

Programmierungsmethodik 1

Programmierertechnik

Collections

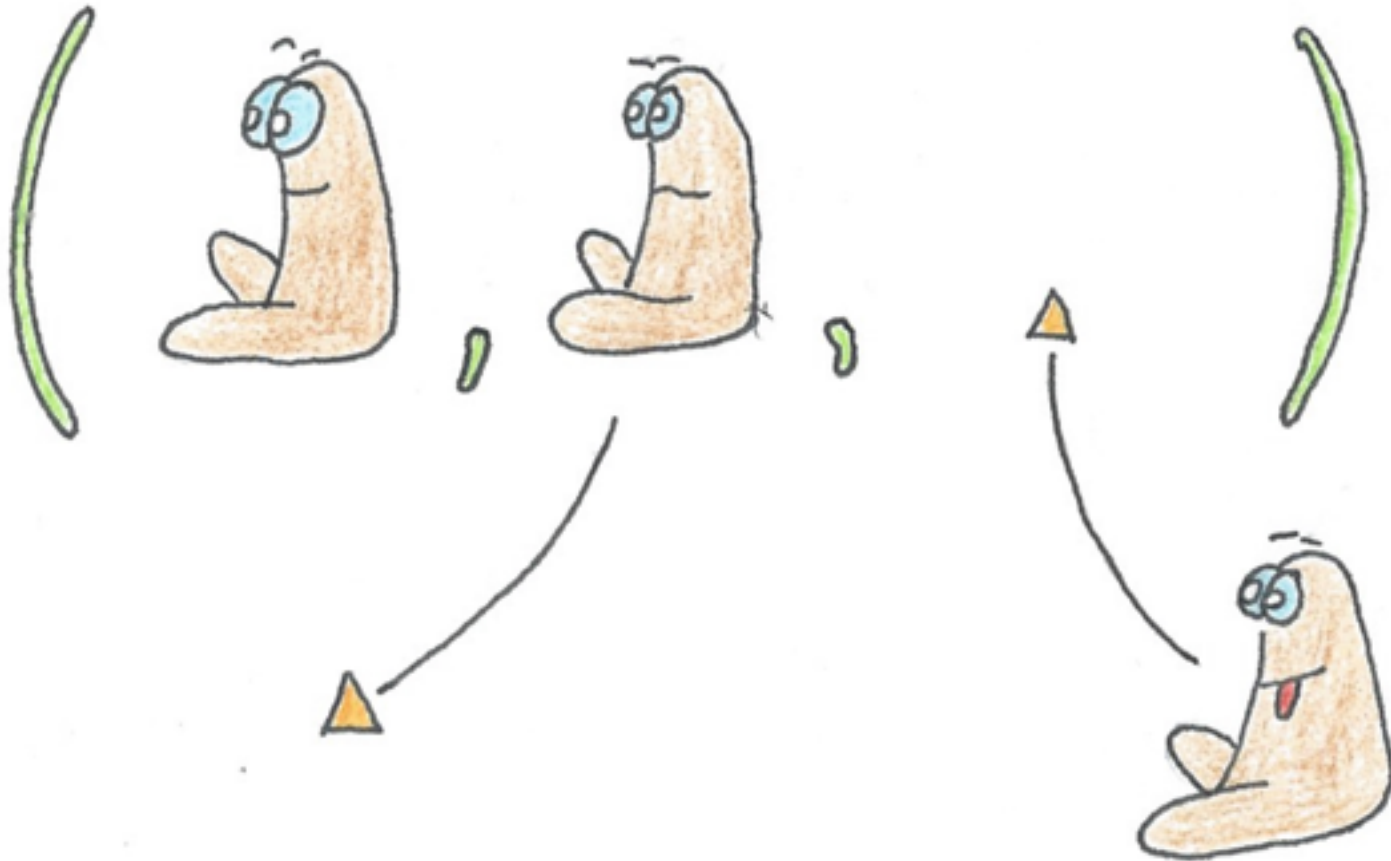
Wiederholung

- Einführung
- Exception-Typen
- catch und finally
- Exceptions werfen
- Logging

Ausblick



Worum gehts?



