Programmieren II

Dr. Klaus Höppner

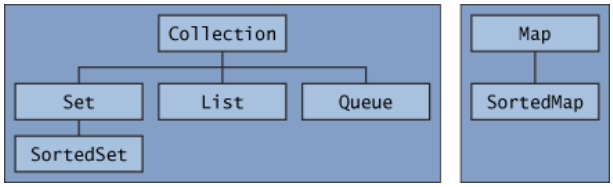
Hochschule Darmstadt – Sommersemester 2010

Hierarchie von Collections

Die *sequenziellen Container* (`ArrayList` & `LinkedList`) und *Mengen* implementieren das Interface `Collection`.

Die *assoziativen Container* werden zwar auch häufig zu den Collections gezählt, sind aber tatsächlich unabhängig vom Interfaxe `Collection`, sondern implementieren in Wirklichkeit das Interfax `Map`.

Schaubild



Das Interface `Collection<E>`

Ein großer Teil der Methoden aus dem Interface `List` der letzten Vorlesung stammen bereits aus `Collection`

`boolean add(E e)` fügt ein Objekt `e` hinzu.

`void clear()` löscht alle Werte aus der COllection.

`boolean contains(Object o)` prüft, ob es ein Element `e` gibt, für das der Ausdruck `e.equals(o)` wahr ist.

`boolean isEmpty()` ist wahr, wenn die COllection leer ist.

`Iterator<E> iterator()` gibt einen `Iterator` zurück. Dies ist ein bisher noch nicht behandeltes Sprachelement, das im Weiteren näher behandelt wird.

Das Interface `Collection<E>`

`boolean remove(Object o)` entfernt das erste Element, das zu `o` äquivalent ist. **`true`**, falls ein Element entfernt wurde.

`int size()` Anzahl der Elemente.

`Object[] toArray()` gibt alle Elemente der COllection als Array vom Typ `Object[]` zurück.

`T[] toArray(T[] a)` füllt den übergebenen Array vom Typ `T[]` mit allen Elementen oder erzeugt einen neuen, wenn die Größe des Array nicht ausreicht. Rückgabewert ist der gefüllte Array.

Grundfunktionalität von Collections

Beim Betrachen der Methoden werden die wesentlichen Gemeinsamkeiten aller Collections deutlich:

- Hinzufügen eines Objekts (`add`),
- Entfernen eines Objekts (`remove`),
- Prüfen, ob ein Objekt enthalten ist (`contains`)

Indexbezogene Operationen kommen erst in `List` hinzu.

Frage:
Aber wie kann man z. B. alle Elemente einer Collection ausgeben?

Iteratoren – Motivation

Betrachten zunächst folgendes Beispiel:

```
List<Double> l = new LinkedList<Double>();
// Füllen
double summe = 0;
for (int i=0; i<l.size(); i++) {
    summe += l.get(i).doubleValue();
}
```

Wie ist das Laufzeitverhalten der **for**-Schleife?
Die Zugriffszeit auf das Element i ist proportional zu i, die Laufzeit der Iteration daher proportional zu $\sum_{i=0}^{n-1} i$. Das Ergebnis dieses Ausdruck verhält sich quadratisch, also ist die Iteration von der Ordnung n^2 (hierbei ist n die Länge der Liste).

Alternative Fassung

Vergleichen wir das Verhalten der Summation über Indexzugriff mit folgender Fassung:

```
List<Double> l = new LinkedList<Double>();
// Füllen
double summe = 0;
for (Double d : l) {
    summe += d.doubleValue();
}
```

Hier stellt man fest, dass die Laufzeit deutlich besser ist, denn in der **for**-Schleife wird tatsächlich von Element zu Element gesprungen (während bei der Summation über Index bei jedem Schleifendurchlauf das passende Element jeweils neu „abgezählt“ wird). Die Laufzeit ist somit insgesamt von der Ordnung n, wächst also linear mit der Länge der Liste.

Iteratoren

Tatsächlich wird der Code `for (Double d : l) {...}` intern wie folgt umgesetzt:

```
for (Iterator<Double> iter=l.iterator(); iter.hasNext(); ) {
    Double d = iter.next();
    // ...
}
```

Ein *Iterator* ist dabei ein *Zeiger*, der über alle Elemente einer Collection iterieren kann. Dieser besitzt die Methoden

`bool hasNext()` Gibt es ein nachfolgendes Element?

