CIT 590 Assignment 11: Fraction API

Fall 2016, David Matuszek

Purpose of this assignment

- To give you practice with creating classes.
- To give you practice reading the API.
- To give you more practice with string manipulation.
- To give you practice creating Javadoc files.

General idea of the assignment

Implement a `Fraction` API. Write very thorough JUnit tests. Write a small "calculator" program to do arithmetic with fractions.

Create a project named `Fraction`, and within that, a package named `fraction`. Your classes, `Fraction`, `FractionTest`, and `FractionCalculator`, will all be within this package.

Declare your class as `public class Fraction implements Comparable {...}`. (The reason for the "implements" part will be explained in class; for now, just do it.)

Note: The `Fraction` class has a lot of methods, but they are all pretty short. The `FractionCalculator` class has to read user input and make sense of it, so it's actually the more difficult part of this assignment.

The API

Java provides several types of numbers, but it does not provide fractions. Your task is to implement a `Fraction` API (Application Programmer's Interface) and write a small program that uses it.

The following table lists some information about your new `Fraction` type.

Instance variables	Two `private` integers, known as the *numerator* and the *denominator*. Do not provide getters or setters for these instance variables; there is no reason for anything outside the `Fraction` class to know about them. **Note:** Even though these instance variables are `private`, they are private *to the class*, not to the object. This means: Any `Fraction` object can access the private variables of any other `Fraction` object; it's only outside this class that you cannot access them.

How written	***n/d***, where ***n*** is the numerator and ***d*** is the denominator.
Restrictions	The denominator may not be zero.
Normalization	The fraction is ***always*** kept in the lowest terms, that is, the Greatest Common Divisor (GCD) of `n` and `d` is `1` (use Euclid's algorithm). The denominator is never negative. (You can give a negative number for the denominator to the constructor when you create the fraction, but a negative fraction should be represented internally with a negative *numerator*.) Zero should be represented as `0/1.`

The following lists the consructors you should have.

Constructor	How it uses its parameters
`public Fraction(int numerator, int denominator)`	Parameters are the *numerator* and the *denominator*. **Normalize** the fraction as you create it. For instance, if the parameters are `(8, -12)`, create a `Fraction` with numerator `-2` and denominator `3`. The constructor should throw an `ArithmeticException` if the denominator is zero.
`public Fraction(int wholeNumber)`	The parameter is the numerator and `1` is the implicit denominator.
`public Fraction(String fraction)`	The parameter is a `String` containing either a whole number, such as `"5"` or `" -3"`, or a fraction, such as `"8/ -12"`. Allow blanks around (but not within) integers. The constructor should throw an `ArithmeticException` if given a string representing a fraction whose denominator is zero. You may find it helpful to look at the available String methods in the Java API.

Notes:

- Java allows you to have more than one constructor, so long as they have different numbers or types of parameters. For example, if you call `new Fraction(3)` you will get the second constructor listed above. A String is a String, though, so the third constructor will have to distinguish between `"3"` and `"3/4"`.
- To throw an Exception, write a statement such as: `throw new ArithmeticException("Divide by zero");` (The String parameter is optional.)
- The java method `Integer(`*`string`*`).parseInt()` will return the `int` equivalent of the *`string`* (assuming that the *`string`* represents a legal integer). Malformed input will cause it to

throw a **NumberFormatException**.

Method signature	What it does
`public Fraction add(Fraction f)`	Returns a new **Fraction** that is the **sum** of **this** and **that**: ***a/b + c/d*** is **(ad + bc)/bd**
`public Fraction subtract(Fraction f)`	Returns a new **Fraction** that is the **difference** of **this** minus **that**: ***a/b - c/d*** is **(ad - bc)/bd**
`public Fraction multiply(Fraction f)`	Returns a new **Fraction** that is the **product** of **this** and **that**: **(a/b) * (c/d)** is **(a*c)/(b*d)**
`public Fraction divide(Fraction f)`	Returns a new **Fraction** that is the **quotient** of dividing **this** by **that**: **(a/b) / (c/d)** is **(a*d)/(b*c)**
`public Fraction abs()`	Returns a new **Fraction** that is the absolute value of **this** fraction.
`public Fraction negate()`	Returns a new **Fraction** that has the same numeric value of **this** fraction, but the opposite sign.
`public Fraction inverse()`	The inverse of ***a/b*** is ***b/a.***
`@Override public boolean equals(Object o)`	Returns **true** if **o** is a **Fraction** equal to **this**, and **false** in all other cases. **You need this method for your assertEquals(*expected*, *actual*) JUnit tests to work!** The **assertEquals** method calls *your* equals method to do its testing.
`@Override public int compareTo(Object o)`	If o is a **Fraction** or an **int**, this method returns: A negative **int** if **this** is less than **o**. Zero if **this** is equal to **o**. A positive **int** if **this** is greater than **o**. If **o** is neither a **Fraction** nor an **int**, this method throws a **ClassCastException**. This method is not used by your calculator, but is included for completeness.
	Returns a **String** of the form ***n/d***, where ***n*** is the numerator and ***d*** is the denominator.

