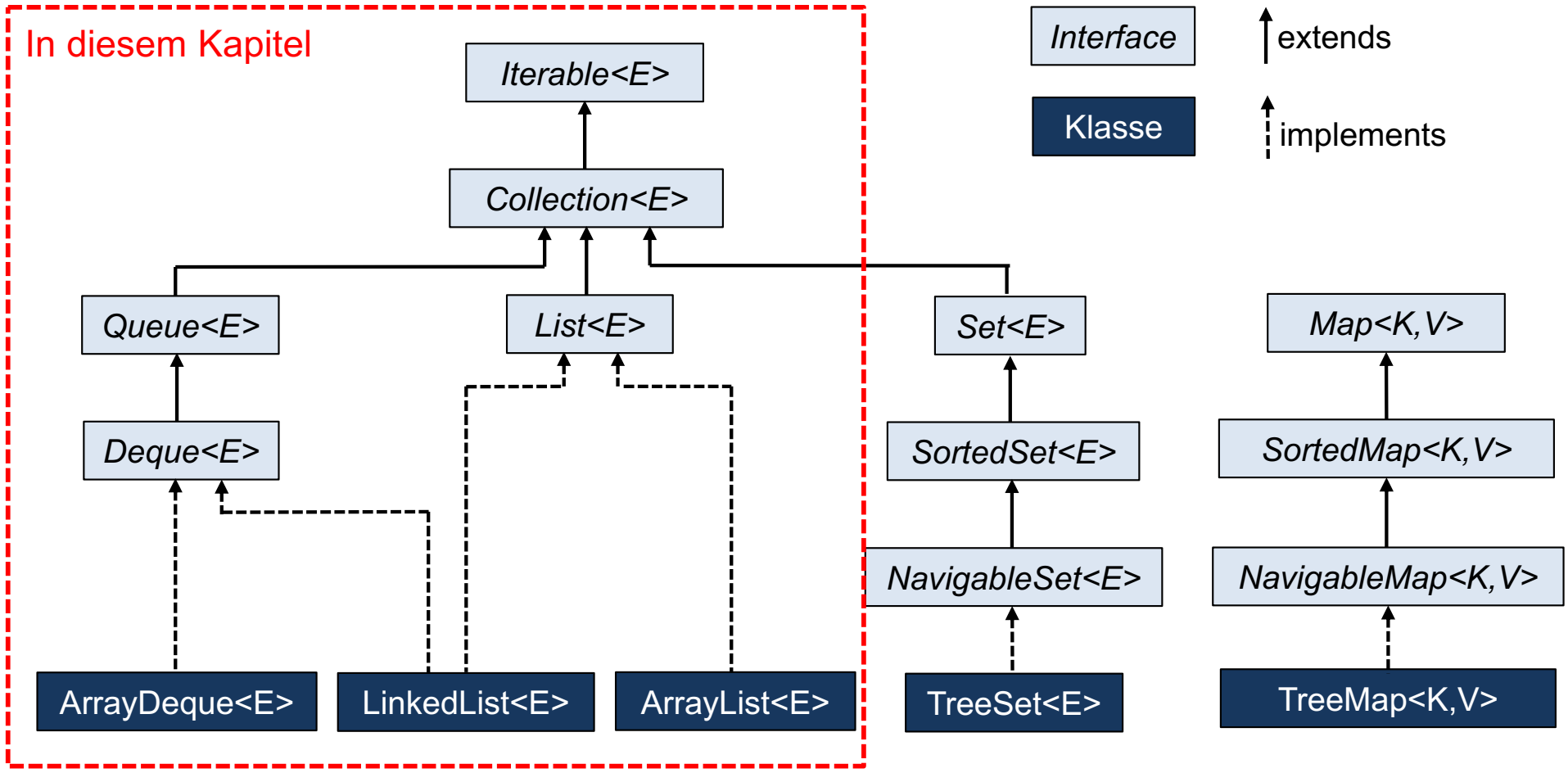
Kapitel 6: Java Collection – Teil I

- Übersicht
- Collection
- List
- LinkedList und ArrayList
- Queue und Deque
- ArrayDeque
- Bemerkungen

In diesem Kapitel: Queue- und List-Typen

- Die Java-API enthält eine Sammlung von generischen Schnittstellen und Klassen zur Realisierung von Containern (mengenartigen Datentypen).



