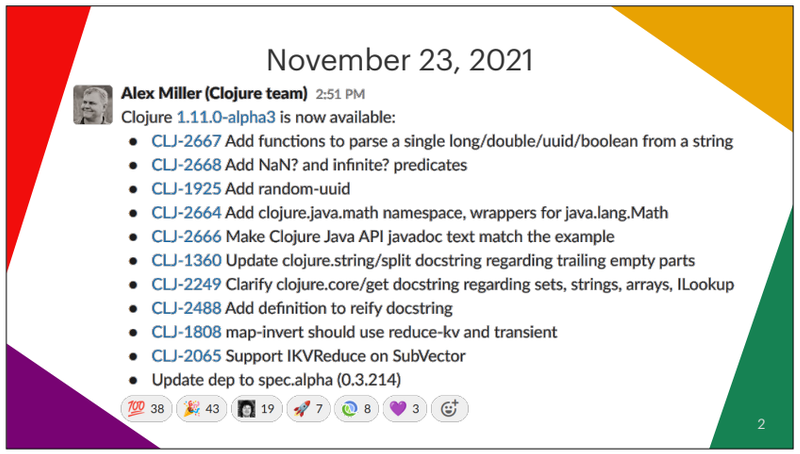
This is a tale of exploration, adventure, heartache and frustrations, achievements, and the friends we made along the way

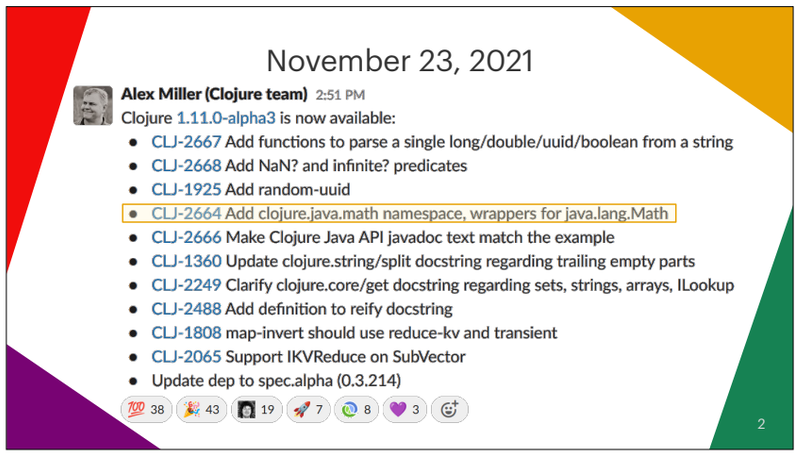
---



A long time ago in a Slack space far, far away…

Alex announced the new features and updates for Clojure 1.11, including `clojure.java.math`

A great development, because it's inconvenient to use interop for common operations. But this is specific to the JVM. When writing cross platform Clojure it would also be very frustrating to write conditional compilation for common operations.



Questions in #clojure-dev

Alex Miller (Clojure team) 11 months ago
there are a lot of guarantees in these methods - can those guarantees be met for JavaScript's notions of numbers (particularly 64-bit floats)? if yes, then that might make sense, but it was more than I wanted to bite off. if not, then what does it mean for it to be portable?

Alex Miller (Clojure team) 11 months ago
the goal here was to expose java.lang.Math, not to make a portable math library

3

So I asked Alex about this. After all, js/Math has a lot of overlap with java.lang.Math

Alex explained that he would be cautious about this, since there are certain numerical guarantees to these functions. Also, java.lang.Math has many more functions than js/Math. These were not issues that had been considered though, since portability had not been a goal for the core team. But this was a goal for me, because I'm someone who often tries to make my libraries cross-platform capable.

So what does Javascript math have in it? How close is it to what we need?

Constants

js/Math	java.lang.Math
E	E
PI	PI
LN10	TAU NEW
LN2	
LOG10E	
LOG2E	
SQRT1_2	
SQRT2	

4

It looks like js/Math has all of the constants, except TAU, but that's newer than any of these events. It's just 2π, and if you haven't heard about it, then I recommend reading the Tau Manifesto!

---

js/Math Functions

abs	ceil	imul	round
acos	clz32	log	sign
acosh	cos	log10	sin
asin	cosh	log1p	sinh
asinh	exp	log2	sqrt
atan	expm1	max	tan
atan2	floor	min	tanh
atanh	fround	pow	trunc
cbrt	hypot	random	

5

The function list is much larger, and looks reasonably capable in Javascript

Ported java.lang.Math Functions

acos	decrementExact	log	round	
addExact	exp	log10	scalb	
asin	expm1	log1p	signum	toRadians
atan	floor	multiplyExact	sin	ulp
atan2	floorDiv	negateExact	sinh	
cbrt	floorMod	nextAfter	sqrt	
ceil	getExponent	nextDown	subtractExact	
copySign	hypot	nextUp	tan	
cos	IEEEremainder	pow	tanh	
cosh	incrementExact	rint	toDegrees	

6

But not as large as the list in Java Math. Then again the source code of all the missing functions is available, so I could probably port it all, so long as the number format was still compatible.

What About Those Guarantees?

- All JavaScript numbers are floating point
- 64 bit IEEE-754 Floating Point Standard

7

What about those numbers?

Well, it turns out that all JavaScript numbers are floating point numbers

* Even the integers. OK. That's a little tricky, but ClojureScript already deals with these differences

But most importantly, JavaScript uses the IEEE-754 standard for 64 bit floating point numbers, which is exactly what Java does. So that was a great sign.

* The Mozilla Developer Network tries to provide a definitive set of documents for Javascript, and they make the point that there can be variations between different implementation, but in practice the divergence appears minimal. This is more of a "cover your bases" mentality.

What About Those Guarantees?

- All JavaScript numbers are floating point

  Yes: even the integers
- 64 bit IEEE-754 Floating Point Standard

7

What About Those Guarantees?

- All JavaScript numbers are floating point

  Yes: even the integers
- 64 bit IEEE-754 Floating Point Standard
- Mozilla notice:

  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!

7

Feedback

quoll 11 months ago
The implementation of js/Math also complies with IEEE-754, including the bit representations. The main thing is that js/Math is missing a few functions, but the source code for the bit manipulation is available in the JVM (e.g. `java.lang.Math/IEEEremainder`). I'm rewriting these for ClojureScript, though I'm probably not the only person doing that 🙂

Alex Miller (Clojure team) 11 months ago
well, happy to have feedback. if it turns out to be closer then maybe we'll move it.

8

Given what I'd seen, I figured I could probably fill in the gaps. So I made a start, and I told Alex what I was trying to do. Reading his response, I figured that he was willing to consider it

---

The Details: IEEE-754 64 Bit

$6.67430\times10^{-11} \Rightarrow 1.00100101100\times2^{-100010}$

$1.602176634\times10^{-19} \Rightarrow 1.01111010010\times2^{-111111}$

$2.7182818284\times10^{0} \Rightarrow 1.01011011111\times2^{1}$

$4.835978484\times10^{14} \Rightarrow 1.10110111110\times2^{110000}$

9

The basis of most of these math functions is the 64 bit floating point number. This is defined in the IEEE-754 spec, which you too can buy for $106, or $83 for IEEE members.

This format essentially represents numbers is scientific notation, only in binary. You can see some constants on the left there, each expressed as a decimal number greater than 1 and less than 10, called the Mantissa. Then there is a exponential of 10, which shows the final placement of the decimal point.

Well, IEEE-754 does the same thing, only with binary. Now we have a mantissa greater than one and less than 2, and an exponential of 2, which shows the final placement of the BINARY point.

