Introduction

This document describes the design, motivation and implementation details of a class representing rational numbers and it's dependencies found in the open source project hosted at *http://sourceforge.net/projects/precisefloating*. The most interesting feature of the *precisefloating.Rational* class is its tight integration with the Java primitives. Because the Java primitives cannot represent any rational number, the conversions from rational numbers to Java primitives are rounding, with a specified rounding direction. The reverse conversions are always exact, as any finite floating point number and all the integers are also rational numbers.

Informally, when a real number x is rounded to a finite subset enhanced with positive and negative infinity, if it belongs to that subset that the rounding result is the same number, possible represented differently. If not, than it must sit between two consecutive numbers in the sorted subset, $a < x < b$. Then x rounded to negative infinity (*RoundingMode.ROUND_FLOOR*) is a. The same real x rounded to positive infinity (*RoundingMode.ROUND_CEILING*) is b. For round to nearest (*RoundingMode.ROUND_HALF_EVEN*), the result is the nearer of the two, if any. If $x = (a + b) / 2$, the nearest value is considered to be the *even* one. The definition of even, of course, depends on the actual subset. For integral value sets, a number is *even* if it is exactly divisible by *two*. For the single and double precision floating point value sets in the *IEEE-754* representation, a number is considered *even* when it has the lowest bit equal to zero. Note that infinities are even in both the single and double precision value sets.

Rational numbers can be used instead of floating-point arithmetic when exact arithmetic operations are needed and they are not achievable using floating-point arithmetic. So if an algorithm needs to operate on exact intermediate results, and the only operations involved are the addition and multiplication in the rational field (or composed operations, such as integer power), and speed is not as important as the exactness of the operations performed, the *precisefloating.Rational* class is a good choice.

Rational public interface

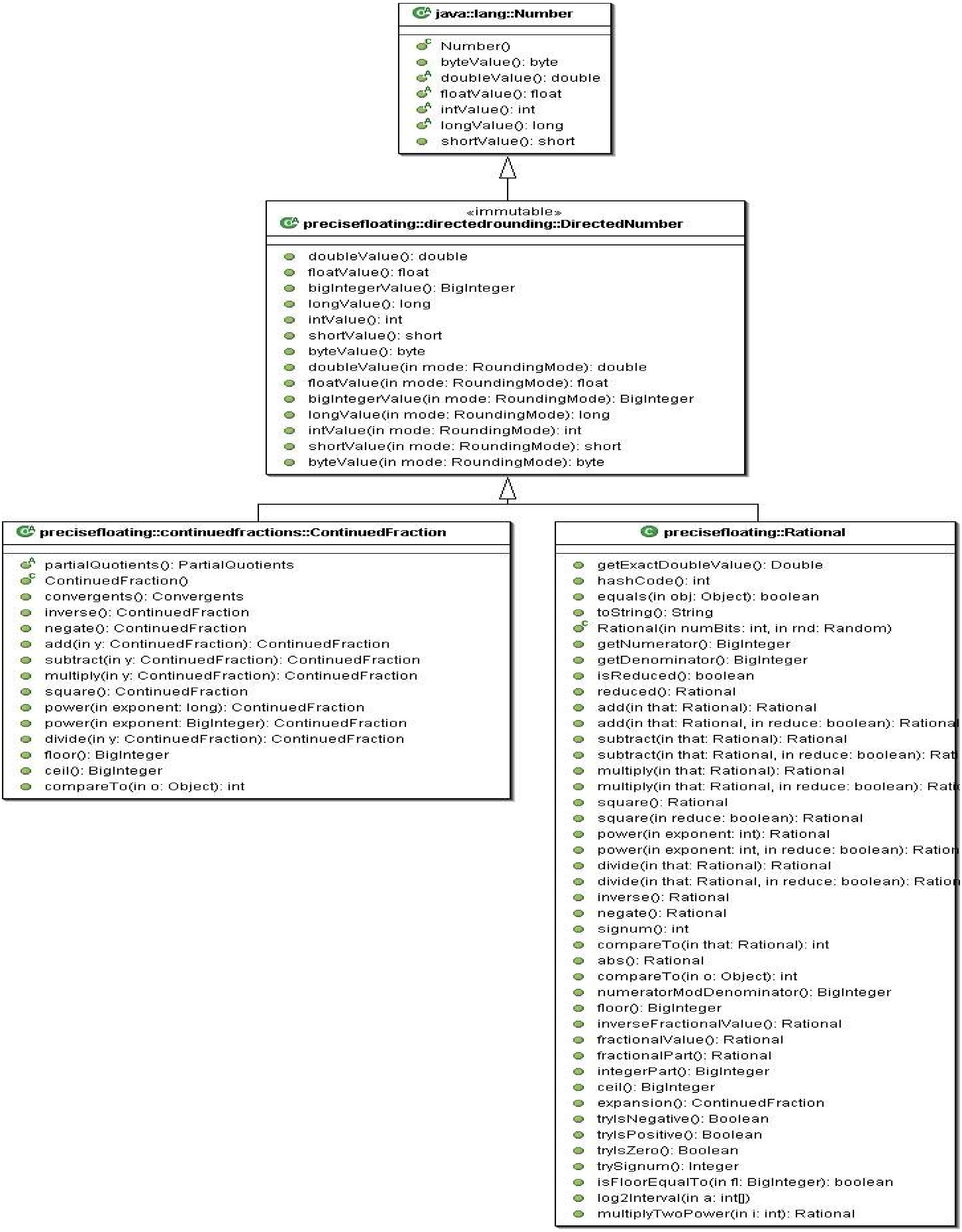
The Rational class has a few *public static final* fields, including the *ZERO* and *ONE* numbers. These fields should be used rather than creating other instances as often as possible, but of course this is not a requirement.

