JavaOne℠

Sun's 2001 Worldwide Java Developer Conference™

What Everybody Using the Java™ Programming Language Should Know About Floating−Point Arithmetic

Joseph D. Darcy

Java Floating−Point Czar

Sun Microsystems, Inc.

Overview: Reduce Surprises, Increase Understanding

- Understand why
  - `0.1+0.1+0.1+0.1+0.1+0.1+0.1+0.1+0.1+0.1 != 1.0`
  - `0.1f != 0.1d`
- Outline
  - Floating−point fundamentals
  - Decimal ↔ binary conversion
  - Top **1.0e1** Floating−point FAQs, Mistakes, Surprises, and Misperceptions

Objectives

- Gain accurate mental model of binary floating−point arithmetic
  - Avoid common floating−point mistakes
- Learn where to find additional information
- Use floating−point more with greater confidence and productivity
- Inspire attendance at:
  - BOF 526 *What Some People Using the Java Programming Language Want to Know About Floating−Point Arithmetic* 11:00pm, Marriot, Salon 10

My Background

- Worked on languages and numerics since 1996
- UC Berkeley master's project:
  Borneo 1.0: Adding IEEE 754 Floating Point Support to Java™
- Active in Java™ Grande Forum, Numerics Working Group
- Assisted in design of revised floating−point semantics for the Java 2 platform
- Java Floating−Point Czar since September 2000
- Participant IEEE 754 revision committee

Why Floating−point?

- Integers aren't convenient for all calculations
- Floating−point arithmetic is a systematic methodology for approximating arithmetic on $\mathbb{R}$
  - Exponent and significand (mantissa) fields
  - "Decimal point" floats according to exponent value
- Exact multiplication can double the number of bits manipulated at each step—must approximate to keep computation tractable!
- Exactness rarely needed to get usable results

What Are Real Numbers?

- Real numbers ($\mathbb{R}$) include:
  - Integers (e.g., 0, −1, 32768)
  - Fractions (rational numbers) (e.g., ½, ¾, 22/7)
  - Irrational numbers (e.g., π, *e*, $\sqrt{2}$)
- Real numbers form a mathematical object called a field; fields have certain properties, field axioms
  - Addition and multiplication are commutative (*a op b = b op a*) and associative *((a op b) op c = a op (b op c))*
  - Closed under addition and multiplication
  - Also identity elements, distributivity, 13 total

JavaOne™

How to Approximate

- Not all approximations equally good!
- Would like approximation to be:
  - Deterministic, reproducible, predictable
  - Reliable, accurate
- Ideally also preserve properties of operations
  - Floating−point addition and multiplication are commutative
  - Round−off precludes most other field axioms
  - Floating−point is fundamentally discrete

Precision and Accuracy

- Precision ≠ Accuracy
  - Precision is a measure of how fine a distinction you can make
  - Accuracy is a measure of error
- Using more precision for intermediate results usually gives a more accurate computed answer

