PM2 Java: Collections

Motivation und Einführung

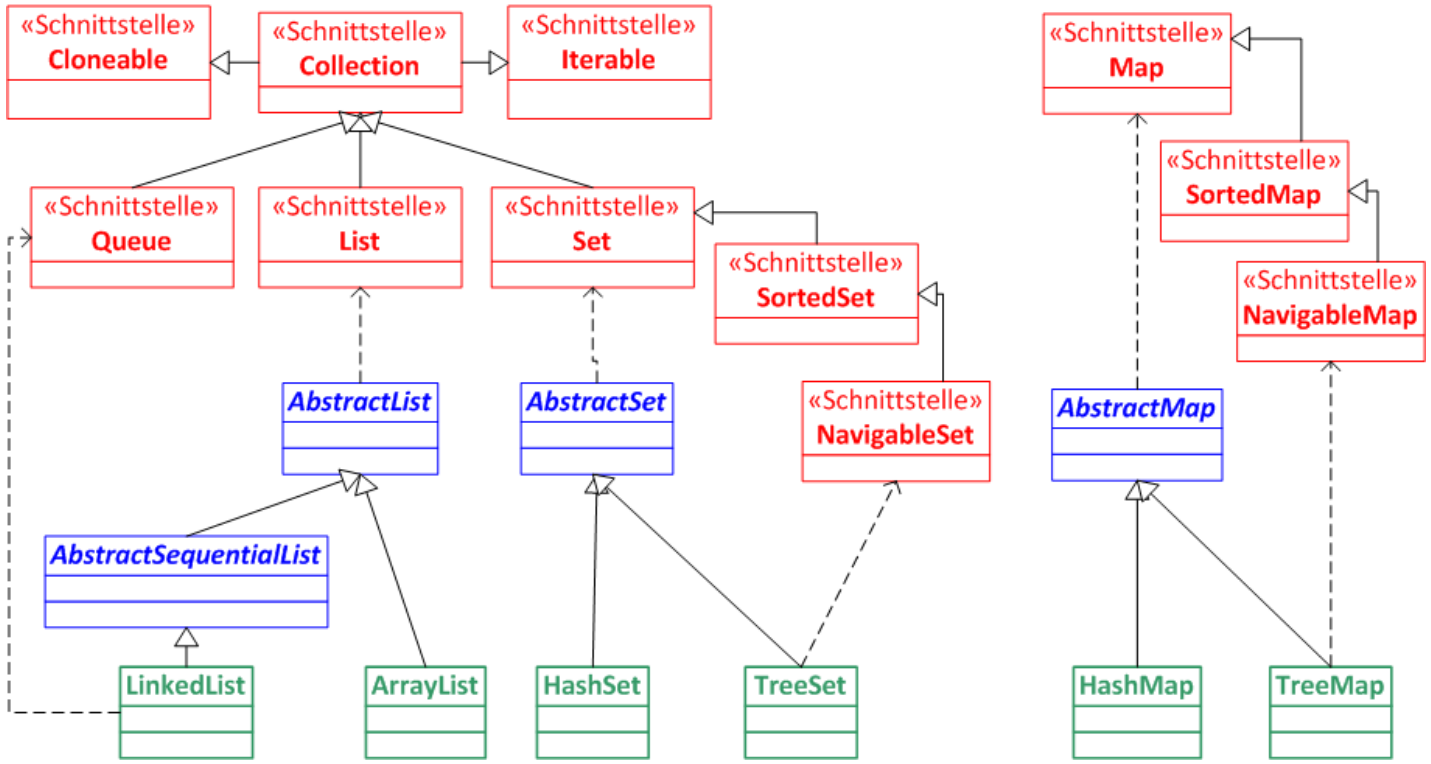
- In der Programmierung verwenden wir an vielen Stellen Objektsammlungen
  - PDA's speichern Notizen und Termine
  - Bibliotheken verwalten Informationen über Bücher und Zeitschriften
  - iTunes verwaltet Musik, Filme, Podcasts etc.
  - Webauktionen verwalten Gebote und Angebote
  - etc.
- Allen diesen Sammlungen ist gemein, dass sich ihre Größe dynamisch ändern kann.
- Objektsammlungen mit fester Größe kennen wir bereits (Java Arrays).
- Objektsammlungen flexibler Größe werden in Java als **Collection** bezeichnet.
- Das Java Collection-Framework enthält Datentypen für geordnete / ungeordnete Sammlungen und Sammlungen mit / ohne Duplikate.
- Neben den eigentlichen Collection Typen enthält das Collection Framework Maps als Datentypen für das Verwalten von Wörterbuch-ähnlichen Strukturen.

AUFBAU DES COLLECTION FRAMEWORKS

Übersicht

extends

implements



Legacy-Klassen und -Interfaces

- Neben den generischen Interfaces und Klassen des Collection Frameworks existieren die sog. **Legacy Klassen und Interfaces**. Sie stammen aus Java 1.1 und sind alle nicht generisch.

- Sie sind in das generische Framework integriert, um Kompatibilität zu „älteren“ Java-Programmen (Legacy Code) zu gewährleisten.

- Die Utility-Klasse ***Collections*** erfüllt für Objektsammlungen die gleiche Funktion wie ***Arrays*** für die Array-Familie.

AbstractCollection<T> (implements Collection<T>)
AbstractList<T> (implements List<T>)
AbstractSequentialList<T>
LinkedList<T> (implements List<T>)
ArrayList<T> (implements List<T>)
…
Vector (implements List) (Java 1.1) **Legacy**
Stack
AbstractSet<T> (implements Set<T>)
HashSet<T> (implements Set<T>)
TreeSet<T> (implements SortedSet<T>)
AbstractMap<K,V> (implements Map<K,V>)
HashMap<K,V> (implements Map<T>)
TreeMap<K,V> (implements SortedMap<K,V>)
…