Interface Collection<E>

```
public interface Collection<E> extends Iterable<E> {

   boolean add(E e);                              // add the element e
   boolean addAll(Collection<? extends E> c);     // add the contents of c

   boolean remove(Object o);                      // remove the element o
   boolean removeAll(Collection<?> c)             // remove the elements in c
   boolean retainAll(Collection<?> c);            // remove the elements not in c
   void clear();                                  // remove all elements

   boolean contains(Object o);                    // true if o is present
   boolean containsAll(Collection<?> c);          // true if all elements of c are present
   boolean isEmpty();                             // true if no element is present
   int size();                                    // number of elements

   Iterator<E> iterator();                        // returns an Iterator over the elements
   Object[ ] toArray();                           // copy contents to an Object[ ]
   <T> T[ ] toArray(T[ ] t);                      // copy contents to a T[ ] for any T
}
```

- Die Methoden add, addAll, remove, removeAll and retainAll liefern true, falls die Collection durch den Aufruf verändert wird

Anwendung mit Collection<E> (1)

```
public static void main(String[] args) {
    Collection<Number> nbCol = new LinkedList<>();
    Collection<Integer> intCol = new LinkedList<>();
    Collection<Double> dbCol = new LinkedList<>();

    nbCol.add(1);
    nbCol.add(1.5);
    nbCol.add(2);
    nbCol.add(2.5);
    System.out.println(nbCol);

    intCol.add(4);
    intCol.add(7);
    intCol.add(3);
    System.out.println(intCol);

    dbCol.add(1.0);
    dbCol.add(1.5);
    dbCol.add(1.5);
    System.out.println(dbCol);

    // ...
```

[1, 1.5, 2, 2.5]

[4, 7, 3]

[1.0, 1.5, 1.5]

Anwendung mit Collection<E> (2)

```
    // ...

    System.out.println(nbCol);

    System.out.println(intCol);


    nbCol.addAll(intCol);
    System.out.println(nbCol);

    // intCol.addAll(dbCol); // Typfehler

    Number n = new Integer(7);
    intCol.remove(n);
    System.out.println(intCol);

    // Alle Doubles löschen:
    Iterator<Number> it = nbCol.iterator();
    while (it.hasNext()){
        Number x = it.next();
        if (x instanceof Double)
            it.remove();
    }
    System.out.println(nbCol);
}
```

[1, 1.5, 2, 2.5]

[4, 7, 3]

[1, 1.5, 2, 2.5, 4, 7, 3]

[4, 3]

[1, 2, 4, 7, 3]

Number wird aus Integer-Collection gelöscht!

Ist erlaubt, da:

```
boolean remove(Object o);
```

Brückenfunktionen zwischen Feldern und Collection-basierten Behältern

```
public static void main(String[ ] args) {

    Collection<String> c = new LinkedList<>();
    c.add("abc");
    c.add("def");
    c.add("xyz");


    String[ ] a = c.toArray(new String[0]);
    System.out.println(Arrays.toString(a));


    List<String> listView = Arrays.asList(a);
    System.out.println(listView);


    List<String> listImmutableCopy = List.of(a);
    System.out.println(listImmutableCopy);
}
```

Liste in Feld kopieren

Feld wird als Liste
(mit fester Länge) zurückgeliefert.

Liste ist eine Sicht (view) auf das Feld `a`
(keine Kopie!), hat aber eine fixe Länge.

Änderungen im Feld `a` wirken sich auch auf
Liste `listView` aus und umgekehrt.