Agenda

- Collections-Framework
- Verkettete Liste und Array-Liste
- Iteratoren
- Vergleichen



Einführung

Motivation

- häufige Anforderung: mehrere Dinge verwalten
- bisher kennengelernt: Arrays
- Nachteile von Arrays
 - feste Länge
 - Einfügen "in der Mitte"
 - kein echter Referenztyp
 - sehr starr, nicht für alle Anwendungsfälle geeignet
- Lösung: Collection-Framework
 - Datenstrukturen
 - Operationen

Collection-Framework

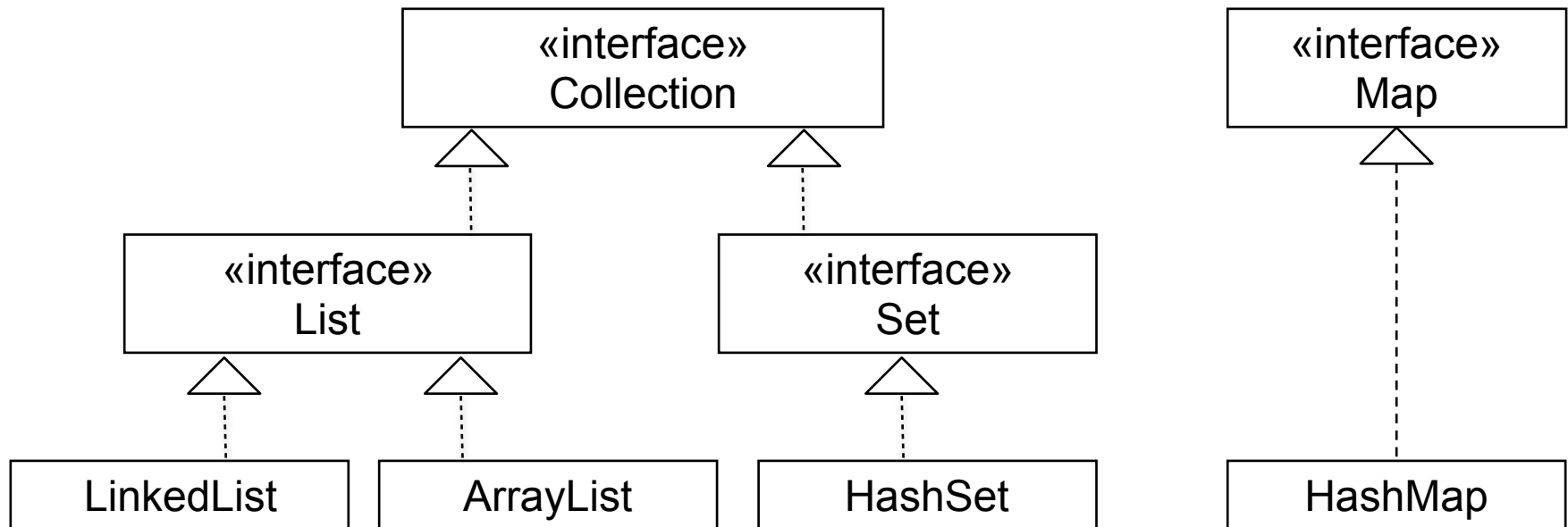
- Die Klassen des Collection-Frameworks ("Collection-Klassen") im Package `java.util` ...
 - sind Containerobjekte (wie Arrays)
 - haben eine veränderliche Elementanzahl
 - keine fest vorgegebene Länge
 - können mit einer `foreach`-Schleife durchlaufen werden (wie Arrays)
 - können Objekte aller Referenztypen als Elemente speichern

