Java の新しい Math クラス: 第 2 回 浮動小数点数

Elliotte Rusty Harold
Software Engineer
Cafe au Lait

2009年 1月 13日

この 2 回シリーズの記事では、Elliotte Rusty Harold が昔ながらの java.lang.Math クラスの「新しい」機能について調べます。第 1 回では純粋に数学的な関数に焦点を当て、第 2 回では浮動小数点数の演算用に設計された関数について調べます。

このシリーズの他の記事を見る

Java™ 言語仕様のバージョン 5 では、`java.lang.Math` と `java.lang.StrictMath` に10 個の新しいメソッドが追加されており、Java 6 ではさらに 10 個のメソッドが追加されています。この 2 回シリーズの第 1 回では、数学的な意味を持つ新しいメソッド、つまりコンピューターが登場する以前の時代の数学者にとっておなじみの関数について調べました。この第 2 回で焦点をあてるのは、数学的な概念上のいわゆる実数の演算用に設計された関数ではなく、コンピューターが数値を扱う際の表現方法である浮動小数点数の演算用に設計された関数であることを認識しないと理解のできない関数です。

第 1 回で説明したとおり、e や 0.2 などの実数と、その実数をコンピューターで表現したもの (例えば Java の `double` など) の違いは重要です。理想的な観念としての数値の精度は無限ですが、その数値を Java で表現した場合には固定のビット数 (`float` では 32 ビット、`double` では 64 ビット) でしか扱うことができません。`float` の最大値は約 $3.4*10^{38}$ ですが、これは例えば宇宙空間の電子の数を表現する場合のように、表現する対象によっては十分ではありません。

`double` は約 $1.8*10^{308}$ までの数を表現することができ、私が知る限りの物理量のほとんどすべてをカバーすることができます。しかし数学上の抽象的な数量を計算する際には、double で表現できる数値を超えてしまうことがありえます。例えば、171! (171 * 170 * 169 * 168 * ... * 1) を計算するだけでも `double` の限界を超えてしまいます。`float` の場合でさえ、35! を計算するだけで限界を超えます。小さな数 (つまりゼロに近い数) の場合にも同じような問題があるため、非常に大きな数を計算する場合にも、極めて小さな数を計算する場合にも、正しい結果を得られない可能性が非常に高くなります。

この問題に対応するために、浮動小数点数演算のための IEEE 754 標準 (「参考文献」を参照) では特別な値として、Infinity (無限) を表現する Inf と、Not a Number (数値ではない) を表現する NaN が追加されています。また IEEE 754 では正のゼロと負のゼロの両方も定義されています。(通常の数学ではゼロは正でも負でもありませんが、コンピューターの計算ではどちらもあり得ます。) これらの値は古典的な証明方法を成り立たなくします。例えば NaN を扱う場合には排中律が成立し

ません。必ずしも x == y または x != y のいずれかが真であるとは限らず、x (または y) が NaN の場合には両方とも偽である可能性があります。

桁数の問題よりも、現実的には精度の方が重要な問題です。私達全員が経験しているように、下記のようなループで 0.1 の加算を 100 回行うと、結果は 10 ではなく 9.99999999999998 になります。

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

計算結果を整数で得ればよいような単純なアプリケーションでは、通常は `java.text.DecimalFormat` 命令を使えば、最終的な出力を計算結果に最も近い整数のフォーマットで得ることができ、それですべてが完了します。しかし、科学アプリケーションや工学アプリケーションでは計算結果が整数とは限らないため、通常以上に注意する必要があります。非常に大きな数同士を引き算して小さな数を得る場合には、非常に注意する必要があります。その小さな数を使って割り算をする場合には、さらにそれ以上に注意する必要があります。こうした演算では、ごくわずかな誤差も大きな誤りへと増幅されてしまい、そうした誤りを含んだ解が物理的な世界に適用されると、目に見える結果となって現れます。数学的な精度を必要とする計算が、有限精度の浮動小数点数によって引き起こされる小さな丸め誤差のおかげで、大幅に違った結果を生じてしまうこともあります。

浮動小数点数と倍精度数のバイナリー表現

Java 言語で実装した IEEE 754 の浮動小数点数は 32 ビットです。最初のビットは符号ビットであり、0 が正、1 が負です。次の 8 ビットは指数部であり、-125 から +127 までの値を持つことができます。最後の 23 ビットは仮数部であり、範囲は 0 から 33,554,431 です。これをまとめると、浮動小数点数の表現は、$## * ### * 2^{###}$ のようになります。

