JavaOne℠

Sun's 2004 Worldwide Java Developer Conference℠

Big News for `BigDecimal`

New features from JSR 13

java.sun.com/javaone/sf

Joseph D. Darcy
Java Floating-Point Czar, Sun Microsystems
Mike Cowlishaw
IBM Fellow, IBM

Sun microsystems

Overview

Understand new floating-point functionality added to **`BigDecimal`**

Outline

Computational Background

Additions from JSR 13

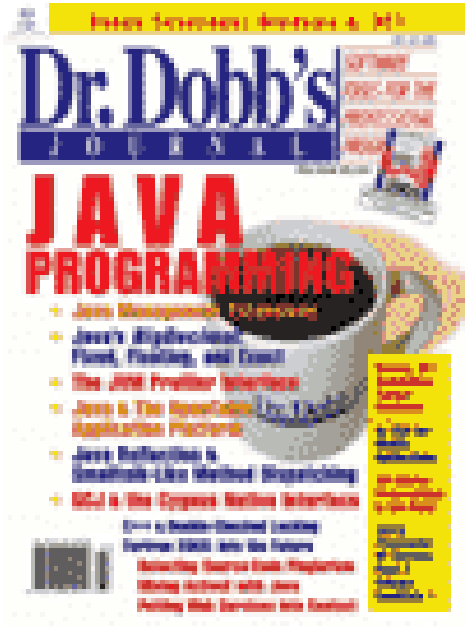
Compatibility

Exact Divide

Other Numerical Features

`BigDecimal` in the Press

- *Fixed, Floating, and Exact Computation with Java's BigDecimal*
  Mike Cowlishaw, Joshua Bloch, Joseph D. Darcy
  Dr. Dobb's Journal, July 2004



Representable values

General: $significand \times base^{exponent}$

Binary: $significand \times 2^{exponent}$

Decimal: $significand \times 10^{exponent}$

- *significand*, *exponent* integers
- Integers can be exactly represented in any integer base
- Representable fractional values differ by base
  - All base-2 fractions are also base-10 fractions
  - 1/10 has no terminating base-2 fraction
- “Everyday” fractions can be represented in decimal

Styles of Computation

- Pre-Tiger **`BigDecimal`** supported two styles of computation
  - Exact
  - Rounding to a chosen *scale*
- Scale is the number of digits to the right of the decimal point
  - Scale is negation of the exponent
- All integral values had scale zero

Same value, different scale

- **`BigDecimal`** values are not normalized
- Multiples numbers with magnitude ½
  - 0.5 significand = 5 scale = 1
  - 0.50 significand = 50 scale = 2
  - 0.500 significand = 500 scale = 3
- Different forms of ½ are *not* **`.equals`** but are **`compareTo == 0`**
- Suitable for *fixed point* computation

Fixed Point Example

- Sales tax calculation

```
BigDecimal price   = new BigDecimal("19.95");
BigDecimal taxRate = new BigDecimal("1.0825");

BigDecimal total =
        price.multiply(taxRate).
        setScale(2, ROUND_DOWN);

total.toString();   // 21.59
```

Floating-point style

- Floating-point is fixed *precision* arithmetic
- Decimal point “floats” among the preserved digits
- Benefits of floating-point
  - Cost of an operation is roughly constant (depending on precision)
  - Operates on values with vastly different magnitudes

Adding Floating-Point to `BigDecimal`

- Need arithmetic operators that round to a given number of digits
  - **`java.math.MathContext`**
    - Precision
    - **`java.math.RoundingMode`**
  - **`add(BigDecimal augend, MathContext mc)`**
- Need to represent integers *without* explicit trailing zeros
  - Now scales can be negative
  - Different representations of 1 million

```
1000000e0     // scale 0, only one possible pre-Tiger
10000e2       // scale -2
1e6           // scale -6
```

