A Fraction Class

Assignment	Write a Fraction class that performs exact fraction arithmetic.
Objectives	1. Practice fundamental methods like equals, toString, and compareTo. 2. Create a useful Fraction data type that allows *exact arithmetic* using Fractions. This is something the Java API doesn't have. 3. Learn about *extended numbers*, part of the IEEE Floating Point standard.
What to Submit	Create Github repo here: **`https://classroom.github.com/a/TsvUlc6K`** Your repo URL will be https://github.com/OOP2018/pa1-fraction-*yourlogin*. Add your own project code, along with a README.md and .gitignore.
Evaluation	1. Correctness of code. 2. Quality of code, coding style, and Javadoc comments. 3. Correctness and completeness of repository. Code in src/, useful README.md, .gitignore, and **no** compiler output (*.class).
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).multiply(trillion);
BigInteger debtBaht = debt.multiply( new BigInteger(33) );
System.out.println("US debt is " +  debtBaht.toString() + " Baht");
```

Many 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 values all create a *new object* -- they don't change the value of an existing BigInteger or BigDecimal. That is, BigDecimal and BigInteger are *immutable* (unchangeable) just like Double and Integer.

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 compute the above as:

```
Fraction a = new Fraction( 1, 3);                        // = 1/3
Fraction b = new Fraction( 17 );                         // = 17
Fraction c = new Fraction(2).add( new Fraction(1,5) ); // = 2+(1/5)
Fraction answer = a.multiply( b.divide(c) );             // = (1/3)*17/(2+(1/5))
System.out.println( answer );    // calls toString() which is "85/33"
```

We also want to allow *extended numbers* as in the IEEE Floating Point Standard. Extended numbers represent the values of +infinity, -infinity, and not-a-number (NaN). These values occur when you perform operations such as:

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

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(100,70);
> f.toString()
  10/7
> f.isInfinite()                  // test for infinity. same as in Double class
  false
> Fraction g = new Fraction(1,0);
> g.isInfinite()                  // test for infinity
  true
> inf.toString()
  Infinity
```

2. Important Properties of Fraction

1. 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.

2. Fractions are stored 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).

   Similarly, `Infinity` and `NaN` should have a unique form. This will simplify your other methods.

   Hint: if the constructor *normalizes* a new fraction by removing common factors (standard form) then the other operations don't need to!

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

   Special cases: NaN is "bigger" than anything, including Infinity. So, NaN.compareTo(Infinity) > 0.

   Infinity is bigger than any finite value (obviously), so Infinity.compareTo(new Fraction(10000000)) >0.

   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.

6. Just like the Double class, Fraction has a constructor that accepts a String and creates a Fraction from the value. For example:

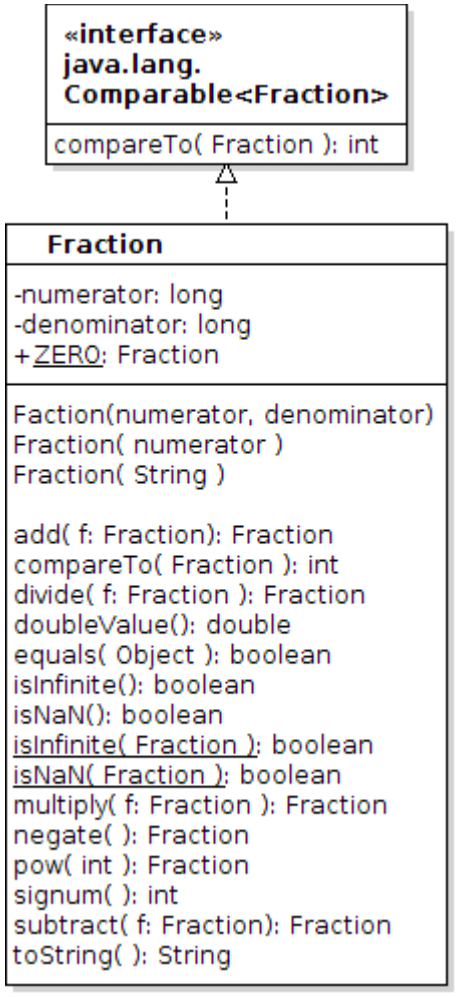
   Fraction f = new Fraction("3/7"); // same as new Fraction(3,7)

   Fraction g = new Fraction("4"); // same as new Fraction(4,1)

   Fraction h = new Fraction("3.1415"); // same as new Fraction(31415, 10000);

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 . n and 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 ("123"), long/long ("12/34"), or a decimal value ("12.34"). This is similar to the BigDecimal class.

