A Fraction Class

Objectives	1. Create a Fraction data type that performs *exact arithmetic* using Fractions. This is something the Java API doesn't have. 2. Practice fundamental methods like equals, toString, and compareTo. 3. Learn about *extended arithmetic*, part of the IEEE Floating Point standard.
Assignment	Write a Fraction class that performs exact fraction arithmetic.
What to Submit	1. Upload the project, including source for the **`Fraction`** class to a Github repository named "**fraction**". Please use this repository name *exactly*. Otherwise, your project won't be graded. 2. Use the package (folder) named **`fraction`** for your code. That means the source code is in **`src/fraction/Fraction.java`**
Evaluation	1. Correctness and completeness of code. 2. Quality of Javadoc comments and coding style.
Individual Work	All submitted work must be your own. Anyone that copies work from another and submit it will receive "F" for the course *and be reported to the university*. You can discuss concepts, method of solution, and conceptual design, but not share actual solutions or code. Do not ask other students for help understanding the problem; ask the TAs or instructor.

1. Problem Description

Java has BigInteger and BigDecimal classes for arbitrary precision arithmetic. Using BigInteger you can add arbitrarily large numbers without overflow

```
// What is the U.S. National debt, in Thai Baht? (17 Trillion times 33 Baht/USD)
BigInteger trillion = new BigInteger("1000000000000"); // preferred constructor
BigInteger debt = new BigInteger(17);                  // constructor using int
BigInteger debtBaht = debt.multiply(trillion).multiply( new BigInteger(33) );
System.out.println("US debt is " +  debtBaht.toString() + " Baht");
```

Software for financial applications use BigDecimal for money values to avoid round-off errors.

BigInteger and BigDecimal have add, subtract, multiply, divide, pow, and other methods for arithmetic. These methods all create a *new object* -- they <u>never</u> change the value of an existing object. That is, BigDecimal and BigInteger are *immutable* (unchangeable) just like Double or Integer.

```
BigDecimal c = new BigDecimal("299792458");  // meters per second
BigDecimal secs = new BigDecimal (31557600); // seconds per year
BigDecimal lightyear = c.multiply(secs);   // this does not change c or secs
```

But, Java doesn't have a class for exact arithmetic using fractions. We want a Fraction class that can do exact fraction arithmetic, such as:

$$\frac{1}{3} * \frac{17}{2+1/5} = \frac{85}{33}$$

Using a Fraction class, we can write:

```
Fraction a = new Fraction(1, 3);
Fraction b = new Fraction( 17 );
Fraction c = new Fraction(2).add( new Fraction(1,5) );
Fraction answer = a.multiply( b.divide(c) );
```

```
System.out.println( answer ); // calls answer.toString() which is "85/33"
```

We also want to allow *extended numbers* as in the IEEE Floating Point Standard:

$$\frac{2}{0}=\infty,\quad \frac{-3}{0}=-\infty,\quad \infty+3=\infty,\quad \frac{0}{0}=NaN,\quad \infty-\infty=NaN$$

We should be able to display a Fraction and test if it is Infinity or NaN, just like the Double class.
For example:

```
> Fraction f = new Fraction(20, 42);  // 20/42 = 10/21
> f.toString()
  10/21
> Fraction f2 = new Fraction(3);  // same as new Fraction(3,1)
> f2.toString()
  3                               // DON'T return "3/1"
> Fraction g = f.multiply(f2);
> g.toString()
  10/7
> g.isInfinite()                  // test for infinity. same as in Double class
  false
> Fraction inf = new Fraction(1,0);
> inf.isInfinite()                // test for infinity
  true
> inf.toString()
  Infinity
```

2. Important Properties of Fraction

1. Fractions should be initialized in *standard form*. That means the denominator ≥ 0, and numerator and denominator have *no common factors*. Example:

   new Fraction(10, -24) creates a fraction with `numerator=-5`, `denominator=8`.

   new Fraction(0, 20) creates a fraction with `numerator=0`, `denominator=1` (not 20).