Generische Typen

- werden ausführlich in Programmiermethodik 2 behandelt
- hier zum Verständnis notwendig
- Beispiel: `List<T>`
 - Deklaration einer Liste von Objekten vom Typ T
 - Typ T? Kenne ich nicht
 - Gibt es auch nicht – tatsächlich verwendeten Typ einsetzen
 - etwa: `List<String>` = Liste von Strings

Collection-Framework

- Listen (List) ordnen die Elemente an
 - erstes .. letztes Element



Mengen (Set) speichern
die Elemente ungeordnet
und ohne Duplikate

Abbildungen
(Map) speichern
Element-Paare
(Schlüssel, Wert)

Wichtige Methoden des Interfaces Collection

`boolean add(Elementtyp element)`

- fügt das Element zur Collection hinzu
- liefert true, wenn das Element eingefügt wurde
- nur bei Set kann false vorkommen

`boolean remove(Elementtyp element)`

- löscht das Element aus der Collection
- liefert true, wenn das Element gefunden und gelöscht wurde

`boolean contains(Elementtyp element)`

- liefert true, wenn das Element in der Collection enthalten ist

Wichtige Methoden des Interfaces Collection

`int size()`

- liefert die Anzahl an Elementen in dieser Collection

`Object[] toArray()`

- liefert ein neues Array mit allen Elementen der Collection
- bei Listen muss die Reihenfolge erhalten bleiben.

`Elementtyp[] toArray(Elementtyp[] array)`

- speichert die Elemente der Collection im Array array
- falls groß genug, sonst wird ein neues Array zurückgeliefert



Verkettete Liste und Array- Liste

Listen

- erste Kategorie von Collections: Listen
- Interface List von Interface Collection abgeleitet
- Eigenschaften
 - Elemente haben Reihenfolge
 - Elemente können mehrfach vorkommen (an unterschiedlichen Positionen)

Wichtige Methoden des Interfaces List

`Elementtyp get(int index)`

- liefert das Element an Listenposition `index`
- löst eine `IndexOutOfBoundsException` aus, wenn der `index` ungültig ist, also wenn `(index < 0 || index >= size())`

`Elementtyp set(int index, Elementtyp element)`

- ersetzt das Element an Listenposition `index` durch `element`
- liefert das alte Element zurück.

