Collections API

Java Foundation Track by Carsten Bokeloh

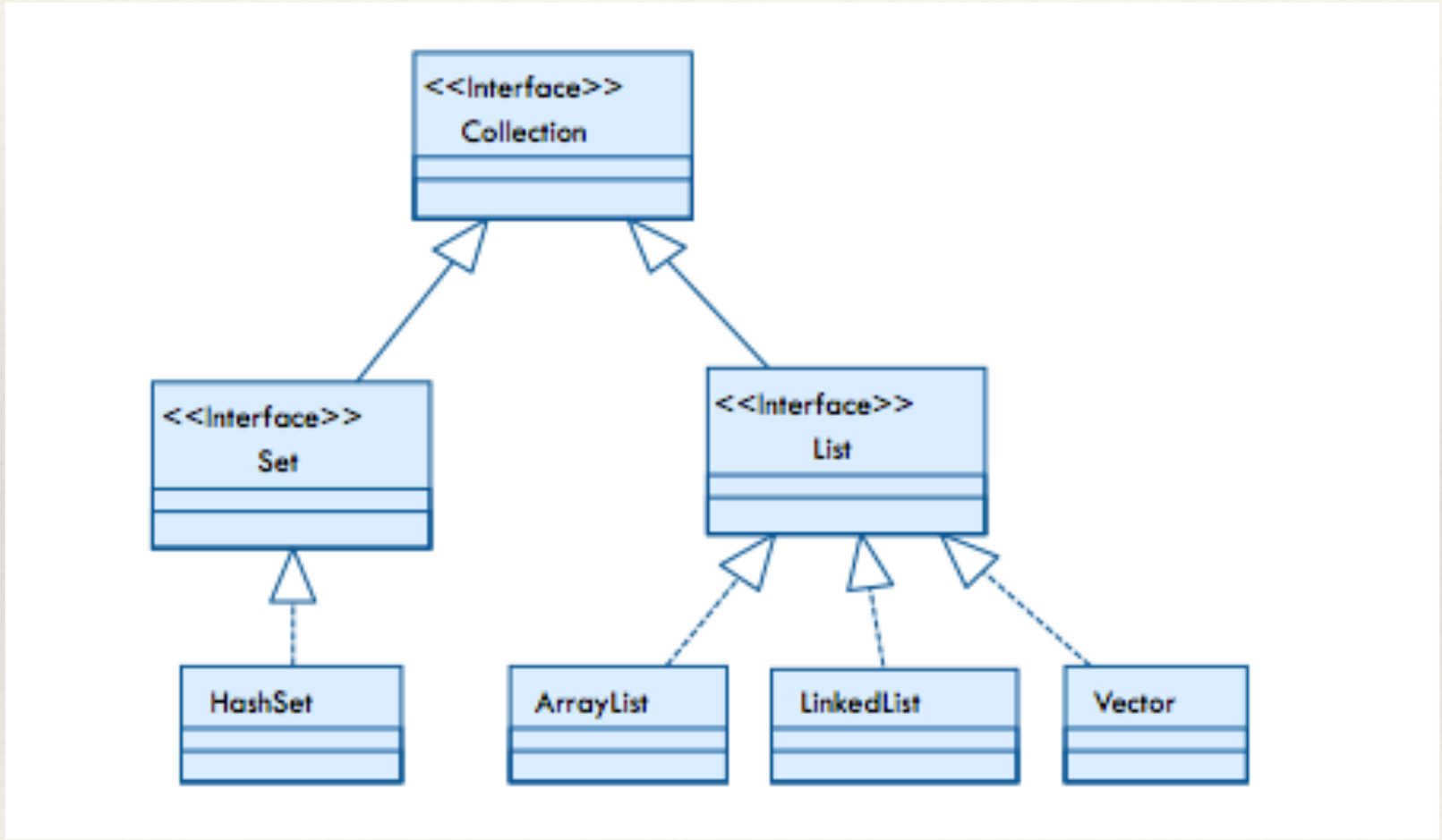
Collections

- diverse Collections
- Iteratoren
- Suchen & Sortieren

Collections

- Eine Collection ist ein Objekt zur Verwaltung einer Menge verschiedenster Objekte
- Objekte bedeutet hier:
  - Instanzen von java.lang.Object
- **Es gibt keine Klasse, die Collection direkt implementiert!!!**

