Homework 8 : Fractions

For HW8, you may work as a group (no more than 2 students). **Please mention your collaborator's name at the top of your code files.**

This homework deals with:

- Test Driven Development
- More experience with class-based object oriented programming
- Static methods (we'll learn about these in the next lecture)
- Fraction arithmetic

You will create a *Fraction* class in Java to represent fractions and to do fraction arithmetic.

To get you used to the idea of unit testing, this homework does not require a *main* method. You can create one if you find it useful, however, we will not be grading or even looking at that code. You should be comfortable enough with the accuracy of your test cases that you do not need to use print statements or a *main* method to even do ad hoc "testing" of your *Fraction* class.

Steps to Follow for this Homework

1. Read through this entire document **without opening Eclipse**
2. In Eclipse, create a new project called *HW8_CIT590*, then create a package in the src folder called "fraction". Create a new class in the "fraction" package named *Fraction.java*, and fill it with empty stubs of the methods you will write **later**. See the *Writing Method Stubs* instructions on page 5.
3. Import the provided *SampleFractionTest.java* into your Java project and run the tests -- all tests should fail (for now)!
4. Implement enough code in the methods in *Fraction.java* to pass all tests.
5. Use pen and paper (or an electronic document) to **generate a list of additional test cases** you will use for each method in the *Fraction* class. You should have a total of **at least 4 distinct and valid test scenarios (with individual test cases)** per method (including the ones provided).
6. Create a second testing class file named *FractionTest.java* and write all of the additional test cases you generated in step 5. See the *Writing Unit Tests* instructions on page 7.
7. Update your implementation in the methods in *Fraction.java* to pass all additional tests.
    a. If something fails:
        i. Make sure all test cases are valid
        ii. Fix any mistakes in your implementation
        iii. Re-run the test file
        iv. Repeat these steps until all tests pass
8. Re-run both test files to make sure all tests pass

The *Fraction* Class

A fraction is a number of the form numerator/denominator where the *numerator* and *denominator* are integers. The denominator cannot be 0. You may assume that no user will input a denominator of 0.

The *Fraction* class needs to have two instance variables: *numerator* and *denominator*. The methods in this class are below. They have been provided with their method signatures.

public Fraction(int numerator, int denominator)

- The constructor to create a Fraction with the given numerator and denominator.
- The constructor should set the numerator and denominator instance variables in the *Fraction* class.
- The constructor should also properly format negative fractions. The convention is that negative fractions have the negative in the numerator.
- For example:
    - Creating a new Fraction(4, 16) would set the numerator to 4 and the denominator to 16
    - Creating a new Fraction(4, -16) would set the numerator to -4 and the denominator to 16
    - Creating a new Fraction(-1, -2) would set the numerator to 1 and the denominator to 2

public void reduceToLowestForm()

- Reduce the current fraction by eliminating common factors.
- That is, turn a fraction like 4/16 into 1/4 and a fraction like 320/240 into 4/3.
- Remember, the convention is that negative fractions have the negative in the numerator.
- For example:
    - A fraction like 4/16 would reduce to 1/4
    - A fraction like 10/-15 would reduce to -2/3
    - The reduced form of any fraction that represents 0 is 0/1
        - e.g. 0/4 reduces to 0/1

public Fraction add(Fraction otherFraction)

- Add the current fraction to the given otherFraction.
- Returns a new Fraction that is the sum of the two Fractions.
- The returned Fraction must be in reduced/lowest form.
- For example:
    - Adding the fraction 3/5 to the fraction 1/4 reduces to 17/20
    - Adding the fraction -1/2 to the fraction 2/-3 reduces to -7/6

public Fraction subtract(Fraction otherFraction)

- Subtract the given otherFraction from the current fraction.
- That is, thisFraction - otherFraction.
- Returns a new Fraction that is the difference of the two Fractions.
- The returned Fraction must be in reduced/lowest form.
- For example:
    - Subtracting the fraction 3/9 from the fraction 5/9 reduces to 2/9

  - Subtracting the fraction 5/16 from the fraction 4/16 reduces to -1/16

public Fraction mul(Fraction otherFraction)

