Contents

1 Case Study: An Extended-Precision Fraction Class **1**

1.1 A Brief Weltanschauung 2

2 Start your Engines! **3**

3 Making a Proper Constructor and `toString()` Method **3**

4 A Static Factory Method **7**

5 Creating an `equals` Method **9**

6 Hello Mrs. Wormwood! Adding Arithmetic **10**

7 The Role of `static` and `final` **13**

8 Using Javadoc **17**

8.1 Triggering Javadoc 18

8.2 Documenting `toString()` and `equals()` 19

8.3 Putting in a Preamble and Documenting the Static Constants . . 20

8.4 Documenting Arithmetic 21

8.5 The Complete Code 23

1 Case Study: An Extended-Precision Fraction Class

We have achieved several goals in the last chapter, the most important of which are understanding what makes up a Java class and understanding the core Java language so as to be Turing-complete.

To tie everything together, we will do a case study of creating a class called `BigFraction`, which will work like the `BigInteger` class and provide many of the same operations, except it will do exact fractional arithmetic. This class will have a professional appearance, and an interface similar to that of `BigInteger`.

During this chapter, you will learn about `javadoc`; this allows you to create an API page for your class that will have the same appearance as the page you see on the web. When you are done with this chapter, you will have a class

suitable for others to use as clients who wish to perform extended-precision rational arithmetic. The `javadoc` feature can be invoked at the UNIX command line. It can also be created for you by DrJava.

1.1 A Brief Weltanschauung

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 creates a computational environment 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` class will create an environment for computing in $\mathbb{Q}$ similar to that which `BigInteger` provides for $\mathbb{Z}$.

Not all real numbers (which we represent with `double`) are rational. It is a well-known fact that $\sqrt{2}$ and the beloved constant π are not rational. In fact, most of the time you take a square root, you will despoil the rationality of any rational number you operate on. The only exception occurs when a fraction, in lowest terms, has a perfect square in the nominator and denominator.

This explains why `BigInteger`'s `pow` method accepts only integers. Moreover it accepts only positive integers or 0 because a negative power of an integer is not an integer, unless the integer happens to be ± 1. As a result, Java becomes irate and produces an abrasive run-time error if you attempt to compute a negative power for an `BigInteger`. You will see that when we create a `pow` method for our `BigFractions` it will only accept integers (any integer in fact), but not any other kind of rational number.

We select this case study because it brings to the for a variety of important design questions. When we are done, we will have a nice facility for computing with fractions. You will get to see the development of a moderately sophisticated class from scratch.

2 Start your Engines!

Let us begin by thinking about fractions. We know a fraction has two important items reflecting its state: its numerator and its denominator. Fractions have some slippery properties. For example, we know that

$$\frac{1}{4} = \frac{256}{1024}.$$

The representation of a fraction in terms of numerator and denominator is not unique.

An interesting collection of numbers is the *harmonic numbers*; they are defined by

$$H_n = \sum_{k=1}^{n} \frac{1}{k}, \qquad n \in \mathbb{N}.$$

Let us show the first few harmonic numbers. It is easy to see that $H_1 = 1$. We have

$$H_2 = 1 + \frac{1}{2} = \frac{3}{2}.$$

Next,

$$H_3 = H_2 + \frac{1}{3} = \frac{3}{2} + \frac{1}{3} = \frac{11}{6}.$$

Now let's skip down to H_{10}.

$$H_{10} = \frac{7381}{2520}.$$

One thing is clear: as we keep adding fractions, their numerators and denominators have a propensity to keep getting larger. We know that the primitive `int` and `long` types are not going to cut the mustard here because they will overflow and produce false results.

We will therefore use `BigInteger` for the numerator and denominator of our `BigFraction`s. We should be able to compute H_{100} or even H_{1000}.

3 Making a Proper Constructor and `toString()` Method

When starting out to build a class, we begin by creating a suitable constructor. Along the way, you will need a `toString()` method so you can see what you are doing.

We begin with this crude attempt. We are mimicking our work on the `Point` class we developed earlier.

```
import java.math.BigInteger;
public class BigFraction
{
    private BigInteger num;
    private BigInteger denom;
    public BigFraction(BigInteger num, BigInteger denom)
    {
        this.num = num;
        this.denom = denom;
    }
}
```

It is easy to see that there will be problems. Suppose a client programmer writes this code.

```
BigInteger a = BigInteger.valueOf(256);
BigInteger b = BigInteger.valueOf(1024);
BigFraction f = new BigFraction(a, b);
```

It seems ridiculous that this fraction should be stored as 256/1024 when it is in fact 1/4. Hence, it seems we should keep our fractions reduced.

