**Schein-/Bachelorklausur Teil 2 am 13.02.2007**

Zulassung: Mindestens 14 Punkte in Teilklausur 1 und
50% der Übungspunkte aus dem 2. Übungsblock.

**Alle Studiengänge außer Bachelor melden sich über die Lehrstuhlwebseite an.**

**Anmeldebeginn: Mo. 29.01.2007**
**Anmeldeschluß: Fr. 02.02.2007** um 17:00 Uhr.

http://media.informatik.rwth-aachen.de/programmierung_ws0607.html

**Schein-/Bachelorklausur Teil 2 am 13.02.2007**

Zulassung: Mindestens 14 Punkte in Teilklausur 1 und
50% der Übungspunkte aus dem 2. Übungsblock.

**Die Anmeldung für die Bachelorstudenten fand im Dezember 2006 statt.**

Die Anmeldung für die
**Schein-/Bachelor-Nachschreibeklausur am 27.03.2007**
erfolgt über die Lehrstuhlseite nach Bekanntgabe der Ergebnisse der Teilklausur 2 (Ende Februar).

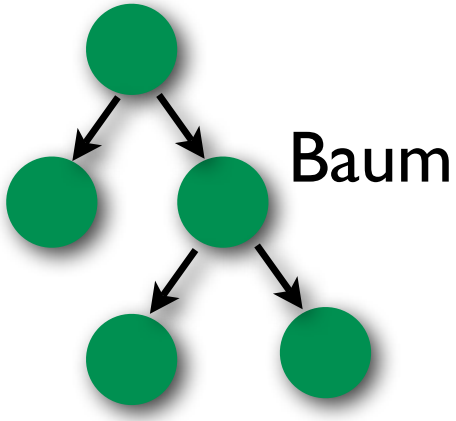
Sie müssen **beide** Schein-/Bachelor Teilklausuren bestehen.

In jeder Teilklausur müssen Sie mindestens
25% der Punkte erreichen.

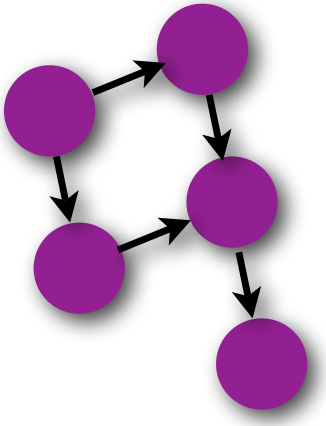
In beiden Teilklausuren zusammen müssen Sie mindestens
50% der Gesamtpunkte **aus beiden Klausuren**
erreichen.