`E next()` holt den Wert des jeweils nächsten Elements.

`void remove()` entfernt das zuvor mit `next()` geholte Element aus der darunterliegenden Collection (optional).

In der bildlichen Vorstellung zeigt der Iterator immer *zwischen* zwei Elemente!

Anwendung

Folgendes Programm „simuliert“ die Ziehung der Lottozahlen (als Vorgeschmack auf Sets):

```
Collection<Integer> c = new TreeSet<Integer>();
c.add(43);
c.add(7);
c.add(2);
c.add(29);
c.add(33);
c.add(12);

for (Iterator<Integer> iter=c.iterator(); iter.hasNext(); ) {
    Integer zahl = iter.next();
    System.out.println(zahl);
}
```

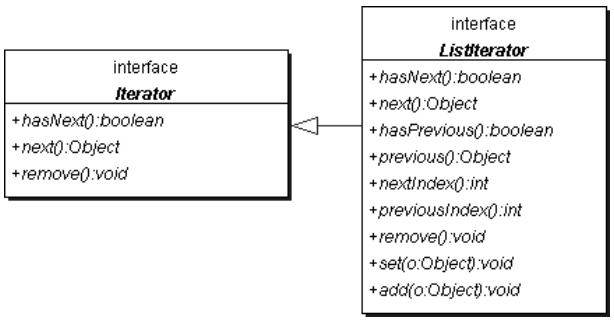

Iterator – ListIterator

Der normale Iterator hat zwei Nachteile:

- Die Iteration ist nur in eine Richtung möglich.
- Über den Iterator können keine Elemente der darunter liegenden Collection ausgetausch werden.

Für das Interface `List` gibt es einen erweiterten Iterator, den `ListIterator`. Dieser erlaubt Iterationen in beide Richtungen, Zugriff auf den Index eines Elementes und das Austauschen von Elementen.

Schaubild von ListIterator<E>



Quelle: Ullenboom, Java ist eine Insel, 7. Aufl.

Das Interface ListIterator<E>

Für `ListIterator<E>` kommen folgende Methoden hinzu:

`boolean hasPrevious()` Gibt es ein Vorgängerelement?

`int nextIndex()` Index des nächsten Elements.

`E previous()` holt das Vorgängerelement.

`int previousIndex()` Index des vorigen Elements.

`void set(E e)` ersetzt das beim letzten `next()` oder `previous()` geholte Element durch `e`.

Beispiel

Im folgenden Beispiel werden in einer String-Liste alle Vorkommen von *Klaus* durch *Otto* ersetzt:

```
List<String> l = new LinkedList<String>();
// Füllen
ListIterator<String> iter;
for (iter=l.listIterator(); iter.hasNext(); ) {
    String wert = iter.next();
    if (wert=="Klaus") {
        iter.set("Otto");
        System.out.println("Element an Index "+
                iter.previousIndex()+" ersetzt");
    }
}
```

Mengentypen: Set

Ein `Set` (deutsch: Menge) in Java ist ein Interface, das von `Collection` erbt. Im Gegensatz zu `List` kommen hier aber keine neuen Methoden hinzu.

Eine Implementation von `Set` unterstützt daher im Wesentlichen folgende Operationen:

- Hinzufügen neuer Element
- Entfernen von Elementen
- Prüfen darauf, ob ein Element vorhanden ist

Hierbei gilt, dass ein einem `Set` niemals doppelte Einträge gibt, also für alle Paare *e1* und *e2* der Ausdruck `e1.equals(e2)` unwahr ist.

Sets und Sortierung

Wie bei allen Collections ermöglicht die Methode `iterator()` eine Iteration über alle Elemente des Sets, was z. B. implizit benutzt wird durch:

```
for (E e : mySet) { ... }
```

Normale Sets stellen dabei keinen Anspruch auf die Reihenfolge der Elemente bei der Iteration. Diese muss keiner Ordnung unterliegen (wie aufsteigend), noch muss diese reproduzierbar sein.

