How numbers are encoded in JavaScript

[2012-04-19] numbers, dev, javascript, jsint, jslang

All numbers in JavaScript are floating point. This blog post explains how those floating point numbers are represented internally in 64 bit binary. Special consideration will be given to integers, so that, after reading this post, you will understand what happens in the following interaction:

```
> 9007199254740992 + 1
9007199254740992

> 9007199254740992 + 2
9007199254740994
```

1 JavaScript numbers

JavaScript numbers are all floating point, stored according to the IEEE 754 standard. That standard has several formats. JavaScript uses *binary64* or *double precision*. As the former name indicates, numbers are stored in a binary format, in 64 bits. These bits are allotted as follows: The *fraction* occupies bits 0 to 51, the *exponent* occupies bits 52 to 62, the *sign* occupies bit 63.

sign (1 bit)	**exponent** (11 bit)	**fraction** (52 bit)
63	62 52	51 0

The components work as follows: If the sign bit is 0, the number is positive, otherwise negative. Roughly, the fraction contains the digits of a number, while the exponent indicates where the point is. In the following, we'll often use binary numbers, which is a bit unusual when it comes to floating point. Binary numbers will be marked by a prefixed percentage sign (%). While JavaScript numbers are stored in binary, the default output is decimal [1]. In the examples, we'll normally work with that default.

2 The fraction

The following is one way of representing non-negative floating point numbers: The *significand* (or *mantissa*) contains the digits, as a natural number, the exponent specifies how many digits to the left (negative exponent) or right (positive exponent) the point should be shifted. JavaScript numbers use a rational number as the significand: 1.f where f is the 52 bit fraction. Ignoring the sign, the number is the significand multiplied by 2^p where p is the exponent (after a transformation that will be explained later).

Examples:

f = %101, $p = 2$	Number: %1.101 × 2^2 = %110.1
f = %101, $p = -2$	Number: %1.101 × 2^{-2} = %0.01101
$f = 0$, $p = 0$	Number: %1.0 × 2^0 = %1

2.1 Representing integers

How many bits does the encoding give you for integers? The significand has 53 digits, one before the point, 52 after the point. With $p = 52$, we have a 53 bit natural number. The only problem is that the highest bit is always 1. That is, we don't have all of the bits freely at our disposal. One removes that limitation in two steps. First, if you need a 53 bit number whose highest bit is 0, followed by 1, you set $p = 51$. The lowest bit of the fraction then becomes the first digit after the point and is 0 for integers. And so on, until you are at $p = 0$ and $f = 0$, which encodes the number 1.

	52	**51**	**50**	**...**	**1**	**0**	**(bits)**
p=52	1	f_{51}	f_{50}	...	f_1	f_0	
p=51	0	1	f_{51}	...	f_2	f_1	f_0=0
...							
p=0	0	0	0	...	0	1	f_{51}=0, etc.

Second, for a full 53 bits, we still need to represent zero. How to do that is explained in the next section. Note that we have the full 53 bits for the magnitude (absolute value) of the integer, as the sign is stored separately.

3 The exponent

The exponent is 11 bit long, meaning its lowest value is 0, its highest value is 2047 ($2^{11}-1$). To support negative exponents, the so-called offset binary encoding is used: 1023 is the zero, all lower numbers are negative, all higher numbers are positive. That means that you subtract 1023 from the exponent to convert it to a normal number. Therefore, the variable p that we previously used equals $e-1023$ and the significand is multiplied by 2^{e-1023}.

A few numbers in offset binary encoding:

```
    %00000000000     0  →  -1023  (lowest number)
    %01111111111  1023  →      0
    %11111111111  2047  →   1024  (highest number)

    %10000000000  1024  →      1
    %01111111110  1022  →     -1
```

To negate a number, you invert its bits and subtract 1.

3.1 Special exponents

Two exponent values are reserved: The lowest one (0) and the highest one (2047). An exponent of 2047 is used for infinity and NaN (not a number) values [2]. The IEEE 754 standard has many NaN values, but JavaScript all represents them as a single value `NaN`. An exponent of 0 is used in two capacities. First, if the fraction is also 0 then the whole number is 0. As the sign is stored separately, we have both −0 and +0 (see [3] for details).