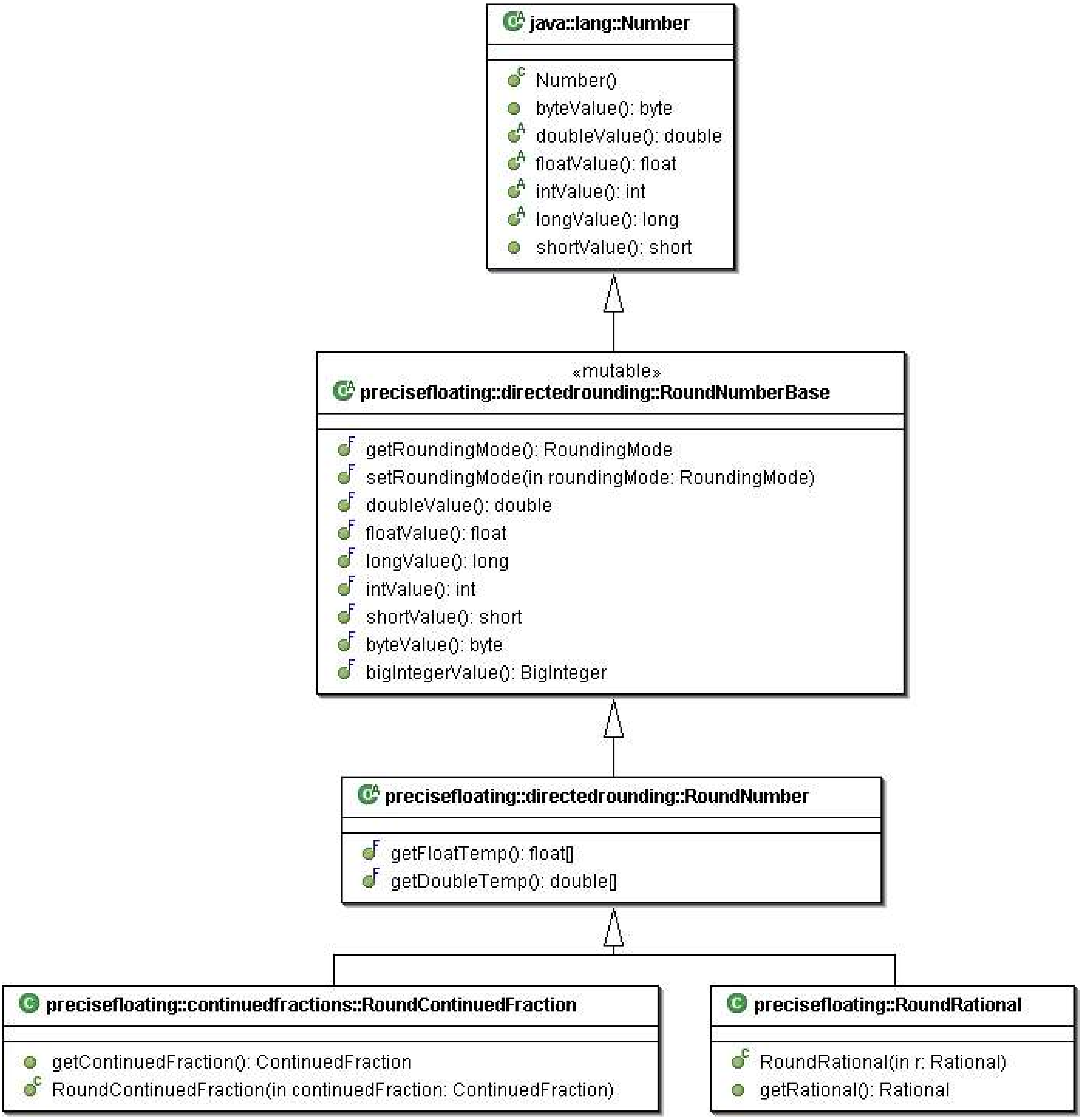
A rational number basically consists of two *BigInteger* numbers, the *numerator* and the *denominator*. The sign of the denominator is invariably positive as represented in this class. If a rational is being constructed with a negative denominator, the sign is simply multiplied with the numerator and kept there. A few *create* factory methods are also provided, one of main importance being *Rational.create(double d)*. The operations currently supported are *addition*, *subtraction*, *multiplication*, *square*, *integer power*, *division*, *inverse*, *modulus*, *floor*, *ceiling*, *fractional part*, *fractional value*, *integer part*, *comparison* and *equality* testing, and the simple continued fraction expansion. But most importantly, this class provides rounding behavior for a rational to a Java primitive type. This is based on the public class *precisefloating.RoundRational* which provides the implementation for the three rounding modes defined in *precisefloating.RoundingMode: RoundingMode.ROUND_FLOOR*, *RoundingMode.ROUND_HALF_EVEN* and *RoundingMode.ROUND_CEILING*. The [*Number*]() inherited methods in the *Rational* class provide the round-to-nearest behavior letting the programmers to use a standard interface for the conversions. Overloads for the method in *Number* are also provided with a parameter of type *RoundingMode*. They can be used if a certain rounding mode is desired for the conversion rather than the by-default round-to-nearest mode.

There is a lot of documentation, both theoretical and implementation details in the the binary distribution, more exactly in the *docs* directory. Further information about the simple continued fraction expansions and arithmetic is also included there.

We support both reduced and unreduced rationals in a flexible manner. A rational is reduced when the numerator and denominator are *relatively prime*. The *Rational* class, once instantiated, cannot be changed. That is, the *reduced()* method creates a distinct instance if the current instance is not already reduced. Reduced rationals take less memory and usually perform better, but the overhead of reducing a rational number shouldn't be ignored.

