Using Java 5.0 BigDecimal

Mike Cowlishaw
IBM Fellow

http://www2.hursley.ibm.com/decimal

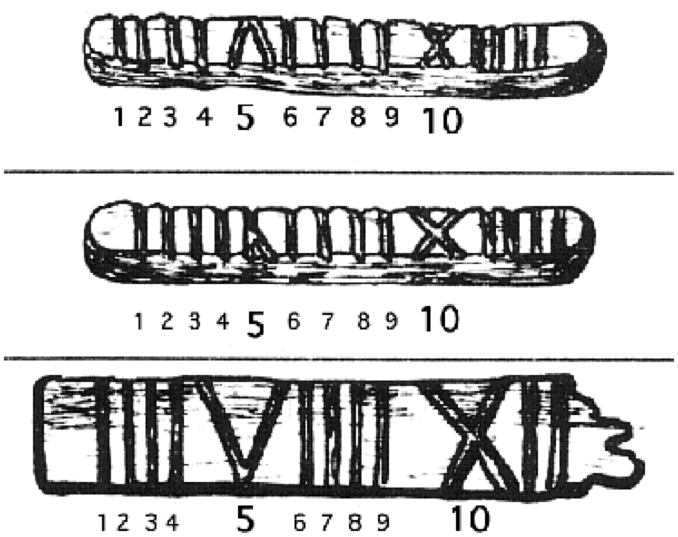
10

Overview

- Background
  - Why decimal arithmetic is important
  - New standards for decimal formats, arithmetic, and hardware
- Java 5.0 BigDecimal (what's new, what's fast?)
- Questions?

Origins of Decimal Arithmetic

- Decimal (base 10) arithmetic has been used for thousands of years



- Algorism (Indo-Arabic place value system) in use since 800 AD
- Mechanical calculators were decimal

Early Computers

- Often derived from mechanical decimal calculator designs
- Often decimal (even addresses)
- But binary was shown to be more efficient
  - minimal storage space
  - more reliable (20% fewer components)

Today's Computer Arithmetic

ALU

Integer arithmetic

FPU

Floating-point arithmetic (IEEE 754)

byte, short, int, long, *etc.*

single, float, double, quad, *etc.*

Today’s Computer Arithmetic

ALU

12 x 12 → 144

byte, short, int, long, *etc.*

FPU

1.2 x 1.2 → 1.44

single, float, double, quad, *etc.*

1.2 x 1.2 = 1.44 ?

- Binary fractions *cannot* exactly represent most decimal fractions (*e.g.,* 0.1 requires an infinitely long binary fraction)
- 1.2 in a 32-bit binary float is actually: 1.2000000476837158203125
- and this squared is: 1.440000057220458984375

Why Not Just Round ?

- Rounding hides but does not help
  - obscures the slight inaccuracies
  - errors accumulate [double rounding]

Decimal	Java float (binary)
9	9
0.9	0.9
0.09	0.089999996
0.009	0.009

Where It Costs Real Money…

- Add 5% sales tax to a $ 0.70 telephone call, rounded to the nearest cent

- 1.05 x 0.70 using binary double is exactly

  0.73499999999999998667732370449812151491641998291015625

  (should have been 0.735)

