CS 0445 – Data Structures (COE 0445)

Sherif Khattab

Lab 1 Objects

Goal

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

Resources

- Prelude: Designing Classes
- Appendix C: Java Classes (Online)
- Appendix D: Creating Classes from Other Classes

In `javadoc` directory

- Rational.html—Interface documentation for the class Rational
- Counter.html—Interface documentation for the class Counter

Java Files

- *Counter.java*
- *CounterInitializationException.java*
- *CounterTest.java*
- *Rational.java*
- *RationalTest.java*
- *ZeroDenominatorException.java*

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.

Step 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.

Step 2. Create the private data fields that will hold the state of a Rational object.

Step 3. Complete the default constructor. It should create the rational number 1.

Step 4. 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.

Step 5. Complete the alternate constructor. It should throw a new ZeroDenominatorException if needed. Don't forget to normalize.

Step 6. Complete the method getNumerator().

Step 7. 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.

Step 8. 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.

Step 9. Complete the method reciprocal().

Checkpoint: Your code should compile and pass all the tests through testInvert(). If it fails any tests, debug and retest. The next two methods chosen for implementation are closely related and will be tested together.

Step 10. Complete the method add(other).

Step 11. 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.

Step 12. Complete the method multiply(other).

Step 13. Complete the method divide(other).

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

Counter

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

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

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. All we want to see is that the Counter class has the correct protocol. Again we will work from the heart of the class outward. Your first task is to complete the constructors.

Step 2. Create private data fields that will hold the state of a Counter object.

Step 3. Complete the default constructor. It should create a counter with a minimum of 0 and a maximum that is the largest possible integer value (Integer.MAX_VALUE).

Step 4. Complete the alternate constructor. It should check to see if the minimum value is less than the maximum value and throw an exception if not.

Checkpoint: At this point we will verify that the exception is correctly generated. Your code should compile and pass all the tests in testConstructor(). If it fails any tests, debug and retest. This is not a complete test of the constructors and you may have to revise them. The toString() method is useful to implement early because it reports on the state of an object without changing it. It can then be used in later test cases. It is also one of the methods that classes typically override.

Step 5. Complete the method toString().

Checkpoint: Your code should compile. There is no mandated format for your toString() method. Check that it produces all the information given by the print statements in testToString. If not, debug and retest. Another method that is typically overridden is the equals() method. You will work with it next.

Step 6. Complete the method equals(). It has been started for you and will test to make sure that the other object is of the same type. Complete the then clause of the if statement to check that all the private state data fields have the same value.

Checkpoint: Your code should compile and pass all the tests through testEquals(). If it fails any tests, debug and retest. There are two final accessor methods to complete and then the mutators will be implemented.

Step 7. Complete the method value().

Step 8. Complete the method rolledOver().

Step 9. Complete the method increase().

Check point: Your code should compile and pass all the tests through testIncrease(). If it fails any tests, debug and retest. This is really the first test that exercises a major portion of the responsibilities of the Counter class. Up until now the state of the class should not have been affected by the methods. We use the accessors to test the state of the object after the mutator has been called.

Step 10. Complete the method decrease().

Checkpoint: Your code should compile and pass all the tests. The tests in testDecrease() are similar to what you have seen before. The decrease mutator is applied and the state is queried using the accessors. There is a different style *of test being performed by testCombined(). It tests to see if the increase and decrease mutators are inverses of one another. Most of the time an increase followed by a decrease should leave the object in its original state.*

Post-Lab Follow-Ups

1. Compare the test cases from the RationalTest class with the ones you created in the pre-lab. Were there kinds of test cases that you did not consider? Were there kinds of test cases that you proposed that were not in the RationalTest class?

2. Compare the constructors and methods from the Counter class with the methods you proposed in the pre-lab. Were there methods that you did not consider? Were there methods you proposed that were not in the Counter class? Do expectations for the methods as expressed in the CounterTest class differ from what you expected? Can you justify your omissions and additions?

3. You probably used two private data fields (numerator and denominator) in the implementation of your Rational class, but there are other options. For example, we could have used three fields (sign, numerator, and denominator). In this case, both the numerator and denominator would be guaranteed to be positive and the sign field indicates whether the object is positive or negative. How would this have changed the implementation of the methods of the Rational class?

4. Come up with a new implementation of the Counter class that uses different data fields. How does this affect the methods of the class?

5. Implement and test equals and toString for the Rational class.

6. Think further about a class that would represent a bank account. Give responsibilities for it. List the data fields and any constraints. Give a list of methods with their pre-conditions, post-conditions, and test cases.

7. Think about a class that would represent a colored triangle that could be displayed on a computer screen. Give responsibilities for it. List the data fields and any constraints. Give a list of methods with their pre-conditions, post-conditions, and test cases.