Objektorientierte Programmierung

Kapitel 21: Einführung in die Collection Klassen

Prof. Dr. Stefan Brass

Martin-Luther-Universität Halle-Wittenberg

Wintersemester 2018/19

http://www.informatik.uni-halle.de/~brass/oop18/

Inhalt

1. Einleitung
2. Collection Interface
3. Iteratoren
4. Listen
5. Ausblick

Einleitung und Motivation (1)

- Arrays erlauben es, mehrere Werte eines Typs in einer Datenstruktur abzulegen.
- Arrays haben aber eine feste Größe:
  - Deklariert man sie zu groß für die jeweilige Eingabe, verschwendet man Speicherplatz.
  - Deklariert man sie zu klein, kann man große Eingaben nicht bearbeiten.
- Natürlich gibt es Lösungen:
  - Verkettete Listen können dynamisch auf jede beliebige Länge wachsen.
  - Man kann bei Bedarf ein größeres Array anfordern und die Daten umkopieren.

Einleitung und Motivation (2)

- Verkettete Listen oder flexible Arrays braucht man recht häufig. Sie sind auch nicht ganz trivial zu programmieren.
- Deswegen gibt es sie schon in der Java Bibliothek (im Paket `java.util`).
- Die Typ-Parameter für generische Klassen wurden hauptsächlich eingeführt, um Datenstrukturen wie Listen typsicher anbieten zu können.
- Neben Listen gibt es weitere oft benötigte Datenstrukturen:
    - Mengen (im Unterschied zu Listen ohne Duplikate)
    - Abbildungen von Schlüsselwerten, z.B. Zeichenketten (Namen), auf Objekte eines beliebigen Typs.

Inhalt

1 Einleitung

2 Collection Interface

3 Iteratoren

4 Listen

5 Ausblick

Collection Interface (1)

- Eine “Collection”-Klasse ist eine, die mehrere Objekte eines Elementtyps zusammenfasst, z.B. Menge oder Liste.

  Etwas deutscher könnte man vielleicht von Container-Klassen reden. Es sind Behälter, in die man mehrere Objekte eines Typs tun kann, teils mit Reihenfolge oder Sortierung, teils ohne.

- Das Interface `Collection<E>` (im Paket `java.util`) definiert gemeinsame Operationen für alle Collection-Klassen.

  [http://docs.oracle.com/javase/7/docs/api/java/util/Collection.html]
  [http://docs.oracle.com/javase/7/docs/technotes/guides/collections/]

- Allerdings sind dabei alle Operationen, die die Collection verändern (also Elemente einfügen oder löschen) “optional”:
  - In einer Klasse, die das Interface implementiert, muss die Methode zwar vorgesehen sein, darf aber sofort eine “`UnsupportedOperationException`” auslösen.

Collection Interface (2)

- `boolean add(E e)`:
  Fügt das Element e in die Collection ein.

  Der boole'sche Ergebniswert ist in erster Linie für Mengen gedacht: Er zeigt an, ob die Menge durch die Einfügung verändert wurde (`true`), oder das Element schon vorher enthalten war, und deswegen die Menge unverändert bleibt (`false`). Die Methode garantiert (wenn sie keine Exception auslöst), dass das Element hinterher in der Collection enthalten ist.

- `boolean remove(Object o)`:
  Entfernt ein Exemplar des Objekts aus der Collection.

  Es wird dabei `o.equals(e)` zum Vergleich benutzt (bzw. falls `o==null` muss `e==null` sein). Falls so ein Element `e` gefunden wurde, wird es entfernt, und `true` geliefert. Falls die Collection kein solches Element enthielt, bleibt sie unverändert, und es wird `false` geliefert. Zur Frage, warum der Parameter `Object` und nicht `E` ist, siehe [Stack Overflow]: Z.B. weil die `equals`-Methode aus der Klasse von `o` kommt nicht unbedingt gleiche Klassen voraussetzt.

