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
  - Performance and Language Comparisons

BEGINNING

Objectives

- Gain accurate mental model of binary floating-point arithmetic
- Avoid common floating-point mistakes
- Use floating-point more with greater confidence and productivity
- Learn where to find additional information

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™
- Participant in IEEE 754 revision committee
- Assisted in design of revised floating-point semantics for the Java 2 platform
- Java Floating-Point Czar since September 2000

Floating-Point Fundamentals

Why Floating-point?

- Integers aren't convenient for all calculations
- Floating-point arithmetic is a systematic methodology for approximating arithmetic on $\Re$
  - 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 ($\Re$) 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

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

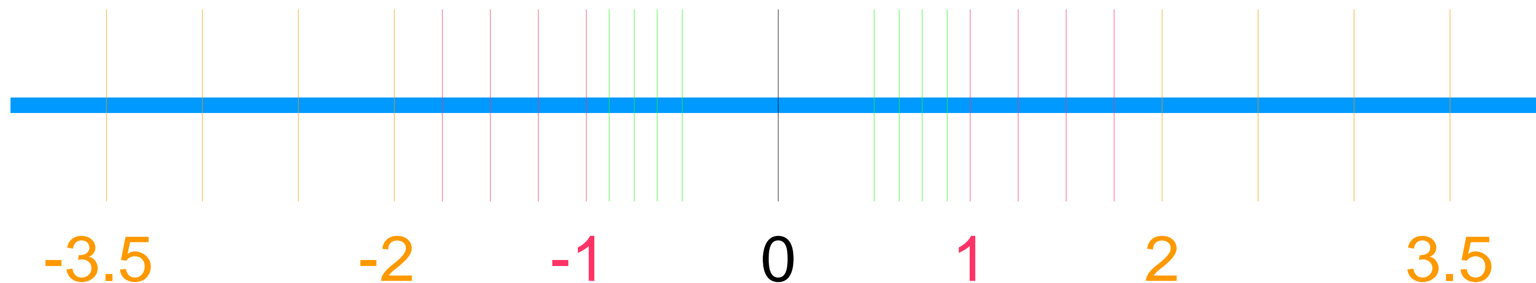
- Precision $\neq$ Accuracy
  - Precision is a measure of how fine a distinction you can make
  - Accuracy is a measure of error
- Using more precision for intermediate results usually gives a more accurate computed answer

Binary Floating-Point Numbers

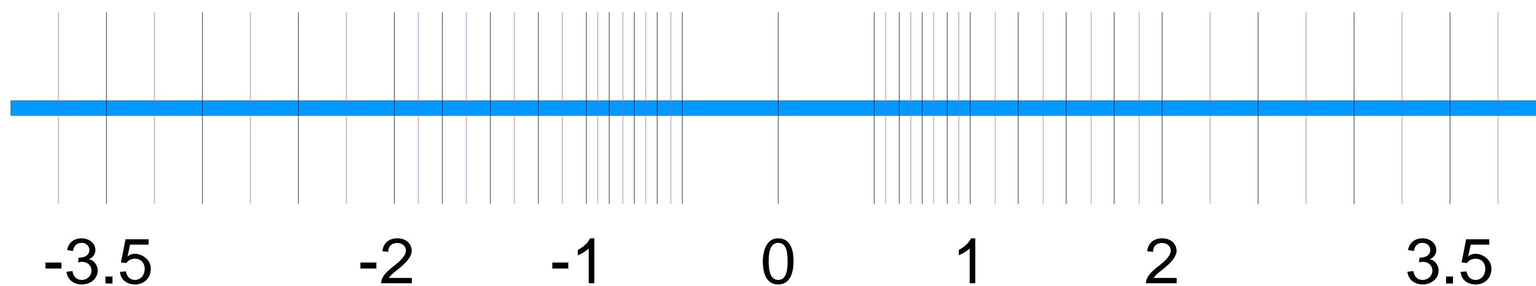
- Infinite number of real numbers, only finite number of floating-point numbers
- Representable numbers: $\pm binarySignificand \cdot 2^{exponent} = \pm b_0.b_1 b_2 \ldots b_{p-1} \cdot 2^{exponent}$
  - *binarySignificand* limited in precision, only has *p* bits
    - normalized: (for now) assume $b_0 = 1$
  - Floating-point numbers are sums of powers of two

