# Collections API


*Java Foundation Track by Carsten Bokeloh*

# Collections

- diverse Collections
- Iteratoren
- Suchen & Sortieren

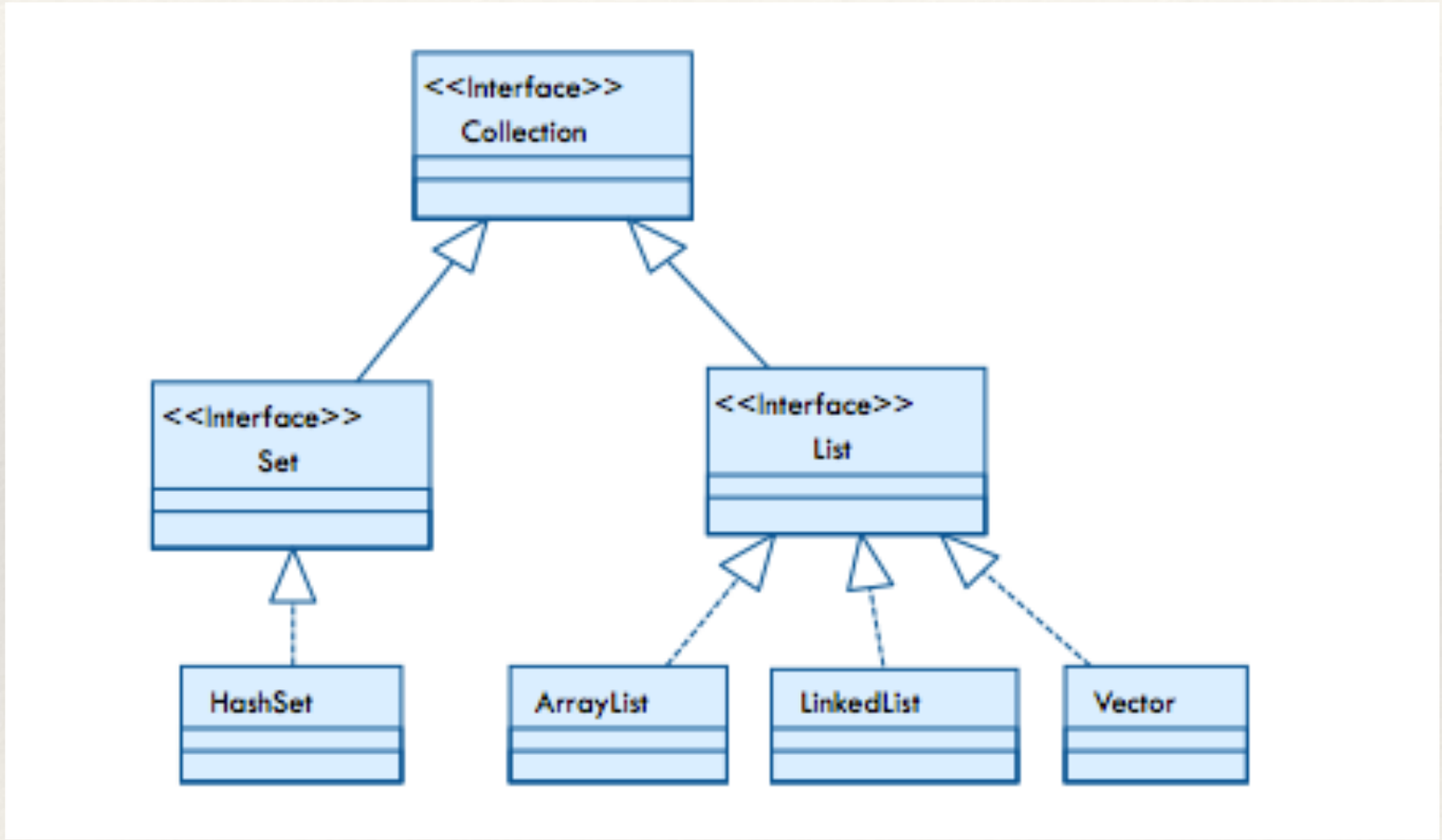
# Collections

- Eine Collection ist ein Objekt zur Verwaltung einer Menge verschiedenster Objekte
- Objekte bedeutet hier:
  - Instanzen von java.lang.Object

