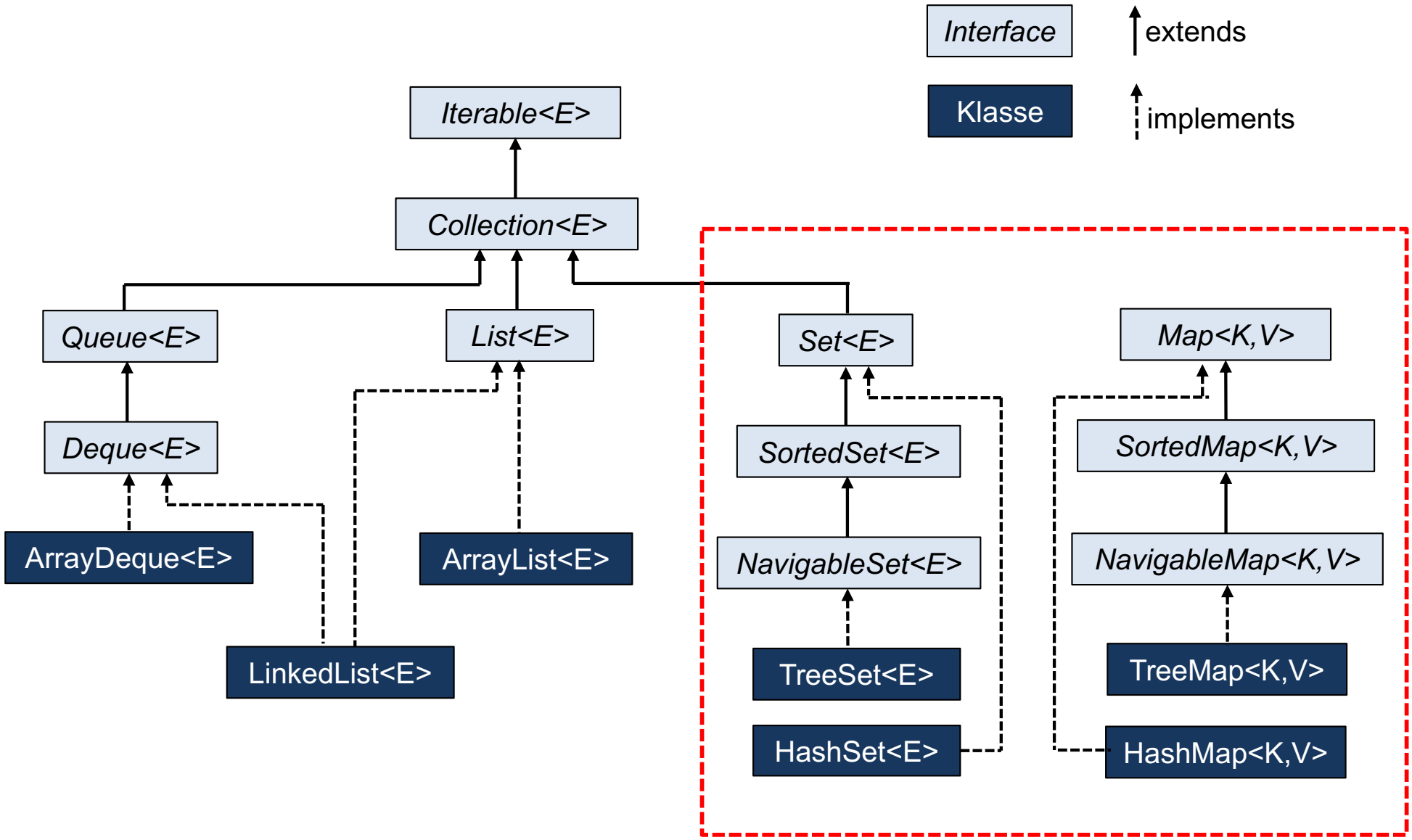
Kapitel 11: Java Collection – Teil II

- Übersicht
- Set und TreeSet
- Map und TreeMap

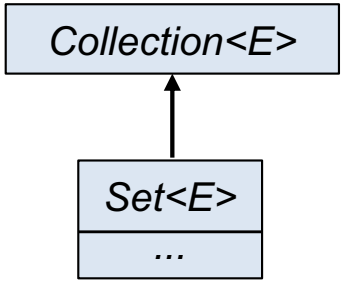
In diesem Kapitel



Wiederholung von Collection<E>

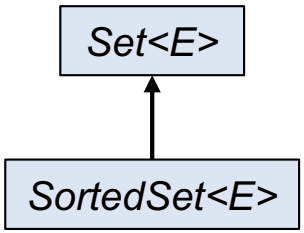
```
public interface Collection<E> extends Iterable<E> {

   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.
- Jedoch werden Vertragsbedingung von add und addAll verschärft.
  Set ist eine Menge im mathematischen Sinne!
  - 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.
- neue Default-Methoden und statische Methoden
  wie z.B. copyOf und of (ähnlich wie bei List)

SortedSet<E>

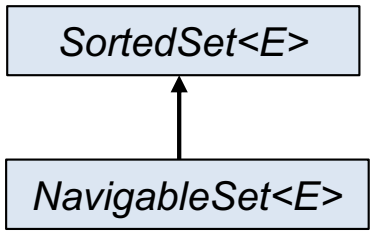


- SortedSet ist eine Menge, deren Elemente sortiert sind.
- Die Elemente sind entweder über compareTo (natürliche Ordnung) oder mit einem Comparator-Objekt sortiert, das typischerweise beim Konstruktoraufruf (siehe TreeSet) übergeben wird.

```
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.

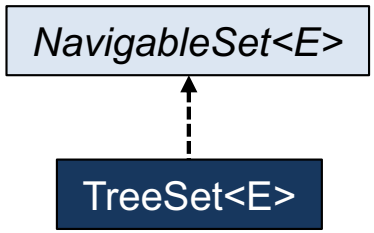
NavigableSet<E>



- 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 als balanzierter binärer Suchbaum implementiert (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 Elementtyp E voraus. (E muss vom Typ Comparable sein.)
- Aus Flexibilitätsgründen gibt es auch einen Konstruktor, dem ein Comparator übergeben wird.
