≈0x1.4p2 minute update on Java Numerics

Joseph D. Darcy (OpenJDK darcy, jddarcy, @jddarcy, @jddarcy)
Java Floating-Point Czar Emeritus
Java Platform Group, Oracle

JVM Language Summit
August 2023

Progress report for JVMLS 2023 compared to JVMLS 2017

Forward to the past: the case for uniformly strict Floating-Point Arithmetic on the JVM

Joseph D. Darcy (@jddarcy)
Java Floating-Point Czar Emeritus
Java Platform Group, Oracle

JVM Language Summit
July 31, 2017

tl;dr

- Back in the 1990's, the x86 architecture was the most popular kind of processor for desktop systems.
- When the x87 FPU was configured to round its 80-bit floating-point registers to double precision, it still used a wider exponent range.
  - Performing *$OP* in double precision and doing a store + reload to 64-bit double was *almost* always enough to get the same results as pure double
  - *But*, a narrow range of subnormal results for multiply and divide was subject to *double-rounding*, some results would end up rounding up *twice* while a direct rounding to pure double would only round up *once*.
- This small deviation was problematic for Java's specification and "write once, run anywhere" reproducibility goals.

JDK 1.2 Floating-point Update: A Pragmatic Compromise

- By default, floating-point expression were done non-strict where the small deviation was tolerated.
- Methods could be declared `strictfp` to indicate the exact expected `float`/`double` semantics were required
- Represented in the JVM with the `ACC_STRICT` method access flag

Class file → assembly, non-strict 32-bit (x87)

```
public static double
  prod(double, double); descriptor: (DD)D
  flags: (0x0009) ACC_PUBLIC, ACC_STATIC
  Code: stack=4, locals=4, args_size=2
   0: dload_0
   1: dload_2
   2: dmul
   3: dreturn
```

```
Fp.prod(DD)D  [0x02a0bb20, 0x02a0bba0]  128 bytes
  # {method} {0x15f0029c} 'prod' '(DD)D' in 'Fp'
  # parm0:    [sp+0x30]   = double  (sp of caller)
  # parm1:    [sp+0x38]   = double
  […]
  ;*dload_0 {reexecute=0 rethrow=0 return_oop=0}
  ; - Fp::prod@0 (line 13)
  fld     qword ptr [esp+38h]
  fld     qword ptr [esp+30h]
  fmulp   st(1),st(0)
  add     esp,28h
  pop     ebp
  test    dword ptr [460000h],eax
  ;   {poll_return}
  ret
```

Instruction sequence can return a result different than the strict product.

Class file → assembly, *strict* 32-bit (x87)

```
public static strictfp double
  strictProd(double, double); descriptor: (DD)D
  flags: (0x0809) ACC_PUBLIC, ACC_STATIC, ACC_STRICT
  Code: stack=4, locals=4, args_size=2
   0: dload_0
   1: dload_2
   2: dmul
   3: dreturn
```

```
Fp.strictProd(DD)D  [0x02c9c720, 0x02c9c7c0]  160 bytes
  # {method} {0x15f00304} 'strictProd' '(DD)D' in 'Fp'
  # parm0:    [sp+0x30]   = double  (sp of caller)
  # parm1:    [sp+0x38]   = double
  […]
  ; - Fp::strictProd@0 (line 17)
  fld      qword ptr [esp+38h]
  fld      qword ptr [esp+30h]
  fld      tbyte ptr [58711c94h]  ;   1.0 × 2^-15360
  fmulp    st(1),st(0)
  fmul     st(0),st(1)
  fld      tbyte ptr [58711ca0h]  ;   1.0 × 2^+15360
  fmulp    st(1),st(0)
  fstp     qword ptr [esp+18h]
  fld      qword ptr [esp+18h]
  […]
```

