

Sammlungen



Inhalt

Einleitung

Listen und Queues

Mengen

Abbildungen

Algorithmen und Wrapper

Zusammenfassung



Einleitung

- *Collections* sind Datenstrukturen für *Sammlungen von Daten*
 - Stacks
 - Queues
 - Mengen
 - Trees
 - Maps
- Collection-Frameworks bieten Schnittstellen und Klassen für den Umgang mit Sammlungen
- Collection-Frameworks sind ein zentraler Bestandteil jeder Klassenbibliothek, z.B.
 - Standard Template Library von C++
 - Smalltalk's collection classes
 - System. Collections Namespace von .NET



Entwurfsziele des Collection-API

- in sich konsistentes Framework für Collections
- stellt alle wesentlichen Features für Collections zur Verfügung
- trotzdem überschaubar und leicht zu erlernen (siehe Zitat)
- effizient
- erweiterbar
- muss alte Collection-Klassen integrieren

Zitat „Collections Framework Overview“:

An interface contains a method only if either:

1. It is a truly *fundamental operation*: a basic operations in terms of which others could be reasonably defined,
2. There is a compelling performance reason why an important implementation would want to override it.



- In der Java-Klassenbibliothek unterscheidet man zwei Mengen von Klassen, die Collections realisieren.
 1. die traditionellen Klassen (seit 1.0)
 - Vector
 - Stack
 - Hashtable
 - Dictionary
 - Bitset
 2. das Collection API (seit 1.2)
 - Umfangreiches API (Sorting, ...)
- 1 & 2 existieren seit 1.2 parallel



Die traditionellen Klassen

Vector

- Realisiert eine Liste mit Direktzugriff, die wachsen kann.

Stack

- Stapel (push, pop)

Hashtabl e & Di cti onary

- Zuordnung von Schlüsseln auf Werte

Bi tset

- Realisiert eine Menge von Bits, die einzeln gesetzt werden können. Unterstützt Operationen wie

