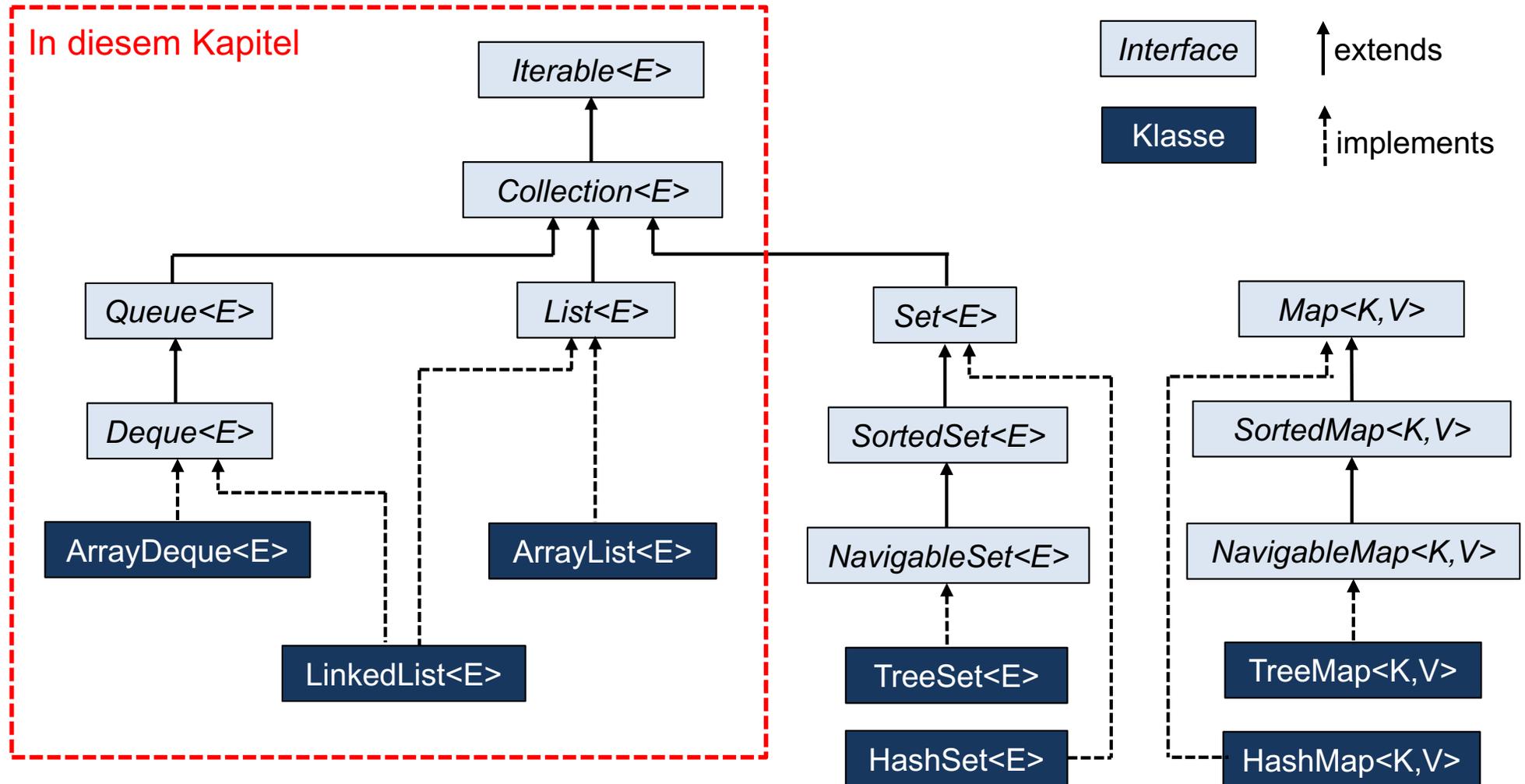


Kapitel 6: Java Collection – Teil I

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

Übersicht über Collection-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);
```

```
[1, 1.5, 2, 2.5]
```

```
System.out.println(intCol);
```

```
[4, 7, 3]
```

```
nbCol.addAll(intCol);
```

```
System.out.println(nbCol);
```

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

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

