Spring 2019: CS 203 - Object-Oriented Programming

Assignment 1

Fractions ***due on September 15***

Objectives:

- To practice types, objects, classes, main in JAVA
- Creating and using test cases

The Float and Double classes (and the respective primitive types) offer a good representation of real values suitable for many numeric problems. However, the finite precision of those types can sometimes lead to surprising results (try to square 0.1 as `float` or `double`)[1]. In some cases, such problems can be remedied by using a `Fraction` class.

Implement a class Fraction that models fractions, where numerator and denominator are represented as `int` (or `Integer`) and offers arithmetic operations. A Fraction object can be created with a constructor that takes a numerator and a denominator as arguments. The Fraction class offers the public methods defined below. Note, the results of arithmetic operations should be simplifyed. e.g., multiplying two fractions $\frac{4}{3}\ \frac{3}{5}$ should return $\frac{4}{5}$ (not $\frac{12}{15}$)[2].

```
/**
* Fraction is a class modeling fractions where numerator
* and denominator are represented as integral number.
**/

public class Fraction {

/**
*Constructs a new Fraction object.
*@param init initial value*/

     public Fraction(/* SOMETYPE */ num, /*SOMETYPE */ denom) {
     /*YOUR CODE */ }
     /**
     * Computes "this+that" and returns the result in a new object.
     * @param that the left-hand side operand.
     * @return a new object representing this+that*/


     public /* SOMETYPE */ add(/* SOMETYPE */ that) {
     /* YOUR CODE */ }
     /**
     * Computes "this+that" and returns the result in a new object.
     * @param that the left-hand side operand.
     * @return a new object representing this+that
     */
```

```
    public /** SOMETYPE */ sub(/* SOMETYPE */ that) { /* YOUR CODE */ }
    /**
    * Computes "this_that" and returns the result in a new object.
    * @param that the left-hand side operand.
    * @return a new object representing this+that
    */


    public /* SOMETYPE */ mul(/* SOMETYPE */ that) { /* YOUR CODE */ }
    /**
    * Computes "this/that" and returns the result in a new object.
    *@param that the left-hand side operand.
    * @return a new object representing this/that
    */


    public /* SOMETYPE */ div(/* SOMETYPE */ that) { /* YOUR CODE */ }
    /**
    * Computes the quotient of the fraction.
    * @return a new object representing the quotient of this/that
    */


    public /* SOMETYPE */ getQuotient() { /* YOUR CODE */ }
    /**
    * Computes the remainder of the fraction.
    * @return a new object representing the remainder of this/that
    */


    public /* SOMETYPE */ getRemainder() { /* YOUR CODE */ }
    /**
    * Computes the oating point value of the fraction.
    * @result a oating point number that approximates this fraction
    */


    public /* SOMETYPE */ doubleValue() { /* YOUR CODE */ }
    /**
    *Returns the textual representation of the fraction in the form
    * of "num / denom". If denom equals 1, only num will be
    * returned. If denom equals 0, the string "in_nity" should be returned.
    *@result a string representing this fraction
    */

    public /* SOMETYPE */ toString() { /* YOUR CODE */ }


/*YOUR CODE - instance variables */
}
```

- Choose appropriate types for the methods' arguments and return values.
- Add instance variables to represent numerator and denominator.
- Implement the methods.
- Implement a tester class that exercises the methods. Do not only test behavior that is well defined. e.g., what does happen, if you create a fraction 1/0?

Turn in a zip file named **blazerid_hw1.zip**. The file should contain an exported Eclipse project[3] with the following items.

- All files needed to compile and run your solution.
- Your tests.
- A document (or text file) that describes your design decisions, your tests, any difficulties you had.

Grading (10 pts max)
_ (10pts) Lab attendance (Lab 2)
_ (20pts) Assignment report (see paragraph above)
_ (20pts) Turned in Fraction and Tester class compile
_ (10pts) Test design and quality + Code quality (e.g.,javadoc)
_ (20pts) Constructor and arithmetic operations work correctly.
_ (20pts) getQuotient, getRemainder, doubleValue, and toString work correctly.

---

[1]A recent article on issues with floating point arithmetic is available here:
https://cacm.acm.org/magazines/2017/8/219594-small-data-computing/fulltext

[2]https://en.wikipedia.org/wiki/Greatest_common_divisor

[3]If you do not use Eclipse, the turned in _le also needs to have a _le readme.txt in its top directory that explains how to compile and test your code.