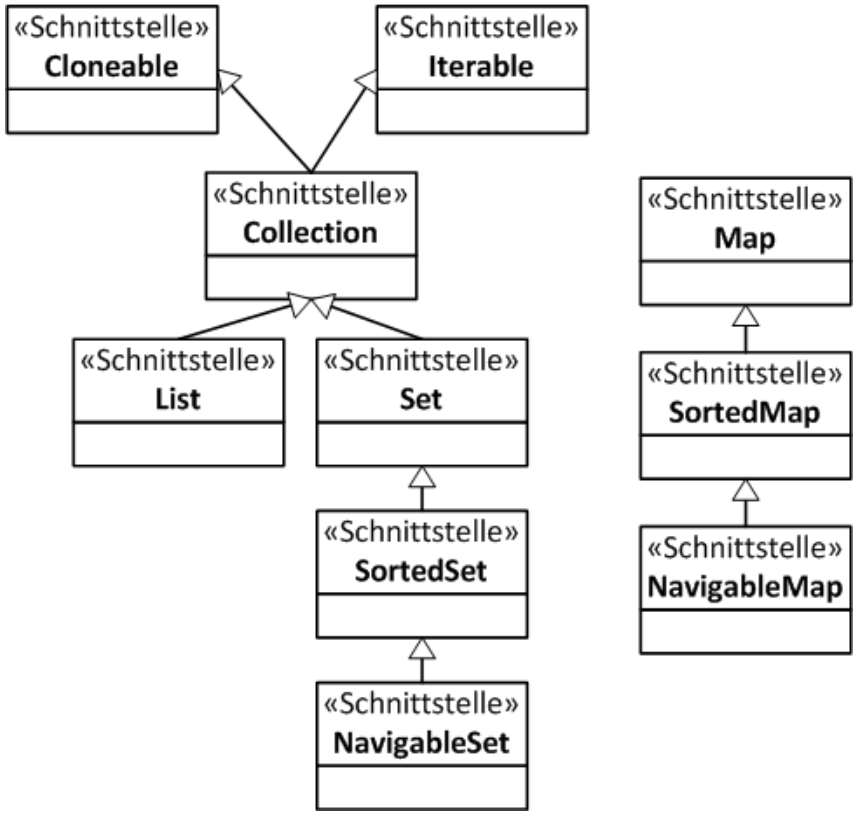
Collections *(repository for searching, sorting etc.)*
Arrays *(repository for searching, sorting methods etc.)*

System *(repository for arraycopy() etc.)*
BitSet *(operations on large bit sequences)*

Dictionary (Java 1.1) **Legacy**
Hashtable (implements Map)
Properties

Interfaces / Schnittstellen des Collection Frameworks

- Basis für das Collection Framework
- definieren verschiedene Typen von Objektsammlungen
- trennen Typ und Implementierung
- zwei unabhängigen Hierarchien:
  - Collection
  - Map
- Collections
  - kopierbar (***Cloneable<T>***)
  - iterierbar (***Iterable<T>***)
- generische Interfaces



Interfaces des Collection-Frameworks

Collection<T>

- Basisverhalten aller Objekt-Sammlungen
- **Verwendung**: Sammlungen zwischen Methoden transportieren und ein Maximum an Allgemeinheit erreichen.
- Objektsammlungen wachsen dynamisch

Set<T> und List<T>

- ***Set*: Menge** von Objekten
  - keine Dubletten
  - keine Ordnung
- ***List*: Sequenz** von Objekten
  - Dubletten erlaubt
  - Anordnung über die Position
  - indizierter Zugriff auf Elemente

Interfaces des Collection-Frameworks

Map<K,V>

- eine Tabelle, in der Schlüssel (***K*** für keys) Werten (***V*** für values) zugeordnet werden
- **keine doppelten Schlüssel**
- ***Map*** auch als tabellierte Funktion verstehen.
- ***Map*** ist in Java das Äquivalent zu Hashes in Ruby.

Anordnung von Objekten

Ordnungen von Objekten

- **2 Techniken**:

- ***Comparable<T>*** Interface
  - natürliche Ordnung für Objekte von Klassen definiert, die das Interface implementieren
  - Klasse **intern** gibt die Ordnung vor

- ***Comparator<T>*** Interface
  - für **externe** / **zusätzliche** Ordnungs-kriterien

SortedSet<T> und *SortedMap<T>*

- ***SortedSet***: eine Menge, die Elemente in aufsteigender Ordnung verwaltet. Das ***SortedSet*** Interface wird z.B. verwendet um Wortlisten zu verwalten.

- ***SortedMap***: Eine ***SortedMap*** ist eine Tabelle, die die Schlüssel in aufsteigender Ordnung verwaltet. Das ***SortedMap*** Interface wird z.B. für Wörter- oder Telefonbücher verwendet.

- Beide Interface verlangen, das Elemente ***Comparable*** sind.

COLLECTIONS UND METHODEN

Interface Collection<T>

```
public interface Collection<T> extends Iterable<T> ... {
    // Basic Operations
    int size();
    boolean isEmpty();
    boolean contains(Object element);
    /**
     * Optional nur für modifizierbare Collections
    *  true, wenn sich die Collection nach add ändert.
     * false, wenn keine Dubletten erlaubt sind und element bereits enthalten ist.
     */
    boolean add(Object element);
    // Optional nur für modifizierbare Collections
    boolean remove(Object element);
    Iterator<T> iterator();
    // Massenoperationen
    boolean containsAll(Collection<?> c); // "Teilmenge"
    boolean addAll(Collection<? extends T>  c); // "Vereinigungs-Menge"
    boolean removeAll(Collection<?> c); // „Mengendifferenz"
    boolean retainAll(Collection<?> c); // „Schnittmenge"
    void clear();                       // Löschen
    // Array Operationen
    Object[] toArray();
    <U> U[] toArray(U a[]);
}
```

Interface Collection<T>

Methoden

- Das Interface definiert Methoden, die
  1. die Anzahl der Elemente zurückgeben oder prüfen (***size, isEmpty***),
  2. prüfen, ob ein gewisses Objekt in einer Collection ist (***contains***),
  3. Elemente einer Collection hinzufügen oder löschen (***add***, ***remove***)
  4. sukzessive auf die Elemente der Collection zugreifen (***iterator***).
  5. 2. und 3. als Massenoperation durchführen.
  6. eine Collection in ein Array konvertieren (***toArray(), toArray(T[] tAry)***).

Nicht modifizierbare Collections

- Die Methoden
  - ***add***, ***remove*** und die destruktiven Massenoperationen sind nur für modifizierbare Collections erlaubt.
- Das **Standardverhalten** der Klassen ***AbstractList<T>*** und ***AbstractSet<T>*** ist das einer **nicht modifizierbaren** ***Collection***.

Iteratoren

Über Collections iterieren

Varianten

1. Mit dem Iterator (***iterator()***) einer Collection und den Methoden ***hasNext()*** und ***next().***

2. Mit dem for-each Konstrukt

- Ein Iterator in Java ist ein Objekt vom Typ ***Iterator<T>*** und implementiert die Methoden ***hasNext(),next()*** und ***remove()***.

