# Schnittstellen implementieren am Beispiel Suchbaum

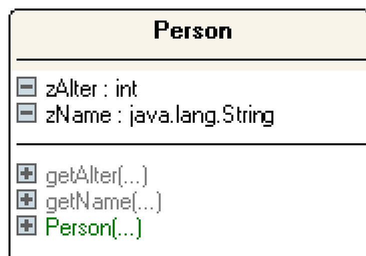
von Bernhard Rosing

## Motivation

Schreiben Sie eine Klasse Person, deren Instanzen in ein TreeSet (Suchbaum) eingefügt werden können, das diese Person-Objekte nach ihrem Alter sortiert.

## Aufgabe 1:

Definieren Sie eine Klasse `Person` nach folgendem Muster




Der Suchbaum weiß nicht, auf welche Weise wir die Elemente sortieren möchten. Wenn man aber eigene Objekte (→ hier Person-Objekte) in den Suchbaum einfügt, muss man auch selbst eine Sortierreihenfolge definieren. Dafür gibt es in Java das vordefinierte Interface `Comparable` (im Package java.lang), mit dessen Hilfe sich Such- und Sortieroperationen für beliebige Objekte definieren lassen, indem man das Interface `Comparable` implementiert.

Die Klasse `String` implementiert bereits das Interface `Comparable`, so dass `String`-Objekte ohne zusätzliche Implementation von Comparable vergleich- und sortiebar sind (Zeichenketten sind auch Objekte.

**Achtung:** Der primitive Datentyp `int` ist keine Klasse , die das Interface `Comparable` implementiert, deswegen meckert der Compiler! ( → keine Objekte von diesem Typ). Man muss stattdessen die Klasse `Integer` verwenden, die einen `int`-Wert kapselt. `Integer` implementiert das Interface `Comparable`, deswegen sollte es damit klappen. (Ab Java 5.0 ist eine Zuweisung wie `Integer k = 42;` möglich! )

Wir verwenden also die vordefinierte Schnittstelle (INTERFACE) **`Comparable`**:

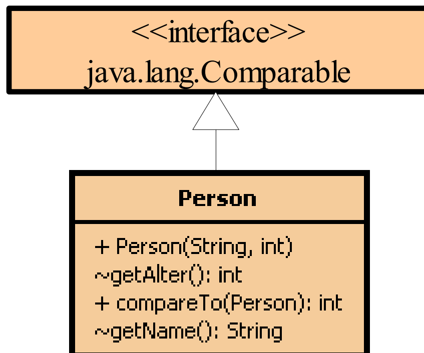
```
public interface Comparable
{
 int compareTo(Object pObject);
}
```

mit der Methode `compareTo(Object pObject)` .

Diese Instanzmethode erwartet als Parameter ein Objekt vom Typ `Object` und gibt einen `Integer` zurück.

**Hinweis:** Die Methode `compareTo` wird im Programm **nicht explizit aufgerufen**. Ihr Aufruf erfolgt über die Methode add() der Klasse TreeSet (s.u.)

**Man kann es so formulieren:** Die Methode add() der Klasse TreeSet verspricht einen Baum von Objekten zu sortieren, falls deren Klassen (→ hier `Person`) sich verpflichten, die Schnittstelle `Comparable` zu implementieren.




Das Interface `Comparable` muss aus dem Paket `java.lang` implementiert werden.

**→ Überblick über die Klassenhierarchien auf Seite 10**

Zentralabitur → Die „Oberklasse“ Item (Vorgaben für das Zentralabitur) dient dazu, Objekte in einen geordneten Baum einzufügen. Es handelt sich hierbei ebenfalls um ein Interface (ADT), weil es eine reine Spezifikation darstellt. Die dort angegebenen abstrakten Methoden (isEqual, isLess und isGreater] müssen in den entsprechenden „Unterklassen“ implementiert werden.

## Das Interface Comparable ( Package java.util)

Das Interface `Comparable` definiert als einzige Methode compareTo(). Jede Klasse, die dieses Interface implementiert, ermöglicht es, ihre Objekte in einer bestimmten Reihenfolge zu sortieren.

