Chapter 9, BigFraction

John M. Morrison

January 19, 2021

Contents

0 Case Study: An Extended-Precision Fraction Class

We have achieved several goals so far, the most important of which are understanding what make up a Java and Python classes and understanding the core both languages so as to be Turing-complete.

To tie everything together, we will do a case study of creating two classes class `BigFraction.java` and `BigFraction.py`, which will impleement extended-precison rational arithmetic in both languagess. This class will have a professional appearance and will have full documentation.

0.1 A Brief Orientation

Before we begin let us remind ourselves of some basic mathematical facts and provide a rationale for what we are about to do. We are all familiar with the *natural* (counting) numbers

$$\mathbb{N} = \{1, 2, 3, 4, ...\}.$$

We can also start counting at zero because we are C family language geeks with

$$\mathbb{N}_0 = \{0, 1, 2, 3, ...\}.$$

The set of all integers (with signs) is often denoted by $\mathbb{Z}$. Why the letter Z? This comes from the German word *zahlen*, meaning “to count.”

The `BigInteger` class in Java and the `int` type in Python create computational environments for computing in $\mathbb{Z}$ without danger of overflow, unless you really go bananas.

The *rational numbers* consist of all numbers that can be represented as a ratio of integers; the symbol used for them is $\mathbb{Q}$. The ‘Q’ is for “quotient.” So,

$$\mathbb{Q} = \{m/n : m \in \mathbb{Z}, n \in \mathbb{N}, n \neq 0\}.$$

The `BigFraction` classes will create an environment for computing in $\mathbb{Q}$ similar to that which `BigInteger` provides for $\mathbb{Z}$.

1 Starting BigFraction.py

We begin by creating a class BigFraction in a file BigFraction.py.

```
class BigFraction:
    pass
```

What does a fraction need to know? It needs to know its numerator and denominator.

```
class BigFraction:
    def __init__(self, num, denom):
        this.num = num
        this.denom = denom
```

The other thing we should do is to give our class a string representatin.

```
class BigFraction:
    def __init__(self, num, denom):
        this.num = num
        this.denom = denom
    def __str__(self):
        return f"{self.num}/{self.denom}"
    def __repr__(self):
        return f"BigFraction({self.num}, {self.denom})"
```

Now let's test our new class in the interactive shell.

```
>>> from BigFraction import BigFraction
>>> b = BigFraction(1,2)
>>> b
BigFraction(1, 2)
>>> print(b)
1/2
>>> b = BigFraction(5,10)
>>> b
BigFraction(5, 10)
>>> print(b)
5/10
>>> b = BigFraction(-2,-4)
>>> print(b)
-2/-4
```

Uh oh. We can foresee problems here. Fractions should be stored in a fully reduced form. And, let's get the negative upstairs; this will turn out to be very beneficial down the road. So, we will put our fractions in a canonical form: fully reduced and any negative in the numerator.

Also, let us raise an error if the client programmer attempts to create a fraction with a zero denomintor.

1.1 Reducing Fractions

Ah, we are now back in the Miss Wormwood days of elementary school. She showed you a fraction such as

$$\frac{32}{12}$$

and you are supposed to reduce it. Well, both the top and bottom are even, so

$$\frac{32}{12} = \frac{16}{6}.$$

Hey we can do that again, and the result is

$$\frac{16}{6} = \frac{8}{3}.$$

Since 8 and 3 have no common factors, we are done.

This, however, is not going to cut the computational mustard. What if you have a fraction with 1000 digits on the top and bottom? Are you going to hunt for the common factors one by one and keep reducing? That would be a joyless slog to code as well as being baronially wasteful. We need a better way.

Suppose that a and b are integers that are not both zero. We define the *greatest common divisor* of a and b to be the largest positive integer dividing both evenly. Such a thing exsts, because 1 is a divisor of every integer. We will denote this function by $\gcd(a, b)$. The two quantities $a/\gcd(a, b)$ and $b/\gcd(a, b)$ are both integers. Also, they have no other common factor than 1. Now you see the *raison d'etre* for our interest in the greatest common divisor. If we have a fraction and we compute the greatest common divisor of its numerator and denominator, the result is a fully-reduced fraction.

Being avid Pythonista, we might take this approach. Suppose we have a and b. We could start at 2 and see if 2 is a common divisor of a and b. If it is, we can keep track of that fact in a variable. We then go to 3 and repeat this procedure. We stop when we get to the smaller of a and b. This is a pretty hackish approach, but let's give it a whirl.

Programming Exercise Implement this function in Python. For what size of numbers does this really begin to bog down?

Where's the catch? Again, consider the case of a couple of integers, each having hundreds of digits. This method could take an eternity. It's time for a little math! We are going to state and prove a simple little theorem that is the key to a fast gcd calculation.

First, let's talk a little bit about divisibility. We will use the notation $a \mid b$ to indicate that a divides b evenly; equivalently $b\%a = 0$. This means that there is some integer q so that $aq = b$.

