CS2030 (2310) Mock PA #2

Tags & Categories

Tags:

Categories:

Related Tutorials

Task Content

Natural Numbers

Natural numbers are commonly observed in nature, and have been used for counting since time immemorial. Natural numbers start with zero, and the successor of a natural number is also a natural number. As an example, one (`1`) is the successor of zero (`succ(0)`), two (`2`) is the successor of one (`succ(1)`), etc.

Simple operations can be performed on natural numbers. In particular, addition can be observed as a repeated successor application. For example, adding two to zero is `succ(succ(0))` which gives two (`2`). With addition defined, other operations such as subtraction and multiplication can follow.

Task

In this task, you are to define the common operations of natural numbers, as well as natural fractions, i.e. fractions whose numerator and denominator are natural numbers.

You are given the `AbstractNum` class shown below.

```
import java.util.Optional;
import java.util.function.Function;
import java.util.function.Predicate;

class AbstractNum<T> {
    protected static final Function<Integer, Integer> s = x -> x + 1;
    protected static final Function<Integer, Integer> n = x -> -x;
    protected static final Predicate<Integer> valid = x -> x >= 0;

    protected final Optional<T> opt;

    protected AbstractNum(T i) {
        this(Optional.<T>of(i));
    }

    protected AbstractNum(Optional<T> opt) {
        this.opt = opt;
    }

    static AbstractNum<Integer> zero() {
        return new AbstractNum<Integer>(0);
    }

    public boolean equals(Object obj) {
        if (obj == this) {
            return true;
        } else {
            if (obj instanceof AbstractNum<?> abs) {
                return abs.opt.equals(this.opt);
```

```
            } else {
                return false;
            }
        }
    }

    protected boolean isValid() {
        return this.opt.map(x -> true).orElse(false);
    }

    public String toString() {
        return this.opt.map(x -> x.toString()).orElse("NaN");
    }
}
```

Note the definitions of the two functions (`s` and `n`) and predicate `valid`. These are the only lambdas that you will need for arithmetic manipulation. In your implementation, you are not allowed to use the operators `+ - * / %` (other similar forms e.g. `+=`, `++`, etc.), or bitwise operations to perform arithmetic manipulation.

There is a single instance property `opt` of type `Optional<T>`. This is the only property that you need. As such, you are not allowed to define any other instance properties, including constants, in your implementation.

The `static` method `zero()` creates a `AbstractNum` with the integer zero. This is the only "value" that you will need in your implementation. As such, you are not allowed to use numerical values in your implementation.

The `equals` method is provided. Only use this to compare natural numbers and fractions. You will not be allowed to use comparison operators `< > == != <= >=`. The predicate `valid` may come in handy.

Two other methods `isValid` and `toString` are provided for you. You will not need to define any `toString` method in your implementation. In addition, there should be no dependency on `String` in your implementation. Also note the use of `orElse` in these methods. In your implementation, you are not allowed to use the `Optional` methods `isPresent`, `isEmpty`, or any form of get methods, i.e. `get`, `orElse`, `orElseGet`, `orElseThrow`.

Other Constraints

The following are the usual constraints. You should keep to the constructs and programming discipline instilled throughout the module.

- Write each `class`/`interface` in its own file.
- Ensure that ALL object properties and class constants are declared `private final`.
- Ensure that that your classes are NOT cyclic dependent.
- ONLY the following java libraries ARE allowed:
    - `java.lang`
    - `java.util.List`
    - `java.util.Optional`
    - `java.util.stream.Stream`, and
    - functional interfaces from `java.util.function`.
- Use `&&`, `||` and `!` in logical expressions.
- You are NOT allowed to use java keywords/literals: `null`, `instanceof`
- You are NOT allowed to use Java reflection, i.e. `Object::getClasses` and other methods from `Class`
- You are NOT allowed to use `*` wildcard imports.
- You are NOT allowed to define anonymous inner classes; define lambdas instead.
- You are NOT allowed to define array constructs, e.g. `String[]` or using ellipsis, e.g.`String...`

This task comprises a number of levels. You are required to complete ALL levels.

Level 1

Write an immutable `Num` class that inherits from `AbstractNum` with the following `static` methods:

- `of(int i)` that takes in an integer `i` and returns it as a natural number if `i >= 0`, or an invalid natural number `NaN` otherwise. You will need to define appropriate `private` constructor(s). Use `AbstractNum.valid` to help you determine the validity of the natural number.
- The `zero()` method is defined for you. It returns the natural number representing zero. You will need to write the accompanying `private` constructor.

```
    static Num zero() {
        return new Num(AbstractNum.zero());
    }
```

```
jshell> Num zero = Num.zero()
zero ==> 0
```

```
jshell> Num one = Num.of(1)
one ==> 1

jshell> Num invalid = Num.of(-1)
invalid ==> NaN

jshell> List<AbstractNum> list = List.<AbstractNum>of(zero, one, invalid)
list ==> [0, 1, NaN]
```

Using the function `s` defined in `AbstractNum`, write a `succ` method that returns the successor of a natural number. If the number is invalid, it's successor is also invalid. In addition, write a `static` method `one()` that returns the natural number `1`.

```
jshell> Num.one()
$.. ==> 1

jshell> Num.zero().succ()
$.. ==> 1

jshell> Num.one().equals(Num.zero().succ()) // using AbstractNum::equals
$.. ==> true

jshell> Num.of(-1).succ()
$.. ==> NaN

jshell> Num.one().succ().succ()
$.. ==> 3

jshell> Num.of(-1).succ().succ()
$.. ==> NaN
```

Level 2

Write the `add` method that takes in a natural number and returns the result of addition as a natural number. You will need to call `succ` repeatedly until the result is reached. Moreover, an add operation is only possible if both operands are valid.