- Multiply the current fraction by the given otherFraction.
- Returns a new Fraction that is the product of this fraction and the otherFraction.
- The returned Fraction must be in reduced/lowest form.
- For example:
  - Multiplying the fraction 1/2 by the fraction 2/3 reduces to 1/3

public Fraction div(Fraction otherFraction)

- Divide the current fraction by the given otherFraction.
- That is, thisFraction / otherFraction.
- Returns a new Fraction that is the quotient of this fraction and the otherFraction.
- The returned Fraction must be in reduced/lowest form.
- For example:
  - Dividing the fraction 4/16 by the fraction 5/16 reduces to 4/5

public double decimal()

- Return this fraction in decimal form.
- For example:
  - For the fraction 2/4, this method should return the value 0.5
  - For the fraction 1/3, this method should return the approximate value 0.3333333333333333
    - Note, to unit test double values like this, use assertEquals with a delta (see the lecture slides on *Comparing Floating Point Types*)

public void sqr()

- Square the current fraction.
- This method modifies the current fraction and reduces it to lowest form.
- For example:
  - A fraction like 2/3 will become 4/9
  - A fraction like 4/16 will become 1/16

public Fraction average(Fraction otherFraction)

- Average the current fraction with the given otherFraction.
- Return a new Fraction that is the average of this fraction and the otherFraction.
- The returned Fraction must be in reduced/lowest form.
- For example:
  - Averaging the fraction 5/8 with the fraction -12/16 reduces to -1/16

public static Fraction average(Fraction[] fractions)

- *Static* method to average all of the fractions in the given array.
  - Note, you don't need to create an instance of the Fraction class in order to call a *static* method

   - For example, you should be able to call this method with the class name (note upper-case in "Fraction")

```
Fraction f = Fraction.average(myArrayOfFractions);
```

- Do not include the current fraction in the average.
- Return the average of the array.
- The returned Fraction must be in reduced/lowest form.
- If the array is empty, return a new Fraction that equals 0. That is 0/1.
- For example:
   - The average of the fractions 3/4, 3/5, and 3/6 reduces to 37/60

public static Fraction average(int[] ints)

- *Static* method to average all the integers in the given array.
   - Again, you don't need to create an instance of the Fraction class in order to call a *static* method
   - For example, you should be able to call this method with the class name (note upper-case in "Fraction")

```
Fraction f = Fraction.average(myArrayOfInts);
```

- Do not include the current fraction in the average.
- Return the average of the array as a new Fraction.
- The returned Fraction must be in reduced/lowest form.
- If the array is empty, return a new Fraction that equals 0. That is 0/1.
- For example:
   - The average of the ints 1, 2, 3, and 4 reduces to 5/2

@Override
public boolean equals(Object object)

- Overriden method to compare the given object (as a fraction) to the current fraction, for equality. (See the lecture slides on *Testing for Equality*)
- Two fractions are considered equal if they have the same numerator and same denominator, after eliminating common factors.
- This method does not (permanently) reduce the current fraction to lowest form.
- For example:
   - The fraction 2/3 is equal to the fraction 2/3
   - The fraction 4/16 is equal to the fraction 1/4, but the fraction 4/16 is not reduced to lowest form.

@Override
public String toString()

- Overriden method to return a string representation of the current fraction.
- A fraction like 2/3 will be represented in string form as "2/3".
- There is a no whitespace in this string.

- If the fraction is negative, it will be expressed as “-2/3”, not “2/-3”.

Tip: You are always encouraged to write additional helper methods! We *highly recommend* that if you write helper methods, you test them.

Tip: You should make sure you know how fraction operations work before you write test cases.