Class Diagrams


java::lang::Number
Number()
byteValue(): byte
doubleValue(): double
floatValue(): float
intValue(): int
longValue(): long
shortValue(): short
«immutable»
precisefloating::directedrounding::DirectedNumber
doubleValue(): double
floatValue(): float
bigIntegerValue(): BigInteger
longValue(): long
intValue(): int
shortValue(): short
byteValue(): byte
doubleValue(in mode: RoundingMode): double
floatValue(in mode: RoundingMode): float
bigIntegerValue(in mode: RoundingMode): BigInteger
longValue(in mode: RoundingMode): long
intValue(in mode: RoundingMode): int
shortValue(in mode: RoundingMode): short
byteValue(in mode: RoundingMode): byte
precisefloating::continuedfractions::ContinuedFraction
partialQuotients(): PartialQuotients
ContinuedFraction()
convergents(): Convergents
inverse(): ContinuedFraction
negate(): ContinuedFraction
add(in y: ContinuedFraction): ContinuedFraction
subtract(in y: ContinuedFraction): ContinuedFraction
multiply(in y: ContinuedFraction): ContinuedFraction
square(): ContinuedFraction
power(in exponent: long): ContinuedFraction
power(in exponent: BigInteger): ContinuedFraction
divide(in y: ContinuedFraction): ContinuedFraction
floor(): BigInteger
ceil(): BigInteger
compareTo(in o: Object): int
precisefloating::Rational
getExactDoubleValue(): Double
hashCode(): int
equals(in obj: Object): boolean
toString(): String
Rational(in numBits: int, in rnd: Random)
getNumerator(): BigInteger
getDenominator(): BigInteger
isReduced(): boolean
reduced(): Rational
add(in that: Rational): Rational
add(in that: Rational, in reduce: boolean): Rationa
subtract(in that: Rational): Rational
subtract(in that: Rational, in reduce: boolean): Rat
multiply(in that: Rational): Rational
multiply(in that: Rational, in reduce: boolean): Rati
square(): Rational
square(in reduce: boolean): Rational
power(in exponent: int): Rational
power(in exponent: int, in reduce: boolean): Ration
divide(in that: Rational): Rational
divide(in that: Rational, in reduce: boolean): Ration
inverse(): Rational
negate(): Rational
signum(): int
compareTo(in that: Rational): int
abs(): Rational
compareTo(in o: Object): int
numeratorModDenominator(): BigInteger
floor(): BigInteger
inverseFractionalValue(): Rational
fractionalValue(): Rational
fractionalPart(): Rational
integerPart(): BigInteger
ceil(): BigInteger
expansion(): ContinuedFraction
tryIsNegative(): Boolean
tryIsPositive(): Boolean
tryIsZero(): Boolean
trySignum(): Integer
isFloorEqualTo(in fl: BigInteger): boolean
log2Interval(in a: int[])
multiplyTwoPower(in i: int): Rational



Rational implementation

The implementation of the methods in the Rational class is generally straightforward. The conversion from double values to rational numbers is handled by this method:

```
public static Rational create(double d) {
    Rational r;
```

```
        if (d == 0) {
            r = ZERO;
        } else {
            DoublePrecisionNo dpn = new DoublePrecisionNo(d);

            // -1022 - 52 <= computedExponent <= 1075 - 52
            // -1074 <= computedExponent <= 1023
            // 2 ^ (e - N + 1)
            int computedExponent = dpn.getExponent() - Formulas.N_DOUBLE + 1;

            BigInteger oddM = BigInteger.valueOf(dpn.getMantissa());

            if (computedExponent < 0) {
                assert -computedExponent < 1075;
                r = create(oddM, Formulas.bigintPow2(-computedExponent));
            } else {
                r = create(leftShift(oddM, computedExponent), BigInteger.ONE);
            }

            r.exactDoubleValue = new Double(d);
        }

        return r;
    }
```

This is actually the double to rational conversion routine. If the double number is zero, we reuse the *ZERO* instance. Otherwise, we use the class *DoublePrecisionNo* to extract the exponent and fraction of the double number. The *DoublePrecisionNo* class is thoroughly explained in another document, available in the binary distribution as *directedrounding.pdf* or as an online article at http://www.codeproject.com/useritems/precisefloating.asp. Note that if *d* is infinite or *NaN* we get an *IllegalArgumentException* being thrown. That is correct because rational numbers are neither infinite nor NotANumber. The *DoublePrecisionNo* class basically decomposes the double value into *dpn.getMantissa()*pow2 (dpn.getExponent() - 52)*, no matter if we deal with a normal or subnormal number. *Formulas.bigintPow2* caches all the possible two powers that can appear in this computation, more exactly from *0* to *1075* exclusive. Also, the field *exactDoubleValue* is initialized to indicate that this rational number is known to be exactly representable in the double precision value set.

DirectedNumber

The *Rational* class extends *precisefloating.directedrounding.DirectedNumber*. That is a thread safe class that can be used as a superclass for immutable numbers. It supports a rounding direction that is only set in the constructor and cannot be changed, but it also provides overloads with a *RoundingMode* parameter. A few excerpts from this class are presented below:

```
public abstract class DirectedNumber extends Number {
```

```
    /**
     * This method is called in a synchronized context.
     *
     * @return the freshly created RoundNumberBase instance
     */
    protected abstract RoundNumberBase createRoundNumber();

    public double doubleValue() {
        return doubleValue(defaultRoundingMode);
    }

    public BigInteger bigIntegerValue() {
        return bigIntegerValue(defaultRoundingMode);
    }

    public double doubleValue(RoundingMode mode) {
        getRoundNumber();

        synchronized (roundNumber) {
            roundNumber.setRoundingMode(mode);
            return roundNumber.doubleValue();
        }
    }

    public BigInteger bigIntegerValue(RoundingMode mode) {
        getRoundNumber();

        synchronized (roundNumber) {
            roundNumber.setRoundingMode(mode);
            return roundNumber.bigIntegerValue();
        }
    }

// other methods and fields

}
```

Aside the *Number* inherited methods, this class adds conversions to *BigInteger.* The derived classes must implement the abstract method *createRoundNumber()*. Of course, both *Rational* and *ContinuedFraction* implemente it by providing specialized instances of *precisefloating.directedrounding.RoundNumberBase*:

Rational:

```
    protected RoundNumberBase createRoundNumber() {
        return new RoundRational(this);
    }
```

ContinuedFraction:

```
  protected RoundNumberBase createRoundNumber() {
    return new RoundContinuedFraction(this);
  }
```

All the rounding operations for rational numbers are actually implemented in the *RoundRational* class, and that inherits the precomputation of values from its abstract base class *precisefloating.directedrounding.RoundNumber*, which, at its turn, inherits the caching strategy from its base class *RoundNumberBase.* We'll discuss each of them from the top of the hierarchy to the bottom.