Dieses Interface wird ab der Version 5.0 von Java als generisches Interface `Comparable`<E> implementiert, d.h. der Typ-Parameter (können auch mehrere sein) wird in spitzen Klammern hinter den Klassennamen geschrieben. Die Angabe des Typ-Parameters ermöglicht akkurate Typ-Prüfung durch den Compiler.

Lässt man den Typ-Parameter weg, wird er durch `Object` ersetzt, was der früheren Definition ohne Generics entspricht.

Der Methode `public int compareTo(Person obj)` wird hier (geht auch nicht anders!) ein Objekt der Klasse `Person` übergeben.

Die Klasse `Person` muss also in unserem Beispiel eine Implementation der abstrakten Methode `int compareTo(Person pObject)` erhalten.

Die Methode **compareTo** wird aufgerufen, um das aktuelle Element mit einem anderen zu vergleichen.

- `compareTo` muss einen Wert kleiner 0 zurückgeben, wenn das aktuelle Element vor dem zu vergleichenden liegt.
- `compareTo` muss einen Wert größer 0 zurückgeben, wenn das aktuelle Element hinter dem zu vergleichenden liegt.
- `CompareTo` muss 0 zurückgeben, wenn das aktuelle Element und das zu vergleichende gleich sind.

**Man beachte:** Java prüft, ob die Klassendeklaration tatsächlich für jede Operation der Schnittstelle eine passende Methodendeklaration enthält.

```
class Person implements Comparable<Person>
…
…

public int compareTo(Person pObject)
     {
          int vergleich = getName().compareTo(pObject.getName());
          if(vergleich != 0)
          return vergleich;
           else
            return 0;
     }
```

weniger Fehler, weil keine explizite Typumwandlung mehr erforderlich ist!

Beachte: kein Typ-Parameter

## Alternative mit Cast:

Hier muss einer Variablen der „Unterklasse“ vom Typ Person eine Variable einer „Oberklasse“ vom Typ Object zugewiesen werden!

```
class Person implements Comparable
…

…
public int compareTo(Object pObject)
     {
          Person lPerson = (Person) pObject;

          int vergleich = getName().compareTo(lPerson.getName());
          if(vergleich != 0)
          return vergleich;
           else
            return 0;
     }
```

## Bemerkung zur Typumwandlung / Wiederholung

Da die Variable pObject vom Typ Object nur die Operationen kennt, die in dieser Klasse gegeben sind, müssen wir den Typ Object der Variablen pObject durch den Cast-Operator (Person) in den Typ Person umwandeln, um die in der Klasse Person implementierten Methoden (also getName() und getAlter() aufrufen zu können.

Also: pObject verweist zwar auf ein Objekt vom Typ Person, kann aber nicht auf die Operationen in der Klasse Person zugreifen, weil sie nicht im Typ der Oberklasse Object vereinbart sind.

## Aufgabe 2

a) Implementieren Sie das Interface Comparable in der Klasse Person, so dass Personen nach ihrem Namen sortiert werden können.

b) Kommentieren Sie nun die entsprechende Methode compareTo aus und schreiben Sie eine neue Methode compareTo, die Personen nun nach ihrem Alter sortiert. Diese Aufgabe können Sie auch später lösen, nachdem Sie bereits eine kleine Testklasse erzeugt haben (Aufgabe 4). Damit lässt sich besser experimentieren.

## Die Klasse TreeSet (java.util)

Die Sammlungsbibliothek verfügt über eine Klasse TreeSet, die eine sortierte Menge modelliert.
Ein TreeSet verwaltet die Elemente immer sortiert. (Intern werden die Elemente in einem balancierten Binärbaum gehalten.) Speichert TreeSet ein neues Element, so fügt TreeSet das Element automatisch sortiert in die Datenstruktur ein.
Auch die Klasse TreeSet ist eine generische Klasse, d.h. auch hier kann durch formale Typ-Parameter realisiert werden, dass die Klasse nur Objekte eines bestimmten Typs aufnimmt.

```
TreeSet<Person>hatTree = new TreeSet<Person>()
```

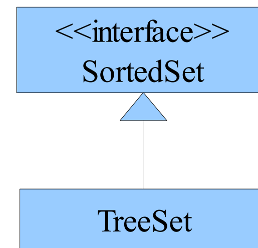
oder auch

```
SortedSet<Person>hatTree = new TreeSet<Person>()
```

(für weitere Funktionalität, s.u.)

