CSC 360 Programming assignment #5
Due Date: Monday, October 6

This assignment will test you on creating and using your own Exception classes as well as File input using File and Scanner and implementing an interface. In this assignment, you will create 5 classes:

- Fraction: to define a Fraction object and various methods that manipulate Fractions
- FractionDenominatorZeroException
- LossOfPrecisionException
- FileInputIncorrectException
  - you can name these Exception classes something else if you like
- FractionUser to create and use Fractions

Fraction's UML----------> +compareTo(Object) : int [implements Comparable]
-numerator: int
-denominator: int
-sign: int // store 1 if sign from input is +, -1 if sign from input is -
---
+Fraction(numerator: int, denominator: int, sign: int)
+Fraction(numerator: int, denominator: int, sign: char)
+Fraction( )
+getNumerator( ): int
+getDenominator( ): int
+getSign( ): int
+toString( ): String
+computeFraction( ): double throws FractionDenominatorZeroException
+getPercentage( ): String (of the form ##.##%) throws FractionDenominatorZeroException, LossOfPrecisionException
+reduce( ): Fraction throws FractionDenominatorZeroException
+add(Fraction): Fraction throws FractionDenominatorZeroException
+multiply(Fraction): Fraction throws FractionDenominatorZeroException
+divide(Fraction): Fraction throws FractionDenominatorZeroException

The no-arg constructor will create the Fraction + 0 / 1. The other constructors should be self-explanatory. Note that you can build a Fraction by passing the sign as a number (1 or -1) or a character ('+' or '-'). Also note that it is permissible to create a Fraction whose denominator is 0 through the constructor. This will be caught using exception handling as described later. The accessor methods should be self-explanatory.

The toString method will return the Fraction as a String in the form `sign numerator / denominator` as in `+ 3 / 1000` or `- 3 / 2` (you can omit the spaces if desired as in `+3/1000`).

The computeFraction method will return the double obtained by performing sign*numerator/denominator. The getPercentage method will return the Fraction as a String in the format of [-]##.##%. The - is only added if the value is negative, do not output a + if the value is positive. There must be at least one digit prior to the decimal point and can be as many as needed and there will be exactly 2 digits to the right of the decimal point even if they are zero. For instance, - 45 / 30 would be returned as -150.00% while the fraction + 1 / 250 would return 0.40%. This method can round or truncate, whichever you prefer. For instance, + 1 / 15000 would be 0.0067%, this can either round up to 0.01% or truncate to 0.00%. To compute the percentage, store as an int the value of sign*10000*numerator/denominator, and then divide

this value by 100.0 and store in a double. Then take the double and cast as a String adding "%" to the end. See also the discussion on Exceptions in four paragraphs.

The reduce method will reduce the Fraction as much as possible. For instance, if we have the Fraction (1, 90, 120), and we call reduce on it, it will change to (1, 3, 4). It will reduce its own numerator and denominator. That is, it operates on *this*. One approach is to mod the numerator and denominator by 2 and if they both result in 0 (no remainder), divide them by 2 and repeat. If they are not divisible by 2, try 3, etc until the divisor reaches the denominator. There are other approaches (you can start with the denominator and decrement the divisor until it reaches 1, do not try to divide by 1 because it could lead to an infinite loop).

The add, multiply and divide methods will each receive a second Fraction and return a new Fraction (they will not alter *this* Fraction) by adding, multiplying or dividing *this* Fraction and the parameter. For instance, if *this* is (1, 3, 2) and we pass it the message `add(new Fraction(-1, 1, 4))` then it will return the new Fraction (1, 5, 4), that is, 3/2 + -1/4 = 5/4. The three methods (add, multiply, divide) should invoke reduce on the new Fraction before returning it so that the computed Fraction is as reduced as possible. To implement add, you will have to normalize the two Fractions to have the same denominators. For instance, (-1, 1, 12) and (1, 6, 5) would need denominators of 60 each, resulting in the values (-1, 5, 60) and (1, 72, 60). You also have to factor in the two signs to see if you would add both numerators or subtract one from the other. In this case, we would have 72 – 5 as the new Fraction's numerator with a sign of 1. Once the new Fraction is obtained by adding, multiplying or dividing, reduce it. The last operation of these methods is to return this reduced Fraction.

The Fraction class will also implement Comparable by implementing a compareTo method. The compareTo method will receive a Fraction as an Object. You will have to perform the proper downcast on the parameter before comparing it to *this*. The compareTo returns an int where this value is positive if *this* > the parameter, negative if *this* < the parameter, and 0 if they are equal. You may return 1, and -1 for > and <, or some other positive and negative value depending on how you implement it. Note that to be equal, they must have the same sign. There are many ways to implement compareTo. One way is to negate the parameter and add it to *this* and if the new Fraction is 0, return 0 otherwise return the sign. To negate the parameter, multiply it by the Fraction (-1, 1, 1) (that is, multiply it by -1). You can also implement a negate method to do this (rather than using multiply). If you implement negate, make it a private method. Note that add (and multiply) has the possibility of throwing an Exception but you cannot throw an Exception from compareTo. Therefore in compareTo you will have to use try/catch blocks. See the next paragraph.