RoundNumberBase

RoundNumberBase is an abstract base class for finite numbers supporting directed rounding. It caches the values computed in every rounding direction. The internal arrays are constructed on the first use, so there is almost no overhead added to an extending class as long as the *Number* methods are not called. This class is mutable because of the setter for *roundingMode*, and it should only be used with delegation by immutable numbers. It is also the superclass of *RoundNumber*. A few methods from this class are presented here:

```
public abstract class RoundNumberBase extends Number {

  public final RoundingMode getRoundingMode() {
    assert roundingMode != null;
    return roundingMode;
  }

  public final void setRoundingMode(RoundingMode roundingMode) {
    if (roundingMode == null) {
      throw new NullPointerException("roundingMode must not be null");
    }
    this.roundingMode = roundingMode;
  }

  protected final boolean isDoubleComputed(int idx) {
    return getComputed()[0][idx];
  }

  protected void computed(int idx, double d) {
    assert idx > 0 || d != Double.POSITIVE_INFINITY
        : "round-to-negative infinity cannot result in positive infinity";
    assert idx < 2 || d != Double.NEGATIVE_INFINITY
        : "round-to-positive infinity cannot result in negative infinity";

    if (!getComputedDoubles()[idx]) {
      getDoubleValues()[idx] = d;
      getComputedDoubles()[idx] = true;
    } else {
      assert getDoubleValues()[idx] == d;
    }
```

```
    }

    protected abstract double computeDoubleValue();

    protected abstract float computeFloatValue();

    protected abstract BigInteger computeBigIntegerValue();

    protected int computeIntValue() {
        BigInteger v = bigIntegerValue();
        return Formulas.truncateToInt(v);
    }

    protected short computeShortValue() {
        BigInteger v = bigIntegerValue();
        return Formulas.truncateToShort(v);
    }

    protected byte computeByteValue() {
        BigInteger v = bigIntegerValue();
        return Formulas.truncateToByte(v);
    }

// other methods and fields

}
```

By default, all the integral rounding values are computed by rounding first to *BigInteger*. Of couse this can be overridden for increased performance in subclasses. The truncation methods from *BigInteger* to other integral types simply return the equivalent primitive integral value if the parameter is in the correct range, or otherwise *MIN_VALUE* or *MAX_VALUE* for the specific integral type. Here is one of the truncation methods from the *precisefloating.Formulas* class:

```
    public static long truncateToLong(BigInteger v) {
        final long l;

        switch (v.signum()) {
            case -1:
                if (v.compareTo(BIG_MIN_LONG) <= 0) {
                    l = Long.MIN_VALUE;
                } else {
                    // exact narrowing primitive conversion
                    l = v.longValue();
                }
                break;
            case 0:
                l = 0;
                break;
            case 1:
```

```
            if (v.compareTo(BIG_MAX_LONG) >= 0) {
                l = Long.MAX_VALUE;
            } else {
                // exact narrowing primitive conversion
                l = v.longValue();
            }
            break;
        default:
            throw new InternalError("v.signum() = " + v.signum());
    }

    return l;
}
```

Note that instead of checking control-flow invariants with *assert false* as it is recommended in the assertion programming guide available at http://java.sun.com/j2se/1.4.2/docs/guide/lang/assert.html, we always explicitly throw an error in such cases. This is much better because the exception would always be thrown, with assertions enabled or not, and it does not have any performance hit, because it will (hopefully) never get to execute.

RoundNumber

The *RoundNumber* class is inserted into the hierarchy between *RoundRational* (or *RoundContinuedFraction*) and *RoundNumberBase*. When a rounded value is computed, this class makes its best effort to deduce the value of the number as rounded in other directions or as other types without creating additional instances of *BigInteger* of *DoublePrecisionNo*, as this class must be very lightweight. The only objects that are created by this class or the transient closure of all the called methods are two temporary two-size arrays, one of type *float[]* and the other *double[]*, but they are instantiated lazily. It serves as a superclass for *RoundNumber* and *RoundContinuedFraction*. Because it is too big to thoroughly explain here, but we only include the list of all its methods:

```
public abstract class RoundNumber extends RoundNumberBase {

    public final float[] getFloatTemp()

    public final double[] getDoubleTemp()

    protected final void computed(int idx, double d)
    private void precomputeDoubles(int idx, double d)
    private void precomputeFloats(int idx, double d)
    private void precomputeLongs(int idx, double d)
    private void precomputeInts(int idx, double d)
    private void precomputeShorts(int idx, double d)
    private void precomputeBytes(int idx, double d)

    protected final void computed(int idx, float f)
    private void precomputeDoubles(float f)
```

```
    private void precomputeFloats(int idx, float f)
    private void precomputeLongs(int idx, float f)
    private void precomputeInts(int idx, float f)
    private void precomputeShorts(int idx, float f)
    private void precomputeBytes(int idx, float f)

    protected final void computed(int idx, long l)
    private void precomputeDoubles(int idx, long l)
    private void precomputeFloats(int idx, long l)
    private void precomputeLongs(int idx, long l)
    private void precomputeInts(int idx, long l)
    private void precomputeShorts(int idx, long l)
    private void precomputeBytes(int idx, long l)

    protected final void computed(int idx, int i)
    private void precomputeDoubles(int i)
    private void precomputeFloats(int idx, int i)
    private void precomputeLongs(int idx, int i)
    private void precomputeInts(int idx, int i)
    private void precomputeShorts(int idx, int i)
    private void precomputeBytes(int idx, int i)

    protected final void computed(int idx, short s)
    private void precomputeDoubles(short s)
    private void precomputeFloats(short s)
    private void precomputeLongs(int idx, short s)
    private void precomputeInts(int idx, short s)
    private void precomputeShorts(int idx, short s)
    private void precomputeBytes(int idx, short s)

    protected final void computed(int idx, byte b)
    private void precomputeDoubles(byte b)
    private void precomputeFloats(byte b)
    private void precomputeLongs(int idx, byte b)
    private void precomputeInts(int idx, byte b)
    private void precomputeShorts(int idx, byte b)
    private void precomputeBytes(int idx, byte b)

}
```

