Kapitel 9

Inner Classes

9.1 Wiederholung: Iteratoren

Ausführen einer Operation auf allen Elementen einer Containerklasse (zB Liste, Baum, ...)

vgl. $map/fold$ in der funktionalen Programmierung. Aber: higher-order Funktionen gibt es in Java/C++ nicht

Java 5 hat dazu neue Syntax. Bsp:

```
LinkedList<Point> polygon;
for (Point p: polygon) {
   ... p ...
}
```

Wird intern in die alten Iteratorklassen übersetzt, die wiederum Anlass gaben für innere Klassen

Iteratorklassen: haben Methoden für Initialisierung, nächstes Objekt sowie Test, ob noch Objekte vorhanden

Momentane Position des Iterators ist in einem *Iteratorobjekt* gekapselt („Laufvariable“)

JAVA hat Schnittstelle `Iterator` (in `java.util`)
Verwendungsschema:

```
import java.util.Iterator;
  // public abstract boolean hasNext();
  // public Object next();
DlList l;
for (Iterator e = l.iterator(); e.hasNext(); ) {
  Object o = e.next();
  Egg o1 = (Egg) o; // nun verarbeite o1
}
```

Iteratorschnittstelle muß für jede Containerklasse implementiert werden,

Containerklasse muss Methode zum Erzeugen des Iteratorobjektes (“`iterator`”) enthalten

Grund: “`iterator`” muss Implementierung des Containers kennen

Beispiel: zyklische doppelt verkettete Liste mit Kopf

```
class ListNode {
  private Object data; // veraltet in Java 5 !
  private ListNode prior, next;
  protected ListNode(Object d) {data = d;}
  protected Object getData() {return data;}
  protected ListNode getNext() {return next;}
  protected Listnode getPrior() {return prior;}
  protected void setNext(ListNode l) {next = l;}
  protected void setPrior(Listnode l) {prior=l;}
}
class DlList {
  private ListNode root;
  public DlList() {
    root = new ListNode(null);
    root.setNext(root);
    root.setPrior(root);
  }
  public void insert(Listnode l) {
    l.setNext(root.getNext());
    l.setPrior(root);
    root.setNext(l);
    (l.getNext()).setPrior(l);
  }
  public void delete(Listnode l) { ... }
  ...
// Methode zur Initialisierung des Iterators
// muss Implementierung von DlList kennen
  public ListIterator iterator() { // ListIterator s.u.
    ListIterator e = new ListIterator();
    e.position = root; e.root = root;
    return e;
  }
}
```

nun: Extra-Klasse für den Iterator. geschachtelte Iteratoren via mehrere Iteratorobjekte + copy-Methode

```
class ListIterator implements Iterator {
  protected ListNode position;
  protected ListNode root;
  public boolean hasNext()
    { return position.getNext() != root; }
  public Object next() {
    position = position.getNext();
    return position.getData()
  }
  public ListIterator copy()  {
    ListIterator c = new ListIterator();
    c.root = this.root; c.position = this.position;
    return c;
  }
}
```

doppelte Schleife über eine Liste:

```
DlList l;
for (ListIterator e1 = l.iterator(); e1.hasNext(); ) {
  Object o1 = e1.next();
  ... (ListElement) o1 ... // bearbeite o1
  for (ListIterator e2 = e1.copy(); e2.hasNext(); ) {
    Object o2 = e2.next();
    ... (ListElement) o2 ... // bearbeite o1, o2
  }
}
```

Alternativen für Iteratoren:

1. zwei separate Klassen, `iterator` in Hauptklasse. Nachteil: enger Zusammenhang zwischen Container und Iterator ist nur indirekt sichtbar ($\rightsquigarrow$ Kohäsion)
2. Friend-Klassen (C++). Nachteil: Geheimnisprinzip wird durchbrochen
3. Inner Classes (Java)

9.2 Inner Classes

- Klasse `ListNode` sollte nach außen unsichtbar sein
- `ListIterator` muss nach aussen sichtbar sein, muss aber `ListNode` kennen

⇒ *Inner Classes*

- *statische* innere Klassen: wie gewöhnliche Klassen, aber nach außen unsichtbar; strukturieren ansonsten gleichrangige Klassen

  Äußere Klasse kann auf statische (!) Members der inneren zugreifen, innere Klasse kann auf statische (!) Members der äußeren zugreifen.

  Objekte statischer innnerer Klassen sind ganz normal, können aber nicht auf dynamische Instanzvariablen der äußeren Klasse zugreifen

  Beispiel: `ListNode` sollte statische innere Klasse von `DlList` sein

  Genaue Sichtbarkeit von inneren Klassen kann noch mit Schutzrechten (zB **`protected`**) gesteuert werden

  Statische innere Klassen wie `Listnode` sollten **`private`** oder **`protected`** sein

- *dynamische* innere Klassen: jedes Objekt hat Verweis auf zugeordnetes Objekt der äußeren Klasse `Outer.this`; wird von Konstruktorfunktion angelegt

  jedes `Inner`-Objekt kann auf Members des (impliziten) `Outer`-Objekts zugreifen und umgekehrt

  Typ der inneren Klassen von außen ansprechbar (wenn nicht als private deklariert)

  Achtung: Inner-Objekt kann auch auf private-Instanzvariablen des Outer-Objektes zugreifen (vgl. Friend-Klassen)

  Beispiel:

```
class A {
  int x;
  class B {
     void f(){ ... Outer.this.x ...}
     ...
  }

  void g(A a) { B b = a.new B(); ...}
}

A.B b = this.new B();
...
```

Iteratorimplementierung mit Inner Classes:

```
class DlList {
  private ListNode root;
  static private class ListNode { //statisch!
     ... Rumpf wie oben ...
  }
  ... weitere members von Dllist wie oben ...
  public class ListIterator implements Iterator {
  //dynamisch!
     private position;
     // eigenes root nicht mehr notwendig
     ... Methoden wie oben ... DlList.this ...
  }
  ListIterator iterator() {
    return new ListIterator();
  }
} ...
DlList l;
for (Iterator i = l.iterator();
                 i.hasNext(); ) {
   o = i.next(); ...
}
```

($\rightsquigarrow$ Übung. Vgl `LinkedList` in `java.util`)