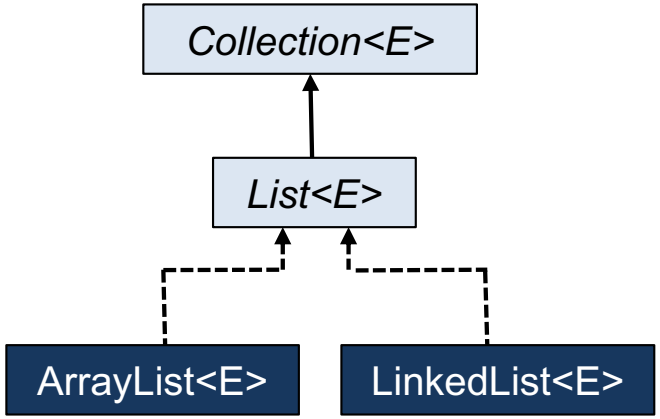
Feld wird als immutable Liste zurückgeliefert.

Liste ist dabei eine Kopie.

Änderungen im Feld `a` wirken sich nicht auf
Liste `listImmutableCopy` aus.

Listen

- Listen sind - wie in Kap. 2 besprochen - eine Folge von Elementen, wobei jedes Element eine Position hat.
- List<E>, ArrayList<E> und LinkedList<E> sind sehr ähnlich zu den in Kap. 2 besprochenen Typen.
- List bietet einen Iterator an, mit dem Listen auch umgekehrt durchlaufen werden können (reverse iterator).
- Da LinkedList als doppelt verkettete Liste realisiert ist, kann der reverse-Iterator effizient implementiert werden.



Interface List<E>

```
public interface List<E> extends Collection<E> {

        boolean add(int idx, E e);
        boolean addAll(int idx, Collection<? extends E> c);

        E set(int idx, E x);
        E get(int idx);

        E remove(int idx);

        int indexOf(Object o);
        int lastIndexOf(Object o);
        List<E> subList(int fromIdx, int toIdx);

        ListIterator<E> listIterator();
        ListIterator<E> listIterator(int index);

        static <E> List<E>copyOf(Collection<? extends E> coll)
        static <E> List<E>of(E... elements)
        static <E> List<E>of(E e)
        static <E> List<E>of(E e1, E e2)
}
```

- subList liefert eine Sicht (keine Kopie!) auf einen Teil der Liste, die von Position fromIdx einschließlich bis toIdx ausschließlich geht. D.h. Änderungen, die auf die Teilliste durchgeführt werden, schlagen auf die Liste durch!
- ListIterator ist ein Iterator, mit dem eine Liste auch rückwärts durchlaufen werden kann. Analog zu next() und hasNext() heißen die Methoden previous() und hasPrevious().
- copyOf und of sind statische Fabrikmethoden die immutable Listen (unmodifiable lists) zurückliefern. copyOf(coll) erstellt dabei eine Kopie der Elemente aus coll.
- ein paar weitere Default-Methoden (später)

Sichten-Konzept bei subList

```
public static void main(String[] args) {

    List<Integer> intList = new LinkedList<>();

    for (int i = 1; i <= 10; i++)
        intList.add(i);
    System.out.println(intList);                    [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

    System.out.println(intList.subList(4,8));       [5 ,6, 7, 8]

    intList.subList(4,8).clear();
    System.out.println(intList);                    [1, 2, 3, 4, 9, 10]
}
```

- subList(4,8) liefert lediglich eine Sicht (keine Kopie!) auf einen Teil der Liste, die von Position fromIdx = 4 einschließlich bis toIdx = 8 ausschließlich geht.
- D.h. Änderungen, die auf der Teilliste durchgeführt werden, wie subList(4,8).clear() schlagen auf die Liste durch!
- In Java API: Sicht = view.