- Elementtyp E muss immutabel sein.

```
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) {...}
}
```

Beispiel: Indexerstellung mit TreeSet

- 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 einfuegen:
        while (in.hasNext())
            words.add(in.next());

        // Wortmenge alphabetisch ausgeben:
        for (String w : words)
            System.out.println(w);
    }
}
```

Beispiel: Verwaltung von Personendaten mit TreeSet

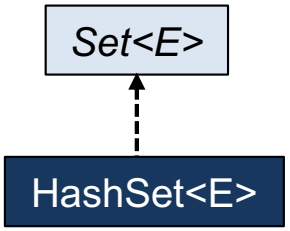
- Verwaltung von Personen-Daten mit einem binären Suchbaum (TreeSet<Person>)
- Wie sollen Personen im Suchbaum angeordnet werden?
- Dazu muss entweder Person eine compareTo-Methode anbieten oder beim TreeSet-Konstruktor wird ein geeigneter Comparator übergeben.

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

        record Person(String name, int gebJahr){ }

        SortedSet<Person> pers = new TreeSet<>( (p1,p2)-> p1.toString().compareTo(p2.toString()) );
        pers.add(new Person("Sonia", 2001));
        pers.add(new Person("Peter", 2003));
        pers.add(new Person("Maria", 2002));
        pers.add(new Person("Sonia", 2004)); // bereits vorhanden

        for (Person p : pers)
                System.out.println(p);
}
```

Comparator für Person:
Personen werden im TreeSet alphabetisch nach Namen+GebJahr geordnet.

Personen werden alphabetisch nach Namen+GebJahr sortiert ausgegeben.

