Technische Universität München
Fakultät für Informatik
C. Harlfinger, C. Herzog

WS 1998/99
Lösungsvorschläge zu Blatt 3
28. Oktober 1998

Übungen zum Programmieren in Java

Aufgabe 1 Objektorientierte Modellierung

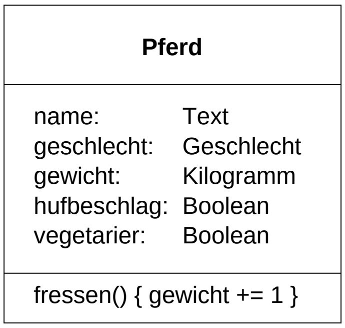
a) In der Objektorientierung bieten sich Klassen zur Modellierung von Tierarten natürlicherweise an: In einer Klasse können alle für eine Tierart spezifischen Eigenschaften modelliert werden. Einzelne Tiere können dann als Instanz einer solchen Klasse beschrieben werden. Jedes Tier stellt also ein eigenes Objekt dar, dessen Aufbau und Verhalten durch die Klasse, der es angehört, festgelegt sind.

Zur Beschreibung von passiven Objekteigenschaften (hier die körperlichen Merkmale eines Tiers, z.B. gewicht) bieten sich Attribute an. Um das Verhalten der Objekte bzw. Tiere zu modellieren, sind dagegen Methoden besser geeignet. Änderungen der körperlichen Merkmale eines Tiers (z.B. Zunahme von gewicht) sind bei strenger Auslegung des Prinzips der *Kapselung* nur als Folge einer Aktivität (z.B. fressen() {gewicht = gewicht + 1}) möglich.

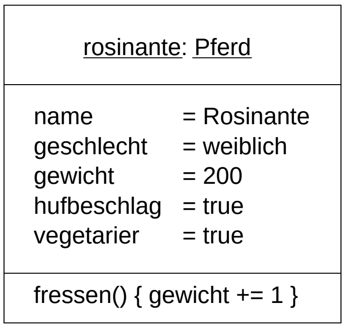
Klassen stellen im Prinzip Baupläne bzw. Schablonen dar, die die Struktur für ihre Instanzen vorgeben. Durch sie werden Namen und Typen der einzelnen Objektattribute festgelegt (z.B. gewicht: Kilogramm). Ihre Individualität erhalten die einzelnen Objekte erst dadurch, daß ihren Attributen konkrete Werte zugewiesen werden, die dann den Merkmalsausprägungen dieses Objekts entsprechen (z.B. gewicht = 50). Die Vorgehensweise bei diesen Zuweisungen ist allerdings wieder für die gesamte Klasse einheitlich durch die in der Klasse vorhandenen Methoden und Konstruktoren geregelt.

b) Die Anzahl der hier angebbaren Eigenschaften ist theoretisch unbegrenzt. Für eine sinnvolle Auswahl sollte deshalb überlegt werden, welche Aufgaben das zu modellierende System erfüllen soll, und welche Eigenschaften dafür unbedingt vorhanden sein müssen.

In dieser Aufgabe erstellen wir nur ein grobes Modell, so daß die Beschreibungen für die Klasse Pferd bzw. für das Objekt rosinante recht einfach ausfallen:

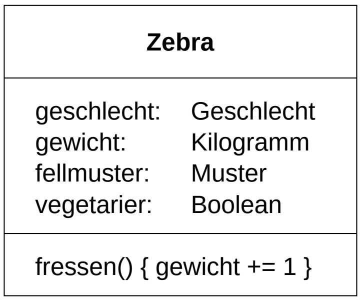


Klassenbeschreibung

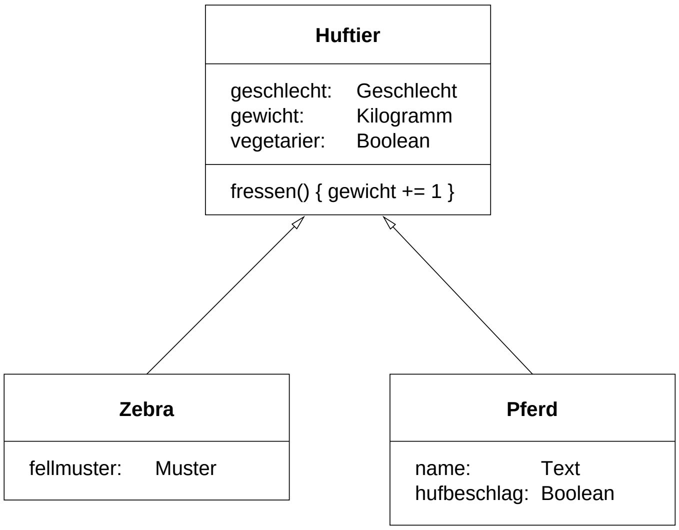


Objektbeschreibung

c) Bei der Betrachtung der Klasse Zebra fallen vor allem die vielen Gemeinsamkeiten mit der Klasse Pferd ins Auge, die die enge Verwandtschaft beider Tierarten auch in der Modellierung wiedergeben. Es bietet sich daher an, durch *Generalisierung* eine gemeinsame Oberklasse zu diesen beiden Klassen zu entwerfen. Aufgrund der engen Verwandtschaft ergibt sich ein relativ kleiner Generalisierungsschritt, d.h. die neu zu erstellende Oberklasse abstrahiert nur geringfügig von ihren Unterklassen.

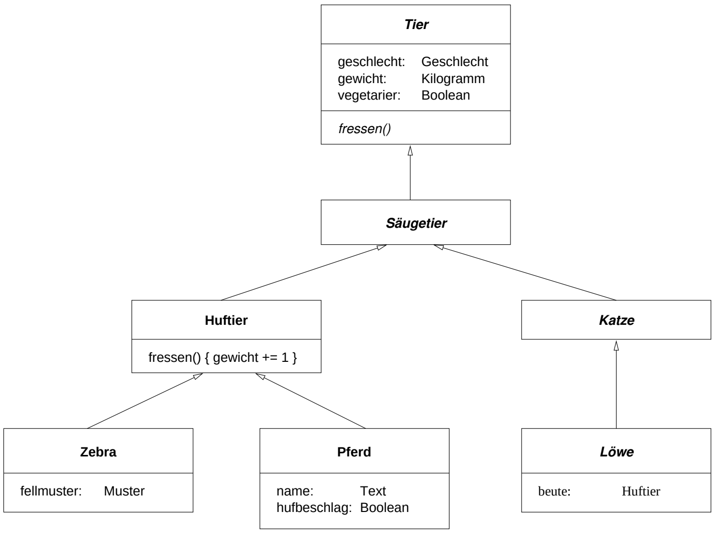


Die aus der Biologie gewohnte Arteneinteilung entspricht wegen ihrer hierarchischen Grundstruktur unserem gewünschten Klassenmodell recht gut. Es ist also naheliegend, auch die Klassenhierarchie entsprechend den Verwandtschaftsbeziehungen der einzelnen Tierarten aufzubauen. Als direkte Oberklasse zu Pferd und Zebra wählen wir daher die Klasse Huftier.



Die Einsparungen, die dadurch erzielt werden, sind offensichtlich. Pferd und Zebra erhalten durch *Vererbung* alle Eigenschaften, die für die Klasse Huftier angegeben sind. Damit wird die mehrfache Deklaration gemeinsamer Attribute und Methoden auf elegante Weise vermieden.

Durch weitere Generalisierungsschritte erhalten wir die Oberklassen Säugetier bzw. Tier, in denen die Methode fressen() nicht mehr konkret angegeben wird, sondern nur noch als *abstrakte Methode* deklariert ist (in Java wird dies durch den Modifikator `abstract` gekennzeichnet). Damit wird lediglich angezeigt, daß diese Methode für alle Tiere deklariert ist. Ihr tatsächliches Verhalten beim Fressen kann in den einzelnen Unterklassen genauer spezifiziert werden. Zwingend notwendig ist die Konkretisierung nur für die Klassen, die tatsächlich instanziiert werden sollen.