- Der Iterator merkt sich intern über die Position eines Lesezeigers (**cursor**), welches Element zuletzt gelesen wurde, liefert mit ***next()*** das nachfolgende Elemente und kann Auskunft darüber geben, ob noch Elemente vorhanden sind.

```
List<Object> lo = Arrays.asList(new
    Object[] { 1, "hallo", 4.5 });

Collection<Object> c = new
          ArrayList<Object>(lo);

for (Iterator<Object> citer =
    c.iterator(); citer.hasNext();) {
    p(citer.next());
}

for (Object object : c) {
    p(object);
}
```

Cursor-Positionen von Iteratoren

- Der Cursor eines Iterators vor dem nächsten zu lesenden Element.
- Bei einer Collection mit ***n*** Elementen gibt es ***n*** gültige Cursor-Positionen vor dem nächsten Element und ***n+1*** Cursor-Positionen (***0..n***).
- Zu Beginn steht der Cursor vor dem ersten Element auf Position 0.
- Mit ***next*** wird der Cursor um eine Position verschoben.
- Wurde die Collection vollständig durchlaufen, dann steht der Cursor hinter dem letzten Element.
- Hat der Cursor die Position hinter dem letzten Element erreicht, dann liefert ***hasNext() false***.

Über Collections iterieren

Arbeitsweise von foreach

- Das foreach Konstrukt setzt voraus, dass Klassen, die sich wie Sammlungen verhalten, das Interface ***Iterable<T>*** implementieren.
- ***Iterable<T>*** fordert die Implementierung der Methode ***public Iterator<T> iterator();***
- foreach ermittelt den Iterator einer Sammlung und verwendet dessen Methoden ***hasNext()*** und ***next()***, um über die Elemente der Sammlung zu iterieren.

```
List<Object> lo = Arrays.asList(new
    Object[] { 1, "hallo", 4.5 });

Collection<Object> c = new
          ArrayList<Object>(lo);

for (Object object : c) {
    p(object);
}
for (Iterator<Object> citer =
    c.iterator(); citer.hasNext();) {
    p(citer.next());
}
```

Das Interface *Iterator<E>*

- Definiert das Standardverhalten eines Iterators, mit den Methoden ***hasNext()*** und ***next()*** für nicht destruktives Iterieren.
- Die Methode ***remove*** entfernt das Element einer Collection, das vom letzten ***next*** zurückgeliefert wurde.

```
public interface Iterator<E> {
    boolean hasNext();
    E next();
    // Optional nur für
    // modifizierbare Collections
    void remove();
}
```

Beispiel: Löscht alle Zahlen aus c!

```
static <T> Collection<T>
          retainNum(Collection<T> c){
     for (Iterator<T> i = c.iterator();
        i.hasNext();)
          if ((i.next() instanceof
             Number))
             i.remove();
     return c;
}
```

Iterator<E> Löschen nur nach Aufruf von *next*

- Pro Aufruf von ***next*** darf nur **ein** ***remove*** Aufruf erfolgen. Sonst generiert die Laufzeitumgebung eine ***IllegalState Exception***.

```
List<Integer> li = new ArrayList<Integer>(Arrays.asList(new Integer[] {
1, 2, 1, 1, 1, 1, 6 }));
Iterator<Integer> iiter = li.iterator();
// Illegal State Exception
// remove darf nur nach next aufgerufen werden
iiter.remove();
```

```
Exception in thread "main" java.lang.IllegalStateException
at java.util.AbstractList$Itr.remove(Unknown Source)
at iteratoren.IllegalModificationsDemo.main(IllegalModificationsDemo.java:17)
```

Iterator<E> keine konkurrierende Modifikation während des Iterierens

- Während ein Iterator über eine Sammlung von Objekte läuft, darf sich die Anzahl der Elemente in der Liste nicht verändern.

```
List<Integer> li = new ArrayList<Integer>(Arrays.asList(new Integer[] {
1, 2, 1, 1, 1, 1, 6 }));
Iterator<Integer> iiter = li.iterator();
// keine konkurriende Modifikationen
while (iiter.hasNext()) {
    li.remove(3);
    if (iiter.next() % 2 == 1)
          iiter.remove();
}
```

```
Exception in thread "main" java.util.ConcurrentModificationException
at java.util.AbstractList$Itr.checkForComodification(Unknown Source)
at java.util.AbstractList$Itr.next(Unknown Source)
at iteratoren.IllegalModificationsDemo.main(IllegalModificationsDemo.java:22)
```

Iterator<E> keine konkurrierende Modifikation während des Iterierens

- Während ein Iterator über eine Sammlung von Objekte läuft, darf sich die Anzahl der Elemente in der Liste nicht verändern.
- ***iiter2*** löscht Elemente der zugrundeliegenden Collection, dann kann der ***iiter1*** nicht mehr verlässlich arbeiten.

```
List<Integer> li = new ArrayList<Integer>(Arrays.asList(new Integer[] {
1, 2, 1, 1, 1, 1, 6 }));
Iterator<Integer> iiter1 = li.iterator();
// keine konkurriende Modifikationen
Iterator<Integer> iiter2 = li.iterator();
while (iiter1.hasNext()) {
    iiter2.next();
    iiter2.remove();
    iiter1.next();
}
```

```
Exception in thread "main" java.util.ConcurrentModificationException
at java.util.AbstractList$Itr.checkForComodification(Unknown Source)
at java.util.AbstractList$Itr.next(Unknown Source)
at iteratoren.IllegalModificationsDemo.main(IllegalModificationsDemo.java:29)
```

Löschen während des Iterierens nur mit *Iterator.remove*

- ***Iterator.remove*** sollte eingesetzt werden, um Elemente während des Iterierens zu löschen, da nur dadurch ein definiertes Löschen und korrektes Anpassen des Lese-Cursors gewährleistet ist.

- Wir sehen unten das korrekte Beispiel für Löschen während des Iterierens. Es werden alle Zahlen == 1 aus der Liste gelöscht.

```
List<Integer> li = new ArrayList<Integer>(Arrays.asList(new
                                                Integer[]{1,1,2,1,1,3,1,1,6}));
Iterator<Integer> iiter = li.iterator();
while (iiter.hasNext()){
     if (iiter.next() ==1 )
        iiter.remove();
}
p(li);
```

[2, 3, 6]

Löschen während des Iterierens nur mit *Iterator.remove*

- Iterieren wir mit einer Zählschleife über eine Liste und löschen die Elemente über den Index, dann wir jedes 2‘te Vorkommen der 1 in einer Abfolge von 1‘en übersprungen.

- Das Beispiel unten zeigt das falsche Verhalten im Resultat nach dem Löschen.

```
li = new ArrayList<Integer>(Arrays.asList(new Integer[]{1,1,2,1,1,3,1,1,6}));
for(int i = 0; i <li.size(); i++){
     if (li.get(i) == 1)
        li.remove(i);
}
p(li);
```

[1, 2, 1, 3, 1, 6]

Eigene Iteratoren schreiben

Eigene Iteratoren schreiben

- Die Methode ***iterator***, des Interfaces ***Iterable*** liefert ein ***Iterator*** Objekt.
- Wir benötigen ***Iterator***-Objekte, wenn wir geschachtelt über eine Objektsammlung iterieren müssen.
- Würde unsere Sammlung das ***Iterator***-Interface selbst zu implementieren, gäbe es zu einer Sammlung immer nur einen ***Iterator***.

