Software and Programming II, 2016-17
Coursework One

For submission details please see the Moodle site.

1 Aims of this Coursework

- To give you some experience with writing a class, both for the internal data representation and the public methods.
- To practice using classes and methods from the Java API with the help of the corresponding documentation.

2 Fractions

The goal is to write a class `Fraction` for representing a data type for *fractions* a/b, where a and b are integer values. Thus, these fractions correspond to the rational numbers. Examples are $12/5$ or $-1/2$.

In this assignment, we want to be able to represent fractions with *arbitrary* precision. Values like $1/12345678901234567890123456789$0 should also be faithfully represented, and no rounding should be used. Thus, for your implementation it is a good idea (but not mandatory) to use objects of type `BigInteger` from the package `java.math` as instance variables (attributes, fields). The official documentation for the class `BigInteger` is available here:

`https://docs.oracle.com/javase/8/docs/api/java/math/BigInteger.html`

We would like to be able to perform arithmetic with fractions, similar to the way we do it with values of type `int` or `double`. Thus, we would like to provide methods for the user of the class `Fraction` (e.g., `add`, `subtract`, `multiply`, `divide`, ...).

One peculiarity of fractions a/b is that normally their value is seen as undefined if $b = 0$. More generally, we are normally not allowed to divide by 0 for this reason. For the scope of this coursework, we thus define that for the type `Fraction`, $a/0 = 0$ holds, regardless of the value of a. Similarly, if we try to divide a `Fraction` object by 0, we define that the result should be 0.[1]

For this coursework, you are not supposed to introduce mutator methods, even if all your instance variables are `private` (which they should be). If an object cannot

[1] An alternative, and generally more desirable, solution would be to *throw* an `ArithmeticException` whenever a division by zero is attempted. However, we have not yet discussed Exceptions in this course, so we will stick with the above specification.

be modified after its construction, neither directly nor indirectly, we also speak of an *immutable* object.[2] Examples for such immutable objects in Java are instances of the classes `String` or `BigInteger`. A benefit of immutable objects is that you do not need to worry that someone else might modify your object as a side effect when you pass it to their method.

You should implement at least the following `public` methods (and constructors) in your class `Fraction`. Of course, additional methods may be helpful, and you will certainly also want to use suitable `private` instance variables (fields, attributes) in your class. Unless specified otherwise, you may assume that the arguments passed to your methods are not the value `null`.

class Fraction

Constructors:

`public Fraction(BigInteger numerator, BigInteger denominator)` Constructs a `Fraction` instance which corresponds to the fraction `numerator/denominator`.

`public Fraction(BigInteger val)` Constructs a `Fraction` instance which corresponds to the fraction `val`/1.

`public Fraction(long numerator, long denominator)` Constructs a `Fraction` instance which corresponds to the fraction `numerator/denominator`.

`public Fraction(long val)` Constructs a `Fraction` instance which corresponds to the fraction `val`/1.

Instance Methods:

`public Fraction add(Fraction val)` Returns a Fraction whose value is (`this` + `val`). (Recall that $(a/b) + (c/d) = (a * d + b * c)/(b * d)$.)

`public Fraction subtract(Fraction val))` Returns a Fraction whose value is (`this` − `val`).

`public Fraction multiply(Fraction val)` Returns a Fraction whose value is (`this` ∗ `val`). (Recall that $(a/b) * (c/d) = (a * c)/(b * d)$.)

`public Fraction divide(Fraction val)` Returns a Fraction whose value is (`this` / `val`).

`public Fraction negate()` Returns a Fraction whose value is (−`this`).

`public Fraction invert()` Returns the inverse of this Fraction, i.e., the Fraction 1/`this`.

`public int signum()` Returns the sign of this Fraction: 1 if its value is positive, 0 if it is zero, −1 if it is negative.

[2] If you would like to learn more about immutable objects, the Wikipedia article makes for a nice read: `https://en.wikipedia.org/wiki/Immutable_object`

`public Fraction abs()` Returns the absolute value of this Fraction, i.e., the value of the Fraction itself if it is non-negative, otherwise the negated value.