This class extensively uses the *floatInterval* method from *Formulas*:

```
    public static void floatInterval(double d, float[] floatTemp) {
        if (Double.isNaN(d)) {
            throw new IllegalArgumentException("d parameter must not be NaN");
        }

        // conversion rounds to nearest
        float f = (float) d;
```

```
    // f suffers widening conversion to double
    if (f < d) {
      if (f == Float.NEGATIVE_INFINITY) {
        assert d < -Float.MAX_VALUE && d >= -Double.MAX_VALUE;
        floatTemp[0] = Float.NEGATIVE_INFINITY;
        floatTemp[1] = -Float.MAX_VALUE;
      } else {
        floatTemp[0] = f;
        floatTemp[1] = next(f);
      }
    } else {
      if (f > d) {
        if (f == Float.POSITIVE_INFINITY) {
          assert d > Float.MAX_VALUE && d <= Double.MAX_VALUE;
          floatTemp[0] = Float.MAX_VALUE;
          floatTemp[1] = Float.POSITIVE_INFINITY;
        } else {
          floatTemp[0] = previous(f);
          floatTemp[1] = f;
        }
      } else {
        assert f == d;
        floatTemp[0] = floatTemp[1] = f;
      }
    }
  }
```

This method returns the smallest possible single precision floating point interval that contains the double precision value given as a parameter. If the double value is infinite or exactly representable in single precision, the same number is returned, exactly converted to single precision. Otherwise, the method computes the smallest open single precision floating point interval that contains the double precision value. At the heart of this method stays the round-to-nearest primiteve conversion

```
float f = (float) d;
```

and the comparisons $f < d$ or $f > d$. These comparisons are done in double precision, by exactly and automatically first converting the single precision value f to a double precision number before the comparison. If $f < d$ then we know for sure that the initial conversion rounded towards negative infinity, while if $f > d$ it was rounded towards positive infinity. The equality holds iff the double value is exactly representable in single precision floating point format. Here and all around the project sources, a lot of *assert* statements are used as a means of documenting the code so one should have no problems understanding this method.

Let's get back at the *RoundNumber* class. In order to permit the reader get a grasp about this class, below we explain the implementation of one of the most illustrative methods:

```
  private void precomputeFloats(int idx, double d) {
```

```
        Formulas.floatInterval(d, getFloatTemp());

        float low = floatTemp[0], high = floatTemp[1];

        if (!Formulas.isFinite(low) || !Formulas.isFinite(high)) {
            if (high == Float.POSITIVE_INFINITY) {
                computed(0, Float.MAX_VALUE);
                computed(2, Float.POSITIVE_INFINITY);

                if (low == Float.POSITIVE_INFINITY) {
                    assert d == Double.POSITIVE_INFINITY;
                    /*
                    As (float)Double.MAX_VALUE = Float.POSITIVE_INFINITY,
                    value > Double.MAX_VALUE implies the value is
Float.POSITIVE_INFINITY
                    in round-to-nearest mode.
                    */
                    computed(1, Float.POSITIVE_INFINITY);
                } else {
                    assert low == Float.MAX_VALUE;
                    assert Float.MAX_VALUE < d && d < Double.POSITIVE_INFINITY;

                    if (d <
Formulas.SINGLE_SMALLEST_NEAREST_POSITIVE_INFINITY) {
                        computed(1, Float.MAX_VALUE);
                    } else {
                        if (d >
Formulas.SINGLE_SMALLEST_NEAREST_POSITIVE_INFINITY) {
                            computed(1, Float.POSITIVE_INFINITY);
                        } else {
                            assert d ==
Formulas.SINGLE_SMALLEST_NEAREST_POSITIVE_INFINITY;

                            if (idx == 0) {
                                //
Formulas.SINGLE_SMALLEST_NEAREST_POSITIVE_INFINITY <= value <
Float.POSITIVE_INFINITY
                                computed(1, Float.POSITIVE_INFINITY);
                            }
                        }
                    }
                }
            } else {
                if (low == Float.NEGATIVE_INFINITY) {
                    computed(0, Float.NEGATIVE_INFINITY);
                    computed(2, -Float.MAX_VALUE);

                    if (high == Float.NEGATIVE_INFINITY) {
                        assert d == Double.NEGATIVE_INFINITY;
                        /*
```

```
                As (float)-Double.MAX_VALUE = Float.NEGATIVE_INFINITY,
                value < -Double.MAX_VALUE implies the value is
Float.NEGATIVE_INFINITY
                in round-to-nearest mode.
                */
                computed(1, Float.NEGATIVE_INFINITY);
            } else {
                assert high == -Float.MAX_VALUE;
                assert Double.NEGATIVE_INFINITY < d && d < -Float.
MAX_VALUE;

                if (d <
Formulas.SINGLE_LARGEST_NEAREST_NEGATIVE_INFINITY) {
                    computed(1, Float.NEGATIVE_INFINITY);
                } else {
                    if (d >
Formulas.SINGLE_LARGEST_NEAREST_NEGATIVE_INFINITY) {
                        computed(1, -Float.MAX_VALUE);
                    } else {
                        assert d ==
Formulas.SINGLE_LARGEST_NEAREST_NEGATIVE_INFINITY;

                        if (idx == 2) {
                            // Float.NEGATIVE_INFINITY < value <=
Formulas.SINGLE_LARGEST_NEAREST_NEGATIVE_INFINITY
                            computed(1, Float.NEGATIVE_INFINITY);
                        }
                    }
                }
            }
        }
    }
} else {
    if (low == high) {
        // can't be (-0.0F, +0.0F)
        assert Float.floatToIntBits(low) == Float.floatToIntBits(high);

        // d is exactly representable as a floating point number
        float f = (float) d;
        assert f == low && f == high && f == d;

        computed(idx, f);
    } else {
        assert low < high;

        // always exact in double precision
        double mid = ((double) low + (double) high) / 2;
        assert Formulas.isFinite(mid);

        log.finest("low = " + low);
```

```
            log.finest("high = " + high);

            computed(0, low);
            computed(2, high);

            if (d < mid) {
               computed(1, low);
            } else {
               if (d > mid) {
                  computed(1, high);
               } else {
                  assert d == mid;

                  switch (idx) {
                     case 0:
                        // mid <= value < high
                        if (Formulas.isEven(high)) {
                           computed(1, high);
                        }

                        break;
                     case 1:
                        // no finestrmation supplied
                        break;
                     case 2:
                        // low < value <= mid
                        if (Formulas.isEven(low)) {
                           computed(1, low);
                        }

                        break;
                     default:
                        throw new InternalError("invalid directionIndex " + idx);
                  }
               }
            }
         }
      }
   }
```