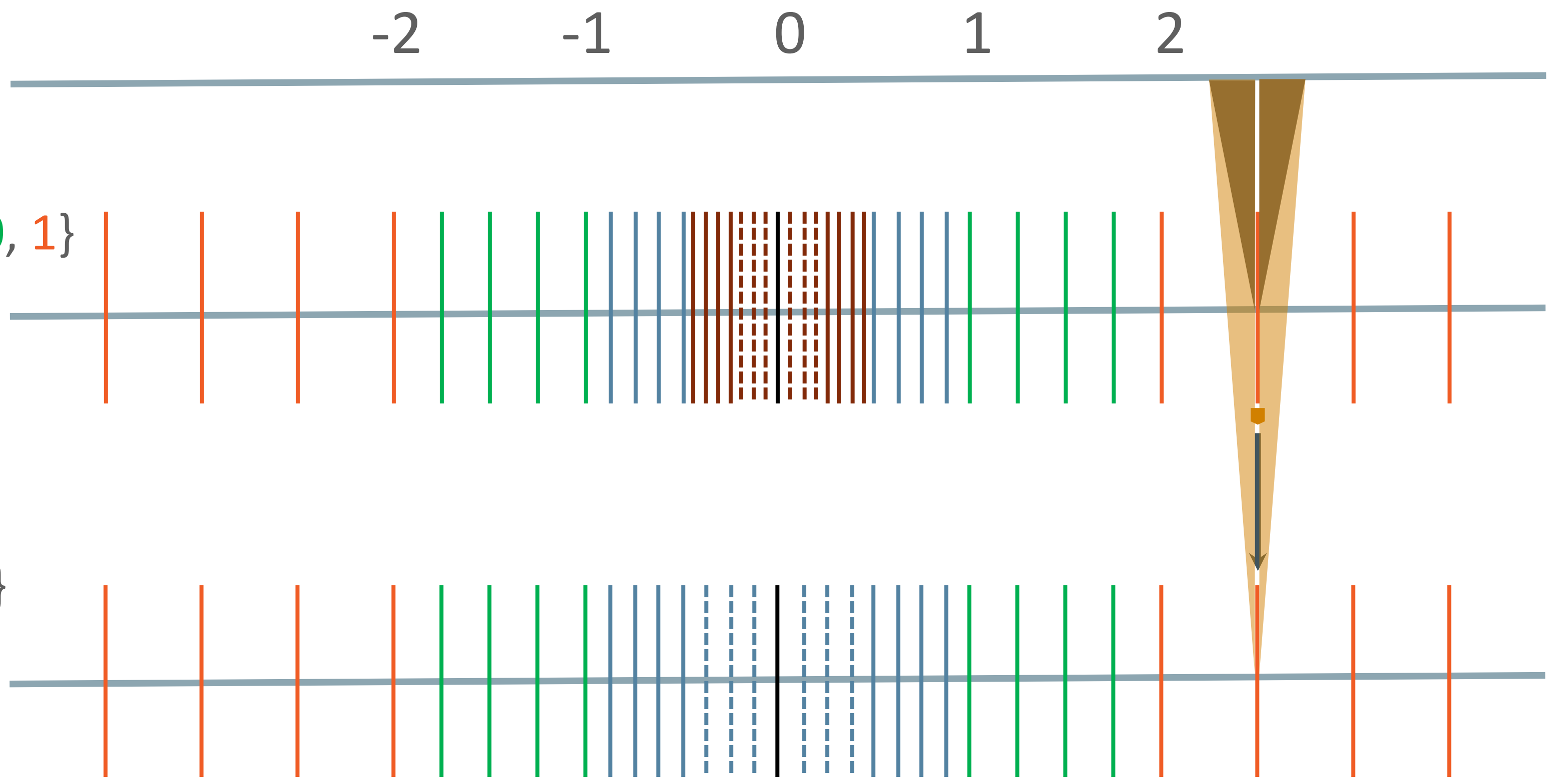
???