Suppose that d is a common divisor of a and b You can choose integers s and t so that $ds = a$ and $dt = b$. Now suppose x and y are any integers. Then

$$ax + by = ads + bty = d(as + by);$$

since $as + by$ is an integer, we have $d|as + by$. The denouement: Any common divisor of a and b will also be a divisor of $ax + by$ for any integers x and y.

Let's call such things as $ax + by$ *integer combinations* of a and b. What we have show is that if d is a common divisor of a and b, it is a divisor of any integer combination of a and b.

Theorem. *Suppose that a, b, q, and r are integers and that $b = aq + r$. Then* $\gcd(b, a) = \gcd(a, r)$.

Proof. Suppose that d is a common divisor of a and r. Since $b = aq + r$, we have represented b as an integer combination of a and r. Therefore $d \mid b$. We conclude that d is a common divisor of a and b. We have just shown that every common divisor of a and r is a common divisor of a and b.

Now suppose d is common divisor of a and b. Since $b = aq + r$, $r = b - aq$. The integer r is an integer combination of a and b; therefore $d \mid r$. We have just shown that d is a common divisor of a and r. We have shown that every common divisor of a and b is a common divisor of a and r.

This tells us that a and b have exactly the same common divisors as a and r. We conclude $\gcd(b, a) = \gcd(a, r)$.

1.2 Speeding things up

One thing we know is that for any integer b and and non-zero integer a,

$$b = a * (b//a) + b\%a.$$

Here is another useful fact, $\gcd(a, 0) = |a|$, provided that $a \neq 0$. Let's compute $\gcd(1048576, 7776)$. We will use Python as our calculator. You should try this on another pair of big numbers.

```
>>> a = 7776
>>> b = 1048576
>>> remainder = 1048576 % 7776
>>> remainder
6592
>>> b, a = a, remainder
>>> b
7776
>>> a
6592
>>> remainder = b%a
>>> b, a = a, remainder
>>> a
1184
>>> b
6592
>>> remainder = b%a
>>> b, a = a, remainder
>>> b
1184
>>> a
672
>>> remainder = b%a
>>> b, a = a, remainder
>>> b
672
>>> a
512
>>> remainder = b%a
>>> b, a = a, remainder
>>> b
512
>>> a
160
>>> remainder = b%a
>>> b, a = a, remainder
>>> b
160
>>> a
32
>>> remainder = b%a
>>> b, a = a, remainder
>>> a
0
>>> b
32
```

Thhis gives us a chain of equalities.

$$\gcd(1048576, 7776) = \gcd(7776, 6592) = \cdots = \gcd(32, 0).$$

Since $\gcd(32, 0) = 32$, we are done. It seems we have a loop here

```
while a > 0:
        b, a = a, b%a
```

Changing signs does not change the gcd; to wit $\gcd(\pm a, \pm b) = \gcd(a, b)$, so we can strip off any negative signs. Also, let's raise an error if some reckless client tries to compute $\gcd(0, 0)$. Now for the coup d'grace.

```
def gcd(a,b):
        if a == 0 and b ==  0:
                raise ValueError
        if a < 0:
                a = -a
        if b < 0:
                b = -b
        while a > 0:
                b, a = a, b%a
        return b
```

Place this in a file named `number_theory.py`. Import it and test it. Ooh, it's quick.

```
>>> from number_theory import gcd
>>> gcd(1048576, 7776)
32
>>> gcd(323980490348, 32980398423123456)
4
```

This algorithm is called *Euclid's Algorithm.* The slowest it can work is the case of two adjacent fibonacci numbers. Since these grow exponentially, the number of iterations is at worst proportional to $\log(n)$, where n is the larger of the two numbers.

1.3 Finishing `__init__`

Now let us write this method. We will take the addional step of kicking any negative upstars. We will rase an error if a zero denominator gets passed in.

```
def __init__(self, num, denom)
    if denom == 0:
            raise ValueError
        if denom < 0:
            num = -num
            denom = -denom
        d = gcd(num, denom)
        self.num = num//d
        self.denom = denom//d
```

This method has the desirable property of storing a fraction in the canonical form we specified. The fraction is stored fully reduced. Any negative is in the denominator. You will see that this design decision will pay dividendds down the road. The care we took here will benefit us very soon. Let us lay out the whole class that we have so far.

```
from number_theory import gcd
class BigFraction:
    def __init__(self, num, denom):
        if denom == 0:
            raise ValueError
        if denom < 0:
            num = -num
            denom = -denom
        d = gcd(num, denom)
        self.num = num//d
        self.denom = denom//d
    def __str__(self):
        return f"{self.num}/{self.denom}"
    def __repr__(self):
        return f"BigFraction({self.num}, {self.denom})"
```

```
>>> from BigFraction import BigFraction
>>> b = BigFraction(7776, 1048576)
>>> b
BigFraction(243, 32768)
>>> print(b)
243/32768
```