Demgegenüber stellt das Interface `SortedSet` sicher, dass die Elemente immer anhand einer festen (vorgegebenen) Ordnung sortiert sind.

Die Implementierung HashSet

Häufig wird für Mengen die Implementierung `HashSet` verwendet.

Der grundsätzliche Aufbau soll durch eine einfache Analogie verdeutlicht werden. In der CD-Abteilung eines Kaufhauses werden die CDs nach folgendem System geordnet:

- Das Sortiment ist in Regale für einzelne Musikrichtungen unterteilt,
- im Regal einer Musikrichtung gibt es (je nach Bedeutung des/der Interpreten) eine Unterteilung nach einzelnen Interpreten bzw. einer Gruppe von Interpreten mit gleichem Anfangsbuchstaben,
- innerhalb der Unterteilung sind die CDs ungeordnet.

Beispiel

Beispiel 1: U2, The Joshua Tree findet man unter Rock & Pop, dort unter U2, dann muss man alle CDs von U2 durchsuchen, bis man die richtige gefunden hat.

Beispiel 2: Mambo Kurt, Back in Beige unter Rock & Pop, dort unter „M diverse“, und dort durchblättern, bis man die CD findet (sehr unwahrscheinlich!)

Diese Ordnung entspricht einer Sortierung nach *Hashes*, hier auf zwei Ebenen: Kategorie und Interpret. Die Hashes engen die Suche ein, innerhalb der ungeordneten Menge von CDs mit gleichen Hashes hilft aber nur lineare Suche.
Die Aufteilung nach Hashes ist dabei dynamisch. Veröffentlicht ein Interpret viele CDs, kriegt er irgendwann sein eigenes Fach!

Technische Funktionsweise von HashSet<E>

`HashSet<E>` unterteilt die Elemente (anhand einer Hash-Funktion) in n verschiedene Kategorien, mit den entsprechenden Indizes $0 \ldots n-1$, und ordnet jedem dieser Indizes ein *Bucket* (Kübel) zu. Hierin liegen die Elemente mit gleicher Kategorie ungeordnet.

Die Hash-Funktion greift dabei auf die Methode `int hashCode()` aus `Object` zurück. Besondere Eigenschaft hierbei ist, dass zunächst die Unterteilung der Hashwerte dabei sehr grob ist (wenig Buckets), und bei wachsender Zahl von Elementen die Unterteilung nach Hashwerten verfeinert wird (mehr Buckets), um die Zahl der Elemente in einem Bucket zu begrenzen. Dies nennt man *Rehashing*.

Zustandsparameter eines HashSet

Der interne Zustand eines `HashSet<E>` wird durch zwei Parameter beschrieben:

Kapazität (*capacity*) gibt die Zahl der Buckets an, d. h. je größer die Kapazität, desto feiner die Unterteilung in Hash-Kategorien.

Füllfaktor (*load factor*) gibt das aktuelle Verhältnis der Gesamtzahl der Elemente zur Kapazität an. Dabei ist immer die Bedingung

$$size \leq load\,factor \times capacity$$

erfüllt, der Füllfaktor bestimmt also, wann ein Rehashing (meist auf doppelte Kapazität) durchgeführt wird. Üblicher Wert für den Füllfaktor ist 0.75.

Laufzeitverhalten

Bezüglich des Laufzeitverhaltens verhalten sich Kapazität und Füllfaktor bei verschiedenen Operationen gegenläufig:

- Bei den Operationen `add`, `remove` und `contains` ist der Zugriff auf das passende Bucket von konstanter Laufzeit, das Durchsuchen der Elemente des Buckets (auch bei `add` zur Duplikatsprüfung) ist proportional zur mittleren Größe des Buckets ($Size/Capacity$), die durch den Load Factor nach oben begrenzt wird. Daher ist die Laufzeit im Prinzip *konstant* – unter der Annahme einer homogenen Verteilung der Elemente auf die Hashes –, wobei hier kleine Füllfaktoren vorteilhaft sind.
- Eine Iteration über alle Elemente braucht einen Iterationsschritt pro Bucket zzgl. einen je Element im Bucket, somit also
  $Capacity \times (Size/Capacity + 1) = Capacity + Size$