Introduction to Computer Science, IDC Herzliya, 2020

Homework 8

In this assignment you will practice working with classes and objects. You will do this both from a client's perspective and from a developer's perspective. The assignment consists of three parts. In Parts I and II you will build an object-oriented infrastructure for representing two abstractions: *fractions*, and *expressions*. In Part III you will use this infrastructure for implementing an educational application that practices children in fraction arithmetic.

Read the entire document, including Part I, even if you've already completed and tested the `Fraction` class. You now have to make some changes to your implementation, as described below.

Part I: Fractions

The fraction data type consists of objects of the form <*numerator*, *denominator*>, where *numerator* is a signed integer and *denominator* is a nonnegative integer. The supplied `Fraction` class provides a representation of fraction objects. Most of the class methods are implemented, except for the following ones:

`f.abs()`: Returns the absolute value of `f`. For example, the absolute value of `-2/3` is `2/3`.

`f.signum()`: Returns -1 if `f` is negative, 0 if `f` equals 0, and 1 if `f` is greater than 0.

`f.convert()`: Returns the negative value of `f`. For example, the negative value of `2/5` is `-2/5`.

`f1.subtract(f2)`: Returns `f1 - f2`. For example, `2/5 - 1/3 = 1/15`.

`f1.compareTo(f2)`: Returns 1 if `f1` is greater than `f2`, 0 if the two fractions are equal, and -1 if `f1` is less than `f2`.

`f1.equals(f2)`: Returns true if `f1` equals `f2`, false otherwise.

`f.power(n)`: Returns `f` raised to the power of `n`. For example, 2/3 raised to the power of 3 is 8/9.

`Fraction(limit)`: A constructor that returns a random fraction. The fraction's denominator is a random integer d so that $2 \leq d < limit$, and the numerator is a random integer n so that $1 \leq n < d$. The given `limit` is assumed to be at least 2.

Reduce (update): In the `Fraction` class presented in lecture 9-1 the fractions were reduced "automatically". For example, a constructor call like `new Fraction(8,16)` produced the fraction 1/2. In the present implementation we no longer reduce fractions. To do this, simply eliminate the call to the `reduce()` method from the `Fraction` constructors. Following this change, the values of arithmetic operations will not be reduced. For example, the operation 1/4 + 1/4 will produce the fraction 2/4, which is left as is. The `reduce` method is still useful, and you may want to use it in other parts of this project.

ToString (update): A "divisible fraction" is an imperfect fraction in which the denominator divides the numerator. For example: 4/2 and 21/7. In the `Fraction` class presented in lecture 9-1 the `toString` method reduced divisible fractions to their common divisors. For example, if the current fraction was 6/3, the `toString` method returned the string `"2"`. In this implementation the `toString` method no longer reduces divisible fractions. Thus if the current fraction is 6/3, the method returns the string `"6/3"`.

The reasons for these two changes will become clear later in the document.

Implementation tips

Important: Implement the methods in the order described below. Test each method in isolation. Don't proceed to implement the next method until the current one is fully written and tested.

`abs:` Read the `Fraction` constructor, and make sure that you understand how it handles negative parameters. You will notice that thanks to the constructor's logic, negative instances of the `Fraction` class are always represented in one way only. Then go ahead and implement the `abs` function. Notice that like several other `Fraction` methods, the `abs` method calls the `Fraction` constructor in order to construct the fraction object that it has to return.

`signum:` Can be implemented by inspecting the numerator and denominator of this fraction. Note: methods of the class don't have to use the class getter methods; they can access the fields directly. The getter methods are designed to allow methods from other classes access the fields.

`convert:` Can be implemented in one line of code. Hint: the fraction -1/1 can come handy.

`subtract:` Can be implemented in one line of code. Hint: use the `convert` function.

`compareTo:` Can be implemented in one line of code. Hint: use two of the methods that you've already implemented.

`equals:` Can be implemented trivially by calling one of the methods that you've already implemented.

`power(n):` A possible implementation is to repetitively call the `multiply` method. This is a bad idea, since each multiplication creates a new `Fraction` object that remains in memory and wastes space. A more sensible solution is to start with the numerator and denominator of this fraction, and create the new fraction from them.

`Fraction(limit):` Don't call the other constructor using `this(`*argsList*`)`, as we did with `BankAccount`. This technique will not work here, for the following technical reason. According to Java's rules, the `this(`*argsList*`)` call – when used – must be the first statement in the constructor. This will not work here, since this particular constructor must first generate some random numbers. Remember *not* to reduce the newly created random fraction.