`public Fraction max(Fraction val)` Returns the maximum of this Fraction and `val`.

`public Fraction min(Fraction val)` Returns the minimum of this Fraction and `val`.

`public Fraction pow(int exponent)` Returns this Fraction taken to the power of `exponent`. Here `exponent` may also be zero or negative. Note that $a^0 = 1$ and $a^b = (1/a)^{(-b)}$ if $b < 0$.

`public int compareTo(Fraction val)` Compares this Fraction with the specified Fraction. Returns -1, 0 or 1 as this Fraction is numerically less than, equal to, or greater than `val`, respectively.

`public boolean isEqualTo(Fraction val)` Checks if this Fraction and `val` represent equal values. Here `val` may also be `null`. In this case your method should return `false`.

`public String toString()` Returns a normalised String representation of this Fraction.

For example, new `Fraction(5,3)` and `new Fraction(-10,-6)` will both be represented as `"(5 / 3)"`. The String representation of `new Fraction(5,-10)` and `new Fraction(-12,24)` is `"(-1 / 2)"`. In general, if this Fraction does not have an integer value, the String representation should be a representation of the "fully reduced fraction" where the denominator is always positive. It should start with an opening parenthesis, then a representation of the numerator, then a whitespace, then a slash, then a whitespace, then a representation of the denominator, and finally a closing parenthesis.

However, in case this Fraction has an integer value, just the String representation of the integer value is returned. For example, `new Fraction(-2)` has the String representation `"-2"`. Moreover, `new Fraction(0)`, `new Fraction(0,3)`, and `new Fraction(4,0)` all have the String representation `"0"`.

Class Methods:

`public static Fraction sumAll(Fraction[] fractions)` Returns the sum of all `Fractions` in `fractions`. If the length of the `fractions` array is 0, the method should return a `Fraction` object corresponding to the value 0.

It is possible that `fractions` is the `null` reference or contains `null` entries. In these cases, your method should not just terminate with an error, but return `null` instead.

Your implementation is not supposed to modify `fractions`.

3 FractionMain

We are providing the file `FractionMain.java` on Moodle which makes use of some of the desired functionalities of the class `Fraction` in its `main` method. You can (and should) test your implementation of `Fraction` by running `main`. It is a requirement that your

implementation of `Fraction` compiles with the *unmodified* `FractionMain.java` from the coursework section in Moodle.

The output that our implementation produces is given here:

```
(01) add:
ACTUAL:   (19 / 10)
EXPECTED: (19 / 10)
(02) add:
ACTUAL:   (2 / 5)
EXPECTED: (2 / 5)
(03) subtract:
ACTUAL:   (-11 / 3)
EXPECTED: (-11 / 3)
(04) multiply:
ACTUAL:   (-1 / 246913578024691357802469135780)
EXPECTED: (-1 / 246913578024691357802469135780)
(05) divide:
ACTUAL:   (6 / 5)
EXPECTED: (6 / 5)
(06) negate:
ACTUAL:   (1 / 2)
EXPECTED: (1 / 2)
(07) negate:
ACTUAL:   (-5 / 3)
EXPECTED: (-5 / 3)
(08) invert:
ACTUAL:   (3 / 5)
EXPECTED: (3 / 5)
(09) signum:
ACTUAL:   1
EXPECTED: 1
(10) signum:
ACTUAL:   0
EXPECTED: 0
(11) signum:
ACTUAL:   -1
EXPECTED: -1
(12) abs:
ACTUAL:   (1 / 12345678901234567890123456789O)
EXPECTED: (1 / 12345678901234567890123456789O)
(13) abs:
ACTUAL:   (1 / 2)
EXPECTED: (1 / 2)
(14) max:
ACTUAL:   (5 / 3)
EXPECTED: (5 / 3)
(15) max:
ACTUAL:   (5 / 3)
EXPECTED: (5 / 3)
(16) min:
ACTUAL:   (1 / 12345678901234567890123456789O)
EXPECTED: (1 / 12345678901234567890123456789O)
```

```
(17) min:
ACTUAL:   (1 / 12345678901234567890123456789O)
EXPECTED: (1 / 12345678901234567890123456789O)
(18) pow:
ACTUAL:   (25 / 9)
EXPECTED: (25 / 9)
(19) pow:
ACTUAL:   1
EXPECTED: 1
(20) pow:
ACTUAL:   (27 / 125)
EXPECTED: (27 / 125)
(21) compareTo:
ACTUAL:   0
EXPECTED: 0
(22) compareTo:
ACTUAL:   0
EXPECTED: 0
(23) compareTo:
ACTUAL:   0
EXPECTED: 0
(24) compareTo:
ACTUAL:   -1
EXPECTED: -1
(25) compareTo:
ACTUAL:   1
EXPECTED: 1
(26) isEqualTo:
ACTUAL:   false
EXPECTED: false
(27) isEqualTo:
ACTUAL:   true
EXPECTED: true
(28) isEqualTo:
ACTUAL:   false
EXPECTED: false
(29) toString:
ACTUAL:   0
EXPECTED: 0
(30) toString:
ACTUAL:   (-1 / 2)
EXPECTED: (-1 / 2)
(31) toString:
ACTUAL:   2
EXPECTED: 2
(32) sumAll:
ACTUAL:   (107 / 30)
EXPECTED: (107 / 30)
(33) sumAll:
ACTUAL:   null
EXPECTED: null
```

