Using Java 5.0 BigDecimal

Mike Cowlishaw
IBM Fellow

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

byte, short, int, long, etc.

FPU

Floating-point arithmetic (IEEE 754)

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]

<table>
<thead>
<tr>
<th>Decimal</th>
<th>Java float (binary)</th>
</tr>
</thead>
<tbody>
<tr>
<td>9</td>
<td>9</td>
</tr>
<tr>
<td>0.9</td>
<td>0.9</td>
</tr>
<tr>
<td>0.09</td>
<td>0.0899999996</td>
</tr>
<tr>
<td>0.009</td>
<td>0.009</td>
</tr>
</tbody>
</table>
Where It Costs Real Money…

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

- $1.05 \times 0.70$ using binary double is exactly
  
  \[ 0.7349999999999998667732370449812151491641998291015625 \]

  (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

<table>
<thead>
<tr>
<th></th>
<th>software penalty</th>
</tr>
</thead>
<tbody>
<tr>
<td>add</td>
<td>210x – 560x</td>
</tr>
<tr>
<td>quantize</td>
<td>90x – 200x</td>
</tr>
<tr>
<td>multiply</td>
<td>40x – 190x</td>
</tr>
<tr>
<td>divide</td>
<td>260x – 290x</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th></th>
<th>C, C# packages</th>
<th>Java BigDecimal</th>
</tr>
</thead>
<tbody>
<tr>
<td>% execution time in decimal operations</td>
<td>72 – 78%</td>
<td>93.2%</td>
</tr>
</tbody>
</table>
Effect on Real Applications

A “Web Warehouse” benchmark uses float binary for currency and tax calculations

- We added realistic decimal processing...

<table>
<thead>
<tr>
<th></th>
<th>orders per second</th>
</tr>
</thead>
<tbody>
<tr>
<td>float binary</td>
<td>3,862</td>
</tr>
<tr>
<td>decimal</td>
<td>1,193</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>size (bits)</th>
<th>digits</th>
<th>exponent range</th>
</tr>
</thead>
<tbody>
<tr>
<td>32</td>
<td>7</td>
<td>-95 to +96</td>
</tr>
<tr>
<td>64</td>
<td>16</td>
<td>-383 to +384</td>
</tr>
<tr>
<td>128</td>
<td>34</td>
<td>-6143 to +6144</td>
</tr>
</tbody>
</table>
The Java BigDecimal Class
In Java since 1.1 (JDBC)
  - BigInteger and int scale

Arbitrary precision

```java
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`
  ```java
  BigDecimal a=new BigDecimal("0.735");
  a.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

```java
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

```
<table>
<thead>
<tr>
<th>BigDecimal</th>
</tr>
</thead>
<tbody>
<tr>
<td>scale</td>
</tr>
<tr>
<td>precision</td>
</tr>
<tr>
<td>ref</td>
</tr>
</tbody>
</table>
```

```
| BigInteger |
|           |
| lookaside 1|
| lookaside 2|
| lookaside 3|
| lookaside 4|
| lookaside 5|
| lookaside 6|

| char [ ] |
| int 1    |
| .. int 2 ..|

ref
```
Taking Advantage of Hardware

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

<table>
<thead>
<tr>
<th>Sign</th>
<th>Comb. field</th>
<th>Exponent</th>
<th>Coefficient</th>
</tr>
</thead>
</table>

- 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**

<table>
<thead>
<tr>
<th>Format</th>
<th>exponent bits</th>
<th>bias</th>
<th>normal range</th>
</tr>
</thead>
<tbody>
<tr>
<td>32-bit</td>
<td>2+6</td>
<td>101</td>
<td>-95 to +96</td>
</tr>
<tr>
<td>64-bit</td>
<td>2+8</td>
<td>398</td>
<td>-383 to +384</td>
</tr>
<tr>
<td>128-bit</td>
<td>2+12</td>
<td>6176</td>
<td>-6143 to +6144</td>
</tr>
</tbody>
</table>

(All ranges larger than binary in same format.)
Coefficient continuation

<table>
<thead>
<tr>
<th>Sign</th>
<th>Comb. field</th>
<th>Exponent</th>
<th>Coefficient</th>
</tr>
</thead>
</table>

- 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