# Problemstellung: Sortieren

## Beispiel: Eine Liste mit Personen

| | | | | | |
|---|---|---|---|---|---|
| Aubine | Boice | 4 | aboice3@si.edu | Female | 168.71.9.183 |
| Ariela | Dearle | 12 | adearleb@eventbrite.com | Female | 173.134.1.151 |
| Adeline | McLeish | 18 | amcleishh@zimbio.com | Female | 175.230.40.94 |
| Bernice | Andrzejczak | 13 | bandrzejczakc@squidoo.com | Female | 98.126.168.21 |
| Burty | Deinhardt | 5 | bdeinhardt4@pcworld.com | Male | 126.152.85.239 |
| Carter | Donet | 2 | cdonet1@unc.edu | Genderqueer | 56.47.22.73 |
| Derrick | Kropp | 14 | dkroppd@icq.com | Male | 172.189.31.27 |
| Ellary | Pask | 8 | epask7@ebay.com | Male | 212.220.123.92 |
| Hollie | Elijah | 9 | helijah8@indiegogo.com | Female | 4.105.153.54 |

Die Liste beinhaltet **Elemente** (Items), ein Element entspricht einer Zeile, sie soll sortiert werden nach einem (Sortier-)**Schlüssel** (Key). In diesem Fall soll der Sortierschlüssel der Nachname sein.

Die Liste ist dann sortiert, wenn die Elemente anhand des Sortierschlüssels in eine **Reihenfolge** gebracht worden sind.

| | | | | | |
|---|---|---|---|---|---|
| Adeline | Boice | 4 | aboice3@si.edu | Female | 168.71.9.183 |
| Ariela | Bartlett | 3 | lbartlett2@technorati.com | Female | 208.201.22.15 |
| Aubine | Andrzejczak | 13 | bandrzejczakc@squidoo.com | Female | 98.126.168.21 |
| Bernice | Curtoys | 15 | lcurtoyse@desdev.cn | Female | 92.18.63.145 |
| Burty | Dearle | 12 | adearleb@eventbrite.com | Female | 173.134.1.151 |
| Carter | Deinhardt | 5 | bdeinhardt4@pcworld.com | Male | 126.152.85.239 |
| Derrick | Donet | 2 | cdonet1@unc.edu | Genderqueer | 56.47.22.73 |
| Ellary | Elijah | 9 | helijah8@indiegogo.com | Female | 4.105.153.54 |
| Hollie | Howes | 7 | khowes6@mozilla.org | Female | 77.185.37.165 |
| Jojo | Kropp | 14 | dkroppd@icq.com | Male | 172.189.31.27 |
| Kai | McBeice | 19 | lmcbeicei@sciencedaily.com | Female | 114.35.75.115 |

Für verschiedene Schlüssel ergeben sich andere Reihenfolgen der Elemente:

| | | | | | |
|---|---|---|---|---|---|
| Adeline | Boice | 4 | aboice3@si.edu | Female | 168.71.9.183 |
| Burty | Dearle | 12 | adearleb@eventbrite.com | Female | 173.134.1.151 |
| Karla | McLeish | 18 | amcleishh@zimbio.com | Female | 175.230.40.94 |
| Aubine | Andrzejczak | 13 | bandrzejczakc@squidoo.com | Female | 98.126.168.21 |
| Carter | Deinhardt | 5 | bdeinhardt4@pcworld.com | Male | 126.152.85.239 |
| Derrick | Donet | 2 | cdonet1@unc.edu | Genderqueer | 56.47.22.73 |
| Jojo | Kropp | 14 | dkroppd@icq.com | Male | 172.189.31.27 |
| Lavina | Pask | 8 | epask7@ebay.com | Male | 212.220.123.92 |
| Ellary | Elijah | 9 | helijah8@indiegogo.com | Female | 4.105.153.54 |
| Ronda | Shakespear | 11 | jshakespeara@google.ru | Female | 225.177.127.18 |
| Hollie | Howes | 7 | khowes6@mozilla.org | Female | 77.185.37.165 |

| | | | | | |
|---|---|---|---|---|---|
| Linnet | Pettigrew | 16 | rpettigrewf@usnews.com | Female | 109.148.204.18 |
| Kai | McPeice | 19 | lmcpeicei@sciencedaily.com | Female | 114.35.75.115 |
| Carter | Deinhardt | 5 | bdeinhardt4@pcworld.com | Male | 126.152.85.239 |
| Lindie | Pettie | 6 | mpettie5@blog.com | Female | 161.132.43.23 |
| Adeline | Boice | 4 | aboice3@si.edu | Female | 168.71.9.183 |
| Jojo | Kropp | 14 | dkroppd@icq.com | Male | 172.189.31.27 |
| Burty | Dearle | 12 | adearleb@eventbrite.com | Female | 173.134.1.151 |
| Karla | McLeish | 18 | amcleishh@zimbio.com | Female | 175.230.40.94 |
| Ariela | Bartlett | 3 | lbartlett2@technorati.com | Female | 208.201.22.15 |
| Lavina | Pask | 8 | epask7@ebay.com | Male | 212.220.123.92 |
| Ronda | Shakespear | 11 | jshakespeara@google.ru | Female | 225.177.127.18 |