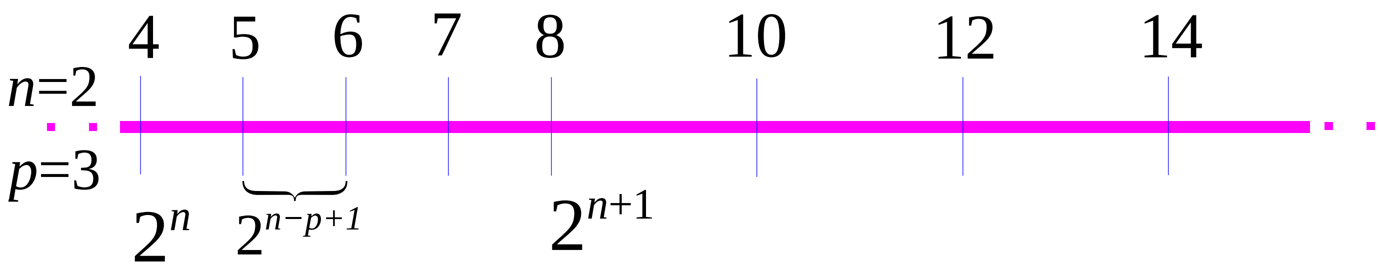
Binary Floating−Point Numbers

- Infinite number of real numbers, only finite number of floating−point numbers
- Representable numbers: $\pm \mathit{binaryFraction} \cdot 2^{\mathit{exponent}}$
  - *binaryFraction* limited in precision, only has a limited number of bits
  - Floating−point numbers are sums of powers of two
    - Ratio of largest to smallest component is at most 2^{p-1}, p is significand width