Klassenhierarchie



Die Schnittstelle Collection

- **boolean add(Object element)**
  - Falls erlaubt, wird *element* hinzugefügt und *true* geliefert
- **boolean remove(Object element)**
  - entfernt *element*, falls vorhanden
- **void clear()**
  - löscht alle Elemente auf einmal
- **int size()**
  - liefert die Anzahl der Elemente in der Collection
- **boolean contains(Object element)**
  - liefert true, falls element in der Collection enthalten ist.
- **Object[] toArray**
  - liefert ein Array, dass alle Elemente der Collection enthält
- **Iterator iterator()**
  - liefert einen Iterator zu der Collection

Die List-Schnittstelle

- Das Interface List erbt die Methoden von Collection und fügt neue hinzu. List fügt einer Collection vor allem hinzu, dass die Elemente in einer Reihenfolge abgelegt werden. Darum ist es möglich über eine Positionsnummer auf die Elemente zuzugreifen und nicht nur per Iterator.

Eine Liste ist…

- sortiert. Jedes Objekt in einer Liste kann über den Index angesprochen werden. Hierdurch auch sichergestellt, dass die Liste zur Laufzeit nicht einfach die Position ändern kann.
- Doppelte Einträge sind erlaubt.
- index fängt bei 0 an zu zählen.
- ListIterator zusätzlich
  - erlaubt es Objekte hinzuzufügen bzw. zu entfernen und stellt eine Funktionalität bereit, um bidirektional darauf zuzugreifen.

Interface java.util.List

- **Object get(int index)**
  - liefert Element an der Position index.
- **Object set(int index, Object element)**
  - Element an der Position index wird ersetzt
- **void add(int index, Object element)**
  - fügt *element* an Position *index* in die Collection ein
- **Object remove(int index)**
  - entfernt und liefert das Objekt an Position *index*
- **Object get(ind index)**
  - liefert das Objekt an Position *index*
- **ListIterator listIterator()**
  - Die Liste liefert durch den Aufruf listIterator einen speziell auf die Liste zugeschnittenen Iterator(RückwärtsLauf möglich).

Implementierungen von List

Typ	Implementierung	Erklärung
Listen (List)	ArrayList	Implementiert Listen-Funktionalität durch die Abbildung auf ein Feld; implementiert die Schnittstelle List.
LinkedList	LinkedList ist eine doppelt verkettete Liste, also eine Liste von Einträgen mit einer Referenz auf den jeweiligen Nachfolger und Vorgänger. Das ist nützlich beim Einfügen und Löschen von Elementen an beliebigen Stellen innerhalb der Liste.	

Konstruktoren ArrayList

```
//Anlage einer leeren Liste mit Anfangskapazität von 10 Elementen.
List list1 = new ArrayList();
list1.add(2);
System.out.println(list1.size());

//Eine Liste mit initialCapacity freien Elementen wird angelegt
List list2 = new ArrayList(12);
list2.add(2);
System.out.println(list2.size());

Collection c = new ArrayList();
c.add(3);
c.add(4);
//Kopiert alle Elemente der Collection c in das neue ArrayList-Objekt
List list3 = new ArrayList(c);
System.out.println(list3.size());
```

ArrayList

```
/**
 *   •Die Klasse ArrayList verwaltetet zwei Größen:
 *   -Zum einen die Anzahl der gespeicherten Elemente
 *   -Zum anderen die interne Größe des Felds
 *   • Ist die Kapazität des Felds größer als die Anzahl der Elemente,
 *   können noch Elemente aufgenommen werden, ohne dass die Liste
 *   etwas unternehmen muss
 *   • Die Anzahl der Elemente in der Liste liefert die Methode size()
 *   • Die Liste vergrößert sich automatisch,
 *   falls mehr Elemente aufgenommen werden,
 *   als ursprünglich am Platz vorgesehen waren
 *   • Diese Operation heißt Resizing
 *   • Dabei spielt die Größe initialCapacity für effizientes Arbeiten
 *   eine wichtige Rolle. Sie sollte passend gewählt sein
 */
```

ArrayList

- Wenn das Array zehn Elemente fasst, nun ein elftes eingefügt werden soll, muss das Laufzeitsystem einen neuen Speicherbereich reservieren und jedes Element des alten Felds in das neue kopieren
- Das kostet Zeit
- Die ArrayList verdoppelt nicht die Größe, sie nimmt die neue Größe mal 1,5