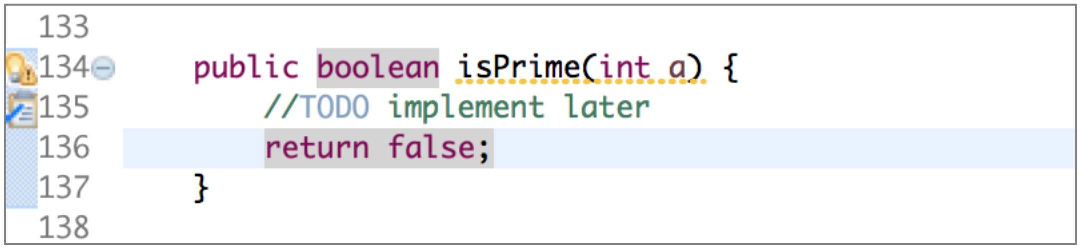
- How Do I Do Basic Operations With Fractions?: https://www.youtube.com/watch?v=i_E8XZm1p_0
- Fraction calculator: https://www.hackmath.net/en/calculator/fraction

Writing Method Stubs

In Eclipse, create a new project called *HW8_CIT590*. Create a package in the src folder called “fraction”. Create a new class in the “fraction” package named *Fraction.java*.

In order to write and run test cases, the class and the methods we are testing must be declared so that Eclipse knows what we are referencing in the unit tests. Therefore, we are going to write an empty shell of a class with each method **declared, but not implemented**.

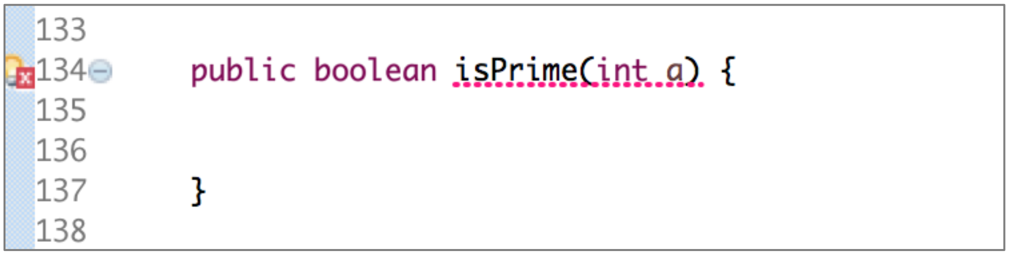
For each method listed in **The *Fraction* Class** section, write the method declaration and, if needed, return some arbitrary value. For example, if you needed the method `boolean isPrime(int a)`, you would write the following code as a stub to be implemented later:



```
    public boolean isPrime(int a) {
        //TODO implement later
        return false;
    }

```

If we don’t include the statement `return false;` Eclipse gives you a warning about a compilation error:

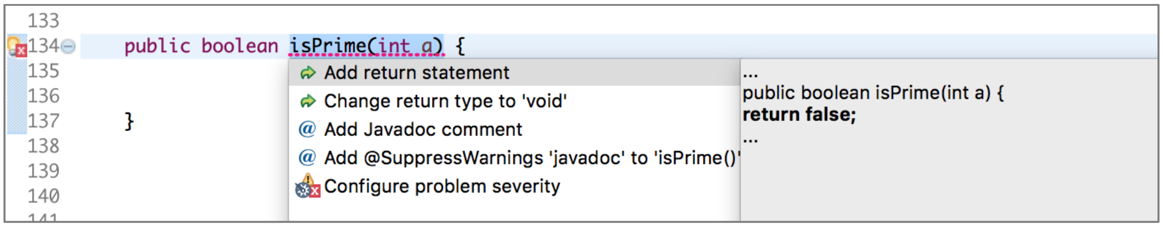


```
    public boolean isPrime(int a) {


    }

```

You can click the white x (in red) on the left, and click on one of the suggested fixes. In this case “Add return statement” will work just fine:



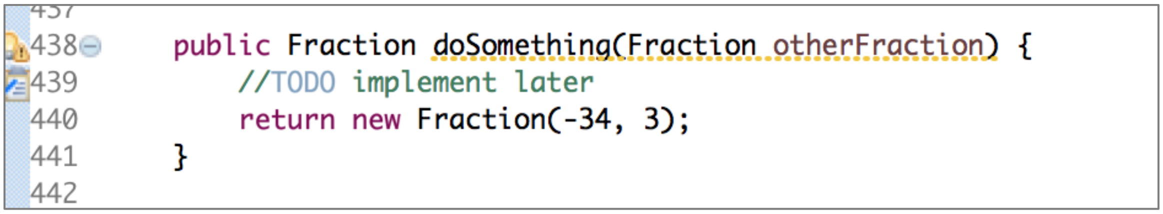
```
    public boolean isPrime(int a) {

    }
```