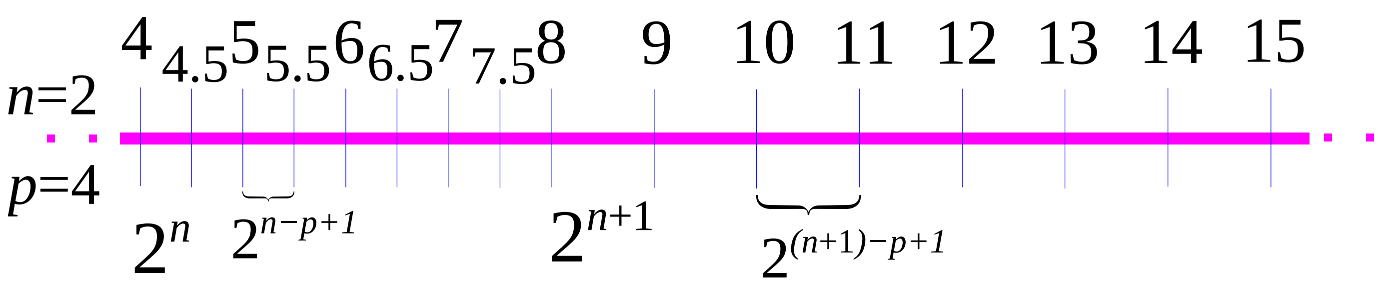
JavaOne

Binary Floating−Point Numbers Illustrated

- Floating−point format with 3 bits of precision



- Floating−point format with 4 bits of precision



- **float** has 24 bits of precision;
  double has 53 bits of precision

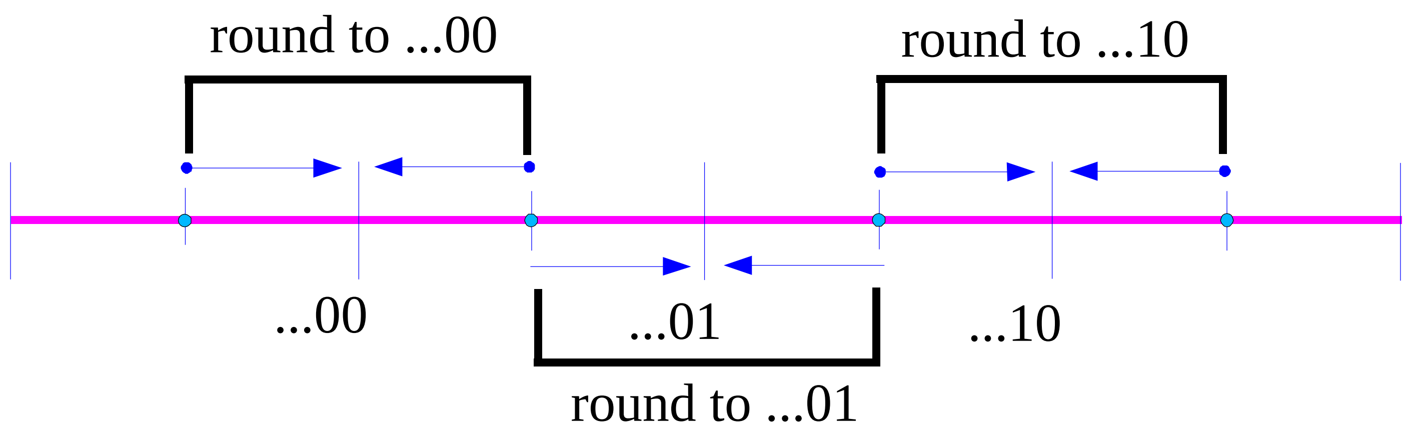
IEEE 754 Floating−Point

- IEEE 754 is the universally used binary floating−point standard
- IEEE 754 is fundamentally simple
- Conceptually, for each of the defined operations {+, −, *, /, √}
  1. First, calculate the infinitely precise result
  2. Second, round this result to the nearest representable number in the target format
     - (If two number are equally close, chose the one with the last bit zero—round to nearest even)

JavaOne

IEEE 754 Rounding Illustrated

- Round to nearest even pictorially



Round to Nearest Even

- Locally optimal, easy to understand and analyze
- But, still lose information, e.g., failure of associativity of addition:

(1.0f + 3.0e−8f) + 3.0e−8f == 1.0f

1.0f + (3.0e−8f + 3.0e−8f) == 1.0000001f

Creating Closure

- When an operation on a set of values doesn't have a defined result, often define a new kind of number
  - Positive integers and subtraction $\Rightarrow$ negative integers
  - Integers and division $\Rightarrow$ rational numbers
  - Rational numbers and square root $\Rightarrow$ complex numbers
- Helps create a closed system
  - Operation has defined result for all inputs

IEEE 754 Special Values

- Want floating−point arithmetic to be closed
  - Can have sensible semantics for new values
  - Allows computation to continue to a point where it is convenient to detect the “error” (e.g., root finder)
- Besides value for real numbers, IEEE 754 has infinities and NaN
  - Infinity: results from overflow **(MAX_VALUE*2.0)** or division by zero **(1.0/0.0)**
  - NaN (Not a Number): represents invalid values (0/0, $\infty * 0$, $\sqrt{-1}$, etc.)

JavaOne™

Base Conversion

- Integers can be represented exactly in any base
- In general, fractional quantities exactly representable as a finite string in one base cannot be exactly represented as a finite string in another base
  - In base 10, 1/3 is the non−terminating expansion 0.33333333...
  - In base 3, 1/3 is $0.1_{(3)}$
- Many floating−point surprises are related to decimal ↔ binary conversion properties

JavaOne

Decimal $\rightarrow$ Binary

- Most terminating decimal fractions cannot be exactly represented as terminating binary fractions
  - Try to convert 0.1 to a binary fraction

    $0.1 \times 2 = \underline{0}.2$ 0

    $0.2 \times 2 = \underline{0}.4$ 0

    $0.4 \times 2 = \underline{0}.8$ 0

    $0.8 \times 2 = \underline{1}.6$ 1

    $0.6 \times 2 = \underline{1}.2$ 1

    $0.2 \times 2 = \underline{0}.4$ 0

    ...

    Repeated state
  - 0.1 is $0.0\overline{0011}...$ in binary

JavaOne

Binary $\rightarrow$ Decimal

- However, all terminating binary fractions can be expressed exactly as terminating base 10 fractions
- Intuition: 10 = 2·5 so all fractions in base 2 or base 5 can also be expressed in base 10
- Proof: $\dfrac{1}{2^k} = \dfrac{5^k}{10^k}$
- 5^k is a representable integer; dividing by 10^k just shifts the decimal point; sums of 2^i still terminate
- Floating−point numbers are sums of power of two

JavaOne

How to Convert?