Wichtige Methoden des Interfaces List

`int indexOf(Elementtyp element)`

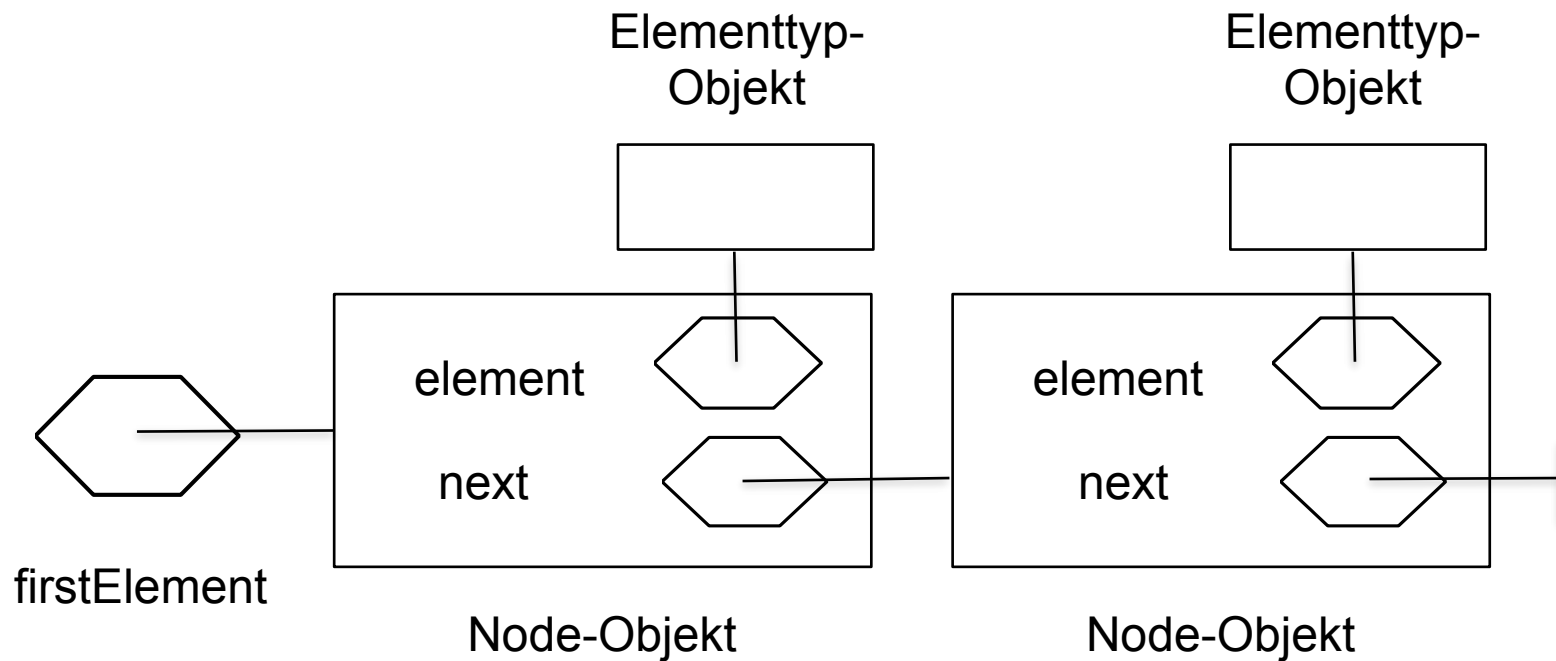
- liefert den Index (Listenposition) des ersten Vorkommens von element in der Liste
- oder -1, falls element nicht in der Liste enthalten ist
- zum Vergleichen wird die equals-Methode des Elementtyps verwendet!

Referenzimplementierungen

- Interface List hat zwei Referenzimplementierungen:
 - Klasse LinkedList: verkettete Liste
 - Klasse ArrayList: verhält sich wie ein Array

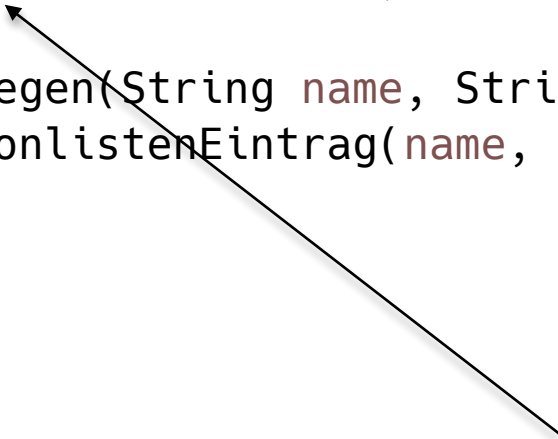
Klasse LinkedList

- Implementierung des Interfaces List durch Objektzeiger



Beispielanwendung: Telefonliste

```
public class Telefonliste {  
  
    private List<TelefonlistenEintrag> eintraege =  
        new LinkedList<TelefonlistenEintrag>( );  
  
    public void eintragHinzufuegen(String name, String nummer) {  
        eintraege.add(new TelefonlistenEintrag(name, nummer));  
    }  
}
```



hier wird der Typ der
Elemente der Liste
festgelegt

Konvertierung

- Collections können in andere Typen überführt werden

```
List<TelefonlistenEintrag> eintraege = new  
    LinkedList<TelefonlistenEintrag>();
```

- Konvertierung in Array:

```
Object[] arrayVonEintraegen = eintraege.toArray();
```

- Problem: Typ Object, daher besser:

```
TelefonlistenEintrag[] arrayVonEintraegen = new  
    TelefonlistenEintrag[eintraege.size()];  
eintraege.toArray(arrayVonEintraegen);
```

