COSC 102 Spring 2015

Rational Class Lab 3

In this lab you will work on the design, implementation and test of a class that creates `Rational` objects.

Introduction

Before you build a class, you should determine what its responsibilities are. Responsibilities express the duties of an object in its interactions with other objects. Some typical kinds of responsibilities are knowing, computing, controlling, and interacting with a user. For example, consider a class representing a bank account. It would have a responsibility to know the balance of the account. An accessor method that returns the value of a private variable holding the balance can fulfill that responsibility. Another responsibility of the bank account class might be to compute the monthly interest. That responsibility can be fulfilled by a mutator method that first computes the interest and then modifies the balance. But who decides when the interest should be computed? This is an example of a controller responsibility and most likely would not be the responsibility of the bank account class. Suppose you wished to withdraw a hundred dollars from your account. What class would be responsible for doing the input and output? Again, it is probably not the bank account class. The interface could be an automated teller, a Java program running on the web, or even just a plain terminal. If the bank account class is responsible for the interaction, it will be susceptible to frequent changes as different technologies are developed and used to allow a customer to interact with his or her account. To protect the bank account from those kinds of changes, the interaction responsibility will be assigned to other classes that have interaction as their primary responsibility. The classes that you will develop for now are intended to be very general and therefore will rarely have interaction responsibilities. Deciding which responsibilities a class should have is a design issue that is the province of a course on object-oriented programming. To do it well takes practice: we need some time.

Once the responsibilities of a class have been determined, an implementation is designed to meet those responsibilities. The implementation will consist of two pieces:

- the variables whose values comprise the state of the object, and
- the methods that comprise the protocol for the class.

But when is the implementation correct? The answer to this question is addressed in two ways.

The first approach is the use of invariants. As a class is being designed, look for constraints (invariants) on the state of the object that should always be true. For example, consider the bank account class. One invariant might be that the interest rate should always be greater than or equal to zero. Another invariant could be that the balance should always be the total amount deposited minus the total amount withdrawn. One of the primary functions of the constructor is to start the object in a valid state. (All invariants are true.) Mutator methods (those that change the state) should guarantee that they leave the object in a valid state.

But this is not enough. Suppose that the bank account class has a deposit method. If that method was invoked with a thousand dollars, but it only added a hundred dollars to the total deposits and the balance, it would meet the invariant. (The balance would be the total deposits minus the total withdrawals). Unfortunately, the bank would have some very unhappy customers. The second approach is to ensure the correct operation of the methods. Besides guaranteeing that the state is valid after the method completes, it must be *the correct state*. Additionally, any value returned by the method must be correct. There are other ways to specify the correct operation of methods, but pre- and post-conditions are very common.

- Pre-conditions specify what the method expects to be true before it is invoked.
- Post-conditions specify what must be true after the method is invoked provided that the pre-conditions were met.

For example, consider a deposit method for the bank account class

```
deposit (int amount)
```

What are the pre-conditions? Certainly the bank account must be in a valid state, but is there anything else? Can the deposit be negative? No. This suggests a pre-condition that the amount must be non-negative. The client has the responsibility to guarantee that the pre-condition is met. What happens if the object's client makes a mistake and accidentally invokes the method with a negative value? There are two ways of handling the situation. The first technique is to be safe and test for the pre-condition. If it is not met, the state of the object will be unchanged and an error will be thrown. With the second technique, instead of having the requirement in the pre-conditions, it will be part of the post-conditions. A Boolean return value is added to the deposit method and if the amount is negative, the state will be unchanged and false will be returned. Otherwise, the total deposits will be increased by amount, the balance will be increased by amount, and true will be returned.

It should be mentioned that besides pre- and post-conditions, another way of specifying the behavior of a class is via the use of test code. While test cases are an important tool and we will use them later on, you should not become reliant on them and first know how to design, implement and test your program in steps so as to catch problems early. Passing test cases does not guarantee that the class is behaving correctly anyway.

In today's lab, you will work writing one class that represents a rational number, which is the ratio of two integer values.

Overall Responsibilities

The class `Rational` represents rational numbers, such as $3/2$, $-1/3$ and so on. Rational object should be stored in normal form where the numerator and the denominator share no common factors and with guarantee that only the numerator is negative.