* But, do you see how the first digit has to always be a 1? The standard takes advantage of this, and just never stores it. So it picks up an extra digit for free. There is an exception to this, and it's when the exponential is as low as it can go. Then it presumes the digit before the point is a 0. These are called the subnormal numbers, but it only applies for numbers smaller than 2 to the -1023

The Details: IEEE-754 64 Bit

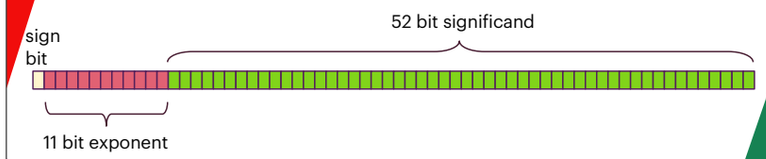
$6.67430\times10^{-11} \Rightarrow 1.00100101100\times2^{-100010}$

$1.602176634\times10^{-19} \Rightarrow 1.01111010010\times2^{-111111}$

$2.7182818284\times10^{0} \Rightarrow 1.01011011111\times2^{1}$

$4.835978484\times10^{14} \Rightarrow 1.10110111110\times2^{110000}$

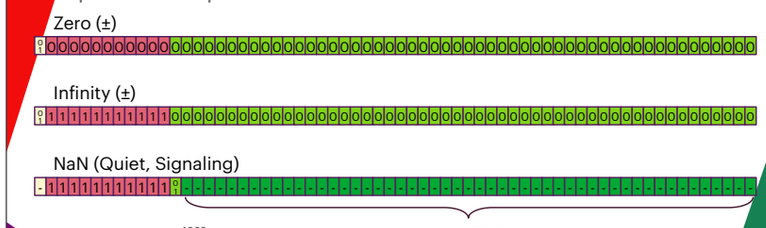
IEEE-754 64 Bit



Looking at the layout of this, we see the sign in the first position, the exponent in the next 11 bits, and the mantissa (WITHOUT the leading bit) in the remaining 52. The exponent is actually stored unsigned and with a bias, so zero in these bits actually indicates 2 to the -1023, 1023 would indicate 2 to the zero, and 2046 would indicate 2 to the 1023

IEEE-754 64 Bit

Special Groups



There are a few special groups of numbers. Zeros in all but the sign bit leads to a value of zero or negative zero. All ones in the exponent , followed by all zeros is infinity, and followed by anything else is a NaN. You can actually store information in the NaN values, and some systems go ahead and do this for representing different kinds of errors, though not Java nor JavaScript. Finally, a zero exponent gives you the subnormal numbers, which is all the numbers smaller than 2 to the -1023

JavaScript Integers

- 12345 => 11000000111001_2

JavaScript integers are just floating point numbers, but in a known range. If we consider the number 12,345 then it has this binary representation.

* It's positive, so the first bit is zero
* Turning this into scientific notation just means adding a point after the first digit and the exponent just says how long the remaining binary number is
* Then we throw away the leading 1
* And that's the number we write into the mantissa. The rest will be left as zero
* Then we add 13 to the bias to get 1036
* and that's the exponential
* The 52 bits of mantissa mean that we can represent up to 2 to the 53 minus one as an integer, either positive or negative, and that's the "SAFE" integer range for JavaScript. Anything with a larger absolute value can still look like an integer, but there aren't enough bits for it, so it will get rounded to the nearest value that it can represent.

JavaScript Integers

- 12345 => 11000000111001_2

JavaScript Integers

- 12345 => 11000000111001_2
- $1.1000000111001_2 \times 2^{13}$

JavaScript Integers

- 12345 => 11000000111001_2
- $1.1000000111001_2 \times 2^{13}$

JavaScript Integers

- 12345 => 11000000111001_2
- $1.1000000111001_2 \times 2^{13}$

JavaScript Integers

- 12345 => 11000000111001_2
- $1.1000000111001_2 \times 2^{13}$
- 13 + EXP-BIAS = 13 + 1023 = 1036 = 10000001100_2

JavaScript Integers

- 12345 => 11000000111001_2
- $1.1000000111001_2 \times 2^{13}$
- 13 + EXP-BIAS = 13 + 1023 = 1036 = 10000001100_2

JavaScript Integers

- 12345 => 11000000111001_2
- $1.1000000111001_2 \times 2^{13}$
- 13 + EXP-BIAS = 13 + 1023 = 1036 = 10000001100_2

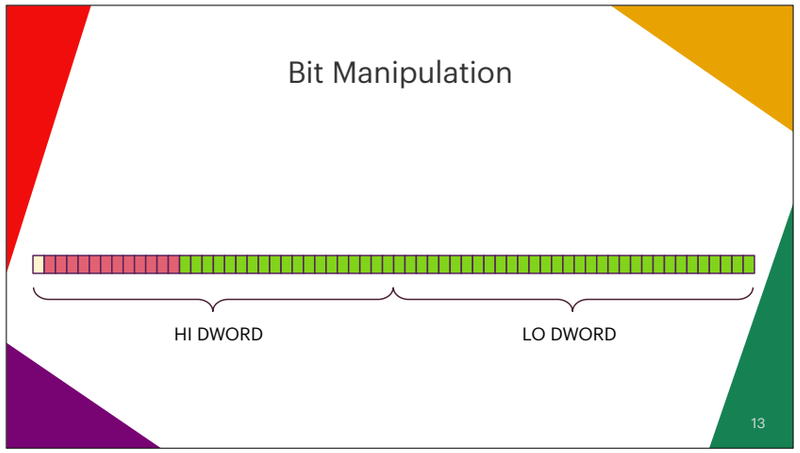
MAX_SAFE_INTEGER $= 2^{52+1} - 1 = 2^{53} - 1 =$ 9007199254740991

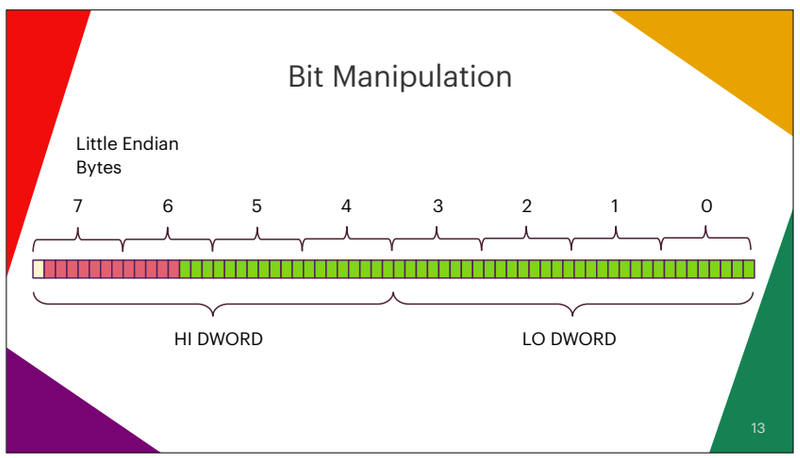
MIN_SAFE_INTEGER $=$ -9007199254740991

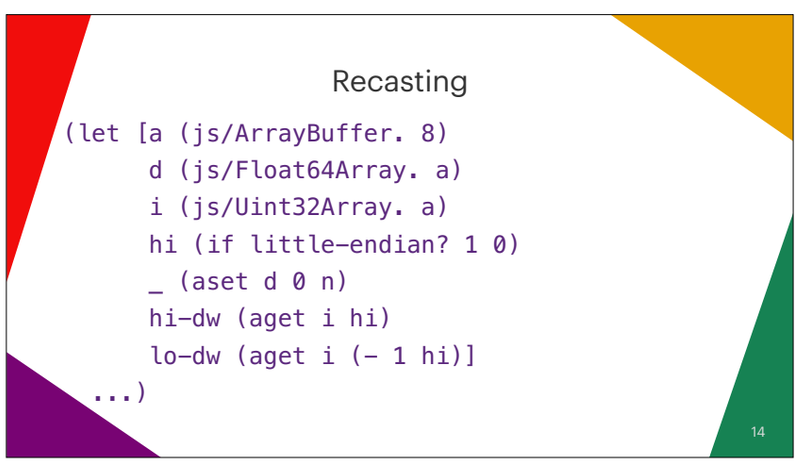
Bit Manipulation

How could we do bit manipulation on this?