LinkedList

```java
//Eine neue leere Liste.
LinkedList list1 = new LinkedList();
list1.add(2);
System.out.println(list1.size());

LinkedList list2 = new LinkedList();
list2.add(1);
list2.add(2);

//Zusätzliche Methoden
list2.addFirst(0);
System.out.println(list2.size());
list2.addLast(3);
System.out.println(list2.size());
list2.removeFirst();
System.out.println(list2.size());
list2.removeLast();
System.out.println(list2.size());

Collection c = new ArrayList();
c.add(3);
c.add(4);
//Kopiert alle Elemente der Collection c in die neue verkettete Liste
LinkedList list3 = new LinkedList(c);
System.out.println(list3.size());
```

LinkedList versus ArrayList

- ArrayList speichert Elemente in einem Array (wie es auch Vector macht)
- LinkedList speichert die Elemente in einer verketteten Liste
- Die Verkettung wird mit einem Hilfsobjekt der Klasse Entry für jedes Listen-Element erreicht
- Je nach Einsatzgebiet, muss ausgewählt werden, ob die LinkedList oder die ArrayList verwendet werden sollte

LinkedList versus ArrayList

- Da ArrayList intern ein Array benutzt, ist der Zugriff auf ein spezielles Element über die Position in der Liste sehr schnell
- Eine LinkedList muss aufwändiger durchsucht werden, und dies kostet Zeit
- Die verkettete Liste ist deutlich im Vorteil, wenn Elemente mitten in der Liste gelöscht oder eingefügt werden
- Es muss nur die Verkettung der Hilfsobjekte an einer Stelle verändert werden

LinkedList versus ArrayList

- Bei einem ArrayList-Objekt als Container bedeutet dies viel Arbeit, es sei denn, das Element muss am Ende gelöscht oder eingefügt werden
  - Zum einen müssen alle nachfolgenden Listen-Elemente beim Einfügen und Löschen im Array verschoben werden
  - Zum anderen kann beim Einfügen das verwendete Array-Objekt zu klein werden
  - Dann bleibt, wie beim Vector, nichts anderes übrig, als ein neues Array-Objekt anzulegen und alle Elemente zu kopieren

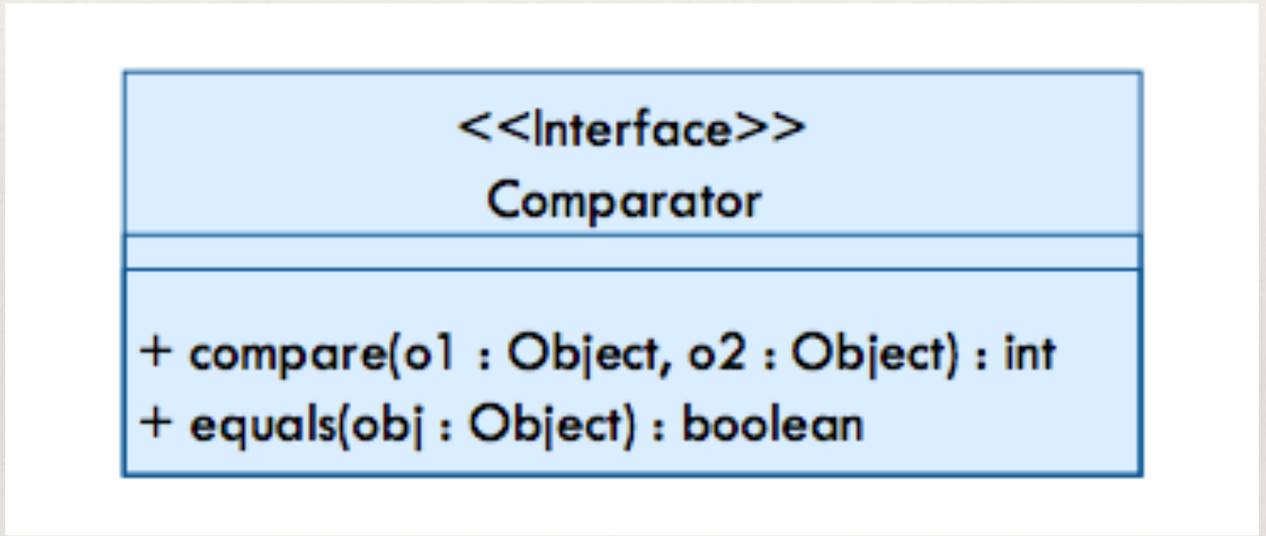
ListIterator

- ListIterator ist eine Erweiterung von Iterator
- Diese Schnittstelle fügt noch Methoden hinzu, damit an der aktuellen Stelle auch Elemente eingefügt werden können
- Mit einem ListIterator lässt sich rückwärts laufen und auf das vorhergehende Element zugreifen

