Java's new math, Part 2: Floating-point numbers

Elliotte Rusty Harold January 13, 2009

Join Elliotte Rusty Harold for a look into "new" features in the classic `java.lang.Math` class in this two-part article. Part 1 focuses on more purely mathematical functions. Part 2 explores the functions designed for operating on floating-point numbers.

View more content in this series

Version 5 of the Java™ Language Specification added 10 new methods to `java.lang.Math` and `java.lang.StrictMath`, and Java 6 added another 10. Part 1 of this two-article series looked at the new methods that make sense mathematically. That is, they provide functions that a pre-computer-era mathematician would find familiar. Here in Part 2, I focus on the functions that make sense only when you realize that they're designed for operating on floating-point numbers instead of abstract real numbers.

As I noted in Part 1, the distinction between a real number such as *e* or 0.2 and its computer representation such as a Java `double` is an important one. The Platonic ideal of the number is infinitely precise, whereas the Java representation has only a fixed number of bits (32 for a `float`, 64 for a `double`) to work with. The maximum value of a `float` is about $3.4*10^{38}$, which isn't large enough for some things you might wish to represent, such as the number of electrons in the universe.

A `double` can represent numbers up to about $1.8*10^{308}$, which covers almost any physical quantity I can think of. However, when you do calculations on abstract mathematical quantities, it is possible to exceed these values. For instance, a mere 171! (171 * 170 * 169 * 168 * ... * 1) is sufficient to exceed the bounds of a `double`. A `float` goes out of bounds at a mere 35!. Small numbers (that is, numbers close to zero) can also be problematic, and calculations that involve both large and small numbers can be positively dangerous.

To work around this, the IEEE 754 standard for floating-point math (see Related topics) adds the special values Inf to represent Infinity and NaN to represent "Not a Number." IEEE 754 also defines both positive and negative zeros. (In regular math, zero is neither positive nor negative. In computer math, it can be either.) These values play havoc with classical proofs. For instance, when NaN is in play, the law of the excluded middle no longer holds. It is not necessarily true that either x == y or x != y. Both can be false if x (or y) is NaN.

Beyond issues of magnitude, precision is an even more practical problem. We've all seen a loop like this one where you add 0.1 one hundred times and end up with 9.99999999999998 instead of 10:

```
for (double x = 0.0; x <= 10.0; x += 0.1) {
    System.err.println(x);
}
```

For simple applications, you usually just ask `java.text.DecimalFormat` to format the final output as the nearest integer, and call it a day. However, in scientific and engineering applications where you're not sure if the calculation will land on an integer, you need to be a lot more careful. If you're subtracting large numbers from each other to get a small number, you need to be *very* careful. If you're dividing by that small number, you need to be more careful still. Such operations can dramatically amplify even tiny errors into large mistakes that have noticeable consequences when the answers are applied in the physical world. Mathematically precise calculations can be thrown severely askew by small round-off errors caused by finite-precision floating-point numbers.

Binary representations of floats and doubles

An IEEE 754 float, as implemented by the Java language, has 32 bits. The first bit is the sign bit, 0 for positive and 1 for negative. The next eight bits are the exponent and can hold a value from -125 to +127. The final 23 bits hold the mantissa (sometimes called the significand), which ranges from 0 to 33,554,431. Putting these together, a float is interpreted as $sign * mantissa * 2^{exponent}$.

Observant readers may notice that these numbers don't quite add up. First, eight bits for the exponent should represent -128 to 127, just like a signed byte. However the exponents are biased by 126. That is, you start with the unsigned value (0 to 255) and then subtract 126 to get the true exponent, which is now -126 to 128. Well, except that 128 and -126 are special values. When the exponent is all 1-bits (128), that's a signal that the number is either Inf, -Inf, or NaN. To figure out which, you have to look at the mantissa. When the exponent is all zero bits (-126), that's a signal that the number is *denormalized* (more on what that means shortly) but the exponent is still -125.

The mantissa is basically a 23-bit unsigned integer — simple enough. Twenty-three bits can hold a number from 0 to 2^{24}-1, which is 16,777,215. Wait a minute, didn't I say that the mantissa ranged from 0 to 33,554,431? That's 2^{25}-1. Where'd the extra bit come from?