Toy Binary Floating-Point Numbers Illustrated

- Toy format 1: p=3, exponent ∈{-1, 0, 1}



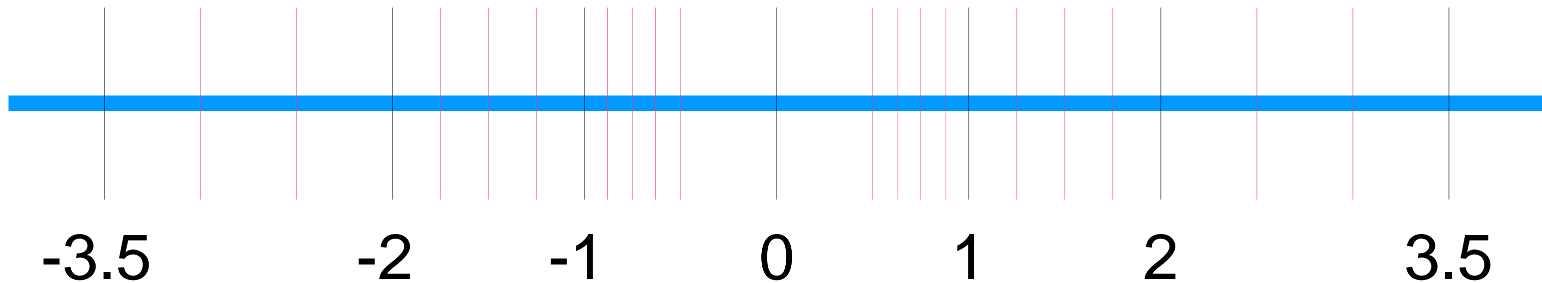
- Toy format 2: p=4, exponent ∈{-1, 0, 1}



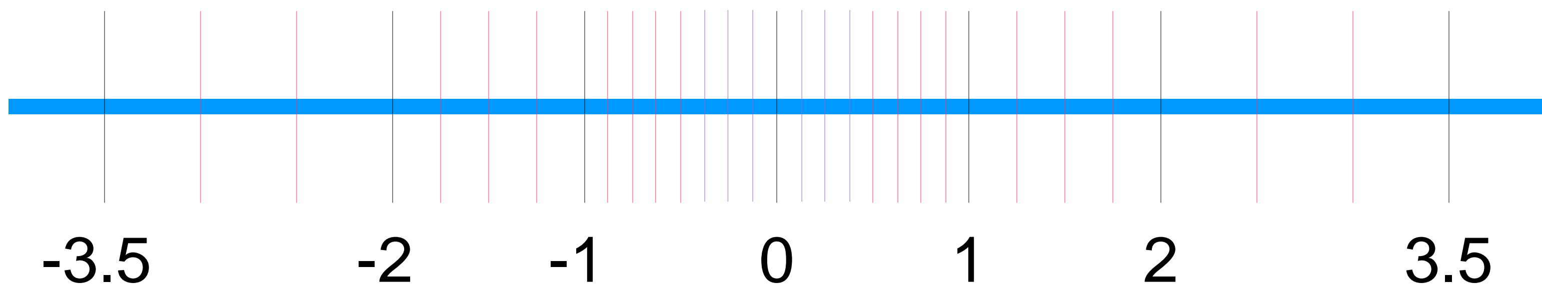
Toy format observations

- Adding one bit of precision doubles the number of representable numbers
  - additional precision slightly increases the largest representable number
- When the exponent increases by one, the distance between adjacent floating-point numbers doubles: relative precision is constant
- Ratio of largest to smallest component of a number is at most 2^{p-1}
  - e.g. in toy format 1, 1.75 = (1 + 0.5 + 0.25); 1/(0.25) = 4 = 2^{3-1}

Subnormals



- Fill-in comparatively large gap around zero
  - for smallest numbers, allow b_0=0; creates subnormal numbers
  - Eases numerical analysis