ListIterator

- das folgende Beispiel zeigt einen rückwärts laufenden ListIterator mit den Methoden hasPrevious() und previous().

```
public static void main(String[] args) {

    List<Integer> intList = new LinkedList<>();

    for (int i = 1; i <= 10; i++)
        intList.add(i);
    System.out.println(intList);

    // Reverse Iterator
    ListIterator<Integer> it
        = intList.listIterator(intList.size());
    while (it.hasPrevious())
        System.out.print(it.previous() + ", ");
    System.out.println();

}
```

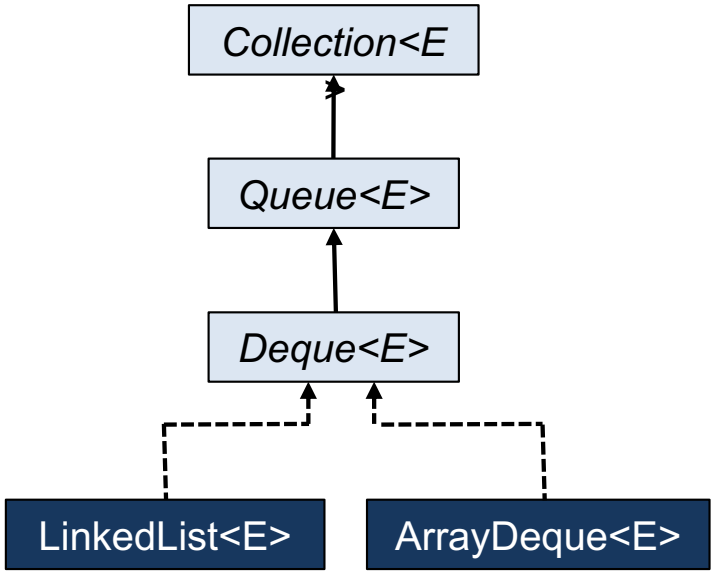
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

ListIterator it wird auf Listenende gesetzt.

[10, 9, 8, 7, 6, 5, 4, 3, 2, 1]

Schlangen

- Das Interface Queue<E> definiert die typischen Operationen für Schlangen.
- Deque<E> (double-ended queue) ist eine Schlange, bei der an beiden Enden eingefügt und gelöscht werden kann.
- Deque<E> kann auch als Keller verwendet werden.
- LinkedList<E> ist eine Implementierung als doppelt verkettete Liste.
- ArrayDeque<E> ist eine Implementierung als zirkuläres Feld.



Queue

- Das Interface Queue<E> definiert
  die typischen Operationen für Schlangen.

	Methode liefert false bzw. null bei Misserfolg	Methode wirft Exception bei Misserfolg
Einfügen am Ende	boolean offer(E x)	boolean add(E x)
Löscht und liefert das vordere Element	E poll()	E remove()
liefert das vordere Element	E peek()	E element()

Deque

- Deque<E> (double-ended queue) ist eine Schlange, bei der an beiden Enden eingefügt und gelöscht werden kann:

Methoden für vorderes Ende	Methode liefert false bzw. null bei Misserfolg	Methode wirft Exception bei Misserfolg
Einfügen	boolean offerFirst(E x)	void addFirst(E x)
Löschen und zurückliefern	E pollFirst()	E removeFirst()
zurückliefern	E peekFirst()	E getFirst()

Methoden für hinteres Ende	Methode liefert false bzw. null bei Misserfolg	Methode wirft Exception bei Misserfolg
Einfügen	boolean offerLast(E x)	void addLast (E x)
Löschen und zurückliefern	E pollLast()	E removeLast ()
zurückliefern	E peekLast()	E getLast()

Deque als Keller

- Deque<E> kann auch als Keller verwendet werden.
- Beachte: Es gibt in der Java-API auch eine Klasse Stack<E>.
  Diese ist jedoch veraltet.
  Deque<E> sollte vorgezogen werden.

Stack-Methode	Äqivalente Deque-Methode	Exception bzw. spezieller Rückgabewert
void push(E x)	void addFirst(E x)	Exception
E pop()	E removeFirst()	Exception und Rückgabe
E peek()	E peekFirst()	Rückgabe (kann null sein)

Anwendung mit Deque<E>

```
public static void main(String[] args) {

        // Schlange als Feld:
        Queue<Integer> queue = new ArrayDeque<>();
        for (int i = 1; i <= 10; i++)
            queue.add(i);

        while (!queue.isEmpty()) {
            int x = queue.remove();
            System.out.print(x + ", ");
        }
        System.out.println();

        // Keller als Feld:
        Deque<Integer> stack = new ArrayDeque<>();
        for (int i = 1; i <= 10; i++)
            stack.push(i);

        while (!stack.isEmpty()) {
            int x = stack.pop();
            System.out.print(x + ", ");
        }
        System.out.println();
    }
```

Schlange als Feld.