Beim Zugreifen auf die Komponenten der Sammlung hatTRee sind keine gefährlichen Cast-Befehle nötig, weil klar ist, dass alle Komponenten der Sammlung hatTree zum Typ Person gehören. Hier wird im Source-Code bereits festgelegt, dass der Baum ein Baum mit Person-Elementen ist. Der Compiler benutzt diese Typinformation und sorgt zur Übersetzungszeit durch entsprechende Typprüfungen dafür, dass nur Person-Objekte in den TreeSet<Person> eingefügt werden.

→ `TreeSet` implementiert das Interface `SortedTree` durch geordnete Bäume




Wichtige Methoden:

**TreeSet():** ist der Konstruktor

**add():** fügt im Parameter angegebenes Element zur Menge hinzu, falls dies noch nicht existiert,

**first():** liefert Element mit dem kleinsten Schlüssel zurück,

**last():** liefert Element mit größtem Schlüsselwert zurück,

**clear():** entfernt alle Elemente aus der Menge,

**contains():** prüft, ob Element (laut Parameter) in Menge enthalten ist,

**remove():** entfernt im Parameter angegebenes Element aus der Menge,

**size():** gibt Anzahl der Element zurück,

**iterator():** liefert eine Iterator-Struktur , die ein Auslesen der Reihe nach ermöglicht

**headSet():** liefert sortierte Elementfolge, die alle kleiner als das im Parameter angegebene Element sind,

## Aufgabe 3:

Lesen Sie die Beschreibung zum Interface `SortedSet` in der Java Klassenbibliothek. Welche zusätzlichen Methoden sind bereits in `TreeSet` implementiert?

## Aufgabe 4:

Definieren Sie nun eine einfache Testklasse, in der eine Sammlung von einigen Person-Objekten erzeugt wird. Gehen Sie nach folgendem Muster vor:

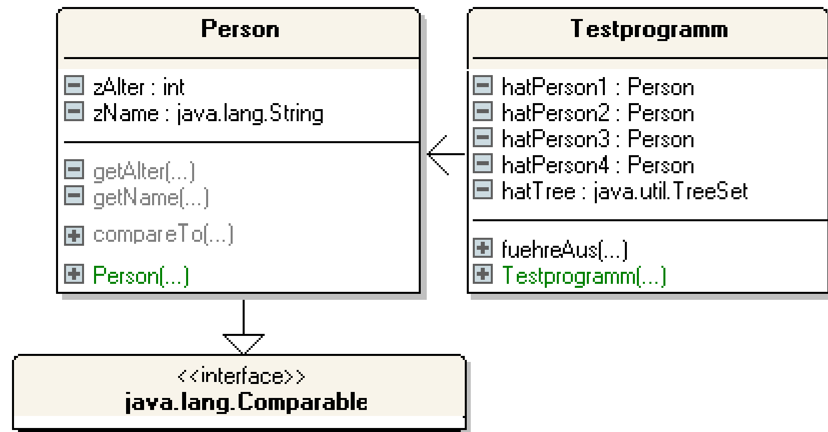
`TreeSet` wird importiert mit `import java.util.TreeSet;`

Wir schreiben die Anweisung

**`TreeSet<Person>hatTree = new TreeSet<Person>()`** z.B. in den Konstruktor der Klasse `Testprogramm`, die ein `TreeSet`-Objekt erzeugt und abspeichert.

