

Praktische Informatik 1

Generisches Programmieren

Thomas Röfer

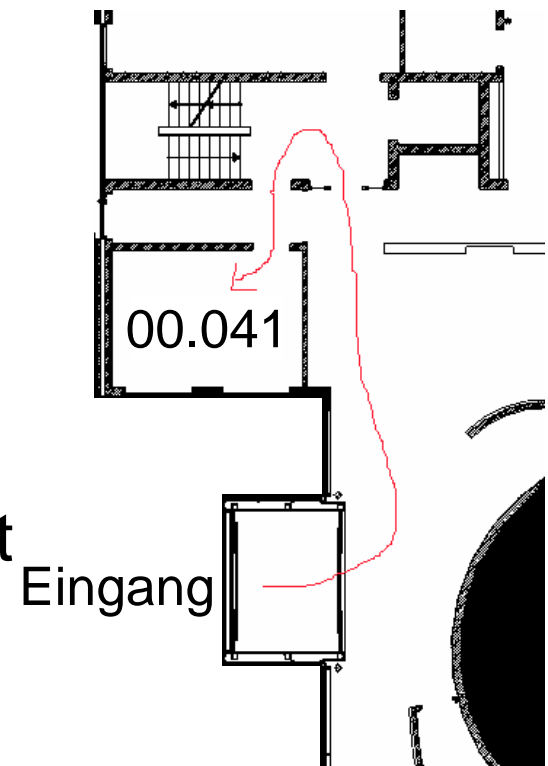
- Generische Klassen und Interfaces
- Generische Typen
- Typebounds
- Wildcard-Typen
- Übersetzung generischer Klassen
- Grenzen generischer Typen
- Polymorphe Methoden

Abgabe von Übungsblatt 10 und 11

- Abgabe Übungsblatt 10
 - Am Mittwoch, 30. Januar 2008
 - Wer schon abgegeben hat, kann nochmal abgeben (muss aber nicht!)
 - Zweite Abgabe als solche kennzeichnen
- Abgabe Übungsblatt 11
 - Am Mittwoch, 6. Februar 2008
- Übungsblatt 12 entfällt

Fachgespräche

- Überprüfung der Individualität der Leistung
- In Dreiergruppen (Übungsgruppen)
- Zwei Teile
 - Kleine Programmieraufgabe
 - Fragen nach Begriffen und Konzepten
- Hartes Kriterium
 - Wer die Programmieraufgabe nicht löst, bekommt keinen Schein
- Ort
 - Cartesium. Raum 00.041



Fachgespräche – Programmieraufgabe

- Vornote 2.7 - 4.0
 - Iterativ mit Array
 - Z.B.: Schreibe eine *iterative* Methode, die alle Elemente einer übergebenen Reihung quadriert
- Vornote 1.7 - 2.3
 - Rekursiv mit Array
 - Z.B.: Schreibe eine *rekursive* Methode, die alle Elemente einer übergebenen Reihung quadriert
- Vornote 1.0 - 1.3
 - Rekursiv mit Liste
 - Z.B.: Gegeben: `class Elem {int value; Elem next;}`
Schreibe eine *rekursive* Methode, die alle Elemente einer übergebenen Liste aus solchen Elementen quadriert

```
static void square(Elem e) {  
    if (e != null) {  
        e.value *= e.value;  
        square(e.next);  
    }  
}
```

