Sven Eric Panitz

Lehrbriefe
Programmierung in Java

Von Schleifen zu Faltungen

Von Schleifen zu Faltungen

Sven Eric Panitz

2. Mai 2023

Inhaltsverzeichnis

1 Von Schleifen zu Faltungen

Wir haben im letzten Lehrbrief Iteratoren kennengelernt als Abstraktion darüber, wie eine Schleife zum Iterieren über einen Iterationsbereich läuft. Statt Schleifen gab es Objekte mit Methoden, die die Schleifenausführung ersetzten.

Wem Methoden höherer Ordnung nicht gefallen, kann aber ebenso als syntaktischen Zucker für Objekte, die die Schnittstelle `Iterable` umsetzen, die for-each-Schleife verwenden.

In diesem Lehrbrief gehen wir einen Schritt weiter und betrachten eine der häufigsten Arten von Schleifen, die veverwendet werden, um mit den Elementen eines Iterationsbereiches ein Ergebnis zu berechnen.

Wir importieren zunächst die benötigten Schnittstellen:

Reduction.java

```
package name.panitz.util;
import java.util.function.Function;
import java.util.function.Predicate;
import java.util.function.BiFunction;
import java.util.*;
import java.util.stream.Collectors;
```

Wir entwickeln in einer Klasse `Reduction` eine Reihe von statischen Methoden, die mit iterierbaren Objekten arbeiten.

Reduction.java

```
public class Reduction{
```

1.1 Beispielfunktionen für Schleifen über einen Iterationsbereich

Betrachten wir ein paar typische Funktionen, die mit den Elementen eines Iterationsbereiches ein Ergebnis errechnen:

Durchzählen der Elemente Folgende einfach zu verstehende Funktion zählt die Anzahl der Elemente einer Iteration durch.

Reduction.java

```
  static public int count(Iterable<String> xs){
    int result = 0;
    for (String x : xs) {result = result + 1;}
    return result;
  }
```

Wir legen eine Ergebnisvariable an, initialisieren diese, iterieren über die Elemente, verändern die Ergebnisvariable bei jedem Schleifendurchlauf und geben diese schließlich zurück.

Anzahl der Gesamtlänge aller String-Objekte Eine andere Aufgabe, soll die Gesamtzahl der Zeichen aller Strings eines Iterationsbereichs berechnen.

Reduction.java

```
  public static int chrNo(Iterable<String> xs){
    int result = 0;
    for (String x : xs) {result = result + x.length();}
    return result;
  }
```

Auch hier wieder: wir legen eine Ergebnisvariable an, initialisieren diese, iterieren über die Elemente, verändern die Ergebnisvariable bei jedem Schleifendurchlauf und geben diese schließlich zurück.

Im Unterschied zur Funktion `count` wird die Ergebnisvariable in jedem Schleifendurchlauf auf andere Art und Weise verändert.

Konkatenation aller Strings mit Leerzeichen Die beiden letzten Funktionen hatten eine Zahl als Ergebnis. Jetzt soll ein String errechnet werden, indem alle Strings einer Iteration aneinander gehängt werden.

Reduction.java

```
public static String ap(Iterable<String> xs){
  String result = " ";
  for (String x : xs) {result = result + " " + x;}
  return result;
}
```

Und schon wieder: wir legen eine Ergebnisvariable an, initialisieren diese, iterieren über die Elemente, verändern die Ergebnisvariable bei jedem Schleifendurchlauf und geben diese schließlich zurück.

Jetzt ist die Ergebnisvariable ein String-Objekt.

Suche nach längsten aller Strings Eine doch recht andere Fragestellung ist die, nach dem längsten String in einem Iterationsbereich.

Reduction.java

```
public static String lo(Iterable<String> xs){
  String result = "";
  for (String x : xs) {result= result.length() > x.length()?result:x;}
  return result;
}
```

Auch in dieser Funktion gehen wir genau so wie bei den drei Vorgängern vor: wir legen eine Ergebnisvariable an, initialisieren diese, iterieren über die Elemente, Verändern die Ergebnisvariable bei jedem Schleifendurchlauf und geben diese schließlich zurück.

Test, ob ein String enthalten ist Und als letztes Beispiel betrachten wir einen Funktion, die testet, ob ein bestimmter String in einem Iterationsbereich enthalten ist.

Reduction.java

```
public static boolean co(String y,Iterable<String> xs){
  boolean result = false;
  for (String x : xs) {result = result || x.equals(y);}
  return result;
}
```

Und ja, auch diese Funktion ist gleich aufgebaut, wie die Vorgänger.

1.2 Faltungsfunktion

So unterschiedlich die Funktionen dieser Beispiele sind, haben sie doch alle die gleiche Struktur.

