foo', ['bar'], {}].map(Number.isNaN));
```

...and it does what we want:

```
> true true true true  
> true false false false
```

Or we can use Object.is:

```
console.log([NaN, 'foo', ['bar'], {}].map(isNaN));  
console.log([NaN, 'foo', ['bar'], {}].map(n => Object.is(n, NaN)));
```

```
> true true true true  
> true false false false
```

This uses the `SameValue` internal operation, which is (mostly) like how a Set distinguishes its elements.

But NaN isn't just a JavaScript
thing...

**NaN is actually defined by the IEEE
754 floating-point standard.**

If you know where NaN appears and how it behaves in one language, that carries over to most others.

Most.

Fun fact about that...

The IEEE 754 spec defines the pow function:

`pow(2, 3) -> 8`

`pow(-1, 1.5) -> NaN`

`pow(NaN, anything) -> NaN`

`pow(anything, NaN) -> NaN`

If either input is NaN, or if the base is negative and the exponent is not an integer, the result is NaN.

Three indeterminate form pows in IEEE 754:

`pow(0, 0) -> 1`

`pow(Infinity, 0) -> 1`

`pow(1, Infinity) -> 1`

This behavior is inherited from C99 and POSIX 2001.

Most languages follow this.

Here's what Python does:

```
[0 ** 0, float("inf") ** 0, 1 ** float("inf")]  
> [1 1.0 1.0]
```

And Ruby:

```
[0 ** 0, Float::INFINITY ** 0, 1 ** Float::INFINITY]  
> [1 1.0 1.0]
```

And Lua:

```
print(math.pow(0, 0), math.pow(math.huge, 0), math.pow(1, math.huge))
```

```
> 1 1 1
```

But JavaScript?

```
Math.pow(0, 0);
```

```
Math.pow(0, 0);
```

```
> 1
```

```
Math.pow(0, 0);
```

```
> 1
```

```
Math.pow(Infinity, 0);
```

```
Math.pow(0, 0);
```

```
> 1
```

```
Math.pow(Infinity, 0);
```

```
> 1
```

```
Math.pow(0, 0);
```

```
> 1
```

```
Math.pow(Infinity, 0);
```

```
> 1
```

```
Math.pow(1, Infinity);
```

```
Math.pow(0, 0);
```

```
> 1
```

```
Math.pow(Infinity, 0);
```

```
> 1
```

```
Math.pow(1, Infinity);
```

```
> NaN
```



Wahou

15.8.2.13 `pow (x, y)`

Returns an implementation-dependent approximation of x^y .

- If y is **NaN**, the result is **NaN**.
- If y is $+0$, the result is 1 , even if x is **NaN**.
- If y is -0 , the result is 1 , even if x is **NaN**.
- If x is **NaN** and y is nonzero, the result is **NaN**.
- If $\text{abs}(x) > 1$ and y is $+\infty$, the result is $+\infty$.
- If $\text{abs}(x) > 1$ and y is $-\infty$, the result is $+0$.
- If $\text{abs}(x) == 1$ and y is $+\infty$, the result is **NaN**.
- If $\text{abs}(x) == 1$ and y is $-\infty$, the result is **NaN**.

12.7.3.4 Applying the ****** Operator

- If **abs(base)** is 1 and *exponent* is $+\infty$, the result is **NaN**.
- If **abs(base)** is 1 and *exponent* is $-\infty$, the result is **NaN**.

- ES1 specifies pow: 1997
- C99 specifies pow: 1999
- POSIX specifies pow: 2001
- IEEE 754 inherits pow: 2008

NOTE

The result of `base ** exponent` when `base` is 1 or -1 and `exponent` is `+Infinity` or `-Infinity` differs from IEEE 754-2008. The first edition of ECMAScript specified a result of `NaN` for this operation, whereas later versions of IEEE 754-2008 specified `1`. The historical ECMAScript behaviour is preserved for compatibility reasons.

So just like every other question about
JavaScript, the answer is...

Backwards
compatibility

So anyway, what does IEEE 754 say about how we represent NaN?

Bit representation of a float32 value:

0 1000000 010000000000000000000000

- 1-bit sign
- 8-bit exponent, offset by 127
- 23-bit significand (with implicit leading 24th bit)
- $(-1)^s \cdot 2^{(exp - 127)} \cdot 1.\text{significand}$

Example float32 value:

0 1000000 010000000000000000000000

$$\Rightarrow (-1)^{\wedge} 0 = 1$$

$$\Rightarrow 2^{\wedge} (1000000b - 127) = 2$$

$$\Rightarrow 1.01b = 1.25$$

$$\Rightarrow 1 * 2 * 1.25 = 2.5$$

Bit representations of special values:

0 1111111 00000000000000000000000000000000 -> Infinity

1 1111111 00000000000000000000000000000000 -> -Infinity

Infinity values have a maximized exponent and a zero significand.

Bit representations of special values:

0 11111111 10000000000000000000000000000000 → NaN

NaN values have a maximized exponent and a nonzero significand.

So these are also all NaN:

| | | |
|-----------|--------------------------|-----------------------------------|
| 1 1111111 | 100000000000000000000000 | -> NaN (quiet, negative) |
| 0 1111111 | 100000000000000000000001 | -> NaN (quiet, but different) |
| 0 1111111 | 000000000000000000000001 | -> NaN (signaling) |
| 0 1111111 | 000000000000000000000010 | -> NaN (signaling, but different) |
| 0 1111111 | 000000000000000000000011 | -> NaN (we can start counting!) |

So these are also all NaN:

| | | | |
|---|----------|--------------------------|-----------------------------------|
| 1 | 11111111 | 100000000000000000000000 | -> NaN (quiet, negative) |
| 0 | 11111111 | 100000000000000000000001 | -> NaN (quiet, but different) |
| 0 | 11111111 | 000000000000000000000001 | -> NaN (signaling) |
| 0 | 11111111 | 000000000000000000000010 | -> NaN (signaling, but different) |
| 0 | 11111111 | 000000000000000000000011 | -> NaN (we can start counting!) |

How many NaNs are there, really?

$$2^{24} - 2 = 16,777,214$$

And that's just with a float32!

What about a double64?

$$2^{52} - 2 =$$

4,503,599,627,370,494

That's $4.5 * 10^{15}$, or 4.5 quadrillion.

4.5 petabytes is about 10,000 years worth of music.

If there are so many different possible NaNs, then it only seems reasonable...

...that one random NaN is unlikely to
be the same as another random NaN!

Thus, $\text{NaN} \neq \text{NaN}^1$.

¹With probability $1/4,503,599,627,370,494$.

Some Related Links

- <http://ariya.ofilabs.com/2014/05/the-curious-case-of-javascript-nan.html>
- <http://www.2ality.com/2012/02/nan-infinity.html>
- <https://en.wikipedia.org/wiki/NaN>
- <https://tc39.github.io/ecma262/#sec-applying-the-exp-operator>

Who are you and where can I find the slides?

- I'm Lewis J Ellis: @lewisjellis on Twitter and Github
- My website is LewisJEllis.com.
- Slides available at Github.com/LewisJEllis/nantalk