Programmieren in Java

http://proglang.informatik.uni-freiburg.de/teaching/java/2015/

Java-Übung Blatt 10 (Iterator, Funktionen höherer Ordnung)

2015-07-06

Hinweise

- Ändern Sie nicht die Schreibweise von Bezeichnen, die auf dem Übungsblatt vorgegeben sind. Dies betrifft sämtliche global sichtbaren Namen von Eclipse-Projekten, Paketen, Klassen, etc.
- Bezeichner und Kommentare bitte auf *Englisch*!
- Schreiben Sie *sinnvolle* Kommentare.
- Laden Sie Ihre Lösungen mit Subversion (SVN) ins Übungssystem hoch. Den entsprechenden Pfad finden Sie online.
- Sollte das Übungssystem Ihre Einreichung nicht übersetzen können, dann wird sie nicht korrigiert und Sie erhalten keine Punkte.
- Die Korrektur Ihrer Abgabe wird unter dem Namen `Feedback-<login>-ex<NN>_<X>.txt` in das jeweilige Projektverzeichnis eingecheckt. Dabei ist `<login>` Ihr *myAccount*-Name und `<NN>_<X>` der Projektname.
- Sie können Ihre Gesamtpunktzahl im Übungsportal einsehen.
- Dokumentieren Sie Ihren Zeitaufwand, den gefühlten Schwierigkeitsgrad, sowie Probleme beim Lösen der Aufgabe oder auch Anregungen und Vorschläge zur Verbesserung in der Datei `<Projektverzeichnis>/erfahrungen.txt`

Exercise 1 (Stream Editor, 4 Punkte)

Projekt: `ex10_1`. Package: `stream`.

Sie modellieren in dieser Aufgabe das Grundgerüst eines einfachen Stream-Editors, mit dem eine Folge von Wörtern modifiziert werden kann. Die Grundbausteine hierzu sollen verkettete String-Iteratoren sein, d.h. Instanzen des aus der Vorlesung bekannten `Iterator<String>` Interfaces.

Die konkrete Aufgabe ist nun:

1. (1.5 Punkte) Schreiben Sie zunächst einen String-Iterator `Words`, der als Eingabestrom dient. Die Klasse soll einen Konstruktor enthalten, der einen `String` übergeben bekommt und diesen in die einzelnen, durch Leerzeichen getrennten Wörter zerlegt (Tipp: nehmen Sie geeignete Methoden aus `java.lang.String` zur Hilfe). Über die einzelnen Wörter soll dann mit `hasNext` und `next` navigiert werden können.
   Testen Sie (mit JUnit) alle Methoden aus dem Iterator-Interface (bis auf `remove`).

2. (2.5 Punkte) Es sollen nun verschiedene Arten Modifikatoren realisiert werden, die mit Hilfe des `Iterator<String>`-Interfaces auf Eingabeströmen (wie z.B. `Words`) arbeiten und dieses Interface auch selbst implementieren (um wiederum anderen Modifikatoren als Eingabestrom zu dienen). Implementieren Sie mindestens die folgenden Modifikatoren:
   - Änderung der Schreibweise zu Großschreibung (Klasse `UpperCase`);
     z.B. wird `("aaa", "bbb", "ccc")` zu `("AAA", "BBB", "CCC")`.
   - Ersetzen eines Wortes durch ein anderes (Klasse `ReplaceWord`);
     z.B. wird `("aaa", "bbb", "ccc")` zu `("ddd", "bbb", "ccc")`, wenn ein `ReplaceWord` benutzt wird, der `"aaa"` gegen `"ddd"` ersetzt.
   - Herausfiltern eines bestimmten Wortes (Klasse `RemoveWord`;
     z.B. wird `("aaa", "bbb", "ccc")` zu `("bbb", "ccc")`, wenn ein `RemoveWord` benutzt wird, der das Word `"aaa"` herausfiltert.

   Insbesondere sollen beliebige Modifikatoren miteinander kombinierbar sein.

Testen Sie (mit JUnit) ihre Implementierungen und verschiedene Kombinationen von Modifikatoren auf interessanten Eingabedaten.