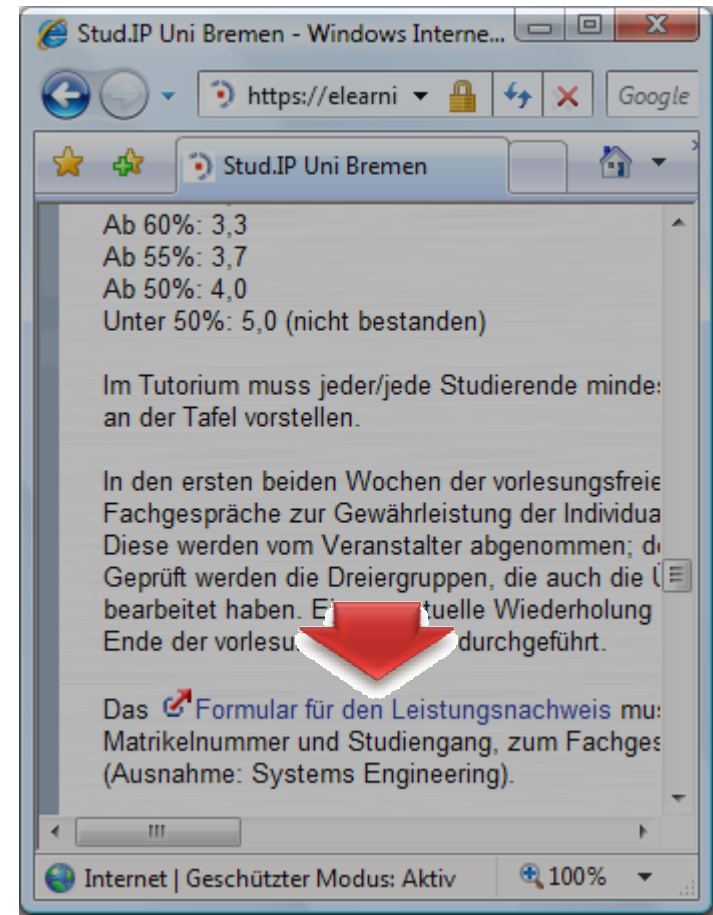

Fachgespräche

Fragen nach Begriffen und Konzepten

- Keine Java-Syntax
- Schwerpunktmäßig aus Vorlesungen 2-6
 - Z.B. Von-Neumann-Architektur, EBNF usw.
 - Fragen nach inhaltlich für das Programmieren relevanten Themen
 - Nicht: „Wann wurde John von Neumann geboren?“
- „Was ist der Unterschied“-Fragen, z.B.
 - Was ist der Unterschied zwischen Vorrang und Assoziativität von Operatoren?
 - Was ist der Unterschied zwischen Überladen und Überschreiben?

Fachgespräche

- Bewertung
 - Ausgangspunkt: Vornote aus dem Übungsbetrieb
 - Aufwertung: Maximal 1/3-Note
 - Abwertung: beliebig
- Wiederholung
 - Gegen Ende März
- SBLN mitbringen!
 - Außer Systems Engineering
 - Haben eigene Scheine
 - Von PI-1-StudIP-Seite laden, zu Ende ausfüllen und ausdrucken
 - Name, Matrikel-Nr., Studiengang