Erinnerung:

Sollen Person-Objekte miteinander vergleichbar sein, so muss man in der Klasse Person die Schnittstelle Comparable<Person> implementieren. Die verlangt, dass man eine Vergleichsmethode compareTo mit einem Person-Parameter vereinbart, also so:

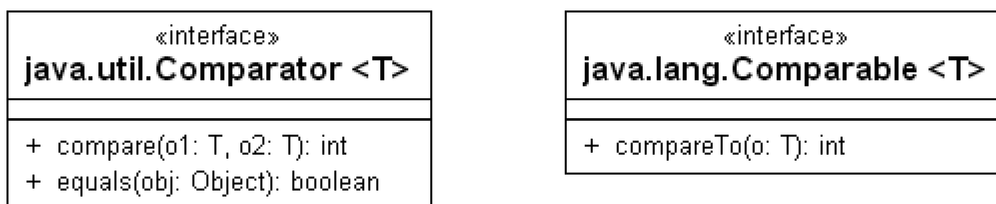



```
class Person implements Comparable<Person>
```

## Aufgabe 5:

Eine flexible Suche anhand verschiedener Kriterien ist nur umständlich über Comparable zu realisieren. (Warum?)

Das Java Collection Framework kennt zwei abstrakte Datentypen, um festzustellen ob ein Objekt grösser, gleich oder kleiner als ein anderes ist: **Comparable** und **Comparator**.




Comparable wird immer von der Klasse des Objekts selbst implementiert. Es stellt somit eine kanonische Ordnung dieser Elemente dar. Darüber hinaus möchte man möglicherweise ein Objekt auch noch nach anderen Kriterien ordnen. In diesem Fall wird jeweils eine neue Klasse programmiert, die Comparator implementiert.

Während Comparable also im Allgemeinen nur ein Sortierkriterium umsetzt, kann es viele Extraklassen vom Typ **Comparator** geben, die jeweils unterschiedliche Ordnungen definieren. Ein Comparator für Namen könnte zum Beispiel die Personen nach Namen sortieren. Ein Comparator für Alter könnte zum Beispiel die Personen nach Alter sortieren.

Comparator definiert eine Methode `int compareTo(Object o1,Object o2)`. Sie hat praktisch das gleiche Verhalten wie bei Comparable - der erste Parameter stellt lediglich das erste zu vergleichende Objekt dar.

```
public interface Comparator<T>
{
  public int compare(T obj1, T obj2);
}
```

Hierbei muss die Funktion **`compareTo(obj1,obj2)`** die folgenden Rückgabewerte liefern:

| | |
|---|---|
| 1 | wenn Objekt obj1 in der gewünschten Ordnung vor obj2 kommt; |
| +1 | wenn obj1 nach obj2 kommt; |
| 0 | wenn obj1 und obj2 an gleicher Stelle kommen. |

**`Comparator`** befindet sich im package java.util.

Unsere `Person`-Klasse implementiert `Comparable`. Aufgrund der oben genannten Vorteile und weil es eben für `Person`-Objekte keine natürliche Ordnung gibt, wollen wir daher zwei externe `Comparator`-Klassen entwickeln, die `Person`-Objekte nach Name bzw. Alter einordnet.

## Aufgabe 6:

Implementieren Sie einen `Comparator` **`PersonNameComparator`**, der ein `Person`-Objekt anhand Ihres Namens vergleicht sowie einen **`PersonAlterComparator`**, der ein `Person`-Objekt anhand ihres Alters vergleicht. Beachten Sie die nachfolgenden Hinweise und testen Sie Ihre Implementierung!