The MyMath class

The supplied `Fraction` class works well, but OOP purists (and job interviewers) will frown at one aspect of its design. Ideally, a class that is designed to represent objects should expose only *member methods*, i.e. methods that operate on the current object. Alas, the supplied `Fraction` class contains one public method that violates this principle.

`static int gcd(int,int):` The inclusion of the `gcd` method in the `Fraction` class presents two problems. First, it is a static method (also called *function*). As such, it doesn't operate on the current object. Therefore, it has no business being in the `Fraction` class. Second, `gcd` is a useful mathematical function that can come to play in many programs that have nothing to do with fractions.

So, where should we put the `gcd(int,int)` function? A natural place will be a class in which we put commonly used mathematical functions.

Let's go ahead and carry out this refactoring. First, cut the `gcd` code out of the supplied `Fraction` class. Second, paste it into the supplied `MyMath` class. Finally, find the place in the `Fraction` class code where we call the `gcd` function, and modify it to call `MyMath.gcd` instead. Then test the `Fraction` class code and make sure that this trivial surgery did not mess up things. Write this testing code in the supplied `HW8Test.testFraction` function, and execute it.

`static int commonDenominator(Fraction,Fraction):` This function (static method) returns the common denominator of the two given fractions. For example, the common denominator of 1/2 and 1/4 is 4, the common denominator of 2/15 and 1/5 is 15, and the common denominator of 3/4 and 1/6 is 24. Notice that this is not the least common denominator. Here is the exact definition: Let $d1$ and $d2$ be the two denominators. If $d1$ is a multiple of $d2$, return $d1$; if $d2$ is a multiple of $d1$, return $d2$; Otherwise, return $d1 \cdot d2$. Hint: you'll have to use one of the `Fraction` class getters.

`static Fraction max(Fraction,Fraction):` This function (static method) returns the maximum of the two given fractions. For example, the maximum of 1/2 and 1/4 is 1/2, and the maximum of 2557/3701 and 312/417 happens to be 312/417.

Hint: Use one of the `Fraction` methods that you developed previously in this homework.

Testing

The supplied `FractionTest` class provides a partial skeleton for unit testing all the testing methods that you have to write (this class replaces the `HW8Test.java` class that was given previously). Here is an example of executing the fully implemented code of this class:

```
% java FractionTest
Testing the abs function:
abs(1/2) = 1/2
abs(-1/2) = 1/2
abs(0) = 0

Testing the signum function:
signum(1/2) = 1
```

```
signum(-1/2) = -1
signum(0) = 0

Testing the convert function:
1/2 converted is -1/2
-1/2 converted is 1/2
0 converted is 0

Testing the subtract function:
1/2 - 1/2 = 0
1/2 - 1/4 = 2/8
1/4 - 1/2 = -2/8
1/2 - -1/2 = 4/4
-1/2 - 1/2 = -4/4

Testing the compareTo function:
1/2 compared to 1/4 returns 1
1/4 compared to 1/2 returns -1
1/2 compared to 1/2 returns 0

Testing the equals function:
1/2 == 1/2? true
1/2 == 1/4? false

Generating 10 random fractions with limit = 7:
4/5
3/4
3/6
1/2
5/6
2/4
1/3
...

Testing the commonDenminator function:
The common denominator of 1/2 and 1/4 is 4
The common denominator of 1/4 and 1/2 is 4
The common denominator of 1/3 and 1/3 is 3
The common denominator of 1/3 and 1/2 is 6

Testing the max function:
max(1/2,1/4) = 1/2
max(1/4,1/2) = 1/2
max(1/2,1/2) = 1/2
```

The testing code that you have to write should generate the same outputs.

This website provides a fraction calculator that may help you in your testing activities.

Part II: Expressions

In this project we use the term *expressions* to refer to objects that represents algebraic expressions of the form *op*1 *op* *op*2, where *op*1 and *op*2 are fractions and *op* is one of the operations `+`, `-`, `*`, `:`, or `>`.

The supplied `Expression` class is designed to represent expressions. It features the following methods:

`toString():` Returns a string representation of the current expression. Negative fractions are enclosed in parentheses. For example, if `op1` is `1/5`, `op2` is `-1/3`, and `op` is `"-"`, the method returns the string `"1/5 - (-1/3)"`.