注意深い読者は、この説明では数字のつじつまが合わないことに気付いたのではないでしょうか。そもそも、指数部用の 8 ビットは符号付きのバイトと同じように -128 から 127 を表現するはずです。しかし指数は 126 だけバイアスされています。つまり符号なしの値 (0 から 255) で開始し、次に 126 を引いて真の指数を得ます。そのため指数は -126 から 128 になります。ただし 128 と -126 は特別な値として除きます。指数部のビットがすべて 1 の場合 (128) には、その数が Inf、-Inf、NaN のいずれかであることを表します。そのいずれであるかを判断するためには仮数部を調べる必要があります。指数部のビットがすべて 0 の場合 (-126) には、その数が正規化されていないことを表します (これについては、このすぐ後に説明します)。ただしそれでも指数は -125 です。

仮数部は基本的に 23 ビットの符号なしの整数であり、ごく単純です。23 ビットによって、0 から 2^{24}-1 (16,777,215) までを表現することができます。しかし、ちょっと待ってください。仮数部は 0 から 33,554,431 までと言ったのではなかったでしょうか。33,554,431 は 2^{25}-1 です。余分の 1 ビットはどこから来たのでしょう。

実は、指数部を使うと仮数部の最初のビットが何かを判断することができるのです。指数部のビットがすべて 0 だとすると、仮数部の最初のビットはゼロです。それ以外の場合には、仮数部の最初のビットは 1 です。このように最初のビットが何かは必ずわかるため、仮数部のビットの

並びに最初のビットを含める必要はありません。つまり 1 ビット分、余分に表現できるのです。非常にうまい方法だと思いませんか。

仮数部の最初のビットが 1 である浮動小数点数は正規化されています。つまり仮数は常に 1 と 2 の間の値を持ちます。仮数部の最初のビットが 0 である浮動小数点数は正規化されておらず、(指数が -125 のままであっても) ずっと小さな数を表現することができます。

倍精度数のエンコード方法もほとんど同じですが、仮数部に 52 ビットを使い、指数部に 11 ビットを使って精度を高めています。倍精度の指数のバイアスは 1023 です。

仮数部と指数部

Java 6 で追加された 2 つの `getExponent()` メソッドは、浮動小数点数または倍精度数で表現されるバイアスされていない指数を返します。浮動小数点数の場合、指数は -125 から +127 の間の数であり、倍精度の場合は -1022 から +1023 の間の数です (Inf と NaN の場合は +128/+1024 です)。例えばリスト 1 では、`getExponent()` メソッドの結果と、2 を底とする従来の対数とを比較しています

リスト 1. `Math.log(x)/Math.log(2)` と `Math.getExponent()` との比較

```
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);
    }
}
```

いくつかの値では丸めが行われ、通常の計算よりも `Math.getExponent()` の方が 1 ビットか 2 ビット精度が高くなっています。

```
 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
```

また、こうした計算を大量に行う場合には `Math.getExponent()` の方が高速です。ただし注意しなければならないのは、この関数によって精度が高くなるのは 2 の累乗の場合のみであることです。例えば 3 の累乗に変更すると、出力は次のようになります。

```
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
```

`getExponent()` では仮数部が考慮されませんが、`Math.log()` では考慮されます。少し手間をかければ、仮数を別に見つけ、その対数を取り、その値を指数に加算することもできますが、とても手間をかけるほどの価値はありません。`Math.getExponent()` は基本的に、素早く概略値を求めたい場合に便利な関数であり、正確な値を求めるためのものではありません。

`Math.getExponent()` は `Math.log()` とは異なり、NaN や Inf を返すことはありません。引数が NaN または Inf の場合、浮動小数点数では結果は 128、倍精度数では 1024 です。引数が 0 の場合には、浮動小数点数では結果は -127、倍精度数では -1023 です。引数が負の数の場合の指数は、その数の絶対値の指数と同じです。例えば -8 の指数は 3 であり、8 の指数と同じです。

`getExponent()` に対応する `getMantissa()` メソッドはありませんが、代数演算を少し行えば次のように容易に作り出すことができます。

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

ビット・マスクを使って仮数部を抽出することもできますが、そのためのアルゴリズムは単純ではありません。ビット列を抽出するためには `Double.doubleToLongBits(x) & 0x000FFFFFFFFFFFFFL` を計算すればよいだけですが、正規化された数の余分な 1 ビットを考慮する必要があり、その後で 1 と 2 の間の浮動小数点数に戻すために変換しなければなりません。

ULP (最終桁単位)