   This is easy: in the constructor you compute the greatest common divisor (GCD) and remove it from both numerator and denominator.

2. Fractions are ***immutable***, just like Double. You can't change a Fraction after you create it.

   Arithmetic methods like add return a *new Fraction*, but they don't change the existing Fraction object.

3. We want to be able to compare fractions so we can test if one fraction is greater than another. Write a compareTo method (see *Fundamental Java Methods* handout) that does this (f and g are Fraction objects):

   f.compareTo(g) > 0 if f is greater than g

   = 0 if f is equal to g

   < 0 if f is less than g

   compareTo is anti-symmetric. If a.compareTo(b) > 0 then b.compareTo(a) < 0.

4. The toString() method should display a Fraction in *standard form* with no common factors.

- f = new Fraction(2, -6) f.toString() is "-1/3"
- f = new Fraction(3, 0) f.toString() is "Infinity" (not "Infinite")
- f = new Fraction(-3, 0) f.toString() is "-Infinity"

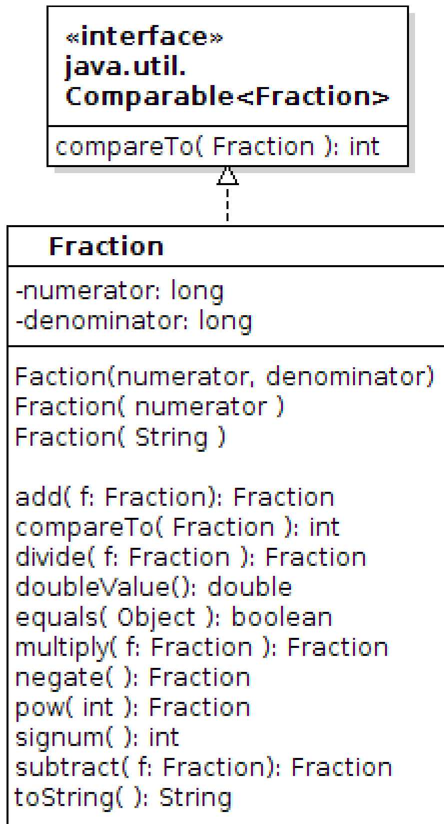
- f = new Fraction(3, 1) f.toString() is "3" (not "3/1")
- f = new Fraction(0, 0) f.toString() is "NaN"

5. Fractions support extended arithmetic. This means arithmetic involving the values Infinity and NaN.

   See below for details of extended arithmetic.

3. UML Diagram of Fraction Class

Your Fraction class should implement this UML diagram.



This means "Fraction implements Comparable<Fraction>", *which means that Fraction has a* **`compareTo`** *method.*

There are many methods, but most methods are very short.

4. Constructors and Methods

Your Fraction class must implement these constructors and methods:

4.1 Constructors

`Fraction( long n , long d )`	create a new fraction with value n / d . d can be zero.
`Fraction( long n )`	create a new fraction with integer value; this is same as **`Fraction(n,1)`**
`Fraction( String value )`	create a new fraction from a String value. The String must contain a long or long/long, such as "1234" or "-1234/5678". (Double and BigDecimal also have a String constructor.)

4.2 Public Methods

`Fraction add( Fraction f )`	return a new fraction that is sum of this fraction and f. Don't modify value of this fraction or f.