Second, an exponent of 0 is also used to represent very small numbers (close to zero). Then the fraction has to be non-zero and, if positive, the number is computed via

$$\%0.f \times 2^{-1022}$$

This representation is called *denormalized*. The previously discussed representation is called *normalized*. The smallest positive (non-zero) number that can be represented in a normalized manner is

$$\%1.0 \times 2^{-1022}$$

The largest denormalized number is

$$\%0.1 \times 2^{-1022}$$

Thus, there is no hole when switching between normalized and denormalized numbers.

3.2 Summary: exponents

$(-1)^s \times \%1.f \times 2^{e-1023}$	normalized, $0 < e < 2047$
$(-1)^s \times \%0.f \times 2^{e-1022}$	denormalized, $e = 0, f > 0$
$(-1)^s \times 0$	$e = 0, f = 0$
NaN	$e = 2047, f > 0$
$(-1)^s \times \infty$ (infinity)	$e = 2047, f = 0$

With $p = e - 1023$, the exponent has a range of

$$-1023 < p < 1024$$

4 Decimal fractions

Not all decimal fractions can be represented precisely in JavaScript, as illustrated by the following result:

```
> 0.1 + 0.2
0.30000000000000004
```

Neither of the decimal fractions 0.1 and 0.2 can be represented precisely as a binary floating point number. However, the deviation from the actual value is usually too small to be displayed. Addition leads to that deviation becoming visible. Another example:

```
> 0.1 + 1 - 1
0.10000000000000009
```

Representing 0.1 amounts to the challenge of representing the fraction 110. The difficult part is the denominator 10, whose prime factorization is 2 × 5. The exponent only lets you divide an integer by a power of 2, so there is no way of getting a 5 in. Compare: 13 cannot be represented precisely as a decimal fraction. It is approximated by 0.333333...

In contrast, representing a binary fraction as a decimal fraction is always possible, you just need to collect enough twos (of which every ten has one). For example:

$\%0.001 = 18 = 12 \times 2 \times 2 = 5 \times 5 \times 5(2\times5) \times (2\times5) \times (2\times5) = 12510 \times 10 \times 10 = 0.125$

4.1 Comparing decimal fractions

Hence, when you work with decimal input that has fractional values, you should never compare them directly. Instead, take an upper bound for rounding errors into consideration. Such an upper bound is called a *machine epsilon*. The standard epsilon value for double precision is 2^{-53}.

```
var epsEqu = function () { // IIFE, keeps EPSILON private
    var EPSILON = Math.pow(2, -53);
    return function epsEqu(x, y) {
        return Math.abs(x - y) < EPSILON;
    };
}();
```

The above function ensures correct results where normal comparison would be inadequate:

```
> 0.1 + 0.2 === 0.3
false
> epsEqu(0.1+0.2, 0.3)
true
```

5 The maximum integer

What does one mean if one says "x is the maximum integer"? It means that every integer n in the range $0 \le n \le x$ can be represented and that the same does not hold for any integer greater than x. 2^{53} fits that bill. All previous numbers can be represented:

```
> Math.pow(2, 53)
9007199254740992
> Math.pow(2, 53) - 1
9007199254740991
> Math.pow(2, 53) - 2
9007199254740990
```

But the next integer cannot be represented:

```
> Math.pow(2, 53) + 1
9007199254740992
```

A few aspects of 2^{53} being the upper limit might be surprising. We will look at them via a series of questions. One thing to keep in mind is that the limiting resource at the high end of the integer range is the fraction; the exponent still has room to grow.

Why 53 bits? You have 53 bits available for the magnitude (excluding the sign), but the fraction comprises only 52 bits. How is that possible? As you have seen above, the exponent provides the 53rd bit: It shifts the fraction, so that all 53 bit numbers except the zero can be represented and it has a special value to represent the zero (in conjunction with a fraction of 0).

Why is the highest integer not 2^{53}–1? Normally, x bit mean that the lowest number is 0 and the highest number is 2^x–1. For example, the highest 8 bit number is 255. In JavaScript, the highest fraction is indeed used for the number 2^{53}–1, but 2^{53} can be represented, thanks to the help of the exponent – it is simply a fraction $f = 0$ and an exponent $p = 53$ (after conversion):

$$\%1.f \times 2^p = \%1.0 \times 2^{53} = 2^{53}$$

Why can numbers higher than 2^{53} be represented? Examples:

```
> Math.pow(2, 53)
9007199254740992
> Math.pow(2, 53) + 1  // not OK
9007199254740992
> Math.pow(2, 53) + 2  // OK
9007199254740994
```

```
> Math.pow(2, 53) * 2  // OK
18014398509481984
```