25% in jeder Teilklausur
**!=**
50% in beiden Teilklausuren

# Collections
# Generics
# Dynamische Datenstrukturen

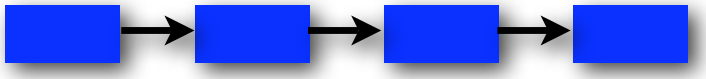


Baum

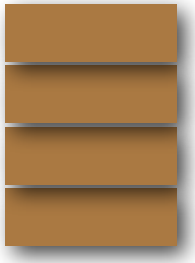


Graph

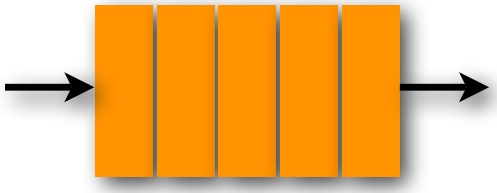
Liste



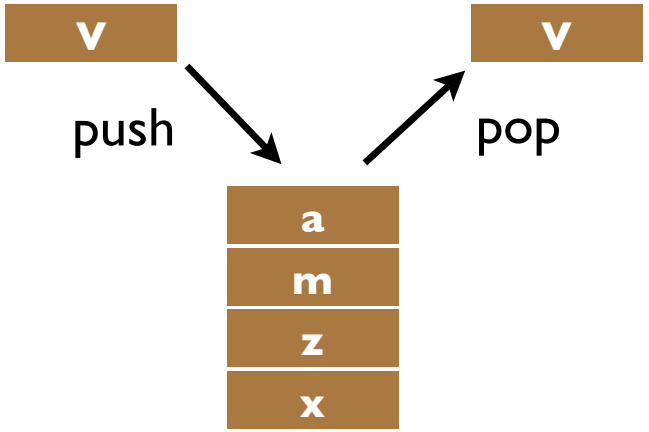
Stapel



Warteschlange

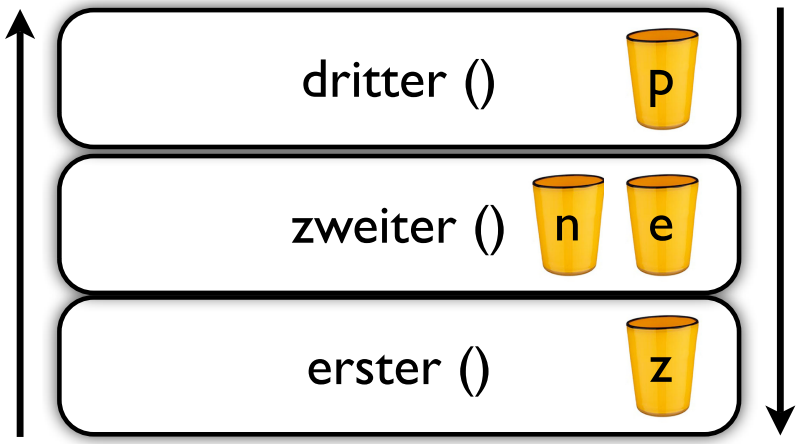


# Stapel / Stack




**LIFO**-Prinzip: Last In, First Out

push(v) speichert oben ein neues Element
v = pop() entfernt das oberste Element



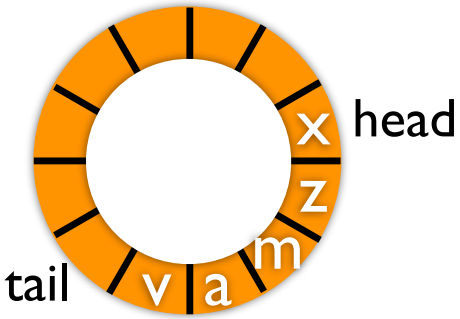

**Stack-Speicher**
**(siehe 07-Kap9.pdf, Folie 15)**

# Warteschlange

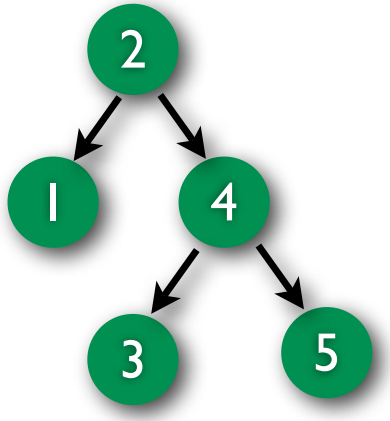

**FIFO**-Prinzip: First In, First Out

put(v) fügt ein Element am Ende der Schlange ein
x = get() entfernt das vorderste Element

Alternative: Warteschlange als *Ringpuffer* — Feste Größe, wobei die ältesten Elemente überschrieben werden können.



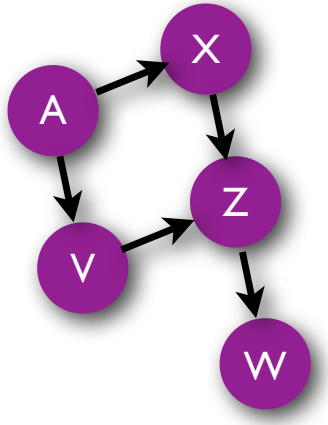
# Bäume und Graphen




## z.B. binärer Suchbaum

Jeder Knoten hat max. 2 Nachfolger: Der linke Unterbaum enthält nur Elemente, deren Werte kleiner als der Wert des Knotens sind. Der rechte Unterbaum enthält nur Elemente, deren Werte größer oder gleich dem Wert des Knotens sind.