Case analysis: double rounding in the normal range

Toy example

p = 3,
exponent $\in$ {-2, -1, 0, 1}

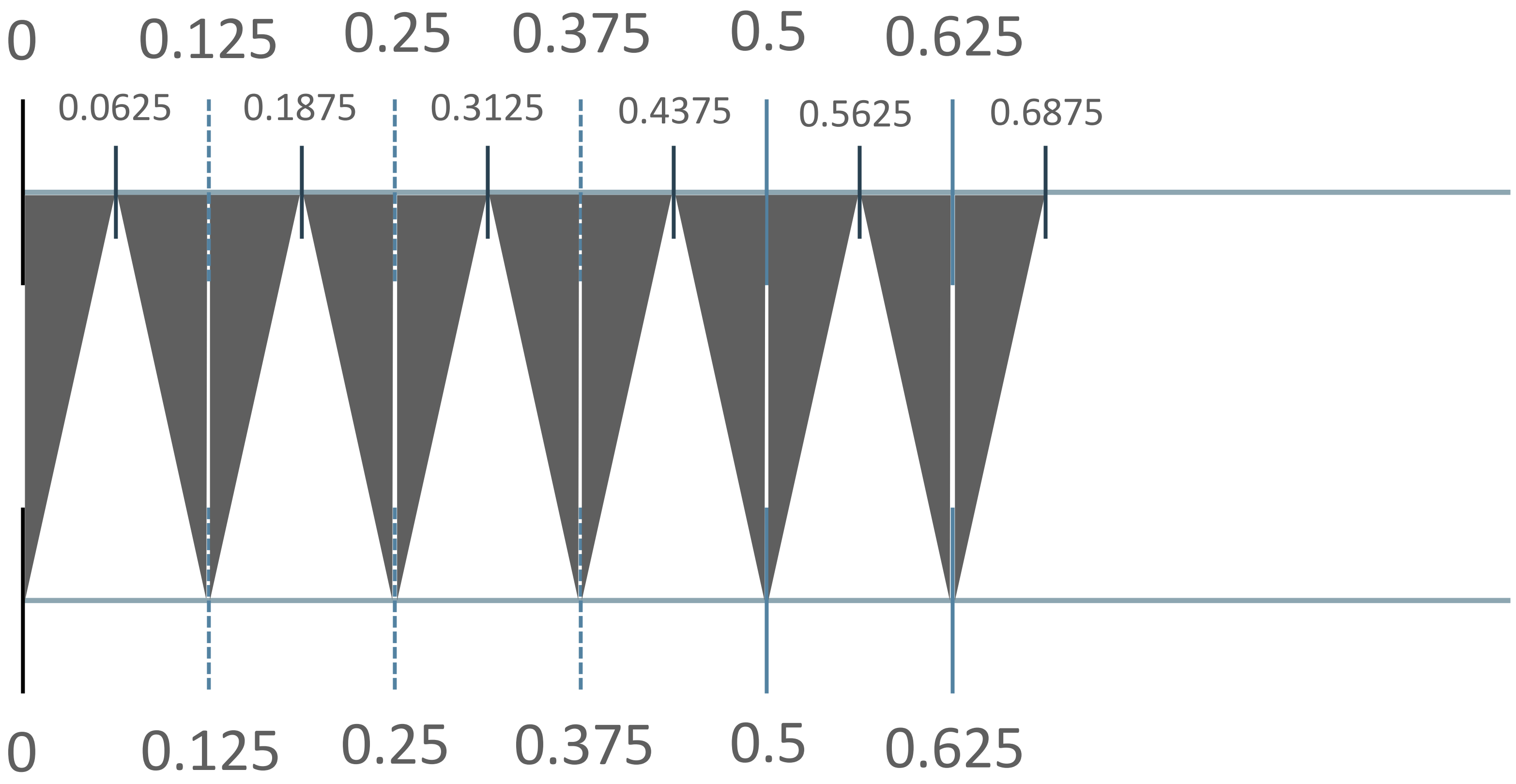
p = 3,
exponent $\in$ {-1, 0, 1}



Special values:
±0.0
NaN
±∞
Overflow

Rounding near minimum normal value of toy format

Representable values are evenly spaced



Toy format with p=3, $Exp_{min} = -1$

0.0 $= 0.00_2 \cdot 2^{-1}$

0.125 $= 0.01_2 \cdot 2^{-1}$

0.25 $= 0.10_2 \cdot 2^{-1}$

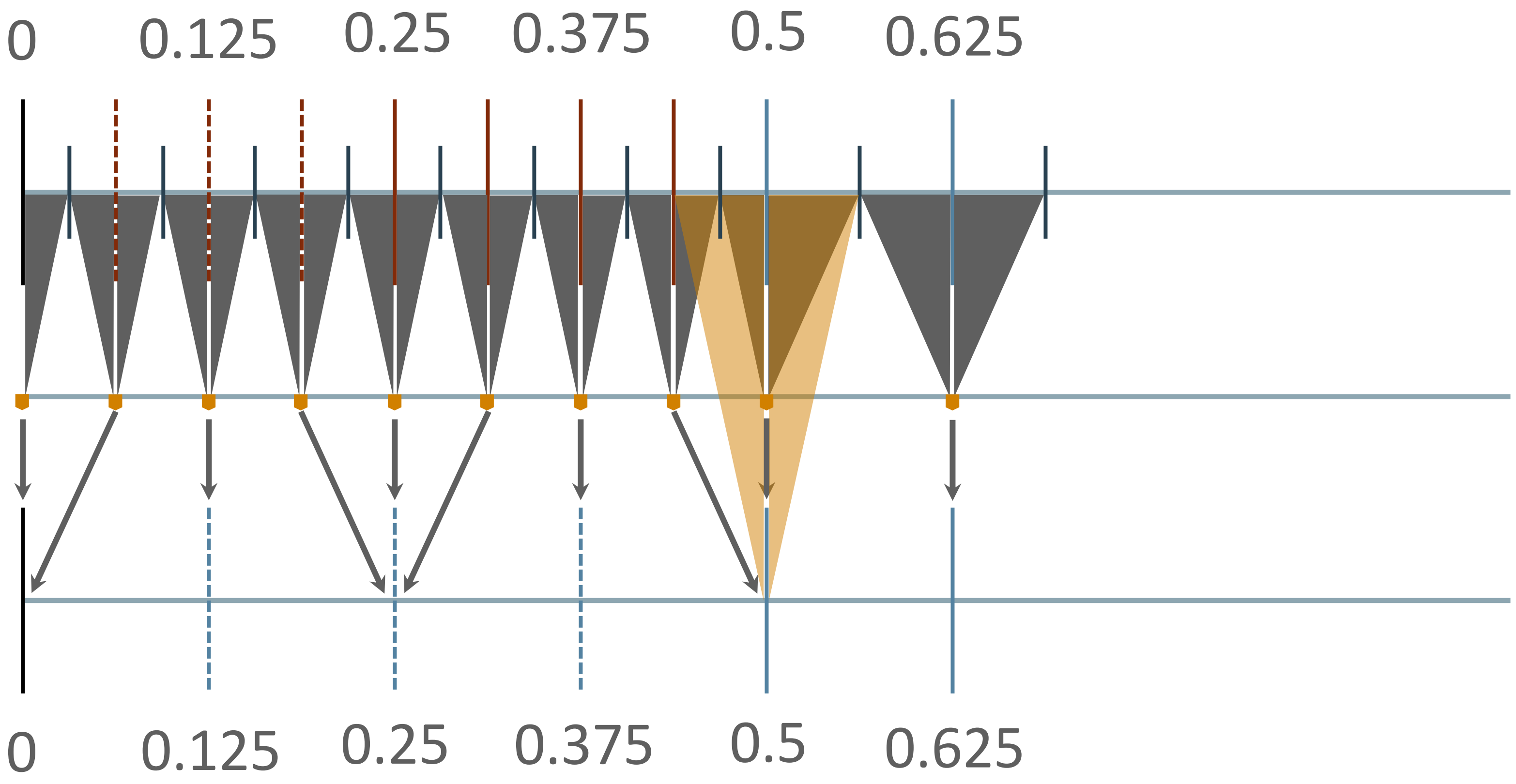
0.375 $= 0.11_2 \cdot 2^{-1}$

0.5 $= 1.00_2 \cdot 2^{-1}$

0.625 $= 1.01_2 \cdot 2^{-1}$

…

Are two roundings necessarily the same as one?



Region near 0.5 with different results from double rounding: [0.40625, 0.4375).

Toy format with p=3, Exp_{min} = –1

$0.0 = 0.00_2 \cdot 2^{-1}$
$0.125 = 0.01_2 \cdot 2^{-1}$