* Well, to work on it, we'll want to be able to slice out the high and low double-words, or 32 bit slices.
* We may also need to work on the 8 individual bytes that make it up. When we're using little endian systems, then these get counted from the lest significant digit to the most.







The trick to doing this in JavaScript is with ArrayBuffers.

* First, we allocate 8 bytes to hold the number
* Then we put a pair of façades over it. One to see it as a one element 64bit floating point number array, and the other as a 2 element 32bit unsigned int array
* And we use the little-endian flag to figure out where the high word is
* Now we can put the floating point number into the buffer
* and read it out in the high and low parts
* On Slack, Mike Fikes expressed concern about this mechanism not being supported on older browsers. Eventually, David Nolan came in on it, asking if it could be done another way. We spoke for a while on this. But in the end he agreed that there really isn't any other good way to do this in JavaScript

Recasting

```
(let [a (js/ArrayBuffer. 8)
      d (js/Float64Array. a)
      i (js/Uint32Array. a)
      hi (if little-endian? 1 0)
      _ (aset d 0 n)
      hi-dw (aget i hi)
      lo-dw (aget i (- 1 hi)]
  ...)
```


Recasting

```
(let [a (js/ArrayBuffer. 8)
      d (js/Float64Array. a)
      i (js/Uint32Array. a)
      hi (if little-endian? 1 0)
      _ (aset d 0 n)
      hi-dw (aget i hi)
      lo-dw (aget i (- 1 hi)]
  ...)
```


Recasting

```
(let [a (js/ArrayBuffer. 8)
      d (js/Float64Array. a)
      i (js/Uint32Array. a)
      hi (if little-endian? 1 0)
      _ (aset d 0 n)
      hi-dw (aget i hi)
      lo-dw (aget i (- 1 hi)]
  ...)
```

Recasting

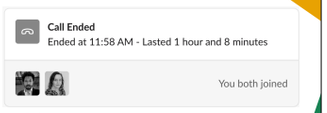
```
(let [a (js/ArrayBuffer. 8)
      d (js/Float64Array. a)
      i (js/Uint32Array. a)
      hi (if little-endian? 1 0)
      _ (aset d 0 n)
      hi-dw (aget i hi)
      lo-dw (aget i (- 1 hi)]
  ...)
```


Recasting

```
(let [a (js/ArrayBuffer. 8)
      d (js/Float64Array. a)
      i (js/Uint32Array. a)
      hi (if little-endian? 1 0)
      _ (aset d 0 n)
      hi-dw (aget i hi)
      lo-dw (aget i (- 1 hi)]
  ...)
```


Recasting

```
(let [a (js/ArrayBuffer. 8)
      d (js/Float64Array. a)
      i (js/Uint32Array. a)
      hi (if little-endian? 1 0)
      _ (aset d 0 n)
      hi-dw (aget i hi)
      lo-dw (aget i (- 1 hi)]
  ...)
```



Bit Operations

- `bit-and, bit-or`
- `bit-shift-left, unsigned-bit-shift-right`

Once we have the high and low words, we can use the typical operators to do bit manipulation. This sounds great, except these are supposed to work on integers, but JavaScript has floating point numbers. How does that work?

* It turns out that JavaScript implicitly converts the number into a signed 32 bit value. Anything larger than that is thrown away. What does that look like?
* Well, let's say I do an "or" with zero. I should get the same number back. And for a number that stays within 31 bits, I do
* But if it uses that top 32nd bit, then suddenly it switches to negative
* And if it's more than 32 bits, it truncates it. This is a mess. But now we know what to look for

Bit Operations

- `bit-and, bit-or`
- `bit-shift-left, unsigned-bit-shift-right`
- Implicit conversion to signed 32-bit

Bit Operations

- `bit-and, bit-or`
- `bit-shift-left, unsigned-bit-shift-right`
- Implicit conversion to signed 32-bit

```
=> (bit-or 1258160128 0)      0x4afe0000
1258160128
```

Bit Operations

- `bit-and`, `bit-or`
- `bit-shift-left`, `unsigned-bit-shift-right`
- Implicit conversion to signed 32-bit

```
=> (bit-or 1258160128 0)      0x4afe0000
1258160128
=> (bit-or 3405643776 0)      0xcafe0000
-889323520
```


Bit Operations

- `bit-and`, `bit-or`
- `bit-shift-left`, `unsigned-bit-shift-right`
- Implicit conversion to signed 32-bit

```
=> (bit-or 1258160128 0)      0x4afe0000
1258160128
=> (bit-or 3405643776 0)      0xcafe0000
-889323520
```

```
=> (bit-or 5553127424 0)      0x14afe0000
1258160128
```


Implementation
Simple Wrapper

```clojure
(defn ^number sin
  {:doc "Returns the sine of an angle.
  If a is ##NaN, ##-Inf, ##Inf => ##NaN
  If a is zero => zero with the same sign as a
  See: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Math/sin"
   :added "1.11.10"}
  [a] (Math/sin a))
```


OK, so let's look at the functions. Many functions are simple, and just like the JVM Clojure version, we just need a simple wrapper

Implementation
Range Checking

```
(defn ^number round
  {:doc "Returns the closest long to a. If equally close to two values, return the one
  closer to ##Inf.
  If a is ##NaN => 0
  If a is ##-Inf => js/Number.MIN_SAFE_INTEGER
  If a is ##Inf => js/Number.MAX_SAFE_INTEGER
  See: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Math/round"
   :added "1.11.10"}
  [a]
  (cond
    ^boolean (js/isNaN a) 0
    ^boolean (js/isFinite a) (Math/round a)
    (== ##Inf a) js/Number.MAX_SAFE_INTEGER
    :default js/Number.MIN_SAFE_INTEGER))
```


Others need to check their range so that they behave the same way as the Java version does for the boundary conditions. But otherwise, it's mostly a simple wrapper

---

Implementation
Simple Reimplementation

```
(defn ^number floor-div
  {:doc "Integer division that rounds to negative infinity (as opposed to zero).
  See: https://docs.oracle.com/javase/8/docs/api/java/lang/Math.html#floorDiv-long-long-"
   :added "1.11.10"}
  [x y]
  (if-not (and ^boolean (js/Number.isSafeInteger x) ^boolean (js/Number.isSafeInteger y))
    (throw (ex-info "floor-div called with non-safe-integer arguments"
                    {:x-int? (js/Number.isSafeInteger x) :y-int? (js/Number.isSafeInteger y)}))
    (let [r (long (/ x y))]
      (if (and (xor (< x 0) (< y 0)) (not (== (* r y) x)))
        (dec r)
        r))))
```


Then there are a number of functions that don't have an equivalent in JavaScript math, but are a straightforward reimplementation, like floor division here. I got through most of these functions in a day or two