Werfen wir einen Blick darauf, was alle diese Funktionen gemeinsam haben. Sie bearbeiten zunächst alle die Elemente eines Iterationsbereichs. Wir können den Typ generisch halten und verwenden hierzu die Typvariable `E` für *Element*.

Alle Funktion machen eine for-each-Scheife über den Iterationsbereich. Innerhalb dieser for-each-Schleife, wird eine Variable `result` neu gesetzt. Diese Variable bekommt zuvor einen initialen Startwert.

Versuchen wir das schrittweise in einer Methode zusammen zu fassen, so bekommen wir zunächst folgendes Fragment.

Javafragment 1

```
public static <E  >  ...  fold( Iterable<E> xs, ...  result, ... ) {
  for (E x : xs) {result = ... }
  return   result;
}
```

Die Argumente sind das Iterable und die mit dem Startwert gesetzte Variable `result`. Die Methode ist generisch über die Elemente des Iterable.

Jetzt fehlt uns noch das Ergebnis. Die Funktionen haben unterschiedliche Ergebnistypen, die unabhängig vom Elementtypen sein können. Daher führen wir eine zweite Typvariable ein, die für das Ergebnis steht. Wir fügen die Typvariable `R` für `Result` hinzu.

Javafragment 2

```
public static <E, R> R fold( Iterable<E> xs, R  result, ... ) {
  for (E x : xs) {result = ...}
  return   result;
}
```

Es fehlt schließlich nur noch die Zuweisung an die Ergebnisvariable innerhalb der Schleife. Betrachtet man alle Beispiele, so stellt man fest, dass die rechte Seite dieser Zuweisung immer ein Ausdruck ist, der aus dem bisherigen Wert von `result` und dem aktuellen Schleifenelement `x` einen neuen Wert berechnet. `result` ist vom Typ `R`, die Variabnle `x` vom Typ `E`. Wir brauchen also eine Funktion, die ein bisheriges Ergebnis `R` und ein aktuelles Element `E` zu einem neuen Teilergebnis `E` verrechnet. Eine solche Funktion ist vom Typ `BiFunction<R,E,R>`.

Reduction.java

```
public static<E,R> R fold(Iterable<E> xs,R result,BiFunction<R,E,R> o){
  for (E x : xs) {result = o.apply(result,x);}
  return result;
}
```

1.3 Anwendung der Faltungsfunktion

Wir können jetzt alle unsere Beispielfunktionen mit einem geschickten Aufruf der Funktion `fold` umsetzen. Dabei ist nur zu überlegen, was der initiale Wert von `result` ist und wie die Funktion aussieht, die ein Zwischenergebnis mit einem Element zu einem weiteren Zwischenergebnis verrechnet.

Durchzählen der Elemente Beim Durchzählen der Elemente starten wir mit 0 für `result` und erhöhen für jedes Element `x` das Ergebnis um 1:

Reduction.java

```
public static int countF(Iterable<String> xs){
  return fold(xs, 0, (result,x) -> result + 1);
}
```

Anzahl der Gesamtlänge aller String-Objekte Auch bei der Berechnung der Gesamtanzahl von Zeichen, starten wir mit 0. Die Funktion addiert die Länge des aktuellen Elements zum Ergebnis.

Reduction.java

```
public static int chrNoF(Iterable<String> xs) {
  return fold(xs,0,(result,x) -> result + x.length());
}
```

Konkatenation aller Strings mit Leerzeichen Sollen alle Strings in einem String zusammen gehängt werden, so starten wir mit dem leeren String. Die Funktion hängt jeweils das aktuelle Elemente mit einem Leerzeichen getrennt an das Ergebnis an.

Reduction.java

```
public static String apF(Iterable<String> xs) {
  return fold( xs, " ", (result,x) ->result + " " + x);
}
```

Suche nach längsten aller Strings Bei der Suche nach dem längsten String, starten wir auch mit einem leeren String. In der Funktion wird geschaut, ob das aktuelle Element länger als das bisherige Ergebnis ist. Das längere von beiden wird das neue Ergebnis.

Reduction.java

```java
public static String loF(Iterable<String> xs) {
  return fold( xs, "", (result,x)->result.length()>x.length()?result:x);
}
```

Test, ob ein String enthalten ist Bei dem Test, ob ein String im Iterationsbereich enthalten ist, starten wir mit `false` für das Ergebnis. In der Funktion vergleichen wir das aktuelle Element mit dem gesuchten und verodern das Ergebnis mit dem bisherigen Ergebnis.

Reduction.java

```java
public static boolean coF(String y,Iterable<String> xs) {
  return fold( xs, false, (result,x)-> result || x.equals(y));
}
```

2 Aufgaben

Aufgabe 1 Schreiben Sie in dieser Aufgabe alle Funktionen, indem Sie einen Aufruf der Funktion `fold` machen.

a) Schreiben Sie eine Funktion die die Summe aller Elemente berechnet.