- Konvertierung von LinkedList in ArrayList

```
List<TelefonlistenEintrag> eintraegeAlsArrayList = new  
    ArrayList<TelefonlistenEintrag>(eintraege);
```

Klasse ArrayList

- Implementierung des Interface List durch ein Array
 - mit automatischer Größenanpassung

- Beispielanwendung: Telefonliste

- einzige Änderung gegenüber LinkedList-Version:

```
private ArrayList<PhoneListEntry> eintraege = new  
    ArrayList<PhoneListEntry>();
```

- anstelle von

```
private LinkedList<PhoneListEntry> eintraege = new  
    LinkedList<PhoneListEntry>();
```

- der restliche Code bei der Verwendung bleibt derselbe wie für LinkedList
 - es werden nur Methoden aus dem Collection-Interface verwendet

Vergleich ArrayList und LinkedList

- Eigenschaften von ArrayList:
 - Indexzugriff auf Elemente (z.B. `get(10)`) ist überall schnell
 - Einfügen und Löschen ist nur am Listenende schnell, am Listenanfang langsam
- Eigenschaften von LinkedList:
 - Indexzugriff auf Elemente ist an den Enden schnell, in der Mitte langsam
 - wegen Java-Implementierung mit doppelten Zeigern
 - jeweils auf Erstes Element + Nachfolger und letztes Element + Vorgänger
 - Einfügen und Löschen ohne Indexzugriff ist überall schnell

Initialisierung nicht-leerer Listen

- bisher bekannt: Methode add(Typ element):

```
List<String> liste = new ArrayList<String>();  
liste.add("Jan");  
liste.add("Hein");  
liste.add("Klaas");
```

- elegantere Möglichkeit

```
List<String> listeMitInit = Arrays.asList("Jan", "Hein",  
"Klaas", "Pit");
```

Übung: Liste

- Schreiben Sie Quellcode, der das Folgende macht:
 - Erzeugen einer Liste (Implementierung: verkettete Liste) mit den Einträgen 23, 42, 12, 11
 - Ausgaben des Listeninhalts auf der Konsole (so soll es aussehen: {23, 42, 12, 11})
 - Ausgaben des zweiten Elements auf der Konsole



Iteratoren

Durchlaufen von Collections

- bisher: foreach-Schleife oder Schleife mit Zähler

```
Collection<String> stadtteile = new  
    LinkedList<String>( Arrays.asList("Hafencity", "Wandsbek",  
    "Altona"));  
for ( String stadtteil : stadtteile ) {  
    System.out.print(stadtteil + " ");  
}
```

- oder

```
for ( int i = 0; i < stadtteile.size(); i++ ) {  
    System.out.print( stadtteile.get(i) + " " );  
}
```

Durchlaufen von Collections

- Probleme
 - gleicher Zugriff für unterschiedliche Collection-Container
 - Mengen? Elemente haben keine Reihenfolge und daher keinen Index
 - kein Zeiger auf das nächste Listenelement benutzbar
 - notwendig zum Löschen

Iterator

- Lösung: Interface `Iterator<Elementtyp>`
- wird erzeugt durch eine Collection-Methode

```
Iterator<String> stadtteilIterator = stadtteile.iterator();
```
- Iteration über alle Elemente einer Collection
- Zusicherung: alle Elemente werden besucht
- keine Zusagen zur Reihenfolge der Elemente
- Iterator "zeigt" zu jedem Zeitpunkt auf ein Element

Methoden

```
public boolean hasNext()
```

- liefert true, wenn der Iterator auf ein existierendes Element zeigt

```
public Elementtyp next()
```

- liefert das aktuelle Element und geht zum nächsten Element weiter
- falls nicht vorhanden: NoSuchElementException