---

Bit Operations

```
(defn ^number ulp
  {:doc "Returns the size of an ulp (unit in last place) for d.
  If d is ##NaN => ##NaN
  If d is ##Inf or ##-Inf => ##Inf
  If d is zero => Number/MIN_VALUE
  If d is +/- Number/MAX_VALUE => 2^971
  See: https://docs.oracle.com/javase/8/docs/api/java/lang/Math.html#ulp-double-"
   :added "1.11.10"}
  [d]
  (cond
    ^boolean (js/isNaN d) d
    ^boolean (js/isFinite d)
    (let [e (get-exponent d)]
      (case e
        1024 (Math/abs d)  ;; EXP-MAX + 1
        -1023 js/Number.MIN_VALUE  ;; EXP-MIN - 1
        (let [e (- e (+ 31 SIGNIFICAND-WIDTH32))]  ;; SIGNIFICAND_WIDTH64 -1
          (if (>= e EXP-MIN)
            (power-of-two e)
            (let [shift (- e (- EXP-MIN 31 SIGNIFICAND-WIDTH32))]
              (if (< shift 32)
                (hi-lo->double 0 (bit-shift-left 1 shift))
                (hi-lo->double (bit-shift-left 1 (- shift 32)) 0)))))))
```


Next came the functions that took more work. For instance, ulp meant I needed to understand the spec. I could even peek at other non GPL systems that implement this. There aren't too many of these, and I got them done in a day or two

... and then

java.lang.Math.IEEEremainder

(10)

…and then I hit IEEE remainder

I looked at what this function does. I looked at how others have implemented it. It's… a lot

* I asked myself if I really needed this? I mean, how many people really need that level of accuracy?
* But I promised that I could do it

... and then

java.lang.Math.IEEEremainder

Do I really need this???

... and then

java.lang.Math.IEEEremainder

Do I really need this???

But I promised Alex...

Provenance

- Originally `fdlibm` from SunSoft (Sun Microsystems) ©1993
  Hosted at the Netlib Math Repository at UTK and ORNL
- Also in `glibc`
- JDK move to native Java starting with pow in 2015 *(ongoing)*
- November 2021: `src/java.base/share/native/libfdlibm/e_remainder.c`
- March 2023: `src/java.base/share/classes/java/lang/FdLibm.java`

Taking a step back, where did this function come from? Originally, it showed up in fdlibm from SunSoft, back in 93. You can find it still in the Netlib Math repository at University of Tennessee, Knoxville, and Oak Ridge National Labs. This is a BSD style attribution license

From there, it made its way into glibc, and then Java copied it almost verbatim into java.lang.Math.

When I was looking at this, it was implemented in native C code and accessed over JNI, but last year it was moved into Java code

IEEERemainder uses fmod

```
/* determine ix = ilogb(x) */
    if(hx<0x00100000) {     /* subnormal x */
        if(hx==0) {
            for (ix = -1043, i=lx; i>0; i<<=1) ix -=1;
        } else {
            for (ix = -1022,i=(hx<<11); i>0; i<<=1) ix -=1;
        }
    } else ix = (hx>>20)-1023;

/* determine iy = ilogb(y) */
    if(hy<0x00100000) {     /* subnormal y */
        if(hy==0) {
            for (iy = -1043, i=ly; i>0; i<<=1) iy -=1;
        } else {
            for (iy = -1022,i=(hy<<11); i>0; i<<=1) iy -=1;
        }
    } else iy = (hy>>20)-1023;
```


I looked at the code, and saw immediately that the bulk of the work is done in the fmod function. That's not actually exposed in Java, but glibc has it. The is a snippet of what I was looking at. You can see in the comments that it's figuring out the binary log of a number, and handling subnormal numbers, which show up a lot when looking for this level of accuracy

ClojureScript

```
(defn ^number ilogb
  {:doc "internal function for ilogb(x)"
   :private true}
  [hx lx]
  (if (< hx 0x00100000) ;; subnormal
    (let [hx-zero? (zero? hx)
          start-ix (if hx-zero? -1043 -1022)
          start-i (if hx-zero? lx (bit-shift-left hx 11))]
      (loop [ix start-ix i start-i]
        (if-not (> i 0)
          ix
          (recur (dec ix) (bit-shift-left i 1)))))
    (- (bit-shift-right hx 20) 1023)))
```


Attempting to do the same thing in ClojureScript, I pulled it out into its own function. What we're seeing here is a direct transliteration

IEEEremainder

```c
/* fix point fmod */
    n = ix - iy;
    while(n--) {
        hz=hx-hy;lz=lx-ly; if(lx<ly) hz -= 1;
        if(hz<0){hx = hx+hx+(lx>>31); lx = lx+lx;}
        else {
            if((hz|lz)==0)          /* return sign(x)*0 */
                return Zero[(unsigned)sx>>31];
            hx = hz+hz+(lz>>31); lx = lz+lz;
        }
    }
    hz=hx-hy;lz=lx-ly; if(lx<ly) hz -= 1;
    if(hz>=0) {hx=hz;lx=lz;}
```

24

Back in the remainder function in the C code, and we're heavily into bit and math operations.

* But see these comparisons? Remember them for a moment

IEEEremainder

```c
/* fix point fmod */
    n = ix - iy;
    while(n--) {
        hz=hx-hy;lz=lx-ly; if(lx<ly) hz -= 1;
        if(hz<0){hx = hx+hx+(lx>>31); lx = lx+lx;}
        else {
            if((hz|lz)==0)          /* return sign(x)*0 */
                return Zero[(unsigned)sx>>31];
            hx = hz+hz+(lz>>31); lx = lz+lz;
        }
    }
    hz=hx-hy;lz=lx-ly; if(lx<ly) hz -= 1;
    if(hz>=0) {hx=hz;lx=lz;}
```

24

ClojureScript

```clojure
;; fix point fmod
[hx lx] (loop [n (- ix iy) hx hx lx lx]
          (if (zero? n)
            [hx lx]
            (let [hz (if (u< lx ly) (- hx hy 1) (- hx hy))
                  lz (- lx ly)
                  [hx lx] (if (< hz 0)
                            [(+ hx hx (unsigned-bit-shift-right lx 31)) (+ lx lx)]
                            (if (zero? (bit-or hz lz))
                              (throw (ex-info "Signed zero" {:zero true}))
                              [(+ hz hz (unsigned-bit-shift-right lz 31)) (+ lz lz)]))]
              (recur (dec n) (bit-and INT32-MASK hx) (bit-and INT32-MASK lx)))))
hz (if (u< lx ly) (- hx hy 1) (- hx hy))
lz (- lx ly)
[hx lx] (if (>= hz 0) [hz lz] [hx lx])
```

25

Converting the code into ClojureScript looked like this. It's so much fun. And it looked like it worked. Until randomly… it didn't. Remember those comparisons earlier? They were on the low order dwords, which are the bottom 32 bits that can suddenly tip negative.

* So the comparisons had to be updated to an unsigned less than

ClojureScript

```
;; fix point fmod
[hx lx] (loop [n (- ix iy) hx hx lx lx]
          (if (zero? n)
            [hx lx]
            (let [hz (if (u< lx ly) (- hx hy 1) (- hx hy))
                  lz (- lx ly)
                  [hx lx] (if (< hz 0)
                            [(+ hx hx (unsigned-bit-shift-right lx 31)) (+ lx lx)]
                            (if (zero? (bit-or hz lz))
                              (throw (ex-info "Signed zero" {:zero true}))
                              [(+ hz hz (unsigned-bit-shift-right lz 31)) (+ lz lz)]))]
              (recur (dec n) (bit-and INT32-MASK hx) (bit-and INT32-MASK lx)))))
hz (if (u< lx ly) (- hx hy 1) (- hx hy))
lz (- lx ly)
[hx lx] (if (>= hz 0) [hz lz] [hx lx])
```


JS Comparisons

```
(defn u<
  {:doc "unsigned less-than comparator for 32-bit values"
   :private true}
  [a b]
  ;; compare the top nybble
  (let [ab (unsigned-bit-shift-right a 28)
        bb (unsigned-bit-shift-right b 28)]
    (or (< ab bb)  ;; if the top nybble of a is less then
                   ;; the whole value is less
        (and (== ab bb)  ;; if the top nybble is equal then
                         ;; compare the remaining bits of both
             (< (bit-and a 0x0fffffff) (bit-and b 0x0fffffff))))))
```

Here's the code for that. It starts with the top nybble. These are safe to compare if they're shifted down. And it only looks at the rest of the number if those nybbles were the same.