実数の密度は無限です。「その次の実数」などというものはありません。2 つの独立した実数をどのように指定しても、その 2 つの数の間には別の実数があります。浮動小数点数の場合には、そうではありません。1 つの浮動小数点数または倍精度数が指定されると、その次の浮動小数点数があり、連続する浮動小数点数や倍精度数の間の最小距離は有限です。`nextUp()` メソッドは、渡された引数よりも大きい浮動小数点数の中でその引数に最も近い浮動小数点数を返します。例えばリスト 2 は、1.0 と 2.0 の間にあるすべての浮動小数点数を、1.0 と 2.0 を含めて出力します。

リスト 2. 浮動小数点数を数える

```
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);
    }

}
```

この結果、1.0 と 2.0 の間には 1.0 と 2.0 を含めて 8,388,609 個の浮動小数点数があることがわかります。この数は大きいですが、この範囲内に存在する無限の実数とは異なり、決して数えられない数ではありません。連続する数同士は 0.0000001 だけ離れています。この距離は ULP (unit of least precision または unit in the last place: 最終桁単位) と呼ばれます。

`nextUp()` とは反対の値が必要な場合、つまり指定された数に最も近く、その指定された数よりも小さな浮動小数点数を見つけるためには、`nextAfter()` メソッドを使います。2 番目の引数は、最も近い数として最初の引数よりも大きな数を見つけるのか小さな数を見つけるのかを指定します。

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

`direction` が `start` よりも大きい場合には、`nextAfter()` は `start` よりも大きな、次の数を返します。`direction` が `start` よりも小さい場合には、`nextAfter()` は `start` よりも小さな、次の数を返します。`direction` と `start` が等しい場合には、`nextAfter()` は `start` そのものを返します。

これらのメソッドは、モデル化やグラフ化のためのアプリケーションで役に立ちます。a と b の間にある 10,000 個の位置で数値をサンプリングしたい場合、a と b の間にある一意の点として 1,000 点しか識別できないのであれば、10 回のうち 9 回は無駄な作業をすることになります。10 分の 1 の作業でも得られる結果はまったく同じです。

もちろん、どうしても追加の精度が必要な場合には、`double` や a `BigDecimal` など、もっと精度の高いデータ型を選択する必要があります。例えば、私はマンデルブロ集合エクスプローラーにこれらのデータ型が登場するのを見たことがあります (このエクスプローラーで極限まで拡大すると、最短距離にある 2 つの倍精度数の間にグラフ全体が収まります)。マンデルブロ集合は無限の深さと複雑さを持ちますが、`float` と `double` は深さが一定以上になると隣接する点同士の間を区別できなくなります。

`Math.ulp()` メソッドは、ある数から直近の隣接数までの距離を返します。リスト 3 は 2 の累乗に対する ULP をリストアップします。

リスト 3. ある浮動小数点数に対する 2 の累乗の ULP

```
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));
        }
    }

}
```

下記は出力の一例です。

```
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
```

これを見るとわかるように、2 を累乗した数が小さな場合には浮動小数点数は非常に正確です。しかし 2^{20} あたりになると、多くのアプリケーションで正確さが問題になってきます。浮動小数点数の桁の限界に近くなると、連続する値の区切りは十垓 (sextillion: 1021) になります (実際には sextillion よりもずっと大きいのですが、私はそんな大きな数字を表す言葉を見つけることができませんでした)。

リスト 3 を見るとわかるように、ULP のサイズは一定ではありません。数値が大きくなると、数値の間にある浮動小数点数の数は少なくなります。例えば 10,000 と 10,001 の間には 1,025 個の浮動小数点数しかなく、数値同士の距離は 0.001 です。1,000,000 と 1,000,001 の間には 17 個の浮動小数点数しかなく、数値同士の間の距離は約 0.05 です。精度は数値の大きさと逆相関関係にあります。実際、10,000,000 という浮動小数点数では ULP が 1.0 にまで大きくなり、それ以上の数になると同じ浮動小数点数に複数の整数値がマッピングされます。倍精度数の場合には、こうしたことは約 4 京 5 千兆 (4.5E15) まで起こりませんが、やはり考慮すべき点であることは確かです。

浮動小数点数の精度が有限であることによって起こる思わぬ結果の 1 つとして、ある数以上になると x+1 == x が真になります。例えば、下記は明らかに単純なループですが、実際には無限ループになってしまいます。

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

実際、このループが無限ループになるのは、正確には浮動小数点数の値が 16,777,216 になった時点からです。これは 2^{24} であり、この数になると ULP が増分よりも大きくなります。

