CS 2300 Project 2: Fraction Class

Due: Monday October 31, 2022 at 11:59 p.m.

Project Goals.

1. Learn to develop a basic data class. In this case, the data is a **fraction**, also called a **Fraction number**.

2. Develop habit of testing each method in the class as soon as it is written.

3. Follow the "Start small. Done one thing at a time. Test it. Continue." paradigm.

1. Start a new project:

1.1. Fire up Intellij IDEA, start a **new project**. Give the new project the name **Fraction**.

1.2. Right click on **scr** and select **New Class**. Give it the name **`fraction.Fraction`** to create a package and class in it as usual. This will create an empty class that is the heart of the project. Do **not** put a main method in this class.

1.2. Right click on the `fraction` package, select new, select Java Class, name the new class `Utility`. This will create an empty class. Do **not** put a main method in this class. This is a "bag class", whose role is a place for one or more **helper methods**.

1.3. Right click on the `fraction` package, select new, select Java Class, name the new class `Tests`. This will create an empty class. Later we will put a bunch of tests in this class to make sure all operations of the Fraction class work correctly.

Put a main method in this class with the following code in it:

```
System.out.println("\nStart Fraction Tests.\n");

System.out.println("\nEnd Fraction Tests.\n");
```

Run the program (SHIFT + F6) to see what you have so far.

1.4. BTW, over on the left is a kind of "project menu", that looks something like this:

```
scr
   fraction
```

```
    Fraction
    Tests
    Utility
```

This shows you what classes you have, and you can double click on them to move from class to class.

2. Create a special exception class for the project.

Discussion. A good property for computer code is being **robust**, which means capable of dealing with unexpected or bad things as gracefully as possible. When bad things (invalid input, attempting to divide an integer by zero, out of memory, etc.) are detected in a program, there are several things you can do, including

(1) halt the program,

(2) overrule the bad data,

(3) if there is an active user, ask for different input, and

(4) **throwing exceptions**.

Here is the deal with throwing exceptions. The idea is to try to let the appropriate part of the program deal with whatever the issue is. This is done by **throwing** some information to be caught in another part of the program. Usually the information is just the name of the exception. You can either throw one of the many built in exception classes, or (usually better) create a new exception class specifically for the issue at hand.

Create the Exception. This is easy to do:

2.1. Right click on the `fraction` package, select new, Java class. Give it the name `ZeroDenomException`.

2.2 Add two words so it looks like this:

```
public class ZeroDenomException extends RuntimeException
```

This is basically all you have to do to create an exception! Everything is done by inheritance. Usually the only thing that matters is the name of the exception.

The Fraction c-tor will throw a `ZeroDenomException` if an attempt to create a fraction with a zero denominator happens.

2.3. The standard exception has two constructors -- one takes zero arguments, and one takes a `String`.

Thus when you throw an exception, you can put a string of additional information into the exception, which is sometimes a good idea.

2.4. This is **pretty easy!** It is done. All of the hard work is **inherited** from the `RuntimeException` class, which is written by experts, so we do not need to figure out how to do it. This is an example of the advantages of **inheritance**. Totally cool!

3. GCD method. In order to be sure the fraction is stored in lowest terms, we need to be able to compute the **greatest common divisor**. You might want to read up on this topic using Google. This function is not available in the standard Java libraries, so we will include a **helper method** to calculate it.

3.1. Put the following code In the `Utility` class,

```
public static long greatestCommonDivisor(long a, long b){
     a = Math.abs(a);
     b = Math.abs(b);
    long c;
    while(b != 0){
        c = a % b;
        a = b;
        b = c;
    }
    return a;
}
```

3.2. This cool method computes the **greatest common divisor** of two integers. This is needed to ensure that the fractions are always in lowest terms. It would be excellent practice to "walk through" the code, starting with something like `a = 300`, `b = 84`, and see how `c`, `a`, and `b` change each time through the loop.

Notice that this method **changes** the value of the input parameters (`a` and `b`), which is OK, because the method gets its own copy of them.