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

Eigene Iteratoren schreiben

- Für die Methoden des Interfaces ***Iterator<T>*** ist das Verhalten wie folgt definiert.
- ***hasNext(): true***, wenn noch ein weiteres Element vorhanden ist, sonst ***false***.
- ***next()***: liefert das nächste Element, wenn vorhanden, sonst wirft die Methode eine ***NoSuchElementException***
- ***remove***(): entfernt das letzte mit ***next()*** gelesene Element aus der zugrunde liegenden Objektsammlung. Wurde ***next()*** nicht aufgerufen, wird eine ***IllegalState-Exception*** ausgelöst. Darf die Sammlung nicht verändert werden, wir eine ***UnsupportedOperationException*** ausgelöst.

```
public interface Iterator<E> {

    boolean hasNext();

    E next();

    void remove();
}
```

Eigene Iteratoren schreiben

- Iteratoren sind sehr eng an die zugrundeliegende Objektsammlung geknüpft. Insbesondere benötigen Sie Zugriff auf die interne Datenstruktur der Sammlung.
- Iteratoren werden daher in den allermeisten Fällen als **private innere Klassen** oder **anonyme Klassen** definiert (siehe eine der nächsten Vorlesung.)
- Nachfolgende Folie skizziert den Iterator für die Klasse ***Liste.***

```
public interface Iterator<E> {

    boolean hasNext();

    E next();

    void remove();
}
```

Eigene Iteratoren schreiben

```
public class Liste {
 private Object[] elements;
 ...
 public Liste(int initCap) {
     elements = new Object[initCap];
     this.initCap = initCap;
     last =0; }
 public void add(Object elem){
     if (last == elements.length) {
     System.arraycopy(elements, 0,
        elements.length, 0,
        last+initCap);
     }
     last++;
     elements[last] = elem;
}
public Iterator<Object> iterator() {
    return new ListeIterator();
}
private class ListeIterator implements
    Iterator<Object> { ... }
}
```

```
private class ListeIterator implements
    Iterator<Object> {

     @Override
     public boolean hasNext() {
        ...
     }

     @Override
     public Object next() {
     ...
     }

     @Override
     public void remove() {
     throw new
        UnsupportedOperationException();
     }
}
```

Eigene Iteratoren schreiben

- ***ListeIterator*** implementiert das generische Interface ***Iterator<T>***.
- Nur die Liste hat Zugriff auf den privaten Iterator.
- Die Typvariable des Iterators entspricht dem Komponententyp von ***Liste<T>***.
- Der ***readCursor*** ist jetzt ein Attribut des Iterators und nicht mehr der Liste.
- ***remove*** wird mit ***Liste.this*** (***siehe nächste Vorlesung***) auf das umgebende Listen-Objekt delegiert.
- ***nextGiven*** merkt sich, ob vorher ein Element gelesen wurde.

```
private class ListeIterator implements
    Iterator<T> {
     private int readCursor=0;
     private boolean nextGiven=false;
     public boolean hasNext() {
         return readCursor<writeCursor;
     }
     public T next() {
          if (!hasNext()) throw new
             NoSuchElementException();
          nextGiven = true;
          return elements[readCursor++];
     }
     public void remove() {
          if (nextGiven) {
             Liste.this.remove(readCursor-
             1);
             readCursor--;
          }
          else throw new
             IllegalStateException();
          nextGiven = !nextGiven;
          }
     }
}
```

MENGEN UND LISTEN

Das *Set<T>* Interface

- Interface ***Set*** Subtyp von Interface ***Collection***.
- ***Set*** definiert keine zusätzlichen Methoden.
- ***Set*** ist ein **Marker-Interface**, das die semantische Eigenschaften von Mengen zusichert.
- Implementierungen
  - ***HashSet<T>***: Verwaltung von Schlüsseln in einer Hashtabelle
  - ***TreeSet<T>:*** Sortierung von Schlüsseln.

```
public interface Set<T> extends
  Collection<T> {
  int size();
  boolean isEmpty();
  boolean contains(Object element);
  boolean add(Object element);
  boolean remove(Object element);
  Iterator<T> iterator();
  // Massenoperationen
  boolean containsAll(Collection<?> c);
  boolean addAll(Collection<? extends T>
  c);
  boolean removeAll(Collection<?> c);
  boolean retainAll(Collection<?> c);
  void clear();
  // Array Operationen
  Object[] toArray();
  <U> U[] toArray(U a[]);
}
```

Anwendungsbeispiel für Sets

- Sets können z.B. verwendet werden, um Duplikate in Eingaben zu entdecken und zu eliminieren.
- Im Beispiel wird die Menge als (***Set<T>***) und deklariert und nicht über einen konkreten Typ (***HashSet<T>***).
- So lassen sich Implementierungen zukünftig leicht austauschen.

```
public class FindDups {
    public static void main(String args[])
    {
      Set<String> s = new
          HashSet<String>();
            for (String a : args)
              if (!s.add(a))
              p("Duplikat: " + a);

      p(s.size() + " Worte ohne Duplikat:
          " + s);
      }
}
```

```
java collections.FindDups 34 nur nur ein
    Duplikat
```

```
Duplikat: nur
4 Worte ohne Duplikat: [Duplikat, ein,
    34, nur]
```

Das Interface *List<E>*

- Interface ***List<E>*** ist ein Subtyp des Interfaces ***Collection<E>*** plus Erweiterungen
- Implementierungen:
  - ***ArrayList***
  - ***LinkedList***.
- indizierter Zugriff auf die Elemente
- Suche nach einem Element liefert einen Positionsindex
- **Iteratoren**:
  - Iteratoren wie für alle Sammlungen
  - spezielle ListIteratoren: Iterieren in beide Richtungen ab einem beliebigen Index.
- Definition von Teillisten über Indexbereiche

```
public interface List<E> extends
    Collection<E> {
     // Indizierter Zugriff
     E get(int index);
     // Optionale Methoden
     E set(int index, Object element);
     boolean add(int index, E element);
     E remove(int index);
     abstract boolean addAll(int index,
        Collection<? extends E> c);
     // Optionale Methoden Ende

     // Suche über den Index
     int indexOf(Object o);
     int lastIndexOf(Object o);
     // Listeniteratoren
     ListIterator<E> listIterator();
     ListIterator<E> listIterator(int
        index);
     // Teillisten
     List<E> subList(int from, int to);
}
```