`Expression(limit):` A constructor that returns a random expression. A random expression consists of three fields. The fields `op1` and `op2` refer to two random `Fraction` objects, each of limit `limit`, and the `char` field `op` contains one of the values `'+'`, `'-'`, `'*'`, `':'`, or `'>'`.

`exp.value():` Returns the value of the expression `exp`, as a fraction, with one exception: If the expression's operator (`op`) is `'>'`, returns the value `op1 - op2`. For example, if `exp` is `1/4 + 1/4`, returns `2/4`. If `exp` is `2/7 > 5/7`, returns `-3/7`.

`exp.hint():` Returns a different expression that has the same value as `exp`. For example, if `exp` is `1/2 - 1/3`, returns the expression `3/6 - 2/6`. Which expression to return out of the infinitely many possibilities? This question is answered in the *Hints* section presented in section III below.

Implementation tips

`Expression(limit):` The two random fractions can be generated by calling the `Fraction(limit)` constructor. The random operator can be generated using the static array `ops`.

`toString():` The sign of each operand can be checked by calling a `Fraction` method.

`value():` The value of the current expression, which is a fraction, can be computed easily by calling relevant `Fraction` methods. Which algebraic operation to compute is determined by the value of the `op` field. This method can be implemented elegantly using Java's switch statement.

`hint():` This method generates the hint expression of the current expression by calling one of four specific hint-generation methods. Which method to call is determined by the `op` field of the current object. This method can also be implemented elegantly using a switch statement. **But, don't implement it until you read Part III.**

Testing

The supplied `ExpressionTest` class provides a partial skeleton for unit testing all the methods that you have to write. Here is an example of executing the fully implemented code of this class:

```
% java TestExpression
Generating 10 random expressions with limit = 7:
1/4 : 1/5
1/3 : 1/2
1/4 > 1/2
```

```
1/2 + 1/2
1/2 * 1/2
1/2 > 4/6
3/5 > 1/2

Generating 10 random expressions with limit = 7,
and printing their values (which may not be reduced -- that's ok):
1/4 + 1/2 = 6/8
1/2 - 3/5 = -1/10
1/3 + 3/6 = 15/18
1/2 > 1/4 = 2/8
1/2 * 1/2 = 1/4
1/2 - 1/2 = 0
1/3 > 1/5 = 2/15
```

Note: The rest of the testing (below) should be done only after reading Part III.

```
Generating 10 random expressions with limit = 7,
and printing their hints (multiplication expressions have no hints -- that’s ok):
The hint for 3/4 : 1/2 is: 3/4 * 2/1
The hint for 1/2 > 1/2 is: 1/2 - 1/2
The hint for 1/3 > 1/2 is: 1/3 - 1/2
The hint for 3/5 + 1/2 is: 6/10 + 5/10
The hint for 1/2 * 1/2 is: null
The hint for 1/2 + 1/3 is: 3/6 + 2/6
The hint for 1/2 * 3/6 is: null
```

Part III: FracPrac

The `Fraction` and `Expression` classes built in parts I and II can serve many different applications. In Part III you will develop one such application, named `FracPrac`.

As you may remember from elementary school, the introduction of fractions can be a traumatic experience to many children. The `FracPrac` program is designed to help children practice fraction arithmetic exercises. The program presents a sequence of algebraic questions, and gives hints when asked to do so. Here is a typical practice session (the user's answers are underlined):

`java FracPrac 7` (the command line argument represents a difficulty level)

```
Welcome to fractions practice!
Here is your first question:
1/2 - 2/3 = ?
hint
Notice that 1/2 - 2/3 = 3/6 - 4/6. Try again:
1/2 - 2/3 = ?
-1/6
Correct! Next question:

1/2 - 1/4 = ?
1/2
Incorrect... Try again:
1/2 - 1/4 = ?
1/4
Correct! Next question:

1/4 : 2/3 = ?
difficult
Enter a valid answer.
1/4 : 2/3 = ?
hint
Notice that 1/4 : 2/3 = 1/4 * 3/2. Try again:
1/4 : 2/3 = ?
3/8
Correct! Next question:

2/3 : 2/5 = ?
pass
2/3 : 2/5 = 5/3
New question:

2/3 > 1/5 = ?
hint
Notice that if 2/3 - 1/5 > 0, the answer must be true. Try again:
2/3 > 1/5 = ?
true
Correct! Next question:
```

```
1/2 * 1/3 = ?
hint
Multiply the two numerators, and divide by the product of the denominators. Try
again:
1/2 * 1/3 = ?
1/6
Correct! Next question:

1/2 : 1/2 = ?
1
Correct! Next question:

1/4 + 3/4 = ?
hint
Add up the numerators and divide by the common denominator. Try again:
1/4 + 3/4 = ?
4/4
Correct! Next question:

1/2 > 1/2 = ?
false
Correct! Next question:

1/2 : 1/4 = ?
quit
Bye now!
```

