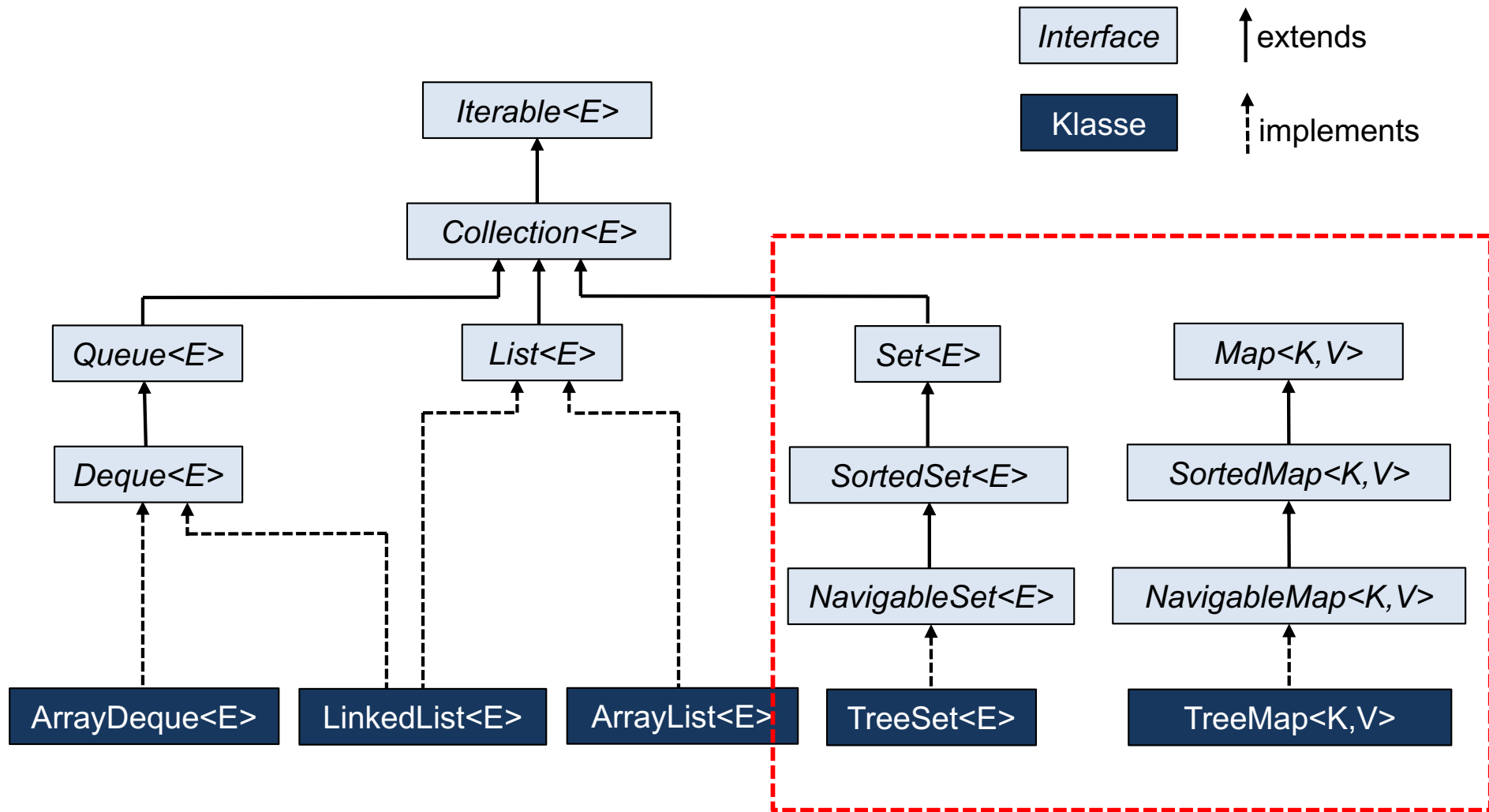


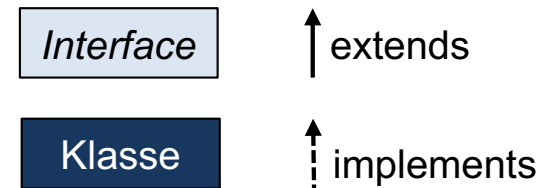
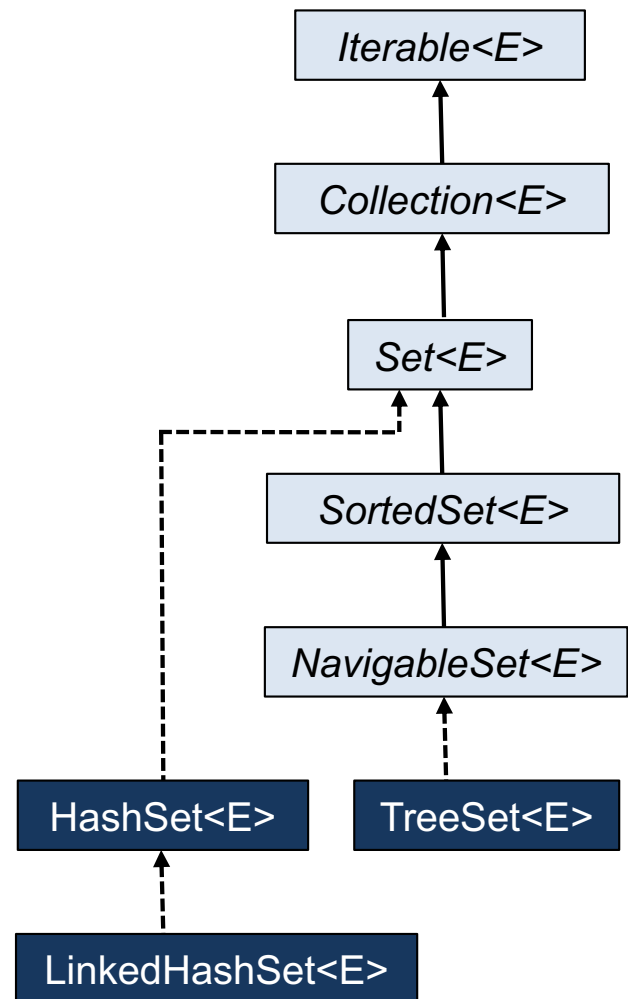
Kapitel 11: Java Collection – Teil II

- Übersicht
- Set und TreeSet
- Map und TreeMap

In diesem Kapitel



Collection: Set und TreeSet

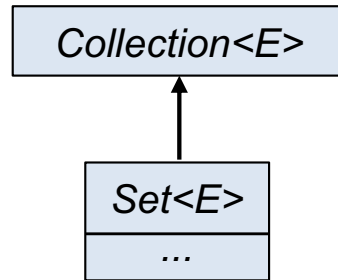


- Auf den folgenden Folien werden nur das Interface `Set` mit ihren Erweiterungen `SortedSet` und `NavigableSet` und die Implementierung `TreeSet` besprochen.
- Hashverfahren und die damit zusammenhängenden `HashSet`-Implementierungen werden im nächsten Semester in Algorithmen und Datenstrukturen besprochen.

Collection<E>

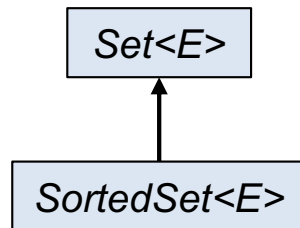
```
public interface Collection<E> extends Iterable<E> {  
  
    boolean add(E e); // add the element e  
    boolean addAll(Collection<? extends E> c); // add the contents of c  
  
    boolean remove(Object o); // remove the element o  
    boolean removeAll(Collection<?> c); // remove the elements 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  
}
```

Set<E>



- Das Interface Set hat gegenüber Collection keine weiteren abstrakten Methoden.
- neue Default-Methoden und statische Methoden wie zB. `copyOf` und `of` (ähnlich wie bei List)
- Vertragsbedingung von `add` und `addAll` werden verschärft.
- `add(x)` von Collection garantiert lediglich, dass sich `x` nach Aufruf von `add(x)` im Container befindet. Es wird keine Aussage für den Fall getroffen, dass sich das Element bereits im Container befindet.
- `add(x)` von Set fügt `x` nur dann zum Container dazu, falls `x` noch nicht im Container enthalten ist.
- `addAll` wird analog verschärft.