2 Starting `BigFraction.java`

In this section, we will build a big fraction class with the same capabilities as the Python class. Note that we will be using `BigInteger`s as numerator and denominator. So, let's get started. We will rough in some items.

```
import java.math.BigInteger;
public class BigFraction
```

```
{
    private BigInteger num;
    private BigInteger denom;
    public BigFraction(BigInteger num, BigInteger denom)
    {
    }
    public String toString()
    {
        return String.format("%s/%s", num, denom);
    }
}
```

Looking at the `BigInteger` docs, we notice several things. One is that `BigIntegerss` are immutable. We can ensure this by making our state variables `final`.

```
import java.math.BigInteger;
public class BigFraction
{
    private final BigInteger num;
    private final BigInteger denom;
    public BigFraction(BigInteger num, BigInteger denom)
    {
    }
    public String toString()
    {
        return String.format("%s/%s", num, denom);
    }
}
```

Another is that `BigInteger` has a `gcd` method. We can avail ourselves of that. We also have to do the correct things to negate a `BigInteger` and to checks its positivity or negativity.

Changing sign is easy; just use the `negate()` method. To check sign, it is handy to compare with the static constant `BigInteger.ZERO`. Dividing is done with the `divide` method.

Let us go to work on the constructor. We show the lines of Python and use them as a guide, translating into Java as we progress.

```
    public BigFraction(BigInteger num, BigInteger denom)
    {
        //if denom == 0:
            //raise ValueError
        if(denom.equals(BigInteger.ZERO))
        {
            throw new IllegalArgumentException();
        }
        //if denom < 0:
            //num = -num
            //denom = -denom
        if(denom.compareTo(BigInteger.ZERO) < 0)
        {
            num = num.negate();
            denom = denom.negate();
        }
        //d = gcd(num, denom)
        BigInteger d = num.gcd(denom);
        //self.num = num//d
        //self.denom = denom//d
        this.num = num.divide(d);
        this.denom = denom.divide(d);
    }
```

Observe that we can only initialize state once because the state variables are `final`. We worked with the constructor's parameters as local variables until the very end.

Now let's assemble our effort.

```
import java.math.BigInteger;
public class BigFraction
{
    private final BigInteger num;
    private final BigInteger denom;
    public BigFraction(BigInteger num, BigInteger denom)
    {
        if(denom.equals(BigInteger.ZERO))
        {
            throw new IllegalArgumentException();
        }
        if(denom.compareTo(BigInteger.ZERO) < 0)
        {
            num = num.negate();
            denom = denom.negate();
        }
        BigInteger d = num.gcd(denom);
        this.num = num.divide(d);
        this.denom = denom.divide(d);
    }
    @Override
    public String toString()
    {
        return String.format("%s/%s", num, denom);
    }
}
```

We are thinking that there might be a constructor that will take a integer and convert it into a `BigInteger`, but there isn't. However, if we look in the docs, we wil see this.

`public static BigInteger valueOf(long val)`
Returns a BigInteger whose value is equal to that of the specified long.
API Note:
This static factory method is provided in preference to a (long) constructor because it allows for reuse of frequently used BigIntegers.
Parameters:
`val` - value of the BigInteger to return.
Returns:
a BigInteger with the specified value.

We will crib from this strategy and make our own static factory method `BigFraction.valueOf(long num, long denom)`. This will save us work down the road.

```
import java.math.BigInteger;
public class BigFraction
{
    private final BigInteger num;
    private final BigInteger denom;
    public BigFraction(BigInteger num, BigInteger denom)
    {
        if(denom.equals(BigInteger.ZERO))
        {
            throw new IllegalArgumentException();
        }
        if(denom.compareTo(BigInteger.ZERO) < 0)
        {
            num = num.negate();
            denom = denom.negate();
        }
        BigInteger d = num.gcd(denom);
        this.num = num.divide(d);
        this.denom = denom.divide(d);
    }
    @Override
    public String toString()
    {
        return String.format("%s/%s", num, denom);
    }
    public static BigFraction valueOf(long num, long denom)
    {
        return new BigFraction(BigInteger.valueOf(num),
            BigInteger.valueOf(denom));
    }
}
```

Et voila! It works!

```
jshell> /open BigFraction.java

jshell> BigFraction b = BigFraction.valueOf(7776, 1048576);
b ==> 243/32768
```

Our two classes are at the same level of progress.

3 Look out Miss Wormwood! Arithmetic!

The aim of this section is to endoow our big fractions with the ability to do arithmetic. Let us focus on these five operations: +, -, *, / and exponentiation.