- Decimal $\rightarrow$ binary (**float** and **double** literals, **{Float, Double}.valueOf** and **parse{Float, Double}** methods)
  - Conversion must in general be inexact
  - Use standard floating−point rounding: return binary floating−point value nearest exact decimal value of input
- Binary $\rightarrow$ decimal (**{Float,Double}.toString**)
  - Feasible to return exact decimal string…

The Cost of Exactness

- Number of decimal digits for 2^{-n} grows with increasingly negative exponents

2^{-n}	Exact decimal string
2^{-1}	0.5
2^{-2}	0.25
2^{-3}	0.125
2^{-4}	0.0625
2^{-5}	0.03125
2^{-6}	0.015625
2^{-7}	0.0078125
2^{-8}	0.00390625
2^{-9}	0.001953125

Extreme Values

- **Double.MIN_VALUE =** 2^{-1074}
- Exact decimal value:
  4.9406564584124654417656879286822137236505980261432476442558568
  250067550727020875186529983636163599237979656469544571773092665
  671035593979639877479601078187812630071319031140452784581716784
  898210368871863605699873072305000638740915356498438731247339727
  316961514003171538539807412623856559117102665855668676818703956
  031062493194527159149245532930545654440112748012970999954193198
  940908041656332452475714786901472678015935523861155013480352649
  347201937902681071074917033322268447533357208324319360923828934
  583680601060115061698097530783422773183292479049825247307763759
  272478746560847782037344696995336470179726777175851256605511991
  315048911014510378627381672509558373897335989936648099411642057
  026370902792427675445652290875386825064197182655334472656 25e–
  324
- Awkward and impractical, is this necessary?

Criteria for Conversions

- Want binary → decimal → binary conversion to reproduce the original value
  - Allows text to be used for reliable data interchange
- Exact decimal value is not necessary to recreate original value
- Decimal → binary conversion must already deal with imprecision and rounding
- Use an inexact decimal string with enough precision to recreate original value

Which String to Use?

- Many decimal strings map to a given floating−point value
  - **"1.0" $\rightarrow$ 2^0**
    "1.00000000000000000000000000000001"$\rightarrow$ 2^0
- Choose shortest string that rounds to the desired floating−point value
- How much precision is needed?

How Long a String Is Needed?

- **float** format has 6 to 9 digits of decimal precision
- **`double`** format has 15 to 17 digits of decimal precision
- (Precision varies since binary and decimal numbers have different relative densities in different ranges)

Implications: WYSI *Not* WYG

- What you see is not what you get
  - **"0.1f"** $\neq$ **0.1** after conversion; exact value: **0.100000001490116119384765625**
  - **"0.1d"** $\neq$ **0.1** after conversion; exact value: **0.1000000000000000055511151231…**
- Correct digits
  - Leading 8 for **float**
  - Leading 17 for **double**

You Are in a Twisty Maze of Little Passages, All Different…

- String representation of a floating−point value is format dependent
  - **Float.toString(0.1f) = "0.1"**
  - **Double.toString(0.1f) = "0.10000000149011612"**
    - **float** approximation has 24 significand bits; **double** approximation has 53 significand bits
  - **Double.toString(0.1d) = "0.1"**
- To preserve values, must print out and read in floating−point numbers in the same format

Base Conversion Summary

- Both decimal to binary and binary to decimal conversions are inexact
  - Decimal $\rightarrow$ binary: fundamentally inexact
  - Binary $\rightarrow$ decimal: done inexactly for practical reasons
- Roundtrip binary $\rightarrow$ decimal $\rightarrow$ binary can be exact since the inexactness is correlated
- Can only exactly represent binary values in floating−point numbers for the Java platform

Top 1.0e1 Floating–Point FAQs, Mistakes, Surprises, and Misperceptions

1 – Expecting Exact Results

- **0.1+0.1+0.1+0.1+0.1+0.1+0.1+0.1+0.1+0.1 != 1.0**
  - The literal **"0.1"** doesn’t equal 0.1
  - Limited precision of floating−point implies roundoff
- More generally, exact results also fail from
  - Limited range (overflow, underflow)
  - Special values

JavaOne

2 – Expecting *All* Results to Be Inexact

- Cases where floating−point computation is exact
  - Operations on “small” integers
  - Representing in−range powers of 2 (e.g., **1.0/8.0**)
  - Special algorithms, e.g., techniques to extend floating−point precision
    (Learn more at the BOF!)