HashSet<E>



- HashSet ist eine Implementierung als Hashtabelle (siehe Algorithmen und Datenstrukturen im nächsten Semester).
- Die Elemente haben eine „zufällige“ Anordnung.
- Die wichtigen Methoden wie add, remove und contains benötigen eine Laufzeit von nur O(1)! (Gilt nur unter bestimmten Voraussetzungen)
- Der Elementtyp E benötigt eine hashCode- und eine equals-Methode. Tipp: records verwenden!
- Elementtyp E muss immutabel sein.

```
public class HashSet<E> implements Set<E> {

    public HashSet() {...}
    public HashSet(Collection<? extends E> c) {...}
    // ...
}
```

Beispiel: Indexerstellung mit HashSet

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

```
public class Demo {

    public static void main(String[] args) throws FileNotFoundException {

        Scanner in = new Scanner(new File("input.txt"));

        // Wortmenge definieren:
        Set<String> words = new HashSet<>();

        // Alle Woerter aus Datei einlesen und in Menge einfuegen:
        while (in.hasNext())
            words.add(in.next());

        // Wortmenge ausgeben:
        for (String w : words)
            System.out.println(w);
    }
}
```

Beispiel: Verwaltung von Personendaten mit HashSet

- Verwaltung von Personen-Daten in einer Hashtabelle (HashSet<Person>)
- Da Person als record definiert ist, werden eine hashCode- und eine equals-Methode automatisch bereitgestellt.

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

        record Person(String name, int gebJahr){ }

        Set<Person> pers = new HashSet<>( );
        pers.add(new Person("Sonia", 2001));
        pers.add(new Person("Peter", 2003));
        pers.add(new Person("Maria", 2002));
        pers.add(new Person("Sonia", 2004));    // bereits vorhanden

        for (Person p : pers)
                System.out.println(p);
}
```

Personen werden in „zufälliger“ Reihenfolge ausgegeben.

- Beachte: mit class Person {...} ohne hashCode-Methode würde Beispiel nicht wie erwartet funktionieren.
- Tipp: record Person verwenden oder in class Person {...} benötigte Methoden mit IDE generieren.

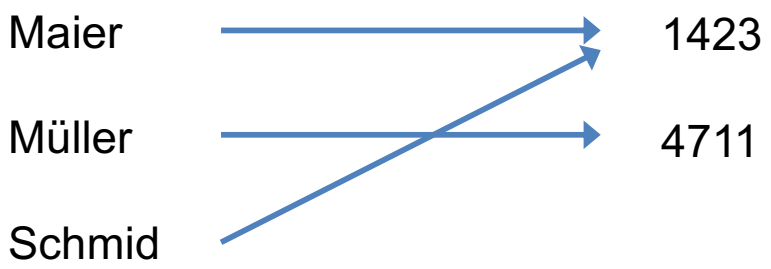
Kapitel 11: Java Collection – Teil II

- Übersicht
- Set und TreeSet
- Map und TreeMap

Map

- Eine Map ist eine Menge 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

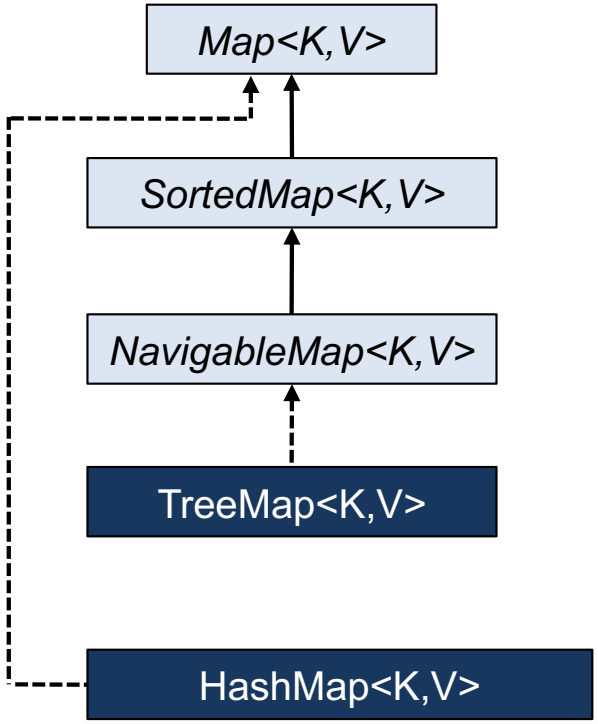
Map<K,V>

- 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, NavigableSet und TreeSet aufgebaut.
- TreeMap ist ein binärer Suchbaum. Schlüsseltyp K muss Comparable sein oder beim Konstruktor muss ein Comparator übergeben werden.
- HashMap ist eine Hashtabelle. K muss eine hashCode-Methode anbieten.
- Schlüsseltyp K für TreeMap und HashMap muss immutabel sein.

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)