3.1 Addition

Recall that

$$\frac{a}{b} + \frac{c}{d} = \frac{ad + bc}{bd}.$$

Let's start on the Python side. To redefine addition, there is the dunder method `__add__`. So in our class we make the header as follows

```
    def __add__(self, other):
```

Both `self` and `other` will be `BigFractions` so each has a numerator and a denominator. We use the addition formula for fractions and compute the numerator for the sum as follows.

```
    top = self.num*other.denom + self.denom*other.num
```

Now let's get the denominator

```
    bottom = self.denom*other.denom
```

Now let's put the whole thing together and ship out a `BigFraction`.

```
    def __add__(self, other):
        top = self.num*other.denom + self.denom*other.num
        bottom = self.denom*other.denom
        return BigFraction(top, bottom)
```

Now we test our work.

```
>>> from BigFraction import BigFraction
>>> a = BigFraction(1,3)
>>> b = BigFraction(1,2)
>>> a + b
BigFraction(5, 6)
>>> print(a + b)
5/6
```

Now it's Java's turn. You cannot redefine operators in Java, so we will do as they did in `BigInteger` and create an `add` method. The lines of Python are show here.

```
    public BigFraction add(BigFraction that)
    {

        //top = self.num*other.denom + self.denom*other.num
        //bottom = self.denom*other.denom
        //return BigFraction(top, bottom)
    }
```

We now translate them.

```
    public BigFraction add(BigFraction that)
    {
        //top = self.num*other.denom + self.denom*other.num
        BigInteger top = num.multiply(that.denom).add(
            denom.multiply(that.num));
        //bottom = self.denom*other.denom
        BigInteger bottom = denom.multiply(that.denom);
        //return BigFraction(top, bottom)
        return new BigFraction(top, bottom);
    }
```

Now test it. You can delete the python comments from your `add` method.

```
jshell> /open BigFraction.java
jshell> BigFraction a = BigFraction.valueOf(1,2);
a ==> 1/2

jshell> BigFraction b = BigFraction.valueOf(1,3);
b ==> 1/3

jshell> a.add(b)
$5 ==> 5/6
```

3.2 Subtraction

This is fish in a barrel since

$$\frac{a}{b} - \frac{c}{d} = \frac{ad - bc}{bd}.$$

We just change the addition in the middle to a subtraction. Here is Python

```
    def __sub__(self, other):
        top = self.num*other.denom - self.denom*other.num
        bottom = self.denom*other.denom
        return BigFraction(top, bottom)
```

And here is Java.

```
    public BigFraction subtract(BigFraction that)
    {

        BigInteger top = num.multiply(that.denom).subtract(
            denom.multiply(that.num));
        BigInteger bottom = denom.multiply(that.denom);
        return new BigFraction(top, bottom);
    }
```

3.3 Multiplication

We all know that

$$\frac{a}{b} \cdot \frac{c}{d} = \frac{ac}{bd}.$$

We therefore proceed as follows in Python.

```
    def __mul__(self, other):
        top = self.num*other.num
        bottom = self.denom*other.denom
        return BigFraction(top, bottom)
```

Now for Java.

```
    public BigFraction multiply(BigFraction that)
    {
        BigInteger top = num.multiply(that.num);
        BigInteger bottom = denom.multiply(that.denom);
        return new BigFraction(top, bottom);
    }
```

3.4 Division

This is just "invert and multiply."

```
    def __truediv__(self, other):
        top = self.num*other.denom
        bottom = self.denom*other.num
        return BigFraction(top, bottom)

    public BigFraction divide(BigFraction that)
    {

        BigInteger top = num.multiply(that.denom);
        BigInteger bottom = denom.multiply(that.num);
        return new BigFraction(top, bottom);
    }
```

Let's test-drive the Python.

```
>>> b = BigFraction(1,4);
>>> c = BigFraction(1,5);
>>> b + c
BigFraction(9, 20)
>>> b - c
BigFraction(1, 20)
>>> b*c
BigFraction(1, 20)
>>> b/c
BigFraction(5, 4)
>>>
```

And now the Java.

```
jshell> /open BigFraction.java
jshell> BigFraction b = BigFraction.valueOf(1,4);
b ==> 1/4

jshell> BigFraction c = BigFraction.valueOf(1,5);
c ==> 1/5

jshell> b.add(c)
$5 ==> 9/20

jshell> b.subtract(c)
$6 ==> 1/20

jshell> b.divide(c)
$7 ==> 5/4

jshell> b.multiply(c)
$8 ==> 1/20
```

3.5 Pow!

Holy exponent, Batman! It's time to compute powers! Note the argument is an integer.

```
    def __pow__(self, n):
        if n == 0:
            return BigFraction(1,1)
        if n > 0:
            return BigFraction(self.num**n, self.denom**n)
        n = -n
            return BigFraction(self.denom**n, self.num**n)

    public BigFraction pow(int n)
    {
        if n == 0:
        {
            return BigFraction.valueOf(1,1);
        }
        if n > 0:
        {
            return BigFraction(num.pow(n), denom.pow(n));
        }
        n = -n;
        return BigFraction(denom.pow(n), num.pow(n));
    }
```

Programming Exercises Add these methods to our existing `BigFraction` class. These will make our `BigFractions` more resemble `BigIntegers`.