- rounds to $ 0.73, instead of $ 0.74

Hence…

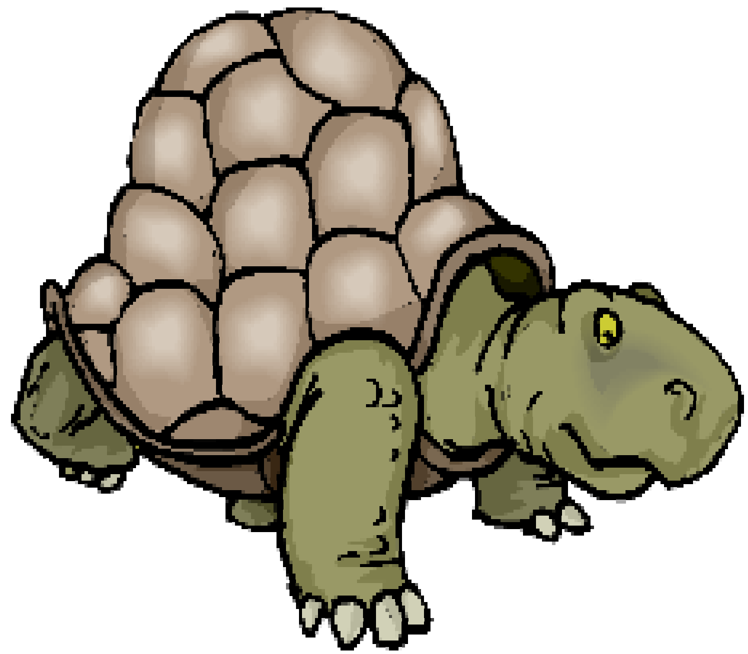
- Binary floating-point cannot be used for commercial or human-centric applications
  - cannot meet legal and financial requirements
- Decimal data and arithmetic are pervasive
- 55% of numeric data in databases are decimal (and a further 43% integers)

Why Floating-point Decimal?

- Traditional integer arithmetic with 'manual' scaling is awkward and error-prone
- Floating-point is increasingly necessary
  - interest calculated daily
  - telephone calls priced by the second
  - taxes more finely specified
  - financial analysis, *etc.*

So now we know what to do…

- but…



… but it's very, very slow …

For example, typical Java BigDecimal add is 1,708 cycles, hardware might take 8 cycles

	software penalty
add	210x – 560x
quantize	90x – 200x
multiply	40x – 190x
divide	260x – 290x

penalty = Java BigDecimal cycles ÷ DFPU clock cycles

Effect on Real Applications

- The ‘telco’ billing application
  1,000,000 calls (two minutes) read from file, priced, taxed, and printed

	C, C# packages	Java BigDecimal
% execution time in decimal operations	72 – 78%	93.2%

Effect on Real Applications [2]

- A "Web Warehouse" benchmark uses float binary for currency and tax calculations
- We added realistic decimal processing…

	orders per second
float binary	3,862
decimal	1,193

69% of workload is decimal arithmetic

Hardware is on the way…

- A 2x to 10x performance improvement in applications makes hardware support *very* attractive
- IBM is building Decimal Floating-Point (DFP) hardware into future processors
- Critical precondition was IEEE 754 Standardization — fortunately under revision (754r) since 2001

IEEE 754 Agreed Draft

- Now has decimal floating-point formats and arithmetic
  - suitable for mathematical applications, too
- Fixed-point and integer arithmetic are subsets (no normalization)
- Compression maximizes precision and exponent range of formats

IEEE 754r Formats

size (bits)	digits	exponent range
32	7	-95 to +96
64	16	-383 to +384
128	34	-6143 to +6144

The Java BigDecimal Class

java.math.BigDecimal

- In Java since 1.1 (JDBC)
  - BigInteger and int scale

- Arbitrary precision

```
BigDecimal z = x.multiply(y);
// 1.2 x 1.2 → 1.44
```

- Eight rounding modes

Constructors

- BigDecimal(String)
- BigDecimal(double)
  - exact conversion
- BigDecimal(BigInteger [, int])
- valueOf(long [, int])

Arithmetic

- add, subtract, multiply
- divide (with given rounding and scale)
- abs, negate
- compareTo, min, max
  (and equals, hashcode)

Miscellaneous

- signum (returns sign)
- scale, unscaledValue
- setScale (with rounding) = IEEE quantize

```
BigDecimal a=new BigDecimal("0.735");
setScale(a, 2, ROUND_HALF_EVEN);
```

➔ 0.74

- movePointLeft, movePointRight

Conversions

- toString, toBigInteger
- intValue, longValue (byteValue and shortValue inherited)
  - these quietly ***decapitate*** !
- floatValue, doubleValue

BigDecimal 1.1 Problems

- No rounding control; results get longer and longer (and slower)
- Dangerous when converting, no exponential
- Important methods missing (remainder, pow, round, *etc.*)
- Hard to use and not intuitive (esp. divide)