In the code above we first compute the single precision floating point interval *low, high*.

If *high* is positive infinity, then we know for sure that *d > Float.MAX_VALUE*. Rounded to negative infinity in single precision, this number is *Float.MAX_VALUE*. Rounded towards positive infinity, it is *Float.POSITIVE_INFINITY*. Otherwise, we make our best effort to find out the value as rounded towards nearest. We know for sure that *low* is negative infinity iff *d* is also negative infinity. In this case, the rounded value is *Float.POSITIVE_INFINITY*. Else we know for sure that *low* is *Float.MAX_VALUE* and that *Float.MAX_VALUE < d && d <*

Double.POSITIVE_INFINITY. If *d* < *Formulas.SINGLE_SMALLEST_NEAREST_POSITIVE_INFINITY* then the rounded value is *Float.MAX_VALUE*. If *d* > *Formulas.SINGLE_SMALLEST_NEAREST_POSITIVE_INFINITY*, the value is *Float.POSITIVE_INFINITY*. Otherwise, the statement *d* == *Formulas.SINGLE_SMALLEST_NEAREST_POSITIVE_INFINITY* must hold true. In this case, if the direction in which *d* was obtained is round to negative infinity, it holds that *Formulas.SINGLE_SMALLEST_NEAREST_POSITIVE_INFINITY* <= *value* < *Float.POSITIVE_INFINITY* and the rounded to nearest single precision value is certainly *Float.POSITIVE_INFINITY*. If the direction was round to nearest we don't know if the actual value (this number instance) is smaller, equal to or greater than *Formulas.SINGLE_SMALLEST_NEAREST_POSITIVE_INFINITY* so we can't infere our result. If the direction was round to positive infinity, the actual value is less than or equal to *Formulas.SINGLE_SMALLEST_NEAREST_POSITIVE_INFINITY.* If it were smaller, our result would be Formulas.MAX_VALUE; if they were equal, the result would be *Formulas.POSITIVE_INFINITY*. But we don't know which one so we again fail to precompute the round to nearest value.

The case when *low* is negative infinity is very similar. Let's now suppose that *low* and *high* are both finite numbers. If they are equal, then the double value *d* is exactly representable in the single precision format and the rounding in the same direction in single precision has the same result, *d*, but we can't infer the rounded value in other directions.

Suppose the statement *low* < *high* holds true. Of course, in round to negative infinity, our result is *low*, while in round to positive infinity, the result is *high*. The we make our best effort to find out the single precision round to nearest value. For this we can compute the middle of the interval *mid* in double precision, and this is an exact operations because the double precision value set includes the mid point of any interval of two consecutive single precision numbers. If *d* < *mid*, *low* is the value we are looking for. If *d* > *mid*, the value is *high*. Otherwise it holds true that *d* == *mid*. If *d* was the result of rounding to negative infinity then we know for sure that *mid* <= *this value* < *high*. It case *mid* == *value* and *high* is odd our result would be *low*; other it is high. Because we don't actually know the value (this as a real number), we only infer that the result is *high* if *high* is even. If *d* was the result of a round to positive infinity, the judgement is similar. In case of a round to nearest rounding, we have no clue about the rounding to single precision format in any direction because the actual value might be less, equal or greater then *mid*.

RoundRational

All the rational specific rounding implementation resides in here. This class is not safe for use in concurrent threads. It could extend RoundNumberBase, but using RoundNumber as the base class adds a lot of precomputed values with only a very small performance hit for the single call scenario.