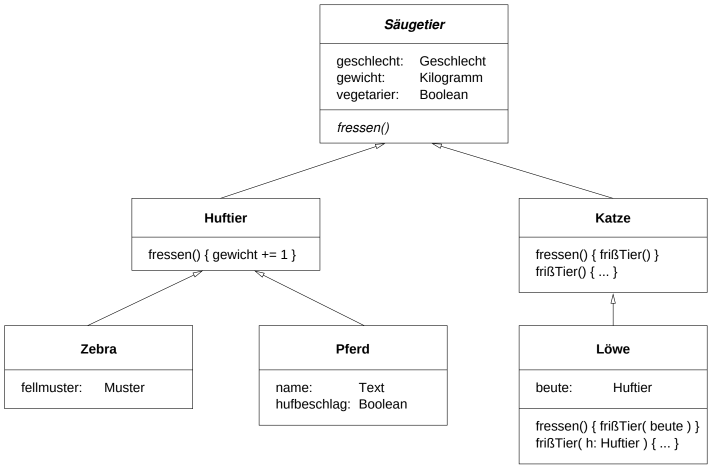


Um die Klasse Löwe in unsere Klassenhierarchie einzubinden, ist eine *Spezialisierung* von Säugetier der naheliegende Weg. Auch an dieser Stelle empfiehlt sich wieder ein mehrstufiges Vorgehen. Zunächst erstellen wir eine Klasse Katze, von der wir dann die Klasse Löwe direkt ableiten können.

Bemerkung: Oft ist es sinnvoll, Attributwerte für ganze Klassen einheitlich festzulegen. Das Attribut vegetarier z.B. könnte für die ganze Klasse Katze auf false gesetzt werden. Dadurch können Konstruktoren für derartige Klassen oft erheblich einfacher aufgebaut sein.

d) Um das Freßverhalten der einzelnen Tierarten darstellen zu können, bietet sich in unserem Modell natürlich die Methode fressen() an, die für alle Tiere deklariert ist. Da für die Katzen noch keine konkrete Methodendeklaration von fressen() angegeben ist, muß die Klasse Katze um eine entsprechende konkrete Methode ergänzt werden. Dazu wird die aus der Klasse Tier bzw. Säugetier geerbte abstrakte Methode fressen() implementiert, d.h. sie erhält einen Methodenrumpf.

Für die Klasse Löwe ist eine weitere Konkretisierung möglich, wenn wir davon ausgehen, daß Löwen bevorzugt Huftiere fressen. Damit können wir durch *Überschreiben* der in der Klasse Katze deklarierten Methode fressen() eine genauer auf Löwen zugeschnittene Methode fressen() erhalten. Entscheidend ist, daß die Methodensignatur dabei nicht verändert wird. Lediglich der Methodenrumpf wird überschrieben. Im Unterschied dazu wird die von der Oberklasse Katze geerbte Methode frißTier() in der Klasse Löwe *überladen*, indem eine gleichnamige, aber anders parametrisierte Methode deklariert wird (frißTier(h: Huftier)).



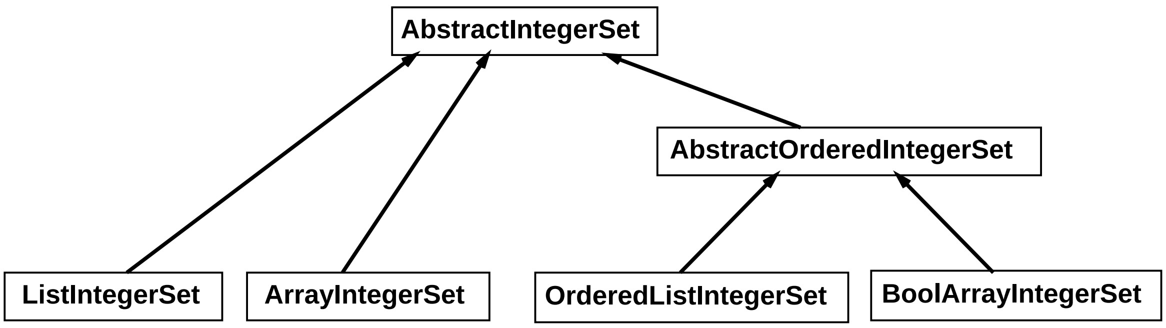
Die Methode fressen() ist für alle Tiere deklariert, d.h. jede Unterklasse von Tier enthält eine Methode mit dieser Signatur. Dank dem *Polymorphismus* kann daher mit einem Methodenaufruf der Form irgendeinTier.fressen() für jedes Tier (irgendeinTier: Tier) das für seine Art typische Freßverhalten dargestellt werden.