Collection Interface (3)

- `void clear()`:
  Alle Elemente aus der Collection entfernen.
- `int size()`:
  Liefert Anzahl Elemente in der Collection.
- `boolean isEmpty()`:
  Liefert `true` gdw. die Collection leer ist.
- `boolean contains(Object o)`:
  Liefert `true` gdw. die Collection das Element `o` enthält.

  Wie bei `remove` funktioniert der Vergleich mit `o.equals(e)` (oder beide null).
- `boolean containsAll(Collection<?> c)`:
  Testet, ob `c` eine Teilmenge dieser Collection ist.

  Auch von Einfügung und Löschung gibt es All-Varianten (inkl. Schnittmenge).

Collection Interface (4)

- Natürlich braucht man auch eine Zugriffsmöglichkeit für die Elemente der Collection:
  - In erster Linie gibt es hierfür die Methode
    `Iterator<E> interator()`
    die einen "Iterator" liefert, mit dem man eine Schleife über alle Elemente der Collection schreiben kann (s.u.).
  - `Collection<E>` bietet auch Methoden `toArray`, mit denen man die Collection in ein Array umwandeln kann.

    Aufgrund der in Kapitel 20 besprochenen Einschränkungen bekommt man entweder ein Array von Typ `Object[]`, oder muss ein Muster-Array vom richtigen Typ übergeben.
  - Spezielle Datenstrukturen, die das Interface `Collection<E>` erweitern (z.B. `List<E>`), fügen weitere Möglichkeiten zum Zugriff auf Elemente hinzu.

Inhalt

1. Einleitung
2. Collection Interface
3. Iteratoren
4. Listen
5. Ausblick

Iteratoren (1)

- Ein Iterator ist ein Objekt, das einen Durchlauf durch eine Liste von Objekten erlaubt.
- Das Interface `Iterator<E>` (im Paket `java.util`) enthält folgende Methoden:
  - `boolean hasNext()`:
    Dieser Iterator ist noch nicht am Ende der Liste.
  - `E next()`:
    Nächstes Element liefern und Iterator eins weiter schalten.

    Falls es kein nächstes Element mehr gibt: `NoSuchElementException`.

    Man kann das vermeiden, indem man vorher `hasNext()` abfragt.
  - `void remove()`:
    Das Element, das `next()` zuletzt geliefert hat, aus der zugrundeliegenden Collection löschen.

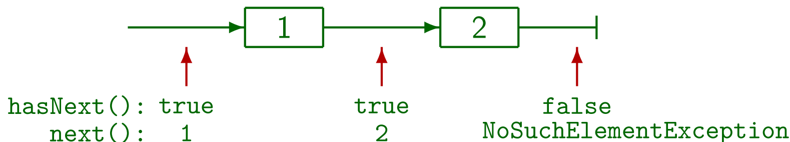
    Operation ist optional, ggf. nur `UnsupportedOperationException`.

Iteratoren (2)

- Ein Iterator verwaltet also eine Position in einer Liste von Werten, die der Iterator liefern kann.

  Je nach Collection kann die Reihenfolge definiert sein (z.B. bei Listen), oder beliebig von der Implementierung gewählt (z.B. bei Mengen).

- Bei einer Liste mit zwei Elementen hat der Iterator drei mögliche Positionen (von “vor dem ersten Element” bis “hinter dem letzten”):



Iteratoren (3)

```
 (1)  import java.util.Collection;
 (2)  import java.util.ArrayList;
 (3)  import java.util.Iterator;
 (4)  class CollTest {
 (5)      public static void main(String[] args) {
 (6)          Collection<Integer> c =
 (7)              new ArrayList<Integer>;
 (8)          c.add(1);
 (9)          c.add(2);
(10)          Iterator<Integer> it = c.iterator();
(11)          while(it.hasNext()) {
(12)              int i = it.next();
(13)              System.out.println(i);
(14)          }
(15)      }
(16)  }
```