Add return statement
Change return type to 'void'
Add Javadoc comment
Add @SuppressWarnings 'javadoc' to 'isPrime()'
Configure problem severity

```
...
public boolean isPrime(int a) {
return false;
...
```

For methods that return a Fraction object, you'll need to return a new Fraction with some arbitrary numerator and denominator. This will require that you have already **declared** the Fraction constructor in this class. (Note: the constructor SHOULD NOT be implemented at this point!)

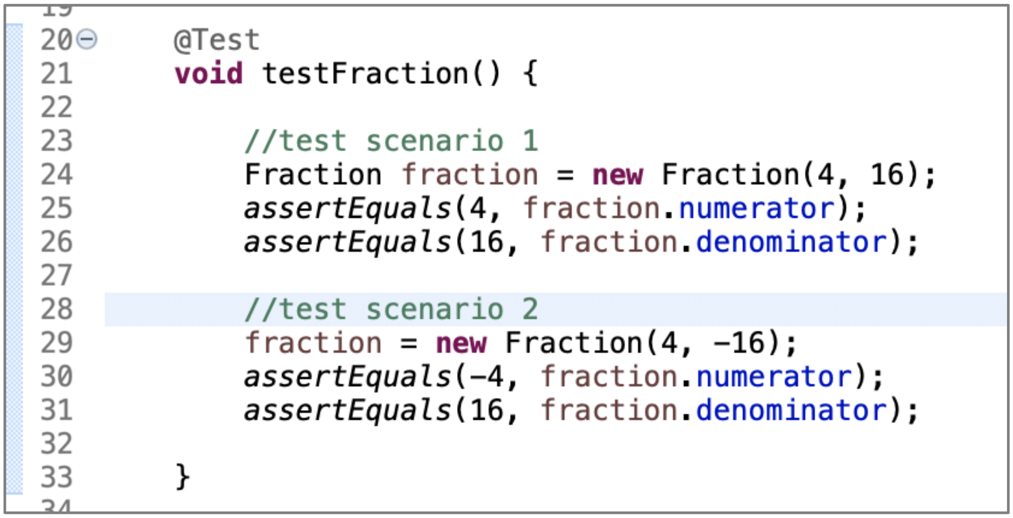


```
    public Fraction doSomething(Fraction otherFraction) {
        //TODO implement later
        return new Fraction(-34, 3);
    }
```

Please note that it really doesn't matter what the values (numerator and denominator) are. Here, I used -34 and 3. Generally, I avoid using edge case values like 0, 1, and -1 because they can give false positives when testing edge cases.

Writing Unit Tests

Including the test cases provided, you must have a total of **at least 4 different and valid test scenarios (with individual test cases) for each** of the Fraction methods. For example, in the *SampleFractionTest.java* file we have provided, there is a "testFraction" method. It contains 2 unique and valid test scenarios (with individual test cases).



```
    @Test
    void testFraction() {

        //test scenario 1
        Fraction fraction = new Fraction(4, 16);
        assertEquals(4, fraction.numerator);
        assertEquals(16, fraction.denominator);

        //test scenario 2
        fraction = new Fraction(4, -16);
        assertEquals(-4, fraction.numerator);
        assertEquals(16, fraction.denominator);

    }
```

Your test cases must be distinct. For example, 1/7 + 3/7 = 4/7 is not mathematically distinct from 2/15 + 6/15 = 7/15.

Your test cases must be valid. In other words, the expected outcome must actually be mathematically correct. For example, 9/2 - 5/2 = -7/9 is not valid, but -2/3 + 7/3 = 5/3 is valid.

So, for each method outlined, you should have at least 4 unique scenarios where that method might be used. It is typically a good idea to start with a general case and then add a few edge cases.

