

# Kapitel 9

## Induktive Datenstrukturen in Java

In Scheme dienen Listen als universelle Datenstruktur für die Unterbringung von „Ansammlungen“ anderer Werte. In Java hat in den Beispielen der vergangenen Kapitel die Java-Klasse `Vector` gedient. Vektoren und Arrays sind *flache* Datenstrukturen, deren Einsatzgebiet im wesentlichen auf Anwendungen beschränkt ist, in denen die Anzahl der Elemente im voraus bekannt sind.

In der praktischen Software-Entwicklung bieten sich darum häufig induktive Datenstrukturen wie Listen und Bäume für effiziente Algorithmen an. Außerdem ergeben sich in der Datenmodellierung häufig ganz von allein induktive Strukturen. Die naive Implementation induktiver Datenstrukturen läßt sich durch die Anwendung eines Patterns deutlich verbessern, das damit auch das Hauptthema dieses Kapitels darstellt: das *Visitor-Pattern*. Es ist das Pendant zu `fold` in Scheme aus der Informatik I.

### 9.1 Listen als induktive Datenstruktur

Die populärste induktive Datenstruktur ist in der Tat die Liste. Zur Erinnerung sei hier die induktive Definition aus der Informatik I wiederholt:

- Die leere Liste ist eine Liste.
- Falls  $l$  eine Liste und  $v$  ein beliebiger Wert ist, so ist das Paar mit  $v$  als `car` und  $l$  als `cdr` ebenfalls eine Liste.
- Nichts sonst ist eine Liste.

Es gibt in dieser Definition drei Größen: *Liste*, *leere Liste* und *nichtleere Liste*. Es gibt verschiedene Methoden, diese drei Größen auf eine oder mehrere Java-Klassen abzubilden. Die einfachste ist vielleicht die direkte Abbildung auf drei Klassen — eine für jede Größe. Der Oberbegriff *Liste* hat für sich — im Moment — noch keine besonderen Eigenschaften:

```
public abstract class List {  
}
```

Bei der leeren Liste ist es ähnlich — es ist im Moment hauptsächlich wichtig, sie von den nichtleeren unterscheiden zu können.

```
public class EmptyList extends List {  
}
```

Bei nichtleeren Liste lassen sich `car` und `cdr` unterscheiden:

```
public class ConsList extends List {
    private Object car;
    private List cdr;

    public ConsList(Object theCar, List theCdr) {
        car = theCar;
        cdr = theCdr;
    }
    public Object getCar() {
        return car;
    }
    public List getCdr() {
        return cdr;
    }
}
```

Das grundsätzliche Problem mit der Verwendung von Klassen zur Repräsentation des Datentyps ist, daß es eine große Sammlung von praktischen Operationen auf Listen gibt; eine gegebene Anwendung wird sich eine ganze Sammlung davon zulegen. In der objektorientierten Programmierung ist es üblich, Operationen, die auf einem Datentyp arbeiten, als Methoden in der entsprechenden Klasse unterzubringen. Dies ist bei Listen jedoch oft nicht praktikabel, da Anwendungen oft spezielle Listenoperationen benötigen, die der Implementierer der Listen-Klasse nicht vorhersehen konnte.

Zum Glück lassen sich Operationen über Listen als ganz normale Klassenmethoden schreiben. Hier eine solche Methode, welche alle Zahlen einer Liste (repräsentiert als `Integer`-Objekte) addiert:

```
public static int listSum(List list) {
    if (list instanceof EmptyList)
        return 0;
    else {
        ConsList consList = (ConsList) list;
        return
            ((Integer) consList.getCar()).intValue() +
            listSum(consList.getCdr());
    }
}
```

Der Operator `instanceof` testet, ob ein Objekt einer bestimmten Klasse oder einer ihrer Unterklassen angehört: falls ja, liefert `instanceof` den Wert `true`, sonst `false`. Hier eine Klassenmethode, die zwei Listen aneinanderhängt:

```
public static List appendLists(List list1, List list2) {
    if (list1 instanceof EmptyList)
        return list2;
    else {
        ConsList consList1 = (ConsList) list1;
        return new ConsList(consList1.getCar(),
            appendLists(consList1.getCdr(), list2));
    }
}
```