To reduce a fraction, you compute the greatest common divisor of the numerator and denominator and divide it out of both. Notice that the `BigInteger` class computes GCDs for you, so we can alter our constructor as follows.

```
    public BigFraction(BigInteger num, BigInteger denom)
    {
        this.num = num;
        this.denom = denom;
        BigInteger d = num.gcd(denom);
        num = num.divide(d);
        denom = denom.divide(d);
    }
```

Let us now see what this looks like.

```
jshell> import java.math.BigInteger;
jshell> BigInteger a = BigInteger.valueOf(256);
a ==> 256
jshell> BigInteger b = BigInteger.valueOf(1024);
a ==> 1024
jshell> BigFraction f = new BigFraction(a,b)
f ==> BigFraction@6ad20835
jshell> f
```

```
BigFraction@6ad20835
jshell>
```

Oops. The built-in `toString()` method is not doing such a great job. Let's override it so it make fractions that look like this: 45/17. Here is our revised class.

```
import java.math.BigInteger;
public class BigFraction
{
    private BigInteger num;
    private BigInteger denom;
    public BigFraction(BigInteger num, BigInteger denom)
    {
        this.num = num;
        this.denom = denom;
        BigInteger d = num.gcd(denom);
        num = num.divide(d);
        denom = denom.divide(d);
    }

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

Now we try our unreduced fraction and find things in a happy state.

```
> import java.math.BigInteger;
jshell> import java.math.BigInteger;
jshell> BigInteger a = BigInteger.valueOf(256);
a ==> 256
jshell> BigInteger b = BigInteger.valueOf(1024);
a ==> 1024
jshell> BigFraction f = new BigFraction(a,b)
f ==> 1/4
jshell> f
1/4
```

There is yet one more thing to do to button this up. This little session should prove convincing.

```
jshell> b = new BigInteger.valueOf(-1024);
jshell> BigFraction f = new BigFraction(a,b);
```

```
f ==> 1/-4
jshell> f
1/-4
```

If we put the negative on the top, the `toString()` method will work nicely. We also get the benefit that we can check fraction equality by just checking for equality of numerator and denominator.

All we need do is to add something like this to the constructor.

```
if(denom < 0)
{
    denom = -denom;
}
```

However, we are indulging here in illegal operations on `BigInteger`s. Looking on the API page, we can see that there is a `negate()` method that returns a copy of the `BigInteger` with its sign changed. Also, there is a `compareTo` method. The expression

```
foo.compareTo(goo)
```

returns a negative integer if `foo < goo`, a positive integer if `foo > goo` and 0 if `foo == goo`. We integrate these features into our class and we now have

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

```
    }
}
```

4 A Static Factory Method

Wouldn't it be nice to be able to make a `BigFraction` with ordinary integers? In fact, it would be a smart play to use the `long` type, since a `long` type argument will happily accept an `int`, `short`, or `byte`. We will use `this` to call the main constructor, so we do not have to repeat all of the heavy lifting it does.

To this end, we avail ourselves of the `valueOf` method for `BigInteger` to make a `valueOf` method for `BigFraction`.

```
    public static BigFraction valueOf(long num, long denom)
    {
        return new BigFraction(BigInteger.valueOf(num), BigInteger.valueOf(denom));
    }
```

While we are here, let's make an (obvious) default constructor.

```
    public BigFraction()
    {
        this(BigInteger.ZERO,BigInteger.ONE);
    }
```

Finally we shall send an ugly message to the woebegone client programmer who tries to create a `BigFraction` with a zero denominator. Insert this line in the main constructor, just after `num` and `denom` are initialized.

```
    if(denom.equals(BigInteger.ZERO))
    {
        throw new IllegalArgumentException();
    }
```

This will bring immediate program death to the miscreant client programmer who calls it.

Here is our class with everything added to it.

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

    private BigInteger num;
```

```
    private BigInteger denom;
    public BigFraction(BigInteger num, BigInteger denom)
    {
        if(denom.equals(BigInteger.ZERO))
            throw new IllegalArgumentException();

        BigInteger d = num.gcd(denom);
        num = num.divide(d);
        denom = denom.divide(d);
        if(denom.compareTo(BigInteger.ZERO) < 0)
        {
            num = num.negate();
            denom = denom.negate();
        }
        this.num = num;
        this.denom = denom;
    }
    public BigFraction(long num, long denom)
    {
        this(BigInteger.valueOf(num), BigInteger.valueOf(denom));
    }
    public BigFraction()
    {
        this(BigInteger.ZERO,BigInteger.ONE);
    }
    public String toString()
    {
        return String.format("%s/%s", num, denom);
    }
}
```

Finally, let's take this all for a test-drive. First we look at our main "workhorse" constructor.

```
jshell> import java.math.BigInteger;
jshell> BigInteger a = BigInteger.valueOf(1048576);
a ==> 1048576
jshell> BigInteger b = BigInteger.valueOf(7776);
b ==> 7776
jshell> BigFraction f = new BigFraction(a,b)
f ==> 32768/243
jshell> f
32768/243
```

Our static factory method makes this process less verbose.