```
    ...

    // 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("Telefonbucheintaege mit M:");
    for (Map.Entry<String,Integer> eintrag
            : telBuch.subMap("M", true, "N", false).entrySet()) {
        System.out.println(eintrag.getKey() + ": " + eintrag.getValue());
    }

}
```

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

```
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()...);
}
```

Anwendung: Telefonbuch als HashMap

```
public class TelBuchAnwendung{

    public static void main(String[] args) {

        Map<String,Integer> telBuch = new HashMap<>();

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

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

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

    }
}
```

Telefonbuch definieren.

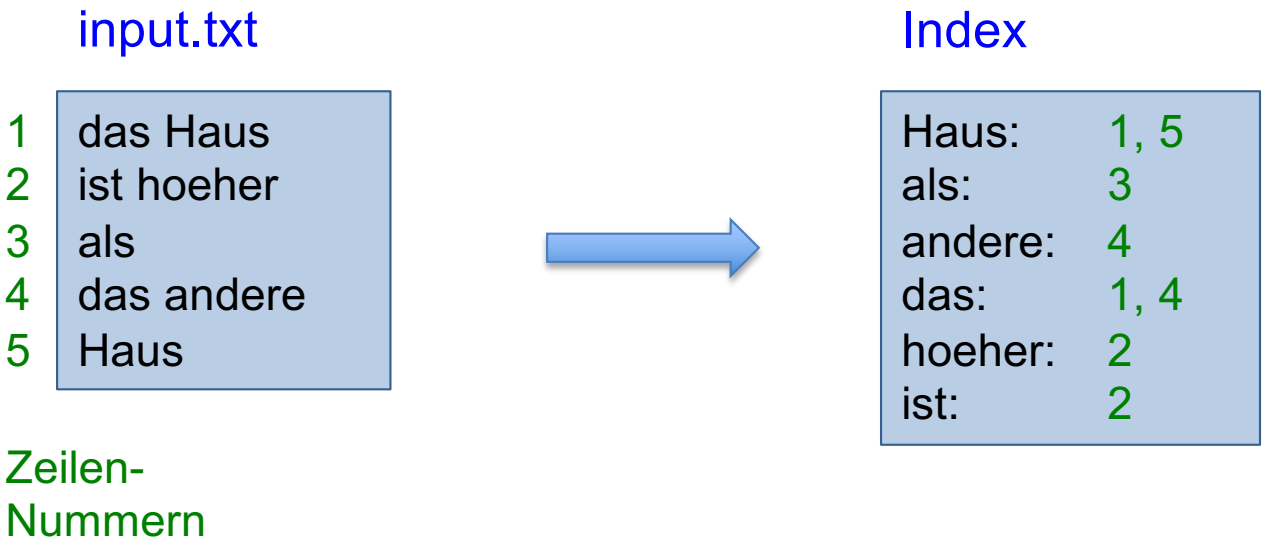
Teilnehmer eintragen.

Telefonnummer nachschlagen.

Telefonnummer ändern.

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.



Bemerkung: Anforderungen an den Elementtyp

- 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 hashCode-Methode.