2.5 – Expecting *All* Results to Be Inexact

- A floating−point number is not a stand−in for nearby values
  - Floating−point arithmetic operations assume their inputs are exact
  - Must use other techniques to estimate overall error (e.g., error analysis, interval arithmetic)

JavaOne

3 – Using Floating–Point For Monetary Calculations

- Fractional $, £, €, ¤ can't be stored exactly
  - Decimal → binary conversion issue
- Operations on values won't be exact (bad for balancing a checkbook!)
- Recommendations
  - Use an integer type (**int** or **long**) operating on cents
  - Use **java.math.BigDecimal** for exact calculations on decimal fractions
  - If you must use floating−point, operate on cents
    - Problems with limited exact range

JavaOne

4 – Preserving Cardinal Values of sin and cos With Arguments in Degrees

- Why doesn't
  - **sin(toRadians(180)) == 0.0**
  - **cos(toRadians(90)) == 0.0**
- **toRadians ≡ angleInDegrees/180.0*Double.PI**
  - Conversion of degrees to radians is inexact
    - **Double.PI** ≠ π **(**∴ **sin(Double.PI**) ≠ sin(π))
    - (in general case, roundoff in multiply, divide)
- Cope with small discrepancies or use degree−based transcendental functions

5 – Comparing Floating−Point Numbers For Equality

- Sometimes okay to compare for equality
  - When calculations are known to be exact
  - To synthesize a comparison
  - Compare against **0.0** to avoid division by zero
- But, floating−point computations are generally inexact
  - Comparing floating−point numbers for equality may have undesirable results

5.5 – Comparing Floating−Point Numbers For Equality, (Cont.)

- An infinite loop:
```
d = 0.0;
while(d != 1.0) {d += 0.1};
```
- For counted loops, use an integer loop count:
```
d = 0.0;
for(int i = 0; i < 10; i++)
  {d += 0.1};
```
- To test against a floating−point value, use ordered comparisons (<, <=, >, >=):
```
d = 0.0;
while(d <= 1.0)
  {d += 0.1};
```


JavaOne

6 – Using `float` For Calculations

- Storing low−precision data as **`float`** is fine, but
- Generally not recommended to use **`float`** for computations
  - **`float`** has less than half the precision of **`double`**
  - Using **`double`** intermediates greatly reduces the risk of roundoff problems polluting the answer
  - Round **`double`** value back to **`float`** to give a **`float`** result
  - (For more information see references)

7 – Trusting Venerable Formulas

- Some formulas found in text books don’t work very well with floating−point numbers
- Formulas may implicitly assume real arithmetic
- Don’t adequately take floating−point rounding into account

7.5 – Trusting Venerable Formulas

- Example: Heron's formula for the area of a triangle given the lengths of its sides:

  $s=((a + b)+c)/2,$
  $\text{Area} = \text{sqrt}(s\cdot(s-a)\cdot(s-b)\cdot(s-c))$

  - Formula can fail for needle like triangles (no bits may be correct!)
  - A better algebraicly equivalent formula is available
  - Can also use more intermediate precision (see references)

8 – How to Round to 2 Decimal Places...

- May want to use "C−style" output for floating−point numbers; e.g., limiting the number of digits after the decimal point
  – see **`java.text.NumberFormat`**
  – **`e.g. DecimalFormat twoDigits = new DecimalFormat( "0.00" );`**
- Default "**`%g`**" format conversion of C's **`printf`** does not print enough digits to recover the original value

9 – What Are Distinguishing Features of Java™ Programming Language Floating−Point?