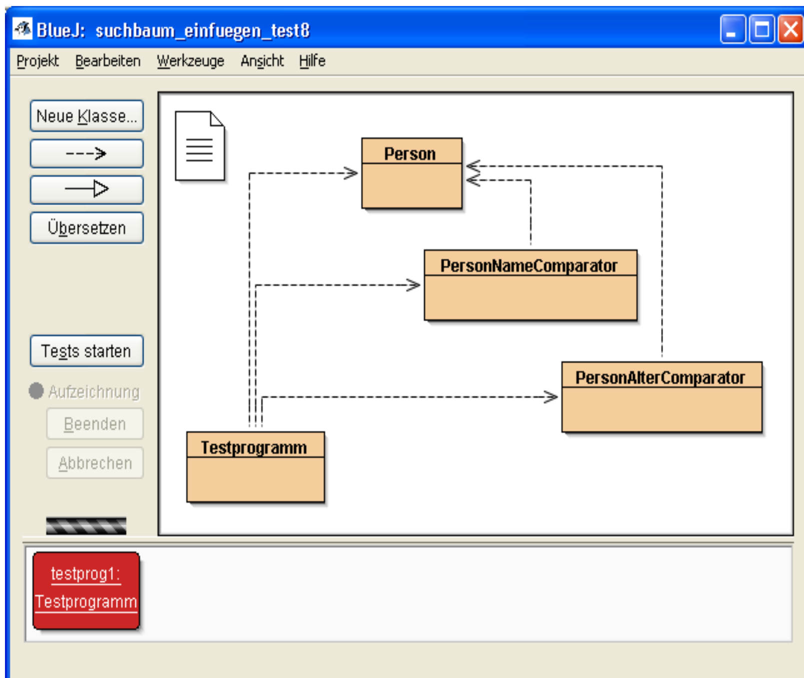
In unserem Testprogramm muss die Anweisung hatTree = new TreeSet ……... neu angepasst werden:

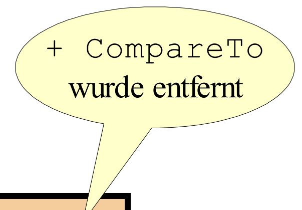
```
hatTree = new TreeSet<Person>(new PersonAlterComparator());

hatTree = new TreeSet<Person>(new PersonNameComparator());
```

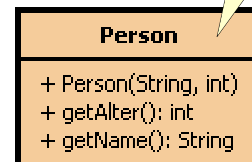
Wird ein `Comparator` an den Konstruktor übergeben, so erfolgt die Einordnung ausschließlich mit Hilfe der Methode `compare` des `Comparator`-Objekts (nicht mit `compareTo` aus `Comparable`!)

Das Testprogramm steht in einer Abhängigkeitsbeziehung zu den beiden Schnittstellen, da sie die Objekte des Schnittstellentyps nutzt.

## Realisierung mit BlueJ:

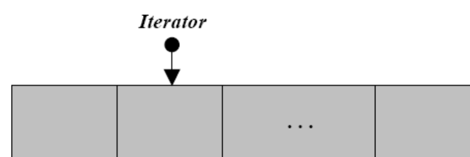





**Noch eine Schwierigkeit:** Wie bekommen wir eine sortierte Konsolenausgabe der `Person`-Objekte?

Der Zugriff auf die Elemente kann über die Schnittstelle `Iterator` erfolgen, die von den Sammlungsklassen bereitgestellt wird. Der Iterationsschritt wird durch den `Iterator` selbst gesteuert; wie er das macht, bleibt ihm selbst überlassen




Iterator definiert drei Methoden:

- `hasNext()` liefert true zurück, wenn der Iterator ein weiteres Element enthält.
- `Next()` liefert dieses nächste Element.
- `Remove()` entfernt das zuletzt geholte Element.

<u>Wichtig:</u> alle anderen Änderungen (Einfügen, Sortieren etc.) an der Datenstruktur führen zu einem undefinierten Zustand des Iterators!

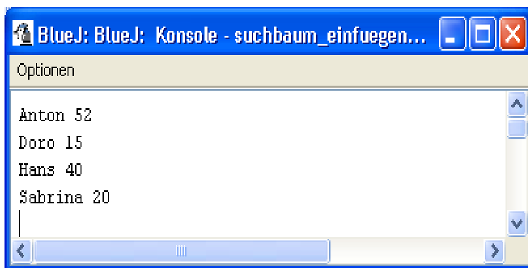
Der Zugriff auf die Elemente des Iterators erfolgt in unserem Beispiel mit der Methode `next()`. Der cast durch (`Person`) ist notwendig, da der Iterator nur `Object` zurückliefern kann.