$0.25 = 0.10_2 \cdot 2^{-1}$
$0.375 = 0.11_2 \cdot 2^{-1}$
$0.5 = 1.00_2 \cdot 2^{-1}$
$0.625 = 1.01_2 \cdot 2^{-1}$
…

Toy format with p=3, Exp_{min} = –2

*$0.0625 = 0.01_2 \cdot 2^{-2}$ **
*$0.125 = 0.10_2 \cdot 2^{-2}$ **
*$0.1875 = 0.11_2 \cdot 2^{-2}$ **
$0.25 = 1.00_2 \cdot 2^{-2}$ *
$0.3125 = 1.01_2 \cdot 2^{-2}$ *
$0.375 = 1.10_2 \cdot 2^{-2}$ *
$0.4375 = 1.11_2 \cdot 2^{-2}$ *

Constructing a problematic example

- `Double.MIN_NORMAL` is: `0x1.0p-1022`
- Therefore max subnormal is: `0x0.ffff_ffff_ffff_fp-1022`
- The real value in the region of interest is equal to:
  `0x0.ffff_ffff_ffff_f_7p-1022`
- The decimal value of `0xffff_ffff_ffff_f_7` is
  `72_057_594_037_927_927`
- Prime factorization:
  72,057,594,037,927,927 = 11 × 13 × 1,877,171 × 268,435,459

Constructing a problematic example, cont.