Bemerkung: Polymorphismus ist möglich, weil eine Oberklasse als Datentyp stellvertretend für ihre Unterklassen verwendet werden kann. In der Klasse Löwe könnte als beute z.B. auch jederzeit ein Zebra eingetragen werden, da es ja ebenfalls ein Huftier ist.

Aufgabe 2 Abstrakte Mengen

Vorbemerkung:

In den Aufgaben 2 bis 4 entsteht folgende Hierarchie von abstrakten und konkreten Klassen zur Darstellung von `Integer`–Mengen:



a) Neben den Konstruktoren lassen sich auch die folgenden beiden Methoden nur realisieren, wenn man die zugrundeliegende Datenstruktur kennt:

```
void insert(Integer i)
Enumeration elements()
```

Alle anderen Methoden lassen sich datenstrukturunabhängig über diesen Grundmethoden formulieren (vgl. Teilaufgabe b).

b) (i) datenstrukturabhängige, abstrakte Methoden:

```
// Einfuegen eines Elementes:
public abstract void insert(Integer i);

// Methode fuer Mengeniteration:
public abstract Enumeration elements();
```

(ii) datenstrukturunabhängige Methoden:

- Abfrage, ob Menge leer ist:

```
public boolean isEmpty() {
    return size() == 0;
}
```

- Abfrage, ob Element enthalten ist:

```
public boolean contains(Integer i) {
    for(Enumeration e = elements(); e.hasMoreElements(); )
        if (((Integer)e.nextElement()).intValue() == i.intValue())
            return true;
    return false;
}
```

- Abfrage nach Größe der Menge:

```
public int size() {
    int s = 0;
    for(Enumeration e = elements(); e.hasMoreElements(); e.nextElement())
        s++;
    return s;
}
```

- Abfrage nach Teilmengeneigenschaft:

```
public boolean isSubset(AbstractIntegerSet s) {
    for(Enumeration e = elements(); e.hasMoreElements(); )
        if (!s.contains((Integer) e.nextElement()))
            return false;
    return true;
}
```

- Operator für Gleichheit:

```
public boolean isEqual(AbstractIntegerSet s) {
    return isSubset(s) && s.isSubset(this);
}
```

- Ausgabefunktion:

```
public String toString() {
    StringBuffer buf = new StringBuffer();
    buf.append("{");
    for (Enumeration e = elements(); e.hasMoreElements();) {
        buf.append(e.nextElement().toString());
        if (e.hasMoreElements())
```

```
                buf.append(", ");
            }
            buf.append("}");
            return buf.toString();
        }
```

c) Zunächst eine Klasse `Node` für die Elemente der Liste:

```
public class Node {

// Datenstruktur:

    Object data;
    Node next;

// Konstruktor:

    Node (Object o, Node n) {
        data = o;
        next = n;
    }
}
```

Dann darauf aufbauend die Klasse `ListIntegerSet`:

```
import java.util.Enumeration;

public class ListIntegerSet extends AbstractIntegerSet {

// Datenstruktur:

    Node head;

// Konstruktor fuer eine leere Menge:

    public ListIntegerSet() {
        head = null;
    }

// Konstruktor, der die Kopie einer Menge liefert:

    public ListIntegerSet(AbstractIntegerSet s) {
        head = null;
        for (Enumeration e = s.elements(); e.hasMoreElements(); )
            head = new Node(e.nextElement(), head);
    }

// datenstrukturabhaengige sonstige Methoden:

    // Einfuegen eines Elementes:
    public void insert(Integer i) {
        if (!contains(i))
            head = new Node(i, head);
    }
```

```
    // Methode fuer Mengeniteration:
    public Enumeration elements() {
        return new ListIntegerEnumerator(this);
    }
}
```

Und schließlich die Klasse `ListIntegerEnumerator` für die Mengeniteration über `ListIntegerSet`:

```
class ListIntegerEnumerator implements Enumeration {
    private Node nextNode;

    ListIntegerEnumerator(ListIntegerSet s) {
        nextNode = s.head;
    }

    public boolean hasMoreElements() {
        return nextNode != null;
    }

    public Object nextElement() {
        Integer i = (Integer) nextNode.data;
        nextNode = nextNode.next;
        return i;
    }
}
```

Aufgabe 3 Mengendarstellung als Feld

(i) Datenstruktur:

```
private static final int DEFAULT_SIZE = 100;
int currentSize;
Integer array[];
```

(ii) verschiedene Konstruktoren für eine leere Menge:

- gewünschte Feldgröße wird übergeben:

```
public ArrayIntegerSet(int maxSize) {
    array = new Integer[maxSize];
    currentSize = 0;
}
```

- Feldgröße richtet sich nach dem Default–Wert `DEFAULT_SIZE`:

```
public ArrayIntegerSet() {
    this(DEFAULT_SIZE);
}
```

(iii) Konstruktor, der die Kopie einer Menge liefert (die Feldgröße wird dabei so gewählt, daß später auch weitere Elemente in der Menge noch Platz finden):

```
public ArrayIntegerSet(AbstractIntegerSet s) {
    currentSize = s.size();

    if (currentSize < DEFAULT_SIZE)
        array = new Integer[DEFAULT_SIZE];
    else
        array = new Integer[currentSize + DEFAULT_SIZE];

    int index = 0;
    for (Enumeration e = s.elements(); e.hasMoreElements(); )
        array[index++] = (Integer) e.nextElement();
}
```

(iv) datenstrukturabhängige sonstige Methoden:

- Einfügen eines Elementes:

```
public void insert(Integer i) {
    if (!contains(i) && currentSize < array.length)
        array[currentSize++] = i;
}
```

- Methode für Mengeniteration:

```
public Enumeration elements() {
    return new ArrayIntegerEnumerator(this);
}
```

- Überschreiben der Methode `size()` durch eine effizientere, datenstrukturabhängige Variante:

```
public int size() {
   return currentSize;
}
```

(v) Die zugehörige Mengeniteration:

```
class ArrayIntegerEnumerator implements Enumeration {
    private ArrayIntegerSet set;
    private int nextIndex;

    ArrayIntegerEnumerator(ArrayIntegerSet s) {
        set = s;
        nextIndex = 0;
    }

    public boolean hasMoreElements() {
        return nextIndex < set.currentSize;
    }

    public Object nextElement() {
        return set.array[nextIndex++];
    }
}
```

Aufgabe 4 Sortierte Mengendarstellungen

a) Die folgenden Methoden lassen sich auch unabhängig von der Datenstruktur effizienter realisieren, da man voraussetzen kann, daß die Mengeniteration die Elemente in aufsteigender Reihenfolge liefert:

- Die Abfrage, ob ein Element enthalten ist, bricht ab, wenn ein größeres gefunden worden ist:

```
public boolean contains(Integer i) {
    for (Enumeration e = elements(); e.hasMoreElements(); ) {
        int j = ((Integer)e.nextElement()).intValue();
        if (j == i.intValue())
            return true;
        if (j > i.intValue())
            return false;
    }
    return false;
}
```

- Bei der Abfrage nach der Teilmengeneigenschaft werden beide Mengen simultan nur einmal durchlaufen:

```
public boolean isSubset(AbstractOrderedIntegerSet s) {
    Enumeration se = s.elements();

    for (Enumeration e = elements(); e.hasMoreElements(); ) {
        int i1 = ((Integer)e.nextElement()).intValue();

        if (!se.hasMoreElements())
            return false;

        int i2;
        do {
            i2 = ((Integer)se.nextElement()).intValue();
        } while (i2 < i1 && se.hasMoreElements());

        if (i2 != i1)
            return false;
    }
    return true;
}
```

- Analog wird die Gleichheit durch eine simultane Schleife durch beide Mengen untersucht:

```
public boolean isEqual (AbstractOrderedIntegerSet s) {
    Enumeration se = s.elements();

    for (Enumeration e = elements(); e.hasMoreElements(); ) {
        if(!se.hasMoreElements() ||
                ((Integer)se.nextElement()).intValue() !=
                ((Integer)e.nextElement()).intValue())
            return false;
    }
    return !se.hasMoreElements();
}
```

b) (i) Die Klasse `OrderedListIntegerSet`:

Es werden hier nur die Methoden angegeben, die wesentlich von denen in der Klasse `ListIntegerSet` abweichen:

- Ein zusätzlicher copy–Konstruktor speziell für sortierte Mengen erlaubt einen schnellen Aufbau der sortierten Liste durch Anhängen der Elemente am Ende:

```
public OrderedListIntegerSet(AbstractOrderedIntegerSet s) {
    Node tail = null;
    head = null;
    for (Enumeration e = s.elements(); e.hasMoreElements(); ) {
        Node nextNode = new Node(e.nextElement(),null);
        if (tail == null) {
            head = nextNode;
            tail = head;
        }
        else {
            tail.next = nextNode;
            tail = tail.next;
        }
    }
}
```