BigDecimal 5.0 Solution

- Core concept:
  Arithmetic operations depend on
  - numbers (many instances)
  - the context in which operations are effected
- This is mirrored by the implementation:
  - enhanced BigDecimal for the numbers; allows both positive and negative scale (*e.g.,* 1.3E+9)
  - new MathContext for the context

java.math.MathContext

- Immutable context object
- Allows future extensions
- Two properties:
  - precision (where to round)
  - rounding mode

```
new mathContext(7, RoundingMode.HALF_EVEN)
```

Using MathContext

```
BigDecimal A = new BigDecimal("2");
BigDecimal B = new BigDecimal("3");
MathContext mc = new MathContext(7,
         RoundingMode.HALF_EVEN);

BigDecimal C = A.divide(B, mc);
```

➔ C has the value 0.6666667

java.math.RoundingMode

- Immutable enumeration, with constants:
  UP, DOWN, CEILING, FLOOR,
  HALF_UP, HALF_DOWN, HALF_EVEN,
  UNNECESSARY
- equals, hashcode, toString, valueOf(String)
- Old rounding mode int constants are still in BigDecimal

New Constructors

- BigDecimal(int), BigDecimal(long)
- BigDecimal(char[ ]) … and sub-array
- All old and new constructors may take a MathContext, for rounding on construction
- New valueOf(double) … rounds as Double

New Arithmetic

- New methods: simpler divide, remainder, divideToIntegralValue, divideAndRemainder, pow, plus, round
- All arithmetic methods may take a MathContext
- setScale may now take a RoundingMode

Miscellaneous

- New methods:
  - precision
  - ulp (unit in last place)
  - scaleByPowerOf10(int)
  - stripTrailingZeros
- Useful constants
  - ZERO
  - ONE
  - TEN

New Conversions

- String/char constructors accept E+n *etc.*
- toEngineeringString for exponent is multiple of 3 (12.3E+6 rather than 1.23E+7), and new toCharArray for efficiency
- Exact, safe, integer conversions:
  toBigIntegerExact, intValueExact, longValueExact, shortValueExact, byteValueExact

Performance

- The internal representation (binary BigInteger) is inherently slow for conversions and rounding (base change)
- However, the class will be able to take advantage of hardware DFP without recompilation
- Especially if use right-size MathContext