Iteratoren (4)

- Die Schleife mit dem Iterator aus obigen Programmstück ist etwas lang/umständlich:

```
(10)        Iterator<Integer> it = c.iterator();
(11)        while(it.hasNext()) {
(12)            int i = it.next();
(13)            System.out.println(i);
(14)        }
```

- Da Schleifen mit Iteratoren recht häufig sind, wurde dafür ein spezielles Sprachkonstrukt eingeführt, die erweiterte `for`-Schleife ("enhanced for-loop", "for-each loop").

```
(10)        for(int i : c)
(11)            System.out.println(i);
```

Die Programmstücke tun genau das Gleiche. Man braucht keine explizite Variable für den Iterator mehr.

Iteratoren (5)

- Wir kennen die erweiterte `for`-Schleife schon für Arrays:

```
(10)        int[] arr = { 10, 20, 30 };
(11)        for(int i : arr)
(12)            System.out.println(i);
```

- Auch hier läuft `i` über den Einträgen/Elementen im Array, nimmt also nacheinander die Werte 10, 20, 30 an.

  Nicht etwa die Index-Werte 0, 1, 2.

- Das geht natürlich für Arrays von beliebigen Typen `T[]`. Die deklarierte Laufvariable muss den Element-Typ `T` haben.

  Zuweisungen an `i` sind schlechter Stil, würden den Array-Eintrag aber nicht ändern.

- Die Laufvariable `i` ist nur im Rumpf der Schleife bekannt.

  Nach der Schleife kann man nicht mehr darauf zugreifen.

Iteratoren (6)

- Es ist verboten, die zugrundeliegende Collection zu ändern, während ein Iterator darüber läuft.

  Das Verhalten ist undefiniert. Bessere Implementierungen von Iteratoren, z.B. bei der Klasse `LinkedList<E>` sind "fail-fast": Sie bemerken die Änderung beim ersten Aufruf danach und liefern dann eine `ConcurrentModificationException`. Es könnte aber auch sein, dass Elemente übersprungen werden, oder das gleiche Element mehrfach geliefert wird, gelöschte Elemente doch noch geliefert werden, oder aber irgendeine Exception auftritt.

- Davon ausgenommen ist natürlich die `remove()`-Operation des Iterators selbst.

  Für Listen gibt es auch erweiterte Iteratoren (`ListIterator<E>`), die auch Einfügungen erlauben.

Eigene Iterator-Klassen (1)

- Selbstverständlich kann man auch eigene Klassen definieren, die das Interface `Iterator<E>` implementieren.
- Natürlich auch für konkrete Element-Typen `E`, z.B. `Integer`.

  Der Typ `int` geht nicht, Typ-Parameter müssen immer Referenztypen sein.

- Das Interface `Iterable<E>` (im Paket `java.lang`) enthält als einzige Methode

```
Iterator<E> iterator();
```

  Es bedeutet also, dass diese Klasse einen Iterator liefern kann.

- Wenn eine Klasse `C` das Interface `Iterable<E>` implementiert, kann man die erweiterte `for`-Schleife für Objekte `c` dieser Klasse verwenden:

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

Eigene Iterator-Klassen (2)

```
 (1)  class MyIter implements Iterator<Integer> {
 (2)      private int i, n;
 (3)      MyIter(int n) {
 (4)          this.n = n;
 (5)          this.i = 0;
 (6)      }
 (7)      public boolean hasNext() {
 (8)          return i < n;
 (9)      }
(10)      public Integer next() {
(11)          return i++;
(12)      }
(13)      public void remove() {
(14)          throw new
(15)              UnsupportedOperationException();
(16)      }
(17)  }
```

Eigene Iterator-Klassen (3)

```
(18)  class MyClass implements Iterable<Integer> {
(19)      private int n;
(20)      MyClass(int n) {
(21)          this.n = n;
(22)      }
(23)
(24)      public MyIter iterator() {
(25)          return new MyIter(this.n);
(26)      }
(27)  }
(28)  class Test {
(29)      public static void main(String[] args) {
(30)          MyClass c = new MyClass(5);
(31)          for(int i : c)
(32)              System.out.println(i);
(33)      }
(34)  }
```

Inhalt

1. Einleitung
2. Collection Interface
3. Iteratoren
4. Listen
5. Ausblick

Listen (1)

- Listen sind vermutlich die am häufigsten verwendeten Collection-Klassen.