Summary of New Methods, Constructors

- Constructors
  - From **`char`** arrays
  - Including **`MathContext`** parameter
- Arithmetic
  - {+, -, *, /} methods taking **`MathContext`** parameter
  - **`scaleByPowerOfTen`**
  - Exact divide, divide to integral value
  - Remainder
  - Pow (integer power)
  - Exact conversions
- Precision

Computation Example

```
static BigDecimal sum(BigDecimal bds[]) {
  BigDecimal sum = BigDecimal.ZERO;
  MathContext mc =
     new MathContext(17,HALF_DOWN);
  for(BigDecimal bd: bds) {
      sum = sum.add(bd, mc);
  }
  return sum;
}
```

Cohorts

- When a numerical value has multiple representations, set of numerically equal values is a *cohort*
- Floating-point operations have *preferred scale* rules to select cohort member
  - Only used when result is exact
  - Add: `max( addend.scale(), augend.scale())`
  - Multiply: `multiplier.scale() + multiplicand.scale()`

Compatibility

- Arithmetic properties
- Set of representable values
  - Text representation
  - `toString` behavior

Arithmetic Properties

- Old
  `c1 = a1.add(b1)`
- New
  `c2 = a2.add(b2)`
- If `a1.equals(a2)` and `b1.equals(b2)`, then `c1.equals(c2)`
- If `a1.compareTo(a2) == 0` and `b1.compareTo(b2) == 0`, then `c1.compareTo(c2) == 0`

Ins and Outs of Conversions

- String constructor: values with negative scales can now be created
  - How to get old behavior:

```
BigDecimal bd =
          new BigDecimal(myString);
if (bd.scale() < 0)
  bd = bd.setScale(0);
```

  - 0 scale can create much larger **`BigInteger`** component!

Ins and Outs of Conversions, Cont.

- Pre-Tiger `toString` method doesn't use exponential notation
  - Lots of leading and trailing zeros
- For positive scales
  - length of output grows linearly with size of exponent; *not* with number of decimal digits in the integer value
- Zero scale output grows with the magnitude of the value
  - No way to suppress trailing zeros
- Without an exponent, no obvious extension to indicate a negative scale

Revised `toString` method

- Scientific notation output with exponent
  - elides exponent if the output only needs a few digits and the digits are near zero
- Provides clear, concise correspondence to
  - numerical value
  - (**`BigInteger`**, scale)
- New output is legal pre-Tiger string constructor input
- To get old style output: **`toPlainString`**

Strings across versions

- With new and old **`BigDecimal`** classes: **`BigDecimal(bd.toString()).equals(bd)`**
- Pre-Tiger **`toString`** output read in current string constructor gives a **`.equals`** result
  - numerical value *and* representation are preserved
- Current **`toString`** output read in with pre-Tiger string constructor
  - values with positive scales will be **`.equals`**
  - values with negative scales get mapped to numerically equal scale 0 value (could overflow)

`BigDecimal`, `float`, and `double`

- **`BigDecimal`**
  - High precision
  - Large exponent range
  - Full control over rounding behavior
  - Transparent string ↔ **`BigDecimal`** conversion
- **`float`** and **`double`**
  - Operators and math library
  - Common hardware support
  - Less memory usage

Type Numerical Properties

Type	Exponent Range	Precision
`float`	2^{-149} to 2^{127} ≈ 10^{-45} to 10^{38}	24 bits ≈ 6 to 9 digits
`double`	2^{-1074} to 2^{1023} ≈ 10^{-324} to 10^{308}	54 bits ≈ 15 to 17 digits
`BigDecimal`	$10^{-2147483647}$ to $10^{2147483648}$	1 to billions of digits

Exacting Divide

- Pre-Tiger, exact methods for add, subtract, multiply
- Division always needed some rounding information
- Can have exact divide too!
- How to tell if quotient is a repeating fraction?
  - 1/3
  - 29/33
- Derive upper bound on digits in exact quotient if exact quotient exists

Preliminaries

- Rational number:
  $\frac{a}{b}$
- Reduce to lowest terms:
  $\frac{a'}{b'}$
- Example:
  $\frac{76}{100} \rightarrow \frac{17}{25}$