`float` and `double` formats

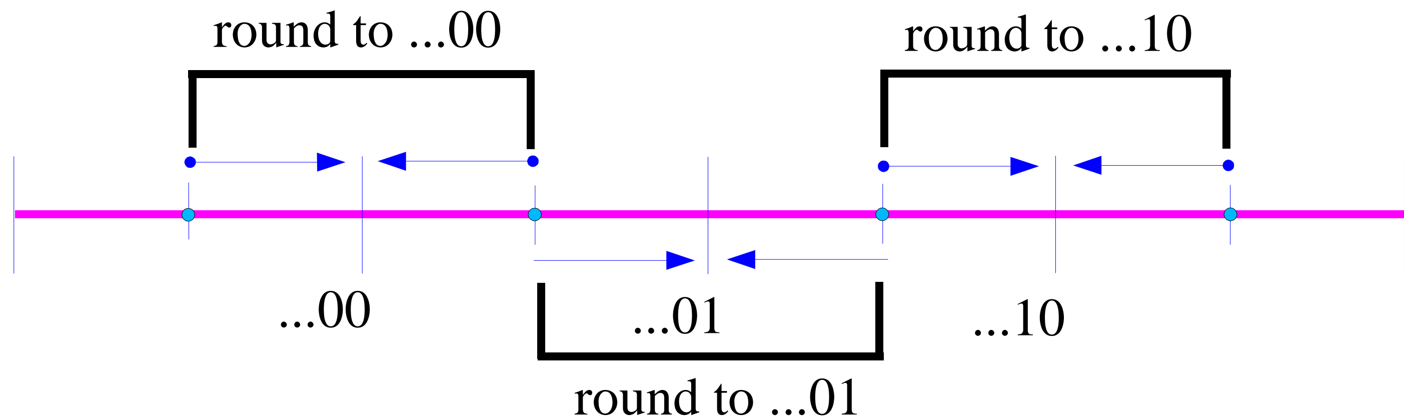
- `float` has *p*=24, 8 exponent bits
- `double` has *p*=53, 11 exponent bits
- All `float` numbers are exactly representable as `double` numbers
- Between adjacent `float` numbers, $2^{(53-24)} = 2^{29} \approx$ 500 million `double` numbers

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)

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
- Helps create a closed system
- Operation has defined result for all inputs
  - new values might be unfamiliar but convenient, or sensible mathematically

Math through the ages

- Positive integers {1, 2, 3, ...} and subtraction ⇒ all integers {..., -1, 0, 1, ...}
- Integers and division ⇒ rational numbers
  - division by zero not allowed
- Rational numbers and square root ⇒
  - irrational numbers (√2)
  - imaginary and complex numbers (√(-1))

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 values for real numbers, IEEE 754 has infinities and NaN

Infinities and NaN

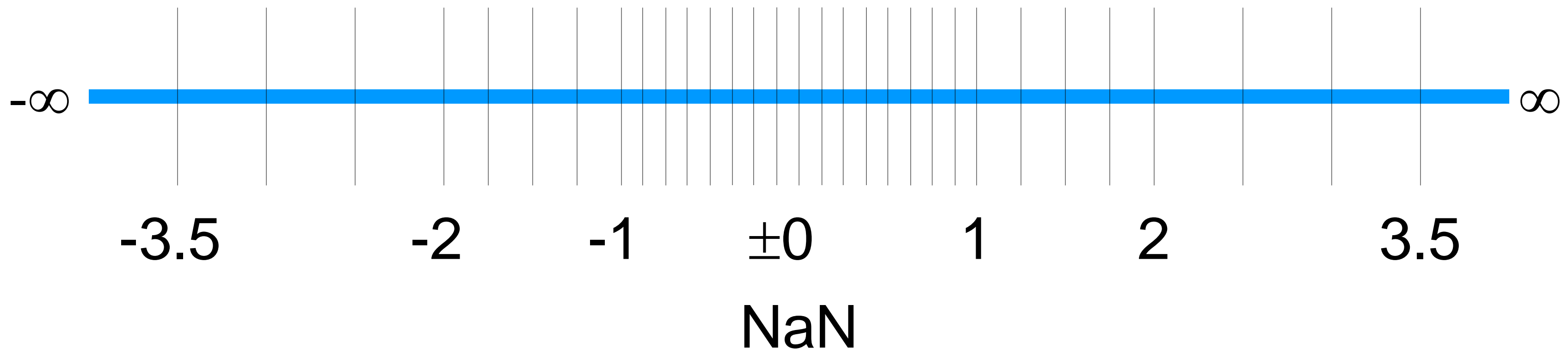
- Infinity results from
  - overflow, **`MAX_VALUE*2.0`**
  - division by zero, `1.0/0.0`
- NaN (Not a Number) represents invalid values
  - 0/0
  - $\infty \cdot 0$
  - $\infty - \infty$
  - $\sqrt{-1}$, etc.
  - NaN shifts code complexity;
    their creation doesn’t have to be prevented