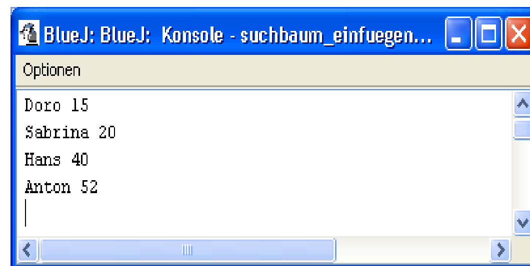
## Aufgabe 7

Ergänzen Sie den folgenden Quelltext und fügen Sie die Programmzeilen in das Testprogramm ein. Damit ist die Aufgabe gelöst!

```
Iterator __________ = __________.iterator();
    while(__________.hasNext())
     {
        ____________________ = (Person)__________.next();
        System.out.println(__________.getName()+ " "
        +__________.getAlter());
     }
```

Konsolenausgabe:




Jetzt sollte es möglich sein, auch ein größeres Projekt in Angriff zu nehmen, in denen die Methoden der Klasse TreeSet zum Tragen kommen. Mit dem SuM-Programmgenerator können dazu entsprechende grafische Benutzungsoberflächen entworfen und eingesetzt werden.

## Beispiel:

Erstellen Sie für das Organisationskomitee der kommenden Fussball-WM ein Programm, das die teilnehmenden Spieler verwaltet. Es sollen Name-Personalnummer-Paare so gespeichert werden, dass die zu einem Namen zugehörige Personalnummer bzw. der zu einer Personalnummer gehörende Name schnell gefunden werden kann. Die Name-Telefonnummer-Paare sollen in Instanzen der Klasse **`Spieler`** abgelegt werden, welche das Interface **`Compararator`** implementiert.

Stellen Sie folgende Operationen bereit:

- Suche eines Namens
- Suche einer Personalnummer
- Einfügen und Löschen einzelner Spieler für Nach- oder Abmeldungen.
- Löschen aller Spieler eines Teams bei dessen Ausscheiden.
- Auflisten aller gemeldeten Spieler in lexikographischer Reihenfolge.

## Überblick

Package java.util

<<interface>> Collection
<<interface>> Set
<<interface>> List
<<interface>> Queue
<<interface>> SortedSet
HashSet
LinkedHashSet
TreeSet
ArrayList
Vector
LinkedList
PriorityQueue

In TreeSet ist das Interface SortedSet bereits implementiert, so dass man weitere nützliche Methoden verwenden kann.

Object
Arrays
Collections
<<interface>> Map
<<interface>> SortedMap
Hashtable
LinkedHashMap
HashMap
TreeMap

implements
extends

Ein Interface in Java ist eine Spezifikation eines Typs, die keine Implementierungen für die Methoden definiert.

Wir haben in unserem Beispiel das Interface `Comparable` und das Interface `Comparator` benutzt sowie das Interface `SortedSet` erwähnt.

In der Klassenbibliothek gibt es eine ganze Reihe von Klassen, die `Comparable` implementieren, z.B. `String` und `Character`. Eigene Klassen müssen jedoch ihre eigene Implementierung von `Comparable` oder `Comparator` beisteuern.

Das Interface `SortedSet` ist bereits in der Klasse `TreeSet` implementiert.

# Lösungen

## Aufgabe 1 / Aufgabe 2

```
import java.util.*;
/**
 * @author b.rosing
 * @version 28.12.2008
*/

//************************************************************************************
/*Die Klasse, deren Objekte später sortiert werden, implementiert Comparable.
  Der Algorithmus wird die Methode compareTo aufrufen, um herauszufinden, welches
  von zwei Objekten grösser ist.
  Die Klasse Person muss folglich das Comparable-Interface implementieren und
  Vergleich der Elemente erfolgt mittels compareTo-Methode
//************************************************************************************
*/

public class Person implements Comparable<Person> //generische Klasse
{
    private String zName;
    private int zAlter;

    // -------------------------Konstruktor------------------------------------

    public Person(String pName, int pAlter)
    {
     zName = pName;
     zAlter = pAlter;
    }

    // ------------------------Dienste----------------------------------------

    public String getName()
    {
      return zName;
    }
    public int getAlter()
    {
     return zAlter;
    }
```

```
//****************************************************************************************
//hier wird der Dienst compareTo implementiert
//****************************************************************************************
/*
 public int compareTo(Person obj)
    {
           if (getAlter() < obj.getAlter())
           return -1;
           else if (getAlter() > obj.getAlter())
            return 1;
                else
                return 0;
    }
*/
//-----------Alternativ: Sortierung nach Namen----------------------------------------
    public int compareTo(Person obj)
    {
        int vergleich = getName().compareTo(obj.getName());
        if(vergleich != 0)
        return vergleich;
         else
          return 0;
    }
//-----------Alternativ: Sortierung nach Namen ohne Typparameter ---------------------
/*
    public int compareTo(Object obj)
    {
        Person lPerson = (Person) obj;
        int vergleich = getName().compareTo(lPerson.getName());
        if(vergleich != 0)
        return vergleich;
         else
          return 0;
    }
*/
}
```