It turns out that you can use the exponent to tell what the first bit is. If the exponent is all zero bits, then the first bit is zero. Otherwise, the first bit is one. Because you always know what the first bit is, it doesn't have to be included in the number. You get an extra bit for free. Pretty sneaky, huh?

Floating-point numbers for which the first bit of the mantissa is one are *normalized*. That is, the mantissa always has a value between 1 and 2. Floating-point numbers for which the first bit of the mantissa is zero are *denormalized* and enable much smaller numbers to be represented, even though the exponent is always -125.

Doubles are encoded in much the same way except that they use a 52-bit mantissa and an 11-bit exponent for higher precision. The bias for a double's exponent is 1023.

Mantissas and exponents

The two `getExponent()` methods added in Java 6 return the *unbiased* exponent used in the representation of the float or double. This is a number between -125 to +127 for floats and -1022 and +1023 for doubles (+128/+1024 for Inf and NaN). For example, Listing 1 compares the results of the `getExponent()` method to a more classic base 2 logarithm:

Listing 1. `Math.log(x)/Math.log(2)` vs. `Math.getExponent()`

```
public class ExponentTest {

    public static void main(String[] args) {
       System.out.println("x\tlg(x)\tMath.getExponent(x)");
       for (int i = -255; i < 256; i++) {
           double x = Math.pow(2, i);
           System.out.println(
                   x + "\t" +
                   lg(x) + "\t" +
                   Math.getExponent(x));
       }
    }

    public static double lg(double x) {
        return Math.log(x)/Math.log(2);
    }
}
```

For a few values where round-off comes into play, `Math.getExponent()` can be a bit or two more accurate than the usual calculation:

```
 x              lg(x)             Math.getExponent(x)
...
2.68435456E8    28.0                        28
5.36870912E8    29.000000000000004          29
1.073741824E9   30.0                        30
2.147483648E9   31.000000000000004          31
4.294967296E9   32.0                        32
```

`Math.getExponent()` may also be faster if you're doing a lot of these calculations. However, the caveat is that this only works for powers of two. For example, here's the output if I change to powers of three:

```
x      lg(x)     Math.getExponent(x)
...
1.0    0.0                 0
3.0    1.584962500721156   1
9.0    3.1699250014423126  3
27.0   4.754887502163469   4
81.0   6.339850002884625   6
```

The mantissa is not considered by `getExponent()` but is considered by `Math.log()`. With some effort, you could separately find the mantissa, take its log, and add that value to the exponent, but that's hardly worth the effort. `Math.getExponent()` is primarily useful when you want a quick estimate of an order of magnitude, not an exact value.

Unlike `Math.log()`, `Math.getExponent()` never returns NaN or Inf. If the argument is NaN or Inf, then the result will be 128 for a float and 1024 for a double. If the argument is zero, then the result

will be -127 for a float and -1023 for a double. If the argument is a negative number, then the exponent is the same as the exponent of the absolute value of that number. For instance, the exponent of -8 is 3, just like the exponent of 8.

There's no corresponding `getMantissa()` method, but it's easy enough to derive one with a little algebra:

```
    public static double getMantissa(double x) {
        int exponent = Math.getExponent(x);
        return x / Math.pow(2, exponent);
    }
```

The mantissa can also be found through bit masking, though the algorithm is far less obvious. To extract the bits, you only need to calculate `Double.doubleToLongBits(x) & 0x000FFFFFFFFFFFFL`. However, you then need to account for the extra 1-bit in a normalized number, and then convert back to a floating-point number between 1 and 2.

Units of last precision

Real numbers are infinitely dense. There's no such thing as a next real number. For any two distinct real numbers you name, I can name another one in between them. The same is not true for floating-point numbers. Given one float or double, there is a next float; and there is a minimum finite distance between successive floats and doubles. The `nextUp()` method returns the nearest floating-point number greater than the first argument. For example, Listing 2 prints all the floats between 1.0 and 2.0 inclusive:

Listing 2. Counting the floats

```
public class FloatCounter {

    public static void main(String[] args) {
        float x = 1.0F;
        int numFloats = 0;
        while (x <= 2.0) {
            numFloats++;
            System.out.println(x);
            x = Math.nextUp(x);
        }
        System.out.println(numFloats);
    }

}
```

It turns out there are exactly 8,388,609 floats between 1.0 and 2.0 inclusive; large but hardly the uncountable infinity of real numbers that exist in this range. Successive numbers are about 0.0000001 apart. This distance is called an *ULP* for *unit of least precision* or *unit in the last place*.