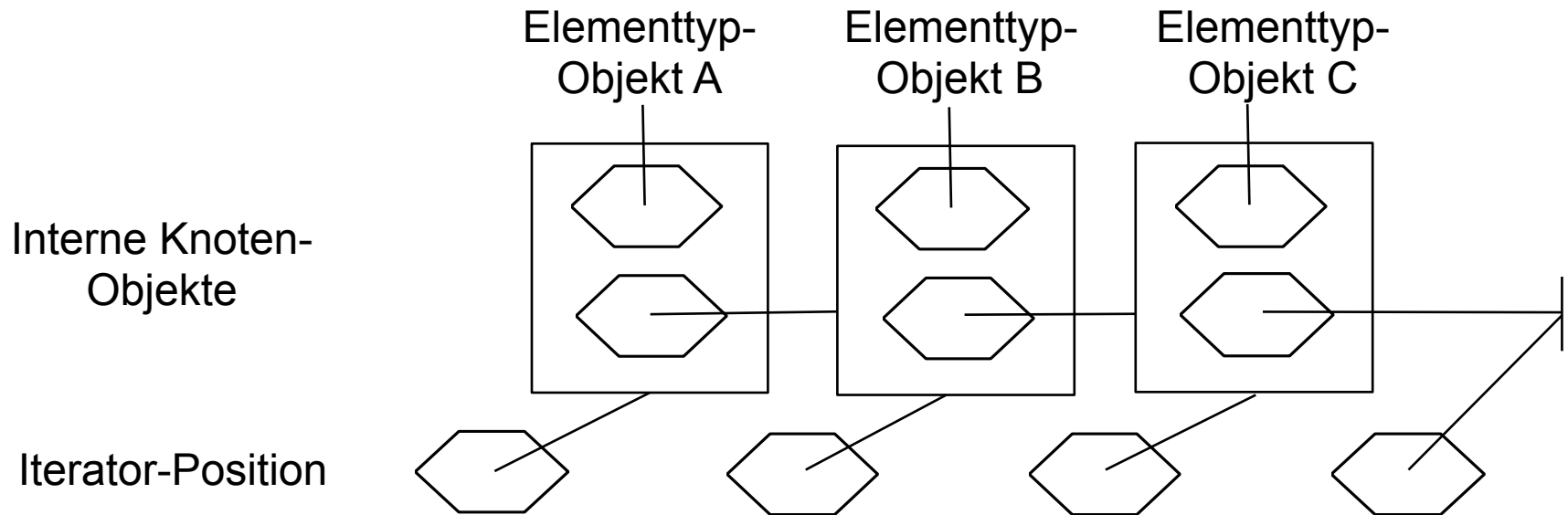
```
public void remove()
```

- entfernt das zuletzt mit next() aktuelle Element aus der Collection
- zeigt dann auf den nachfolgenden Knoten falls nicht vorhanden: IllegalStateException

Iteratoren

```
Iterator<String> iter = liste.iterator();  
while (iter.hasNext()) {  
    String stadtteil = iter.next();  
    System.out.println(stadtteil);  
}
```

Iteratoren



<code>hasNext()</code>	true	true	true	false
<code>next()</code>	Elementtyp-Objekt A	Elementtyp-Objekt B	Elementtyp-Objekt C	NoSuch-Element-Exception
<code>remove()</code>	entfernt Objekt A	entfernt Objekt B	entfernt Objekt C	Illegal-State-Exception

Beispiel: Nummer in Telefonliste finden

// "Traditionelle" for-Schleife

```
for (TelefonlistenEintrag eintrag : eintraege) {  
    if (eintrag.getName().equals(name)) {  
        return eintrag.getNummer();  
    }  
}
```

// Iterator mit while-Schleife

```
Iterator<TelefonlistenEintrag> it = eintraege.iterator();  
while (it.hasNext()) {  
    TelefonlistenEintrag eintrag = it.next();  
    if (eintrag.getName().equals(name)) {  
        return eintrag.getNummer();  
    }  
}
```

// Iterator mit for-Schleife

```
for (it = eintraege.iterator(); it.hasNext();) {  
    TelefonlistenEintrag eintrag = it.next();  
    if (eintrag.getName().equals(name)) {  
        return eintrag.getNummer();  
    }  
}
```