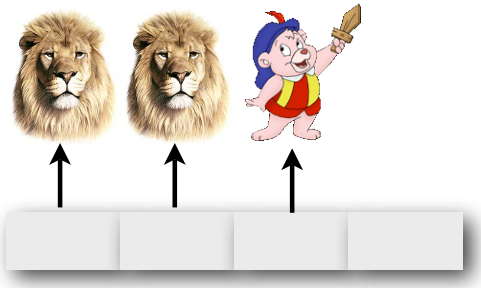
## Graphen



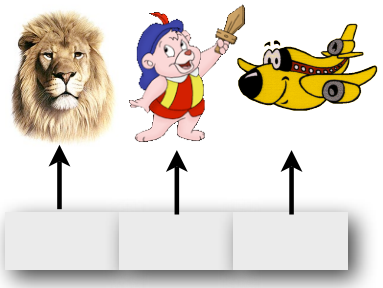

Verbindung zwischen Punkten (Straßen, Eisenbahn), Bestimmung von Routen und Entfernungen, Telefonnetz...

# Collections in Java: Beispiele

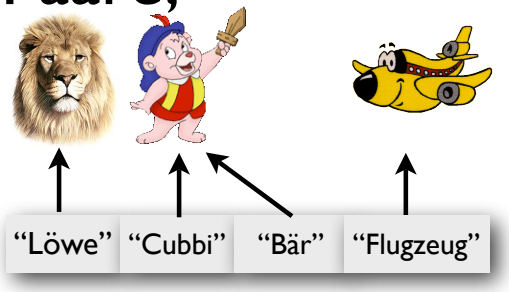
**List:** Indexposition bekannt,
speichert doppelte Elemente



**Set:** Speichert keine doppelten Elemente
(Menge)



**Map:** Elemente sind Name/Wert-Paare,
keine doppelten Schlüssel



# Collections in Java: Beispiele

// Array als Liste (dynamisch)
**ArrayList**

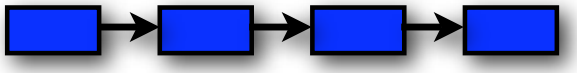
// schnelles Einfügen/Löschen von Elementen in der Mitte der Liste
**LinkedList**

// Elemente in sortierter Reihenfolge ohne doppelte Einträge speichern
**TreeSet**

// keine doppelten Elemente, schnelles Auffinden von Elementen
**HashSet**

// Elemente als Name/Wert-Paare speichern
**HashMap**

# ArrayList



```
String[] namen = new String[100];  // Array hat unveränderbare Länge,
namen[0] = "Hans";                 // zum Suchen eines Wertes muß
...                                // das gesamte Array durchlaufen werden
namen[99] = "Werner";

// ArrayList hat dynamisch Länge
ArrayList<String> namenAL = new ArrayList<String>();
namenAL.add("Hans");
namenAL.add("Werner");
...
namenAL.add(10,"Stefan");          // an Position 10 einfügen, Rest verschieben
namenAL.remove("Stefan");          // Stefan entfernen
namenAL.remove(2);                 // Element an Position 2 entfernen

// suchen & entfernen über Indexposition
namenAL.remove(namenAL.contains("Hans"));
```