Es ist bereits das typische Muster erkennbar, welches die meisten Prozeduren über Listen gemeinsam haben. Hier noch eine praktische Methode, welche die Elemente einer Liste ausdrückt:

```

public static void printList(List list) {
    if (list instanceof EmptyList)
        ;
    else {
        ConsList consList = (ConsList) list;
        System.out.println(consList.getCar());
        printList(consList.getCdr());
    }
}

```

Hier ein Beispielprogramm, welches die Verwendung der List-Klassen demonstriert:

```

public static void main(String[] args) {
    List a =
        new ConsList(new Integer(3),
                    new ConsList(new Integer(5),
                                new ConsList(new Integer(7),
                                            new EmptyList())));
    List b =
        new ConsList(new Integer(14),
                    new ConsList(new Integer(23),
                                new EmptyList()));

    System.out.println("a:");
    printList(a);
    System.out.println("b:");
    printList(b);
    System.out.println("sum(a):");
    System.out.println(listSum(a));
    System.out.println("append(a, b):");
    printList(appendLists(a, b));
}

```

Das Programm erzeugt folgende Ausgabe:

```

a:
3
5
7
b:
14
23
sum(a):
15
append(a, b):
3
5
7
14
23

```

## 9.2 Besucher für Listen

Die drei Klassenmethoden `listSum`, `listAppend` und `printList` folgen alle demselben Muster:

```

public static List op(List list, ...) {
    if (list instanceof EmptyList)
        return ...;
    else {
        ConsList consList = (ConsList) list;
        return ...consList.getCar() ...
                op(consList.getCdr(), ...) ...;
    }
}

```

Es stellt sich die Frage, ob dieses Muster nicht programmatisch festgehalten werden kann, und dabei den `instanceof`-Test sowie die Aufrufe von `getCar` und `getCdr` überflüssig machen. Dies ist in der Tat möglich mit Hilfe eines Patterns, das bei der Programmierung einer Klasse für eine induktive Datenstruktur angewendet wird: das *Visitor-Pattern*. Es beruht auf der Annahme, daß eine Methode, die eine Berechnung über eine Liste durchführt, jede Element der Liste „besuchen“ will. Der Ablauf des „Besuchens“ wird dabei in den Klassen festgelegt, welche den Datentyp repräsentieren.

Es ist am einfachsten, erst einmal eine der einfachen Operationen näher zu betrachten. Zur Erinnerung noch einmal der Code der Summierungs-Methode:

```

public static int listSum(List list) {
    if (list instanceof EmptyList)
        return 0;
    else {
        ConsList consList = (ConsList) list;
        return
            ((Integer) consList.getCar()).intValue() +
            listSum(consList.getCdr());
    }
}

```

Die Konsequente und Alternative des `ifs` lassen sich einzeln betrachten und auf zwei Methoden einer neuen Klasse `SumCalculator` verteilen, eine, die leere Listen aufsummiert, und eine für nichtleere Listen, die einen `car` und einen `cdr` besitzen:

```

public class SumCalculator {
    public int sumEmptyList() {
        return 0;
    }
    public int sumConsList(Object car, List cdr) {
        return ((Integer) car).intValue() + cdr.sumFrom(this);
    }
}

```

Ausgangspunkt für den Besuch aller Elemente einer Liste ist eine Methode `sumFrom`, welche die passende Methode eines `SumCalculator`-Objekts aufruft:

```

public abstract class List {
    public abstract int sumFrom(SumCalculator calc);
}

```

Es ist jetzt Aufgabe der Implementationen von `sumFrom`, die passende Methode eines `SumCalculator` aufzurufen. Eine leere Liste hat keine Elemente, darum übergibt die `sumFrom`-Implementation in `EmptyList` auch keine Parameter:

```
public class EmptyList extends List {
    public int sumFrom(SumCalculator calc) {
        return calc.sumEmptyList();
    }
}
```

Eine nichtleere Liste besteht aus den Bestandteilen `car` und `cdr`, die an den `SumCalculator` übergeben werden:

```
public class ConsList extends List {
    public int sumFrom(SumCalculator calc) {
        return calc.sumConsList(car, cdr);
    }
}
```