Übung: Iteratoren

- Durchlaufen Sie die List<Integer> liste (23,42,11,12) zweimal
 - einmal mit einer for-Schleife
 - einmal mit einer while-Schleife
- Verwenden Sie in beiden Fällen Iteratoren



Vergleichen

Motivation

- häufige Anforderung (nicht nur bei Collections): Vergleichen zweier Objekte
- bisher kennengelernt: equals()
 - nicht vergessen: pro Klasse überschreiben
 - nur Vergleich "gleich" oder "ungleich"
 - fehlt: größer oder kleiner
- Möglichkeit 1: eigene Methode implementieren
 - z.B. boolean istGroesser(Bruch andererBruch)
 - Nachteil: keine Standardisierung
- besser: Standard-Interface Comparable<T>

Interface Comparable

- definiert in java.lang
- Generisches Interface
 - d.h. automatische Definition von Comparable<Typ>
- einzige Methode:
`int compareTo(Typ anderesObjekt)`
- Aufrufer-Objekt (this) wird mit einem Objekt derselben Klasse (anderesObjekt) verglichen
- es wird ein int-Wert als Ergebnis zurückgeliefert:
 - > 0: wenn this > anderesObjekt
 - < 0: wenn this < anderesObjekt
 - 0: wenn this und anderesObjekt gleich sind

Beispiel: Sortieren der Telefonlisteneinträge

```
public class TelefonlistenEintrag implements  
    Comparable<TelefonlistenEintrag> {
```

Implementieren des
Interfaces

```
    private final String name;  
    private final String nummer;
```

diese Methode muss
implementiert werden

```
    @Override  
    public int compareTo(TelefonlistenEintrag andererEintrag) {  
        return getName().compareTo(andererEintrag.getName());  
    }  
}
```

String implementiert
ebenfalls das Interface,
Wiederverwendung

Interface Comparable

- Anwendung
 - viele Einsatzmöglichkeiten
 - z.B. Sortieren: `Collections.sort(eintraege);`

- Beispiel:

```
System.out.println(telefonliste);  
telefonliste.sortieren();  
System.out.println(telefonliste);
```

- liefert

*Mehmet Scholl: 1121718, Ikke Häßler: 736712027, Zizu Zidane:
674237423*

*Ikke Häßler: 736712027, Mehmet Scholl: 1121718, Zizu Zidane:
674237423*

Interface Comparator

- manchmal Comparable nicht ausreichend
 - Sortieren von Objekten einer Klasse, die nicht das Interface Comparable implementieren
 - Implementierung verschiedener Sortierkriterien
 - Lösung
 - Definition einer Klasse, die das Interface `Comparator<ImplementingClass>` (Package: `java.util`) implementiert mit der einzigen Methode
- ```
public int compare(Typ objekt1, Typ object2)
```
- Rückgabewerte wie bei `compareTo`

# Beispiel: Komparator für Telefonliste

```
public class TelefonlisteEintragComparator implements
 Comparator<TelefonlistenEintrag> {

 @Override
 public int compare(TelefonlistenEintrag eintrag1,
 TelefonlistenEintrag eintrag2) {
 return eintrag1.compareTo(eintrag2);
 }
}
```

- Sortieren

```
// Verwendung des Interfaces Comparable in TelefonlistenEintrag
Collections.sort(eintraege);
```

```
// Verwendung eines Komparators
Collections.sort(eintraege, new TelefonlisteEintragComparator());
```



# Übung: Vergleichen

- Erweitern Sie die Klasse Bruch, sodass sie das Interface Comparable implementiert.
- Erinnerung:

```
class Bruch {

 private int zaehler;

 private int nenner;

 public Bruch(int zaehler, int nenner) {
 this.zaehler = zaehler;
 this.nenner = nenner;
 }
}
```

# Zusammenfassung

- Collections-Framework
- Verkettete Liste und Array-Liste
- Iteratoren
- Vergleichen