Allgemeines Programmierpraktikum **Sommersemester 2023**
Dr. Henrik Brosenne Georg-August-Universität Göttingen
Institut für Informatik

Übung 02

~~**Das *tagging* muss vor dem Abgabetermin Do., 11.05., 8 Uhr** erfolgen.~~
Das *tagging* muss vor dem Abgabetermin Mo., 15.05., 8 Uhr erfolgen.
Testate ab Mo., 15.05.

Vorbereitung

Prüfsumme und Testat

Bitte verwenden Sie ihr GitLab Projekt, das Sie für die Übung 01 erstellt haben, auch für diese und alle weiteren Übungen. Folgen Sie der gleichen Abgabeprozedur (*tagging*).

Java

Lesen Sie das Kapitel *5.4 Abstrakte Datentypen* aus dem Skript *Informatik I, Wintersemester 20/21.*

Informieren Sie sich in der *Java Platform, Standard Edition 11 API Specification* `https://docs.oracle.com/javase/11/docs/api/` über die nicht typsichere Version der Collection `java.util.Stack`.

Bemerkung

Bei den Aufgaben dieser Übung können *unchecked warnings*, z.B. die Folgende, ignoriert werden.

```
Note: xxx.java uses unchecked or unsafe operations.
Note: Recompile with -Xlint:unchecked for details.
```

Allgemein

1. Verwenden Sie `ant` zum automatisierten Übersetzen des Quelltext und zum Erzeugen der Dokumentation. Nach dem Übersetzen befinden sich die `java`-, die `class`-Dateien und die Dokumentation in unterschiedlichen Verzeichnissen.

2. Kommentieren Sie alle Klassen ausführlich. Erstellen Sie für die Klassen selbst und alle Attribute Dokumentationskommentare für `javadoc`. Erzeugen Sie mit `javadoc` eine API-Dokumentation, inklusive `private`-Attribute, der Klasse.

Aufgabe 1

Rationale Zahlen

Gegeben ist die Schnittstelle `Fractional` für Brüche mit Zähler (*numerator*) und Nenner (*denominator*), auf denen einfache arithmetische Operationen möglich sind. Siehe `uebung/uebung02-data` im GitLab der Vorlesung `https://gitlab.gwdg.de/app/2023ss/lecture`.

```
package app.exercise.algebra;

public interface Fractional {
    // get numerator
    long getN();
    // get denominator
    long getD();
    // add operand to object
    void add(Fractional operand);
    // subtract operand from object
    void sub(Fractional operand);
    // multiply object by operand
    void mul(Fractional operand);
    // divide object by operand
    void div(Fractional operand);
    // new additive inverse object
    Fractional negation ();
    // new multiplicative inverse object
    Fractional reciprocal ();
}
```

1. Implementieren Sie im Package `app.exercise.algebra` eine `abstract` Klasse `BasisFraction`, die die Schnittstelle `Fractional` implementiert.

   a) Die Klasse hat keine Objekt- oder Klassenvariablen.

   b) Es wird folgende Methode zum Setzen von Zähler und Nenner deklariert.

   ```
   protected abstract void setND(long numerator, long denominator);
   ```

c) Die Methoden `add`, `sub`, `mul` und `div` werden definiert. Wobei `negation` für die Implementation von `sub` und `reciprocal` für die Implementation von `div` verwendet wird.

2. Implementieren Sie im Package `app.exercise.algebra` eine Klasse `Rational`, die von der Klasse `BasisFraction` aus Aufgabenteil 1. erbt.

   a) Die Klasse `Rational` speichert rationale Zahlen in einer Normalform für die Folgendes gilt.