1. Write a method `public BigInteger bigIntValue()` that divides the denominator into the numerator and which truncates towards zero.
2. Write the method `public BigFraction abs()` which returns the absolute value of this `BigFraction`.
3. Write a method `public BigFraction negate()` which returns a copy of this `BigFraction` with its sign changed.
4. Write the method `public BigFraction max(BigFraction)` which returns the larger of this `BigFraction` and `that`.
5. Write the method `public BigFraction min(BigFraction)` which returns the smaller of this `BigFraction` and `that`.
6. Write the method `public int signum()` which returns +1 if this `BigFraction` is positive, -1 if it is negative and 0 if it is zero.
7. Write the method `public int compareTo(BigFraction that)` which returns +1 if this `BigFraction` is larger than `that`, -1 if `that` is larger than this `BigFraction` and 0 if this `BigFraction` equals `that`.
8. Add a static method public BigFraction harmonic(int n) which computes the nth harmonic number. Throw an `IllegalArgumentException` if the client passes an `n` that is negative.
9. When should division throw an `IllegalArgumentException`? Add this feature to the class.
10. (Quite Challenging) Write the method `public double doubleValue()` which returns a floating point value for this `BigFraction`. It should return `Double.NEGATIVE_INFINITY` or `Double.POSITIVE_INFINITY` where appropriate. Test this very carefully; it is not easy to get it right.

4 Adding Static Constants

In Java, we have a notion of constness; this consists of a `final` variable pointing at an immutable object. Let's give our `BigFraction` class static constants `ZERO`, `ONE`, and `HALF`. We will create them at the top of the class and initialize them in a `static` block.

To do this just insert this right at the top of your class

```
public static final BigFraction ZERO;
public static final BigFraction HALF;
public static final BigFraction ONE;
static
{
        ZERO = BigFraction.valueOf(0,1);
        HALF = BigFraction.valueOf(1,2);
        ONE = BigFraction.valueOf(1,1);
    }
```

5 Documenting Your Code

These two classes could be quite useful to others. Now we need to give our users documentation so they can learn how to use our classes effectively.

We will document our Java code using the `javadoc` system. This system is easy to use and it creates a professional-looking API page that is similar in a appearance to the ones you see for the standard libraries.

Javadoc comments are delimited by the starting token `/**` and the ending token `*/`. C/C++ style comments delimited by `//` and `/* ........ */` do not appear on Javadoc pages. You may use HTML markup in your javadoc where needed.

Use Javadoc to document your *interface*, the public portion of your class. Do not javadoc `private` methods or state variables. We will produce a full javadoc page for our `BigFraction` class.

We will use docstings to document our Python class. These will be displayed in response to the `help` command or with the command

```
unix> python -m pydoc YourModule.py
```

5.1 Documenting `BigFraction.py`

Begin by describing the class right after the class header like so.

```
from number_theory import gcd
class BigFraction:
    """This is a class for performing extended-precision rational
arithmetic.  It includes a full suite of operators for arithmetic
and it produces a sortable objects, ordered by their numerical
values.

All BigFractions are stored in a canonical form: they are fully
reduced and any negative is stored in the numerator.
"""
```

Now for the `__init__` method.

```
    def __init__(self, num, denom):
        """This accepts two integer argments, a numerator
and a denominator.  A zero denominator will trigger a ValueError."""
        if denom == 0:
            raise ValueError
        if denom < 0:
            num = -num
            denom = -denom
        d = gcd(num, denom)
        self.num = num//d
        self.denom = denom//d
```

If you type the command `python -m pydoc BigFraction`, in a command window you will see this. Users can also see this documentaton in an interactive session using the `help` command.

```
NAME
    BigFraction

CLASSES
    builtins.object
        BigFraction

    class BigFraction(builtins.object)
     |  BigFraction(num, denom)
     |
     |  This is a class for performing extended-precision rational
     |  arithmetic.  It includes a full suite of operators for arithmetic
     |  and it produces a sortable objects, ordered by their numerical
     |  values.
     |
     |  All BigFractions are stored in a canonical form: they are fully
     |  reduced and any negative is stored in the numerator.
     |
     |  Methods defined here:
     |
     |  __add__(self, other)
     |
     |  __init__(self, num, denom)
     |      This accepts two integer argments, a numerator
     |      and a denominator.  A zero denominator will trigger a ValueError.
```