`set(i nt bi t)`

`and(Bi tset s)`

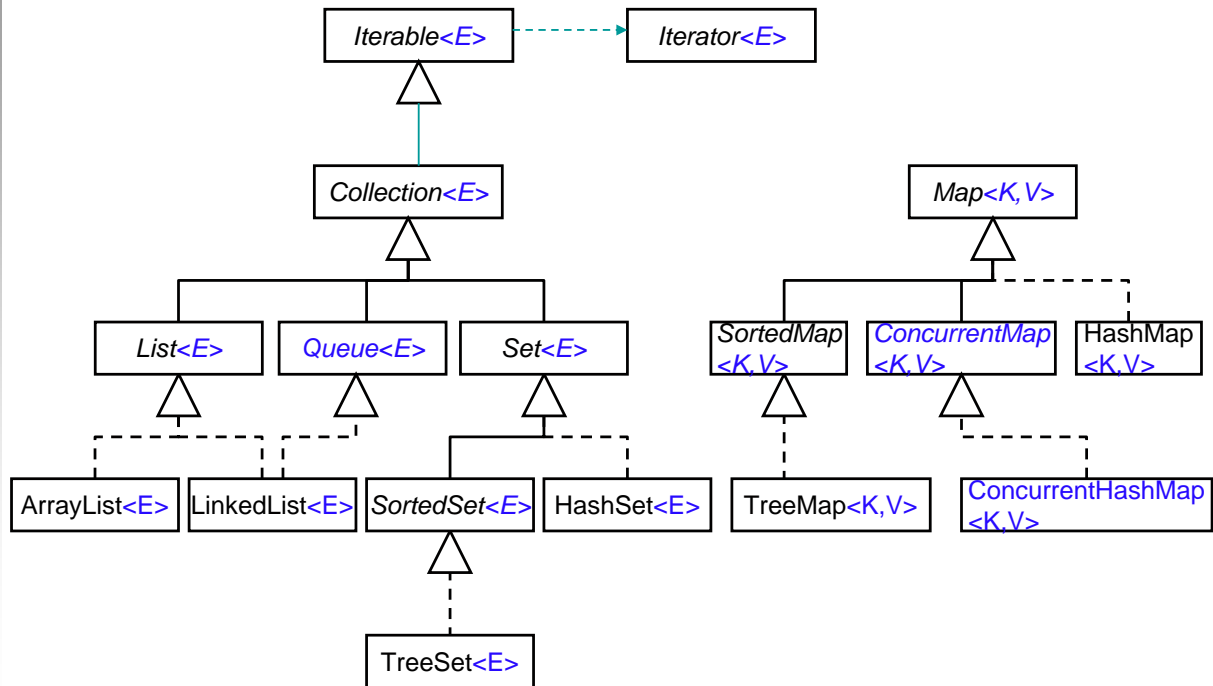
`or(Bi tset s)`



Klassenhierarchie: Interfaces und konkrete Klassen

package java.util

Seit Java 1.5

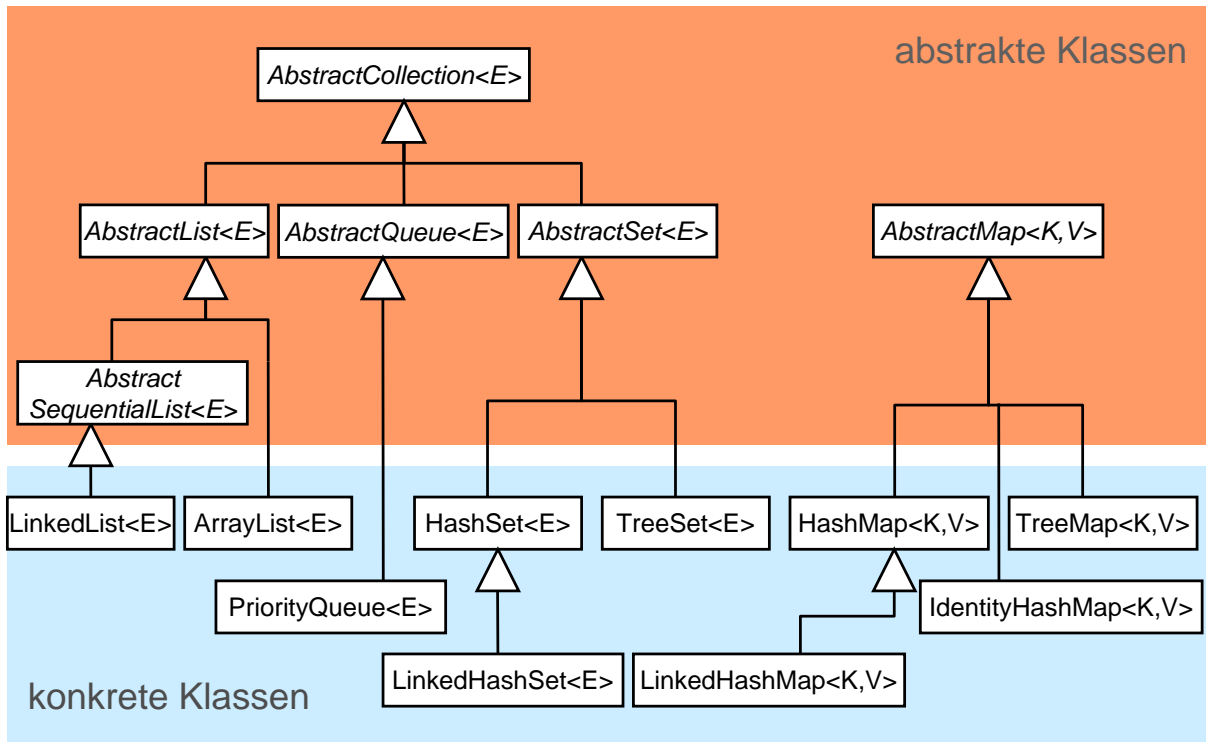


Implementierungen von Collection-Interfaces: Überblick

		Implementations				
		Hash Table	Resizable Array	Balanced Tree	Linked List	Hash Table + Linked List
Interfaces	Set	HashSet		TreeSet		LinkedHashSet
	List		ArrayList		LinkedList	
	Map	HashMap		TreeMap		LinkedHashMap



Implementierungen von Collection-Interfaces



Iterable und Iterator

- Für alles was aufzählbar ist
- erlaubt die Verwendung der foreach-Anweisung
- Zugriff auf Iterator

```
package java.lang;

public interface Iterable<E> {
    Iterator<E> iterator();
}
```

```
package java.util;

public interface Iterator<E> {
    boolean hasNext();
    E next();
    void remove();
}
```



Iteration über Elemente (nicht-generische Version)