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

Here we see our default constructor.

```
jshell> BigFraction z = new BigFraction()
z ==> 0/1
jshell> z
0/1
```

Finally we tempt and see death.

```
jshell> BigFraction rotten = BigFraction.valueOf(5,0);
|  Exception java.lang.IllegalArgumentException
|        at BigFraction.<init> (#2:36)
|        at BigFraction.valueOf (#2:69)
|        at (#3:1)
```

This exception object will immediately halt any program that is running and that calls the static factory illegally; notice how it shows the path the exception takes. This will do a nice job of flagging the error for the malefactor who perpetrates it.

5 Creating an `equals` Method

This process is always the same. First do the species test. Then cast the `Object` in the argument list to a `BigFraction`. Once this is done, creating `equals` is easy, since all we need to is to compare equality of numerator and denominator.

```
public boolean equals(Object o)
{
    if(! (o instanceof BigFraction))
        return false;
    BigFraction that = (BigFraction) o;
    return num.equals(that.num) && denom.equals(that.denom);
}
```

Note that since we are comparing `BigIntegers` in the `return` statement, we must use the `equals` method for `BigInteger`.

Now lets take this for a walk. We begin by making some instances.

```
jshell> BigFraction f = new BigFraction(1,3);
f ==> 1/3
jshell> BigFraction g = new BigFraction(1,2);
g ==> 1/2
jshell> BigFraction h = new BigFraction(2,4);
h ==> 1/2
jshell> f
1/3
jshell> g
1/2
jshell> h
1/2
>
```

Notice that none are equal under `==`.

```
jshell> f == g
false
jshell> f == h
false
jshell> g == h
false
>
```

Next, we trot out our shiny new `equals` method.

```
jshell> f.equals(g)
false
jshell> f.equals(h)
false
jshell> g.equals(h)
true
```

Finally, we violate the species test and watch a `false` come right back at us as it should.

```
> f.equals("platypus")
false
>
```

6 Hello Mrs. Wormwood! Adding Arithmetic

To as great an extent as possible, we shall imitate the interface that is presented to us by the `BigInteger` class. We need to define four methods: `add`, `subtract`,

`multiply`, and `divide`. Each of these methods will take a `BigFraction` as an argument, and will return a `BigFraction`. We begin with addition.

We learned from Mrs. Wormwood that

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

The header for our `add` method will be

```
public BigFraction add(BigFraction that)
```

Remember, since we are programming in `BigFraction`, we have a `num` and a `denom` and we are

$$\frac{\texttt{num}}{\texttt{denom}}.$$

We are going to add ourselves to `that`. Since `that` is a `BigFraction` has a `num` and a `denom`, too. These are known as `that.num` and `that.denom`.

So, we will wind up doing this little arithmetic arabesque to provide us with a framework for writing the actual code.

$$\frac{\texttt{num}}{\texttt{denom}} + \frac{\texttt{that.num}}{\texttt{that.denom}} = \frac{\texttt{num*that.denom + denom*that.num}}{\texttt{denom*that.denom}}$$

Let's take this a piece at a time, beginning with the first term in the numerator of the sum. We are not allowed to write

```
num*that.denom
```

We have to translate it into the language of `BigInteger`, which says we do the following; we elect to store the result in the `BigInteger term1`.

```
BigInteger term1 = num.multiply(that.denom);
```

Now do the same thing with the second term.

```
BigInteger term2 = denom.multiply(that.num);
```

As a result, the numerator will be

```
term1.add(term2)
```

Now we deal with the denominator

```
BigInteger bottom = denom.multiply(that.denom);
```

Our entire fraction in these terms is

$$\frac{\texttt{term1} + \texttt{term2}}{\texttt{bottom}}.$$

So our `return` statement reads

```
return new BigFraction(term1.add(term2), bottom);
```

Assembling it all we have the completed `add` method.

```
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);
}
```

Let's now do a little test.

```
> BigFraction f = new BigFraction(1,2)
> BigFraction g = new BigFraction(1,3)
> f.add(g)
5/6
>
```

Subtraction is easy, we just change an `add` into a `subtract`.

```
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.subtract(term2), bottom);
}
```

Multiplication is done "straight across."