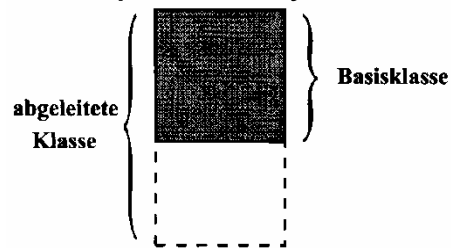
Rückblick „Vererbung“

Pakete

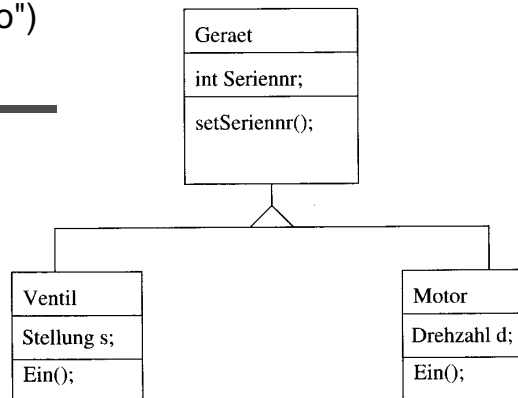
`java.lang.System.out.println("Hallo")`

Paket Paket Klasse Attribut Methode

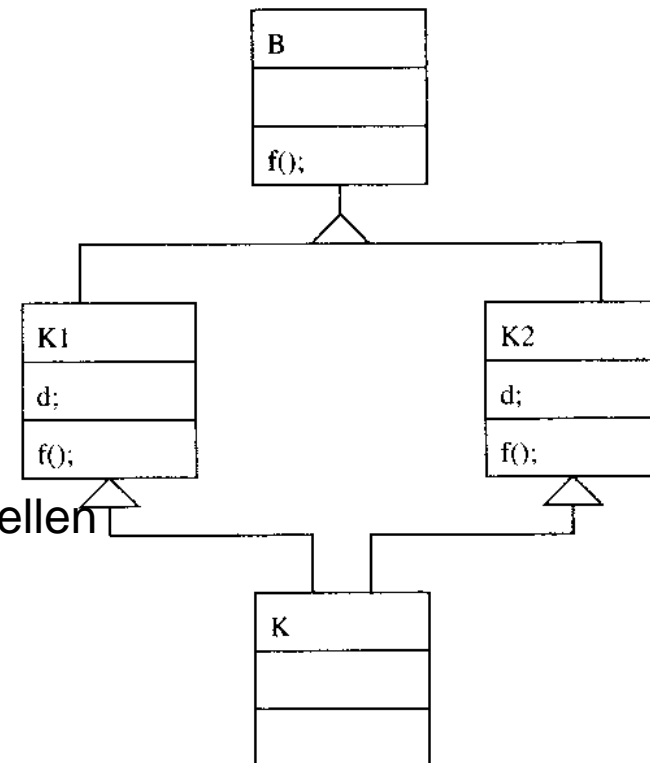
Speicherlayout



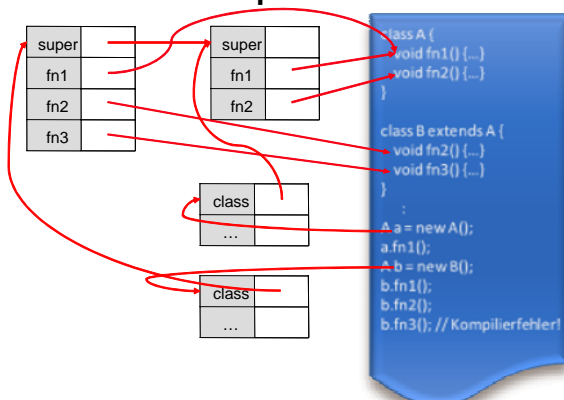
Vererbung



Mehrfaches Erben



Frühes/spätes Binden Abstrakte Klassen / Schnittstellen



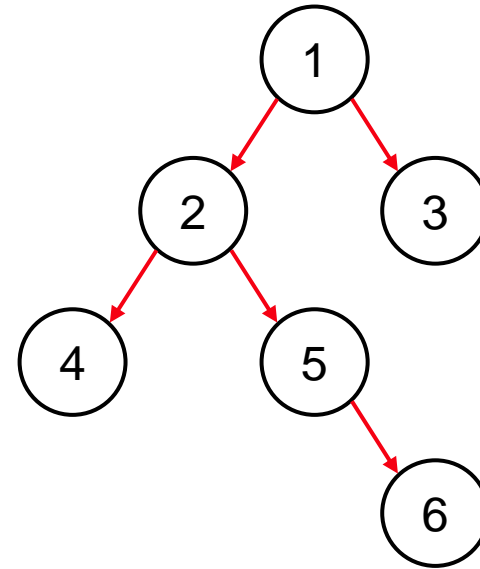
```

abstract class Geraet {
    int seriennummer;
    abstract void ein();
}

interface Printable {
    void print();
}
    
```

Motivation

```
class Node {  
    private Object info;  
    private Node left, right;  
  
    Node(Object i) {info = i;}  
    Node(Object i, Node l, Node r) {  
        info = i; left = l; right = r;  
    }  
    Object getInfo() {return info;}  
    Node getLeft() {return left;}  
    Node getRight() {return right;}  
    void setInfo(Object i) {info = i;}  
}
```



```
Node n1 = new Node("foo");  
Node n2 = new Node(3.14);  
Node n0 = new Node(23, n1, n2);  
  
String s = (String) n1.getInfo();
```

- Problem: Zugriff nicht Typ-sicher!

