

# Inhalt

- 9 Vergleiche
  - Schnittstelle Comparable<T>
  - Schnittstelle Comparator<T>

# Motivation

- ▶ Szenario: (sehr) große Datenmenge
- ▶ Aufgabe: Suche eines Elements mit gegebenem Wert, zB:  
`boolean contains(T e)`
- ▶ Aufwand (in allen bisher bekannten Datenstrukturen):  $\mathcal{O}(n)$

Der Aufwand könnte erheblich reduziert werden, wenn der Datenbestand **sortiert** vorliegen würde!

dazu notwendig:

- ▶ Datensätze miteinander **vergleichen** können
- ▶ Daten des zugrunde liegende Typs  $T$  sollten „**vergleichbar**“ sein

# Schnittstelle Comparable<T>

Das (API-)Interface Comparable<T> fordert eine Vergleichsfunktion:

```
int compareTo(T o)
```

Mittels dieser Methode kann „kleiner“, „gleich“ und „größer“ für den Typ  $T$  definiert werden:

Der Aufruf `a.compareTo(b)` liefert

- ▶ einen **negativen** Wert, gdw  $a$  „kleiner“ als  $b$
- ▶ den int-Wert **0**, gdw  $a$  „gleich“  $b$
- ▶ einen **positiven** Wert, gdw  $a$  „größer“ als  $b$

# Die Methode compareTo(T e)

Die Java-Dokumentation formuliert folgende Bedingungen an die Implementierung:

- ▶ Es ist „dringend empfohlen“ die Methode so zu implementieren, dass sie „konsistent“ mit equals ist, dass also  $a.compareTo(b) == 0 \Leftrightarrow a.equals(b)$

Es ist **sicher zu stellen**, dass für alle  $a, b, c \in T$  gilt:

- ▶  $a.compareTo(b) < 0 \Leftrightarrow b.compareTo(a) > 0$
- ▶  $(a.compareTo(b) < 0 \wedge b.compareTo(c) < 0) \Rightarrow a.compareTo(c) < 0$
- ▶  $a.compareTo(b) == 0 \Rightarrow (a.compareTo(c) < 0 \Leftrightarrow b.compareTo(c) < 0)$

# Natürliche Ordnung

- ▶ Durch die oben genannten Bedingungen induziert die `compareTo()`-Methode eine (totale) **Ordnungsrelation** auf den Objekten des Datentyps.
- ▶ Diese Ordnungsrelation heißt „innere Ordnung“ oder „**natürliche Ordnung**“ des Typs  $T$ .
- ▶ Klassen, die das Interface `Comparable<T>` implementieren, heißen auch „natürlich geordnet“.

Beispiele:

Folgende Klassen der Java-Bibliothek haben eine natürliche Ordnung:

- ▶ die Wrapper-Klassen `Double`, `Float`, `Integer` ...  
Ordnung entspricht der üblichen  $\leq$ -Ordnung
- ▶ `String`: Ordnung im lexikographischen Sinne

## Beispiel: Klasse Stud

Wir wollen Stud-Objekte nach ihrer Matrikelnummer vergleichen:

```
public class Stud implements Comparable<Stud> {  
  
    private String name;  
    private int matNr;  
  
    // ...  
  
    @Override  
    public int compareTo(Stud s) {  
        return this.matNr - s.matNr;  
    }  
}
```

# Inhalt

- 9 Vergleiche
  - Schnittstelle Comparable<T>
  - Schnittstelle Comparator<T>

# Funktionales Interface Comparator<T>

- ▶ Das Interface Comparator<T> definiert (ausschließlich) die Methode `compare(T t, T s)`
- ▶ Die Methode `compare(T t, T s)` hat die gleiche Semantik wie `t.compareTo(s)` der Schnittstelle Comparable<T> ,
- ▶ das heißt: für einen Komparator `c` und zwei `T`-Objekte `a` und `b` liefert der Aufruf `c.compare(a, b)`
  - einen **negativen** Wert, gdw `a` „kleiner“ als `b`
  - den int-Wert `0`, gdw `a` „gleich“ `b`
  - einen **positiven** Wert, gdw `a` „größer“ als `b`
- ▶ Auch diese Methode „sollte“ verträglich mit `equals()` sein, dh es sollte `a.equals(b)` immer denselben Wahrheitswert liefern wie `c.compare(a, b)` (für jeden Comparator `c`).
- ▶ Auch diese Methode induziert eine **Ordnungsrelation** auf dem Datentyp `T`.



# Comparable vs Comparator

Unterschiede zur Schnittstelle Comparable<T>

- ▶ Durch einen Komparator kann einer Klasse eine „**äußere Ordnung**“ gegeben werden, auch wenn die Klasse keine „innere“ Ordnung hat.
- ▶ Einer Klasse mit innerer (natürlicher) Ordnung können weitere alternative Ordnungskriterien gegeben werden.

Die beiden Schnittstellen Comparable<T> und Comparator<T> mit ihren beiden Methoden `t.compareTo(s)` bzw. `c.compare(a, b)` stellen **alternative** Möglichkeiten dar, Objekte miteinander zu vergleichen.

Sie sind **nicht** - wie die Schnittstellen `Iterator<T>` und `Iterable<T>` - einander ergänzende Konzepte.

# Beispiel BubbleSort

Mit innerer Ordnung

(denn Integer **implements** Comparable<Integer>)

```
public static void bubbleSort(Integer[] a) {  
    for (int n = a.length; n > 1; n--)  
        for (int i = 0; i < n-1; i++)  
            // if (a[i] > a[i+1])  
            if (a[i].compareTo(a[i+1]) > 0)  
                swap(a, i, i+1);  
}
```

Mit äusserer Ordnung

```
public static <T> void bubbleSort(T[] a, Comparator<T> c) {  
    for (int n = a.length; n > 1; n--)  
        for (int i = 0; i < n-1; i++)  
            if (c.compare(a[i], a[i+1]) > 0)  
                swap(a, i, i+1);  
}
```

# Anwendung Personen-Array (1)

Wenn wir ein Personen-Array nach Name sortieren wollen:

Nutze die natürliche Ordnung von Strings:

```
public class NameComparator implements Comparator<Person> {  
    @Override  
    public int compare(Person o1, Person o2) {  
        return o1.name().compareTo(o2.name());  
    }  
}
```

Nutzung des Komparators:

```
Person d = new Person("Deniz", 1987)  
Person a = new Person("Anna", 2000);  
// ...  
Person[] p = {d, a, c, e, b};  
bubbleSort(p, new NameComparator());
```

## Anwendung Personen-Array (2)

Wenn wir ein Personen-Array nach Alter sortieren wollen:

```
public class AlterComparator implements Comparator<Person> {  
    @Override  
    public int compare(Person o1, Person o2) {  
        return o1.gebJahr() - o2.gebJahr();  
    }  
}
```

Nutzung des Komparators:

```
Person d = new Person("Deniz", 1987)  
Person a = new Person("Anna", 2000);  
// ...  
Person[] p = {d, a, c, e, b};  
bubbleSort(p, new AlterComparator());
```