Klasse: java.util.Collections

- Sortier-Methoden
  - static void sort(List list)
    sortiert die Liste in aufsteigender Reihenfolge oder nach der natürlichen Ordnung der Elemente, alle Objekte der Liste müssen Comparable-Interface implementieren
  - static void sort(List list, Comparator c)

Interface: java.util.Comparator



Comparable

```
List empColl = new ArrayList();
empColl.add(new Employee(2, 3000, "Müller", "Peter"));
empColl.add(new Employee(4, 5000, "Schulz", "Rolf"));
empColl.add(new Employee(3, 2000, "Ahrens", "Dieter"));
empColl.add(new Employee(1, 6000, "Schmidt", "Franz"));
System.out.println("Unsortiert");
for (Object object : empColl) {
    System.out.println((Employee)object);
}

//RuntimeException
System.out.println("Sortiert");
Collections.sort(empColl);

//warum Employee implementiert nicht Comparable
//nach Implementierung von Comparable wird sortiert.
for (Object object : empColl) {
    System.out.println((Employee)object);
}
```

Comparable

```
public Employee(int id, int gehalt, String nachname, String vorname) {
    this.id = id;
    this.gehalt = gehalt;
    this.nachname = nachname;
    this.vorname = vorname;
}

private int id;
private int gehalt;
private String nachname;
private String vorname;
public int getGehalt() {
    return gehalt;
}
public void setGehalt(int gehalt) {
public String getNachname() {
public void setNachname(String nachname) {
public String getVorname() {
public void setVorname(String vorname) {
public int getId() {
public void setId(int id) {
@Override
public String toString() {
    return "Employee [gehalt=" + gehalt + ", nachname=" + nachname
            + ", vorname=" + vorname + "]";
}
@Override
public int compareTo(Object o) {
    return this.getId() - ((Employee)o).getId();
}
```

Comparator

```
public class NameComparator implements Comparator{

    @Override
    public int compare(Object o1, Object o2) {
        return ((Employee)o1).getNachname().compareTo(((Employee)o2).getNachname());
    }

}
```

Comparator

```
//Nachteil: nur eine Vergleichsmöglichkeit pro Objekt
//Lösung: Comparator

//Sortierung nach Name
System.out.println("Sortiert nach Name ");
Collections.sort(empColl, new NameComparator());

for (Object object : empColl) {
    System.out.println((Employee)object);
}
```

Übungen zu List

- UebList1
- UebList2

Die Schnittstelle Set

- Ein Set ist eine (im Prinzip) ungeordnete Sammlung von Elementen. Jedes Element darf nur einmal vorkommen
- Für Mengen sieht die Java-Bibliothek die Schnittstelle java.util.Set vor

Die Schnittstelle Set

- Implementierende Klassen sind unter anderem:
  - HashSet: Schnelle Mengenimplementierung durch Hashing-Verfahren (Dahinter steckt eine HashMap)
  - TreeSet: die eine Sortierung ermöglichen
  - LinkedHashSet: Schnelle Mengenimplementierung unter Beibehaltung der Einfügereihenfolge

Die Schnittstelle Set

```
//Erzeugt ein neues HashSet-Objekt mit 16 freien Plätzen und einem Füllfaktor von 0,75
HashSet set1 = new HashSet();
set1.add("Berlin");
System.out.println(set1.size());

//Erzeugt ein neues Set aus der Menge gegebener Elemente
Collection c = new ArrayList();
c.add(3);
c.add(4);
HashSet set2 = new HashSet(c);
System.out.println(set2.size());

//Erzeugt ein neues HashSet mit einer gegebenen Anzahl freier Plätze
//und dem Füllfaktor von 0,75
HashSet set3 = new HashSet(3);
set3.add("Berlin");
System.out.println(set3.size());

//Erzeugt ein neues leeres HashSet mit einer Startkapazität
//und einem gegebenen Füllfaktor
//Die Startgröße ist für die Performance wichtig
//Ist die Größe zu klein gewählt, muss die Datenstruktur bei neu
//hinzugefügten Elementen vergrößert werden
HashSet set4 = new HashSet(3, 0.5f);
set3.add("Berlin");
System.out.println(set3.size());
```