# ArrayList sortieren

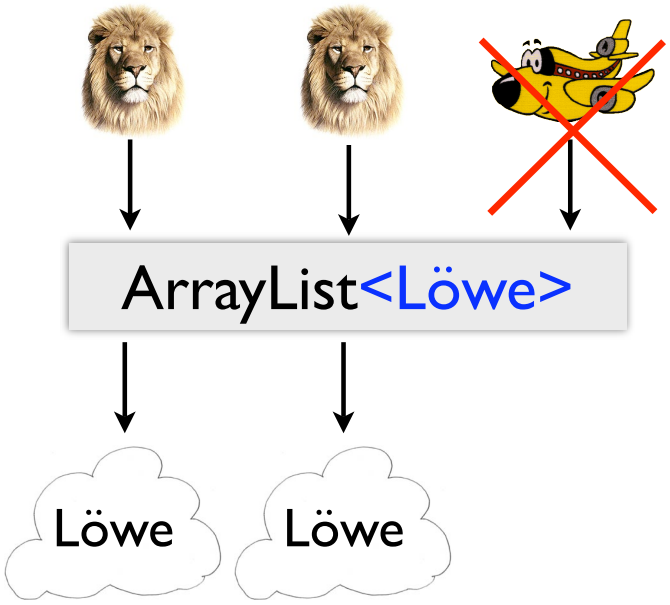

```
String[] namen = new String[100];  // unveränderbare Länge
namen[0] = "Hans";
...
namen[99] = "Werner";

// Die Klasse Arrays sortiert Arrays mit primitiven Elementen / Strings
Arrays.sort(namen);

ArrayList<String> namenAL = new ArrayList<String>();  // dynamisch Länge
namenAL.add("Stefan");
namenAL.add("Werner");
namenAL.add("Hans");

// Die Klasse Collections sortiert Listen mit primitiven Elementen / Strings
Collections.sort(namenAL);
```

# Geht das?


```
public class Buch {
   String titel, autor;  int jahr;

   public Buch (String t, String a, int j) {          // Konstruktor
      titel = t; autor = a; jahr = j;
   }

   public String getTitel() { return titel; }         // Getter & Setter
   public String getAutor() { return autor; }
}

// irgendwo im Programm
ArrayList<Buch> bücherListe = new ArrayList<Buch>();
bücherListe.add(new Buch("Java von Kopf bis Fuß", "Sierra/Bates", 2006));
bücherListe.add(new Buch("Java for Dummies", "Burd", 2006));

// Nein! Wie werden Objekte der Klasse Buch sortiert?
Collections.sort(bücherListe);
```

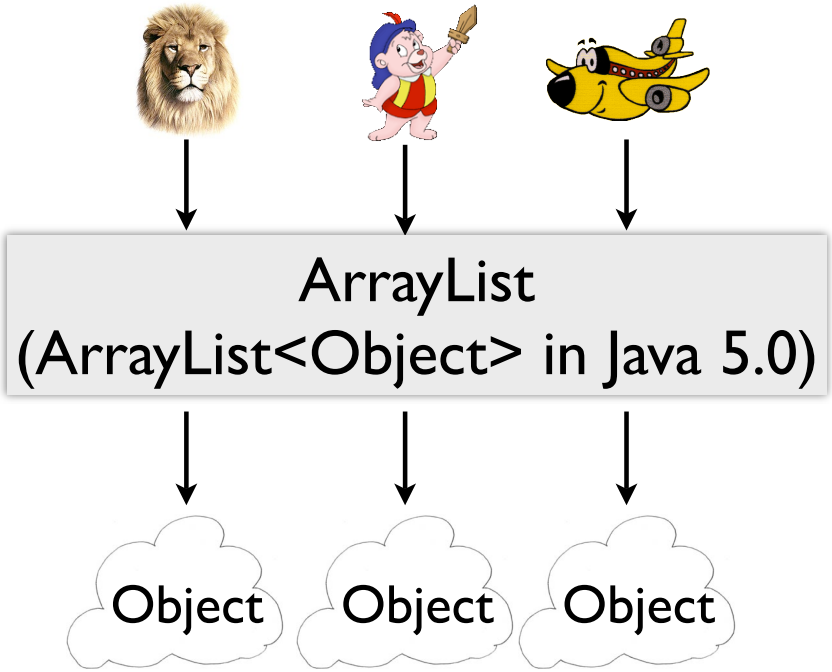
# Typsicherheit durch Generics

mit Java 5.0




nimmt nur Löwe-Objekte auf (oder Unterklassenobjekte von Löwe)