SortedSet<E>

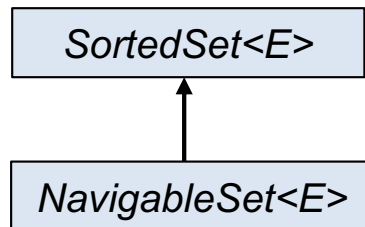


- SortedSet ist eine Menge, deren Elemente sortiert sind.
- Die Elemente sind entweder über compareTo (natürliche Ordnung) oder mit einem Comparator-Objekt sortiert, das typischerweise beim Konstruktoraufruf (siehe TreeSet) übergeben wird.

```
public interface SortedSet<E> extends Set<E> {
    Comparator<? super E> comparator();
    SortedSet<E> subSet(E fromElementInclusive, E toElementExclusive); // returns a range view.
    SortedSet<E> headSet(E toElementExclusive ); // returns a range view.
    SortedSet<E> tailSet(E fromElementInclusive); // returns a range view.
    E first();
    E last();
}
```

- comparator liefert das Vergleichsobjekt zurück, nach dem geordnet wird.
- subSet, headSet und tailSet liefern entsprechende Teilmengen als Sichten (views) zurück.

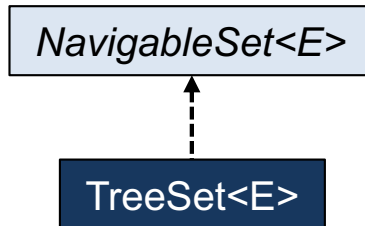
NavigableSet<E>



- Gegenüber `SortedSet` gibt es Navigationsmethoden, die zu einem Element das nächst kleinere bzw. nächst größere Element zurückliefern.
- Statt "kleiner" bzw. "größer" geht auch "kleiner gleich" bzw. "größer gleich".
- Es gibt einen rückwärtslaufenden Iterator.

```
public interface NavigableSet<E> extends SortedSet<E> {  
  
    E lower(E e);           // greatest element less than e, or null if there is no such element  
    E higher(E e);         // least element greater than e, or null if there is no such element  
    E floor(E e);          // greatest element less than or equal to e, or null if there is no such element  
    E ceiling(E e);        // least element greater than or equal to e, or null if there is no such element  
    E pollFirst();  
    E pollLast();  
    NavigableSet<E> descendingSet();           // returns a reverse-order view.  
    Iterator<E> descendingIterator();         // returns a reverse-order iterator.  
    NavigableSet<E> subSet(E fromElement, boolean fromInclusive, E toElement, boolean toInclusive);  
    NavigableSet<E> headSet(E toElement, boolean inclusive);  
    NavigableSet<E> tailSet(E fromElement, boolean inclusive);  
}
```

TreeSet<E>



- TreeSet ist eine Implementierung als balanzierter Binärbaum (genauer: Rot-Schwarz-Baum; siehe Algorithmen und Datenstrukturen im nächsten Semester).
- Die wichtigen Methoden wie `add`, `remove` und `contains` benötigen daher nur eine Laufzeit von $O(\log n)$.
- Der parameterlose Konstruktor setzt eine `compareTo`-Methode für die Elemente voraus.
(Elemente müssen vom Typ `Comparable` sein.)
- Aus Flexibilitätsgründen gibt es auch einen Konstruktor, dem ein Vergleichsobjekt für die Festlegung der Reihenfolge der Elemente übergeben wird.

```
public class TreeSet<E> implements NavigableSet<E> {  
  
    public TreeSet() {...}  
    public TreeSet(Comparator<? super E> comparator) {...}  
    public TreeSet(Collection<? extends E> c) {...}  
    public TreeSet(SortedSet<E> s) {...}  
  
}
```


Anwendungsbeispiel mit TreeSet<E>

- Indexerstellung für eine Datei:
alle Wörter, die in einer Datei vorkommen,
werden alphabetisch ausgegeben.

```
public class Demo {  
  
    public static void main(String[] args) throws FileNotFoundException {  
  
        Scanner in = new Scanner(new File("input.txt"));  
  
        // Wortmenge definieren:  
        NavigableSet<String> words = new TreeSet<>();  
  
        // Alle Woerter aus Datei einlesen und in Menge einfüegen:  
        while (in.hasNext())  
            words.add(in.next());  
  
        // Wortmenge alphabetisch ausgeben:  
        for (String w : words)  
            System.out.println(w);  
    }  
}
```