```
Number n = Integer.valueOf(7);
```

```
intCol.remove(n);
```

```
System.out.println(intCol);
```

```
[4, 3]
```

Number wird aus Integer-Collection gelöscht!

Ist erlaubt, da:

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

```
// 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, 2, 4, 7, 3]
```

```
}
```

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);  
}
```

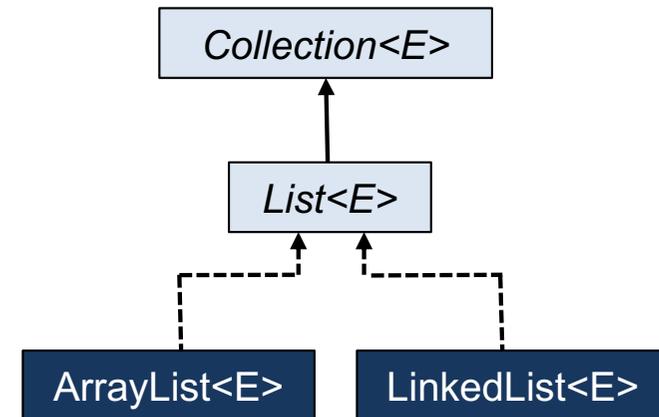
Liste in Feld kopieren

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);  
  
    System.out.println(intList.subList(4, 8));  
  
    intList.subList(4, 8).clear();  
    System.out.println(intList);  
}
```

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

[5, 6, 7, 8]

[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();  
  
}
```

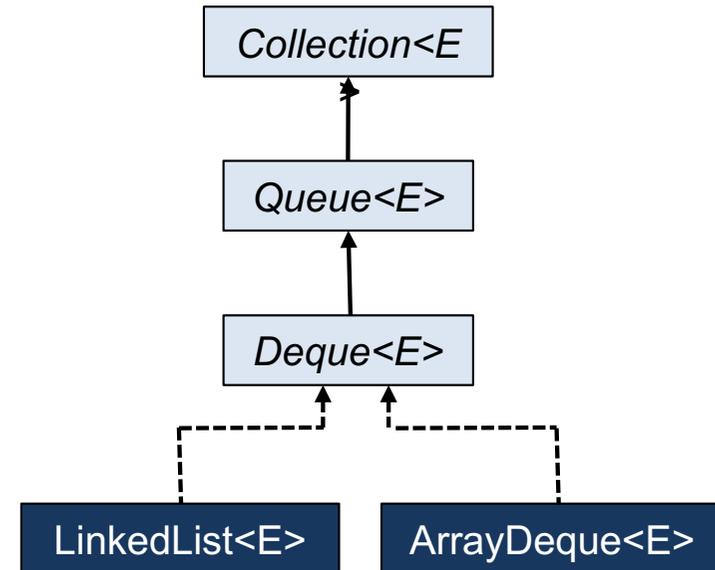
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

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	<code>boolean offer(E x)</code>	<code>boolean add(E x)</code>
Löscht und liefert das vordere Element	<code>E poll()</code>	<code>E remove()</code>
liefert das vordere Element	<code>E peek()</code>	<code>E element()</code>

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	<code>boolean offerFirst(E x)</code>	<code>void addFirst(E x)</code>
Löschen und zurückliefern	<code>E pollFirst()</code>	<code>E removeFirst()</code>
zurückliefern	<code>E peekFirst()</code>	<code>E getFirst()</code>

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

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	Äquivalente Deque-Methode	Exception bzw. spezieller Rückgabewert
<code>void push(E x)</code>	<code>void addFirst(E x)</code>	Exception
<code>E pop()</code>	<code>E removeFirst()</code>	Exception und Rückgabe
<code>E peek()</code>	<code>E peekFirst()</code>	Rückgabe (kann null sein)

Anwendung mit Deque<E>

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

```
    // Schlange als Feld:
```

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 + ", ");
    }
```

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

```
    System.out.println();
```

```
    // Keller als Feld:
```

Keller als Feld.

