

Lab 1 Objects

Goal

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

Resources

- Prelude: Designing Classes
- Appendix B: Java Classes
- Appendix C: 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`

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 two classes. The first class will represent a rational number that is the ratio of two integer values. The second class will be a counter that has both a minimum and maximum value.

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?



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



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()`



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 $1/2$;	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)`



Pre-condition:



Post-condition:



Test cases:

`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 $1/2$;	result is " $1/2$ "
n/d is $0/1$;	result is " $0/1$ "
n/d is $-2/5$;	result is " $-2/5$ "

Counter

Our counter will be a class that acts like a simple click counter (used for counting attendance) with a few improvements. The click counter will have a minimum and maximum value. It will start at the minimum value. Each click will add one to the counter, except when the counter hits the maximum value, where it will roll back over to the minimum. The click counter will also support an operation that decreases the value of the counter by one. If this would decrease the value below the minimum, it will roll over to the maximum value.

Think about the preceding description and give a list of responsibilities for the `Counter` class.



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



Are there any constraints on these data fields?



Give a list of constructors and methods that will be used to implement the responsibilities you have listed. Fill in the pre-conditions, post-conditions, and test cases.



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 data that you did not consider? Were there kinds of test data 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 data 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.