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 power n. 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(Fraction.NaN) < 0 for any a != NaN
`double doubleValue( )`	return value of this fraction as a `double`. May be inaccurate.
`boolean equals( Object obj )`	return `true` if obj is a Fraction and has the same value.
`boolean isNaN( )`	return true if this fraction is Not a Number (NaN). Internally, NaN is represented as numerator = denominator = 0.
`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( ) to make it easy to test values, like in the Double class. This is Easy.
`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, with no spaces. Return the String "`Infinity`", "`-Infinity`", or "`NaN`" for extended numbers. NOT "`Infinite`".

Public Static Value

`public static final Fraction ZERO;`	Constant with the value 0. A convenience for applications using Fraction, since zero occurs a lot.

4.3 Class Definition

The declaration of your Fraction class should look like this.

```
package fraction;

/**
 * Javadoc comment describing this class.
 *
 * @author Your Name
 */
public class Fraction implements Comparable<Fraction>
```

Comparable is an interface in the package java.lang. To implement it you must write the method "int compareTo(Fraction other)", as described above. compareTo is used for comparisons and sorting.

5.1 Sample Method

```
/**
 * Multiply this fraction by another fraction and return the result.
 * @param  f the fraction to multiply by this fraction.
 * @return a new Fraction that is the result of this fraction multiplied by f.
 */
public Fraction multiply( Fraction f ) {
    //TODO verify that multiply involving NaN or Infinity yields correct result
    //NOTE the constructor will remove common factors, so we don't need to
    return new Fraction( this.numerator*f.numerator ,
                         this.denominator*f.denominator );
}
```

This code works but has one weakness: the computation might overflow a **long** even though the final result would fit in longs. For example:

```
// in Java 7+ you cam put _ in numbers to improve readability
f = new Fraction(100_000_000_000L, 111_111_111_111L);
g = new Fraction(333_333_333_333L,  5_000_000_000L);
Fraction product = f.multiply(g);
```

The simple code computes 100,000,000,000 * 333,333,333,333 and 111,111,111,111*5,000,000,000. If we could remove common factors first, we'd only need to compute numerator = 100*3 and denominator = 1*5. This reduces the chance of the computation exceeding the largest long.

Your code doesn't have to do this, but if you want to write the most robust Fraction code, consider it. For example

```
public Fraction multiply( Fraction f ) {
    // remove common factor between this numerator and other denominator.
    // gcd is always > 0, even if both args are 0.
    // This won't change value of the product, since we are removing same
    // factor from numerator of one fraction and denominator of the other.
    long gcd = gcd(this.numerator, f.denominator);
    long numer1 = this.numerator/gcd;
    long denom2 = f.denominator/gcd;
    // do same thing for pair using this numerator and other denominator
    gcd = gcd(this.denominator, f.numerator);
    long denom1 = this.denominator/gcd;
    long numer2 = f.numerator/gcd;
    // now compute new fraction with common factors removed
    return new Fraction( numer1*numer2, denom1*denom2 );
}
```

5.2 Pseudo-code for Euclid's 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`. The constructor should remove the greatest common divisor (GCD) of the numerator and denominator. The GCD of two values should always be > 0, even if both values are zero. Euclid's Algorithm, that efficiently finds the GCD of two integer values, is:

```
long gcd(a, b):
     while b != 0:
          remainder = a % b
          a = b
          b = remainder
     end while
     if a == 0 return 1
     return (a > 0) ? a : -a   // always return a positive value
```

Wikipedia has an insightful article about Euclid's Algorithm for GCD. Be careful using % (modulo) in Java. The result has the sign of the *numerator*. `15 % 7` is `1`, and `15 % -7` is `1`, but `-15 % 7` is `-6`.