```
    Deque<Integer> stack = new ArrayDeque<>();
```

```
    for (int i = 1; i <= 10; i++)
        stack.push(i);
```

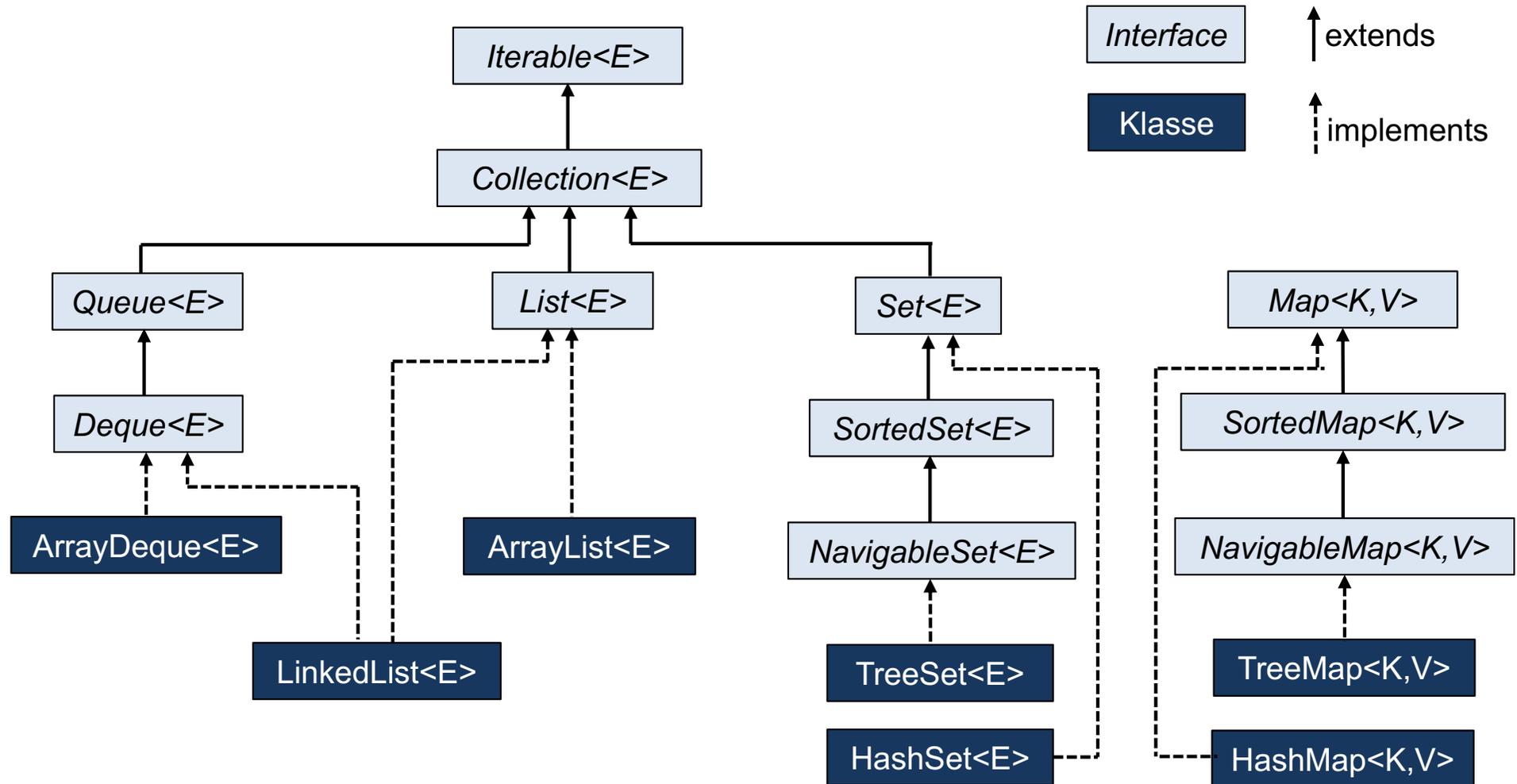
```
    while (!stack.isEmpty()) {
        int x = stack.pop();
        System.out.print(x + ", ");
    }
```

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