Generische Klassen

- Generische Klassen haben *Typvariablen*, die im Klassenrumpf verwendet werden können
 - *class Node<T> { T info; ... }*
- Typvariablen können (fast) wie ein normaler Typ benutzt werden
- Mehrere Typvariablen sind

```
class Pair<T, U> {  
    private T first;  
    private U second;  
    Pair(T t, U u) {first = t; second = u;}  
    T getFirst() {return first;}  
    U getSecond() {return second;}  
}
```

```
class Node<T> {  
    private T info;  
    private Node<T> left, right;  
  
    Node(T i) {info = i;}  
    Node(T i, Node<T> l, Node<T> r) {  
        info = i; left = l; right = r;  
    }  
    T getInfo() {return info;}  
    Node<T> getLeft() {return left;}  
    Node<T> getRight() {return right;}  
    void setInfo(T i) {info = i;}  
}
```


Generische Schnittstellen

```
interface Taggable<T> {  
    void setTag(T tag);  
    T getTag();  
}
```

```
abstract class B<T>  
    implements Taggable<T> {  
    :  
}
```

```
class A implements Taggable<String> {  
    private String tag;  
  
    public void setTag(String t) {  
        tag = t;  
    }  
  
    public String getTag() {  
        return tag;  
    }  
}
```

Generische Typen

- Eine *generische Klasse* oder *generische Schnittstelle* ist eine Klassendefinition, in der unbekannte Typen durch Typvariablen vertreten sind
 - *class Node<T> { T info ...*
- Ein *generischer Typ* ist eine Typangabe, in der eine generische Klasse mit einem konkreten Typargument versehen wird
 - *Node<Integer> n = new Node<Integer>(23);*
 - *Pair<Integer, String> p = new Pair<Integer, String>(28359, "Bremen");*

Beispiel – Die Klasse *Set<T>*

- *Set<T>*
 - Eine Menge von Objekten des Typs *T*
- *insert(T t)*
 - Einfügen eines neuen Elements in die Menge
- *contains(T t)*
 - Testen, ob ein bestimmtes Element in der Menge enthalten ist
- *intersect(Set<T> s), union(Set<T> s), diff(Set<T> s)*
 - Mengenschnitt, -vereinigung und -differenz
 - $A \cap B$, $A \cup B$, $A \setminus B$
- *toString()*
 - Umwandlung der Menge in eine Zeichenkette
- Implementierung der Schnittstelle *Iterable*, um *for (:)* zu unterstützen

Beispiel – Die Klasse *Set<T>*

```
import java.util.Iterator;

public class Set<T>
    implements Iterable<T> {
    protected Element<T> first = null;

    protected static class Element<T> {
        public T element;
        public Element<T> next;

        public Element(T e, Element<T> n) {
            element = e;
            next = n;
        }
    }
}
```

```
public void insert(T element) {
    if (!contains(element)) {
        first = new Element<T>(element,
                                first);
    }
}

public boolean contains(T element) {
    for (T t : this) {
        if (element.equals(t)) {
            return true;
        }
    }
    return false;
}
```

Beispiel – Die Klasse *Set<T>*

```
public Set<T> intersect(Set<T> other) {  
    Set<T> result = new Set<T>();  
    for (T t : this) {  
        if (other.contains(t)) {  
            result.insert(t);  
        }  
    }  
    return result;  
}
```

```
public Set<T> union(Set<T> other) {  
    Set<T> result = new Set<T>();  
    for (T t : this) {  
        result.insert(t);  
    }  
}
```

```
for (T t : other) {  
    result.insert(t);  
}  
return result;  
}
```

```
public Set<T> diff(Set<T> other) {  
    Set<T> result = new Set<T>();  
    for (T t : this) {  
        if (!other.contains(t)) {  
            result.insert(t);  
        }  
    }  
    return result;  
}
```