Finally, it was all running. I was done, right?

JVM Testing

- Run all code on the JVM
- Easy comparison to `java.lang.Math`
- Can use generative testing
- Requires stubs for `js` functions and symbols

e.g.
```
(js/isFinite js/Number.MAX_SAFE_INTEGER)
(js/isNaN js/Number.POSITIVE_INFINITY)
```


Well, it needed to be tested, and this needed to be rigorous.

The first step was to run it all in the JVM. It was Clojure code, so it should "just work", and we could compare the results to java.lang.Math directly. Even better, we can use generative testing so we can't hide our indiscretions.

But of course, some of the code doesn't just work. I was making references to several js entities

JVM Testing Stubs

```clojure
(ns ^{:doc "Duplicate some of the functionality of ClojureScript for testing purposes"
      :author "Paula Gearon"}
    js
    (:refer-clojure :exclude [aset aget +])
    (:import [java.nio ByteBuffer IntBuffer DoubleBuffer]))

(def ^:const Number.MAX_SAFE_INTEGER  9007199254740991)
(def ^:const Number.MIN_SAFE_INTEGER -9007199254740991)
(def ^:const Number.POSITIVE_INFINITY ##Inf)
(def ^:const Number.MAX_VALUE Double/MAX_VALUE)
(def ^:const Number.MIN_VALUE Double/MIN_VALUE)

(defn isNaN [^double d] (Double/isNaN d))

(defn isFinite [^double d] (not (Double/isInfinite d)))

(defn isInteger [^double d] (int? d))

(defn Number.isSafeInteger
  [n]
  (and (>= n Number.MIN_SAFE_INTEGER)
       (<= n Number.MAX_SAFE_INTEGER)))
```

28

To make that work, I stubbed out a namespace that implemented all of the js stuff in Java. So the Java code could load it, and emulate the Javascript environment

JVM Testing Stubs

```clojure
(defprotocol ArrayAccess
  (aget [a n] "Retrieves data at position n")
  (aset [a n v] "Sets a value n at position n"))

(gen-class
 :name js.ArrayBuffer
 :state buffer
 :init init
 :constructors {[int] []}
 :prefix "array-buffer-")

(gen-class
 :name js.Uint8Array
 :state buffer
 :init init
 :constructors {[js.ArrayBuffer] []}
 :prefix "uint8-")

(gen-class
 :name js.Uint32Array
 :state buffer
 :init init
 :constructors {[js.ArrayBuffer] []}
 :prefix "uint32-")

(gen-class
 :name js.Float64Array
 :state buffer
 :init init
 :constructors {[js.ArrayBuffer] []}
 :prefix "float64-")
```

29

The ArrayBuffer code was handled as a set of classes and a protocol for accessing them as arrays

JVM Testing Stubs

Constructors

```clojure
(defn array-buffer-init [s] [[] (ByteBuffer/allocate s)])

(defn uint8-init [a] [[] (.buffer a)])

(defn uint32-init [a] [[] (.asIntBuffer (.buffer a))])

(defn float64-init [a] [[] (.asDoubleBuffer (.buffer a))])
```

Override

clojure.core/aget

clojure.core/aset

```clojure
(extend-protocol ArrayAccess
  js.Uint8Array
  (aget [a n] (.get ^ByteBuffer (.buffer a) ^int n))
  (aset [a n v]
    (let [v (bit-and 0xff v)
          v (byte (if (zero? (bit-and 0x80 v)) v (bit-or -0x100 v)))]
      (.put ^ByteBuffer (.buffer a) ^int n v)))

  js.Uint32Array
  (aget [a n] (.get ^IntBuffer (.buffer a) n))
  (aset [a n v]
    (let [v (bit-and 0xffffffff v)
          v (int (if (zero? (bit-and 0x80000000 v)) v (bit-or -0x100000000 v)))]
      (.put ^IntBuffer (.buffer a) ^int n v)))

  js.Float64Array
  (aget [a n]
    (.get ^DoubleBuffer (.buffer a) (int (as-int n))))
  (aset [a n v] (.put ^DoubleBuffer (.buffer a) ^int n ^double v)))
```

30

The constructors used java ByteBuffers, with appropriate views to make them appear as Integers or doubles, and then the protocol implementation handled the setting and getting from these virtual arrays. The only trick was making everything going in or out appear unsigned.

Generative Testing

```
(defn d= [a b] (or (= a b) (and (Double/isNaN a) (Double/isNaN b))))

(def safe-integer (gen/choose js/Number.MIN_SAFE_INTEGER js/Number.MAX_SAFE_INTEGER))

;; The following test functions that are wrappers for functionality provided in Math
(defspec sin-test 100
  (prop/for-all [v gen/double]
    (d= (m/sin v) (Math/sin v))))

(defspec cos-test 100
  (prop/for-all [v gen/double]
    (d= (m/cos v) (Math/cos v))))
```

31

With these in place, we could do some generative testing. Some of the comparisons needed updating, since Not-a-numbers normally compare as not equal, which isn't what I needed. But with that in place, we could define the ranges of numbers being tested, and ask if our code returned the same things the Java library returned.

Generative Testing

```
;; The following test functions that have been implemented by cljs-math
(defspec to-radians-test 1000
  (prop/for-all [v gen/double]
    (d= (m/to-radians v) (Math/toRadians v))))

(defspec to-degrees-test 1000
  (prop/for-all [v gen/double]
    (d= (m/to-degrees v) (Math/toDegrees v))))

(defspec ieee-remainder-test 1000
  (prop/for-all [dividend gen/double divisor gen/double]
    (d= (m/IEEE-remainder dividend divisor) (Math/IEEEremainder dividend divisor))))
```

32

I can't tell you just how excited I was when ieee-remainder passed this. I was giddy

---

Testing

Problems!

```
(Math/floorMod 1 -2147483691)  →  1
```

33

But it wasn't all smooth sailing. Some things returned weird values. For instance, this value showed up in generative testing, and should return 1.

* but instead it returned this. I asked in Clojurians, and people like Adrian Smith immediately confirmed what I was seeing
* It turns out that the reflection code will pick up the wrong method overload, and cast arguments to 32 bit ints.
* It ended up being easy to work around just y casting. This was a good reminder that I wasn't being careful about reflection.

Testing

Problems!

```
(Math/floorMod 1 -2147483691)  → 1
  → -2147483690
```


Testing

Problems!

```
(Math/floorMod 1 -2147483691)  → 1
  → -2147483690
```

CLJ-2677:
clojure math methods resolve to single overload in fn bodies and should also resolve to single overload in inlines

Testing

Problems!

```
(Math/floorMod 1 -2147483691)  → 1
  → -2147483690
```

CLJ-2677:
clojure math methods resolve to single overload in fn bodies and should also resolve to single overload in inlines

Work around:

```
(Math/floorMod 1 ^long -2147483691)
```

Testing

Still an issue with interop in 1.12.0

```
(doseq [x (range -2147482742 -2147484000 -110)]
  (when (= 1 (Math/floorMod 1 x)) (println x)))
```


We do still see it in Clojure today. This bit of code will print whenever it get a bad result.