For example, suppose a method exists that is expected to calculate the sum of the elements of an array:

	Input	*Expected Output*
General/Typical Case	[3, 8, -2]	9
Edge Case (one element)	[4]	4
Edge Case (empty array)	[]	0
Less Common Case	[-5, -5, -5, -5, -5, -5, -5, -5]	-40

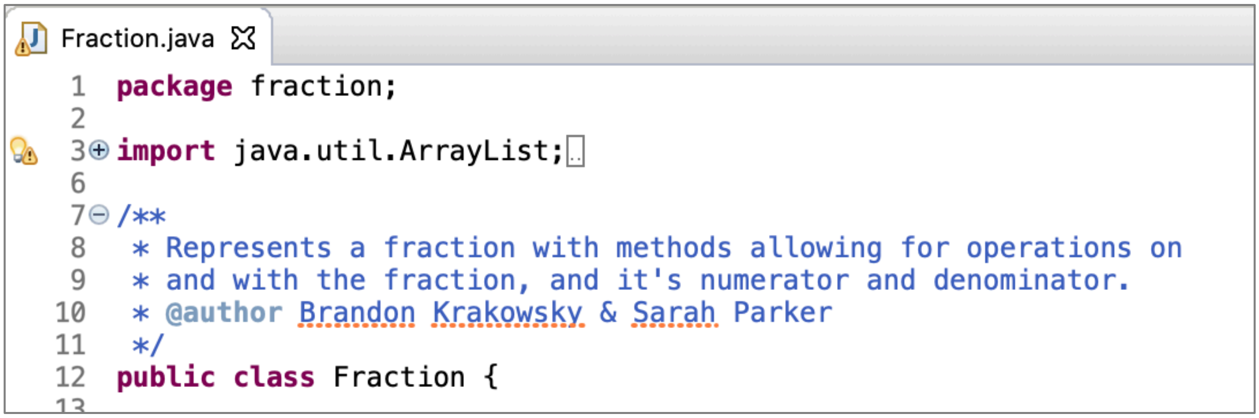
Eclipse Instructions

1. Create a new test class by right-clicking on the *Fraction* class and choosing “New” -> “JUnit Test Case”
2. Select “New JUnit Jupiter test” at the top
3. Name the new test class *FractionTest*
4. Use the checkboxes to specify the methods in your *Fraction* class for testing
5. Code up all your additional test cases. Follow the procedures demonstrated in lecture.

What to Submit

Please submit all your Java classes to Canvas. Make sure it is all put in the “src” folder. Included should be 3 files: your *Fraction.java*, your *FractionTest.java*, and our *SampleFractionTest.java*. Do not remove the package declaration in each file and keep the entire program in the “fraction” subdirectory.

If you’re working as part of a team, only one student from your team needs to submit the files. Include the members of your team as part of the @author tag in the Javadocs for the Fraction class. If Brandon was working with Sarah, their Javadocs and @author tag would look something like:



```
package fraction;

import java.util.ArrayList;

/**
 * Represents a fraction with methods allowing for operations on
 * and with the fraction, and it's numerator and denominator.
 * @author Brandon Krakowsky & Sarah Parker
 */
public class Fraction {
```

Evaluation

1. Does your code function? Does it do what the specifications require? **(**AUTOGRADED**)** (12 pts)
    a. Did you implement the methods in the *Fraction* class exactly as they have been defined in this document?
    b. We will run our own unit tests in addition to the ones provided and the ones you write.
2. Did you have a total of at least 4 distinct, valid test scenarios (with individual test cases) for each test method? (10 pts)
    a. Did you pass your own test cases?
    b. Did you include general test case scenarios as well as edge cases?
3. Did you follow good programming practices? (5 pts)
    a. Did you reuse code to avoid repetition (e.g. put repeated code in a helper method)?
    b. Did you name additional variables and methods descriptively with camelCase?
    c. Do you use “this.” when referencing instance variables and methods of the class?
    d. Did you add javadocs to classes, instance variables, and methods, and comments to all non-trivial code?
4. Did you set up the files correctly? Does it compile and is everything named correctly? (3 pts)