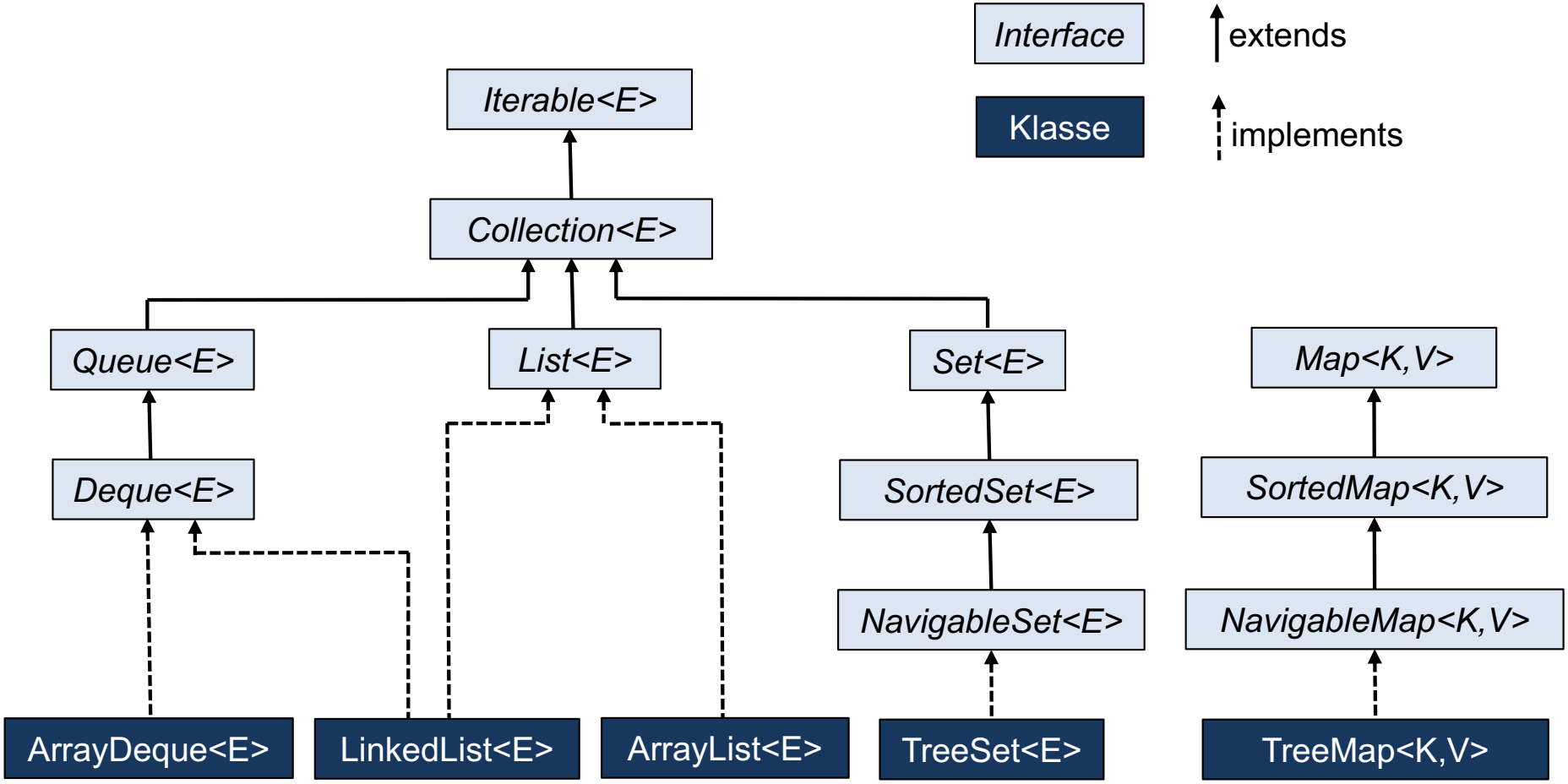
1, 2, 3, 4, 5, 6, 7, 8, 9, 10,

Keller als Feld.