Reduction.java

```java
public static int sum(Iterable<Integer> xs) {
  return 0;
}
```

Ein Beispielaufruf:

Shell

```
jshell> import static name.panitz.util.Reduction.*

jshell> sum(List.of(1,2,3,4))
$2 ==> 10
```

b) Schreiben Sie eine Funktion die das Produkt aller Elemente berechnet.

Reduction.java

```
public static long product(Iterable<Long> xs) {
  return 0;
}
```

Ein Beispielaufruf:

Shell

```
jshell> product(List.of(1L,2L,3L,4L))
$3 ==> 24
```

c) Schreiben Sie eine Funktion die das größte Element berechnet.

Reduction.java

```
public static long maximum(Iterable<Long> xs) {
  return 0;
}
```

Ein Beispielaufruf:

Shell

```
jshell> maximum(List.of(1L,2L,33L,4L))
$4 ==> 33
```

Aufgabe 2 Schreiben Sie auch in dieser Aufgabe alle Funktionen, indem Sie einen Aufruf der Funktion `fold` machen.

a) Schreiben Sie eine Funktion, die wahr ist, wenn mindestens ein Element der Iteration das Prädikat erfüllt.

Reduction.java

```
public static <E>
boolean exists(Iterable<E> xs, Predicate<? super E> p) {
  return false;
}
```

Ein Beispielaufruf:

Shell

```
jshell> exists(List.of(1L,2L,3L,4L),x->x==3L)
$5 ==> true
```

b) Schreiben Sie eine Funktion, die wahr ist, wenn alle Elemente der Iteration das Prädikat erfüllen.

Reduction.java

```
public static <E>
boolean all(Iterable<E> xs, Predicate<? super E> p) {
  return false;
}
```

Ein Beispielaufruf:

Shell

```
jshell> all(List.of(1L,2L,3L,4L),x->x<=4L)
$6 ==> true
```

c) Schreiben Sie eine Funktion, die alle Elemente, die das Prädikat erfüllen, in einer Menge sammeln.

Reduction.java

```
public static <E>
Set<E> collect(Iterable<E> xs, Predicate<? super E> p) {
  return Set.of();
}
```

Ein Beispielaufruf:

Shell

```
jshell> collect(List.of(1L,2L,3L,4L,4L,4L),x->x%2L==0)
$8 ==> [2, 4]
```

Aufgabe 3 Schreiben Sie auch in dieser Aufgabe alle Funktionen, indem Sie einen Aufruf der Funktion `fold` machen.

a) Gegeben sei eine Funktion, die einen String als Liste von Character darstellt:

Reduction.java

```java
public static List<Character> toList(String str) {
  return str.chars()
      .mapToObj(c -> (char) c).collect(Collectors.toList());
}
```

Schreiben Sie mit Hilfe von `fold` eine Funktion, die den String als Dualzahl einliest.

Reduction.java

```java
public static int readBinary(String str) {
  return readBinary(toList(str));
}

public static int readBinary(Iterable<Character> xs) {
  return 0;
}
```

Ein Beispielaufruf:

Shell

```
jshell> readBinary("101010")
$2 ==> 42
```

b) Gegeben sei die Record-Klasse für Paare von Objekten:

Reduction.java

```java
public static record Pair<A,B>(A fst,B snd){}
```

Desweiteren sei gegeben eine Funktion, die Zeichen als Ziffer der römischen Zahlen erkennt.

Reduction.java

```java
public static int romanDigit(char c){
  return switch(c){
    case 'I' -> 1;
    case 'V' -> 5;
    case 'X' -> 10;
    case 'L' -> 50;
    case 'C' -> 100;
```

```
      case 'D' -> 500;
      case 'M' -> 1000;
      default ->
          throw new RuntimeException("Illegal roman digit: "+c);
    };
  }
```

Schreiben Sie mit Hilfe von `fold` eine Funktion, die den String als römische Zahl einliest. Dabei benötigen Sie den Kontext der zuletzt gelesenen Ziffer. Deshalb ist das Ergebnis der eigentlichen Funktion ein Paar, deren erste Komponente das bisher gelesenen Gesamtergebnis darstellt und deren zweite Komponente den Wert der zuletzt gelesenen römischen Ziffer darstellt.

Reduction.java

```
public static int readRoman(String str) {
  return readRoman(toList(str)).fst();
}

public static
Pair<Integer,Integer> readRoman(Iterable<Character> xs) {
  return null; /* ToDo */
}
```

Ein Beispielaufruf:

Shell

```
jshell> readRoman("CMXCIX")
$2 ==> 999

jshell> readRoman(toList("CMXCIX"))
$3 ==> Pair[fst=999, snd=10]
```

Reduction.java

```
}
```

Fold
Reduce