Here is a list of responsibilities for the rational class:

1. Know the value of the denominator.
2. Know the value of the numerator.
3. Be able to compute the negation of a rational number.
4. Be able to compute the reciprocal of a rational number.
5. Be able to compare two rational numbers for equality.
6. Be able to compute the sum of two rational numbers.
7. Be able to compute the difference of two rational numbers.
8. Be able to compute the result of multiplying two rational numbers.
9. Be able to compute the result of dividing two rational numbers.
10. Be able to compute a printable representation of the rational number.

Design Description

Answer the following questions on this print-out.

What values will the `Rational` class need to implement these responsibilities?

Are there any constraints on these values?

Here is a list of constructors and methods that will be used to implement the responsibilities. Fill in the missing pre-conditions, post-conditions, and test cases.

```
Rational()

  Pre-condition: none.
  Post-condition: The rational number 1 has been constructed.
  Test cases: none.


Rational(n, d)

  Pre-condition: The denominator d is non-zero.
  Post-condition: The rational number n/d has been constructed and is in normal form.
  Test cases:
  n = 2, d = 4   ->  1/2
  n = 0, d = 7   -> 0/1
  n = 12, d =-30 ->  -2/5
  n = 4,  d = 0  -> `Exception`


int getNumerator()

   Pre-condition: The rational n/d is in a valid state.
   Post-condition: The value n is returned.
   Test cases:
   n/d is 1/2   -> 1
   n/d is 0/1   -> 0
   n/d is -2/5  -> -2


int getDenominator()

   Pre-condition:
   Post-condition:
   Test cases:
   ..
   ..
   ..
   ..
```

Rational negate()

   Pre-condition: The rational n/d is in a valid state.
   Post-condition: The rational number -n/d has been returned.
   Test cases:
   ..
   ..
   ..
   ..

Rational reciprocal()

   Pre-condition:
   Post-condition:
   Test cases:
   ..
   ..
   ..
   ..

boolean equals(Object other)

   Pre-condition:
   Post-condition:
   Test cases:
   ..
   ..
   ..
   ..

Rational add(Rational other)

   Pre-condition: The rational n/d is in a valid state and other is the valid rational x/y.
   Post-condition: The rational number (ny+xd)/dy has been returned.
   Test cases:
   n/d is 1/2, x/y is 1/2; -> 1/1
   n/d is 1/2' x/y is 1/6; -> 2/3
   n/d is 3/4, x/y is 5/6; -> 19/12
   n/d is 1/3, x/y is -2/3; -> -1/3

Rational multiply(Rational other)

   Pre-condition:
   Post-condition:
   Test cases:
   ..
   ..
   ..
   ..

```
Rational divide(Rational other)

   Pre-condition:
   Post-condition:
   Test cases:
   ..
   ..
   ..
   ..


String toString()

    Pre-condition: The rational n/d is in a valid state.
    Post-condition: The string "n/d" has been returned.
    Test cases:
    n/d is 1/2;   -> "1/2"
    n/d is 0/1;   -> "0/1"
    n/d is -2/5;  -> "-2/5"
```

Implementation

As usual create a directory `Lab3` in your course directory.

Inside it create two files: `Rational.java` and `RationalTest.java`.

Develop them in parallel, i.e. as you define methods in the `Rational` class, create instances of that class in the `main` of `RationalTest` and print values to test your code.

In `Rational.java` you should start with the following steps.

- Create private variables to hold the state of a `Rational` object.
- Implement the default constructor, which creates the rational number 1.
- Implement a private method `normalize` that put the rational number in a normal form. To do so a private class method `gcd` that returns the greatest common divisor of two integers is useful.
- Implement the other constructor. Don't forget to normalize and to handle the zero denominator case: print an error message and `System.exit();` is fine for now.

Hint: Euclid's Algorithm for the gcd evaluation is based on the two following expressions

$gcd(m, 0) = m$

$gcd(m, n) = gcd(n, m \mod n)$

Try some gcd computations by hand to figure out an algorithm.

Checkpoint—At this point you have almost enough to do some testing. First make sure the `Rational.java` code compile. Then add simple methods such as the accessors and/or the `toString`. Create `Rational` object in the `main` method of `RationalTest.java` and use `println` to check the values stored in each instantiated object: are they correct? are they reduced, i.e. the rational in normal form?

At this point show the lab instructor your work: the two classes and the handout. You should not leave the lab without showing us what you accomplished.

To finish the lab implement the six methods described above: `equals` returns a boolean, while the others return a new object of `Rational` type.