Listen-Iteratoren

- Iterieren in zwei Richtungen (***next,previous***)
- Modifikation einer Liste während der Iteration (***set,remove,add***)
- Auskunft über nächsten/vorhergehenden Index bzgl. der aktuellen Cursor-Position (***nextIndex, previousIndex***).

```
public interface ListIterator<E>
    extends Iterator<E> {
     boolean hasNext();
     E next();

     boolean hasPrevious();
     E previous();

     int nextIndex();
     int previousIndex();
    // Optionale Methoden
     void remove();
     void set(E o);
     void add(E o);
    // Ende Optionale Methoden
}
```

Verwendung des *ListIterator* (s)

- Iterationsmuster für Listen:
  - Traversieren von vorne nach hinten
  - Traversieren von hinten nach vorne
  - Traversieren in einem definierten Bereich

```
public static void main(String[] args) {
List<Character> li = new
    ArrayList<Character>(Arrays.asList(new
    Character[] {
'a','b', 'c' }));
ListIterator<Character> liter =
    li.listIterator();

// Listen von vorne durchlaufen
while(liter.hasNext()){
p("next_i: " + liter.nextIndex());
p("prev_i: " +liter.previousIndex());
p("e     : " + liter.next());
}
p("next_i: " + liter.nextIndex());
p("prev_i: " +liter.previousIndex());

// Listen von hinten durchlaufen
p("-----------------------------------");
while(liter.hasPrevious()){
p("next_i: " + liter.nextIndex());
p("prev_i: " +liter.previousIndex());
p("e     : " + liter.previous());
}
p("next_i: " + liter.nextIndex());
p("prev_i: " +liter.previousIndex());
```

Typische Verwendung vom *ListIterator*

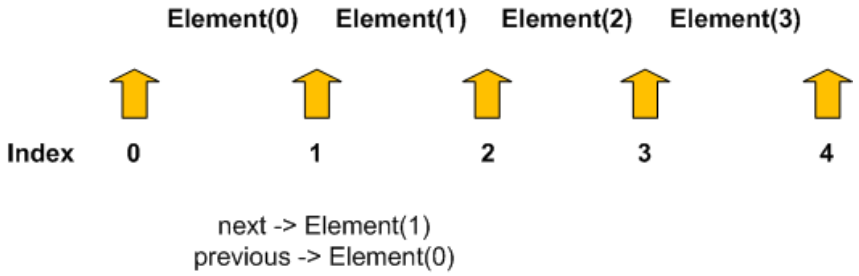
```
next_i: 0
prev_i: -1
e     : a
next_i: 1
prev_i: 0
e     : b
next_i: 2
prev_i: 1
e     : c
next_i: 3
prev_i: 2
-----------------------------------
next_i: 3
prev_i: 2
e     : c
next_i: 2
prev_i: 1
e     : b
next_i: 1
prev_i: 0
e     : a
next_i: 0
prev_i: -1
```

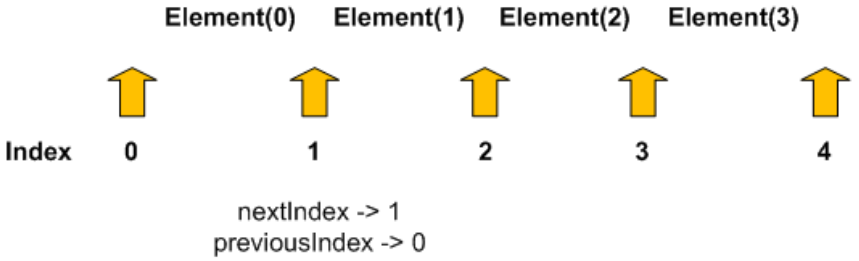
```
public static void main(String[] args) {
List<Character> li = new ArrayList<Character>(
          Arrays.asList(new Character[] {'a','b', 'c' }));
ListIterator<Character> liter = li.listIterator();

// Listen von vorne nach hinten durchlaufen
while(liter.hasNext()){
p("next_i: " + liter.nextIndex());
p("prev_i: " +liter.previousIndex());
p("e     : " + liter.next());
}
p("next_i: " + liter.nextIndex());
p("prev_i: " +liter.previousIndex());

// Listen von hinten nach vorne durchlaufen
p("-----------------------------------");
while(liter.hasPrevious()){
p("next_i: " + liter.nextIndex());
p("prev_i: " +liter.previousIndex());
p("e     : " + liter.previous());
}
p("next_i: " + liter.nextIndex());
p("prev_i: " +liter.previousIndex());
```

Cursor-Positionen von Listen-Iteratoren

- Länge ***n*** dann ***n+1*** gültige Index Werte:
  - Zwischenräume: n-1 Positionen
  - vor dem ersten Element: 1 Position
  - nach dem letzten Elemente: 1 Position

- Cursor zwischen zwei Elementen:
  - ***next*** liefert das Element hinter der Position
  - ***previous*** liefert das Element vor der Position
  - ***nextIndex*** liefert den Index des Elements, das als nächstes gelesen wird.
  - ***previousIndex*** liefert den Index des Elements, das zuvor gelesen wurde.





Cursor-Positionen von Listen-Iteratoren

- Mischen von ***next*** und ***previous:***
  - Aufruf von ***previous*** nach ***next*** liefert das gleiche Element wie der ***next*** Aufruf zuvor.
  - Aufruf von ***next*** nach einem ***previous*** liefert das gleiche Element wie der ***previous*** Aufruf zuvor.
- ***nextIndex*** und ***previousIndex*** liefern den Index des Elementes, das beim nächsten ***next/previous*** gelesen würde.
- ***nextIndex*** ist immer um 1 größer als ***previousIndex***.
- **Grenzfälle**:
  1. ***previousIndex*** liefert -1, wenn der Cursor vor dem ersten Element steht.
  2. ***nextIndex*** liefert ***list.size(),*** wenn der cursor hinter dem letzten Element steht.

Index eines Elementes einer Liste ermitteln

```
public static <T> int findIndex(List<T> list, T elem){
     for (ListIterator<T> liter = list.listIterator(); liter.hasNext();) {
        if (elem == null ? liter.next()== null : elem.equals(liter.next()))
          return liter.previousIndex();
     }
return -1;
}
```

- Iterieren über die Liste ***list*** bis ***elem*** gleich dem nächsten Element von ***list*** ist.
- Durch ***next*** steht der Cursor hinter dem gefundenen Element.
- ***previousIndex*** liefert den Index des gesuchten Elements.
- **Sonderfall**: ***elem*** ist ***null***, dann wird das nächste Element mit == auf ***null*** geprüft.