Preferred MathContexts

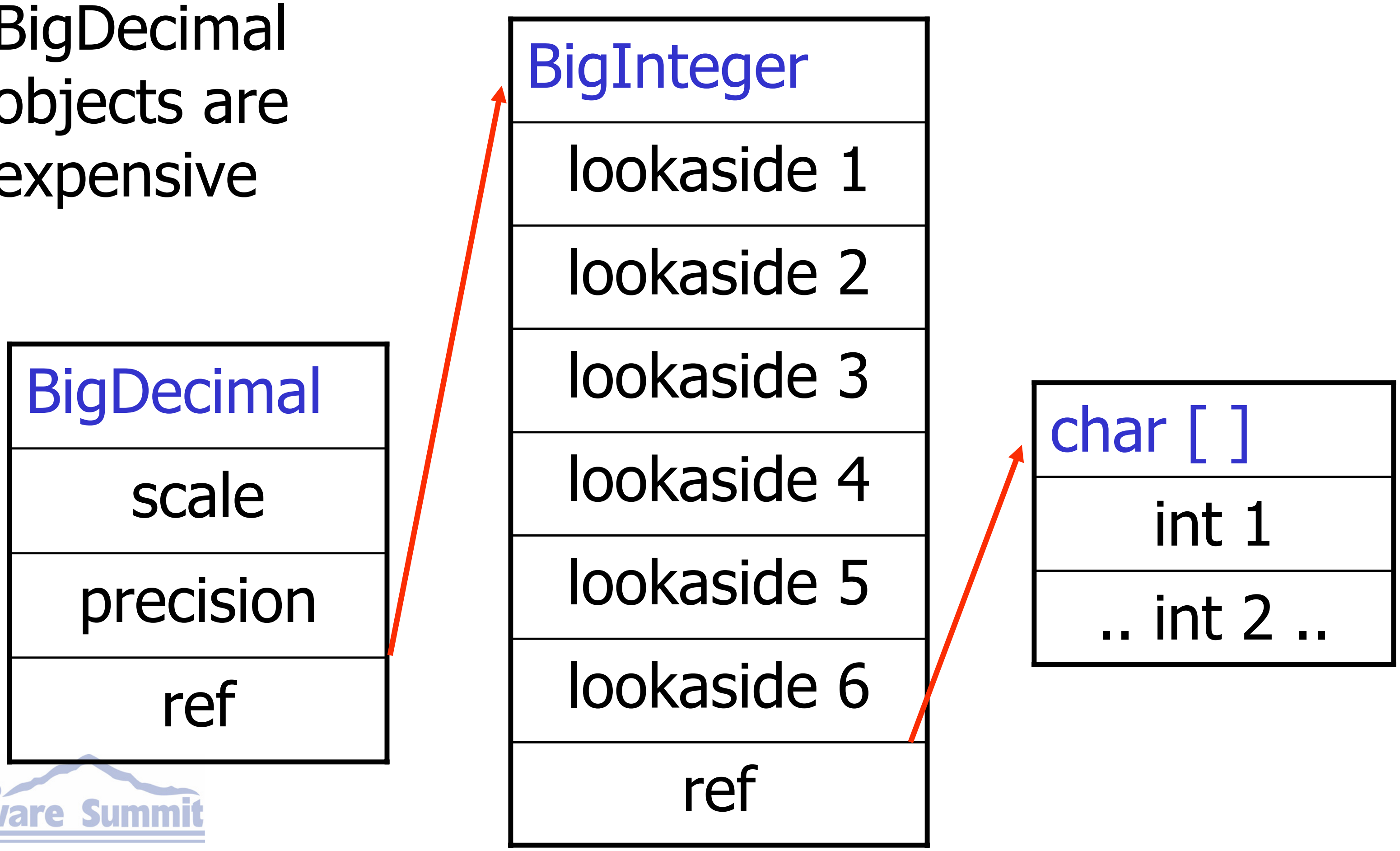
- The MathContext class provides contexts which match the IEEE 754r sizes and default rounding (HALF_EVEN):

  DECIMAL32 (7 digits)
  DECIMAL64 (16 digits)
  DECIMAL128 (34 digits)

- (Also UNLIMITED, to match old behavior)

Taking Advantage of Hardware

- BigDecimal objects are expensive



Taking Advantage of Hardware

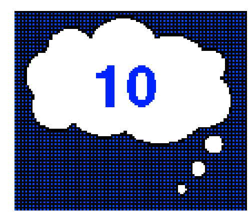
- BigDecimal with hardware lookaside (managed by the JIT compiler)
- BigInteger only created when number is too large for hardware (rare)

BigDecimal
scale
precision
(ref)
Hardware lookaside storage

Summary

- Major enhancements to the BigDecimal class make it much more useful and easier to use
- New MathContext and RoundingMode classes give better control of arithmetic
- Hardware on the way will dramatically improve performance

Questions?



Google: decimal arithmetic

How Computers Compute

- Binary arithmetic will continue to be used, but, perhaps …

  "in the relatively distant future, the continuing decline in the cost of processors and of memory will result (in applications intended for human interaction) in the displacement of substantially all binary floating-point arithmetic by decimal"

Professor W. Kahan, UCB

IEEE 754r Format Details

Common ‘shape’ for All Formats

Sign	Comb. field	Exponent	Coefficient

- Sign and combination field fit in first byte
  - combination field (5 bits) combines 2 bits of the exponent (0–2), first digit of the coefficient (0–9), and the two special values
  - allows ‘bulk initialization’ to zero, NaNs, and ± Infinity by byte replication

Exponent continuation

Sign	Comb. field	Exponent	Coefficient

Format	exponent bits	bias	normal range
32-bit	2+6	101	-95 to +96
64-bit	2+8	398	-383 to +384
128-bit	2+12	6176	-6143 to +6144

(All ranges larger than binary in same format.)

Coefficient continuation

Sign	Comb. field	Exponent	Coefficient

- Densely Packed Decimal – 3 digits in each group of 10 bits (6, 15, or 33 in all)
- Derived from Chen-Ho encoding, which uses a Huffman code to allow expansion or compression in 2–3 gate delays