* These errors are what we get
* But if we cast it, then nothing comes out
* This is actually one of the reasons why clojure.math is a good idea, since it ensures that we're not accidentally reflecting

Testing

Still an issue with interop in 1.12.0

```
(doseq [x (range -2147482742 -2147484000 -110)]
  (when (= 1 (Math/floorMod 1 x)) (println x)))
-2147483732
-2147483842
-2147483952
nil
```


Testing

Still an issue with interop in 1.12.0

```
(doseq [x (range -2147482742 -2147484000 -110)]
  (when (= 1 (Math/floorMod 1 ^long x)) (println x)))
nil
```

Testing

Still an issue with interop in 1.12.0

```
(doseq [x (range -2147482742 -2147484000 -110)]
  (when (= 1 (Math/floorMod 1 ^long x)) (println x)))
nil
```

one of many reasons

This is ^ why clojure.math exists

Alex checked in with me around that time, and I was happy to assure him that it looked good. Now I needed to get it into ClojureScript proper

clojure.java.math → clojure.math

Alex Miller (Clojure team) Nov 30th, 2021 at 11:39 AM
if you have a confidence level that cljs can support all (or nearly all) the math functions, that would be useful to know. doesn't have to be done, just looking for whether you believe it can be

3 replies

quoll
Yes, I'm confident.
I have implementations for almost everything now (not all tested). The most complex ones are tractable

quoll
They rely on bit operations which I've been able to completely duplicate in ClojureScript

Alex Miller (Clojure team)
Thanks

Mike Fikes had been following along with my public discussions on Slack, and then suddenly came back with a whole lot of comments. After the highs of the IEEERemainder success, this brought me back to earth. The suggestions themselves were mostly for performance and were all excellent. He was more than happy to help me out at every step, and knew so much about the ClojureScript system that I'd never considered.

Mike Fikes

mfikes 8:34 PM
Comments on the math namespace

- All of the math wrappers should be hinted as returning numbers, for example: `(defn ^number abs [x] (Math/abs x))`
- Maybe name things like `DEGREES_TO_RADIANS` instead as `DEGREES-TO-RADIANS`
- Maybe upper-case the symbol `two-to-the-52`
- Expressions like `(js/Number.isNaN x)` and `(js/Number.isFinite y)` can be written instead as `(js/isNaN x)` and `(js/isFinite y)` (This is done in `cljs.core` and might actually be faster.)
- When using primitive JavaScript predicates like `js/isNaN` unchecked `if` can be enabled by throwing a `^boolean` typehint in there. Example: `(if ^boolean (js/isNan x) a b)`
- Line 394 has the odd expression `(* 0.0 dividend)`. Maybe this is simply part of the port, but it seems like it could be replaced with `0`.
- Maybe `(if a (Math/ceil a) (throw ,,,))` should instead be `(if (some? a) ...)` (During compilation, the call to `some?` gets converted to a fast JavaScript check against `null`.)
- I wonder if the expression `(or (> r js/Number.MAX_SAFE_INTEGER) (< r js/Number.MIN_SAFE_INTEGER))` can be shortened to `(not ^boolean (js/Number.isSafeInteger r))`
- Perhaps the bodies of `increment-exact` and `decrement-exact` should make use of `inc` and `dec`.
- Arguments to `xor` could be hinted as in `[^boolean a ^boolean b]`. If you do this you get nice compact JavaScript: `((((a) && ((!(b)))) || ((((!(a))) && (b))))` as opposed to a bunch of `if`-based JavaScript
- Maybe `(js/Number.isInteger x)` could be replaced with `(integer? x)` in some places. If not, throw a `^boolean` hint on the expression.
- `not=` is used. I think for numerics `(not (== x y))` generates more compact and more efficient JavaScript
- `signum` uses the expression `(= 0.0 d)` which could be replaced with `(zero? d)` (The generated JavaScript is fast.) (There is a similar `(= 0 (aget b 0))` at the end of `get-little-endian`.

Mike Fikes

mfikes 8:42 PM

`(set! *print-fn-bodies* true)`

Then `(defn xor [a b] (or (and a (not b)) (and (not a) b)))`

Then if you evaluate `xor` you will see

```
#object[cljs$user$xor "function cljs$user$xor(a,b){
var or__10116__auto__ = (function (){var and__10091__auto__ = a;
if(cljs.core.truth_(and__10091__auto__)){
return cljs.core.not.call(null,b);
} else {
return and__10091__auto__;
}
})();
if(cljs.core.truth_(or__10116__auto__)){
return or__10116__auto__;
} else {
if(cljs.core.not.call(null,a)){
return b;
} else {
return false;
}
}
}"]
```

37

For instance, setting `print-fn-bodies` lets you see the raw javascript that the ClojureScript compiler generates, so you can see which idioms are more efficient. But this was just the start. He had so much more to offer. I can't tell you how much I appreciated his help on this.

Making it Official

December 10th, 2021

quoll 2:39 PM

I have cljs-math at a point where I would like to propose it for inclusion with ClojureScript.
I'm expecting that there may be requests to modify it (e.g. changing the namespace to `clojure.math`, and maybe strip out the Clojure cross compilation), but the functionality is there, and it's 100% ClojureScript (I showed @dnolen a version that embedded JavaScript, but that's gone now).
What process should I go through to make this request? Just add it as a PR to https://github.com/clojure/clojurescript and go from there? Something else?

> **clojure/clojurescript**
> Clojure to JS compiler
> **Stars** 8776 **Language** Clojure
> Added by GitHub

🎉 4 👍 1

38

On December 10, I was feeling confident enough that I made a formal request on Slack about how to include it in ClojureScript. It turns out that most of that is documented. Here I was reading formal specs, but forgetting to go to obvious places like clojurescript.org. But it did inform people what I was doing.

JavaScript Testing

- Can generative testing compare JVM to JavaScript?

39

But now I was committed and it had to be really robust. I put in basic tests, but if this was in the core language I couldn't afford to let problems through.

I'd really appreciated how I could run the code on the JVM and compare it to the system library code. But could I do something between the JVM and JavaScript?

* Step one was to create an nrepl in the JVM. But you can only really talk to that via strings.
* So let's do the generative testing using strings!

JavaScript Testing

- Can generative testing compare JVM to JavaScript?
- Create an nrepl in a JVM

JavaScript Testing

- Can generative testing compare JVM to JavaScript?
- Create an nrepl in a JVM
- Generative testing via strings

Generative Testing

```
(defn n==
  [a b]
  (or (and (Double/isNaN a) (Double/isNaN b))
      (and (number? a) (number? b) (== a b))
      (= a b)))


(deftest get-exponent-test
  (let [ds (gen/sample gen/double 100)]
       (is (every? identity
            (map n==
                 (map cljs-get-exponent ds)
                 (map get-exponent ds))))))
```


You can see here that I have a number equality function at the top. This is similar to the previous one, where NaN can also be equal. This gets used in the example test below it.

* But how do I make this cljs-get-exponent part work?

Generative Testing

```clojure
(defn n==
  [a b]
  (or (and (Double/isNaN a) (Double/isNaN b))
      (and (number? a) (number? b) (== a b))
      (= a b)))


(deftest get-exponent-test
  (let [ds (gen/sample gen/double 100)]
       (is (every? identity
            (map n==
                 (map cljs-get-exponent ds)
                 (map get-exponent ds))))))
```


I was thinking I could create a connection to a repl, probably over a pipe, and use that. But no sooner had I mentioned it online, when Michiel Borkent suddenly popped in with a solution. His uses a socket, but that's fine and I'm definitely happy about working code that I didn't need to write!