Diese Lösung hat jetzt kaum Vorteile gegenüber der `listSum`-Klassenmethode. Immerhin ist der Code, welcher die Struktur der Liste abläuft, getrennt von dem, in dem die tatsächliche Additionsoperation steht. Prinzipiell könnte also in `SumCalculator` (oder einer Unterklasse) die 0 durch eine 1 sowie das + durch ein \* ausgewechselt werden, und schon würde `sumFrom` nicht aufsummieren sondern aufmultiplizieren. Drei Einschränkungen verhindern allgemeinere Einsetzbarkeit:

- `SumCalculator` ist eine Klasse und kein Interface, obwohl potentiell verschiedene Listenoperationen `sumEmptyList`- und `sumConsList`-Methoden haben könnten, die nichts miteinander zu tun haben.
- Das Wort „sum“ suggeriert, daß immer summiert wird.
- Der Rückgabotyp der Listenoperationen ist auf `int`-Zahlen beschränkt

Diese Einschränkungen lassen sich leicht beseitigen; aus dem Prinzip des „Listen-Summatoren“ wird dann ein Pattern: Der Listen-Summatoren wird dabei zu einem allgemeinen Besucher, der eine beliebige Operation über die Listenoperationen ausführt: ein solcher Besucher heißt *Visitor*. Da es viele Visitors für Listen geben kann und wird, legt ein Interface fest, was einen Visitor ausmacht. Dabei wird im wesentlichen der Rückgabotyp `int` verallgemeinert:

```
public interface ListVisitor {
    Object visitedEmptyList();
    Object visitedConsList(Object car, List cdr);
}
```

Solch ein Visitor wird von einer Methode `visitedFrom` in den Listenklassen aktiviert:

```
public abstract class List {
    public abstract Object visitFrom(ListVisitor visitor);
}
```

Die passenden Implementationen in `EmptyList` und `ConsList` sind fast identisch mit den `sumFrom`-Methoden aus:

```
public class EmptyList extends List {
    public Object visitFrom(ListVisitor visitor) {
        return visitor.visitedEmptyList();
    }
}
```

```
public class ConsList extends List {
    ...
    public Object visitFrom(ListVisitor visitor) {
        return visitor.visitedConsList(car, cdr);
    }
}
```

Nun läßt sich unter diesem Interface ein Visitor programmieren, der Listenelemente aufsummiert:

```
public class SumListVisitor implements ListVisitor {
    public Object visitedEmptyList() {
        return new Integer(0);
    }
    public Object visitedConsList(Object car, List cdr) {
        return
            new Integer(((Integer) car).intValue() +
                ((Integer) (cdr.visitFrom(this))).intValue());
    }
}
```

Für die Benutzung wird ein `SumListVisitor`-Objekt erzeugt und an die `visitFrom`-Methode der aufzusummierenden Liste übergeben, etwa so:

```
Integer sum = a.visitFrom(new SumListVisitor());
```

Das Aneinanderhängen zweier Listen funktioniert nach dem gleichen Prinzip. Der entscheidende Trick ist es, die anzuhängende Liste in einer Instanzvariable unterzubringen:

```
public class AppendListVisitor implements ListVisitor {
    private List list2;

    public AppendListVisitor(List theList2) {
        list2 = theList2;
    }
    public Object visitedEmptyList() {
        return list2;
    }
    public Object visitedConsList(Object car, List cdr) {
        return new ConsList(car,
            (List) cdr.visitFrom(this));
    }
}
```

Auch hier muß für die Benutzung ein `AppendListVisitor`-Objekt erzeugt werden:

```
List c = (List) a.visitFrom(new AppendListVisitor(b));
```

### 9.3 Binärbäume mit dem Visitor-Pattern

Der nächste Kandidat für eine zu realisierende induktive Datenstruktur ist der Binärbaum. Hier wird ein Binärbaum implementiert, der entweder ein Blatt oder ein Knoten sein kann:

- Knoten haben hier eine `int`-Zahl als Markierung und sowohl einen linken und rechten Teilbaum.