A Distinction with some Difference

- IEEE 754 has signed zeros as two distinct values
  - `+0.0 == -0.0` but,
  - While, `F(+0.0)` is usually the same as `F(-0.0)` it is not always
    - `1.0/+0.0 == +∞`
    - `1.0/-0.0 == -∞`
- The sign of a zero input
  - Usually only effects of sign of a zero result (divide exception above)
  - Otherwise, can mostly be ignored

Complete set of toy floating-point values

- All operations on toy floating-point values will result in some other toy floating-point value



May I take your order?

- IEEE 754 defines, <, >, and == relations
- NaN is unordered with respect to other values
  - NaN < 5 is false; NaN > 5 is false;
    NaN == 5 is false; NaN != 5 is true
  - NaN == NaN is false too!
- Because of signed zeros and NaN, IEEE 754 < does not define a total ordering
  - Use **`{Float, Double}.compareTo`** if you need a total ordering
- IEEE 754 == is not an equivalence relation

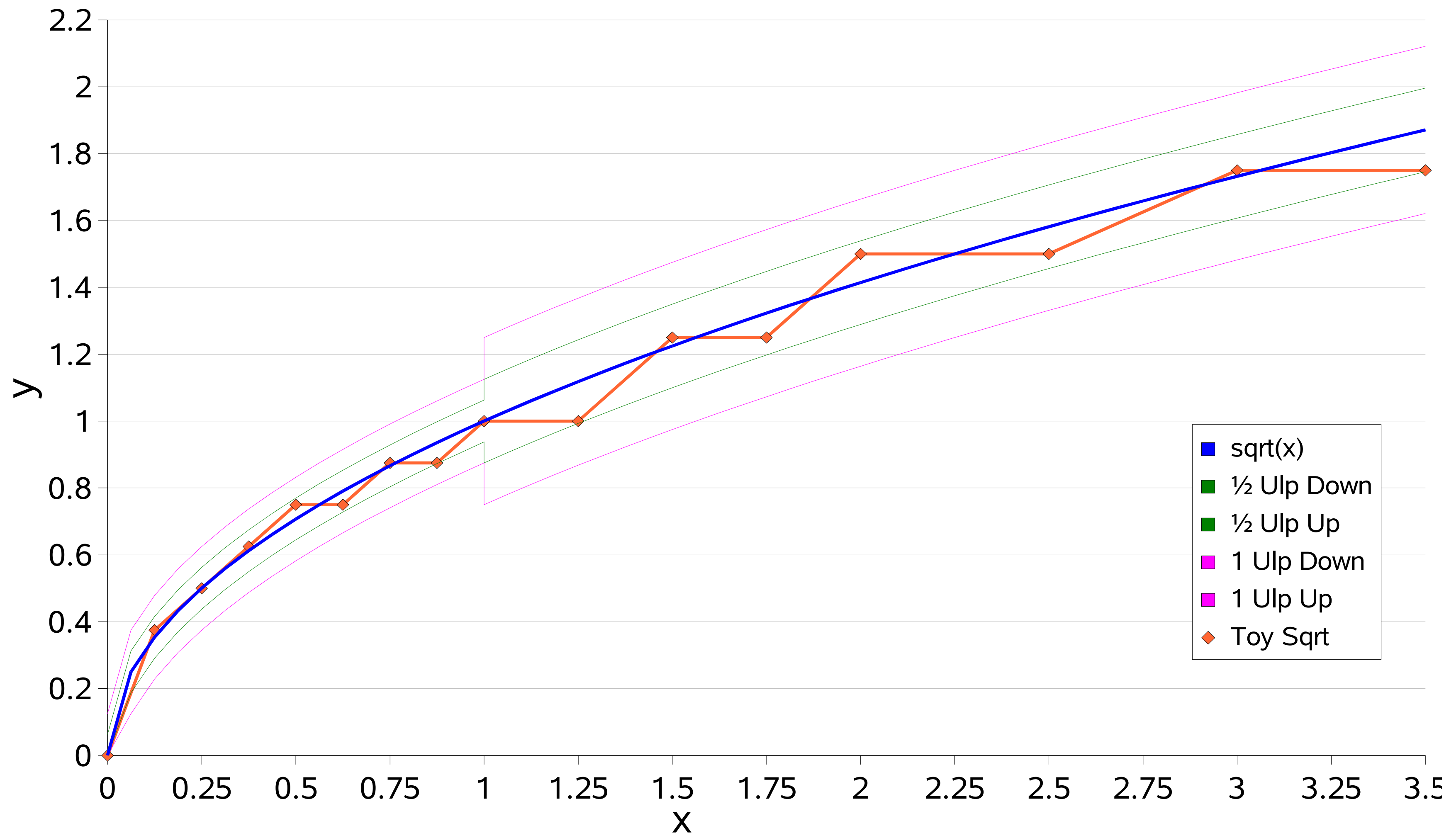
Approximation Accuracy: ulps make a bit of difference