vor Java 5.0



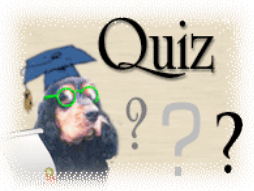

nimmt beliebige Objekte auf

# Generics mit Java 5.0

```
// Definition der Klasse ArrayList
public class ArrayList<E> extends AbstractList<E> implements List<E> ... {
   public boolean add(E o) {
      // E ist Platzhalter für den Elementtyp, den die ArrayList enthalten kann
   }
   ...
}

ArrayList<String> namenListe = new ArrayList<String>();  // E durch String bzw.
ArrayList<Buch> bücherListe = new ArrayList<Buch>();   // durch Buch ersetzen
```

```
// Definition der sort-Methode in der Klasse Collections
public static <T extends Comparable<? super T>> void sort (List<T> list) {
   // T muß vom Typ Comparable sein
   // "T extends" bedeutet bei Generics entweder "T extends" oder "T implements"
   // <? super T> bedeutet: T oder Supertyp von T
   // Der Typ der Liste (list) muß also Comparable erweitern bzw. implementieren.
}
```

```
ArrayList<Flugzeug> flugzeuge = new ArrayList<Flugzeug>();
ArrayList<A380> vieleA380 = new ArrayList<A380>();

// Geht das?
befüllen(flugzeuge);        // ok
befüllen(vieleA380);        // geht nicht!

tanken(flugzeuge);          // ok
tanken(vieleA380);          // A380 extends Flugzeug
```




```
class Flugzeug { ... }
class A380 extends Flugzeug {...}

// nur eine ArrayList<Flugzeug> kann übergeben werden
public void befüllen(ArrayList<Flugzeug> liste) { ... }

// ArrayList<Flugzeug> oder ein Untertyp von Flugzeug kann übergeben werden
public <T extends Flugzeug> void tanken(ArrayList<T> liste) { ... }

public void tanken(ArrayList<? extends Flugzeug> liste) { ... } // alternative Schreibweise
```

# java.lang.Comparable


```
public interface Comparable<T> {
   // liefert -1, 0 oder 1 zurück, je nachdem ob das Objekt "this" kleiner, gleich
   // oder größer als das übergebene Objekt "o" ist.
   int compareTo (T o);
}
```

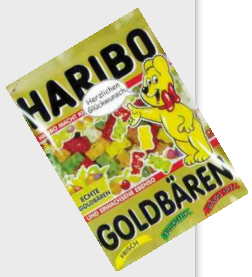
```
// um Buch-Objekte zu sortieren, muß Buch "Comparable<Buch>" implementieren
public class Buch implements Comparable<Buch> {
   String titel, autor;
   int jahr;
   ...
   public int compareTo (Buch b) {          // Bücher nach Titel sortieren über
      return titel.compareTo(b.getTitel()); // compareTo() der Klasse String
   }
}

ArrayList<Buch> bücherListe = new ArrayList<Buch>();  // Bücher hinzufügen...
Collections.sort(bücherListe);                        // ArrayList<Buch> sortieren
```

Wie lautet die Return-Anweisung, um Bücher nach **Autor** zu sortieren?


```
public class Buch implements Comparable<Buch> {
   String titel, autor;
   int jahr;
   ...
   // Bücher nach Autor sortieren lassen
   public int compareTo (Buch b) {
      return ???
   }
}

ArrayList<Buch> bücherListe = new ArrayList<Buch>();
...
Collections.sort(bücherListe);
```

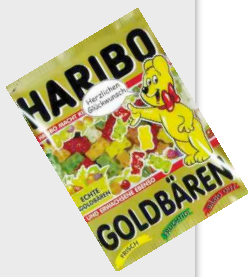
SMS mit Ergebnis an:

**01577 - 287 21 96**



Wie lautet die Return-Anweisung, um Bücher nach **Autor** zu sortieren?


```
public class Buch implements Comparable<Buch> {
   String titel, autor;
   int jahr;
   ...
   // Bücher nach Autor sortieren lassen
   public int compareTo (Buch b) {
      return autor.compareTo(b.getAutor());
   }
}

ArrayList<Buch> bücherListe = new ArrayList<Buch>();
...
Collections.sort(bücherListe);
```