Exercise 2 (Iterator Transformer, 6 Punkte)

Projekt: `ex10_2`. Package: `iter`. In der Vorlesung haben Sie gesehen, wie die Elemente von Collections mittels `ITransform` Implementierungen beliebig verändert werden können. Das `ITransform`-Interface finden sie auch noch einmal im Template zu dieser Aufgabe.

1. (2 Punkte) In dieser Aufgabe sollen Sie die `Transform` Klasse für Iteratoren implementieren. Beispiel: Eine `map` Operation mit Verdopplungs-`ITransform` auf einem Integer-Iterator der die Zahlen 1,2,3,4 liefert, ergibt einen Ausgabeiterator, der die Zahlen 2,4,6,8 liefert.

   Die Signatur von Transform soll also wie folgt aussehen:

```
class Transform {
    /**
     * Applies an ITransform from type T to S on each element
     * of an iterator.
     *
     * @param source The input elements.
     * @param trans The transform.
     * @return Provides a new iterator with the transformed elements.
     *
     * @param <T> The type of the input elements.
     * @param <S> The type of the output elements.
     */
    public static <T, S> Iterator<S> map(Iterator<T> source,
                        ITransform<? super T, ? extends S> trans) { ... }
}
```

2. (0,5 Punkte) Implementieren Sie den Modifikator `UpperCase` aus `ex10_1` erneut, aber benutzen Sie diesmal die `map` Methode aus der `Transform` Klasse.

3. (3 Punkte) Eine weitere manchmal interessante Operation auf Iteratoren sind sogenannte Scans. Ein Scan wendet eine zweistellige Operation sukzessive auf eine Reihe Elemente an, und gibt dabei alle Zwischenergebnisse zurück.

   Ein Beispiel für einen Scan mit der Operation „+" (Addition) wäre die Umwandlung der Integer Reihe $1, 2, 3, 4$ in $1, 3, 6, 10$ (also $0 + 1, 1 + 2, 3 + 3, 6 + 4$). Die 0 muss hier als Startwert (oder „neutrales Element") beim Aufruf der Operation mitangegeben.

   Ein anderes Beispiel ist der Scan mit der Operation „add" (entsprechend der `add` Methode des `List` Interfaces): Die Reihe $1, 2, 3, 4$ würde in $[1], [1, 2], [1, 2, 3], [1, 2, 3, 4]$ umgewandelt werden[1]. Das neutrale Element ist hier die leere Liste „[]".

   Die binären Operationen sollen, ähnlich wie die einstelligen Transformationen aus dem ersten Teil, durch Objekte repräsentiert werden. Das Interface hierzu soll `ScanOp` heißen. Definieren Sie ein `IScanOp` Interface; Sie benötigen zwei Typparameter.

   Implementieren Sie zusätzlich die `Scan` Klasse für Iteratoren. Dabei hat `Scan` folgende Signatur:

```
class Scan {
    /**
     * Applies a fold from types T and S to S on elements of an Iterator<T>
     * with a start value, yielding all intermediate results in the process.
     *
     * @param source The source iterator.
     * @param start The start value, or ''neutral element''
     * @param op The binary operation.
     * @return The resulting iterator after the scan.
     * @param <T> The type of the input elements.
     * @param <S> The type of the output elements.
     */
    public static <T,S> Iterator<S> scan(Iterator<T> source, S start,
        IScanOp<? super T,S> op) { ... }
}
```

   Testen Sie Ihren Code mit JUnit, und mindestens drei verschiedenen `IScanOp` Implementierungen!

4. (0,5 Punkte) Begründen Sie warum in der obigen Signatur von scan der formale Parameter op keine IScanOp<? **super** T,? **extends** S> (im Gegensatz zu IScanOp<? **super** T,S>) sein kann.

[1] $[x, y, z]$ symbolisiert hier, als abkürzende Schreibweise, ein `List` Objekt mit den Elementen x, y und z. Ausgeschrieben hieße das etwas wie `Arrays.asList(new int[] {x, y, z})` Die leere Liste (`new ArrayList<Integer>()`) schreiben wir als []