`Math.ulp()` メソッドの現実的な用途としてテストがあります。皆さんもよくご存じのとおり、通常は浮動小数点数同士が完全に等しいかどうかを比べることはなく、ある許容範囲内で等しいかどうかをチェックします。例えば、想定される浮動小数点数の値と実際の浮動小数点数の値とを JUnit で比較する場合には下記のようにします。

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

ここでは、実際の値が想定値の 0.02 以内にあれば等しいとしていますが、0.02 は妥当な許容範囲なのでしょうか。想定される値が 10.5 または -107.82 であれば、0.02 はおそらく適切でしょう。しかし想定される値が数十億であれば、0.02 と 0 とはまったく区別できません。多くの場合、テストは ULP の観点で相対誤差を考慮して判断する必要があります。計算にどの程度の精度が必要かにより、通常は許容範囲を 1 ULP から 10 ULP の間で選択する必要があります。例えば下記では、実際の結果が真の値の 5 ULP 以内でなければならないと指定しています。

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

想定される値により、許容範囲は何兆という場合もあれば、何百万という場合もあります。

scalb

`Math.scalb(x, y)` は x を 2^y 倍します (`scalb` は「scale binary」の略です)。

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

例えば `Math.scalb(3, 4)` は $3 * 2^4$ を返しますが、これは 3*16、つまり 48.0 です。`Math.scalb()` は下記のように `getMantissa()` の代わりの実装として使うことができます。

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

`Math.scalb()` は `x*Math.pow(2, scaleFactor)` とどう違うのでしょうか。実は最終的な結果に違いはありません。私がどのような値を入力しても、返される値は 1 ビットも異なりませんでした。しかしパフォーマンスは検討に値します。`Math.pow()` はパフォーマンスが悪いことで有名ですが、非常に特殊なケース (3.14 を -0.078 乗するなど) を処理する場合に使うことができます。`Math.pow()` は通常、2 や 3 といった小さな整数の累乗や 2 を底とするような特殊な場合、まったく不適切なアルゴリズムを選択してしまいます。

パフォーマンスに関する他の一般的な評価と同様、`Math.scalb` と `Math.pow` のパフォーマンス比較の評価は非常に慎重に行う必要があります。一部のコンパイラーや VM は他のものよりも優れています。一部の最適化プログラムは `x*Math.pow(2, y)` を特別なケースとして認識し、`Math.scalb(x, y)` に変換するか、あるいは `Math.scalb(x, y)` に非常に近いものに変換しま

す。そのためパフォーマンスにはまったく違いがないかもしれません。しかし私は、一部の VM がそれほど賢くはないことを確認しました。例えば Apple の Java 6 VM でテストしている際、何度テストを繰り返しても `Math.scalb()` は `x*Math.pow(2, y)` よりも 2 桁高速でした。もちろん、通常これはまったく問題にはなりません。しかし何百万回もの指数演算を行うような特殊な場合では、`Math.pow()` の代わりに `Math.scalb()` を使って演算できないか検討する必要があります。

`copySign`

`Math.copySign()` メソッドは、1 番目の引数の符号を 2 番目の引数の符号に設定します。あまり深く考えずに実装した例をリスト 4 に示します。

リスト 4. `copySign` アルゴリズムとして考えられる例

```
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;
}
```

しかし実際の実装はリスト 5 のようになります。

リスト 5. `sun.misc.FpUtils` から引用した実際のアルゴリズム

```
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)));
}
```

この実装について注意深く調べ、ビットを抽出してみると、NaN 符号が正として扱われていることがわかります。厳密に言えば、`Math.copySign()` によって必ずそうなるわけではなく、必ずそうなるのは `StrictMath.copySign()` の場合のみです。しかし実際には、どちらの関数も同じビット操作コードを呼び出しています。

リスト 5 はリスト 4 よりもわずかに高速かもしれませんが、リスト 5 のようにする主な理由は負のゼロを適切に処理できる点にあります。`Math.copySign(10, -0.0)` は -10 を返しますが、`Math.copySign(10, 0.0)` は 10.0 を返します。あまり深く考えずに実装したリスト 4 のアルゴリズムでは、どちらの場合にも 10.0 を返します。負のゼロが発生するのは、非常に小さな負の倍精度数を非常に大きな正の倍精度数で割る場合など、繊細な注意を要する演算の場合です。例えば `-1.0E-147/2.1E189` は負のゼロを返しますが、`1.0E-147/2.1E189` は正のゼロを返します。しかしこの 2 つの値は `==` で比較すると等価なため、この 2 つの値を区別したい場合には `Math.copySign(10, -0.0)` または `Math.signum()` を使って両者を比較する必要があります (`Math.signum()` は `Math.copySign(10, -0.0)` を呼び出します)。