Beispiel: Palindrome erkennen

- Ein Palindrom ist eine Zeichenkette, die von vorne wie von hinten gelesen gleich ist.
  - **Wortpalindrome**: Abba, Otto, Kajak, Lagerregal, Radar, Uhu,
  - **Satzpalindrom**: Trug Tim eine so helle Hose nie mit Gurt

- **Aufgabe**: Schreiben Sie eine Methode, die Palindrome erkennt.

- **Lösung**:
  - Wandeln einer Zeichenkette in ein Character-Array
  - Übertragen in eine Liste von Zeichen.
  - Zwei List-Iteratoren, der eine startet am Anfang, der andere am Ende der Liste .
  - Zeichenweise vergleichen bis ***previousIndex*** des rückwärts laufenden Iterators <= ***nextIndex*** des vorwärts laufenden Iterators ist.

Beispiel: Palindrome erkennen

```
public class PalindromErkenner {

    public static void main(String[] args) {
        p(istPalindrom("Uhu"));
        p(istPalindrom("Trug Tim eine so helle Hose nie mit Gurt"));
    }
    private static boolean istPalindrom(String string) {
        char[] chars = string.toLowerCase().toCharArray();
        List<Character> lc = new ArrayList<Character>();
        for (Character c : chars) {
            if (!Character.isSpaceChar(c)) {
                lc.add(c);
            }
        }
        ListIterator<Character> lcif, lcib;
        lcif = lc.listIterator();
        lcib = lc.listIterator(lc.size());
        while (lcif.hasNext() && lcib.hasPrevious() && (lcib.previousIndex() > lcif.nextIndex())) {
            Character cf, cb;
            if ((cf = lcif.next()) != (cb = lcib.previous())) {
                return false;
            }
            p("cf=" + cf + ":cb=" + cb);
        }
        return true;
    }
}
```

Beispiel Listenvergleich

- Eine Methode ***listCompare(List l1, List l2)*** gibt
  - → -1 zurück, wenn l1 < l2
  - → 0 zurück, wenn l1 inhaltsgleich l2
  - → +1 zurück, wenn l1 > l2

- l1 < (>) l2 gdw l1 und l2 einen gemeinsamen Präfix haben und für das erste Element (e1,e2), in dem sich l1 und l2 unterscheiden, gilt dass e1 < (>) e2, oder wenn l1 (l2) echter Präfix von l2 (l1)

- sonst sind l1 inhaltsgleich l2

Listenvergleich

```
private static <T extends Comparable<? super T>> int  listCompare(List<T> l1, List<T> l2) {
    Iterator<T> l1Iter = l1.iterator();
    Iterator<T> l2Iter = l2.iterator();

    /*
     * l1 und l2 unterscheiden sich in einem Element
     */
    while (l1Iter.hasNext() && l2Iter.hasNext()) {

        T l1Next = l1Iter.next();
        T l2Next = l2Iter.next();
        if (l1Next.compareTo(l2Next) !=0) {
            return l1Next.compareTo(l2Next);
        }
    }
    /*
     * l2 ist echter Präfix von l1
     */
    if (l1Iter.hasNext() && ! l2Iter.hasNext()) {
        return 1;
    }
    /*
     * l1 ist echter Präfix von l2
     */
    if (l2Iter.hasNext() && ! l1Iter.hasNext()) {
        return -1;
    }
    // l1 und l2 sind inhaltsgleich
    return 0;
}
```

Utility für Collection Klassen

UTILITY KLASSE COLLECTIONS

Die Utilityklasse *Collections*

- statische Methoden die auf Collections arbeiten
- Algorithmen sind zum großen Teil polymorph
  - arbeiten mit allen Implementierungen von ***Collection, List*** und ***Set***.
  - die meisten beziehen sich auf den ***List*** Typ
- Generatoren für nicht modifizierbare Sichten auf Collections, z.B. die Methode ***unmodifiableSet.***

```
public class Collections {

public static Object max(Collection coll)
public static Object max(Collection coll,
          Comparator comp)
// analog min
// ...
public static int binarySearch(List list,
                    Object key)
public static void copy(List to, List or)
public static void reverse(List list)
public static void sort(List list)
public static void sort(List list,
                    Comparator comp)

//Generatoren: nicht-modifizierbare Colls
public static Set unmodifiableSet(Set
                    aSet)
// ...
```

Generische Methoden der Klasse *Collections* (seit Java 5)

```
public class Collections {

    public static <T extends Comparable<? super T>> T max(Collection<? extends T>
       c)
    public static <T> T max(Collection<? extends T> coll,Comparator<? super T>
       comp)

    public static <T> int binarySearch(List<? extends Comparable<? super T>> l,
                                                    T key)
    public static <T> void copy(List<? super T> dest, List<? extends T> src;
    public static <T> void fill(List<? super T> list, T o)
    public static <T>  void reverse(List<?> list)
    public static <T extends Comparable<? super T>>  void sort(List<? extends T>
       l)
    public static <T> void sort(List<? extends T> l,Comparator<? super T> comp)
    public static <T> List<T> unmodifiableList(List<? extends T> list)
    public static <T> Set<T> unmodifiableSet(Set<? extends T> list)
    ... USW
}
```

Die Methode der Klasse *Collections* sind polymorph

```java
public class CollectionsMethods {
    public static void main(String[] args) {
        List<String> lst1 = new ArrayList<String>(
                Arrays.asList(new String[] { "eins", "zwei", "drei", "vier", "fuenf" }));
        List<Integer> lst2 = new LinkedList<>(Arrays.asList(1, 2, 3, 4, 5));
        p("Original: " + lst1);
        polymorphMethodsColls(lst1);
        p("Original: " + lst2);
        polymorphMethodsColls(lst2);
    }

    private static <T extends Comparable<? super T>> void polymorphMethodsColls(List<? extends T> aList) {
        p("   * executing: polymorphMethodsColls");
        p("   * shuffle, sort, reverse arbeiten destruktiv");
        Collections.shuffle(aList);
        p("random: " + aList);
        Collections.sort(aList);
        p("sort: " + aList);
        Collections.reverse(aList);
        p("reverse: " + aList);
    }
}
```

```
Original: [eins, zwei, drei, vier, fuenf]
  * executing: polymorphMethodsColls
Original: [eins, zwei, drei, vier, fuenf]
  * executing: polymorphMethodsColls
  * shuffle, sort, reverse arbeiten destruktiv
random: [zwei, drei, fuenf, vier, eins]
sort: [drei, eins, fuenf, vier, zwei]
reverse: [zwei, vier, fuenf, eins, drei]
Original: [1, 2, 3, 4, 5]
  * executing: polymorphMethodsColls
  * shuffle, sort, reverse arbeiten destruktiv
random: [3, 4, 2, 1, 5]
sort: [1, 2, 3, 4, 5]
reverse: [5, 4, 3, 2, 1]
```