5.3 Extended Arithmetic

The only tricky thing about Fraction is handling arithmetic with Infinity (1/0), -Infinity (-1/0), and NaN (0/0).

This section explains the rules for extended arithmetic.

Try extended arithmetic using **`double`** or **`Double`** values in BlueJ:

```
> double x = 2.0/0.0;
> x
  Infinity (double)
> Double.isInfinite(x)
  true
> x + 3
  Infinity (double)
> x * -1
  -Infinity (double)
> x * 0
  NaN
> double max = Double.MAX_VALUE;
> max
  1.797693134823157E+308
> 1.0000001*max
  Infinity (double)
```

The IEEE floating point 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 with magnitude too large 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
```

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).

(c) Every fraction has a unique form, e.g. 1/2, 3/6, -2/-4 are all stored as numerator=1, denominator=2. +Infinity, -Infinity, and NaN should always be stored in a unique form.

2. First write the **`add`** and **`subtract`** methods for finite numbers and test them. 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 or both operands is an extended number! But check this.
You should not need to write a lot of "if (denominator == 0) ..." or other special logic.

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

7. A Fraction With Unlimited Precision (Optional)

You can implement Fraction using long for the numerator and denominator, but sometimes the values will overflow when a value is greater than Long.MAX_VALUE. For a Fraction that never overflows, use BigInteger for the numerator and denominator.

8. JUnit Tests

Some test cases are provided as a JUnit test suite. This file is in the class week2 folder of class repository, named FractionTest.java.

Copy (drag and drop) this file into your project src/fraction directory. For Eclipse, Netbeans, and IntelliJ you will need to add the JUnit 4 library to the project "Build Path" (Eclipse) or "Libraries" (Netbeans). The IDEs include JUnit 4 libraries, but they aren't part of projects by default.

Please **don't add the JUnit test file** to your Github repository.

Verify Your Fraction Method Signatures

If you write the wrong method signature(s), JUnit will issue errors for those methods or you may not be able to run the tests at all. If you get errors from JUnit, check your method signatures using the table above.

9. Fraction Calculator Application (for Amusement)

There is a console interface for performing fraction arithmetic in the file **`FractionCalc.jar`**.

It uses ***your*** `fraction.Fraction` class, and your class must implement all the API methods above.

To run it, do:

1. Copy FractionCalc.jar someplace, such as the top directory of your Fraction project. **Don't** put it in the "src" directory or in the fraction package. Please **don't** add the JAR file to your Git repo, too.

2. The "main" class in the JAR file is **`calculator.Main`**. Since its going to use *your* Fraction class, you need to tell it where your code is when you run it.

Suppose that you have copied FractionCalc.jar to your pa1-fraction project directory, and your compiled code is in the "**`bin`**" directory (default location for Eclipse projects). So your project looks like:

```
pa1-fraction/
    bin/
    src/
    src/fraction/
    .gitignore
    FractionCalc.jar
```

Then you would type:

```
cmd> cd pa1-fraction
```

On MS Windows then type:

```
cmd> java -cp .;bin;FractionCalc.jar  calculator.Main
```

On Mac OS or Linux type:

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

The **`-cp`** flag (classpath) is used to add directories and JAR files to the "class path" that Java searches for classes it needs. The above command means "*add the current directory (.), the bin directory, and the contents of FractionCalc.jar to the search path*".

IntelliJ Users: the output directory is typically "**`out/production`**" instead of "**`bin`**". Use "out/production" in the classpath instead of "bin".

Netbeans Users: the output directory is typically "**`build/classes`**" instead of "**`bin`**". Use "build/classes" in the classpath instead of "bin".

You should see a message and a prompt to type something. Here is example:

```
Fraction Calculator.  Enter ? for help.
Input: 3/4 + 1/9
31/36
Input: x = 1/4 + 2/3
11/12
Input: 12*x
11
Input: x - 1/6
3/4
Input: y = 10/20
1/2
Input: test x > y
true
Input: x^5
161051/248832
```