Note, however, that the tests performed by `FractionMain` are not meant to be exhaustive – so even if `FractionMain` produces the same output as given above, this does not automatically imply that your implementation is necessarily correct for all purposes. Thus, it is a good idea to not only test your code, but also to review it before you hand in your solution.

4 Additional Requirements

- Out of the four constructors, try to write only a single one that explicitly writes to the instance variables of the class. The other three constructors should then just call that constructor via `this(...)` with suitable arguments.
- Your implementations of the above methods should not modify their arguments.
- Every method (including constructors) should have Javadoc comments. You can use Eclipse's `Source` → `Generate Element Comment` to get a suitable template with `@param` entries for all method parameters and with `@return` for methods that have a non-void return type. (Here we have already provided comments for the required methods and constructors, but if you write additional methods or constructors, you should of course document them.)
- Every class you write should have your name in the Javadoc comment for the class itself, using the Javadoc tag `@author`.
- Your source code should be properly formatted. You can use Eclipse's `Source` → `Format` for this purpose.
- *Reminder – use (also auxiliary) methods.* Don't cram everything into one or two methods, but try to divide up the work into sensible parts with reasonable names. Every method should be short enough to see all at once on the screen.
- Your submission should contain at least the file `Fraction.java`. You may also write further classes, which you should then include as well. (Note however that our solution only consists of the file `Fraction.java`.)
- We are making a git repository including both `FractionMain` and a skeleton implementation of `Fraction` available on Moodle. Please copy this repository and *commit your changes* to your copy. Use meaningful commit messages. In the end, please submit your repository with your commit history.

5 Hints

- Observe that the fractions $-4/8$, $12/-24$, and $-1/2$ are all *equal*, i.e., they describe the same value.

  It may ease your implementation effort if your constructors for the `Fraction` class constructs objects where the instance variables have the *same values* in all three cases, regardless of whether we call `new Fraction(-4, 8)`, `new Fraction(12, -24)`, or `new Fraction(-1, 2)`. To implement such a *normalisation*, the method `gcd` from the class `BigInteger` may be helpful.

- Note that many of the methods in the class `Fraction` have related functionalities. For instance, if you need to `subtract` two fractions, you might as well *call* your method to `add` two fractions on suitable arguments instead of writing a slightly modified version of your `add` method.

- For those methods where you need to write own code to do something, it can be convenient to just use the methods from the class `BigInteger`. Is there a way you can reduce the problem of the computation you are supposed to be doing to a computation that is performed by `BigInteger` objects? You do not need to scrutinise the whole documentation of `BigInteger` – just try to find method names that look suitable and then read up on what this particular method does.

- Keep an eye on the way the methods are supposed to deal with `null` as an actual parameter (or as an element of an array that is an actual parameter).