# java.util.Comparator


```
public interface Comparator<T> {
   // liefert -1, 0 oder 1 zurück, je nachdem ob das Objekt "o1" kleiner, gleich
   // oder größer als das Objekt "o2" ist.
   int compare (T o1, T o2);
}
```

```
// Generics in der überladenen sort-Methode der Klasse Collections
public static <T> void sort (List<T> list, Comparator<? super T> c) {
  // Wir übergeben ein Comparator-Objekt, das die Sortierung der Elemente
  // in der Liste übernimmt.  Für jede gewünschte Sortierreihenfolge schreiben wir
  // eine eigene Klasse, die Comparator implementiert.
}
```

Eigene Comparator-Klassen erlauben, mehr als eine Sortierreihenfolge zu definieren.

```
public class Buch implements Comparable<Buch> {
   String titel, autor;
   int jahr;
   ...
   public int compareTo (Buch b) {                    // Bücher nach Titel sortieren
      return titel.compareTo(b.getTitel());
   }
}
public class SortTest {
   ArrayList<Buch> bücherListe = new ArrayList<Buch>();

   // eine innere Comparator-Klasse, die Bücher nach Autor sortiert
   class AutorenVergleich implements Comparator<Buch> {
      int compare (Buch b1, Buch b2) {
         return b1.getAutor().compareTo(b2.getAutor());    // String-Vergleich
      }
   } // evtl. weitere Comparator-Klassen, die Bücher nach Jahr, ISBN, etc. sortieren

   Collections.sort(bücherListe);                          // nach Titel sortieren
   Collections.sort(bücherListe, new AutorenVergleich());  // nach Autor sortieren
}
```

# HashSet: doppelte Elemente entfernen

```
public class Buch implements Comparable<Buch> {
   String titel, autor;
   int jahr;
   ...
}

ArrayList<Buch> bücherListe = new ArrayList<Buch>();
bücherListe.add(new Buch("Java von Kopf bis Fuß", "Sierra, Bates", 2006));
bücherListe.add(new Buch("Java von Kopf bis Fuß", "Sierra, Bates", 2006));
bücherListe.add(new Buch("Java for Dummies", "Burd", 2006));
...
// alle Bücher in ein HashSet hinzufügen, damit Duplikate entfernt werden
HashSet<Buch> bücherSet = new HashSet<Buch>();
bücherSet.addAll(bücherListe);   // alle Elemente auf einmal hinzufügen

// Hmm... "Java von Kopf bis Fuß" gibt es immer noch doppelt?
System.out.println(bücherSet);
```

# Gleichheit von Objekten

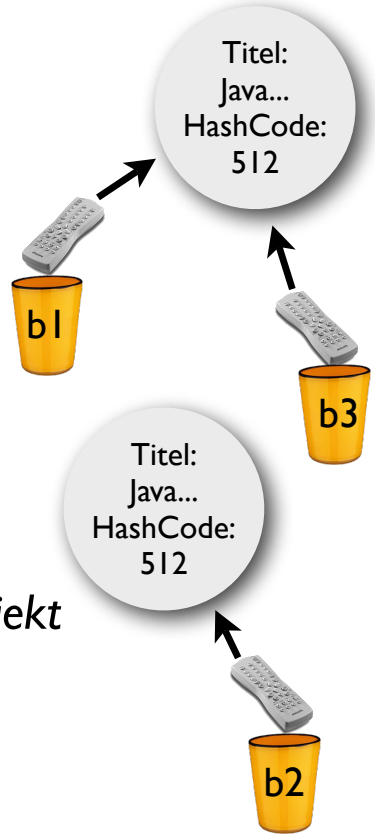
```
// Wann sind zwei Bücher für ein HashSet gleich?
Buch b1 = new Buch(new Buch("Java von Kopf bis Fuß", "Sierra, Bates", 2006)
Buch b2 = new Buch(new Buch("Java von Kopf bis Fuß", "Sierra, Bates", 2006)
Buch b3 = b1;
```