Necessary and Sufficient

- Claim: a'/b' has a terminating decimal expansion if and only if

  $b' = 2^i \cdot 5^j$

  for integral $i, j \geq 0$
- Why?
  - 0.34902 =

$$\frac{34902}{100000} = \frac{34902}{10^6} = \frac{17451}{2^5 \cdot 5^6}$$

  - Therefore, all terminating fractions have a denominator of the required form

Denominator to Termination

- Prove
  $$\frac{a'}{2^i \cdot 5^j}$$
  has a terminating expansion
- Transform to denominator that is a power of 10
  - If $i > j$, multiply by 5^{i-j}
    $$\frac{a'}{2^i \cdot 5^j} \frac{5^{i-j}}{5^{i-j}} = \frac{a'5^{i-j}}{2^i \cdot 5^{i-j}} = \frac{a'5^{i-j}}{10^i}$$
- Therefore, if denominator is of the desired form, the fraction has a terminating expansion

Getting a bound

- The number of digits in
  c/d
  will be the same as the number of digits in
  $c * (1/d)$
- Therefore, just need to get bound for
  $1/(2^i \cdot 5^j)$
- If $i == j$, only need one digit
- Difference in i and j is what matters

Powers of 2 and 5

- $1/2$ $1/2$ 0.5 $5 = 5$
  $1/2^2$ $1/4$ 0.25 $5^2 = 25$
  $1/2^3$ $1/8$ 0.125 $5^3 = 125$
- $1/5$ $1/5$ 0.2 $2 = 2$
  $1/5^2$ $1/25$ 0.04 $2^2 = 4$
  $1/5^3$ $1/125$ 0.008 $2^3 = 8$
- These observations + algebra can give a bound on the digits of the reciprocal

The Bound

- max(ceil((i-j)·$\log_{10}(5)$),
  ceil((j-i)·$\log_{10}(2)$),
  1) ≤
- max(ceil((i-j)·0.7),
  ceil((j-i)·0.302),
  1)
- Without factoring information:
  ceil(10·precision(*b*)/3) ≤ 4·precision(*b*)
- Number of digits in exact quotient of *a*/*b*:
  precision(*a*) + ceil(10·precision(*b*)/3))

Other fun Tiger math features: Integers

- Bit twiddling methods in **`Integer`**, **`Long`**
  - **`bitCount`**, **`highestOneBit`**, **`lowestOneBit`**
  - **`numberOfLeadingZeros`**, **`numberOfTrailingZeros`**
  - **`reverse`**, **`reverseBytes`**
  - **`rotateLeft`**, **`rotateRight`**
- Useful in cryptography, etc.
  - See Henry S. Warren, Jr.'s *Hacker's Delight*

Fun floating-point functionality

- New methods in **`Math`**, **`StrictMath`**
  - **`cbrt`**, **`hypot`**, **`log10`**
  - **`cosh`**, **`sinh`**, **`tanh`**
  - **`expm1`**, **`log1p`**
  - **`signum`**, **`ulp`**
- Latest fdlibm version
- Hexadecimal floating-point values
  - Literals: **`0x1.fffffffffffffp+1023`**
    `0x0.0000000000001p-1022`
  - **`toHexString`**, **`parseDouble`**, **`printf %a`**

The Future

- More “math library” methods for **`BigDecimal`**?
  - sqrt
  - pow
  - sin, cos, tan hard
- More methods in **`Math`**, **`StrictMath`**?
  - Fused multiply accumulate
  - log2, exp2
- Unsigned integer library support?

Summary

- **`BigDecimal`** now supports floating-point!
- Many other mathematical additions in Tiger

Q&A

JavaOne℠

Sun's 2004 Worldwide Java Developer Conference℠

Big News for `BigDecimal`

New features from JSR 13

java.sun.com/javaone/sf

Sun microsystems

Joseph D. Darcy
Java Floating-Point Czar, Sun Microsystems
Mike Cowlishaw
IBM Fellow, IBM