For floating point numbers, the negative rationals are rounded in terms of their absolute value, with the opposite rounding direction. This is illustrated below:

public class RoundRational extends RoundNumber {

```
    private final Rational r;
    private final int[] intTemp = new int[2];

    public RoundRational(Rational r) {
        this.r = r.reduced();

        if (r.getExactDoubleValue() != null) {
            Arrays.fill(getDoubleValues(), r.getExactDoubleValue().doubleValue());
            Arrays.fill(getComputedDoubles(), true);
        }
    }

    public Rational getRational() {
        return r;
    }

    protected double computeDoubleValue() {
        assert r.getExactDoubleValue() == null : "doubleValue() should have already
been computed";

        final double d;

        switch (r.signum()) {
            case -1:
                d = -positiveDoubleValue(2 - directionIndex(), r.negate());
                break;
            case 0:
                d = +0.0;
                break;
            case 1:
                d = positiveDoubleValue(directionIndex(), r);
                break;
            default:
                throw new InternalError("r.signum() = " + r.signum());
        }

        return d;
    }

    private double positiveDoubleValue(int direction, Rational rational)

    protected float computeFloatValue() {
        final float f;

        switch (r.signum()) {
            case -1:
                f = -positiveFloatValue(2 - directionIndex(), r.negate());
                break;
            case 0:
```

```
            f = +0.0F;
            break;
        case 1:
            f = positiveFloatValue(directionIndex(), r);
            break;
        default:
            throw new InternalError("r.signum() = " + r.signum());
    }

    return f;
}

private float positiveFloatValue(int direction, Rational rational)

/**
 * Cached but not precomputed.
 */
protected BigInteger computeBigIntegerValue() {
    BigInteger v;

    switch (directionIndex()) {
        case 0:
            v = r.floor();
            break;
        case 1:
            if (r.numeratorModDenominator().signum() == 0) {
                v = r.floor();
            } else {
                // floor < v < ceil
                BigInteger twoMid = r.floor().shiftLeft(1).add(BigInteger.ONE);
                Rational twoMidRational = Rational.create(twoMid, BigInteger.ONE);
                Rational twoThis = r.multiplyTwoPower(1);

                int cmp = twoThis.compareTo(twoMidRational);

                switch (cmp) {
                    case -1:
                        v = r.floor();
                        break;
                    case 0:
                        if (r.floor().testBit(0)) {
                            // floor is odd
                            v = r.ceil();
                        } else {
                            // floor is even
                            v = r.floor();
                        }
                        break;
                    case 1:
                        v = r.ceil();
```

```
                        break;
                    default:
                        throw new InternalError("cmp = " + cmp);
                }
            }
            break;
        case 2:
            v = r.ceil();
            break;
        default:
            throw new InternalError();
    }

    return v;
}

}
```

The rounding to *BigInteger* is straightforward and we do not explain it here further. Let's concentrate on the *double positiveDoubleValue(int direction, Rational rational)* method. That should suffice because *float positiveFloatValue(int direction, Rational rational)* is very similar.

A very important method we use from the *Rational* class is *public void log2Interval (int[] a)*. As described in the javadoc, if this rational is an exact two power (this = pow2(n)), then *a[0] = a[1] = n*. Otherwise, the *a[0]* and *a[1]* values are computed such that *a[1] = a[0] + 1* and *pow2(a[0])* < this < *pow2(a[1])*. The method is safe for use in concurrent threads so we don't have to deal with threading issues when using it. Its implementation is rather straightforward:

```
public void log2Interval(int[] a) {
    if (signum() <= 0) {
        throw new ArithmeticException("log2(x) only for x > 0");
    }

    if (!log2Computed) {
        synchronized (this) {
            if (!log2Computed) {
                // could be cached
                int cmp = numerator.compareTo(denominator);

                int shcmp;

                switch (cmp) {
                    case -1:
                        BigInteger shiftedNumerator = numerator.shiftLeft(
                                denominator.bitLength() - numerator.bitLength());
                        shcmp = shiftedNumerator.compareTo(denominator);
                        break;
                    case 0:
```

```
                shcmp = 0;
                assert numerator.bitLength() == denominator.bitLength();
                break;
            case +1:
                BigInteger shiftedDenominator = denominator.shiftLeft(
                        numerator.bitLength() - denominator.bitLength());
                shcmp = numerator.compareTo(shiftedDenominator);
                break;
            default:
                throw new InternalError("cmp = " + cmp);
            }

            int bitLengthDiff = numerator.bitLength() - denominator.bitLength();

            switch (shcmp) {
            case -1:
                log2Low = bitLengthDiff - 1;
                log2High = bitLengthDiff;
                break;
            case 0:
                log2Low = log2High = bitLengthDiff;
                break;
            case +1:
                log2Low = bitLengthDiff;
                log2High = bitLengthDiff + 1;
                break;
            default:
                throw new InternalError("shcmp = " + shcmp);
            }

            assert log2Low <= log2High;
            log2Computed = true;
        }
    }
}

a[0] = log2Low;
a[1] = log2High;
}
```