- Idea: construct a pair of `double`'s whose exact product has the set of bits of interest and has the necessary exponent value
  - `Long.toHexString(11 * 1877171)      => 0x13b13b1`
  - `Long.toHexString(13 * 268435459L)  => 0xd0000027`
- Candidate FP values (taking care to not exceed *p* = 53 bits per candidate):
  - `0x1.3b13b1p-1022   (2.738552410633924E-308)`
  - `0xd.0000027p-4     (0.8125000090803951)`
- Results:
  - Strict product: `0x0.fffffffffffffp-1022` `(2.225073858507201E-308)`
  - Legal non-strict product: `0x1.0p-1022` `(2.2250738585072014E-308)`

Since SSE2 (Pentium 4 and later, circa 2001), x86 chips could do `float` + `double` without x87 so…

Wouldn't it be great if we never had to think about default mode floating-point semantics ever again?

JEP 306: Restore always strict floating-point semantics

- Restores original floating-point semantics to language and VM
  - No extended exponent value sets, etc.
  - `strictfp` /ACC_STRICT is a no-op since strict is the only defined semantics
  - Potentially frees up a rare method access flag bit position *Bidding starts at 400 quatloos!*
- x86 hardware *without* SSE is not a large market anymore; only x87 hardware helped by laxness granted by default floating-point
- Simple floating-point semantics everywhere is more important than simple code generation for a legacy instruction set

JEP 306: delivered in JDK 17, GA September 2021

- JLS and JVMS updates (small edits in many sections of the specs); built on earlier upgrade of floating-point terminology to IEEE 754-2019 in JDK 15.
- Release-sensitive warnings from `javac` about use of `strictfp`
- ACC_STRICT ignored for class file major version 61 and higher, freeing up that bit position (Valhalla already has designs for this bit…)
  - `AccessFlag` support as of JDK 20 supports decoding reuse of a bit position in different class file versions with different meanings.
- Removed HotSpot support for non-strict execution (HT David Holmes and Vladimir Ivanov)

JEP 306 facilitated completing port of FDLIBM to Java

- Ported remaining FDLIBM methods to Java for JDK 21: `log10`, `log1p`, `expm1`, `log`, `asin`, `acos`, `atan`, `atan2`, `sinh`, `cosh`, `tanh`, `sin`, `cos`, `tan`, `sqrt`, and `IEEEremainder`
- Don't have to worry about `strictfp`/`non-strictfp` inlining complications anymore since everything is strict

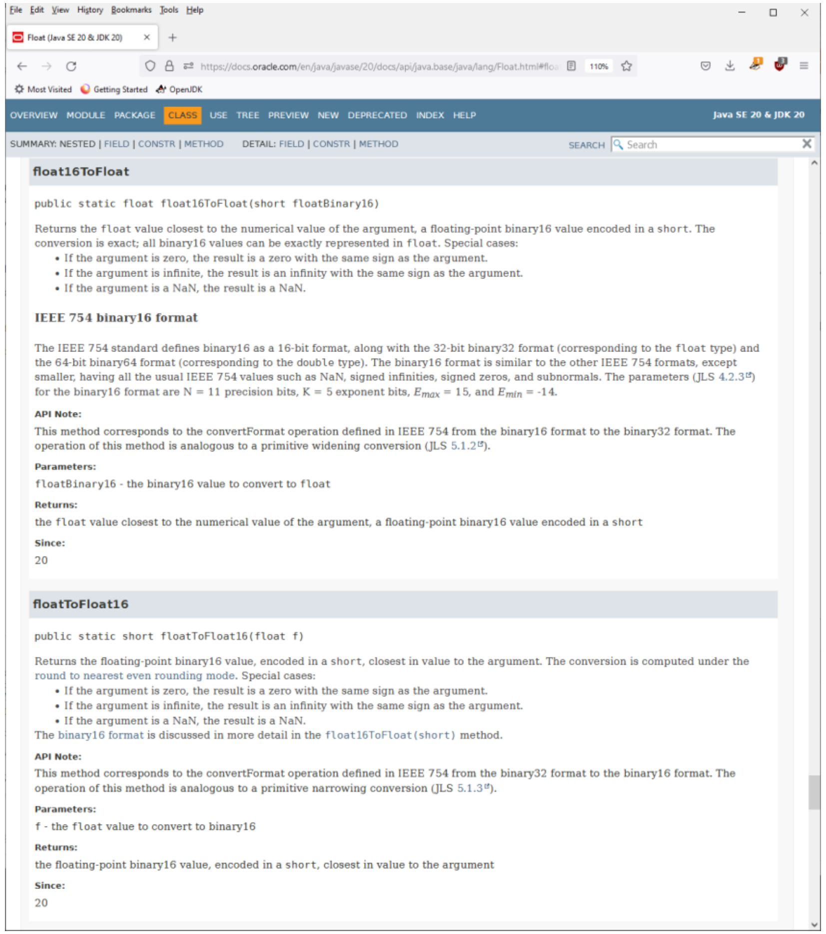
More progress on past possible future work…