TreeSet

- Die Klasse java.util.TreeSet verfolgt eine andere Implementierungsstrategie als ein HashSet
- Ein TreeSet verwaltet die Elemente immer sortiert
- Speichert TreeSet ein neues Element, so fügt TreeSet das Element automatisch sortiert in die Datenstruktur ein
- Das kostet etwas mehr Zeit als ein HashSet, dafür ist diese Sortierung dauerhaft

TreeSet

- Daher ist es auch nicht zeitaufwändig, alle Elemente geordnet auszugeben
- Die Suche nach einem einzigen Element ist aber etwas langsamer als im HashSet
- Der Begriff langsamer muss jedoch relativiert werden, da die Suche logarithmisch ist und daher nicht wirklich langsam
- Beim Einfügen und Löschen muss bei bestimmten Konstellationen eine Reorganisation des Baumes in Kauf genommen werden, was die Einfüge-/Löschzeit verschlechtert

TreeSet

```
//Erzeugt ein neues leeres TreeSet
TreeSet ts1 = new TreeSet();
ts1.add("Berlin");
ts1.add("Aachen");
ts1.add("Zürich");
ts1.add("Frankfurt");
System.out.println("TS1:"+ts1);
ts1.add("Köln");
System.out.println("TS1:"+ts1);

//Erzeugt ein neues TreeSet aus der gegebenen Collection
Collection c = new ArrayList();
c.add("Berlin");
c.add("Aachen");
TreeSet ts2 = new TreeSet(c);
System.out.println("TS2:"+ts2);

TreeSet ts3 = new TreeSet(new NameComparator());
ts3.add(new Employee(100, "Baier", "Gustav"));
ts3.add(new Employee(200, "Ahrens", "Helmut"));
System.out.println(ts3);

TreeSet ts4 = new TreeSet(new TreeSet(new GehaltsComparator()));
ts4.add(new Employee(500, "Baier", "Gustav"));
ts4.add(new Employee(900, "Meier", "Günther"));
ts4.add(new Employee(200, "Ahrens", "Helmut"));
System.out.println(ts4);
```

TreeSet

```
System.out.println("First of TS1:"+ts1.first());
System.out.println("Last of TS1:"+ts1.last());
System.out.println("Comparator of TS3:"+ts3.comparator());
System.out.println("Head of TS1:"+ts1.headSet("Köln", true));
System.out.println("Tail of TS1:"+ts1.tailSet("Köln", true));
System.out.println("Subset of TS1:"+ts1.subSet("Berlin", "Köln"));
```

LinkedHashSet

- Ein LinkedHashSet vereint die Reihenfolgentreue einer Liste und die hohe Performance für Mengenoperationen vom HashSet
- Dabei bietet die Klasse keine Listen-Methoden wie first() oder get(index), sondern ist eine Implementierung ausschließlich der SetSchnittstelle, in der der Iterator die Elemente in der Einfüge-Reihenfolge liefert

Übung

- UebSet1.txt

Interface java.util.Map

- Ein Map-Objekt ist eine Menge von Schüssel-Werte-Paaren(Assoziativspeicher)
- Vergleichbar mit einer Tabelle, die aus zwei Spalten besteht
- Sowohl die Schüssel als auch die Werte sind Instanzen von java.lang.Object
- Ein Objekt vom Typ Map ist eine heterogene Collection, die auf der Schlüsselseite keine Duplikate erlaubt

Map

- Java implementiert assoziative Speicher auf zwei unterschiedliche Arten:
  - Eine übliche und sehr schnelle Implementierung ist die Hash-Tabelle (engl.hashtable), die in Java durch java.util.HashMap implementiert ist
  - Daneben existiert die Klasse java.util.TreeMap, die etwas langsamer im Zugriff ist, doch dafür alle Schlüssel in einem Baum sortiert abspeichert

Map

- Die Klasse HashMap eignet sich ideal dazu,viele Elemente unsortiert zu speichern und sie über die Schlüssel schnell wieder verfügbar zu machen

