5-1 Making a Fractions Class

In his book *Mathematics for the Nonmathematician* (NY: Dover, 1985), pp. 64-65, Morris Kline points out:

> The disappointing feature of the decimal representation of fractions is that some simple fractions cannot be represented as fractions with a finite number of digits. Thus when we seek to express 1/3 as a decimal, we find that neither 0.3, nor 0.33, nor 0.333, and so on, suffices. All one can say in this and similar cases is that by carrying more and more decimal digits, one comes closer and closer to the fraction, but no finite number of digits will ever be the exact answer. This fact is expressed by the notation 1/3 = 0.333..., where the dots indicate that we must keep on adding threes to approach the fraction 1/3 more and more closely.

In most programming languages, including Java, this leads to well-known problems, as the following code fragment illustrates:

```
double f = 1.0/7.0;
double sum = 0.0;
for (int i = 0; i < 7; i++) {
  sum += f;
}
System.out.println (sum);
```

We have attempted to compute the sum of seven sevenths, which should of course be one. But when these statements are executed, the output is 0.9999999999999998.

Close, but not close enough. Although there are ways to deal with this, it often pays to dispense with the decimal representation of fractions altogether, and instead deal with fractions simply as fractions. We are going to be writing a Fraction class that which allows us to do just that.

Describing the Fraction Class

Fields / Instance Variables: Instances of the Fraction class will have the following ints:

- The Numerator
- The Denominator

They will be declared as private to force users of this class to go through our predefined methods that include error checking.

Constructors: These are the special methods that are called when a new object is created. Constructors are always public. Make the following:

- A default constructor that takes no parameters and creates a default fraction
- A constructor that takes two int parameters, the value for the numerator and the value for the

denominator.
- A constructor that takes 1 String parameter that represents a fraction using a forward slash character, such as “1/2” or “7/8”.
- A copy constructor that takes a Fraction object as a parameter and creates a new fraction that has the same numerator and denominator.

Under no circumstance should our constructors allow a fraction to be created that has a zero denominator!

Accessor Methods: These are public methods that return the values of the instance values. This way, we can still peak at the private instance variables, but we cannot directly change them. Include the following accessor methods:

- getNum() and getDenom() - methods that return the values of the numerator and denominator.
- toString() - a method that returns the Fraction as a String. For example, the fraction one half should be represented as “1/2”
- toDouble() - a method that returns the Fraction as a decimal number. For example, the fraction one half should be represented as 0.5

Mutator Methods: These are the public methods that operate on the state of the variables themselves.

- reduce() - a method for reducing the Fraction object to lowest terms. This can be a void method with no parameters.
- setNum() and setDenom() - methods for changing the values of the numerator and denominator

Static Methods: These are methods that do not act on an individual fraction but are tools for using fractions.

- multiply(), divide(), add(), subtract() - method for arithmetic operations on fractions. I suggest to do them in this order. These methods should take two Fraction objects as a parameter, and then return a new Fraction object that applies the corresponding operation to these parameters. When working with the objects in the driver file, this is what using these methods will look like: to add 1/2 + 1/3:

```
Fraction half = new Fraction(1,2);
Fraction third = new Fraction (1,3);
Fraction sum = Fraction.add(half, third);
```

Helper Methods: These are private methods that help the mutators, but do not need to be directly accessed by things outside of the Fraction object.

- A method that returns the GCF (greatest common factor) between two integers. You will also have to make a main method in a driver class that will create Fraction objects and demonstrate

the various methods.

Helpful Hints

Create the methods in the following order. After you write each method, write some code in your Driver class to test it.

1. Constructors, otherwise you can't make objects
2. Accessor methods – they are the easiest to write, and you will need them to get the numerators and denominators of fraction objects when adding, subtracting, etc. Also, they will be helpful for debugging your project.
3. GCF method is next, because you need it to write the reduce method.
4. Reduce method is next, because you need it to properly write the arithmetic operation methods.
5. Multiply is the first arithmetic method because it is the easiest and you can it to write the divide method.
6. Next do divide, adding, and subtracting. Leaving adding and subtracting to the end, because they are the hardest. They are also almost exactly the same.

How to Write the GCF method?

To do find the GCF, we can use Euclid's Algorithm, which is the world's oldest numerical algorithm still used today. Let's take 252 and 105, which have a GCF of 21. Euclid's Algorithm takes advantage of the fact that if you reduce the larger number by subtracting the smaller one from it, the GCF doesn't change. So, 242 – 105 = 147, and the GCF between 147 and 105 is still 21. You keep on repeating this process, subtracting the smaller number from the larger one until the numbers are equal. That number is the GCF!

Let's illustrate this process:

1. 252, 105 (Our two starting numbers)
2. 147, 105 (252 – 105 to get 147)
3. 42, 105 (147 – 105 to get 42)
4. 42, 63 (105 – 42 to get 63)
5. 42, 21 (63 – 42 to get 21)
6. 21, 21 (42 – 21 to get 21)

Our GCF is 21! In Java, you can use a while loop to repeat this process until the two numbers are
e
q
u
a
l
.

L
a
t
e
r