# Collections API

- Package java.util.*
- Basis-Interfaces der Collections API
  - Collection
    - ungeordnet, Duplikate erlaubt
  - Set
    - ungeordnet, keine Duplikate erlaubt
  - List
    - geordnet, Duplikate erlaubt

# Klassenhierarchie



# Interface java.util.Collection

- boolean add(Object element)
  - Falls erlaubt, wird *element* hinzugefügt und *true* geliefert
- boolean remove(Object element)
  - entfernt *element*, falls vorhanden
- int size()
  - liefert die Anzahl der Elemente in der Collection
- boolean contains(Object element)
  - liefert true, falls element in der Collection enthalten ist.
- Iterator iterator()
  - liefert einen Iterator zu der Collection

# Interface java.util.List

- 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()
  - liefert einen ListIterator

# Interface java.util.Set

- Enthält die gleichen Methoden wie das Collection-Interface
- Unterschiede machen sicht erst bei den implementierenden Klassen bemerkbar

# Beispiel List

```
List list = new ArrayList();
list.add("erster Eintrag");
list.add(new Integer(2));
list.add(new Float(3.0F));
list.add(new Integer(2)); //Duplikat !
System.out.println(list);
```

```
[erster Eintrag, 2, 3.0, 2]
```

# Beispiel:Set

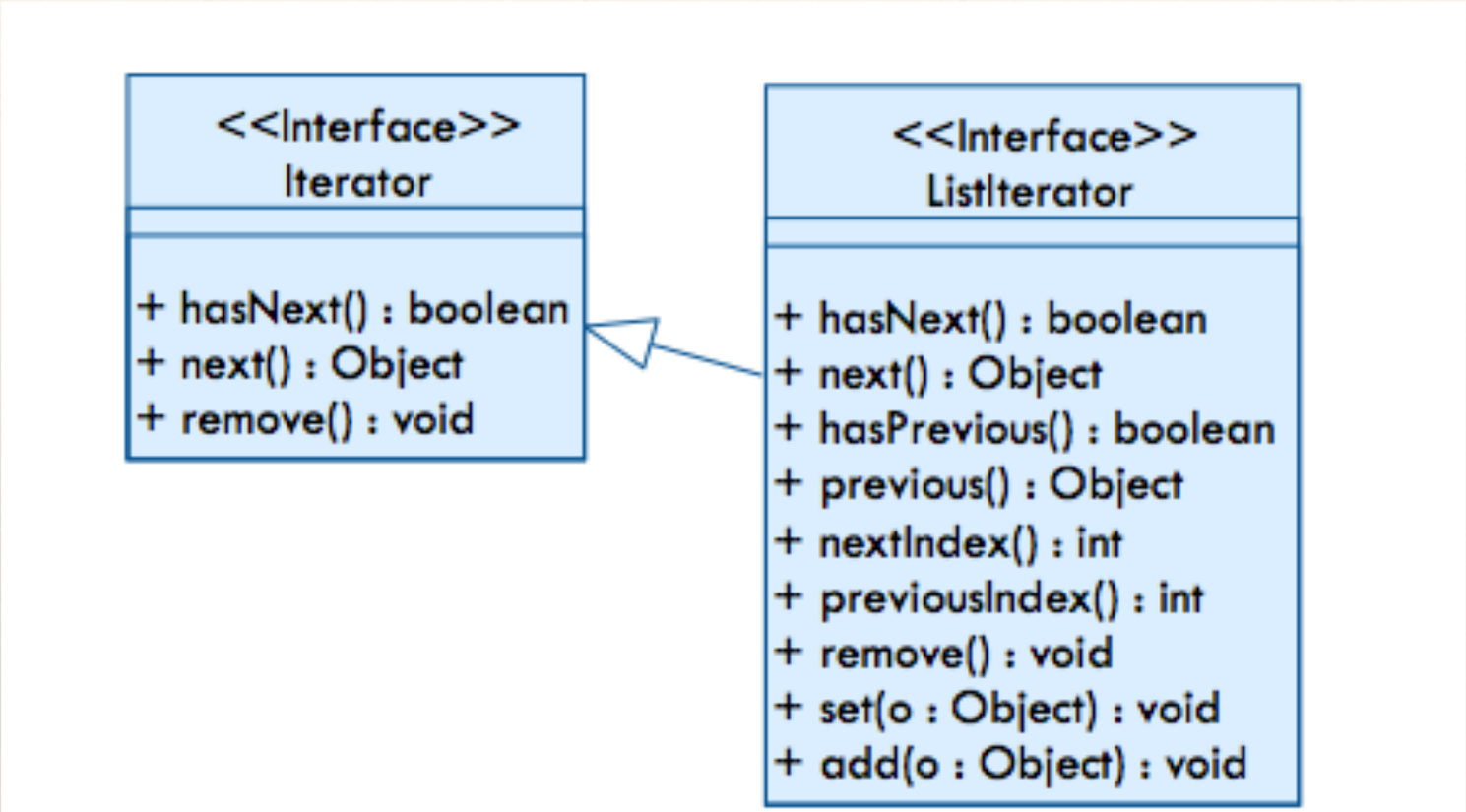
```
Set set = new HashSet();
set.add( "erster Eintrag" );
set.add(new Integer(2));
set.add(new Float(3.0F));
set.add(new Integer(2));  //Duplikat !
System.out.println(set);
```

```
[3.0, 2, erster Eintrag]
```