BorkDude to the Rescue!

```clojure
(def reader (atom nil))
(def writer (atom nil))

(defn cljs-eval [expr]
  (-> (binding [*out* @writer
                *in* @reader]
        (println expr)
        (read-line))
      edn/read-string
      :val))

(t/use-fixtures :once
  (fn [f]
    (println "Launching test pREPL.")
    (let [server (server/start-server {:accept 'cljs.core.server/io-prepl
                                       :address "127.0.0.1"
                                       :port 0
                                       :name "cljs.math-repl"
                                       :args [:repl-env (cljs.repl.node/repl-env)]})
          port (-> server (.getLocalPort))]
      (println "Server opened on port" port)
      (with-open [socket (java.net.Socket. "127.0.0.1" port)
                  rdr (io/reader socket)
                  wrtr (io/writer socket)]
        (reset! reader rdr)
        (reset! writer wrtr)
        (println "Executing tests")
        (cljs-eval "(require 'cljs.math)")
        (f)
        (println "Tearing down test pREPL."))))
```


Sure enough, run cljs-eval on a string, and it returns a string with the expected output

BorkDude

```clojure
(deftest sanity-test
  (is (= "6" (cljs-eval "(+ 1 2 3)"))))
```

Generative Testing

```
(deftest get-exponent-test
  (let [ds (gen/sample gen/double 100)]
       (is (every? identity
            (map n==
                 (read-string
                  (cljs-eval (str "(->> '" (pr-str ds)
                                  " (map double)"
                                  " (map cljs.math/get-exponent"))")))
                 (map #(Math/getExponent %) ds))))))
```


Now, we can generate the sample for generative testing, print it into the string expression to be read and evaluated, and compare the results to the running the same JVM function over the sequence. The two sequences are compared with the equality operator, and checked.

* This effectively ends up looking like this was evaluated, except the ClojureScript system ran the first expression while the JVM ran the second

Generative Testing

```
(deftest get-exponent-test
  (let [ds (gen/sample gen/double 100)]
       (is (every? identity
            (map n==
                 (->> '[0.000000000 -0.000000000 3.025521743245 ...
                      (map double)
                      (map cljs.math/get-exponent)

                 (map #(Math/getExponent %) ds))))))
```


Macro Club

```
(defmacro test-t->t
  [n jfn cfn gen & [equals]]
  (let [jmfn (symbol "Math" (str jfn))
        cmfn (name cfn)
        eq (or equals n==)]
    `(let [ds# (gen/sample ~gen ~n)]
       (is (every? identity
            (map ~eq
                 (read-string
                  (cljs-eval (str "(->> '" (pr-str ds#)
                                  " (map double)"
                                  " (map cljs.math/" ~cmfn "))")))
                 (map #(~jmfn %) ds#)))
           (str "data: " (pr-str ds#))))))
```


But we don't want to write that structure over and over for every function, right? So what about Macros? Well Fogus can explain why I can't tell you about macros, but this is the basic structure:

This macro runs on functions that take a single argument of a type, like a double, and return that same type.

* The first thing the generated code does is to create the sequence of samples
* Then it does what we were seeing earlier, where we emit the numbers and convert to doubles
* map the expression of interest over them, and then this expressions string can be sent to the the clojurescript repl
* Map the Java equation function over the same sample
* and finally, compare

Macro Club

```
(defmacro test-t->t
  [n jfn cfn gen & [equals]]
  (let [jmfn (symbol "Math" (str jfn))
        cmfn (name cfn)
        eq (or equals n==)]
    `(let [ds# (gen/sample ~gen ~n)]
       (is (every? identity
            (map ~eq
                 (read-string
                  (cljs-eval (str "(->> '" (pr-str ds#)
                                  " (map double)"
                                  " (map cljs.math/" ~cmfn "))")))
                 (map #(~jmfn %) ds#)))
           (str "data: " (pr-str ds#))))))
```

generate numbers

Macro Club

```
(defmacro test-t->t
  [n jfn cfn gen & [equals]]
  (let [jmfn (symbol "Math" (str jfn))
        cmfn (name cfn)
        eq (or equals n==)]
    `(let [ds# (gen/sample ~gen ~n)]
       (is (every? identity
            (map ~eq
                 (read-string
                  (cljs-eval (str "(->> '" (pr-str ds#)
                                  " (map double)"
                                  " (map cljs.math/" ~cmfn "))")))
                 (map #(~jmfn %) ds#)))
           (str "data: " (pr-str ds#))))))
```


to string then to doubles

Macro Club

```
(defmacro test-t->t
  [n jfn cfn gen & [equals]]
  (let [jmfn (symbol "Math" (str jfn))
        cmfn (name cfn)
        eq (or equals n==)]
    `(let [ds# (gen/sample ~gen ~n)]
       (is (every? identity
            (map ~eq
                 (read-string
                  (cljs-eval (str "(->> '" (pr-str ds#)
                                  " (map double)"
                                  " (map cljs.math/" ~cmfn "))")))
                 (map #(~jmfn %) ds#)))
           (str "data: " (pr-str ds#))))))
```


map the fn over the doubles

Macro Club

```
(defmacro test-t->t
  [n jfn cfn gen & [equals]]
  (let [jmfn (symbol "Math" (str jfn))
        cmfn (name cfn)
        eq (or equals n==)]
    `(let [ds# (gen/sample ~gen ~n)]
       (is (every? identity
            (map ~eq
                 (read-string
                  (cljs-eval (str "(->> '" (pr-str ds#)
                                  " (map double)"
                                  " (map cljs.math/" ~cmfn "))")))
                 (map #(~jmfn %) ds#)))
           (str "data: " (pr-str ds#))))))
```

map the same fn in the JVM

44

Macro Club

```
(defmacro test-t->t
  [n jfn cfn gen & [equals]]
  (let [jmfn (symbol "Math" (str jfn))
        cmfn (name cfn)
        eq (or equals n==)]
    `(let [ds# (gen/sample ~gen ~n)]
       (is (every? identity
            (map ~eq
                 (read-string
                  (cljs-eval (str "(->> '" (pr-str ds#)
                                  " (map double)"
                                  " (map cljs.math/" ~cmfn "))")))
                 (map #(~jmfn %) ds#)))
           (str "data: " (pr-str ds#))))))
```

compare

44

If it's your first time at Macro Club... You have to Macro

```
(defmacro test-double->double
  [n jfn cfn & [equals]]
  `(test-t->t ~n ~jfn ~cfn gen/double ~equals))

(defmacro test-double-double->double
  [n jfn cfn & [equals]]
  `(test-t-t->double ~n ~jfn ~cfn gen/double gen/double ~equals))
```

45

---

(20)

With a couple of these set up, we can build specialized macros that work for specific function types, like a function that takes a double and returns a double. Or one that takes 2 doubles and returns a double. There's a few variations in there now.

Generative Testing

```
(deftest get-exponent-test
  (test-double->double 100 getExponent get-exponent))
```

46

So then we get to write a nice, simple expressions like this, and it executes equivalent operations on the JVM and a ClojureScript repl.