10, 9, 8, 7, 6, 5, 4, 3, 2, 1,

Bemerkungen (1)

- Darstellung der Collection-Typen ist nicht vollständig!

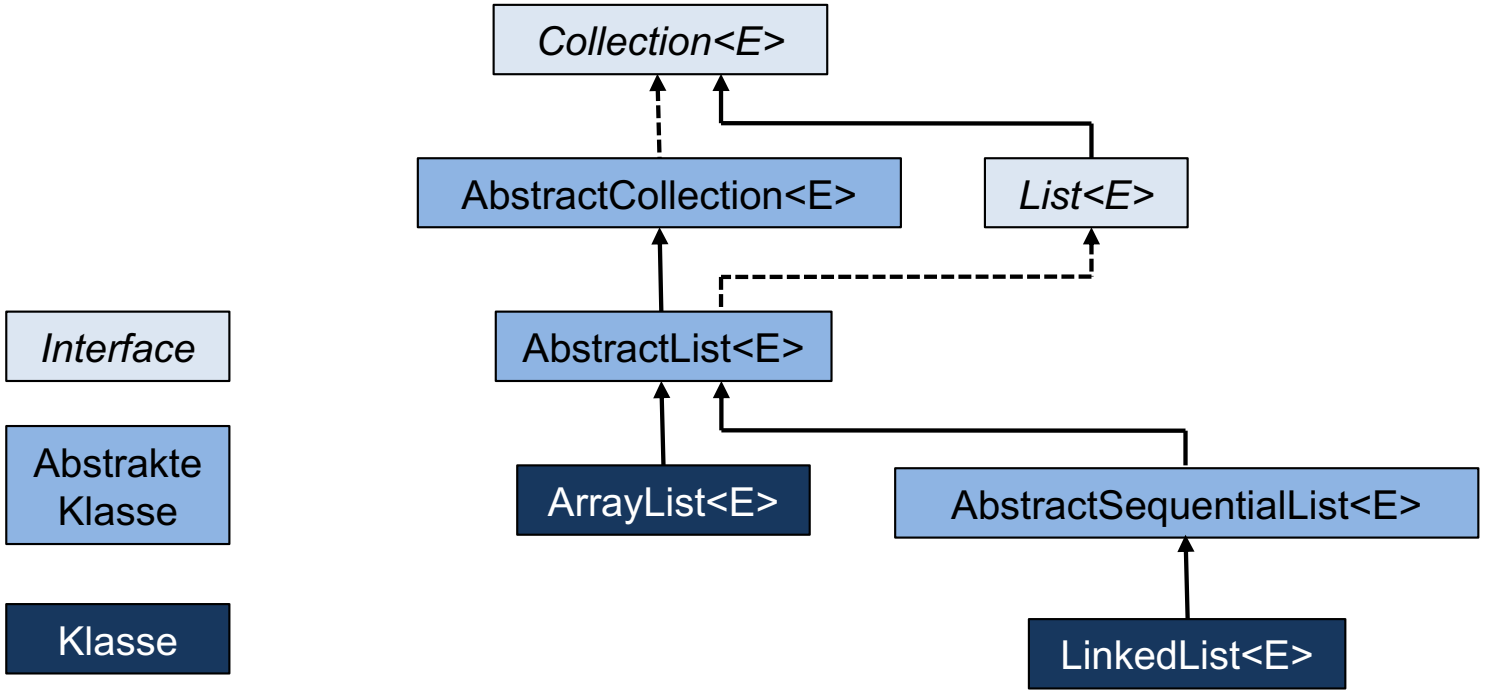


Bemerkungen (2)