Map

```
HashMap map1 = new HashMap();
map1.put("Berlin", "Deutschlands Hauptstadt");
map1.put("Frankfurt", "Hessens schönste Stadt");
map1.put("Köln", "schönste Stadt Deutschlands");
map1.put("Köln", "Karnevalsstadt Deutschlands");
map1.put("Düsseldorf", null);
System.out.println(map1);
System.out.println(map1.get("Frankfurt"));
System.out.println(map1.containsKey("Frankfurt"));
//containsValue ist wesentliuch langsamer,
//da alle Werte der Reihe nach durchsucht werden müssen
System.out.println(map1.containsValue("schönste Stadt Deutschlands"));
System.out.println(map1.isEmpty());
```

equals/hashcode

- Wenn Objekte in eine Map eingefügt werden, stellt sich die Frage, wonach die Identität zweier Keys überprüft wird
- Zunächst wird der Hashwert des Schlüssel-Objektes ermittelt, ist dieser gleich wird zusätzlich auf equals() überprüft
- Erst wenn equals() true liefert ist klar, dass der Key bereits enthalten ist
- Dies macht deutlich, dass die Implementierung von equals() und hashCode() bei eigenen Klassen wichtig ist
- Vor allem müssen die Methoden korrespondierend implementiert werden:
  - Liefert equals() für zwei Objekte true, muss auch der HashCode gleich sein

equals/hashcode

- Die verwendeten Keys sollten immutable sein
- Sind sie dies nicht, kann das Objekt verändert werden und der HashCode ändert sich
- Dadurch kann das Element nicht mehr gefunden werden und liegt als Leiche in der Map
- **Achtung**: Bei einer TreeMap wird (analog zum TreeSet) mittels compareTo() die Gleichheit überprüft

CollectionView einer Map

```
Set<Entry> entries = map1.entrySet();
for(Entry entry : entries) {
    System.out.println(entry);
}
System.out.println("--------");
```

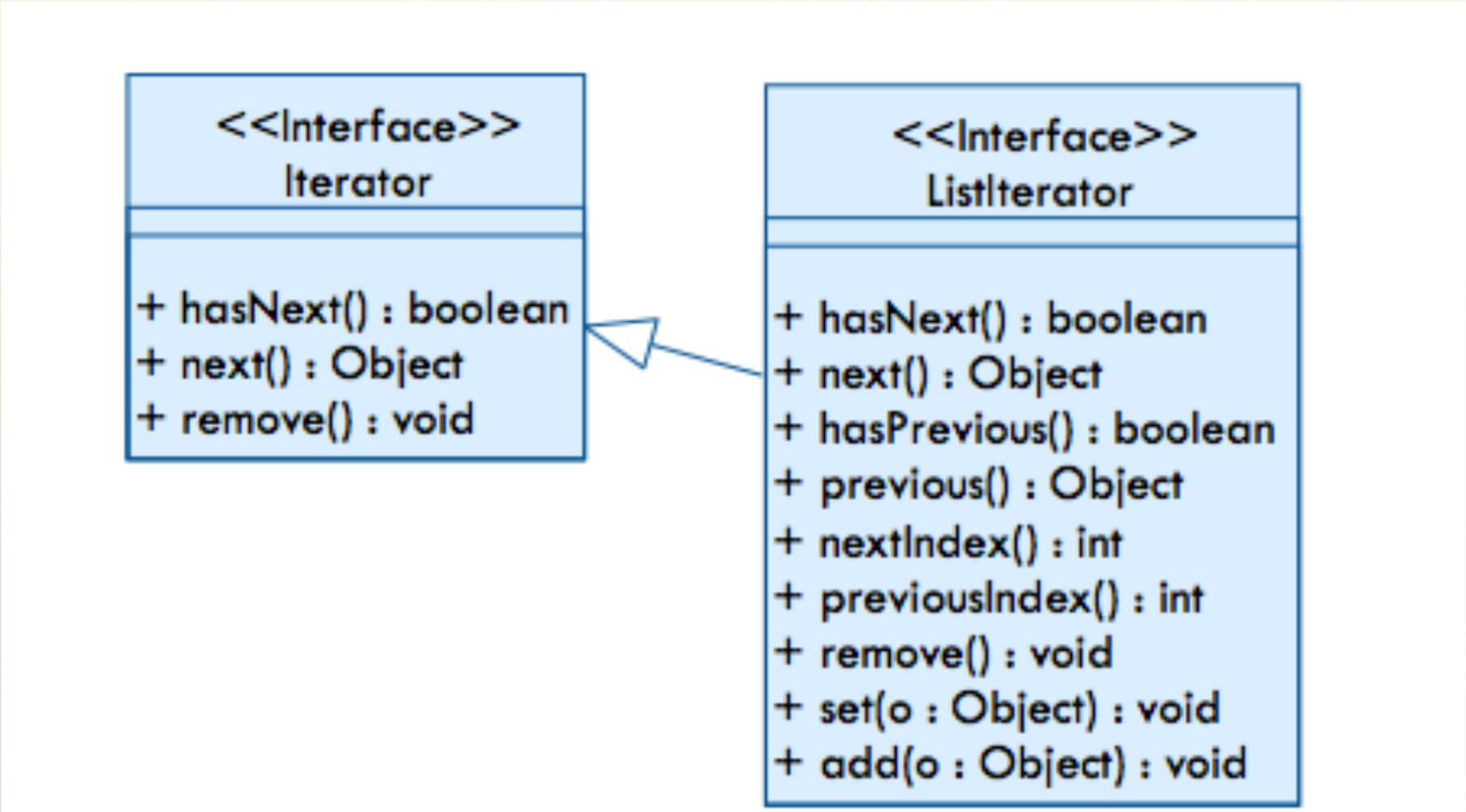
```
Set keys = map1.keySet();
for(Object obj : keys) {
    System.out.println((String)obj);
}

System.out.println("--------");
Collection c = map1.values();
for(Object ob : c) {
    System.out.println((String)ob);
}
```