If you need to go backwards — that is, find the nearest floating-point number less than a specified number — you can use the `nextAfter()` method instead. The second argument specifies whether to find the nearest number above or below the first argument:

```
public static double nextAfter(float start, float direction)
public static double nextAfter(double start, double direction)
```

If `direction` is greater than `start`, then `nextAfter()` returns the next number above `start`. If `direction` is less than `start`, `nextAfter()` returns the next number below `start`. If `direction` equals `start`, `nextAfter()` returns `start` itself.

These methods can be useful in certain modeling and graphing applications. Numerically, you might want to sample a value at 10,000 positions between *a* and *b*, but if you're getting only enough precision to identify 1,000 unique points between *a* and *b*, then you're doing redundant work nine times out of ten. You can do a tenth of the work, and get results that are just as good.

Of course, if you really do need the extra precision, then you'll need to pick a data type with more precision, such as a `double` or a `BigDecimal`. For instance, I've seen this come up in Mandelbrot set explorers where you can zoom in so far that the entire graph falls between the nearest two doubles. The Mandelbrot set is infinitely deep and complex at all scales, but a `float` or a `double` can go only so deep before losing the ability to distinguish adjacent points.

The `Math.ulp()` method returns the distance from a number to its nearest neighbors. Listing 3 lists the ULPs for various powers of two:

Listing 3. ULPs of powers of two for a float

```
public class UlpPrinter {

    public static void main(String[] args) {
        for (float x = 1.0f; x <= Float.MAX_VALUE; x *= 2.0f) {
            System.out.println(Math.getExponent(x) + "\t" + x + "\t" + Math.ulp(x));
        }
    }

}
```

Here's some of the output:

```
0   1.0   1.1920929E-7
1   2.0   2.3841858E-7
2   4.0   4.7683716E-7
3   8.0   9.536743E-7
4   16.0  1.9073486E-6
...
20  1048576.0   0.125
21  2097152.0   0.25
22  4194304.0   0.5
23  8388608.0   1.0
24  1.6777216E7 2.0
25  3.3554432E7 4.0
...
125 4.2535296E37    5.0706024E30
126 8.507059E37     1.0141205E31
127 1.7014118E38    2.028241E31
```

The finite precision of floating-point numbers has one unexpected consequence: past a certain point x+1 == x is true. For instance, this apparently simple loop is in fact infinite:

```
for (float x = 16777213f; x <
  16777218f; x += 1.0f) {
    System.out.println(x);
}
```

In fact, this loop gets stuck at a fixed point at precisely 16,777,216. That's 2^{24}, and the point at which the ULP is now larger than the increment.

As you can see, floats are pretty accurate for small powers of two. However, the accuracy becomes problematic for many applications by around 2^{20}. Near the limit of magnitude for a float, successive values are separated by *sextillions* (in fact, quite a bit more, but I couldn't find a word that means anything that high).

As Listing 3 demonstrates, the size of an ULP is not constant. As numbers get larger, there are fewer floats between them. For instance, there are only 1,025 floats between 10,000 and 10,001; and they're 0.001 apart. Between 1,000,000 and 1,000,001 there are only 17 floats, and they're about 0.05 apart. Accuracy is anticorrelated with magnitude. For a float at 10,000,000, the ULP has actually grown to 1.0, and past that there are multiple integral values that map to the same float. For a double this doesn't happen until about 45 quadrillion (4.5E15), but it's still a concern.

The `Math.ulp()` method has a practical use in testing. As you undoubtedly know, you should usually not compare floating-point numbers for exact equality. Instead, you check that they are equal within a certain tolerance. For example, in JUnit you compare expected to actual floating-point values like so:

```
assertEquals(expectedValue, actualValue, 0.02);
```

This asserts that the actual value is within 0.02 of the expected value. However is 0.02 a reasonable tolerance? If the expected value is 10.5 or -107.82, then 0.02 is probably fine. However, if the expected value is several billion, then the 0.02 may be completely indistinguishable from zero. Often what you should test is the relative error in terms of ULPs. Depending on how much accuracy the calculation requires, you usually select a tolerance somewhere between 1 and 10 ULPs. For example, here I specify that the actual result needs to be within 5 ULPs of the true value:

```
assertEquals(expectedValue, actualValue, 5*Math.ulp(expectedValue));
```

Depending on what the expected value is, that could be trillionths or it could be millions.