4. It is important to test this and see if it works correctly.

4.1. Go to the `Test` class and put the following code above the `main` method.

Right after this method, put in the following method:

```
public static void testGcd(){
    System.out.println("");
    System.out.println("Test Greatest Common Divisor");
    System.out.println("");
    for (long i = -21; i < 31; i += 2) {
        for (long j = -17; j < 31; j += 3) {
            System.out.printf("\n(%5d,%5d) --> %5d", i, j,
                    greatestCommonDivisor(i, j));
        }
    }
        System.out.println("");
}
```

This will cause the line *`greatestCommonDivisor`* to turn red. Click on it. It will say to fix it by ALT+ENTER. Do that, and notice an import statement added at the top.

This test has a double loop, trying out the new utility method on many combinations of numbers.

4.2. Inside the `main` method, between two lines already there, put in

```
    testGcd();
```

to call this test method. Run the program and examine the output. In most cases, the numbers are **relatively prime**, so the GCD is 1. When the GCD is not 1, check to be sure that the GCD is the largest number that divides both of `a` and `b`. You can adjust the limits of the two loops if you want to do some additional testing. When you are satisfied that this method works correctly, "comment out" the call to the test, like this:

```
    // testGcd();
```

Run the program again, and see that the (rather long and boring) test is not done. The reason for "commenting out" rather than erasing, is that it makes it easy to run the test later if needed.

Notice that there are a lot of tests. In most of them, the GCD is 1 because there are no common divisors. In this case they are said to be **relatively prime**. For a few cases, such as 21 and 14, they are not relatively prime, so the **common divisor**, namely 7, should appear. If this does not work correctly, figure out why and fix it!

5. Add the data fields to the Fraction class.

5.1. Edit the `Fraction` class. At the top, add the following comments:

```
    // the numer and denom fields represent a fraction
    // CLASS INVARIANTS:
    // CI1: denom is not 0,
    // CI2: denom is positive,
    // CI3: numer and denom are in lowest terms.
```

Class invariants are a powerful concept. The idea is that these three conditions are like a **contract** with users of the class. We (the class designer) must always be sure they are true, and everyone can count on this fact without worrying about it. Usually you set up the constructor to enforce the class invariants. It is a good idea to specifically state any class invariants with comments, as above. It might be good to also put similar comments at the top of the file (above the class) as a **javadoc comment**.

5.2. To enable our class to work with fractions as large as possible, we will use the `long` data type instead of the `int` data type. This costs us very little (more storage is needed), and the program runs just as fast as with `int`s.

After the above comments put in the two fields like this:

```
    private final long numer;
    private final long denom;
```

This will cause errors (red lines under the code). BTW, notice there are also red marks on the class menu on the right. Everything is red "all the way to the top". We will fix this by adding a c-tor.

6. Add constructors.

All of the hard work will be in the "full service constructor".

6.1. Add the following:

```
    // full service c-tor for fractions
    public Fraction(long numer, long denom) {
        // CI1. Cannot fix -- must throw exception.
        if (denom == 0) {
            throw new ZeroDenomException();
        }

        // CI2. Can fix.
        if (denom < 0){
            numer *= -1;
            denom *= -1;
        }
        // CI3. Can fix.
        long gcd = greatestCommonDivisor(numer, denom);
        if (gcd != 1){
            numer /= gcd;
            denom /= gcd;
        }

        // all class invariants now satisfied, initialize fields:
        this.numer = numer;
        this.denom = denom;
    }
```

You will have to fix an import.

6.2. This constructor is set up to ensure the fraction is valid by enforcing the class invariants, as follows:

6.2.1. The first `if` statement checks for a zero denominator. If there is one, there is no way to fix this, so an **exception is thrown**. We created an exception for exactly for this situation.

6.2.2. The second `if` statement checks for a negative denominator. If there is one, this can be fixed by “multiplying up and down by -1”.

6.2.3. The next line gets the greatest common divisor. This is then used to divide the input parameters `up` and `down` to ensure the fraction is in lowest terms.

Now we are good to go.