- Es sind nur die Typen aus JDK 1.2 Collection aufgeführt.
- Die Typen aus JDK 1.0 Collection wie Vector, Stack, Dictionary und Hashtable sind weggelassen.
  Grund: sie sind wenig performant und sollten vermieden werden.
  Die JDK 1.0 Klassen sind synchronisiert d.h. thread-tauglich. Ein Thread ist ein Ausführungsstrang. Es können mehrere Threads parallel laufen und auf die gleichen Datenstrukturen zugreifen. Dazu müssen die Datenstrukturen synchronisiert werden, was mit einem nicht unerheblichen Overhead verbunden ist.
- Die aufgeführten JDK 1.2 Klassen sind nicht synchronisiert und daher performanter.
- Für den thread-tauglichen Gebrauch gibt es zusätzlich verschiedene Collection-Typen (siehe Kapitel 14).
- Container, die aus Grundlage von Hashverfahren implementiert sind, fehlen ebenfalls (siehe Vorlesung Algorithmen und Datenstrukturen).
- In den verschiedenen Interfaces fehlen außerdem die default-Methoden, die in Java 8 neu dazugekommen sind (siehe Kapitel 13).

Bemerkungen (3)

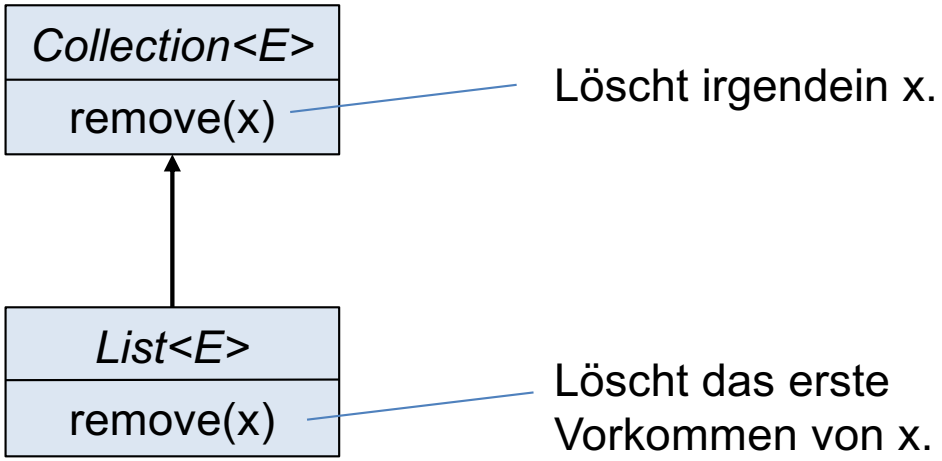
- In der Darstellung fehlen diverse abstrakte Klassen. Beispielweise werden ArrayList<E> und LinkedList<E> über verschiedene abstrakte Klassen durch Vererbung realisiert.



- Abstrakte Klassen stellen Grundfunktionalitäten zur Verfügung und vermeiden Code-Redundanz.
  Außerdem wird das Erstellen eigener Container vereinfacht.

Bemerkungen (4)

- Vor Gebrauch der Container sind unbedingt die Spezifikationen in der Java API zu lesen.
- Es ist insbesondere zu beachten, dass in erweiterten Schnittstellen auch die Spezifikationen verschärft werden können.



Bemerkungen (5)

- Beachte, ob Werte- oder Referenz-Container realisiert werden soll.
- Bei einem Werte-Container (z.B. List<String>) müssen für die Elementtypen bestimmte Methoden überschrieben werden bzw. vorhanden sein:
  - `boolean equals(Object o);` (Gleichheit)
  - `int compareTo(T o);` (Vergleichsoperator)

  Die equals-Methode ist für alle Collection-Typen wichtig.
  Mit der compareTo-Methode werden z.B. die Elemente in TreeSet und TreeMap geordnet. (siehe Kapitel 11).