$2^{53}×2$ works, because the exponent can be used. Each multiplication by 2 simply increments the exponent by 1 and does not affect the fraction. So multiplying by a power of 2 is not a problem as far as the maximum fraction is concerned. To see why one can add 2 to 2^{53}, but not 1, we extend the previous table with the additional bits 53 and 54 and rows for $p = 53$ and $p = 54$:

	54	**53**	**52**	**51**	**50**	**...**	**2**	**1**	**0**	**(bits)**
p=54	1	f_{51}	f_{50}	f_{49}	f_{48}	...	f_0	0	0	
p=53		1	f_{51}	f_{50}	f_{49}	...	f_1	f_0	0	
p=52			1	f_{51}	f_{50}	...	f_2	f_1	f_0	

Looking at the row (p=53), it should be obvious that JavaScript numbers can have bit 53 set to 1. But as the fraction f only has 52 bits, bit 0 must be zero. Hence, only even numbers x can be represented in the range $2^{53} \le x < 2^{54}$. In row (p=54), that spacing increases to multiples of four, in the range $2^{54} \le x < 2^{55}$:

```
> Math.pow(2, 54)
18014398509481984
> Math.pow(2, 54) + 1
18014398509481984
> Math.pow(2, 54) + 2
18014398509481984
> Math.pow(2, 54) + 3
18014398509481988
> Math.pow(2, 54) + 4
18014398509481988
```

And so on...

6 IEEE 754 exceptions

The IEEE 754 standard describes five *exceptions*, where one cannot compute a precise value:

1. **Invalid:** An invalid operation has been performed. For example, computing the square root of a negative number. Returns NaN [2].

```
> Math.sqrt(-1)
NaN
```

2. **Division by zero:** returns plus or minus infinity [2].

```
> 3 / 0
Infinity
> -5 / 0
-Infinity
```

3. **Overflow:** The result is too large to be represented. That means that the exponent is too high ($p \geq 1024$). Depending on the sign, there is positive and negative overflow. Returns plus or minus infinity.

```
> Math.pow(2, 2048)
Infinity
> -Math.pow(2, 2048)
-Infinity
```

4. **Underflow:** The result is too close to zero to be represented. That means that the exponent is too low ($p \leq -1023$). Returns a denormalized value or zero.

```
> Math.pow(2, -2048)
0
```

5. **Inexact:** An operation has produced an inexact result – there are too many significant digits for the fraction to hold. Returns a rounded result.

```
> 0.1 + 0.2
0.30000000000000004

> 9007199254740992 + 1
9007199254740992
```

#3 and #4 are about the exponent, #5 is about the fraction. The difference between #3 and #5 is very subtle: In the second example given for #5, we are exceeding the upper limit of the fraction (which would be an overflow in integer computation). But only exceeding the upper limit of the exponent is called an overflow in IEEE 754.

7 Conclusion

In this blog post, we looked at how JavaScript fits its floating point numbers into 64 bits. It does so according to *double precision* in the IEEE 754 standard. Due to how numbers are displayed, one tends to forget that JavaScript cannot precisely represent a decimal fraction whose denominator's prime factorization contains a number other than 2. For example, 0.5 (12) can be represented, while 0.6 (35) cannot. One also tends to forget that the three components sign, exponent, fraction of a number work together to represent an integer. But one is confronted with that fact when `Math.pow(2, 53) + 2` can be represented, but `Math.pow(2, 53) + 1` cannot.

Bonus: The web page "IEEE-754 Analysis" allows you to enter a number and look at its internal representation.

8 Sources and related reading

Sources of this post:

- "IEEE Standard 754 Floating-Point" by Steve Hollasch.
- "Data Types and Scaling (Fixed-Point Blockset)" in the MATLAB documentation.
- "IEEE 754-2008" on Wikipedia.

This post is part of a series on JavaScript numbers, which includes:

1. Displaying numbers in JavaScript
2. NaN and Infinity in JavaScript
3. JavaScript's two zeros

2ality – JavaScript and more Comment Policy

HTML tags work! Use <pre><code> or <pre> for code blocks, <code> for inline code.

11 Comments **2ality – JavaScript and more** 1 **Login**

Recommend 3 Tweet Share Sort by Best

Join the discussion…

LOG IN WITH

OR SIGN UP WITH DISQUS ?

Name

Dave Cottlehuber • 8 years ago

Nice explanation, thanks for putting the work in on this. It would be useful to point out that JSON doesn't share the same properties