To exit, type `q`. Now we document the rest of the class

```
from number_theory import gcd
class BigFraction:
    """This is a class for performing extended-precision rational
arithmetic.  It includes a full suite of operators for arithmetic
and it produces a sortable objects, ordered by their numerical
values.

All BigFractions are stored in a canonical form: they are fully
reduced and any negative is stored in the numerator.

This class has three static constants

ZERO, the BigFraction representing 0
ONE, the BigFraction representing 1
HALF, the BigFraction representing 1/2
"""
    ZERO = None
    ONE = None
    HALF = None
    def __init__(self, num, denom):
        """This accepts two integer argments, a numerator
and a denominator.  A zero denominator will trigger a ValueError."""
        if denom == 0:
            raise ValueError
        if denom < 0:
            num = -num
            denom = -denom
        d = gcd(num, denom)
        self.num = num//d
        self.denom = denom//d
    def __str__(self):
        """This returns a string represenation for this
        BigFraction of the form numerator/denominator."""
        return f"{self.num}/{self.denom}"
    def __repr__(self):
        """This returns a string representation of the form
        BigFraction(numerator, denominator) suitable for the Python
        REPL"""
        return f"BigFraction({self.num}, {self.denom})"
    def __add__(self, other):
        """This defines + and  returns the sum of two BigFractions."""
        top = self.num*other.denom + self.denom*other.num
        bottom = self.denom*other.denom
        return BigFraction(top, bottom)
    def __sub__(self, other):
        """This defines - and  returns the difference of two BigFractions."""
        top = self.num*other.denom - self.denom*other.num
        bottom = self.denom*other.denom
        return BigFraction(top, bottom)
    def __mul__(self, other):
        """This defines * and  returns the product of two BigFractions."""
        top = self.num*other.num
        bottom = self.denom*other.denom
        return BigFraction(top, bottom)
    def __truediv__(self, other):
        """This defines / and  returns the quotient of two BigFractions."""
        top = self.num*other.denom
        bottom = self.denom*other.num
        return BigFraction(top, bottom)
    def __pow__(self, n):
        """This defines ** and returns this BigFraction raised to the
        nth power. This works for both positive and negative integers."""
        if n == 0:
            return BigFraction(1,1)
        if n > 0:
            return BigFraction(self.num**n, self.denom**n)
        n = -n
        return BigFraction(self.denom**n, self.num**n)
    @staticmethod
    def init_static():
        """initializes the static constants ZERO, HALF, and ONE."""
        BigFraction.ZERO = BigFraction(0,1)
        """The Big Fraction 0/1"""
        BigFraction.ONE  = BigFraction(1,1)
        """The Big Fraction 1/1"""
        BigFraction.HALF = BigFraction(1,2)
        """The Big Fraction 1/2"""
BigFraction.init_static()
```

5.2 Documenting `BigFraction.java`

The kind of class we have created represents a real extension of the Java language that could be useful to others. Now we need to give our class an API page so it has a professional appearance and so it can easily be used by others.

Javadoc comments are delimited by the starting token `/**` and the ending token `*/`. C/C++ style comments delimited by `//` and `/* ........ */` do not appear on Javadoc pages.

You may use HTML markup in your javadoc where needed.

Use Javadoc to document your *interface*, the public portion of your class. Do not javadoc `private` methods or state variables.

We will produce a full javadoc page for our `BigFraction` class. Let us begin with the constructors.

```
     /**
      * This constructor stores a <code>BigFraction</code> in
      * reduced form, with any negative factor appearing in
      * the numerator.
      * @param num the numerator of this <code>BigFraction</code>
      * @param denom the denomnator of this <code>BigFraction</code>
      * @throws IllegalArgumentException if a zero
      * denominator is passed in
      */
    public BigFraction(BigInteger num, BigInteger denom)
    {
        if(denom.equals(BigInteger.ZERO))
            throw new IllegalArgumentException();

        BigInteger d = num.gcd(denom);
        if(denom.compareTo(BigInteger.ZERO) < 0)
        {
            num = num.negate();
            denom = denom.negate();
        }
        this.num = num.divide(d);
        this.denom = denom.divide(d);
    }
    /**
     * This default constructor produces BigFraction 0/1.
     */
    public BigFraction()
    {
        this(BigInteger.ZERO,BigInteger.ONE);
    }
```

We see the special markup `@param`; this is the description given for each parameter. The markup `@throws` warns the client that an exception can be thrown by a method. You should always tell exactly what triggers the throwing of an exception, as the penalty for an exception is program death.

5.3 Triggering Javadoc

First we give instructions for DrJava. Bring up the Preferences by hitting control-; or by selecting the Preferences item from the bottom of the Edit menu. Under Web browser put the path to your web browser. An example of a valid path is

```
/usr/lib/firefox/firefox.sh
```

If you use Windoze, your path should begin with `\tt C:\`. If you use a Mac, it will be in your Applications folder. You can browse for it by hitting the `...` button just to the right of the Web Browser text field.

The javadoc will be saved in a directory called `doc` that is created in same directory as your class's code. Allow the javadoc to be saved in that folder, or files will "spray" all over your directory and make a big mess.

You can also javadoc at the command line with

```
unix> javadoc -d someDirectory BigFraction.java
```

The javadoc output will be placed in the directory `someDirectory` that you specify. Make sure you use the `-d` option to avoid spraying. To see your objet d'art, select File Open... in your browser and then navigate to the file `index.html` in your `doc` directory and open it.

Note that yoiur program *must* compile before any javadoc will be generated.

I don't see my javadoc! Make sure you are using the javadoc comment tokens like so.

```
/**
*    stuff
*/
```

and not regular multiline comment token that look like this.

```
/*
*    stuff
*/
```

5.4 Documenting `toString()` and `equals()`

You will see a new markup device `@return` and `overrides` which tells you what these methods override. You will notice if you look in the javadoc you generated, that an `overrides` tag is already in the method detail.

```
    /**
    * @return a string representing this BigFraction of  the form
    * numerator/denominator.
    */
   @Override
```

```
    public String toString()
    {
        return "" + num + "/" + denom;
    }
```

Note the use of the `@Override` construct just after our javadoc markup. This is called an *annotation*, and the compiler checks that you have used the right signature to actual override the method. If you don't it will be flagged as a compiler error. Always use this annotation if you are implementing the methods `public boolean equals(Object o)` or `public String toString()`.