- Ein Blatt ist „leer“, besitzt also weder Markierung noch Teilbäume.

```
public abstract class BinTree {
    public abstract Object visitFrom(BinTreeVisitor v);
}
```

Der passende Visitor hat zwei Methoden, eine für Blätter und eine für Knoten:

```
public interface BinTreeVisitor {
    public Object visitedLeaf();
    public Object visitedNode(int val, BinTree left, BinTree right);
}
```

Hier die Klassen Leaf und Node:

```
public class Leaf extends BinTree {
    public Object visitFrom(BinTreeVisitor visitor) {
        return visitor.visitedLeaf();
    }
}

public class Node extends BinTree {
    private int val;
    private BinTree left;
    private BinTree right;

    public Node(int theVal, BinTree theLeft, BinTree theRight) {
        val = theVal;
        left = theLeft;
        right = theRight;
    }

    public Object visitFrom(BinTreeVisitor visitor) {
        return visitor.visitedNode(val, left, right);
    }
}
```

Mit den Klassenconstructoren lassen sich Bäume annähernd durch ihre Termschreibweise konstruieren:

```
BinTree sample =
new Node(7, new Node(3, new Node(1,
                            new Leaf(),
                            new Leaf()),
            new Node(5,
                    new Node(4,
                            new Leaf(),
                            new Leaf()),
                    new Leaf()),
            new Node(8,
                    new Leaf(),
                    new Leaf()));
```

Ein einfacher Visitor druckt einen Binärbaum in Termschreibweise aus:

```
public class BinTreePrinter implements BinTreeVisitor {
    private String indent = "";
```

```

public Object visitedLeaf() {
    System.out.println(indent + "Leaf()");
    return null;
}
public Object visitedNode(int val, BinTree left, BinTree right) {
    System.out.println(indent + "Node( " + val);
    String saveIndent = indent;
    indent = indent + " ";
    left.visitFrom(this);
    right.visitFrom(this);
    indent = saveIndent;
    System.out.println(indent + ")");
    return null;
}
}
}

```

Der oben konstruierte Baum lässt sich mit

```
sample.visitFrom(new BinTreePrinter());
```

ausdrucken. Es erscheint folgende Ausgabe:

```

Node( 7
  Node( 3
    Node( 1
      Leaf()
      Leaf()
    )
    Node( 5
      Node( 4
        Leaf()
        Leaf()
      )
      Leaf()
    )
  )
)
Node( 8
  Leaf()
  Leaf()
)
)

```

## 9.4 Suchbäume mit Visitors

Eine Hauptanwendung für Binärbäume ist deren Verwendung als *Suchbäume*. Zur Erinnerung: ein Suchbaum ist ein Baum, in dem bei jedem Knoten der linke Teilbaum nur Markierungen enthält, die kleiner als die Markierung des Knotens selbst sind, und in dem der rechte Teilbaum nur Markierungen enthält, die größer sind. Diese Invariante lässt sich ausnutzen, um effizient nach einer Zahl in einem Suchbaum zu suchen:

```

public class BinTreeSearcher implements BinTreeVisitor {
    private int number;

    public BinTreeSearcher(int theNumber) {

```

```

        number = theNumber;
    }
    public Object visitedLeaf() {
        return new Boolean(false);
    }
    public Object visitedNode(int val, BinTree left, BinTree right) {
        if (val == number)
            return new Boolean(true);
        else if (number < val)
            return left.visitFrom(this);
        else
            return right.visitFrom(this);
    }
}

```

So läßt sich folgendermaßen feststellen, ob etwa die Zahl 4 in einem binären Suchbaum steckt:

```
boolean t = ((Boolean) sample.visitFrom(new Searcher(4))).booleanValue();
```

Nach dem gleichen Prinzip läßt sich auch die Einfügeoperation als Visitor realisieren:

```

public class BinTreeInserter implements BinTreeVisitor {
    private int number;

    public BinTreeInserter(int theNumber) {
        number = theNumber;
    }

    public Object visitedLeaf() {
        return new Node(number, new Leaf(), new Leaf());
    }
    public Object visitedNode(int val, BinTree left, BinTree right) {
        if (number < val) {
            BinTree newTree = (BinTree) (left.visitFrom(this));
            return new Node(val, newTree, right);
        } else {
            BinTree newTree = (BinTree) (right.visitFrom(this));
            return new Node(val, left, newTree);
        }
    }
}

```

Ein Suchbaum (der zufällig identisch ist zu dem obigen Demo-Baum `sample`) läßt sich z.B. folgendermaßen konstruieren:

```

int[] numbers = {7, 8, 3, 1, 5, 4};
BinTree sample = new Leaf();
for (int i = 0; i < numbers.length; i = i+1 ) {
    sample = (BinTree) sample.visitFrom(new Insert(numbers[i]));
}

```