# 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.

# Interface java.util.ListIterator

- Erbt von java.util.Iterator
- besitzt zusätzliche Methoden
  - **boolean** hasPrevious()
  - Object previous()
  - **void** add(Object element)
  - **void** set(Object element)

# 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
```

# Maps

- Ein Map-Objekt ist eine Menge von Schüssel-Werte-Paaren
- 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 Schüsselseite keine Duplikate erlaubt

# Maps

- Maps erlauben drei Sichten auf ihre Elemente
  - als eine Menge von Schüsseln
  - als eine Menge von Werten
  - als eine Menge von Schüssel-Werte-Paaren

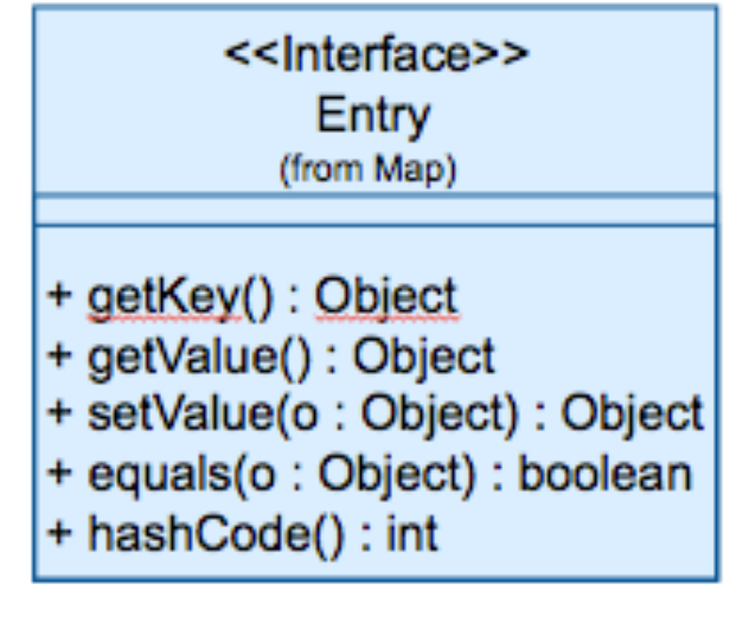
# java.util.Map

- **Object put(Object key,Object value)**
  - fügt ein Paar ein
- **Object get(Object key)**
  - liefert das Wert-Object mit dem angegebenen Schüssel
  - *null,* wenn kein Eintrag zu dem Schüssel existiert
- **Object remove(Object key)**
- **void clear()**
- **int size()**

# Iteration über Maps

- Schüssel-Sicht
  - Methode *keySet*() liefert ein *Set*-Objekt, über das iteriert wird.
- Werte-Sicht
  - Methode *values()* liefert ein *Collection*-Objekt, über das iteriert wird.
- Schüssel-Wert-Sicht
  - Methode *entrySet()* liefert ein *Set-Objekt*, über das iteriert wird, wobei jedes Element wiederum ein *Map.Entry-Objekt* ist.

# Interface: Map.Entry



# Beispiel:entrySet