* There are several variants here including comparisons that can print what's going on when values are close but not equal, or operations that can't take zeros for some arguments

Generative Testing

```
(deftest get-exponent-test
  (test-double->double 100 getExponent get-exponent))

(deftest ieee-remainder-test
  (test-double-double->double 100 IEEEremainder IEEE-remainder))

(deftest sin-test
  (test-double->double 100 sin sin #(nd== "sin()" %1 %2)))

(deftest floor-div-test
  (test-zlong-long->long 100 floorDiv floor-div))
```

46

Check In

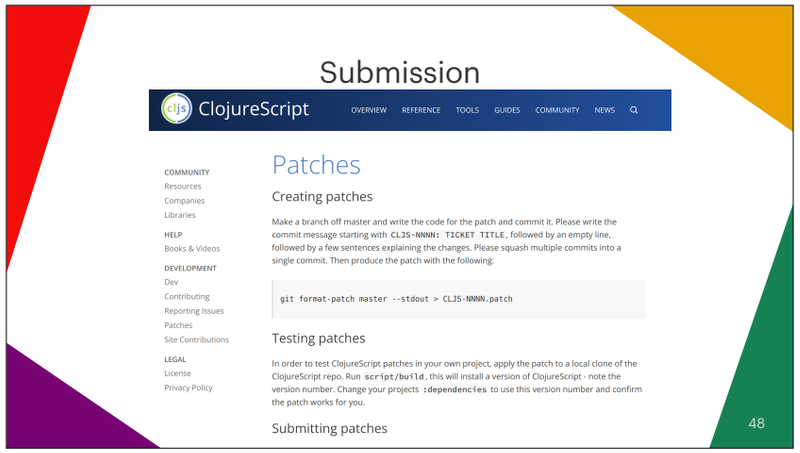
quoll 12:25 PM
Next question: I'd like to bring in the tests from https://github.com/quoll/cljs-math/blob/main/test/cljs/math/prepl_test.clj
This is a Clojure file that starts a prepl over a socket, then runs generative tests that compare cljs.math functions to java.lang.Math functions (this should eventually move to clj.math functions).
Is this appropriate to submit with the math.cljs file, and if so, then where would you like it to be, please? I was thinking src/test/clojure/cljs, but I don't want to be doing the wrong thing.

dnolen 1:19 PM
yes that's fine - that's actually ... very clever 🙂

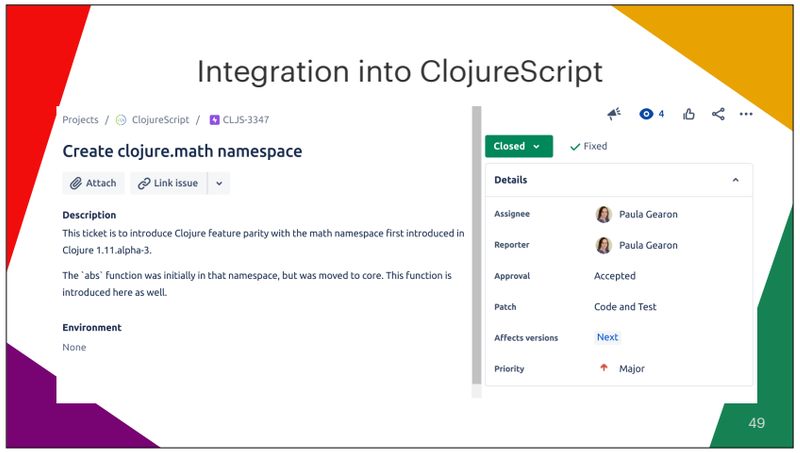
we should actually write some REPL tests using this pattern

47

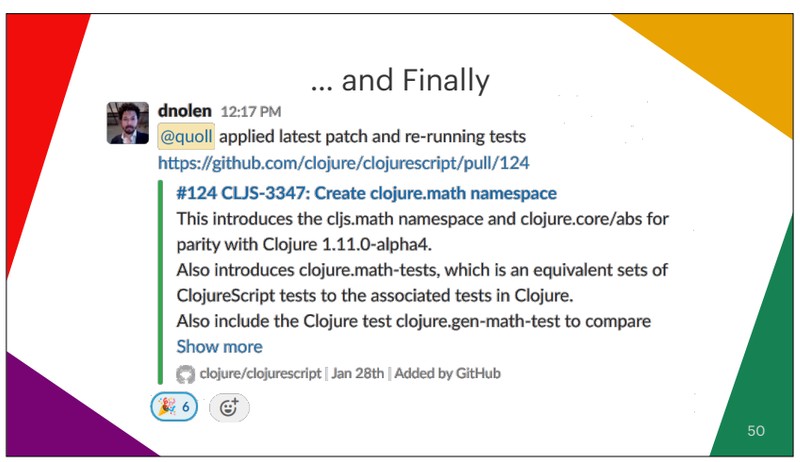
These tests made me far more confident, and so I explained what I did for David. He approved. And David's standards are so high that getting something approved by him was almost as good as getting IEEE remainder to work



So now I learned how to follow the formal patch process

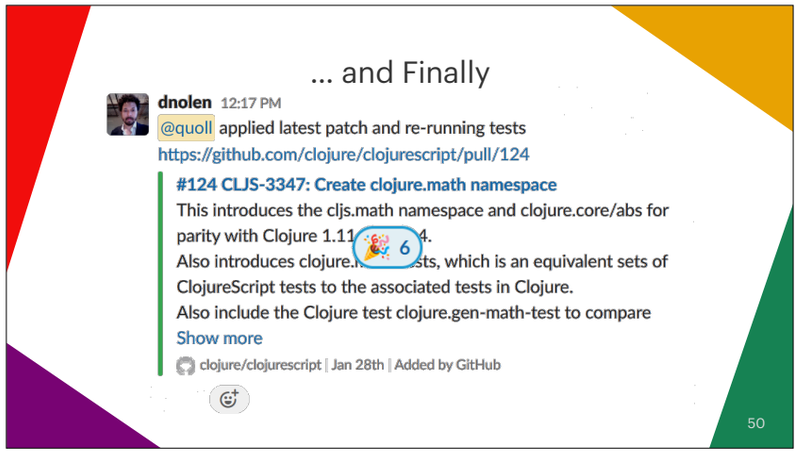


I created the appropriate Jira ticket, and attached the patch



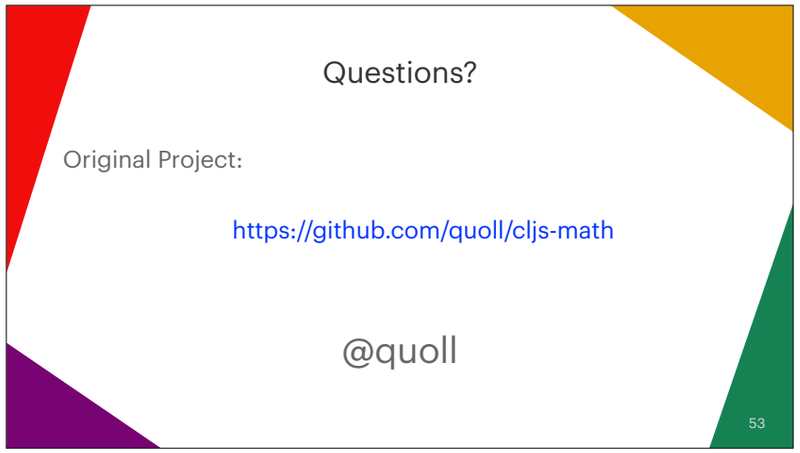
Then David said that he'd merged it

* and a handful of people liked me



But when was that release going to happen? I was SOOOO impatient. So I looked at the other new functions in Clojure 1.11 and reimplemented those for ClojureScript too. Of course, now I had a much better idea of how to set up testing, and submit the patches appropriately

Finally, 5½ months after starting down this path, it was released as an official part of ClojureScript



It's somewhat belated, but I'd like to say a special thank you to everyone who helped get this done.
Alex Miller, for being willing to hear me out
Mike Fikes for his friendship, thorough evaluations, and patient and knowledgable explanations
David Nolan for his advice and willingness to work through this with me
Michiel Borkent for saving me a couple of days of work when I really needed it
Everyone who interacted with me over Slack with comments, suggestions, or encouragement during that time, particularly in #cljs-dev