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

To divide, invert and multiply.

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

7 The Role of `static` and `final`

In keeping with the behavior of `BigInteger` we will make our `BigFraction`s immutable. You will notice that none of our methods we have created so far allow changes in state.

To make this intent clear, we should mark the `num` and `denom` state variables `final`. Since `BigInteger`s are immutable, this renders the state variables constant. Our class creates immutable objects.

The `BigInteger` class has static constants `ONE` and `ZERO`. We add constants like this to our class as follows. First we create the static objects `ONE` and `ZERO`. We shall make them `public`.

```
public static final BigFraction ZERO;
public static final BigFraction ONE;
```

Place these before the declarations for the state variables in the class. Note: these are *not* state variables, since they reflect a property of the class as a whole, not the state of any particular object. To initialize them, create a `static` block. You do so as follows.

```
static
{
    ZERO = new BigFraction();
    ONE = new BigFraction(1,1);
}
```

Clients of your class can now use `BigFraction.ZERO` to get 0 as a `BigFraction` and `BigFraction.ONE` to get 1 as a `BigFraction`.

If you compile now, you will get errors because the constructor performs some reassignments. We can reëngineer it as follows to get rid of the reassignments.

```
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);
    }
```

Now you should be glad you used `this` in the sibling constructors. No modification of these is necessary.

You will notice that `BigInteger` has a static `valueOf` method that converts `long`s to `BigIntegers`. We now make two static factory methods named `valueOf`. One will take a `long` and promote it to a `BigFraction`. The other will do this service for `BigInteger`.

```
    public static BigFraction valueOf(long n)
    {
        return new BigFraction(n, 1);
    }
    public static BigFraction valueOf(BigInteger num)
    {
        return new BigFraction(num, BigInteger.ONE);
    }
```

Here is the current appearance of the entire class.

```
import java.math.BigInteger;
public class BigFraction
{
    public static final BigFraction ZERO;
    public static final BigFraction ONE;
    static
    {
        ZERO = new BigFraction();
        ONE = new BigFraction(1,1);
    }
    private final BigInteger num;
    private final BigInteger denom;
    public BigFraction(BigInteger num, BigInteger denom)
    {
        this.num = num;
```

```
        this.denom = denom;
        if(denom.equals(BigInteger.ZERO))
            throw new IllegalArgumentException();

        BigInteger d = num.gcd(denom);
        num = num.divide(d);
        denom = denom.divide(d);
        if(denom.compareTo(BigInteger.ZERO) < 0)
        {
            num = num.negate();
            denom = denom.negate();
        }
    }
    public static BigFraction valueOf(long num, long denom)
    {
        return new BigFraction(BigInteger.valueOf(num), BigInteger.valueOf(denom));
    }
    public BigFraction()
    {
        this(BigInteger.ZERO,BigInteger.ONE);
    }
    public String toString()
    {
        return String.format("%s/%s", num, denom);
    }
    public boolean equals(Object o)
    {
        if(! (o instanceof BigFraction))
            return false;
        BigFraction that = (BigFraction) o;
        return num.equals(that.num) && denom.equals(that.denom);
    }
    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);
    }
    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);
    }
```

```
    public BigFraction multiply(BigFraction that)
    {
        BigInteger top = num.multiply(that.num);
        BigInteger bottom = denom.multiply(that.denom);
        return new 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);
    }
    public static BigFraction valueOf(long n)
    {
        return new BigFraction(n, 1);
    }
    public static BigFraction valueOf(BigInteger num)
    {
        return new BigFraction(num, BigInteger.ONE);
    }
}
```

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

1. Write a `publlic pow(int n)` method that works for both positive and negative integers
2. Write a method `public BigInteger bigIntValue()` that divides the denominator into the numerator and which truncates towards zero.
3. Write the method `public BigFraction abs()` which returns the absolute value of this `BigFraction`.
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 a method `public BigFraction negate()` which returns a copy of this `BigFraction` with its sign changed.
7. 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.
8. 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`.

9. 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.
10. When should division throw an `IllegalArgumentException`? Add this feature to the class.
11. (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.

8 Using Javadoc

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.

8.1 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
*/
```

8.2 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);
    }
```

8.3 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;
```

8.4 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);
}
```

8.5 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.