- Ulp = unit in the last place
  - ulp($r \in \Re$) = distance between floating-point numbers bracketing *r*
  - If *r* is exactly representable, ulp(*r*) is the distance to the next larger floating-point value
  - Convenient measure of relative error
  - Error ≤ ½ ulp ⇒ function is correctly rounded
    - Floating-point value closest to exact result is returned, best a floating-point function can do
    - Basic arithmetic and sqrt are correctly rounded
  - Most single argument methods in **`Math`** and **`StrictMath`** have error < 1 ulp

Round, round, get around

- A correctly rounded approximation most likely to preserve other properties of interest
  - Cardinal values are correct;
    sqrt(9.0) =3.0
  - Monotonicity;
    $x, y > 1;\ x \le y \Rightarrow \mathrm{sqrt}(x) \le \mathrm{sqrt}(y)$
- Other fairly accurate approximations don't necessarily preserve desired properties

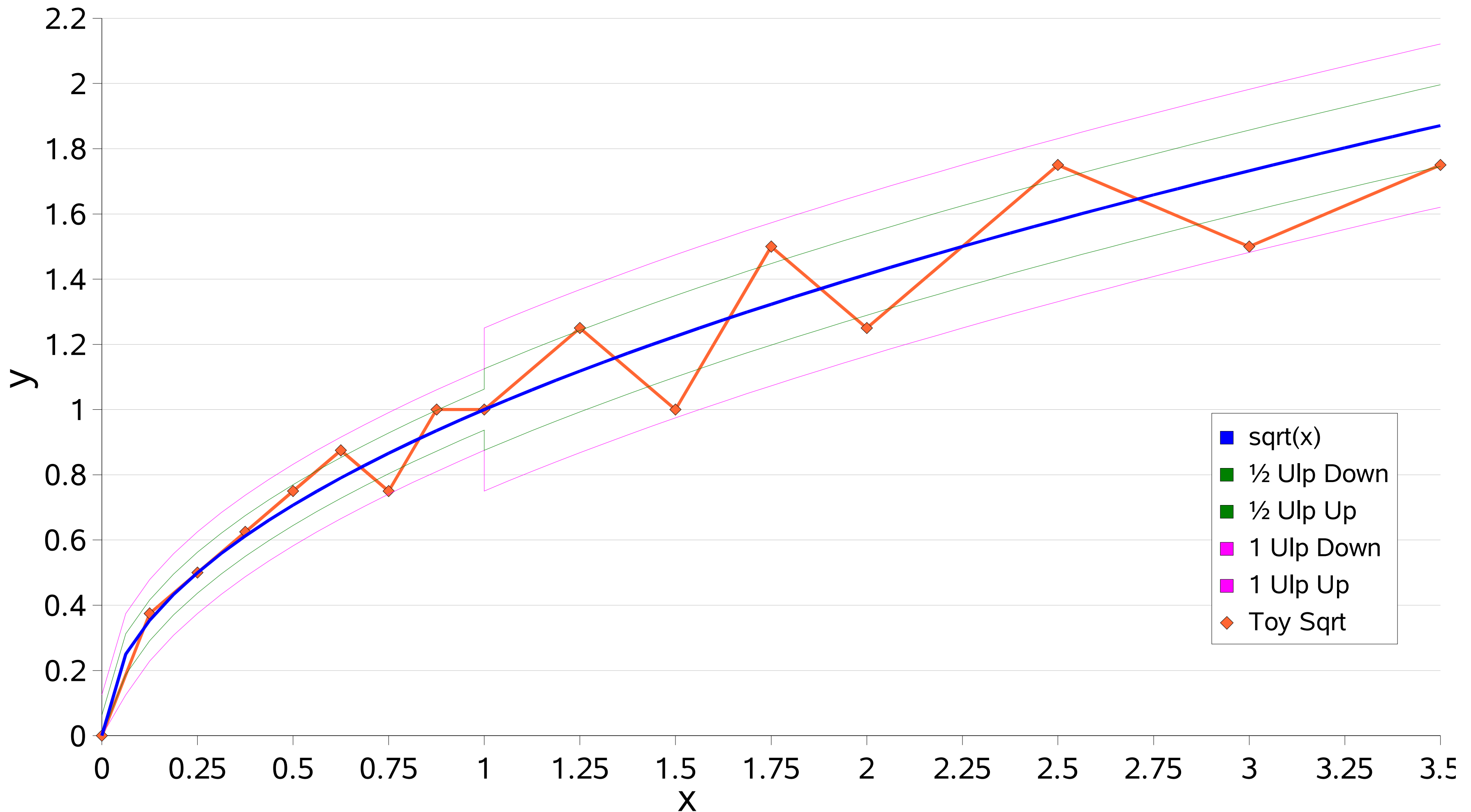
Toy Square Root, correctly rounded



Toy Square Root Tabulated

x	sqrt(x)
0.0	0.0
0.125	0.375
0.250	0.500
0.375	0.625
0.500	0.750
0.625	0.750
0.750	0.875
0.875	0.875
1.000	1.000
1.250	1.000
1.500	1.250
1.750	1.250
2.000	1.500
2.500	1.500
3.000	1.750
3.500	1.750

Toy Square Root, not so correctly rounded



JavaOne
Sun's 2003 Worldwide Java Developer Conference

Decimal ↔ Binary Conversion

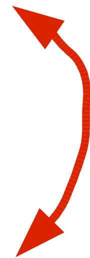
Base Conversion Basics

- 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

Decimal → 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

Binary → 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: $\frac{1}{2^k} = \frac{5^k}{10^k}$
- 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

How to Convert?

- Decimal → 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 → 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:
  4.9406564584124654417656879286822137236505980261432476442558
  568250067550727020875186529983636163599237979656469544571773
  092665671035593979639877479601078187812630071319031140452784
  581716784898210368871863605699873072305000638740915356498438
  731247339727316961514003171538539807412623856559117102665855
  668676818703956031062493194527159149245532930545654440112748
  012970999954193198940908041656332452475714786901472678015935
  523861155013480352649347201937902681071074917033322268447533
  357208324319360923828934583680601060115061698097530783422773
  183292479049825247307763759272478746560847782037344696995336
  470179726777175851256605511991315048911014510378627381672509
  558373897335989936648099411642057026370902792427675445652290
  87538682506419718265533447265625e-324
- Awkward and impractical, is this necessary?