  Listen unterscheiden sich von Mengen dadurch, dass die Elemente einer Liste eine bestimmte Reihenfolge haben, und ein Objekt mehrfach in einer Liste enthalten sein kann (an unterschiedlichen Positionen). Eine Liste ist im wesentlichen ein flexibles Array (ohne feste Größe).

- Das Interface `List<E>` erweitert `Collection<E>` um Methoden, die auf der Reihenfolge / Index-Position von Elementen in einer Liste basieren.

  Da `Collection<E>` auch von Mengen implementiert werden muss, fehlen dort solche Operationen.

- Es gibt verschiedene Implementierungen von `List<E>`, besonders die Klassen `ArrayList<E>` und `LinkedList<E>`.

  Sie unterscheiden sich in der Effizienz unterschiedlicher Operationen, s.u.

Listen (2)

- `boolean add(E e)`:
  Fügt `e` an das Ende der Liste an.

  Für Listen liefert diese Operation immer `true`. Implementierungen des `List<E>`-Interfaces könnten aber eine Exception auslösen, wenn sie das Element nicht akzeptieren. Die Klassen `ArrayList<E>` und `LinkedList<E>` tun das aber nicht. Sie akzeptieren sogar die Einfügung von `null`-Elementen.

- `boolean add(int i, E e)`:
  Fügt `e` an Index-Position `i` in die Liste ein.

  Elemente mit einer Position $j \geq i$ werden nach rechts/hinten geschoben, also an die Index-Position $j + 1$. Die Index-Position `i` darf maximal die aktuelle Länge der Liste sein, bei der Einfügung dürfen also keine Lücken entstehen.

- `E set(int i, E e)`:
  Ersetzt das Element Index-Position `i` durch `e`,
  liefert das alte Element an dieser Index-Position.

Listen (3)

- `E remove(int i)`:
  Liefert das Element an Index-Position `i` und entfernt es aus der Liste.

  Weiter rechts/hinten stehende Elemente rücken also um eins auf (ihr Index wird um 1 verkleinert).

- `E get(int i)`:
  Liefert das Element an Index-Position `i`.

  Die Liste wird nicht verändert, das Element bleibt darin.

- `int indexOf(Object o)`:
  Liefert die Index-Position des ersten Vorkommens des Objektes `o` in der Liste, oder −1 wenn `o` nicht in der Liste vorkomment.

  Es gibt auch `lastIndexOf(Object o)` für die Suche von hinten.

Listen (4)

- Selbstverständlich haben Listen auch alle oben für das Interface `Collection<E>` genannten Methoden.
- Listen haben auch spezielle Listen-Iteratoren, die mehr können, als gewöhnliche Iteratoren, nämlich:
  - Bewegung in beiden Richtungen: außer `hasNext()` und `next()` gibt es auch `hasPrevious()` und `previous()`.
  - Mehr Möglichkeiten für Änderungen der zugrundeliegenden Liste beim Durchlauf: außer `remove()` gibt es auch `add(E e)` und `set(E e)`.
  - Abfrage der aktuellen Index-Position mit `nextIndex()` und `previousIndex()`.
- Man erhält einen `ListIterator<E>` durch Aufruf der Methode `listIterator()` von `List<E>`.