`scalb`

`Math.scalb(x, y)` multiplies x by 2^y (`scalb` is an abbreviation for "scale binary").

```
public static double scalb(float f, int scaleFactor)
public static double scalb(double d, int scaleFactor)
```

For example, `Math.scalb(3, 4)` returns $3 * 2^4$, which is 3*16, which is 48.0. You could use `Math.scalb()` in an alternate implementation of `getMantissa()`:

```
public static double getMantissa(double x) {
    int exponent = Math.getExponent(x);
    return x / Math.scalb(1.0, exponent);
}
```

How does `Math.scalb()` differ from `x*Math.pow(2, scaleFactor)`? In fact, the ultimate result doesn't differ. I was unable to contrive any inputs where the return value was even a single bit different. However, the performance is worth a second look. `Math.pow()` is a notorious performance killer. It needs to be able to handle really weird cases like raising 3.14 to the -0.078 power. It usually chooses completely the wrong algorithm for small integral powers like two and three, or for special cases like a base of two.

As with any other general performance claim, I have to be very tentative about this. Some compilers and VMs are smarter than others. Some optimizers may recognize `x*Math.pow(2, y)` as a special case and convert it into `Math.scalb(x, y)` or into something very close to that. Thus there may be no performance difference at all. However, I have verified that at least some VMs are not that clever. When testing with Apple's Java 6 VM, for example, `Math.scalb()` was reproducibly two orders of magnitude faster than `x*Math.pow(2, y)`. Of course, usually this is not going to matter in the slightest. However, in those rare cases where you are doing many millions of exponentiations, you may want to think about whether you can convert them to use `Math.scalb()` instead.

`Copysign`

The `Math.copySign()` method sets the sign of the first argument to the sign of the second argument. A naive implementation might look like Listing 4:

Listing 4. A possible `copysign` algorithm

```
public static double copySign(double magnitude, double sign) {
    if (magnitude == 0.0) return 0.0;
    else if (sign < 0) {
      if (magnitude < 0) return magnitude;
      else return -magnitude;
    }
    else if (sign > 0) {
      if (magnitude < 0) return -magnitude;
      else return magnitude;
    }
    return magnitude;
}
```

However, the real implementation looks like Listing 5:

Listing 5. The real algorithm from `sun.misc.FpUtils`

```
public static double rawCopySign(double magnitude, double sign) {
    return Double.longBitsToDouble((Double.doubleToRawLongBits(sign) &
                                    (DoubleConsts.SIGN_BIT_MASK)) |
                                    (Double.doubleToRawLongBits(magnitude) &
                                    (DoubleConsts.EXP_BIT_MASK |
                                    DoubleConsts.SIGNIF_BIT_MASK)));
}
```

If you think about this carefully and draw out the bits, you'll see that NaN signs are treated as positive. Technically, `Math.copySign()` doesn't promise that — only `StrictMath.copySign()` does — but in practice, they both invoke the same bit-twiddling code.

Listing 5 may perhaps be marginally faster than Listing 4, but its main raison d'être is to handle negative zero properly. `Math.copySign(10, -0.0)` returns -10, whereas `Math.copySign(10, 0.0)` returns 10.0. The naive algorithm in Listing 4 returns 10.0 in both cases. Negative zero can arise when you perform sensitive operations such as dividing an extremely small negative double by an extremely large positive double. For instance, `-1.0E-147/2.1E189` returns negative zero, whereas `1.0E-147/2.1E189` returns positive zero. However, these two values compare equal with `==` so if you want to distinguish between them, you need to use `Math.copySign(10, -0.0)` or `Math.signum()` (which calls `Math.copySign(10, -0.0)`) to compare them.