- Required use of IEEE 754 numbers
  - Subnormals must be supported, flush to zero not allowed
- Correctly rounded decimal ↔ binary conversion
- Well−defined expression evaluation rules
  - Yields predictable results
  - Code semantics depend on source, not compiler flags

10 – What is `strictfp`?

- Java 2 method and class qualifier
- Indicates floating−point computations must get exactly reproducible results
- Without **`strictfp`**, some variation is allowed
  – Intermediate results can have extended exponent range
  – Only makes a difference if an overflow or underflow would occur
- Only need to use **`strictfp`** if you want exactly reproducible results

Philosophical Note: The Need for Speed

- At times the speed of a program is critical; a late answer is not useful
- However, speed is not the only criterion
- Speed is comparatively easy to measure compared to accuracy or robustness
- If you don’t care what is computed, why do you care how fast it is computed?
- Other design values robustness, predictability, and repeatability, not just speed

How Fast Is Java™ Platform Floating−Point Today?

- Much faster than it used to be :−)
  - Modern vm's can generate code similar to static C compilers
- Benchmark results across languages vary
  Benchmarking Java™ against C and Fortran for Scientific Applications,
  Bull, Smith, Pottage, Freeman
  http://www.epcc.ed.ac.uk/research/publications/conference/jgflangcomp_final.ps.gz
  - PIII running NT, mean ratio to C: 1.23
  - PIII running Linux, mean ratio to C: 1.07
  - Solaris™ UltraSPARC™, mean ratio to C: 1.61

C and FORTRAN Comparison

- C and FORTRAN compilers have been around longer
- The Java™ programming language has tighter semantics than C or FORTRAN
  - Can't "optimize" floating−point as much
  - Can't assume arrays aren't aliased
- JSRs are addressing speed and expressibility issues

Recommendations and Rules of Thumb

- Sometimes okay to break the rules
- Store large amounts of data no more precisely than necessary
- Take advantage of **double** precision
- See references for further suggestions

Summary

- Floating−point arithmetic only approximates real arithmetic
  – Floating−point approximation is predictable
- Avoid surprises from base conversion
- Understand when exact results should be expected
- The Java™ programming language makes different floating−point design choices than other languages

Conclusions

- Floating−point arithmetic follows rules; just not the rules you are accustomed to :−)
- Use knowledge of floating−point to
  – Reduce numerical surprises
  – Productively use Java technology's numerics
  – Take advantage of floating−point semantics
- More floating−point material at companion BOF *What Some People Using the Java™ Programming Language Want to Know About Floating−Point Arithmetic* 11pm Marriot Salon–10

JavaOne

Where to Get More Information

- Professor Kahan’s webpages
  - *Marketing vs. Mathematics,* http://www.cs.berkeley.edu/~wkahan/MktgMath.pdf
  - *What has the Volume of a Tetrahedron to do with Computer Programming Languages?* http://www.cs.berkeley.edu/~wkahan/VtetLang.pdf
  - *Miscalculating Area and Angles of a Needle−like Triangle,* http://www.cs.berkeley.edu/~wkahan/Triangle.pdf
  - *Lecture Notes on the Status of the IEEE Standard 754 for Binary Floating−Point Arithmetic,* http://www.cs.berkeley.edu/~wkahan/ieee754status/ieee754.ps

Where to Get Still More Information

- *What Every Computer Scientist Should Know About Floating Point Arithmetic*,David Goldberg, (with commentary by Doug Priest) http://www.validgh.com/goldberg/paper.ps
- *Computer Arithmetic: Algorithms and Hardware Designs*, Behrooz Parhami, ISBN 0−19−512583−5
- *The Art of Computer Programming, Vol. 2, Seminumerical Algorithms*, Donald Knuth
- Java™ Grande Forum, Numerics Working Group http://math.nist.gov/javanumerics/

Q&A

JavaOne℠

Sun's 2001 Worldwide Java Developer Conference