# Ziel: Wir wollen alles mögliche Sortieren

## Beispiel 1:

Eine Liste zufälliger reeller Zahlen soll sortiert werden[1)]:

```
public class Experiment
{
   public static void main(String[] args)
   {
      int N = Integer.parseInt(args[0]);
      Double[] a = new Double[N];
      for (int i = 0; i < N; i++)
         a[i] = StdRandom.uniform();
      Insertion.sort(a);
      for (int i = 0; i < N; i++)
         StdOut.println(a[i]);
   }
}
```

```
% java Experiment 10
0.08614716385210452
0.09054270895414829
0.10708746304898642
0.21166190071646818
0.363292849257276
0.460954145685913
0.5340026311350087
0.7216129793703496
0.9003500354411443
0.9293994908845686
```

## Beispiel 2:

Eine Wort-Liste soll sortiert werden:

```
public class StringSorter
{
   public static void main(String[] args)
   {
      String[] a = In.readStrings(args[0]);
      Insertion.sort(a);
      for (int i = 0; i < a.length; i++)
         StdOut.println(a[i]);
   }
}
```

```
% more words3.txt
bed bug dad yet zoo ... all bad yes

% java StringSorter words3.txt
all bad bed bug dad ... yes yet zoo
```

### Beispiel 3:

Die Dateien in einem Verzeichnis sollen nach ihrem Namen sortiert werden:

```
import java.io.File;
public class FileSorter
{
   public static void main(String[] args)
   {
      File directory = new File(args[0]);
      File[] files = directory.listFiles();
      Insertion.sort(files);
      for (int i = 0; i < files.length; i++)
         StdOut.println(files[i].getName());
   }
}
```

```
% java FileSorter .
Insertion.class
Insertion.java
InsertionX.class
InsertionX.java
Selection.class
Selection.java
Shell.class
Shell.java
ShellX.class
ShellX.java
```

### Gemeinsamkeiten

In allen Beispielen wurde ein Array mit der (noch zu implementierenden) Methode `Insertion.sort(array)` sortiert. Dabei hatten sowohl Array-Elemente als auch die Sortierschlüssel verschiedenste Typen – was für den Aufruf der Methode jedoch unerheblich war: Die Methode erhält als Argument das Array und sortiert es. Der Vorgang des Sortierens war vom Typ des Schlüssels, nach dem sortiert werden sollte ist vollkommen abstrahiert – die `sort()` Methode hat immer funktioniert.

## Callbacks und Interfaces

Ein Callback ist eine Referenz auf ausführbaren Code - bei den Sortierbeispielen oben geschieht stets folgendes:

- Das Programm ruft die `sort()`-Methode mit einem Array von Objekten auf.
- Die `sort()`-Methode ruft ihrerseits die `compareTo()` Methode der Objekte auf, um zu ermitteln, in welcher Reihenfolge diese angeordnet werden müssen.

In Java werden solche Callbacks als **Interfaces** implementiert. Datentypen, die man sortieren kann, müssen in Java das **Comparable-Interface** implementieren. Zahlreiche Datentypen machen das "von sich aus", z.B. Integer, Double, String, Date, und File.

## Eigene sortierbare Objekte

Wenn man eigene Objekte implementiert, die sortiert werden sollen, müssen diese eine Methode `compareTo()` implementieren, die entweder -1, +1 oder 0 zurückgibt, je nachdem wie der Vergleich ausfällt. Am besten sieht man das an einem Beispiel:

```
public class Time implements Comparable<Time> // Hier wird angegeben, dass
das Comparable Interface implementiert wird.
{
    private int hour, minute;

    public Time(int h, int m)
    {
        hour = h;
        minute = m;
    }

    public int compareTo(Time that)
    {
       if (this.hour < that.hour ) return -1;
       if (this.hour > that.hour ) return +1;
       if (this.minute > that.minute ) return +1;
       if (this.minute < that.minute ) return -1;
       return 0;
    }
}
```

[1)]

Hört sich erst mal sinnlos an, eine Anwendung kommt später...

From:
https://info-bw.de/ -

Permanent link:
**https://info-bw.de/faecher:informatik:oberstufe:algorithmen:sorting:problemstellung:start?rev=1675677607**

Last update: **06.02.2023 10:00**