Aside from Fraction, implement three Exception classes, all of which extend Exception. These classes will consist only of two constructors, a no-arg constructor and a constructor which receives a String (the message). You may name your Exceptions with different names that those listed here. The FractionDenominatorZeroException will be thrown if you attempt to use a Fraction with a 0 denominator. This Exception should be thrown from the methods computeFraction, getPercentage, reduce, add, multiply and divide if there is any attempt to operate on a Fraction with a 0 denominator. A Fraction with a 0 denominator can still be used if you are only using its accessors or toString methods. The LossOfPrecisionException will be thrown by getPercentage if, in converting the Fraction into a String you lose precision. Aside from computing the int value sign*10000*numerator/denominator and then the double of this value/100.0, you will need to compare this double to that returned by computeFraction( ). If they are the same then there is no loss of precision. If they differ, then there is a loss of precision and the getPercentage method should throw LossOfPrecisionException. The message should look something like this:

```
Warning:  Fraction + 1 / 20000 as a percentage is 0.01% which has
a loss of precision
```

You can generate such a message by using the Fraction's toString and the value computed as the String for getPercentage. As you can see here, we have a loss of precision. Note that in this implementation of getPercentage, the percentage was rounded up instead of truncated. If you truncate, the message would be the same except the percentage would appear as 0.00%. The two exceptions described here are thrown only from within the Fraction class. You will have to catch the FractionDenominatorZeroException in your compareTo method because the interface class defines compareTo without a throws statement. So you must put the code to compare the Fractions in compareTo within a try/catch block. However, the remainder of the methods should explicit have `throws` *`ExceptionType`* in their header to throw the Exception. You will catch any of these Exceptions in your FractionUser class.

The third Exception will be thrown from main when inputting information from a text file. This Exception will be a FileInputIncorrectException. You will throw this Exception if the input does not match what is expected. This will be the case if there is a non '+' or '-' character for a sign or if the operation to perform is not one of the legal operators (listed below). In addition, you should implement a catch block for a InputMismatchException in case a value input as an int is not an int.

Your FractionUser program (the main method) should have a while loop that iterates while there is still content in the input text file. For each line of input, your program will parse it one String at a time using `input.next( );`. Your program will then determine if the input is as expected and if so, attempt to perform the operation. The textfile consists of several lines, formatted as follows:

`S N D` *`operator`* `[S N D]`

where S is the next Fraction's sign (a '+' or '-'), N and D are int values for the numerator and denominator respectively. [S N D] means optional and is determined based on the operator. *Operator* will be one of the following:

- o (output using toString)
- c (compute the double of the Fraction and output the result)
- p (compute the String percentage and output the result)
- r (reduce the Fraction and output the result)
- + followed by another S N D to add the two Fractions together and output the result
- * followed by another S N D to multiply the Fractions together and output the result
- / followed by another S N D to divide the Fractions and output the result
- ? followed by another S N D to compare the two Fractions and output the result

Every item is separated by a space as in `- 5 12 o` or `+ 1 10 * - 2 5`.

If when reading the input text file you come across something unexpected, then throw a FileInputIncorrectException and report that the given line of input is being skipped. Your program should then continue with the next line of input. Use proper try-catch blocks in main, and place this inside a while loop that iterates until you reach the end of the input file. Your code might look something like the following (assuming `in` is your Scanner).

```
while(in.hasNext( )) {
      try {
            temp=in.next( );
            sign=temp.charAt(0);
            if(sign!= '+'&&sign!='-') throw new … (sign +
                  " is an incorrect sign, skipping line of input");
            num=in.nextInt( );
            denom=in.nextInt( );
            temp=in.next( );
            operator=temp.charAt(0);
            if (operator is not legal) throw new … (operator +
                  "is an incorrect operator, skipping line");
```

```
                else if operator is one of +, *, /, ?
                {
                      temp=in.next( )
                      sign2=temp.charAt(0);
                      if(sign!= '+'&&sign!='-') throw new … (sign +
                            " is an incorrect sign, ...");
                      num2=in.nextInt( );
                      denom2=in.nextInt( );
                }
                // create the Fraction (sign, num, denom)
                //    and process the operation
                //   creating a second Fraction if there is one
                // output results
          }
          catch(…) {…}     // catch the various types of
          catch(…) {…}     // exceptions that can arise including
          catch(…) {…}     // InputMismatchException
          catch(…) {…}
    } // end while loop
```

As described above, main will iterate line by line through the input file and create a Fraction (or two) and process the operation. But if an Exception arises either during the input process or because of the operation, your program will throw the exception, catch the exception, output the exception, indicate that the current line of input is being skipped, and then read the remainder of the line using `in.nextLine( )` (this will ensure that the next `in.next( )` statement takes place at the beginning of the next line). Since the try/catch blocks are all within the while loop, if the file still has another line, the while loop continues. Thus, you skip one input but the program continues instead of terminating. The while loop will exit only when you have reached the end of the file. Outside of your while loop, make sure you close your input file.

Run your user program on the file posted to the website and hand in the 5 classes and the output from running the user class. The following is the output generated from the first 6 lines of the program:

```
50.0%

- 6 / 13

-0.46153846153846156

LossOfPrecisionException: Fraction is - 6 / 13 which is -
0.46153846153846156 as a double, loss of precision to -46.15%

- 6 / 0

FractionDenominatorZeroException:  Fraction  is  -  6  /  0,  with  0
denominator cannot compute fraction
```