You may choose to write a loop construct or use `Stream`. Do not use recursion. Also pay attention to the restriction on `Optional` methods.

```
jshell> Num one = Num.one()
one ==> 1

jshell> Num zero = Num.zero()
zero ==> 0

jshell> Num invalid = Num.of(-1)
invalid ==> NaN

jshell> zero.add(zero)
$.. ==> 0

jshell> zero.add(one)
$.. ==> 1

jshell> zero.add(invalid)
$.. ==> NaN

jshell> one.add(zero)
$.. ==> 1

jshell> one.add(one)
$.. ==> 2

jshell> one.add(invalid)
$.. ==> NaN

jshell> invalid.add(zero)
$.. ==> NaN

jshell> Num result = zero.add(one).add(zero).add(one)
result ==> 2
```

```
jshell> result.add(result)
$.. ==> 4

jshell> zero.add(invalid).add(zero).add(one)
$.. ==> NaN
```

Level 3

Include the `mul` and `sub` methods to perform multiplication and subtraction respectively. Suppose `m` and `n` are valid natural numbers,

- For multiplication, say `m.mul(n)`, it can be seen as `m` repeated additions of `n`, or vice-versa.
- For subtraction, say `m.sub(n)`, it is observed that it is equivalent to the addition, i.e. `(-n).add(m)`. That is to say, we need to negate the argument `n` (without invalidating it), and then add `m` to it. Use the `n` function in `AbstractNum`.

Just like addition, the `mul` and `sub` operations are only possible if both operands are valid.

```
jshell> Num one = Num.one()
one ==> 1

jshell> Num zero = Num.zero()
zero ==> 0

jshell> Num invalid = Num.of(-1)
invalid ==> NaN

jshell> zero.sub(zero)
$.. ==> 0

jshell> zero.sub(one)
$.. ==> NaN

jshell> zero.sub(invalid)
$.. ==> NaN

jshell> one.sub(zero)
$.. ==> 1

jshell> one.sub(one)
$.. ==> 0

jshell> one.sub(invalid)
$.. ==> NaN

jshell> invalid.sub(zero)
$.. ==> NaN

jshell> zero.mul(zero)
$.. ==> 0

jshell> zero.mul(one)
$.. ==> 0

jshell> zero.mul(invalid)
$.. ==> NaN

jshell> one.mul(zero)
$.. ==> 0

jshell> one.mul(one)
$.. ==> 1

jshell> one.mul(invalid)
$.. ==> NaN

jshell> invalid.mul(zero)
$.. ==> NaN

jshell> Num.of(2).mul(Num.of(3))
$.. ==> 6
```

Level 4

Natural numbers are pretty boring. Let's include natural fractions!

A valid natural fraction comprises a numerator and denominator part that are valid natural numbers. In addition, the denominator must not be zero.

The `Frac` class is given to you.

```
class Frac extends Pair<Num, Num> {
    private Frac(Num n, Num d) {
        super(n, d);
    }

    static Frac of(Num n, Num d) {
        return new Frac(n, d);
    }

    @Override
    public String toString() {
        return String.format("%s : %s", first(), second());
    }
}
```

It's a simplified `Pair` class where both values of the pair are of type `Num`. It also contains a `toString` method that returns the appropriate `String` representation of the fraction.

Write an immutable `Fraction` class that inherits from `AbstractNum` with a `static` method `of` (with appropriate constructors) that takes in two integer values `n` and `d` and creates a fraction `n : d`

```
jshell> Fraction.of(1, 2)
$.. ==> 1 : 2

jshell> Fraction.of(2, 1)
$.. ==> 2 : 1

jshell> Fraction.of(2, 2)
$.. ==> 2 : 2

jshell> Fraction.of(0, 2)
$.. ==> 0 : 2

jshell> Fraction.of(2, 0)
$.. ==> NaN

jshell> Fraction.of(-1, 2)
$.. ==> NaN

jshell> Fraction.of(1, -2)
$.. ==> NaN
```

Level 5

Include the `add`, `sub` and `mul` operations to operate on two fractions.

In case you forget,

```
 a     c     ad + bc
--- + --- = ---------
 b     d       bd

 a     c     ad - bc
--- - --- = ---------
 b     d       bd

 a     c     ac
--- x --- = ----
 b     d     bd
```

Just like `Num`, operations are only possible if both operands are valid.

```
jshell> Fraction.of(1, 2).add(Fraction.of(1, 4))
$.. ==> 6 : 8

jshell> Fraction.of(-1, 2).add(Fraction.of(1, 4))
```

```
$.. ==> NaN

jshell> Fraction.of(1, 2).sub(Fraction.of(1, 4))
$.. ==> 2 : 8

jshell> Fraction.of(1, 4).sub(Fraction.of(1, 2))
$.. ==> NaN

jshell> Fraction.of(1, 2).mul(Fraction.of(2, 1))
$.. ==> 2 : 2

jshell> Fraction.of(1, 2).mul(Fraction.of(2, 0))
$.. ==> NaN
```

Self-challenge... you may want to try implementing a method `reduce()` to simplify the fraction.

Submission (Course)

Select course: CS2030 (2023/2024 Sem 1) - Programming Methodology II

Your Files:

BROWSE

SUBMIT (only .java, .c, .cpp, .h, .jsh, and .py extensions allowed)

To submit multiple files, click on the Browse button, then select one or more files. The selected file(s) will be added to the upload queue. You can repeat this step to add more files. Check that you have all the files needed for your submission. Then click on the Submit button to upload your submission.