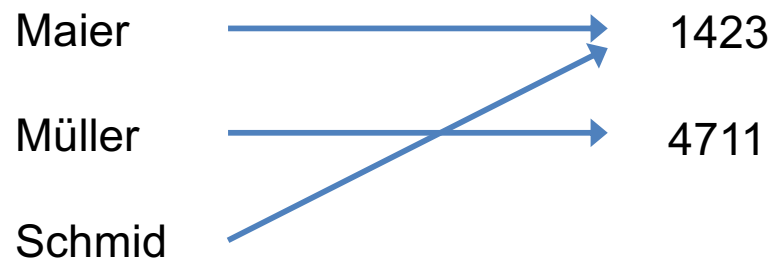
Kapitel 11: Java Collection – Teil II

- Übersicht
- Set und TreeSet
- Map und TreeMap

Map

- Maps sind Mengen von **Schlüssel-Wert-Paaren (key-value-pairs)**, wobei ein Schlüssel nicht mehrfach vorkommen darf.
- Eine Map bildet einen Schlüssel auf einen Wert ab.
Daher auch der Name: Map = Abbildung.
- **Beispiel Telefonbuch:**
 - Schlüssel = Familienname
 - Wert = Telefonnummer

(Es sei angenommen, dass der Familienname eindeutig ist, ansonsten Vorname und Adresse dazu nehmen)



Schlüssel =
Familienname

Wert =
Telefonnummer

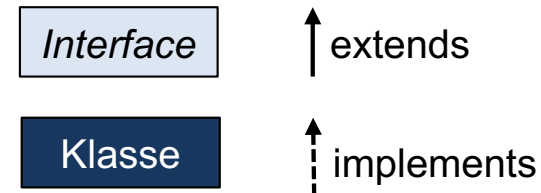
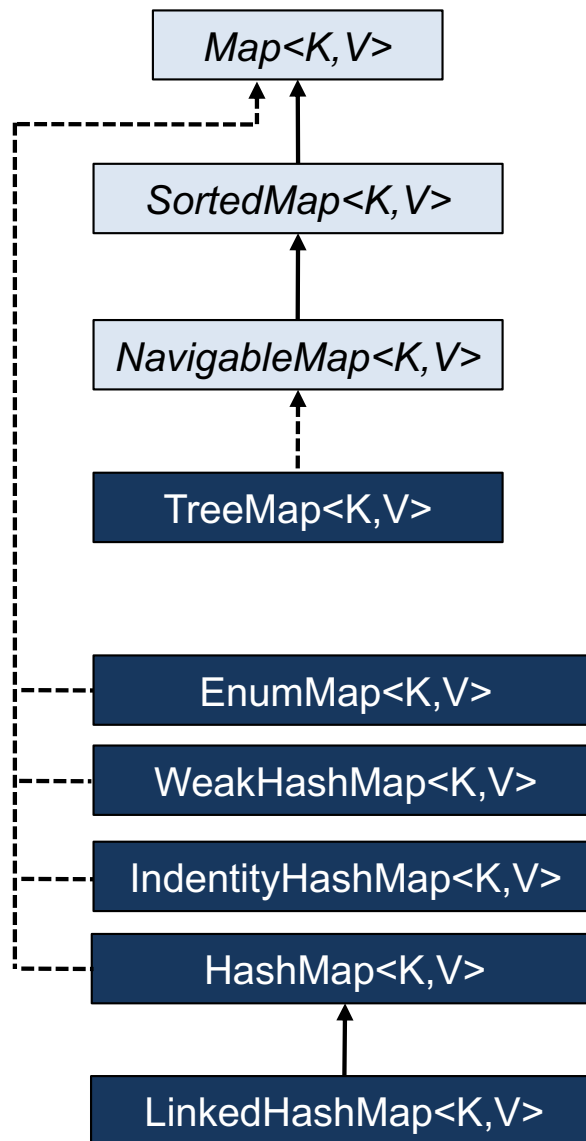
Map<K,V>

- Map<K,V> ist ein generischer Typ.
K steht für Key und ist der Schlüsseltyp. V steht für Value und ist der Werttyp.

```
public interface Map<K, V> {  
  
    V put(K key, V value);           // add or replace a key-value-pair  
    void putAll(Map<? extends K, ? extends V> m); // add all key-value-pairs of m  
  
    void clear();                   // remove all key-value-pairs  
    V remove(Object key);           // remove key-value-pair  
  
    V get(Object key);               // return the value corresponding to key  
    boolean containsKey(Object key); // return true if key is present in the map  
    boolean containsValue(Object value); // return true if value is present in the map  
    boolean isEmpty();              // true if no key-value-pair is present  
    int size();                     // number of key-value-pairs  
  
    Set< Map.Entry<K, V> > entrySet(); // return a Set view of the key-value-pairs  
    Set<K> keySet();                 // return a Set view of the keys  
    Collection<V> values();          // return a Collection view of the values  
}
```

- Map.Entry<K,V> ist der Typ für die Schlüssel-Wert-Paare.
Es gibt u.a. die Methoden getKey(), getValue() und setValue(V value).
- Es gibt für Maps keine Iteratoren!