対数と指数

無限精度の実数ではなく有限精度の浮動小数点数を扱う際に、どれほど注意が必要かを理解するための好例として、指数関数があります。e^x (`Math.exp()`) は多くの等式で登場します。例えば、第 1 回で説明した cosh 関数を定義するために ex (Math.exp()) が使われます。

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

しかし x の値が負の場合 (大まかには -4 よりも小さな場合) には、`Math.exp()` を計算するために使われるアルゴリズムの動作はあまり適切ではなく、丸め誤差を生じがちです。$e^x - 1$ を別のアルゴリズムで計算し、その結果に 1 を加えた方が正確です。`Math.expm1()` メソッドは、この別のアルゴリズムを実装しています。(`m1` は「マイナス 1」を表しています。) 例えばリスト 6 は、`x` のサイズに応じて 2 つのアルゴリズムを切り換える cosh 関数を示しています。

リスト 6. cosh 関数

```
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;
}
```

この例は少しばかり学術的です。なぜなら `Math.exp()` と `Math.expm1() + 1` の違いが大きい場合には、e^x 項が e^{-x} 項を完全に圧倒するからです。しかし `Math.expm1()` は財務計算で利率が低い場合には非常に現実的です (例えば米国財務省発行の国債の日歩の計算など)。

`Math.log1p()` は `Math.expm1()` の逆関数であり、これは `Math.log()` が `Math.exp()` の逆関数であるのと同じです。`Math.log1p()` は 1 に引数の値を加えた数の対数を計算します。(`1p` は「プラス 1」を表しています。) この関数は 1 に近い値に対して使用します。例えば、`Math.log(1.0002)` と計算するのではなく、`Math.log1p(0.0002)` と計算します。

一例として、日歩が 0.03 の場合に 1,000 ドルを投資して 1,100 ドルに増えるまでに必要な日数を知りたいとしましょう。リスト 7 ではこれを行います。

リスト 7. 現在の投資額が将来の指定額になるまでに必要な期間

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

この場合の `1p` は非常に自然に解釈することができます。こうした値を計算する際の通常の公式で 1+r が使われているからです。つまり、貸し手は通常、1 に追加されるパーセント部分 (+r の部分) で利息を表現しますが、投資する人は初期投資の $(1+r)^n$ が返ってくることを期待します。実際、3 パーセントの利息でお金を貸して投資額の 3 パーセントしか戻ってこないとしたら、ひどい投資だということになります。

倍精度数は実数ではない

浮動小数点数は実数ではありません。浮動小数点数の数は有限です。浮動小数点数で表現できる値には最大と最小があります。最も重要な点として、浮動小数点数の精度は桁数が大きくても有限であり、丸めの誤差が生じます。もちろん、整数を扱う際には、浮動小数点数や倍精度数は int や long よりも明らかに精度が劣っています。こうした制限を注意深く考慮した上で、科学アプリケーションや工学アプリケーションでは特に、堅牢で信頼性の高いコードを作成する必要があります。また財務アプリケーションでも (最後の 1 セントまで精度が要求される会計アプリケーションでは特に)、浮動小数点数と倍精度数の扱いには十分すぎるほどの注意が必要です。

`java.lang.Math` クラスと `java.lang.StrictMath` クラスは、こうした問題に対処するために注意深く設計されています。これらのクラスやそのメソッドを適切にすれば、プログラムを改善することができます。この記事では何よりも、浮動小数点演算を適切に行うためにはいかに慎重でなければならないかを説明しました。独自のアルゴリズムを作り出すよりも、可能な限りエキスパートに任せた方が適切です。`java.lang.Math` と `java.lang.StrictMath` のメソッドを使える場合には、これらのメソッドを使うべきです。ほとんどすべての場合において、これらのメソッドの方が選択肢として適切なのです。

著者について

Elliotte Rusty Harold

Elliotte Rusty Harold はニューオーリンズ出身であり、時たま、おいしいガンボ (gumbo: オクラ入りのスープ) を食べに帰っています。ただし現在はアーヴィン近郊の University Town Center に、妻の Beth と猫の Charm (charmed quarkからとりました) と Marjorie（ 義理の母の名前からとりました ）と一緒に住んでいます。彼の Webサイト Cafe au Lait は、インターネット上で最も人気のある独立系 Java サイトの 1 つです。また、そこから派生した Cafe con Leche は、最も人気のある XML サイトの一つです。彼の最近の著作には『Refactoring HTML』があります。