- Interface `Iterator` dient zum Iterieren über die Elemente
- ersetzt `Enumeration` aus alter API
 - Elemente können entfernt werden
 - Methodennamen wurden verkürzt

```
public interface Iterator {
    boolean hasNext();
    Object next();
    void remove();
}
```

Beispiel:

```
static void removeNumbers(Collection coll) {
    Iterator iter = coll.iterator();
    while (iter.hasNext()) {
        Object elem = iter.next();
        if (elem instanceof Number) {
            iter.remove();
        }
    }
}
```



Iteration über Elemente (generische Variante)

- Interface `Iterator<T>` dient zum Iterieren über die Elemente

```
public interface Iterator<E> {
    boolean hasNext();
    E next();
    void remove();
}
```

Beispiel:

```
static void iterateOverPersons(Collection<Person> coll) {
    Iterator<Person> iter = coll.iterator();
    while (iter.hasNext()) {
        Person p = iter.next();
        // do something with p
    }
}
```



Iteration über Elemente (Verwendung von foreach-Anweisung)

- Collection mit Iterator und Verwendung der foreach-Schleife

```
public interface Iterator<E> {  
    boolean hasNext();  
    E next();  
    void remove();  
}
```

Beispiel:

```
static void removeNumbers(Collection<Person> coll) {  
    for (Person p: coll) {  
        // do something with p  
    }  
}
```



Interface Collection

Collection ist elementares Interface für Sammlungen

- Basic Operations (müssen implementiert werden):
 - enthalten sein
 - ist leer und Anzahl der Elemente
 - Iterator
- Optionale Operationen (können implementiert sein):
 - Anfügen von Elementen
 - Löschen
 - Beibehalten von Elementen)
- Umwandlung in Arrays

```
package java.util;  
  
public interface Collection<E> {  
    // basic operations  
    boolean contains(Object o);  
    boolean containsAll(Collection<?> c);  
    boolean isEmpty();  
    Iterator<E> iterator();  
    int size();  
  
    // optional operations  
    boolean add(E o);  
    boolean addAll(Collection<? extends E> c);  
    void clear();  
    boolean remove(Object o);  
    boolean removeAll(Collection<?> c);  
    boolean retainAll(Collection<?> c);  
  
    // conversion  
    Object[] toArray();  
    <T> T[] toArray(T[] a);  
}
```



Anmerkungen zu Gestaltung der Methoden

- Konkrete Klassen sollten zwei Konstruktoren besitzen
 - Zum Erzeugen einer leeren Sammlung
 - Zum Erzeugen mit Elementen einer existierenden Sammlung
- Optionale Operationen können `UnsupportedOperationException` werfen
 - z.B. Hinzufügen zu unveränderbare Sammlungen
- Boolesche Methoden
 - Rückgabewert gibt an, ob sich Sammlung verändert hat
- Verwendung von Wrapper-Klassen für primitive Datentypen
 - Boolean, Byte, Character, Double, Float, Integer, Long, Short
- Konvertierung in Array
 - Als Object-Array
 - Als Array eines bestimmten Typs (Array des gewünschten Typs wird als Parameter übergeben)

```
List l = new ArrayList(elements);
Object[] lObjects = l.toArray();
Point[] lPoints = (Point[]) l.toArray(new Point[0]);

List<Integer> i1 = new ArrayList<Integer>();
Integer[] intArr = i1.toArray(new Integer[0]);
```



Verwendung von toArray

- `toArray` liefert Elemente in Array
- Frage des Typs des Arrays
 - Object-Array: `Object[] toArray();`
 - Array bestimmten Typs: `<T> T[] toArray(T[] a);`
→ Typ des Arrays wird mit Parameter angegeben!

Beispiel

- Nicht-generische Variante

```
List l = new ArrayList(points);
Object[] lObjects = l.toArray();
Point[] lPoints = (Point[]) l.toArray(new Point[0]);
```

- Generische Variante

```
List<Point> l = new ArrayList<Point>(points);
Point[] intArr = l.toArray(new Point[0]);
```



Einleitung

Listen und Queues

Mengen

Abbildungen

Algorithmen und Wrapper

Zusammenfassung



Interface List

- List ist ein spezielles Interface, das von einer Reihenfolge der Elemente ausgeht
- Damit hat jedes Element eine Position
- Es sind folgende zusätzliche Operationen definiert:

- Einfügen an einer bestimmten Position
- Zugriff auf ein Element an einer bestimmten Position
- Löschen eines Element an einer bestimmten Position
- Position eines Elements
- Spezieller ListIterator
- Elemente in einem bestimmten Bereich

```
public interface List<E> extends Collection<E> {  
    // positional access  
    void add(int index, E element);  
    E get(int index);  
    E remove(int index);  
    E set(int index, E element);  
  
    // search  
    int indexOf(Object o);  
    int lastIndexOf(Object o);  
  
    // iteration  
    ListIterator<E> listIterator();  
    ListIterator<E> listIterator(int index);  
  
    // sublist  
    List<E> subList(int fromIndex, int toIndex);  
}
```



Interface ListIterator

- Interface ListIterator erlaubt erweiterte Iteration über Elemente
- Damit sind folgende zusätzliche Methoden definiert:

- Iteration nach hinten und nach vorne
- Einfügen an der aktuellen Position
- Setzen des aktuell gelesenen Elements
- Abfrage der Position des nächsten und vorherigen Elements

```
public interface ListIterator<E>
    extends Iterator<E> {
    boolean hasNext();
    boolean hasPrevious();
    E next();
    int nextIndex();
    E previous();
    int previousIndex();
    void add(E o);
    void remove();
    void set(E o);
}
```



Implementierungen von List

Es gibt zwei alternative Implementierungen von List:

- **LinkedList:**
 - doppelt verkettete Liste
 - gut bei Einfügen in der Mitte
 - schlecht bei direktem Zugriff auf ein Element
- **ArrayList:**
 - auf der Basis von Arrays
 - gut bei direktem Zugriff
 - schlecht beim Einfügen von Elementen



Queues

- Neu in Java 1.5
- Interface für Queues

Typischerweise *FIFO*, First In First out



- API
 - Abfragen, nicht entfernen
 - Einfügen eines Elements
 - Abfragen, nicht entfernen evtl. null
 - Abfragen, entfernen evtl. null
 - Abfragen, entfernen
- Implementierungen
 - LinkedList
 - PriorityQueue: mit Sortierung

```
public interface Queue<E>
    extends Collection<E> {
    E    element();
    boolean offer(E o);
    E    peek();
    E    poll();
    E    remove();
}
```

	Throws exception	Returns special value
Insert	add(e)	offer(e)
Remove	remove()	poll()
Examine	element()	peek()



BlockingQueue

- Synchronisierte Queue-Interface mit blockierenden Methoden
 - Warten bis Queue nicht-leer bei Versuch der Entnahme
 - bzw. Warten bis Platz bei Versuch der Speicherung
- mit TimeOuts

```
public interface BlockingQueue<E>
    extends Queue<E> {
    boolean offer(E o, long timeout, TimeUnit unit)
        throws InterruptedException;
    E poll(long timeout, TimeUnit unit)
        throws InterruptedException;
    E take() throws InterruptedException;
    void put(E o) throws InterruptedException;
    int remainingCapacity();
    int drainTo(Collection<? super E> c);
    ...
}
```

- Implementierungen:
 - ArrayBlockingQueue
 - LinkedBlockingQueue
 - DelayQueue
 - PriorityBlockingQueue
 - SynchronousQueue

package java.util.concurrent



Einleitung

Listen und Queues

Mengen

Abbildungen

Algorithmen und Wrapper

Zusammenfassung



Interface Set

- Entspricht dem mathematischen Konzept der Menge
- Elemente können maximal einmal enthalten sein,
- Vergleich der Elemente erfolgt mit `equals`
- `null` prinzipiell als Element zulässig
- Keine Aussage über Reihenfolge der Elemente

```
public interface Set<E>  
    extends Collection<E>, Iterable<E> {  
  
    // Identische Methoden wie Collection (!)  
  
}
```



Interface SortedSet

- Wie Set aber Sortierung der Elemente
- Sortierkriterium kann sein:
 - Comparable: Elemente müssen Comparable-Interface implementieren und Vergleich der Elemente erfolgt mittels compareTo-Methoden
 - Comparator: dem SortedSet wird ein eigenes Comparator-Objekt übergeben, mit dem die Elemente verglichen werden

```
public interface SortedSet<E>
    extends Set<E> {
    Comparator<? super E> comparator();
    E first();
    SortedSet<E> headSet(E toElement);
    E last();
    SortedSet<E> subSet(E fromElement, E toElement);
    SortedSet<E> tailSet(E fromElement);
}
```

```
public interface Comparable<T> {
    int compareTo(T o);
}
```

```
public interface Comparator<T> {
    int compare(T o1, T o2);
    boolean equals(Object obj);
}
```