Beispiel – Set<T>

```
public Iterator<T> iterator() {  
    return new Iterator<T>() {  
        private Element<T> actual = first;  
  
        public boolean hasNext() {  
            return actual != null;  
        }  
  
        public T next() {  
            T result = actual.element;  
            actual = actual.next;  
            return result;  
        }  
  
        public void remove() {}  
    };  
}
```

```
public String toString() {  
    String s = "{";  
    for (T t : this) {  
        s += s.equals("{}") ? "" : ", ";  
        s += t.toString();  
    }  
    return s + "}";  
}
```

```
Set<Integer> a = new Set<Integer>();  
a.insert(1); a.insert(2); a.insert(1);  
a.toString()  
"{2, 1}" (String)  
Set<Integer> b = new Set<Integer>();  
b.insert(2); b.insert(3);  
a.intersect(b).toString()  
"{2}" (String)  
a.union(b).toString()  
"{3, 1, 2}" (String)  
a.diff(b).toString()  
"{1}" (String)
```

Typebounds

- Manchmal sollen nicht alle Typen für die Belegung der Typvariablen zulässig sein, z.B. wenn es Anforderungen an die Typargumente gibt
- Ein *Typebound* bedeutet „ist kompatibel zu“
- *Typebounds* können Klassen und auch Interfaces sein
 - Auch bei Interfaces wird hier *extends* verwendet
- Es kann mehrere *Typebounds* pro Typvariable geben
 - Sie werden durch & aufgezählt werden, z.B.
`class Node<T extends A & B & C> { ...`
 - Erster *Typebound* kann eine Klasse oder ein Interface sein, weitere können nur Interfaces sein
- Ein *Typebound* legt Mindestanforderungen für eine Typvariable fest
- Dadurch wird auch definiert, was man von diesem Typ an Funktionalität erwarten kann
- *Typebounds* können wiederum die Typvariable enthalten
 - `class Node<T extends Comparable<T>> { ...`

Typebounds – Beispiel

```
class SortedSet<T extends Comparable<T>> extends Set<T> {  
    public void insert(T element) {  
        if (!contains(element)) {  
            if (first == null || element.compareTo(first.element) <= 0) {  
                first = new Element<T>(element, first);  
            } else {  
                Element<T> e = first;  
                while (e.next != null && element.compareTo(e.next.element) > 0) {  
                    e = e.next;  
                }  
                e.next = new Element<T>(element, e.next);  
            }  
        }  
    }  
}
```

Invarianz generischer Typen

- Fragestellung
 - Jede generische Klasse erzeugt viele generische Typen
 - *Node<T>* erzeugt *Node<Number>*, *Node<Integer>*, *Node<Double>*...
 - Wie stehen die von einer generischen Klasse erzeugten Typen zueinander?
- Vererbung generischer Typen
 - Eine Ableitungsbeziehung zwischen Typargumenten überträgt sich nicht auf die generischen Typen
 - *class Integer extends Number { ...*
 - *Number n = new Integer(23); // ok*
 - *Node<Number> nn = new Node<Integer>(23); // Fehler!*
 - Dies wird als *Invarianz* bezeichnet

Bivarianz generischer Typen

- *Wildcard (Joker)* bei Typangaben
- Generische Typen können unbestimmte Typargumente nennen
 - *Node<?> nx;*
 - *nx = new Node<String>("foo");*
 - *nx = new Node<Integer>(1);*
 - *nx = new Node<Double>(3.14);*
- Kein Zugriff auf Methoden oder Attribute, die Typargument verwenden
 - *double d = nx.getInfo(); // Fehler*
 - *nx.setInfo(2.72); // Fehler*

```
static int count(Node<?> n) {  
    if (n == null) {  
        return 0;  
    } else {  
        return 1  
            + count(n.getLeft())  
            + count(n.getRight());  
    }  
}
```

Covarianz generischer Typen

- Zu einem Wildcard-Typ mit Typargument ? sind alle generischen Typen der betreffenden generischen Klasse kompatibel
- Manchmal möchte man dies durch einen sog. *Upper-Typebound* einschränken (der *allgemeinste* erlaubte Typparameter)
 - *Node<? extends Number> nb;*
 - *nb = new Node<Integer>(23);*
 - *nb = new Node<Object>(new Object()); // Fehler!*

```
static double sum(Node<? extends Number> n) {  
    if (n == null) {  
        return 0;  
    } else {  
        return n.getInfo().doubleValue()  
            + sum(n.getLeft()) + sum(n.getRight());  
    }  
}
```