# Lab 2.0 Objects

## Martin Nguyen

## Points:100pts

## Goal

In this lab you will explore constructing and testing an object.

## Resources

* Prelude: Designing Classes
* Appendix : Java Classes
* Appendix : Creating Classes from Other Classes

In javadoc directory

* Rational.html—Interface documentation for the class Rational

## Java Files

* *Rational.java*
* *RationalTest.java*
* *ZeroDenominatorException.java*

## 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 data field 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 will be developed in this lab manual are intended to be very general and therefore will usually not have any 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.

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 private data fields 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? If the object just uses the bad value, it could end up in an incorrect state. On the other hand, if the object checks the preconditions, we end up with redundant checks of the preconditions already done by the client. One way to deal with this situation is to use an assertion of the precondition. In our example, it might be something like

assert amount > 0;

If the assertion fails, an error is thrown and the program halts. The advantage of this is that while we are testing our client, we can run with assertions on. Once the testing is complete, we can run the production code with assertions off and avoid the redundant checks.

A different design philosophy would put the entire responsibility for guaranteeing the precondition on the object itself. Trusting another class to do it for you is potentially dangerous. There are two common techniques for dealing with this. In the first technique, if the precondition fails the state of the object is unchanged and an exception is thrown. The client can either catch and handle the exception and continue running, or the program halts. 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 these labs will use them extensively, do not become overly reliant on them. Passing the test cases does not guarantee that the class is behaving correctly.

In today's lab, you will work with one project. The project will represent a rational number that is the ratio of two integer values.

## Pre-Lab Visualization

### Rational

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.

What data fields will the Rational class need to implement these responsibilities?

images:bighand.jpg Get denominator and get numerator.

Are there any constraints on the values of the data fields?

images:bighand.jpg

Yes, when the denominator is zero the ZeroDenominatorException will be thrown here.

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; result is 1/2

n = 0, d = 7; result is 0/1

n = 12, d =–30; result is –2/5

n = 4, d = 0; result is 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; result is 1

n/d is 0/1; result is 0

n/d is –2/5; result is –2

int getDenominator()

images:bighand.jpg Pre-condition:

Make sure that the denominator isn’t zero.

images:bighand.jpg Post-condition:

If it is zero throw a ZeroDenominatorException.

If not then returns value of denominator.

images:bighand.jpg Test cases:

n/d is 1/2; returns 2.

n/d is 2/0 returns ZeroDenominatorException.

Rational negate()

Pre-condition: The rational n/d is in a valid state.

Post-condition: The rational number –n/d has been returned.

images:bighand.jpg 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; result is 1/1

n/d is 1/2' x/y is 1/6; result is 2/3

n/d is 3/4, x/y is 5/6; result is 19/12

n/d is 1/3, x/y is –2/3; result is –1/3

Rational subtract(Rational other)

images:bighand.jpg Pre-condition:

Both rationals are valid (n/d and x/y) and normalized form.

images:bighand.jpg Post-condition:

The rational number (ny-xd)/dy is returned.

images:bighand.jpg Test cases:

n/d is 1/2 , x/y is 1/2 result 0 is returned.

Rational multiply(Rational other)

images:bighand.jpg Pre-condition:

The rational n/d and other is the valid rational x/y.

images:bighand.jpg Post-condition:

The rational number (nx/dy) is returned.

images:bighand.jpg Test cases:

n/d is 2/3; x/y is 4/5; result is 8/15;

Rational divide(Rational other)

images:bighand.jpg Pre-condition:

The rational n/d and other is the valid rational x/y.

images:bighand.jpg Post-condition:

The rational (ny/dx) is returned.

images:bighand.jpg Test cases:

n/d is 1/4 ; x/y is 3/2; result is 1/4 \* 2/3 which is 1/6.

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; result is “1/2”

n/d is 0/1; result is “0/1”

n/d is –2/5; result is “–2/5”

## Directed Lab Work

### Rational

The skeleton of the Rational class already exists and is in *Rational.java*. Test code has been created and is in *RationalTest.java*. You will complete the methods for the Rational class.

1. If you have not done so, look at the interface documentation in *Rational.html*. Look at the skeleton in *Rational.java*. All of the methods exist, but do not yet do anything. Compile the classes ZeroDenominatorException, Rational, and RationalTest. Run the main method in RationalTest.

*Checkpoint: If all has gone well, you should see test results. Don't worry for now about whether the test cases indicate pass or fail. Don't worry about the null pointer exception. All we want to see is that the Rational class has the correct protocol. Now you will complete the heart of the Rational class, its constructors, and basic accessor methods.*

1. Create the private data fields that will hold the state of a Rational object.
2. Complete the default constructor. It should create the rational number 1.
3. Complete the private method normalize. It should put the rational number in a normal form where the numerator and denominator share no common factors. Also, guarantee that only the numerator is negative. The gcd (greatest common divisor) method may be of use to you.
4. Complete the alternate constructor. It should throw a new ZeroDenominatorException if needed. Don't forget to normalize.
5. Complete the method getNumerator().
6. Complete the method getDenominator().

*Checkpoint: At this point there is enough to test. Your code should compile and pass all the tests in testConstructor(). If it fails any tests, debug and retest. The next two methods chosen for implementation are simple methods that construct a new rational number from an existing rational object.*

1. Complete the method negate(). Note that this method should not change the rational number it is invoked on, but instead return a new rational object. Don't forget to change the return statement. Currently it returns null, which means after executing the line of code

Rational r2 = r1.negate();

the variable r2 will have the value null. If any methods are invoked on null (e.g., r2.getNumerator()) a null pointer exception will occur.

*Checkpoint: Your code should compile and pass all the tests up to and including testNegate(). If it fails any tests, debug and retest. If you get null pointer exception before the test indicates it is finished with the negate testing, check what you are returning.*

1. Complete the method add(other).
2. Complete the method subtract(other). There are a couple of ways that you can implement subtraction. One way is to use a formula similar to the one used for addition. Another way is to negate the second argument and then add. Either technique will work.

*Checkpoint: Your code should compile and pass all the tests through testAddSubtract(). If it fails any tests, debug and retest. Again the next two methods are closely related and will be implemented together.*

1. Complete the method multiply(other).
2. Complete the method divide(other).

*Final checkpoint: Your code should compile and pass all the tests.*

1. Submit the .**doc** and .**java** files on iCollege