      - Der Zähler und Nenner sind teilerfremd, das wird erreicht indem beide durch ihren größten gemeinsamen Teiler (*greatest common divisor*, `gcd`) geteilt werden. Der `gcd` kann z.B. mit dem Euklidischen Algorithmus (siehe Skript von *Informatik I*) ermittelt werden.
      - Der Nenner ist immer positiv, d.h. negative Brüche besitzen einen negativen Zähler.

   b) Implementieren Sie die noch nicht definierten Methoden der Schnittstelle `Fractional`.

   c) Überschreiben Sie die von der Superklasse `java.lang.Object` geerbten Methoden `clone` und `toString` sinnvoll. Implementieren Sie `toString` so, dass eine möglichst kompakte Darstellung der rationalen Zahl erzeugt wird.

   d) Überschreiben Sie die von der Superklasse `java.lang.Object` geerbten Methoden `equals` und `hashCode`. Stellen Sie die in der API für `equals` geforderten Eigenschaften (reflexive, symmetric, transitive, consistent) sicher. Sorgen Sie weiterhin dafür, dass für zwei Objekte, die von `equals` als gleich erkannt werden, `hashCode` denselben Wert zurückliefert, wie in der API gefordert.

3. Schreiben Sie im Package `app.exercise.testing` eine ausführbare Klasse `RationalTesting`, die `Rational` rudimentär testet. Es sollen mindestens zwei Objekte erzeugt und alle Methoden der Klasse wenigstens zweimal aufgerufen werden.

Aufgabe 2

Umgekehrte polnische Notation

Unten finden Sie weitere Erläuterungen zur *umgekehrten polnischen Notation*.

1. Schreiben Sie im Package `app.exercise.testing` eine ausführbare Klasse `RPN` (*Reverse Polish Notation*), die Ausdrücke in umgekehrter polnischer Notation auf der Kommandozeile übergeben bekommt und das Ergebnis ausgibt.

   Zulässige Ausdrücke sind wie folgt aufgebaut.

   - Zahlen sind ganzzahlig und nicht-negativ, d.h. sie bestehen nur aus Ziffern. Das schließt nicht aus, dass das Resultat eines Ausdrucks negativ oder rational ist.

- Operatoren sind `+` (Addition), `-` (Subtraktion), `*` (Multiplikation) und `/` (Division).

Nicht zulässige Ausdrücke werden erkannt und gemeldet.

Für die Arithmetik werden ausschließlich Variablen der Schnittstelle `Fractional` verwendet, in denen Referenzen auf Objekte der Klasse `Rational` gespeichert sind.

Der Stack, auf dem die Operanden abgelegt werden, ist ein Objekt der Klasse `java.util.Stack`.

Umgekehrte polnische Notation

Eine Modifikation der polnischen Notation ist die umgekehrte polnische Notation (Postfix-Notation, *Reverse Polish Notation, RPN*), bei der der Operator hinter die verbundenen Operaden geschrieben wird.

- Zahlen sind Operanden
- Operatoren beschreiben Verknüpfungen zwischen zwei Operanden. Ein Operator wirkt auf die beiden Operanden direkt vor ihm. Die Anwendung eines Operators auf seine Operanden wird als Einheit betrachtet und ist wieder ein Operand.

Diese Forderungen macht Klammern und Bindungsregeln für Operatoren überflüssig.

Beispiel

```
  1 2 + = 3                          RPN
  1 2 + 3 * = 9
  1 2 3 * + = 7

  1 2 + 3 / 4 * 5 6 - +              RPN
= ((((1 2 +) 3 /) 4 *) (5 6 -) +)    Klammerung durch Operatoren erzungen
= ((((1 + 2) / 3) * 4) + (5 - 6))    Infix-Notation
= 3

  5 4 3 2 1 + * - /                  RPN
= (5 (4 (3 (2 1 +) *) -) /)          Klammerung durch Operatoren erzungen
= (5 / (4 - (3 * (2 + 1))))          Infix-Notation
= -1
```

Zur Auswertung eines *RPN*-Ausdrucks wird die Eingabe von links nach rechts durchlaufen. Wird eine Zahl gelesen, wird diese auf den Stack gelegt. Wird ein Operator gelesen, werden zwei Zahlen vom Stack geholt, mit dem Operator verknüpft und das Ergebnis auf den Stack gelegt. Nach dem Abarbeiten der Eingabe liegt genau eine Zahl, das Ergebnis des Ausdrucks, auf dem Stack.