- Die Methode `insert()` wird wesentlich komplizierter: Es muß erst die richtige Einfügestelle für das Element gefunden werden:

```
public void insert(Integer i) {
    int k = i.intValue();
    if (head == null || ((Integer)head.data).intValue() > k)
        head = new Node(i, head);
    else
    {
        Node currNode = head;
        while(currNode.next != null &&
               ((Integer)currNode.next.data).intValue() < k)
            currNode = currNode.next;
        if(currNode.next == null ||
           ((Integer)head.next.data).intValue() > k)
            currNode.next = new Node(i, currNode.next);
    }
}
```

(ii) Die Klasse `BoolArrayIntegerSet`:

- Die Datenstruktur enthält neben dem Boolschen Feld auch den kleinsten und größten zulässigen `Integer`–Wert:

```
private static final int DEFAULT_LOW = -100;
private static final int DEFAULT_HIGH = 100;

int low;
int high;
boolean array[];
```

- verschiedene Konstruktoren für eine leere Menge:

```
public BoolArrayIntegerSet(int low, int high) {
    if (low <= high)
    {
       this.low = low;
```

```
            this.high = high;
        }
        else
        {
            this.low = DEFAULT_LOW;
            this.high = DEFAULT_HIGH;
        }
        array = new boolean[this.high - this.low + 1];
        for (int index=0; index<array.length; index++)
            array[index] = false;
    }

    public BoolArrayIntegerSet(int range) {
        this(0, range-1);
    }

    public BoolArrayIntegerSet() {
        this(DEFAULT_LOW, DEFAULT_HIGH);
    }
```

- Der Konstruktor, der die Kopie einer Menge liefert, nimmt statt der Default–Werte für `low` und `high` gegebenenfalls den kleinsten bzw. größten tatsächlich vorkommenden Wert:

```
    public BoolArrayIntegerSet(AbstractIntegerSet s) {
        low = DEFAULT_LOW;
        high = DEFAULT_HIGH;
        for(Enumeration e = s.elements(); e.hasMoreElements(); ) {
            int i = ((Integer)e.nextElement()).intValue();
            if (i < low)
                low = i;
            else if (i > high)
                high = i;
        }
        array = new boolean[high - low + 1];
        for (int index=0; index<array.length; index++)
            array[index] = false;
        for(Enumeration e = s.elements(); e.hasMoreElements(); )
            insert((Integer) e.nextElement());
    }
```

- datenstrukturabhängige sonstige Methoden:
  - eine Hilfsmethode, die aus dem `Integer`–Wert den zugehörigen Index berechnet:

```
    private int indexOf(Integer i) {
        return i.intValue() - low;
    }
```

  - Einfügen eines Elementes:

```
    public void insert(Integer i) {
        int index = indexOf(i);

        if (index >= 0 && index < array.length)
            array[index] = true;
    }
```

  - auch die Methode `contains()` läßt sich mit dieser Datenstruktur besonders effizient realisieren:

```
public boolean contains(Integer i) {
    int index = indexOf(i);

    if (index >= 0 && index < array.length)
       return array[index];
    else
       return false;
}
```

- wie üblich die Methode zur Mengeniteration:

```
public Enumeration elements() {
    return new BoolArrayIntegerEnumerator(this);
}
```

- Die zugehörige Mengeniteration:

```
class BoolArrayIntegerEnumerator implements Enumeration {
    private BoolArrayIntegerSet set;
    private int nextIndex;

    BoolArrayIntegerEnumerator(BoolArrayIntegerSet s) {
        set = s;

        // nextIndex auf das erste Element setzen:
        nextIndex = 0;
        while (nextIndex < set.array.length && !set.array[nextIndex])
            nextIndex++;
    }

    public boolean hasMoreElements() {
        return nextIndex < set.array.length;
    }

    public Object nextElement() {
        // Resultat festhalten:
        Integer result = new Integer(set.low + nextIndex);

        // nextIndex weiterschalten:
        do
            nextIndex++;
        while (nextIndex<set.array.length && !set.array[nextIndex]);

        // Ergebnis ausliefern:
        return result;
    }
}
```