6.3. Any other constructors should work by calling this “full service” constructor.

6.3.1. Add a **one argument constructor**. A fairly common situation is for a fraction to actually represent a whole number, such as 7 / 1. You can do it like this:

```
    // one arg c-tor for longs
    public Fraction(long number){
        this(number, 1);
    }
```

6.3.2. Also add a **zero argument (default) constructor** that gives the default fraction: `0 / 1`.

7. toString and toDouble methods.

7.1. Most classes have a `toString` method. Add the following after the constructors:

```
    // toString and toDouble
    public String toString() {
        return numer + " / " + denom;
    }
```

This will write a fraction in the standard fashion, something like: `22 / 7`.

7.2. For a fraction class, it is also a good idea to have a `toDouble` method, in case you want to translate a fraction into a `double`.

```
    public double toDouble(){
        return 1.0 * numer / denom;
    }
```

Observe that the division will **not** be integer division, because of the `1.0 *` in front.

8. The test method. As soon as you add any method to the class, you want to test it. We will put a bunch of tests into one method. Edit the `Tests` class. Put in this method:

```
    private static void testCTors(){
        System.out.println("Fraction tests.");
        System.out.println("");
    }
```

Also put

```
    testCTors();
```

In the `main` method, right after `testGcd()`, to run the tests. Run the program and fix if needed.

9. Test the constructors. Add the following code to the `testCTors` method. These test cases test all the things that are likely to happen, other than trying to put zero in the denominator.

```
        System.out.println("Test C-tor");
        System.out.println("Expected outcome: 4/25, -4/25, -4/25, "
            + "4/25, 17/1, 0/1.");
        System.out.println("");
        Fraction rat01 = new Fraction(144, 900);
        Fraction rat02 = new Fraction(-144, 900);
        Fraction rat03 = new Fraction(144, -900);
        Fraction rat04 = new Fraction(-144, -900);
        Fraction rat05 = new Fraction(17);
        Fraction rat06 = new Fraction();
        System.out.println("rat01 = " + rat01);
        System.out.println("rat02 = " + rat02);
        System.out.println("rat03 = " + rat03);
        System.out.println("rat04 = " + rat04);
        System.out.println("rat05 = " + rat05);
        System.out.println("rat06 = " + rat06);
```

Run this and see if the constructors are working correctly. If not, figure out why and fix it.

9.1. It is important to also test what happens with bad input. We will test what happens if a denominator is zero. This should cause an exception to be thrown, which will be slightly ugly. To avoid this, we must put the test inside a **try block** so that the program does not immediately halt. The following is an example of how to use a try block.

Add the following:

```
        System.out.println("");
        System.out.println("Try bad input");
        try {
            Fraction rat00 = new Fraction(0, 0);
            System.out.println("Bad test for Zero Denominator
Exception");
        } catch (ZeroDenomException zde) {
            System.out.println("Expected Zero Denominator
Exception: " + zde);
        } catch (Exception e) {
            System.out.println("This should never happen.");
        }
        System.out.println("End of exception test.");
```

Run and fix if needed. Here we first try to catch a `ZeroDenomException`, which is what is thrown by the constructor. After that, we catch **any** other exception that might be thrown. We do not expect this to ever happen, so we print out a message that something is awry.

9.2. Even though the `toDouble` method is totally simple, test it anyway! Put the following code at the end of the `testCTors` method:

```
        System.out.println("");
        System.out.println("Test toDouble. Expect 0.16");
        System.out.println("rat04 to double: " + rat04.toDouble());
```

Run the tests again.

10. Math methods. Now we will add a few math operations to the class.

10.1. Add the code for this method at the end of the `Fraction` class:

```
    // adds the rhs object to the calling object
    // using a/b + c/d = (ad + bc) / bd
    public Fraction plus(Fraction rhs){
        long up = numer * rhs.denom + rhs.numer * denom;
        long dn = denom * rhs.denom;
        return new Fraction(up, dn);
    }
```

Note that you do not have to worry about being in lowest terms or anything, because the c-tor takes care of that. This is a key idea: let the c-tor do all the hard thinking about creating a valid object.

