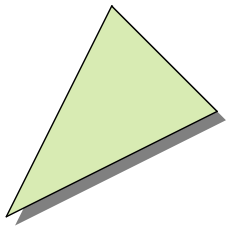

Programming Exercise 12:

More on Constructible Classes

Purpose: Writing a class that interacts with objects of the same class
Background readings from textbook: Liang, Chapter 9.1-9.5, 9.8-9.10, 9.14

Due date for section 001: Monday, April 11 by 10 am
Due date for section 002: Wednesday, April 13 by 10 am

Overview

The String class has a method called equals which receives another String as a parameter to determine if the two Strings share the same value. This is an example of a class which has methods that are passed another instance of the same class. The method has to compare or use its instance data with the instance data of the parameter. In this lab, we learn more about writing classes by dealing with a class that has methods that will receive other instances of the same class as parameters. We will also see the value of having a private method.

Part 1: Public Methods, Private Instance Data and Private Methods

We want to make the instance data of our classes private so that they are not directly alterable from user classes and other programs. We add public mutators and accessors to provide the interface to these internal data. Assume that we want to write a method that receives a parameter of the same class. The method will need to access some of *this* class' instance data and some instance data of the parameter. Since the instance data are private, the method must access the parameter's instance data via accessors. For instance, imagine a class called Foo which has three int instance data, `a`, `b` and `c`. We want to write a method called `isEqualTo` which will compare this Foo to the parameter Foo and return true if both have the same int values for each of `a`, `b` and `c`. The method's outline is:

```
public boolean isEqualTo(Foo other) {
      if(…) return true;
      else return false;
}
```

The if condition needs to test `a`, `b` and `c` of this Foo to `a`, `b` and `c` of `other`. If the three instance data were public, we could do so with the condition `(a==other.a&&b==other.b&&c==other.c)`. But because they are private, we have to do this instead using accessors. We will assume they are called getA, getB and getC. The condition is now `(a==other.getA()&&b==other.getB()&&c==other.getC())`.

Now let's consider whether we want all of our methods to be public. Imagine that Foo has a method called `outdated`. A Foo is considered outdated if the values of `a`, `b` and `c` are such that the equation `3*a – b + 2*c < 0`. As the equation may change from time to time, we decide to divide the decision of outdated into two separate methods. First, outdated is shown below where it called another method called `compute`. The compute method has our most up-to-date equation and returns the int value of it to be compared against 0.

```
public boolean outdated( ) {
      if(compute() < 0) return true;
      else return false;
}
```

```
private int compute( ) {
      return 3 * a – b + 2 * c;
}
```

Since we expect compute to only be called from within the class Foo, we make it private. Thus, if `f` is an object of type `Foo`, we can do `f.outdated()` but not `f.compute()`. Most of the time, our methods will be public, but examples like this show a reason to also have private methods.

Part 2: The Rational Class

In this program, you will write a class to represent rational numbers. A rational number is any number the can be stored in a fraction where both the numerator and denominator are int values. For instance, 1 / 3 is a rational number but 1 / 3.333333333 is not. Your Rational class will have two instance data, `numerator` (which can be positive, 0 or negative) and `denominator`, which *must be* positive. We will have methods that will perform arithmetic operations on two Rational objects similar to Foo's toEqualTo method. For instance, given two Rationals r1 and r2, we could do `r1.add(r2);` to get a new Rational value that sums the two Rationals together. You will define methods of `add`, `subtract`, `multiply`, `divide`, `comparesTo` and a `toString` which will all be public, and a private method called `reduce`. You will also need accessors and mutators for both instance data.

There will be two constructors. The first receives two parameters (the numerator and denominator) and sets the instance data appropriately but if the denominator is negative, make it positive and multiply the numerator by -1, and if the denominator is 0, reset the number to be 0 / 1. For instance, if the two numbers are 2 / -3 then this becomes -2 / 3 and if the two numbers are 2 / 0 then this becomes 0 / 1. Use proper logic in the constructor. The second constructor receives no parameters and sets the Rational to 0 / 1.

The accessors simply return the appropriate int value. The mutator for the denominator must have proper logic to change a negative to a positive (by multiplying the numerator by -1) and if 0 change the Rational to 0 / 1. The toString should return the number written like this: `numerator / denominator` such as -5 / 3. The comparesTo method will return -1 if this Rational < the parameter, 1 if this Rational > the parameter, and 0 if they are the same value. To perform this comparison, multiply this Rational's numerator by the other's denominator and the other's numerator by this Rational's denominator and compare the two products.

To add two Rationals, you will multiply both numerators by the other Rational's denominator. You will then add the two adjusted numerators together for the new numerator and the denominator is the product of both denominators. The method will return a new Rational. Here is what it would look like assuming methods getNumerator and getDenominator.

```
public Rational add(Rational other) {
      int num1 = numerator;
      int num2 = other.getNumerator();
      int den1 = denominator;
      int den2 = other.getDenominator;
      int num3 = num1 * den2 + num2 * den1;
      int den3 = den1 * den2;
      Rational r1 = new Rational(num3, den3);
      return r1;
}
```

The subtract method is identical except that num3 will subtract the second part from the first. To multiply, the numerator is the two numerators multiplied together and the denominator is the two denominators multiplied together. For division, it is num1 * den2 / num2 * den1. Remember that each the constructor will test and

adjust values appropriately if the denominator is 0 or negative so you don't have to worry about that in any of these four methods.

The last method to write is reduce. This receives no parameters and adjusts *this* Rational by reducing its numerator and denominator as much as possible. This will be a private method and called by the 2-parameter constructor and the two mutators. You do not need to call it in add for instance because add creates a new Rational and in doing so, that Rational's constructor calls reduce. Let's imagine that the Rational has a numerator of 120 and a denominator of 100. This would reduce to 6 / 5. How do we compute this reduction? The easiest way is to start a divisor at 2 and see if we can divide both numerator and denominator by our divisor. If so, we repeat, otherwise we add 1 to the divisor. We continue to do this until the divisor is greater than either the numerator or denominator. Here is the pseudocode:

```
divisor = 2
while the divisor is less than the absolute value of
      the numerator and less than the denominator
      if numerator % divisor == 0 and denominator % divisor == 0
            reset numerator = numerator / divisor
            reset denominator = denominator / divisor
      else divisor++
```

Part 3: Finishing Up and Submitting Your Assignment

Write a User class that will create and use some Rationals. The User class will include a main method so it will be static. It does not need any other methods. Use the following code for main's body.

```
Rational r1 = new Rational(1, 6);
Rational r2 = new Rational(-5, -7);       // this will become 5/7
Rational r3 = new Rational(4, 2);         // this will become 2/1
Rational r4 = new Rational(3, -2);        // this will become -3/2
Rational r5 = r1.plus(r3);
Rational r6 = r5.times(r4);
Rational r7 = r2.minus(r2);
r1.setDenominator(0);                     // this will become 0/1
r2.setNumerator(42);                      // r2 will become 6/1
r3 = r4.subtract(r3);
System.out.println(r4.comparesTo(r5));
System.out.println(r1.comparesTo(new Rational(10, 60));
// print out all 7 Rationals
```

Check to see if you got the correct results and comment both classes (although the User class can be minimally commented). Submit (by hardcopy or email) both classes (Rational, User) and your output from the User class.