## Aufgabe 4:

```
import java.util.*;
/**
 * @author b.rosing
 * @version 28.12.2008
 */
public class Testprogramm
{//----Objektreferenzen-----------------------------------------------------------------
    private Person hatPerson1;
    private Person hatPerson2;
    private Person hatPerson3;
    private Person hatPerson4;
    //private TreeSet hatTree;
    private TreeSet<Person> hatTree;
    // Konstruktor
    public Testprogramm()
    {
      hatPerson1 = new Person("Hans",40);
      hatPerson2 = new Person("Anton",52);
      hatPerson3 = new Person("Sabrina",20);
      hatPerson4 = new Person("Doro",15);
//*****************************************************************************************
/*TreeSet für Person-Objekte deklarieren und TreeSet erzeugen. TreeSet verwendet als
Datenstruktur einen Binärbaum. Die Elemente Baums sind nun nicht vom Typ Object, sondern
Instanzen der Klasse Person. TreeSet erlaubt keine zwei Einträge, die gleich sind!*/
//*****************************************************************************************
      hatTree = new TreeSet<Person>();  //hatSortedSet = new TreeSet<Person>();
    }
//-----------------------------------Dienste-----------------------------------------
   public void fuehreAus()
   {
     //Person-Objekte anfügen, diese werden nach dem Sortierkriterium sortiert
     hatTree.add(hatPerson1);
     hatTree.add(hatPerson2);
     hatTree.add(hatPerson3);
     hatTree.add(hatPerson4);
//*****************************************************************************************
/*Zugriff auf die Elemente über Iterator, der von den Sammlungsklassen bereitgestellt
wird. Der Iterationsschritt wird durch den Iterator selbst gesteuert; wie er das macht,
bleibt ihm selbst überlassen. Zugriff auf die Elemente des Iterators mit der Methode
next(). Der cast durch (Person) ist notwendig, da der Iterator nur Object zurückliefern
kann. */
//*****************************************************************************************
//Aufgabe 7
}}
```

## Aufgabe 5:

Der Mechanismus ist nicht flexibel genug, da im Allgemeinen nur ein Sortierkriterium implementiert wird (natürliche Ordnung, engl. natural ordering). Mit dem Comparartor-Interface kann einfach je nach Anwendungsfall ein anderer `Comparator` eingesetzt werden.
Eine Softwareanwendung könnte Personen vielleicht auch nach unterschiedlichen Kriterien (Name, Alter, Geburtstag) chronologisch ordnen wollen.

## Aufgabe 6:

```
import java.util.*;
public class PersonNameComparator implements Comparator<Person>
{
  public int compare(Person p1, Person p2)
   {
       String name1 = p1.getName();
       String name2 = p2.getName();
       return name1.compareTo(name2);

       // ODER: return p1.getName().compareTo(p2.getName());
   }
}

import java.util.*;
public class PersonAlterComparator implements Comparator<Person>
{
     public int compare(Person p1, Person p2)
    {
       return p1.getAlter()- p2.getAlter();
    }
}
```

## Aufgabe 7:

```
 Iterator it = hatTree.iterator();
     while(it.hasNext())
     {
      Person lPerson = (Person)it.next();
      System.out.println(lPerson.getName()+ " " +lPerson.getAlter());
     }
```