Other formats of potential interest

- Support for “half” precision (16 bit) numbers;
  useful in machine learning (low-precision matrix-vector multiply), etc.
- Support for quad precision (128 bit)
  - Included as optional part of IEEE 754-2008 standard
  - 15 exponent bits, 113 precision bits
  - Some support in instruction sets (even predating the new standard), but not necessarily with direct hardware execution

Conversion for float16 ↔ float in JDK 20

- In `java.lang.Float`:
  - `public static float float16ToFloat(short floatBinary16)`
  - `public static short floatToFloat16(float f)`
- Better support for other numerical formats will be possible with Valhalla.


OVERVIEW MODULE PACKAGE CLASS USE TREE PREVIEW NEW DEPRECATED INDEX HELP

Java SE 20 & JDK 20

SUMMARY: NESTED | FIELD | CONSTR | METHOD DETAIL: FIELD | CONSTR | METHOD

float16ToFloat

`public static float float16ToFloat(short floatBinary16)`

Returns the `float` value closest to the numerical value of the argument, a floating-point binary16 value encoded in a `short`. The conversion is exact; all binary16 values can be exactly represented in `float`. Special cases:

- If the argument is zero, the result is a zero with the same sign as the argument.
- If the argument is infinite, the result is an infinity with the same sign as the argument.
- If the argument is a NaN, the result is a NaN.

IEEE 754 binary16 format

The IEEE 754 standard defines binary16 as a 16-bit format, along with the 32-bit binary32 format (corresponding to the `float` type) and the 64-bit binary64 format (corresponding to the `double` type). The binary16 format is similar to the other IEEE 754 formats, except smaller, having all the usual IEEE 754 values such as NaN, signed infinities, signed zeros, and subnormals. The parameters (JLS 4.2.3) for the binary16 format are N = 11 precision bits, K = 5 exponent bits, E_{max} = 15, and E_{min} = -14.

API Note:

This method corresponds to the convertFormat operation defined in IEEE 754 from the binary16 format to the binary32 format. The operation of this method is analogous to a primitive widening conversion (JLS 5.1.2).

Parameters:

`floatBinary16` - the binary16 value to convert to `float`

Returns:

the `float` value closest to the numerical value of the argument, a floating-point binary16 value encoded in a `short`

Since:

20

floatToFloat16

`public static short floatToFloat16(float f)`

Returns the floating-point binary16 value, encoded in a `short`, closest in value to the argument. The conversion is computed under the round to nearest even rounding mode. Special cases:

- If the argument is zero, the result is a zero with the same sign as the argument.
- If the argument is infinite, the result is an infinity with the same sign as the argument.
- If the argument is a NaN, the result is a NaN.

The binary16 format is discussed in more detail in the `float16ToFloat(short)` method.

API Note:

This method corresponds to the convertFormat operation defined in IEEE 754 from the binary32 format to the binary16 format. The operation of this method is analogous to a primitive narrowing conversion (JLS 5.1.3).

Parameters:

`f` - the `float` value to convert to binary16

Returns:

the floating-point binary16 value, encoded in a `short`, closest in value to the argument

Since:

20

To watch on the standards front…

- IEEE P3109: Standard for Arithmetic Formats for Machine Learning
  - 8-bit floating-point in the works

Thanks!