`@Override public String toString()`	However, if *d* is `1`, just return *n* (as a `String`). The returned `String` should not contain any blanks. If the fraction represents a negative number, a minus sign should precede the *n*. **This should be one of the first methods you write, because it will help you in debugging.**

Notes:

- All `Fraction`s should be *immutable*, that is, there should be no way to change their components after the numbers have been created. Most of your methods simply return a *new* number.
- When you define `equals`, notice that it requires an `Object` as a parameter. This means that the first thing the method should do is make sure that its parameter is in fact a `Fraction`, and return `false` if it is not.
    - You can test with `o instanceof Fraction`.
    - You can use `o` as a `Fraction` by saying `((Fraction)o)`, or you can save it in a `Fraction` variable by saying `Fraction f =(Fraction)o;` and then using `f`.
- You can create additional "helper" methods (for example, to compute the GCD, or to normalize a fraction), if you wish. I recommend doing so. If you do, **do not** make these methods `public`, but **do** test them (if you can figure out how).

To put a fraction into its lowest terms, divide both the numerator and the denominator by their Greatest Common Divisor (GCD). **Euclid's algorithm** finds the GCD of two integers. It works as follows: As long as the two numbers are not equal, replace the larger number with the remainder of dividing the larger by the smaller (that is, `larger = larger % smaller`). When the two numbers are equal, that value is the GCD. (If this brief explanation isn't enough, look up *Euclid's algorithm* on the Web.)

JUnit testing

Your goal in writing the JUnit class is to test for ***every*** possible error. This includes making sure that the correct `Exception`s are thrown when appropriate.

Most of your tests will have this form:

```
@Test
public void testAdd() {
    // Tests that are expected to succeed
}
```

To test for an exception, use this form:

```
@Test(expected=ArithmeticException.class)
```

```
public void testDivideByZero() {
    // test that should throw an ArithmeticException
}
```

You can find the various kinds of assertions you can make in the JUnit API.

The FractionCalculator

Write a `FractionCalculator` class (containing a `main` method) that does calculations with Fractions. The first thing this program does is print a zero (indicating the current contents of the calculator), and a prompt. It then accepts commands from the user, and after each command, prints the result, and a new prompt. It should accept *exactly the following commands*, and nothing else:

- `a` To take the absolute value of the number currently displayed.
- `c` To clear (reset to zero) the calculator.
- `i` To invert the number currently displayed.
- `s` *n* To discard the number currently displayed and replace it with *n*.
- `q` To quit the program.
- `+` *n* To add *n* to the number currently displayed.
- `-` *n* To subtract *n* from the number currently displayed.
- `*` *n* To multiply the number currently displayed by *n*.
- `/` *n* To divide the number currently displayed by *n*.

In each case, the user may enter *either* a whole number *or* a fraction for *n*.

- Fractions may be written with or without spaces, as for example `27 / 99` or `27/99`.
- You can require that there be no space after a unary minus, so for example `-3` is legal, but `- 3` is not.
- You don't have to handle unary `+`.
- You can require at least one space after the initial `+`, `-`, `*`, `/`, or `s`, so for example `- -3/5` is legal, but `--3/5` is not.

No user input, however illegal, should crash the program. Instead, the program should print a short error message. Illegal input should **not** affect the state of the computation.

As this class does little computation on its own (it's all done in the `Fraction` class), and is mostly concerned with input/output, no unit testing is necessary.

You will need to use the `Scanner`. You can find a description (skip over the confusing details) in the Scanner page in the Java API.

Recommendation: Just use the Scanner's **`readLine()`** method, and use String methods to work with the input. Some useful methods are **`trim()`**, **`length()`**, and **`substring(beginIndex)`**. I found this somewhat easier that working with the Scanner methods to read the input in smaller pieces.

Comments

All methods in your **`Fraction`** and **`FractionCalculator`** class should have Javadoc comments. The methods in your test class do not need to be commented, unless they are particularly complex or non-obvious.

Eclipse can help you by writing the Javadoc "outline" for you, which you can fill in. For example, suppose you have written the method stub (you *are* using TDD, aren't you?):

```
public Fraction add(Fraction f) {
    return null;
}
```

Place your cursor in the method and choose `Source -> Generate Block Comment` (or hit `Ctrl-Alt-J`). Eclipse creates the comment

```
/**
 * @param f
 * @return
 */
```

which you can then fill in, as for example:

```
/**
 * Returns the sum of this Fraction and the Fraction f.

 * @param f The Fraction to be added to this one.
 * @return The sum of the two Fractions.
 */
```

Javadoc

Create Javadoc comments for all your methods (except your test methods). Eclipse will create the skeleton of a method or class comment if you position the cursor on the method/class name and choose **`Source → Generate Element Comment`**, or click **`ctrl-alt-J`**. You have to fill in the details, both the general description and after each **`@`** tag.

After writing the comments, use Eclipse to generate the actual Javadoc files, with **`Project → Generate Javadoc...`**. Step through the dialog boxes carefully, and generate Javadoc for **`Fraction`** and **`FractionCalculator`**, but not **`FractionTest`**.

Finally, *look at* the Javadoc files, and make sure they are complete and correct.

Grading

Your grade will be based on:

- How correct and complete your number class is.
- How complete and correct your JUnit tests are.
- How complete your Javadoc is (and whether you remembered to generate it!)
- How well your "calculator" works.
- Your comments and general programming style.

We will use our own programs and our own unit tests for grading purposes--therefore, you must have method names and parameter types ***exactly*** as shown.

Due date

Zip your project and turn it in to Canvas before **11:59pm Tuesday, November 22**.