Ordnung von Objekten und Sortieren von Listen

- Um Liste sortieren zu können (***Collections.sort***) muss für die Elemente der Liste eine Ordnung definiert sein.
- ➔ Klasse der Objekte muss das Interface ***Comparable<T>*** implementieren.
- ***Comparable<T>*** fordert die Implementierung der ***compareTo(<T> other)*** Methode. Die Methode gibt ***0*** zurück, wenn ***other*** gleich ist, >= ***1***, wenn ***other*** kleiner und <= ***-1*** wenn ***other*** größer ist.
- Die Alternative ist die Übergabe eines Comparators (***Comparator<T>***), der die Ordnungsrelation in der Methode ***compare(T o1, T o2)*** implementiert.

```
class MyComparator implements
    Comparator<Integer> {
    public int compare(Integer j1,
          Integer j2) {
     if (j1 > j2)
     return +1; // aufsteigend
     if (j1 < j2)
     return -1; // aufsteigend
     else
     return 0;
     }
}
```

Ordnung von Objekten und Sortieren von Listen

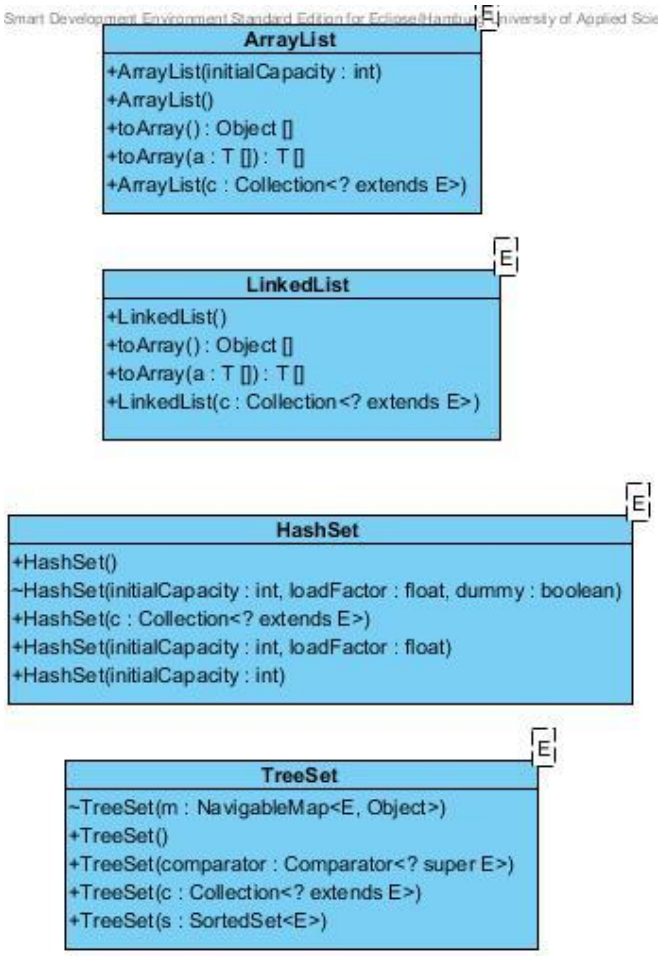
- ***MyComparator*** vergleicht ***Integer*** auf ihre natürliche aufsteigende Ordnung. Durch Tausch der Vergleiche erhält man eine absteigende Ordnung.
- Die Anwendung von ***MyComparator*** bei der Sortierung zeigt das Beispiel rechts.
- In Ruby hatten wir die Möglichkeit das Ordnungskriterium im ***sort***-Block zu übergeben.
- In Java übergeben wir eine Implementierung eines Interfaces.

```
Random rand = new Random();
List<Integer> lst = new
    ArrayList<Integer>();
for (int i = 0; i < 20; i++)
lst.add(rand.nextInt(100));
p("before sorting, a = " + lst);
Collections.sort(lst, new
    MyComparator());
p("after sorting, a = " + lst);
```

```
before sorting, a = [7, 78, 0, 65, 39, 36, 24, 88, 70, 62, 5, 45, 5, 69, 27, 1, 41, 54, 70, 76]
after sorting, a = [0, 1, 5, 5, 7, 24, 27, 36, 39, 41, 45, 54, 62, 65, 69, 70, 70, 76, 78, 88]
```

KONVERTIERUNGEN

Konvertierungen mit generischen Kopier-Konstruktoren



- Alle Collection-Klassen verfügen über Konstruktoren, die das Konvertieren beliebiger Collections in Instanzen der eigenen Klassen ermöglichen.
- Die Konstruktoren heißen auch generische Kopier-Konstruktoren.

Konvertierung von Listen in Arrays

- Listen sind den Builtin Arrays sehr ähnlich. Daher ist es möglich Listen in Arrays zu konvertieren.
- Die Methoden dazu lauten
  toArray(T[] tary)
  toArray()
- Bei der ersten Methode erhalten wir Arrays mit dem Komponententyp ***T***, der Ergebnistyp der zweiten Methode ist ein ***Object*** Array. (***Die Hintergründe erfahren wir in der Vorlesung über generische Typen***).

```
public static void listAndArray() {
     List<String> lst = Arrays.asList(new
        String[] { "eins", "zwei", "drei"
        });
     Object[] obj = lst.toArray();
     String[] strings = lst.toArray(new
        String[]{});
     p("lst: " + lst);
     p("obj: " + obj + "\n");
     p("strings: " + strings + "\n");
     for (String s : lst) pnb (s + " ");
     p("");
     for (Object o : obj) pnb(o + " ");
}
```

Konvertierung von Listen in Arrays

```
public static void listAndArray() {
      List<String> lst = Arrays.asList(new String[] { "eins", "zwei", "drei" });
      Object[] obj = lst.toArray();
      String[] strings = lst.toArray(new String[]{});
      p("lst: " + lst);
      p("obj: " + obj + "\n");
      p("strings: " + strings + "\n");
      for (String s : lst) pnb (s + " ");
      p("");
      for (Object o : obj) pnb(o + " ");
}
```

```
lst: [eins, zwei, drei]
obj: [Ljava.lang.String;@7987aeca
strings: [Ljava.lang.String;@3ae48e1b

eins zwei drei
eins zwei drei
```

- Bei der Konvertierung einer Liste in ein Array, bleibt der (dynamische) Komponenten-Typ der generischen Liste erhalten.