Listen-Implementierungen (1)

- `List<E>` ist ein Interface. Es legt also nur fest, welche Operationen es geben muss.
- Klassen, die das Interface implementieren, sind u.a.
  - `LinkedList<E>`: Implementierung mit einer verketteten Liste.

    Doppelt verkettet mit Zeigern auf nächstes und voriges Element.
  - `ArrayList<E>`: Implementierung mit einem Array.

    Bei Bedarf wird das Array durch ein größeres ersetzt.
- Die Implementierungen unterscheiden sich in der Zeit, die verschiedene Operationen benötigen:
  - Operationen, die auf der Index-Position basieren, gehen beim Array schnell, bei der verketteten Liste langsam.
  - Einfügung/Löschung eines Elementes sind bei der verketteten Liste schnell, beim Array langsam (außer am Ende).

Listen-Implementierungen (2)

- Wenn man die Variable mit dem Interface-Typ `List<E>` deklariert,
  - muss man nur bei der Objekt-Erzeugung festlegen, welche Implementierung man will:
    ```
    List<Integer> l = new LinkedList<Integer>();
    ```
  - Der ganze Rest des Programms ist dagegen unabhängig von dieser Wahl.
- Man könnte später also leicht die Implementierung ändern.
- Allerdings hat jede Implementierung noch Spezialoperationen, die über das Interface `List<E>` hinausgehen.

  Bei der `LinkedList<E>` gibt es z.B. noch `addFirst(E e)` und `addLast(E e)` (das ist aber äquivalent zu `add(e)`), und entsprechend `getFirst()` und `getLast()`, sowie `removeFirst()` und `removeLast()`.

Inhalt

1. Einleitung
2. Collection Interface
3. Iteratoren
4. Listen
5. Ausblick

Mengen

- Das Interface `Set<E>` enthält Operationen für die Verwaltung von endlichen Mengen.

  Wesentlicher Punkt: Mengen enthalten jedes Element nur einmal.

- Es fügt keine Methoden von den von `Collection<E>` ererbten hinzu, spezifiziert aber das Verhalten genauer.
- Implementierungen sind u.a.
  - `TreeSet<E>` (mit balanciertem Suchbaum)
  - `HashSet<E>` (mit Hashtabelle)
- Insbesondere soll der Element-Test mit `contains(Object o)` schnell gehen.

  Weitere Infos: Vorlesung “Datenstrukturen und effiziente Algorithmen I”.

Abbildungen (1)

- Das Interface `Map<K,V>` (im Paket `java.util`) erlaubt es,
  - Abbildungen mit endlicher Menge von Eingabewerten zu verwalten (Eingabe: “Schlüssel”, Ausgabe: “Wert”),
  - also eine Tabelle mit zwei Spalten:

Schlüssel (“Key”)	Wert (“Value”)
Brass	→ Dozentenobjekt 1
Zimmermann	→ Dozentenobjekt 2

- Dabei wird bei den meisten Implementierungen der Wert zu einem Schlüssel schnell gefunden.

  Deutlich schneller als bei linearer Suche durch eine Liste oder ein Array.

- Klassen, die dieses Interface implementieren, sind z.B. `HashMap<K,V>` und `TreeMap<K,V>`.

Abbildungen (2)

- Die wichtigsten Methoden des Interfaces sind:

  Es ist kein Subinterface von `Collection<E>`.

  - `V put(K key, V value)`:
    Trägt das Schlüssel-Wert-Paar in die Abbildung ein.

    Falls es schon einen Eintrag mit dem Schlüssel `key` gab, wird der alte Wert geliefert, sonst `null`.

  - `V get(Object key)`:
    Liefert den Wert zum Schlüssel `key`.

    Bzw. `null`, falls es keinen Eintrag mit diesem Schlüssel gibt.

  - `boolean containsKey(Object key)`:
    Prüft, ob es einen Eintrag für `key` gibt.

  - `V remove(Object key)`:
    Löscht den Eintrag für `key`, liefert alten Wert.