`Fraction subtract( Fraction f )`	return a new fraction that is difference of this fraction and f. Don't modify value of this fraction or f.
`Fraction multiply( Fraction f )`	return a new fraction that is product of this fraction and f.
`Fraction divide( Fraction f )`	return a new fraction that is this fraction divided by f.
`Fraction negate( )`	return a new fraction that is the *negative* of this Fraction. negate of Infinity is -Infinity. negate of NaN is NaN.
`Fraction pow(int n)`	return a new fraction that is this fraction raised to the n power. n may be zero or negative.
`int compareTo( Fraction f )`	compare this fraction to `f`. The return value should be: a.compareTo(b) < 0 if a is less than b a.compareTo(b) = 0 if a has same value as b a.compareTo(b) > 0 if a is greater than b a.compareTo(NaN) < 0 for any a != NaN
`double doubleValue( )`	return the value of this fraction as a double.
`boolean equals( Object obj )`	return true if obj is a Fraction and has the same value.
`int signum( )`	Return +1 if this fraction is greater than zero (including +Infinity), 0 if fraction is 0 or NaN, -1 if this fraction is less than zero (including -Infinity).
`String toString( )`	return a String representation of the fraction such as "3/8", with no spaces. If the denomiator is 1, just display the numerator. For example, "5" instead of "5/1". Return the String "`Infinity`", "`-Infinity`", or "`NaN`" for extended numbers. NOT "`Infinite`".
`Optional methods for extended arithmetic:`	
`boolean isNaN( )`	return true if this fraction is Not a Number (NaN).
`boolean isInfinite( )`	return true if this fraction is positive or negative infinity.
`Fraction.isNaN( Fraction f )` **`Fraction.isInfinite(Fraction f)`**	static versions of isNaN( ) and isInfinite( ), like in the Double class. This is easy -- just invoke the instance method.

4.3 Class Definition

The declaration of your Fraction class should look like this:

```
package fraction;
import java.util.Comparable;
/**
 * Numeric type for a fraction, with standard arithmetic operations.
 *
 * @author Your Name
 */
public class Fraction implements Comparable<Fraction>
```

5.1 Sample Method

```
/**
 * Divide this fraction by another fraction and return the result.
 * @param  f the fraction to divide into this fraction (divisor)
 * @return a new Fraction that is the result of this fraction divided by f.
 */
public Fraction divide( Fraction f ) {
    //NOTE the constructor will remove common factors, so we don't need to.
    // But, you may avoid overflow by removing common factors in the pairs
    // (this.numerator,f.numerator) and (this.denominator, f.denominator)
    // before multipling them together.
    return new Fraction( numerator*f.denominator ,
                         denominator*f.numerator );
}
```

5.2 Euclid's GCD Algorithm

The Fraction constructor needs to remove common factors from the numerator and denominator of the Fraction. For example, new Fraction(12,20) should be 3/5 (numerator=3, denominator=5). To put a fraction in normalized form, remove the greatest common divisor (GCD) from numerator and denominator, and make sure the denominator is not negative. The GCD of two values should always be > 0, even if both values are zero. Euclid's algorithm for GCD is:

```
gcd( a, b ):
    if ( a < 0 ) a = -a   // to avoid problems with sign
    while b != 0:
        remainder = a % b
        a = b
        b = remainder
    if a == 0 return 1
    return a
```

Wikipedia has an insightful article about Euclid's Algorithm for GCD. In Java:

```
static long gcd( long a, long b) {
    if (a < 0) a = -a;   // in Java, result of (a%b) always has same sign as a.
    while (b != 0) {
        long remainder = a % b;
        a = b;
        b = remainder;
    }
    return (a==0) ? 1L : a;  // always return a positive value
```

5.3 Extended Arithmetic (Optional)

This section explains the rules for extended arithmetic, involving Infinity (1/0), -Infinity, and NaN (0/0), which are part of the IEEE Floating Point standard.

You should try extended arithmetic using **double** or **Double** values in BlueJ. For example:

```
> double x = 2.0/0.0;
> x
  Infinity (double)
```

```
> Double.isInfinite(x)
  true
> x + 3
  Infinity (double)
> x * -1
  -Infinity (double)
> x * 0
  NaN
```

The IEEE standard defines 3 special value for *extended numbers*: Infinity, -Infinity, and NaN (Not a Number).

The meanings of the special values (and how they occur) are:

Infinity a value larger than any finite number. Infinity results from x/0 when x > 0, or creating a fraction that is too large to store.