```
    System.out.println();
```

```
}
```

Bemerkung: Darstellung nicht vollständig (1)

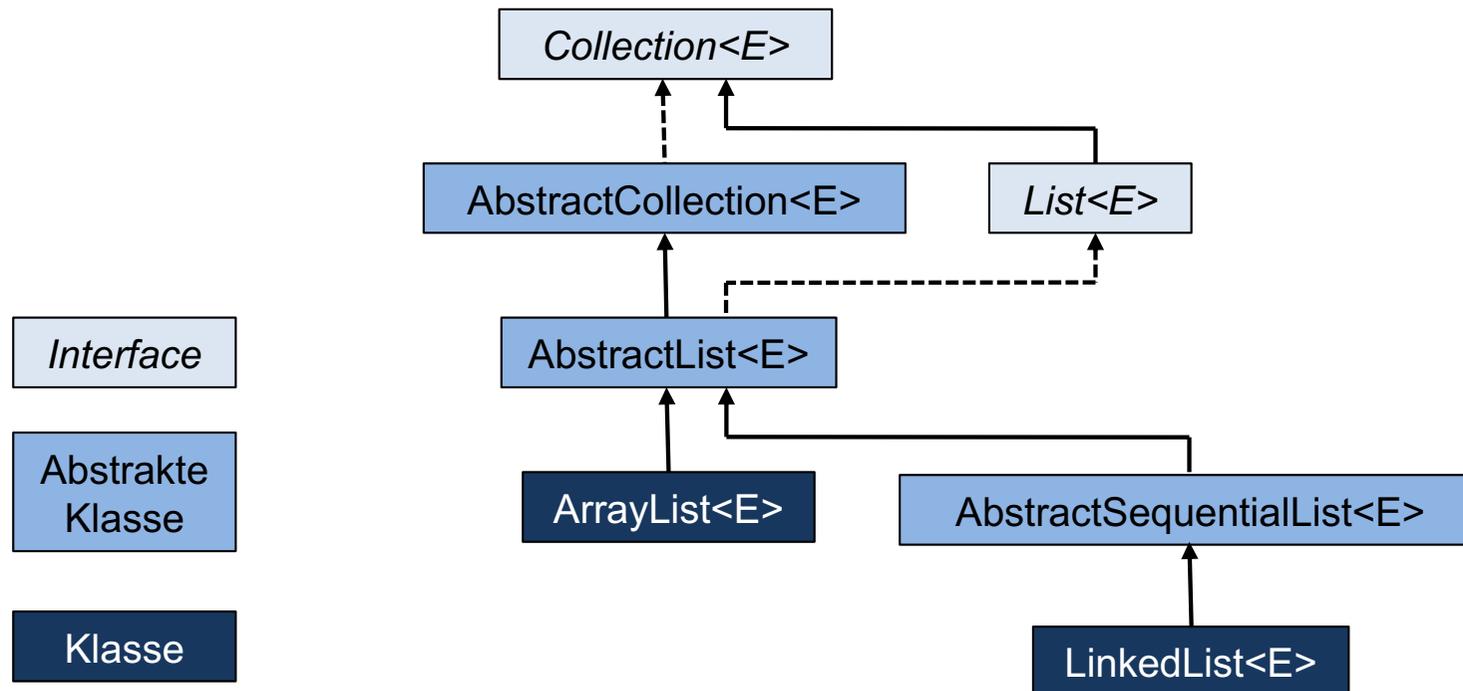


Bemerkung: Darstellung nicht vollständig (2)

- 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).
- In den verschiedenen Interfaces gibt es noch verschiedene default-Methoden, die ab [Java 8](#) neu dazugekommen sind (werden zum Teil in Kapitel 13 behandelt).

Bemerkung: Darstellung nicht vollständig (3)

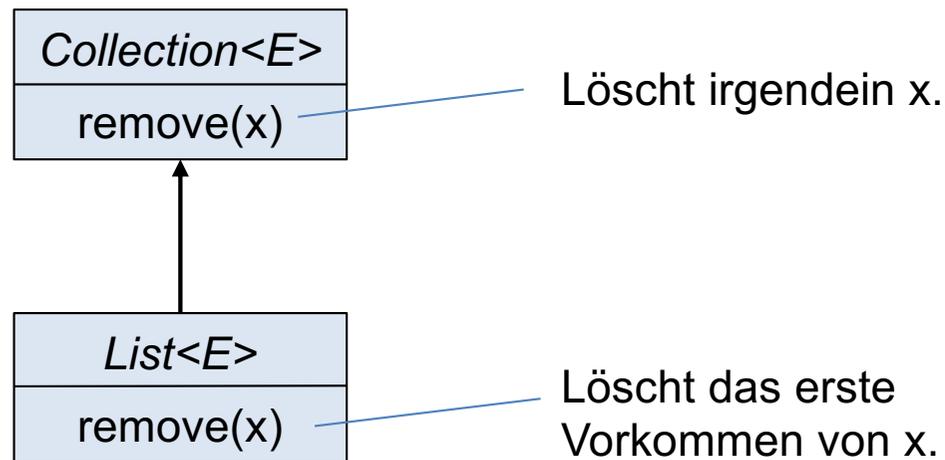
- In der Darstellung fehlen diverse **abstrakte Klassen**. Beispielweise werden `ArrayList<E>` und `LinkedList<E>` über verschiedene abstrakte Klassen durch Vererbung realisiert.



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

Bemerkung: Java-API lesen!!!

- Vor Gebrauch der Container sind unbedingt die Spezifikationen in der [Java API](#) zu lesen.
- Es ist insbesondere zu beachten, dass in abgeleiteten Schnittstellen die Spezifikationen verschärft werden können.



- **Tipp:**
Bei IntelliJ lässt sich mit Mouseover über die Typ- bzw. Methodennamen die Javadoc direkt anzeigen, ohne über die Web-Seiten der Java-API zu gehen!

Bemerkung: Anforderungen an den Elementtyp (1)

- Beispiel: Personencontainer.
Welche Anforderung besteht an Klasse Person?