Logarithms and exponentials

The exponential function serves as a good example of how careful you have to be when dealing with limited-precision floating-point numbers instead of infinitely precise real numbers. e^x (`Math.exp()`) shows up in a lot of equations. For example, it's used to define the cosh function as discussed in Part 1:

cosh(x) = (e^x + e^{-x})/2.

However, for negative values of x, roughly -4 and lower, the algorithm used to calculate `Math.exp()` is relatively ill-behaved and subject to round-off error. It's more accurate to calculate e^x - 1 with a different algorithm and then add 1 to the final result. The `Math.expm1()` method implements this different algorithm. (The `m1` stands for "minus 1.") For example, Listing 6 demonstrates a cosh function that switches between the two algorithms depending on the size of `x`:

Listing 6. A cosh function

```
public static double cosh(double x) {
    if (x < 0) x = -x;
    double term1 = Math.exp(x);
    double term2 = Math.expm1(-x) + 1;
    return (term1 + term2)/2;
}
```

This example is somewhat academic because the e^x term will completely dominate the e^{-x} term in any case where the difference between `Math.exp()` and `Math.expm1() + 1` is significant. However. `Math.expm1()` is quite practical in financial calculations with small amounts of interest, such as the daily rate on a Treasury bill.

`Math.log1p()` is the inverse of `Math.expm1()`, just as `Math.log()` is the inverse of `Math.exp()`. It calculates the logarithm of 1 plus its argument. (The `1p` stands for "plus 1.") Use this for values close to 1. For instance, instead of calculating `Math.log(1.0002)`, you should calculate `Math.log1p(0.0002)`.

As an example, suppose you want to know the number of days required for $1,000 invested to grow to $1,100 at a daily interest rate of 0.03. Listing 7 would do this:

Listing 7. Find the number of periods needed to achieve a specified future value from a current investment

```
public static double calculateNumberOfPeriods(
        double presentValue, double futureValue, double rate) {
    return (Math.log(futureValue) - Math.log(presentValue))/Math.log1p(rate);
}
```

In this case, the `1p` has a very natural interpretation, since 1+*r* shows up in the usual formulas for calculating these things. In other words, lenders usually quote interest rates as the additional percent (the +*r* part) even though investors certainly hope to get back $(1+r)^n$ of their initial investment. Indeed, any investor who loans money at 3 percent and ends up with only 3 percent of the investment back would be doing poorly indeed.

Doubles aren't real numbers

Floating-point numbers are not real numbers. There are a finite number of them. There are maximum and minimum values they can represent. Most important, they have limited, though large, precision and are subject to round-off error. Indeed, when working with integers, floats and doubles can have decidedly worse accuracy than ints and longs. You should carefully consider these limitations to produce robust, reliable code, particularly in scientific and engineering applications. Financial applications (and especially accounting applications that require accuracy to the last cent) also need to be exceedingly careful when manipulating floats and doubles.

The `java.lang.Math` and `java.lang.StrictMath` classes have been carefully designed to address these issues. Proper use of these classes and their methods will improve your programs. If nothing else, this article should have shown you just how tricky good floating-point arithmetic really is. It's better to delegate to the experts where you can rather than roll your own algorithms. If you can use the methods of `java.lang.Math` and `java.lang.StrictMath`, do so. They're almost always the better choice.

Related topics

- "Java's new math, Part 1: Real numbers" (Elliotte Rusty Harold, developerWorks, October 2008): Part 1 of this series covers more abstract mathematical methods in the `java.lang.Math` class.
- IEEE standard for binary floating-point arithmetic: The IEEE 754 standard defines floating-point arithmetic in most modern processors and languages, including the Java language.
- Types, Values, and Variables: Chapter 4 of the Java Language Specification defines the subset of IEEE 754 arithmetic used in the Java programing language.
- "Floating-point arithmetic" (Bill Venners, JavaWorld, October 1996): Venners explores floating-point arithmetic in the JVM and covers the bytecodes that perform floating-point arithmetic operations.
- Unit in the last place: Wikipedia's article on this topic is informative.
- java.lang.Math: Javadoc for the class that provides the functions discussed in this article.
- OpenJDK: Look into the source code of the math classes inside this open source Java SE implementation.
- Download IBM® product evaluation versions and get your hands on application development tools and middleware products from DB2®, Lotus®, Rational®, Tivoli®, and WebSphere®.