Now we deal similarly with the `equals` method.

```
     /**
     * @param o an Object we are comparing this BigFraction to
     * @return true iff this BigFraction and that are equal numerically.
     * A value of <tt>false</tt> will be returned if the Object o is not
     * a BigFraction.
     */
    @Override
    public boolean equals(Object o)
    {
        if(! (o instanceof BigFraction))
            return false;
        BigFraction that = (BigFraction) o;
        return num.equals(that.num) && denom.equals(that.denom);
    }
```

5.5 Putting in a Preamble and Documenting the Static Constants

We show where to preamble goes, after the imports and before the head for the class. Place a succinct description of your class here to let your clients know what it does.

```
import java.math.BigInteger
/**
 * This is a class of immutable arbitrary-precision
 * rational numbers.  BigFraction provides
 * extended-precision fractional arithmetic
 * operations, including + with the <code>add</code> method,
 * - with the <code>subtract</code>
 * method, * with the <code>multiply</code> method,
 * and / with the <code>divide</code> method.
 * It computes integer powers
 * of fractions using the <code>pow</code> method.
 */
public class BigFraction
{
    //code
}
```

Documenting the static constants is very straightforward.

```
    /**
     * This is the BigFraction constant 0, which is 0/1.
     */
    public static final BigFraction ZERO;
    /**
     * This is the BigFraction constant 1, which is 1/1.
     */
    public static final BigFraction ONE;
```

5.6 Documenting Arithmetic

Next we javadoc all of the arithmetic operations we have provided the client. Notice how we add an exception if the client attempts to divide by zero.

```
    /**
     * This add BigFractions.
     * @param that a BigFraction we are adding to this BigFraction
     * @return <code>this</code> + <code>that</code>
     */
    public BigFraction add(BigFraction that)
    {
        BigInteger term1 = num.multiply(that.denom);
        BigInteger term2 = denom.multiply(that.num);
        BigInteger bottom = denom.multiply(that.denom);
        return new BigFraction(term1.add(term2), bottom);
    }
    /**
     * This subtracts BigFractions.
     * @param that a BigFraction we are adding to this BigFraction
     * @return <code>this</code> - <code>that</code>
     */
    public BigFraction subtract(BigFraction that)
    {
        BigInteger term1 = num.multiply(that.denom);
        BigInteger term2 = denom.multiply(that.num);
        BigInteger bottom = denom.multiply(that.denom);
        return new BigFraction(term1.subtract(term2), bottom);
    }
    /**
     * This multiplies BigFractions.
     * @param that a BigFraction we are adding to this BigFraction
     * @return <code>this</code> * <code>that</code>
     */
    public BigFraction multiply(BigFraction that)
    {
        BigInteger top = num.multiply(that.num);
        BigInteger bottom = denom.multiply(that.denom);
        return new BigFraction(top, bottom);
    }
    /**
     * This divides BigFractions.
     * @param that a BigFraction we are adding to this BigFraction
     * @return <code>this</code>/<code>that</code>
     * @throws <code>IllegalArgumentException</code> if division by
     * 0 is attempted.
     */
    public BigFraction divide(BigFraction that)
    {
        if(that.equals(BigFraction.ZERO))
            throw new IllegalArgumentException();
        BigInteger top = num.multiply(that.denom);
        BigInteger bottom = denom.multiply(that.num);
        return new BigFraction(top, bottom);
    }
     /**
     * This computes an integer power of BigFraction.
     * @param n an integer power
     * @return <code>this</code><sup><code>n</code></sup>
     */
    public BigFraction pow(int n)
    {
        if(n > 0)
            return new BigFraction(num.pow(n), denom.pow(n));
        if(n == 0)
            return new BigFraction(1,1);
        else
        {
            n = -n;   //strip sign
            return new BigFraction(denom.pow(n), num.pow(n));
        }
    }
```

Finally, we will take care of our two `valueOf` methods.

```
     /**
     * @param n a long we wish to promote to a BigFraction.
     * @return A BigFraction object wrapping n
     */
    public static BigFraction valueOf(long n)
    {
        return new BigFraction(n, 1);
    }
    /**
    * @param num a BigInteger we wish to promote to a BigFraction.
    * @return A BigFraction object wrapping num
    */
    public static BigFraction valueOf(BigInteger num)
    {
        return new BigFraction(num, BigInteger.ONE);
    }
```