Let's start explaining the *positiveDoubleValue* method implementation.We will only explain the round to negative infinity case, because the other too are very similar. The implementation for the rounding to negative infinity is shown below:

```
if (intTemp[0] < Formulas.DOUBLE_MIN_TWO_EXPONENT) {
    assert intTemp[1] <= Formulas.DOUBLE_MIN_TWO_EXPONENT;
    assert rational.compareTo(Rational.DOUBLE_MIN_VALUE) == -1;
    d = +0.0;
} else {
```

```
            assert rational.compareTo(Rational.DOUBLE_MIN_VALUE) >= 0;

            // quick check for the intTemp[0] >
Formulas.DOUBLE_MAX_TWO_EXPONENT condition
            if (intTemp[0] > Formulas.DOUBLE_MAX_TWO_EXPONENT
                    || rational.compareTo(Rational.DOUBLE_MAX_VALUE) >= 0) {
                // rational >= Double.MAX_VALUE
                d = Double.MAX_VALUE;
            } else {
                // Double.MIN_VALUE <= rational < Double.MAX_VALUE
                assert Formulas.DOUBLE_MIN_TWO_EXPONENT <= intTemp[0]
                        && intTemp[1] <=
Formulas.DOUBLE_MAX_TWO_EXPONENT + 1;
                assert rational.compareTo(Rational.DOUBLE_MAX_VALUE) == -1;

                if (intTemp[0] == intTemp[1]) {
                    // exact double value
                    d = Formulas.pow2(intTemp[0]);
                } else {
                    // pow2(intTemp[0]) < rational < pow2(intTemp[1])
                    assert rational.compareTo(Rational.create(Formulas.pow2(intTemp
[0]))) == +1
                            && rational.compareTo(Rational.create(Formulas.pow2
(intTemp[1]))) == -1;
                    if (intTemp[1] <= 1 - Formulas.DOUBLE_BIAS) {
                        // the result is a subnormal number > Double.MIN_VALUE
                        assert rational.compareTo(Rational.DOUBLE_MIN_NORMAL)
== -1;
                        /*
                        pow2(-1074) < rational < pow2(-1022) implies 1 < rational *
pow2(1074) < pow2(52)
                        The fraction is floor(rational * pow2(1074)).
                        */

                        Rational shifted = rational.multiplyTwoPower(1074);
                        BigInteger shiftedFloor = shifted.floor();
                        assert shiftedFloor.compareTo(BigInteger.ONE) >= 0
                                && shiftedFloor.compareTo(Formulas.bigintPow2(52)) ==
-1;
                        assert shiftedFloor.bitLength() <= 52;

                        // exact narrowing primitive conversion
                        long bits = shiftedFloor.longValue();
                        d = Double.longBitsToDouble(bits);
                        assert Double.MIN_VALUE <= d && d <=
Formulas.DOUBLE_MAX_SUBNORMAL;
                    } else {
                        // Formulas.DOUBLE_MIN_NORMAL < rational <
Double.MAX_VALUE
                        // Formulas.DOUBLE_MIN_NORMAL <= result <
```

```
Double.MAX_VALUE
                        assert rational.compareTo(Rational.DOUBLE_MIN_NORMAL)
== +1;
                        // the exponent is intTemp[0]
                        Rational shifted = rational.multiplyTwoPower
(Formulas.N_DOUBLE - 1 - intTemp[0]);
                        BigInteger shiftedFloor = shifted.floor();
                        assert shiftedFloor.compareTo(Formulas.bigintPow2
(Formulas.N_DOUBLE - 1)) >= 0
                              && shiftedFloor.compareTo(Formulas.bigintPow2
(Formulas.N_DOUBLE)) == -1;

                        // exact narrowing primitive conversion
                        long shiftedFloorBits = shiftedFloor.longValue();

                        // exact conversion
                        double shiftedFloorAsDouble = (double) shiftedFloorBits;

                        // exact
                        d = shiftedFloorAsDouble / Formulas.pow2
(Formulas.N_DOUBLE - 1)
                              * Formulas.pow2(intTemp[0]);

                        assert Formulas.DOUBLE_MIN_NORMAL <= d && d <
Double.MAX_VALUE;
                    }
                }
            }
            assert Formulas.isFinite(d);
        }
```

Because *Double.MIN_VALUE* is an exact two power, we can very efficiently find out the ordering of *rational* and *Rational.DOUBLE_MIN_VALUE* by comparing *intTemp [0]* against *Formulas.DOUBLE_MIN_TWO_EXPONENT*. If *rational* is smaller than *Rational.DOUBLE_MIN_VALUE*, then the result is certainly *+0.0*. Otherwise, if *rational* is greater than or equal to *Rational.DOUBLE_MAX_VALUE*, then the result is *Double.MAX_VALUE*. As a trick, we do not directly compare rational to Rational.DOUBLE_MAX_VALUE, but we first try to find out if *intTemp[0] > Formulas.DOUBLE_MAX_TWO_EXPONENT*. Thus, in case this very cheap test succeeds, we avoid the more expensive rational comparison. If *Double.MIN_VALUE <= rational < Double.MAX_VALUE*, we test for an exact two power double value. If exact, then the result is *Formulas.pow2(intTemp[0])*.
Otherwise, it holds true that *pow2(intTemp[0]) < rational < pow2(intTemp[1])* and, of course, the result will have to be a double precision finite floating point number. The result is a subnormal number iff *rational < Rational.DOUBLE_MIN_NORMAL*. Of course, we check it very cheaply using the equivalend boolean expression *intTemp [1] <= 1 – Formulas.DOUBLE_BIAS* instead.
In case the result must truly be subnormal, we only need to find out the fraction of the result. But the fraction part is actually *floor(rational * pow2(1074))*. We know that *pow2(-1074) < rational < pow2(-1022)* and that implies *1 < rational * pow2(1074) <*

pow2(52), so not only *floor(rational * pow2(1074))* is an exact long value, but it is actually the fraction for the subnormal result as it fits quite nice into *52* bits. The result in this case is *floor(rational * pow2(1074))*, exactly represented as a long primitive value and transformed using the method *Double.longBitsToDouble* to a double precision floating point number.
Otherwise the result must be a normal number, but it can't be *Double.MAX_VALUE*. Basically we find the fraction part of the result, multiplied by *pow2(52)* to be an integer value, cast it exactly to double and multiply it with *pow2(intTemp[0])*, which is the correct exponent. Given that *shiftedFloor = rational.multiplyTwoPower (Formulas.N_DOUBLE - 1 – intTemp[0]).floor()*, it holds true that *shiftedFloor >= pow2(52)* and *shiftedFloor < pow2(53)*, so exactly 53 bits are required to represent the *shiftedFloor* integer number. That constitutes a guarantee that the operation *(double) shiftedFloor.longValue() / Formulas.pow2(Formulas.N_DOUBLE – 1) * Formulas.pow2(intTemp[0])* is exactly computed in double precision floating point format.

Conclusions

The *Rational* class presented above hides all these implementation details under the *Number* methods overrides and overloads. The casual user of the class does not need to understand any of these implementation details, but he only needs to know about the standard *Number* methods and how to create a *Rational* instance using the constructors of factory methods. The class is extensively used with success by the simple continued fractions package in the same project. For more information, the documentation in the *docs* directory of the distribution and the javadocs can be consulted.

Project: ***PreciseFloating*** *- Author:* ***Daniel Aioanei*** *(aioaneid@go.ro)*