Map<K,V> und ihre Implementierungen



- `SortedMap`, `NavigableMap` und `TreeMap` sind analog zu `SortedSet`, `NavigableSort` und `TreeSet` aufgebaut. Die Elemente sind nach Ihrem Schlüssel sortiert.
- `TreeMap` bietet wie erwartet einen Konstruktor an, dem ein Vergleichsobjekt für die Festlegung der Reihenfolge der Schlüssel übergeben werden kann.
- Hashverfahren und die damit zusammenhängenden `HashMap`-Implementierungen werden im nächsten Semester in Algorithmen und Datenstrukturen besprochen.

Anwendung: Telefonbuch als TreeMap (1)

```
public class TelBuchAnwendung{

    public static void main(String[] args) {

        NavigableMap<String,Integer> telBuch = new TreeMap<>();

        // Kunden eintragen:
        telBuch.put("Maier", 1234);
        telBuch.put("Anton", 4567);
        telBuch.put("Meyer", 4711);
        telBuch.put("Mueller", 7890);
        telBuch.put("Vogel", 1357);
        telBuch.put("Baier", 2468);

        // TelNummer nachschlagen:
        Integer telNr;
        if ((telNr = telBuch.get("Vogel")) != null) {
            System.out.println("Vogel: " + telNr);
        }

        // TelNummer aendern:
        telBuch.put("Maier", 4321);

        ...
    }
}
```

Telefonbuch definieren.

Teilnehmer eintragen.

Telefonnummer nachschlagen.

Telefonnummer ändern.

Anwendung: Telefonbuch als TreeMap (2)

Anton: 4567
Baier: 2468
Maier: 4321
Meyer: 4711
Mueller: 7890
Vogel: 1357

```
...  
  
// TelBuch sortiert ausgeben:  
for (Map.Entry<String,Integer> eintrag : telBuch.entrySet()) {  
    System.out.println(eintrag.getKey() + ": " + eintrag.getValue());  
}  
  
// Nur Kundennamen des TelBuchs ausgeben:  
for (String kunde : telBuch.keySet()) {  
    System.out.println(kunde);  
}  
  
// Bereichssichten: TelBuch nur mit 'M' ausgeben:  
System.out.println("Telefonbucheinträge mit M:");  
for (Map.Entry<String,Integer> eintrag  
    : telBuch.subMap("M", true, "N", false).entrySet()) {  
    System.out.println(eintrag.getKey() + ": " + eintrag.getValue());  
}  
}
```

Anton
Baier
Maier
Meyer
Mueller
Vogel

Maier: 4321
Meyer: 4711
Mueller: 7890

Typinferenz durch das Schlüsselwort var

- Mit Java 10 kann bei lokalen Variablen das Schreiben komplizierter Typausdrücke vermieden werden, indem der universelle Typ **var** benutzt wird.
- Der Compiler leitet durch Typinferenz den Typ selbst her.

```
// TelBuch sortiert ausgeben:  
for (Map.Entry<String, Integer> eintrag : telBuch.entrySet()) {  
    System.out.println(eintrag.getKey() ...);  
}
```

kann kürzer geschrieben werden durch:

```
// TelBuch sortiert ausgeben:  
for (var eintrag : telBuch.entrySet()) {  
    System.out.println(eintrag.getKey() ...);  
}
```


Aufgabe - Indexerstellung

- Schreiben Sie ein Programm, das für eine Eingabedatei input.txt einen Index erstellt und ausgibt.
- Ein Index ist eine alphabetisch sortierte Folge der im Text vorkommenden Wörter. Für jedes Wort werden außerdem die Nummern der Zeilen angegeben, in denen das Wort vorkommt.
- Verwenden Sie geeignete Container aus der Java-API.