Achtung: Dieses equals wird verwendet, um die Gleichheit von Comparatoren zu bestimmen



Implementierungen von Set

Implementierungen von Set:

- HashSet:
 - Auf der Basis von Hashtabellen
 - schneller Zugriff und schnelles Einfügen
- EnumSet:
 - Alle Elemente müssen Werte *einer* Enumeration sein

Implementierungen von SortedSet:

- TreeSet:
 - Auf der Basis von Red-Black-Trees
 - binäre Sortierung



Beispiel SortedSet und Comparator

Nicht-generisch:

```
class PersonComparator implements Comparator {  
    public int compare(Object o1, Object o2) {  
        Person a = (Person)o1;  
        Person b = (Person)o2;  
        return a.getSurName().compareTo(b.getSurName());  
    }  
}
```

```
SortedSet s = new TreeSet(new PersonComparator());  
s.add(new Person("Hans", "Berger", "Linz"));  
s.add(new Person("Franz", "Mayr", "Graz"));  
s.add(new Person("Otto", "Berger", "Wien")); // fails
```

Generisch:

```
class PersonComparator implements Comparator<Person> {  
    public int compare(Person p1, Person p2) {  
        return p1.getSurName().compareTo(p2.getSurName());  
    }  
}
```

```
SortedSet<Person> s = new TreeSet<Person>(new PersonComparator());  
s.add(new Person("Hans", "Berger", "Linz"));  
s.add(new Person("Franz", "Mayr", "Graz"));  
s.add(new Person("Otto", "Berger", "Wien")); // fails
```



Inhalt

Einleitung

Listen und Queues

Mengen

Abbildungen

Algorithmen und Wrapper

Zusammenfassung



Interface Map

- Abbildung von Schlüsseln auf Werte
 - Jeder Schlüssel maximal einmal enthalten
 - Vorsicht bei veränderlichen Schlüsselwerten
- Jeder Eintrag ist ein Objekt vom Typ Map. Entry

```
public interface Map<K, V> {  
    // basic operations  
    boolean containsKey(Object key);  
    boolean containsValue(Object value);  
    V get(Object key);  
    boolean isEmpty();  
    int size();  
  
    // optional operations  
    void clear();  
    V put(K key, V value);  
    void putAll(Map<? extends K, ? extends V> t);  
    V remove(Object key);  
  
    // collection view  
    Set<Map.Entry<K, V>> entrySet();  
    Set<K> keySet();  
    Collection<V> values();  
}
```

```
public static interface Map.Entry<K, V> {  
    K getKey();  
    V getValue();  
    V setValue(V value);  
}
```



Interface SortedMap

- Wie Map aber mit Sortierung der Schlüssel

```
public interface SortedMap<K, V> extends Map<K, V> {  
    Comparator<? super K> comparator();  
    K firstKey();  
    K lastKey();  
    SortedMap<K, V> subMap(Object fromKey, Object toKey);  
}
```



Implementierungen von Map

Implementierungen von Map:

- **HashMap:**
 - Auf der Basis von Hashtabellen
 - schneller Zugriff und Einfügen
- **Li nkedHashMap:**
 - zusätzlich definierte Reihenfolge auf Basis von Linked-Lists
- **WeakHashMap:**
 - schwache Referenzen der Werte
- **I denti tyHashMap:**
 - Vergleich der Keys auf Referenzgleichheit
- **EnumMap**
 - Alle Schlüssel müssen aus einer Enumeration stammen.

Implementierungen von SortedMap:

- **TreeMap:**
 - Auf der Basis von Red-Black-Trees
 - binäre Sortierung



Inhalt

Einleitung

Listen und Queues

Mengen

Abbildungen

Algorithmen und Wrapper

Zusammenfassung



Hilfsklasse Collections

- Klasse Collections bietet eine Vielzahl von static-Methoden für:
 - Algorithmen
 - Erzeugen von Wrapper für die Collection-Implementierungen