**Referenzgleichheit (identische Bits, ==)**

```
if (b1 == b3) {
   // beide Referenzen verweisen auf dasselbe Objekt auf dem Heap
   // b1.hashCode() == b3.hashCode()
}
```

**Objektgleichheit**

```
if (b1.equals(b2) && b1.hashCode() == b2.hashCode()) {
   // beide Referenzen beziehen sich entweder auf ein und dasselbe Objekt
   // oder auf zwei als gleich geltende Objekte
}
```



# Gleichheit von Objekten

Die Methoden der ultimativen Superklasse Object:

```
public boolean equals(Object obj) {
   // Standardverhalten: liefert true, wenn this == obj
}

public int hashCode() {
   // Standardverhalten: das Objekt erhält eine unverwechselbare Nummer
   // auf Basis der Speicheradresse, die für jedes Objekt eindeutig ist.
}
```

**Für *Objektgleichheit* müssen wir *equals()* und *hashCode()* überschreiben, damit ein Set zwei *unterschiedliche* Objekte, die *inhaltlich gleich* sind, als "gleich" behandelt.**

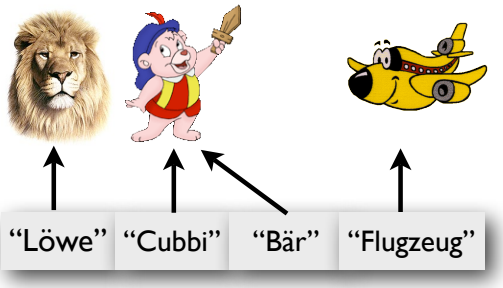
# HashSet: doppelte Elemente entfernen

```
public class Buch implements Comparable<Buch> {
   String titel, autor;
   int jahr;
   ...
   // wir definieren "gleiche Bücher" als "gleiche Titel"
   public boolean equals (Object obj) {        // Strings haben eine überschriebene
      Buch buch = (Buch) obj;                  // equals() und hashCode() Methode
      return getTitel().equals(buch.getTitel()); // "this" mit "o" vergleichen
   }
   public int hashCode () { return titel.hashCode(); }
}

ArrayList<Buch> bücherListe = new ArrayList<Buch>();
HashSet<Buch> bücherSet = new HashSet<Buch>();

// alle Bücher hinzufügen, Duplikate werden jetzt wirklich entfernt :-)
bücherSet.addAll (bücherListe);
```

# TreeSet: Elemente sortieren


```
// Buch muß "Comparable" implementieren
public class Buch implements Comparable<Buch> {
   String titel, autor;
   int jahr;
   ...
   public int compareTo (Buch b) {                    // Bücher nach Titel sortieren
      return titel.compareTo(b.getTitel());
   }
}

ArrayList<Buch> bücherListe = new ArrayList<Buch>();
TreeSet<Buch> bücherTreeSet = new TreeSet<Buch>();  // nach Titel

// alternativ über Comparator-Objekt nach Autor sortieren (siehe Folie 11)
// TreeSet<Buch> bücherTreeSet = new TreeSet<Buch>(new AutorenVergleich());

// alle Bücher hinzufügen, sortieren und Duplikate entfernen
bücherTreeSet.addAll (bücherListe);
```

# HashMap: Schlüssel/Wert-Paare




```
import java.util.*;

class MapTest {
   public static void main(String[] args) {
      // zwei Typparameter: <Schlüsseltyp, Werttyp>
      HashMap<String, Integer> buecherMap = new HashMap<String, Integer>();

      buecherMap.put("Java von Kopf bis Fuß", 2006);
      buecherMap.put("Java for Dummies", 2006);      // gleicher Wert ist ok
      buecherMap.put("Java for Dummies", 2004);      // gleicher Schlüssel geht nicht!

      Integer wert = buecherMap.get("Java for Dummies");  // 2006
      System.out.println(buecherMap);
   }
}
```

Jetzt sind Sie wieder an der Reihe!


Lesen Sie zu Collections und Generics
**Kapitel 16** (Seiten 529-567).