5.7 The Complete Code

Here it is! We have dropped in javadoc for our stateic factory method as well.

```
import java.math.BigInteger;
/**
 * This is a class of immutable arbitrary-precision
 * rational numbers.  BigFraction provides
 * extended-precision fractional arithmetic
 * operations, including + with the <code>add</code> method,
 * - with the <code>subtract</code>
 * method, * with the <code>multiply</code> method,
 * and / with the <code>divide</code> method.
 * It computes integer powers
 * of fractions using the <code>pow</code> method.
 */
public class BigFraction
{
    /**
     * This is the BigFraction constant 0, which is 0/1.
     */
    public static final BigFraction ZERO;
    /**
     * This is the BigFraction constant 1, which is 1/1.
     */
    public static final BigFraction ONE;
```

```
static
{
    ZERO = new BigFraction();
    ONE = new BigFraction(1,1);
}
private final BigInteger num;
private final BigInteger denom;
/**
 * This constructor stores a <code>BigFraction</code> in
 * reduced form, with any negative factor appearing in
 * the numerator.
 * @param num the numerator of the <code>BigFraction</code>
 * @param denom the denominator of the <code>BigFraction</code>
 * @throws <code>IllegalArgumentException</code> if the creation
 * of a zero-denominator <code>BigFraction</code> is attempted.
 */
public BigFraction(BigInteger num, BigInteger denom)
{
    if(denom.equals(BigInteger.ZERO))
        throw new IllegalArgumentException();

    BigInteger d = num.gcd(denom);
    if(denom.compareTo(BigInteger.ZERO) < 0)
    {
        num = num.negate();
        denom = denom.negate();
    }
    num = num.divide(d);
    denom = denom.divide(d);
}
/**
 * This default constructor produces BigFraction 0/1.
 */
public BigFraction()
{
    this(BigInteger.ZERO,BigInteger.ONE);
}
/**
 * @return a string representing this BigFraction of  the form
 * numerator/denominator.
 */
@Override
public String toString()
{
    return String.format("%s/%s", num, denom);
```

```
}
/**
 * @param o an Object we are comparing this BigFraction to
 * @return true iff this BigFraction and that are equal numerically.
 * A value of <code>false</code> will be returned if the Object o is not
 * a BigFraction.
 */
@Override
public boolean equals(Object o)
{
    if(! (o instanceof BigFraction))
        return false;
    BigFraction that = (BigFraction) o;
    return num.equals(that.num) && denom.equals(that.denom);
}
/**
 * This static factory produces num/denom as a BigFraction.
 * @param num the numerator for this BigFraction
 * @param denom the denominator for this BigFraction
 * @return A <code>BigFraction</code> representing num/denom.
 */
public static BigFraction valueOf(long num, long denom)
{
    return new BigFraction(BigInteger.valueOf(num),
        BigInteger.valueOf(denom));
}
/**
 * This add BigFractions.
 * @param that a BigFraction we are adding to this BigFraction
 * @return <code>this</code> + <code>that</code>
 */
public BigFraction add(BigFraction that)
{
    BigInteger term1 = num.multiply(that.denom);
    BigInteger term2 = denom.multiply(that.num);
    BigInteger bottom = denom.multiply(that.denom);
    return new BigFraction(term1.add(term2), bottom);
}
/**
 * This subtracts BigFractions.
 * @param that a BigFraction we are adding to this BigFraction
 * @return <code>this</code> - <code>that</code>
 */
public BigFraction subtract(BigFraction that)
{
    BigInteger term1 = num.multiply(that.denom);
```

```
        BigInteger term2 = denom.multiply(that.num);
        BigInteger bottom = denom.multiply(that.denom);
        return new BigFraction(term1.subtract(term2), bottom);
    }
    /**
     * This multiplies BigFractions.
     * @param that a BigFraction we are adding to this BigFraction
     * @return <code>this</code> * <code>that</code>
     */
    public BigFraction multiply(BigFraction that)
    {
        BigInteger top = num.multiply(that.num);
        BigInteger bottom = denom.multiply(that.denom);
        return new BigFraction(top, bottom);
    }
    /**
     * This divides BigFractions.
     * @param that a BigFraction we are adding to this BigFraction
     * @return <code>this</code>/<code>that</code>
     * @throws <code>IllegalArgumentException</code> if division by
     * 0 is attempted.
     */
    public BigFraction divide(BigFraction that)
    {
        if(that.equals(BigFraction.ZERO))
            throw new IllegalArgumentException();
        BigInteger top = num.multiply(that.denom);
        BigInteger bottom = denom.multiply(that.num);
        return new BigFraction(top, bottom);
    }
    /**
     * @param n a long we wish to promote to a BigFraction.
     * @return A BigFraction object wrapping n
     */
    public static BigFraction valueOf(long n)
    {
        return new BigFraction(n, 1);
    }
     /**
     * @param num a BigInteger we wish to promote to a BigFraction.
     * @return A BigFraction object wrapping num
     */
    public static BigFraction valueOf(BigInteger num)
    {
        return new BigFraction(num, BigInteger.ONE);
    }
```

```
}
```

Programming Exercises

1. Add javadoc for all of the methods you wrote in the previous set of programming exercises.
2. Write a second class called `TestBigFraction`. Place a `main` method in this class and have it test `BigFraction` and its methods. Place the classes in the same directory.