```
Iterator iter = System.getProperties().entrySet().iterator();
while (iter.hasNext()) {
     Map.Entry entry = (Map.Entry) iter.next();
     System.out.println("K= " + entry.getKey());
     System.out.println("V= " + entry.getValue());
}
```

# Such- und Sortieralgorithmen

- 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)

# Klasse: java.util.Collections

- Such-Methoden
  - static int binarySearch(List list, Object key);
  - static int binarySearch(List list,Object key,Comperator c)

- 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)

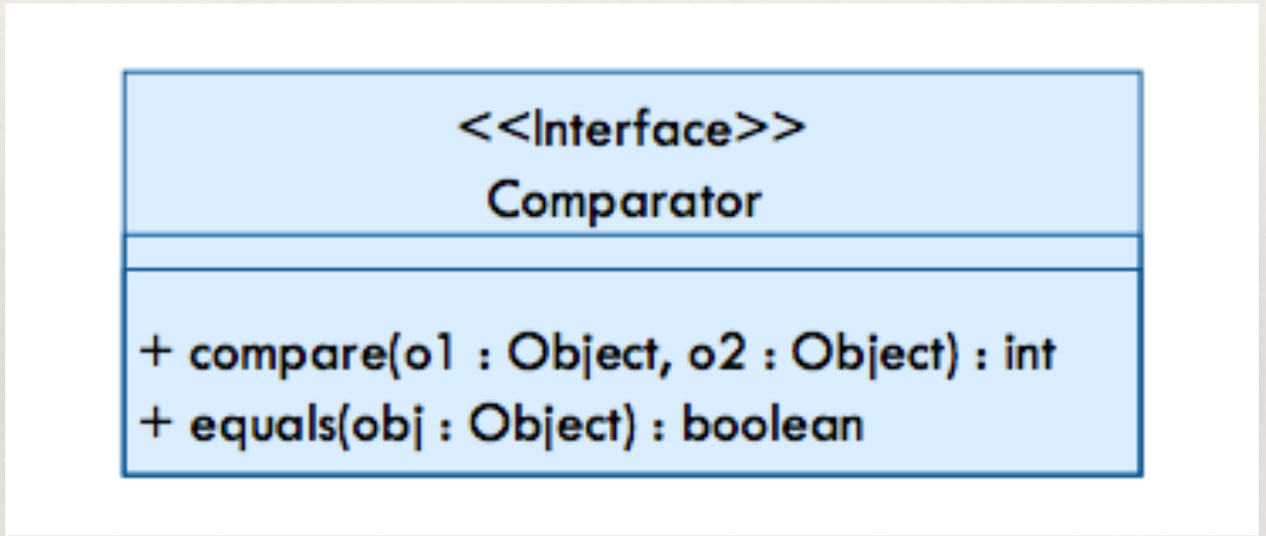
# Benutzerdefiniertes Sortieren

- Auf den Elementen muss eine totale Ordnung definiert werden

- Implementieren des *Comparator-Interfaces*

- Übergabe eines *Comparator-Objekts* an die entsprechende Variante der Methode *Collections.sort(…)*

# Interface: java.util.Comparator



# Interface: java.util.Comparator

- int compare (Object o1, Object o2)
  - vergleicht zwei Elemente
    - x > y impliziert compare(x,y) > 0
    - x = y impliziert compare(x,y) = 0
    - x < y impliziert compare(x,y) < 0

- boolean equals(Object obj)
  - gibt an, ob ein anderes *Comparator-Objekt* mit diesem identisch ist.

# Übung

Schreiben Sie eine Klasse Employee mit den Attributen name(String), vorname(String) und gehalt(int).
Implementieren Sie die jeweiligen getter und setter für die Attribute.
Erzeugen Sie einen Konstruktor, der die o.g. Attribute mit Inhalt füllt.
Spendieren Sie der Klasse ebenfalls eine geeignete toString-metshode.
Daraufhin schreiben Sie eine Implementierung des Interface java.util.Comparator mit dem Namen Gehaltscomparator. Es ist ausreichend die Methode
*public int compare(Object o1, Object o2)*
zu implementieren. Erzeugen Sie eine weitere Klasse, die diverse Employee Objekte initialisiert, sie in eine ArrayList packt, um Sie daraufhin zu sortieren mit der statischen sort Methode aus der Collections-Klasse.
Geben Sie die sortierte Liste auf der Konsole aus.
Wahlweise implementieren Sie einen weiteren Comparator, der Employees nach dem Nachnamen sortiert.