Hints

The user's answer to each question can be one of four things: a string like `"3/4"` or `"2"`, which stands for "this is my answer to the question", the string `"hint"`, which stands for "give me a hint", the string `"pass"`, which stands for "show me the answer and give me another question", or the string `"quit"`, which stands for "quit the program". There are four generic hints, as follows.

Addition or subtraction questions: If the two fractions have the same denominator, the program hints as follows (example):

```
2/3 + 4/3 = ?
hint
Add up the numerators and divide by the common denominator. Try again:
2/3 + 4/3 = ?
```

If the two fractions have different denominators, the program hints as follows (example):

```
1/2 - 1/3 = ?
hint
Notice that 1/2 - 1/3 = 3/6 - 2/6. Try again:
1/2 - 1/3 = ?
```

Multiplication questions: There are no customized hints for multiplication questions; The program always hints as follows (example):

```
1/2 * 2/3 = ?
hint
Add up the numerators and divide by the common denominator. Try again:
1/2 * 2/3 = ?
```

Division questions: The program hints as follows (example):

```
2/7 : 1/5 = ?
hint
Notice that 2/7 : 1/5 = 2/7 * 5/1. Try again:
2/7 : 1/5 = ?
```

Comparison questions: The program hints as follows (example):

```
2/7 > 1/5 = ?
hint
Notice that if 2/7 - 1/5 > 0, the answer must be true. Try again:
2/7 > 1/5 = ?
```

Implementation tips

`The main method:` The code of this method is given. Read it carefully, and make sure that you understand the logic. This code illustrates the use of two new things: the `switch` statement, and the handling of an *exception* (`try - catch`). You can read about `switch` here, and exceptions will be discussed in the recitations of this week.

`PrintCorrectAnswer:` The implementation of this method is simple. It uses the services of (calls methods of) the `Fraction` and `Expression` classes.

`isCorrect:` The code of this simple method is given. If the question is an *addition*, *subtraction*, *multiplication*, or *division*, we check the user's answer by calling **`isCorrectAddSubMultDiv`**. If the question's expression is a *comparison*, we call **`isCorrectComp`**. These two methods are described next.

`isCorrectAddSubMultDiv``(Expression exp, String ans):` In this method we finally get to the point where we try to convert the user's answer (`ans`) to a fraction, and compare this fraction to the value of `exp`. The string `ans` should be either "`int/int`", or "`int`". If it's the former, you can convert it to a fraction and proceed to perform the comparison. If it's the latter, you can do the same with a fraction whose denominator is 1. The bad news is that the user can enter whatever he or she pleases, not necessarily valid inputs. The good news is that all these "number format exceptions" are caught by Java's `Integer.parseInt` method, which you'll probably want to use. If you'll inspect the code of the `main` method, you will notice that these exceptions are propagated to, and handled, by the method's `try-catch` logic (more about this in this week's recitations). For this reason, there is no need to write any input validation code. Assume that `ans` is either "`int/int`" or "`int`", and take it from there.

`isCorrectComp``(Expression exp, String ans):` This method is similar to the one just described. Here `ans` must be either `true`, or `false`.

`printHint(Expression exp)`: This simple method uses one `Expression` method to figure out which expression we have, and another `Expression` method to produce the necessary hint. Consider using a `switch` statement.

Take Home Lessons

In this course we believe in learning by doing. If you completed this program successfully, you've experienced, hands-on, the power and elegance of Object Oriented Programming. Take some time to play with `FracPrac` and enjoy the fruits of your work.

Submission

Zip the following files into the file `hw8.zip`:

- `Fraction.java`
- `Expression.java`
- `MyMath.java`
- `FracPrac.java`

GETFEED will be available soon.

Deadline: Submit Homework 8 no later than January 5, 2021, 23:55.