-Infinity a value smaller than any finite number. -Infinity results from x/0 when x < 0, x*Infinity when x < 0, or an operation that produces a negative Fraction too large in magnitude to store.

NaN *not a number* and not +/-**Infinity**. **NaN** is the result of: 0/0, Infinity - Infinity, Infinity/Infinity, or 0 * Infinity. Any operation involving **NaN** results in **NaN**.

Other Rules for Extended Arithmetic

```
    x/Infinity = 0                  for any finite x including x=0
    Infinity + x = Infinity         for any finite x
    -Infinity + x = -Infinity       for any finite x
    Infinity + Infinity = Infinity
    -Infinity - Infinity = -Infinity
but Infinity - Infinity = NaN
    Infinity * 0 = NaN
```

compareTo() with extended values:

1. If x is any finite Fraction, then x.compareTo(Infinity) < 0, x.compareTo(Negative_Infinity) > 0.

2. Conversely, for any finite Fraction x, Infinity.compareTo(x) > 0 and Negative_Infinity.compareTo(x) < 0.

3. NaN is "bigger" than anything, including Infinity. So, NaN.compareTo(x) > 0 always.

6. Programming Hints

1. Make sure your constructors always initialize a new fraction in *normalized form*.
Normalized form is what you learned in elementary school. a/b is *normalized* when:

(a) b >= 0 (no negative denominator)

(b) a and b are relatively prime (no common factors. use the gcd to remove them).

2. Incremental code and test: First write the constructor and toString methods and test them. Then write `add` and `subtract` methods for finite numbers and test them. Then use a "truth table" to check if they return the correct result for extended numbers, too. In most cases, you will get the correct result even when one of the operands is an extended number! No special code or "if" tests are needed for most operations.

You should <u>not</u> need to write a lot of "if (denominator == 0) ..." or other special logic.

After the `add` and `subtract` methods are 100% correct, then implement other methods like `multiply` and `divide`.

7. A Fraction With Unlimited Precision (Optional)

When we add, subtract, or multiply fractions the denominator tends to get bigger. Eventually it will become too big for a "long" (greater than Long.MAX_VALUE) and the results will be incorrect. You can implement a Fraction that never overflows using BigInteger for the numerator and denominator. This is a change in the implementation only. The constructor and methods don't change their external signature (parameters).

8. JUnit Tests

Some test cases are provided as a JUnit test suite. I will post this separately.

Copy (drag and drop) the JUnit file into BlueJ or Eclipse in the fraction package. In BlueJ it will create a (green) JUnit test file. In Eclipse, you'll get a lot of errors that can be fixed by adding the JUnit 4 library to the project (Project -> Properties -> Build path -> Libraries).

To run tests in BlueJ: Right click and select Test All to run the tests. In Eclipse: right click on the test class and choose Run As -> JUnit Tests.

9. Graphical Fraction Calculator and Fraction Console (Interpretter)

There is a graphical Fraction calculator for the Fraction class in the file **`FractionCalc.jar`**.

It uses *your* Fraction class, so it won't work unless your Fraction class is complete.

To use this with *your* Fraction class, do the following:

1. Copy FractionCalc.jar into the top directory of your fraction project; don't put it in the fraction package.

2. Run it from the command line like this:

```
cmd> cd /somepath/workspace/fraction
cmd> java -cp .;FractionCalc.jar calculator.FractionCalc
```

On Mac OS or Linux use:

```
cmd> java -cp .:FractionCalc.jar calculator.FractionCalc
```

The source code for both a graphical UI and console UI is also provided.

FractionConsole

The source code contains a FractionConsole class that lets you type math expressions and see the results as Fractions. It also uses your Fraction class.

Unzip the source code into your src/ directory. It creates a subdirectory named **`calculator`** that contains all the source code. You should (**must**) exclude this code from your Git repository, so edit your `.gitignore` file and add:

```
# add to .gitignore to ignore demo code
src/calculator
FractionCalc.jar
```