```
List<Person> persList = new LinkedList<>();  
Set<Person> persTreeSet = new TreeSet<>();  
Set<Person> persHashSet = new HashSet<>();
```

- Was läuft mit class Person schief?

```
class Person {  
    private String name;  
    private int geburtsJahr;  
    public Person(String s, int j) {  
        name = s;  
        geburtsJahr = j;  
    }  
}
```

```
persList.add(new Person("Peter", 2001));    // Don't do this!  
persTreeSet.add(new Person("Uli", 2003));    // Don't do this!  
persHashSet.add(new Person("Ute", 2002));    // Don't do this!  
...
```

Bemerkung: Anforderungen an den Elementtyp (2)

- Anforderung an den Elementtyp (z.B. Person) ergibt sich aus der verwendeten Collection-Klasse:

Collection-Klasse	erforderliche Methoden	Immutabilität	siehe Kap.
ArrayDeque<E> LinkedList<E> ArrayList<E>	equals-Methode für E	egal	6
TreeSet<K> TreeMap<K,V>	equals- und compareTo-Methode für K (oder Comparator beim Konstruktor)	K muss immutabel sein	11
HashSet<K> HashMap<K,V>	equals- und hashCode-Methode für K	K muss immutabel sein	11

- **Tipp:**
 - bei Standardtypen aus der Java-API wie [String](#), [Integer](#), [LocalDate](#), ... ist **nichts zu beachten**.
 - Ansonsten möglichst **record-Typen** verwenden.
record-Typen sind immutabel und enthalten bereits eine equals- und eine hashCode-Methode.

Beispiel: Liste mit Personen

```
public static void main(String[ ] args) {  
  
    record Person(String name, int gebJahr){ }  
  
    List<Person> persList = new LinkedList<>( );  
  
    persList.add(new Person("Peter", 2001));  
    persTreeSet.add(new Person("Uli", 2003));  
    persHashSet.add(new Person("Ute", 2002));  
  
    System.out.println(persList.contains(new Person("Peter", 2001)));  
}
```

Gibt true aus!