Iteratoren

- *Iteratoren* stellen den Zugriff auf die Elemente einer Collection zur Verfügung
- *Iteratoren* werden von den Collections zur Verfügung gestellt
- Auswahl der Elemente hängt bei den *Iteratoren* von der Collections-Klasse ab
- Sind vergleichbar mit einem Cursor auf eine Datenbank Tabelle

Klassenhierarchie der Iteratoren



Interface java.util.Iterator

- **boolean hasNext()**
  - liefert *true,* wenn die Collection weitere Elemente enthält
- **Object next()**
  - liefert das nächste Element der Iteration
- **void remove()**
  - entfernt das zuletzt gelieferte Objekt aus der Collection, auf der der Iterator operiert.

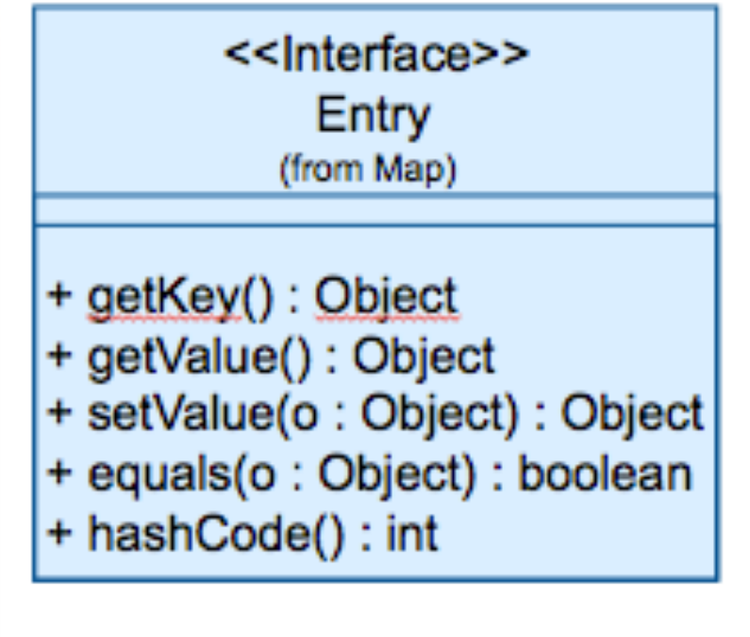
Beispiel:Iterator

```
Set set = new HashSet();
set.add("erster Eintrag");
set.add(new Integer(2));
set.add(new Float(3.0F));
set.add(new Integer(2));   //Duplikat !


Iterator iterator = set.iterator();
while ( iterator.hasNext() )  {
     System.out.println( iterator.next() );
}
```

```
3.0
2
erster Eintrag
```

Interface: Map.Entry



Nützliches aus Collections

- Klasse java.util.Collections
- Enthält nur statische Methoden, die auf Collections operieren oder sie zurückliefern
- Enthält neben Such- und Sortiermethoden auch Methoden zum Erzeugen von threadsicheren Collections
  - **static** Map synchronizedMap (Map m)
  - **static** Set synchronizedSet (Set s)

Übung

- UebMap1.txt
- UebMap2.txt
- UebAutovermietung.txt