Algorithmen in Collections:

```
public class Collections {
    static <T>int binarySearch(List<? extends Comparable<? super T>> list, T key);
    static <T>int binarySearch(List<? extends T> list, T key, Comparator<? super T> c);
    static <T>void copy(List<? super T> dest, List src<? extends T>);
    static <T>void fill(List<? super T> list, T o);
    static <T extends Object & Comparable<? super T>> T max(Collection<? extends T> coll);
    static <T>T max(Collection<? extends T> coll, Comparator<? super T> comp);
    static <T extends Object & Comparable<? super T>> T min(Collection<? extends T> coll);
    static <T>T min(Collection<? extends T> coll, Comparator<? super T> comp);
    static <T>List<T> nCopies(int n, T o);
    static void reverse(List<?> l);
    static void shuffle(List<?> list);
    static void shuffle(List<?> list, Random rnd);
    static <T>Set<T> singleton(T o);
    static <T>List<T> singletonList(T o);
    static <K, V>Map<K, V> singletonMap(K key, V value);
    static <T extends Comparable<? super T>> void sort(List<T> list);
    static <T>void sort(List<T> list, Comparator<? Super T> c);
    ...
}
```



Beispiel Algorithmen: Mischen von Karten

- Folgende Methode mischt Karten

```
static List shuffleCards() {
    String[] suit = new String[] {"spades", "hearts", "di amonds", "clubs"};
    String[] rank = new String[] {"ace", "2", "3", "4", "5", "6", "7", "8",
        "9", "10", "jack", "queen", "king"};

    List deck = new ArrayList();
    for (int i = 0; i < suit.length; i++) {
        for (int j = 0; j < rank.length; j++) {
            deck.add(rank[j] + " of " + suit[i]);
        }
    }
    Collections.shuffle(deck);
    return deck;
}
```



Wrapper

- Collection-Frameworks erlaubt über Wrapper bestimmte *Sichten* von Collection-Objekten zu erzeugen
- Diese Sichten realisieren das entsprechende Interface mit bestimmten Zusätzen
- Zwei wichtige Wrapper sind Wrapper mit:
 - synchronisierte-Methoden: exklusiver Zugriff auf Collection pro Thread
 - unmodifizierte-Methoden: unveränderliche Sicht auf Collection
- Klasse Collections bietet Methode solche Wrapper zu erzeugen:

```
static <T>Collection<T>    synchronisiertCollection(Collection<T> c);
static <T>Set<T>          synchronisiertSet(Set<T> s);
static <T>List<T>         synchronisiertList(List<T> list);
static <K, V>Map<K, V>     synchronisiertMap(Map<K, V> m);
static <T>SortedSet<T>    synchronisiertSortedSet(SortedSet<T> s);
static <K, V>SortedMap<K, V> synchronisiertSortedMap(SortedMap<K, V> m);

static <T>Collection<T>    unmodifizierteCollection(Collection<? extends T> c);
static <T>Set<T>          unmodifizierteSet(Set<? extends T> s);
static <T>List<T>         unmodifizierteList(List<? extends T> list);
static <K, V>Map<K, V>     unmodifizierteMap(Map<? extends K, ? extends V> m);
static <T>SortedSet<T>    unmodifizierteSortedSet(SortedSet<T> s);
static <K, V>SortedMap<K, V> unmodifizierteSortedMap(SortedMap<K, ? extends V> m);
```



Inhalt

Einleitung

Listen und Queues

Mengen

Abbildungen

Algorithmen und Wrapper

Zusammenfassung



Zusammenfassung

- Listen
 - Elemente in fester Reihenfolge
 - Zugriff auf Elemente über Index möglich
 - Suchfunktionen
- Mengen
 - Elemente maximal einmal enthalten
 - Eventuell sortiert
- Abbildungen
 - Abbildung von Schlüsselobjekten auf Wertobjekte
- Iteration über Elemente
 - Mittels Iterator und ListIterator
- Hilfsklasse Collections
 - Suchen, Füllen, Mischen und Sortieren
 - Erzeugen synchronisierter oder unveränderbarer Sammlungen



Literatur

- Java 2 SDK, Standard Edition Documentation, Guide to Features - Collection Framework, <http://java.sun.com/j2se/1.5.0/docs/guide/collections/index.html>
- Sprechen Sie Java?, 3 Auflage, dpunkt.verlag, 2005: Kapitel 15 Generizität
- Head First Java, 2nd Edition, O'Reilly, 2005: Chapter 16 collections and generics
- Horstmann, Cornell, Core Java 2, Band 2 - Expertenwissen, Markt und Technik, 2002: Kapitel 2
- Krüger, Handbuch der Java-Programmierung, 3. Auflage, Addison-Wesley, 2003, <http://www.javabuch.de>: Kapitel 14 und 15

