Rationals Lab: the next generation

T. Reinhardt

1 Description

By now, you should have completed the original Rationals Lab. In that project, you implemented a class that might be used as a rational number, i.e., a number that can be represented as two integers, a/b, where $b \neq 0$. You also implemented several common arithmetic operations for your class: addition, multiplication, and division (by obtaining the inverse).

For this lab, you will refine your implementation of rational numbers by adding some important functionalities:

- A *copy constructor* which is a special constructor that Java will use to make *copies* of your object;
- A `reduce()` method that reduces a `Rational` to its primitive form, and
- An `equals` method that returns `true` only when two `Rational` numbers have the same *reduced* form.

2 The details . . .

2.1 Copy constructors

Let's begin with the copy constructor. When Java sees certain "patterns" such as

```
Rational rat1 = new Rational( 2,3 );
Rational rat2 = new Rational( rat1 );
```

it interprets that second usage as saying:

> Take the incoming argument which is an object of the same class as me, and create a *copy* of that argument by reading its properties's values and setting up a new object to have those very same values.

To see what this looks like with a coding example, consider the `MovingObject` class that you have already seen:

```
public class MovingObject {
   private double x_pos;
   private double y_pos;
   private double mass;

   public MovingObject( double x_pos, double y_pos, double mass ) {
       ... }

  /**
    * The copy constructor for Moving Object.
    */
   public MovingObject( MovingObject otherObject ) {
        this.x_pos = otherObject.x_pos;
        this.y_pos = otherObject.y_pos;
        this.mass = otherObject.mass;
   }

...
}
```

Things to keep in mind:

1. Only one copy constructor is (generally) available on a class definition (why?).
2. Because the copy constructor is defined within the class, it has access to the `private` data of `this` as well as the incoming object (why?).
3. The definition and use of a copy constructor is *optional*, but it's *recommended* if we are interested in designing *immutable* classes.

2.2 Reducing Rational Numbers

You recall from elementary school arithmetic that 1/2 is the same as 2/4 which is the same as 5/10, and so on .... Any rational number can be "reduced" in the sense that the greatest common divisor between its numerator and its denominator is 1. Again, the greatest common divisor between 1 and 2 is 1, whereas the greatest common divisor between 2 and 4 is 2, and we could extend this pattern indefinitely.

In a very meaningful sense, all rational numbers have a "most reduced form." For example, 1/2 is the most reduced form of an infinite number of variations that we could create by multiplying the numerator and the denominator by a fixed integer. One way to compare two rational numbers for equality is to see if one reduces to the other. Of course, we have other ways of determining if two rationals are equal as well: such as dividing one

by the other and ensuring that we get back a rational number whose numerator equals its denominator (ask: why does this work?).

As part of your lab, we require that you implement the `reduce()` method:

```
public Rational reduce() {
    // return a new rational number that is result of reducing
    // this rational number
}
```

2.2.1 How do we reduce rational numbers?

We need to be able to find the *greatest common divisor* between any two (positive) integers (in our case). For example, between 4 and 2, we see that 2 is the greatest common divisor. But, between 4 and 3, we have only 1; and sure enough, either rational number, 3/4 or 4/3 is in its *most reduced form*. You will need to write a `private` method to do this:

```
private static int gcd( int a, int b)  {
....
}
```

Your Teaching Assistants will help you, but here's one interpretation of a famous algorithm for computing greatest common divisors:

Algorithm 1 Euclid's GCD algorithm

```
function GCD(a, b)                          ▷ Two ints, a and b, are input.
    r ← a%b                                 ▷ Recall that this is Java's remainder operator here.
    while r ≠ 0 do                          ▷ Done when r (the remainder of a and b) is 0.
        a ← b
        b ← r
        r ← a%b
    end while
    return b                                ▷ The GCD is b.
end function
```

Your mission: translate this into working Java code, and write some tests to ensure it works. Once you have this algorithm working, reducing any rational number requires that you iteratively use it with the numerator and denominator until you obtain a gcd of 1. This will give you another numerator denominator pair that you might want to store on the `Rational` object while keeping the original numbers provided by the caller of the constructor also on the object. Why? Think about the `toString()` method.

2.3 Implementing the equals method for Rationals

This is a good time to review what's needed to complete the `equals` method:

```
public boolean equals( Object other ) { ... }
```

Recall: under no circumstances should this method throw an exception! If you don't know the expected answers to each of these items, talk with your partner or with a TA.

1. Test that `this` is actually `other`. (Cheap shot, but if it works . . .)
2. Test that `other` is `null`.
3. Test that `other` is an instance of the `Rational` class.
4. Depending upon that last result, divide one by the other and test that the numerator equals the denominator. If it does, announce success; fail otherwise.[1]

2.4 Other tasks

2.4.1 Design issues

Consider the following *optional redesign possibilities* of the `Rational` class:

- Modify the original constructor for the `Rational` class to compute the "reduced form" and store this in two new instance variables. (This is *in addition to what it does now.*)
- Implement a copy-constructor for the `Rational` class.
- Simplify the `equal` logic to just compare the "reduced forms" of `this` and the other `Rational` number.

2.4.2 Testing & Sumission issues

Important: if you wish to re-use code from the original Rationals Lab, copy the method definitions *from* the original *into* this lab. **Do not try to re-submit the original Rationals Lab for this lab because the Submit Server will disallow it.**

You should write your own Unit Tests in the `StudentTests` file, which should be provided with your project. I strongly encourage you to test the `equals` method under a variety of circumstances: make sure it works when it should, and fails when it should.[2]

We've simplified this by not requiring you to mess with negative rationals . . . have fun!

[1] If you have reduced representations of rational numbers what else might you try here?

[2] Submitting your Lab to the Submit Server should also run your Student Tests along with the Public Tests.