Konvertierung von Arrays in Listen

- Die Utility Klasse ***Arrays*** hat eine Methode mit denen Arrays in Listen gewandelt werden können (***asList(T...)***).
- ***asList(T... tAry)*** liefert eine **nicht modifizierbare** Liste mit dem Komponententyp ***T***.
- Nicht modifizierbare Listen sind Listen, deren **Länge nicht verändert** werden darf. Der **Inhalt darf** aber durchaus **verändert** werden. (***lst.set(dummy)***)

```
private static void arraysAsList() {
    String[] strings = new String[] { "eins", "zwei", "drei" };
    List<String> lst = Arrays.asList(strings);
    for (String s : strings)
        pnb(s + " ");
    lst.set(0, "dummy");
    for (String s : strings)
        pnb(s + " ");
    p(" ");
}
```

```
eins zwei drei
dummy zwei drei
```

Konvertierung von Arrays in Listen

```
private static void unmodifiableLists() {
    List<String> lst1 = new ArrayList<String>();
    lst1.add("eins");
    lst1.add("zwei");
    lst1.clear();
    List<String> lst2 = Arrays.asList(new String[] { "eins", "zwei" });
    // lst2.clear(); // UnsupportedOperationException
    // lst2.add("anything"); // UnsupportedOperationException
    // Get modifiable List with explicit constructor
    lst2 = new LinkedList<String>(Arrays.asList(new String[] { "eins", "zwei" }));
    lst2.clear();
    lst2.add("anything");
}
```

- ***Arrays.asList*** erzeugt eine nicht modifizierbare Liste:
  - Die **Länge** der Liste **darf nicht verändert** werden.
  - Da ***clear*** und ***add*** den Inhalt **und** die Länge verändern wird eine ***UnupportedOperationException*** geworfen.
- Soll aus einem Array eine modifizierbare Liste entstehen, dann muss das Ergebnis von ***Arrays.asList(...)*** einer Liste im Konstruktor übergeben werden.

Was steckt technisch hinter den nicht modifizierbaren Listen?

- Die Klasse ***AbstractList<E>*** implementiert das ***List<E>*** Interface als nicht modifizierbare Liste. Alle Methoden, die die Länge einer Liste verändern, erzeugen eine ***UnsupportedOperationException***.

- Die Klasse Arrays erzeugt in der Methode ***asList*** eine private innere Klasse die von ***AbstractList<T>*** ableitet, die u. a. die Methoden ***set*** und ***get*** überschreibt, aber nicht die Methoden, die die Länge einer Liste verändern.

- **Merke**: Wenn eigene Klassen von ***AbstractList*** ableiten, ohne die Methoden zu überschreiben, die die Länge einer Liste modifizieren, dann verhält sich diese Klasse wie eine **nicht modifizierbare Liste**.

MAPS

Das Map<K,V> Interface

- Abbildung von Schlüsseln (keys) auf Werte (values):
  - keine doppelten Schlüssel
- ***HashMap***, eine Hashtabelle mit einer effizienten Implementierung für das Einfügen und den Zugriff auf Elemente.
- ***TreeMap***, eine Map, die die Schlüssel aufsteigend sortiert verwaltet.

```
public interface Map<K,V> {
    // Basis
    V put(K key, V value);
    V get(K key);
    V remove(Object key);
    boolean containsKey(Object key);
    boolean containsValue(Object value);
    int size();
    boolean isEmpty();
    // Massenoperationen
    void putAll(Map<? extends K,? extends V> t);
    void clear();
    // Collection Views
    public Set<K> keySet();
    public Collection<V> values();
    public Set<Map.Entry<K,V>> entrySet();
    // Interface entrySet Elemente
    public interface Entry<K,V> {
        K getKey();
        V getValue();
        V setValue(V value);
    }
}
```

Maps sind keine Collections, bieten aber Collection Sichten auf Schlüssel, Werte und Einträge

- Drei **Collection Sichten** auf eine ***Map***:
  - ***keySet*: die Schlüssel einer *Map* als Menge
  - ***values***: die Werte einer ***Map*** als Liste. Da Werte mehr als einmal auftreten können, kann diese Sicht keine Menge sein.
  - ***entrySet***: die Menge der Schlüssel-Wert Paare einer ***Map*** . ***Map*** enthält ein eingebettetes Interface ***Entry<K,V>***, das den Typ der Paare definiert.
- Die Collection-Sichten sind die **einzige Möglichkeit**, über eine ***Map*** zu **iterieren**.

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

for (Integer key : mis.keySet())
    p(key);

for (Iterator<Integer> i=
    mis.keySet().iterator(); i.hasNext();
    )
    p(i.next());

for (Map.Entry<Integer, String> e :
    mis.entrySet() )
p(e.getKey() + ": " + e.getValue());
```

Füllen einer *Map* und Auslesen

```
public class MapDemo {
     public static void main(String[] args) {
          p("MapDemo");
          String[] names = { "Mueller", "Schaefer", "Schulze", "Meier" };
          String[] hobbies = { "Fussball", "Wandern", "Tauchen", "Segeln" };
          Map<String, String> tb= new HashMap<String, String>();
          // Map<String, String> table = new TreeMap<String, String>();
          for (int i = 0; i < names.length; i++) tb.put(names[i], hobbies[i]);
          for (String key : table.keySet())  p("key->" + key);
          for (String val : table.values()) p("value->" + val);
          for (Map.Entry<String, String> e : table.entrySet())
          // effizienteste Variante
            p(e.getKey() + "->" + e.getValue());
     }
}
```

```
Map Demo
 key->Meier key->Mueller key->Schulze key->Schaefer
 value->Segeln value->Fussball value->Tauchen value->WandernMeier->Segeln
Mueller->Fussball
Schulze->Tauchen
Schaefer->Wandern
```

Mit einer *Map* die Häufigkeit von Zeichen protokollieren

```
public class OneMoreIterableDemo {
     public static void main(String[] args) {
          String str = new String("ein kurzer Teststring mit: aoaoei");
          Map<Character, Integer> freqDict = new TreeMap<Character, Integer>();
          for (int i = 0; i < str.length(); i++) {
             Character c = new Character(str.charAt(i));
             Integer freq = freqDict.get(c);
             if (freq == null) freqDict.put(c, 1);
             else freqDict.put(c, freq + 1);
          }
          for (Map.Entry<Character, Integer> e : freqDict.entrySet())
                    pnb(e.getKey() + ": " + e.getValue() + " ");
     }
}
```

```
Häufigkeit von Zeichen
 : 4 :: 1 T: 1 a: 2 e: 4 g: 1 i: 4 k: 1 m: 1 n: 2 o: 2 r: 3 s: 2 t: 3 u: 1 z: 1
```

- In ***freqDict*** wird die Häufigkeit des Auftretens eines Zeichens in ***str*** protokolliert.
- Direktes Iterieren über einen ***String*** ist nicht möglich. ***String*** implementiert ***Iterable*** nicht.

Übungen

- Schreiben Sie eine statische Methode, die eine Map invertiert.
- Schreiben Sie eine statische Methode, die zwei Maps konkateniert.