Criteria for Conversions

- Want binary → decimal → binary conversion to reproduce the original value
  - **`d2`** in **`d2=parseDouble(toString(d1))`** is the same as **`d1`**
  - Allows text to be used for reliable data interchange
  - Exact decimal value is not necessary
- 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"` $\rightarrow 2^0$
    `"1.000000000000000000000000001"` $\rightarrow 2^0$
- 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"`** ≠ **`0.1`** after conversion; exact value: **`0.100000001490116119384765625`**
  - **`"0.1d"`** ≠ **`0.1`** after conversion; exact value: **`0.1000000000000000055511151231`**…
- Correct digits
  - Leading 8 for **`float`**
  - Leading 17 for **`double`**
- When you see a decimal string, think of its nearest binary neighbor

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 → binary: fundamentally inexact
  - Binary → decimal: done inexactly for practical reasons
- Roundtrip binary → decimal → 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

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

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)

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

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` $\neq \pi$; $\therefore$ `sin(Double.PI)` $\neq$ 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};
```

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

7.25 – Trusting Venerable Formulas

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

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

  - Most often works just fine:
    Area(△(3, 4, 5)), $s = 6$, Area = sqrt($6 \cdot 3 \cdot 2 \cdot 1$) = 6
    Area(△(3, 4, 5)) = $\frac{1}{2} b \cdot h = \frac{1}{2} \cdot 4 \cdot 3 = 6$
  - But, formula can fail for needle like triangles (no bits may be correct!)

7.5 – Trusting Venerable Formulas

- Calculate the area of △(12,345,679, 12,345,679, 1.01234)
  - ½$b \cdot h$ should be approx. 6,000,000
  - Area from Heron's in **`float`** precision: 12,345,680.00, 2X too big!
  - Use alternative formula
    Area = sqrt((a+(b+c))(c-(a-b))(c+(a-b))(a+(b-c)))/4:
    6,249,012.00, Correct answer.
  - Area from Heron's with **`float`** inputs but **`double`** intermediates:
    6,249,012.00, Correct answer.

7.75 – Trusting Venerable Formulas

- Calculate the area of △(12,345,679, 12,345,678, 1.01234)
  - ½*b*·*h* should be >> 0
  - Area from Heron's in **`float`** precision: 0.0 !!!
  - Use alternative formula
    Area = sqrt((a+(b+c))(c-(a-b))(c+(a-b))(a+(b-c)))/4:
    972,730.06 Correct answer.
  - Area from Heron's with **`float`** inputs but **`double`** intermediates:
    972,730.06, Correct answer.

7.875 – Trusting Venerable Formulas

- Problem with Heron's formula hard to recognize and find since it doesn't occur all the time
- Alternative formula is algebraicly equivalent but more robust computationally
  - Where to find such formulas?
  - How to you know you need one?
- Can use more intermediate precision to help avoid roundoff issues
  - Good insurance policy
  - See references for discussion

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
- See JDC article “Formatting Decimal Numbers” http://developer.java.sun.com/developer/TechTips/2000/tt0411.html#tip1

9 – What is `strictfp`?

- Java™ 2 platform 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

10 – What is the difference between `Math` and `StrictMath`?

- Not directly related to **`strictfp`**
- **`Math`** and **`StrictMath`** have the same set of static methods
  - **`StrictMath`** methods must use a particular implementation algorithm, FDLIBM
  - **`Math`** methods can use other algorithms as long as quality of implementation criteria met
    - accuracy
    - monotonicity
  - **`Math`** methods can be platform-optimized

JavaOne℠

Sun's 2003 Worldwide Java Developer Conference

Language Comparisons

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 on selection of compiler flags

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
  - Can't make use of unspecified order of operations or undefined behavior
- Different languages have different goals

What's New in 1.5?

- More methods in **`Math`** and **`StrictMath`**

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

E
N
D

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

E N D

Where to Get More Information

- Professor Kahan's webpages
  - *Marketing vs. Mathematics,* http://www.cs.berkeley.edu/~wkahan/MktgMath.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.validlab.com/goldberg/paper.pdf
- *Numerical Computing with IEEE Floating Point Arithmetic*, Michael L. Overton, ISBN 0-89871-482-6
- *Floating-Point Fallacies*, Dan Zuras, http://zuras.org/~dan/FPFallacies.html
- *The Art of Computer Programming, Vol. 2, Seminumerical Algorithms*, Donald Knuth

Q&A

JAVA™

JavaOne℠

Sun's 2003 Worldwide Java Developer Conference

JAVA™

java.sun.com/javaone/sf