10.2. Now return to the `Tests` class, and add this method (right above the `main` method):

```
public static void testMath() {
    Fraction threeHalves = new Fraction(3, 2);
    Fraction oneQuarter = new Fraction(1,4);
    Fraction zero = new Fraction();
    Fraction five = new Fraction(5);

    System.out.println("\nMath Tests\n");

    System.out.println("\nTest plus:\n");
    System.out.println("should be 7/4  : " +
threeHalves.plus(oneQuarter));
    System.out.println("should be 7/4  : " +
oneQuarter.plus(threeHalves));
    System.out.println("should be 3/1  : " +
threeHalves.plus(threeHalves));
    System.out.println("should be 13/2 : " +
threeHalves.plus(five));
    System.out.println("should be 3/2  : " +
threeHalves.plus(zero));
}
```

Also put `testMath()` in the main method to call this test function. Run it and be sure all the tests are done correctly. Note the style of test makes it easy check!

10.3. Return to the Fraction class, and add a method with signature

```
public Fraction minus(Fraction rhs)
```

to do subtraction. Hint: It is almost exactly the same as the `plus` method, with one obvious change.

10.4. Now return to the `Tests` class, and add some more tests for this new method. Probably a good idea to copy and paste the existing five tests, then change `plus` to `minus`. You will also have to change the expected result in the “should be” part.

Run the program and make sure it works.

10.5. Return to the Fraction class, and add a method with signature

```
public Fraction times(Fraction rhs)
```

to do multiplication of fractions. Hint: this is easier than adding or subtracting.

10.6. Now return to the `Tests` class, and add some more tests for this new method. Probably the same five tests, except use times instead of plus or minus.

10.7. Return to the Fraction class, and add a method with signature

```
public Fraction dividedBy(Fraction rhs)
```

to do division of fractions. Hint: this is very similar to the `times` method.

10.8. Now return to the `Tests` class, and add some more tests for this new method. Probably the same five tests, except with `dividedBy`.

NOTE: if you have a `dividedBy(zero)`, you will get an exception thrown. Figure out how to put that in a try block. Hint: check how this is done in the next part.

11. The exam method.

This method gives your work a quick “exam”, and grades it on how many things are right. Put one more method in the `Tests` class, **exactly as follows**:

```
    public static void exam() {
        Fraction a = new Fraction(1284, 2889);
        Fraction b = new Fraction(385,462);
        Fraction c = new Fraction();
        System.out.println(" 1: " + a);
        System.out.println(" 2: " + b);
        System.out.println(" 3: " + b.plus(a));
        System.out.println(" 4: " + b.minus(c));
        System.out.println(" 5: " + b.times(a));
        System.out.println(" 6: " + a.dividedBy(b));
        System.out.println(" 7: " + a.dividedBy(a));
        System.out.println(" 8: " + a.times(a));
        System.out.println(" 9: " + a.times(c));
        try {
            System.out.print("10: " );
            System.out.println(a.dividedBy(c));
        } catch (ZeroDenomException zde) {
            System.out.println("ZeroDenomException");
        }
    }
```

Also add the line

```
exam();
```

to the main method. Be sure the exam method runs and is printed out. (be sure the answers are correct!)

This will be part of the grade!

12. That's about it!

How to turn in Project. This is slightly complicated, so follow these instructions for submitting exactly:

1. Open file explorer
2. Find your project (likely C:\Users\<user>\IdeaProjects\Fraction)
3. Right click on your project folder (this should contain src/fraction with all your .java files)
4. Select **Send to > Compressed (zipped) folder**
5. Submit Fraction.zip to Blackboard

You can submit as often as you want, each one replaces the previous one, but has a later time stamp. A day or so after the due date, the projects will be graded, and any further submissions will be ignored.

Grading:

40 points total.

1. 20 points for the entire programming running. Points will be deducted for any style errors. You should not have any of these, because Intellij IDEA will try to make you do it right.

2. 20 points for the output produced by the `exam` method, 2 points for each line of output.