

Tutorium Programmieren

Tut Nr.9: Vererbung

Michael Friedrich | 07. / 09.01.2014

INSTITUT FÜR THEORETISCHE INFORMATIK



1 Übungsblatt4

2 Vererbung

3 Rekursion

4 Generics

- Nutzt die öffentlichen Test als Hilfe

- Nutzt die öffentlichen Test als Hilfe
→ Bei Abschlussaufgaben Abgabekriterium!
ABER kein hardcoden!!! - Zählt als Betrugsversuch

Wird genutzt, um Gleiches zu „gruppieren“

Beispiel

```
class Person {
    String name;
    int age;
}
class Student extends Person {
    int matriculation;
    Tutor tutor;
}
class Tutor extends Student {
    //hat auch name, age, matriculation
    String name; //anderes name als Person.name!
    Student[] students;
}
```

ACHTUNG Überschreiben von Attributen nicht möglich



- Vererbung wird gerne falsch gemacht! \Rightarrow immer IST-Beziehung

SO NICHT

```
class Point {  
    int x, y;  
}  
class Linie extends Point {  
    int x2, y2; // huh?  
}
```

- Vererbung wird gerne falsch gemacht! \Rightarrow immer IST-Beziehung

SO NICHT

```
class Point {  
    int x, y;  
}  
class Linie extends Point {  
    int x2, y2; // huh?  
}
```

Sondern?

- Vererbung wird gerne falsch gemacht! \Rightarrow immer IST-Beziehung

SO NICHT

```
class Point {  
    int x, y;  
}  
class Linie extends Point {  
    int x2, y2; // huh?  
}
```

Sondern?

```
class Linie {  
    Point p1, p2; //HAT-Beziehung  
}
```


- habt ihr alle schon implizit genutzt, Beispiele?

- habt ihr alle schon implizit genutzt, Beispiele?
 - `toString`, `equals(Object o)`, ...

- habt ihr alle schon implizit genutzt, Beispiele?
 - `toString`, `equals(Object o)`, ...
- diese Methoden erbt JEDE Klasse von `java.lang.Object`

- habt ihr alle schon implizit genutzt, Beispiele?
 - `toString`, `equals(Object o)`, ...
- diese Methoden erbt JEDE Klasse von `java.lang.Object`
Habt ihr schon überschrieben (daher `@Override`)

- habt ihr alle schon implizit genutzt, Beispiele?
 - `toString`, `equals(Object o)`, ...
- diese Methoden erbt JEDE Klasse von `java.lang.Object`
Habt ihr schon überschrieben (daher `@Override`)
- Konstruktoren werden mitvererbt und **müssen immer** in der Subklasse implementiert werden
→ Zugriff auf den Oberklassen Methoden (also auch Konstruktor) über `super()`

- habt ihr alle schon implizit genutzt, Beispiele?
 - `toString`, `equals(Object o)`, ...
- diese Methoden erbt JEDE Klasse von `java.lang.Object`
Habt ihr schon überschrieben (daher `@Override`)
- Konstruktoren werden mitvererbt und **müssen immer** in der Subklasse implementiert werden
→ Zugriff auf den Oberklassen Methoden (also auch Konstruktor) über `super()`

super

```
public Student(name, age, matriculation) {  
    super(name, age);  
    this.matriculation = matriculation;  
}
```

- Eine Unterklasse kann alles, was die Oberklasse kann
→ exakt gleiche Methoden, auch wenn eventuell überschrieben

Beispiel:

```
Student michael = new Student();  
Person mike = michael; //Jeder Student IST also gleichzeitig eine  
    Person  
Person anna = new Student();  
Student anne = new Person(); ⚡ NEIN!
```

Abstrakte (**abstract**) Klassen

- Klassen: Nicht instantiierbar
- Methoden: MÜSSEN in Subklasse instantiiert werden
 - nur in abstrakten Klassen erlaubt

Abstrakte (**abstract**) Klassen

- Klassen: Nicht instantiierbar
- Methoden: MÜSSEN in Subklasse instantiiert werden
 - nur in abstrakten Klassen erlaubt

```
abstract class mammal {  
    //protected nur in erbenden Klassen sichtbar  
    protected String s = "schnaufen";  
  
    abstract public void breathe();  
}  
  
class human extends mammal {  
  
    public void breathe() {  
        System.out.print(s);  
    }  
}
```

Interfaces - implements

- ähnlich den abstrakten Klassen aber keine Attribute

Interfaces - implements

- ähnlich den abstrakten Klassen aber keine Attribute
- definieren Schnittstelle
 - also anderer Sinn als abstrakte Klassen!

- ähnlich den abstrakten Klassen aber keine Attribute
- definieren Schnittstelle
 - also anderer Sinn als abstrakte Klassen!
 - definieren die Methoden, die eine Klasse implementieren **MUSS**

```
public interface Comparable {  
    boolean equals();  
    int compareTo();  
}  
  
public Letter implements Comparable {  
    char c;  
  
    public boolean equals(Letter c) {  
        return this.c==c;  
    }  
}
```

- ähnlich den abstrakten Klassen aber keine Attribute
- definieren Schnittstelle
 - also anderer Sinn als abstrakte Klassen!
 - definieren die Methoden, die eine Klasse implementieren **MUSS**

```
public interface Comparable {  
    boolean equals();  
    int compareTo();  
}  
  
public Letter implements Comparable {  
    char c;  
  
    public boolean equals(Letter c) {  
        return this.c==c;  
    }  
}
```

Methoden automatisch **abstract** und **public**

- Eine Klasse kann beliebig viele Interfaces implementieren

- Eine Klasse kann beliebig viele Interfaces implementieren
- Beziehung zwischen Interfaces über `extends` möglich

- Eine Klasse kann beliebig viele Interfaces implementieren
- Beziehung zwischen Interfaces über `extends` möglich

Tut Aufgabe zu Vererbung/Polymorphismus

Bedeutung: Selbstaufruf einer Funktion

Bedeutung: Selbstaufzuruf einer Funktion

```
public class Fibs {  
    public static int fib(final int n) {  
        if (n <= 1) {  
            return 1;  
        }  
        return fib(n - 1) + fib(n - 2);  
    }  
}
```

Bedeutung: Selbstaufzuruf einer Funktion

```
public class Fibs {  
    public static int fib(final int n) {  
        if (n <= 1) {  
            return 1;  
        }  
        return fib(n - 1) + fib(n - 2);  
    }  
}
```

- Was passiert bei dem Aufruf von `fib(4)`?

Bedeutung: Selbstaufzuruf einer Funktion

```
public class Fibs {  
    public static int fib(final int n) {  
        if (n <= 1) {  
            return 1;  
        }  
        return fib(n - 1) + fib(n - 2);  
    }  
}
```

- Was passiert bei dem Aufruf von `fib(4)`?
- Man braucht immer einen **terminierenden** Fall!

Wie schon letztes Blatt gemerkt, ein Tool, um Gleiches zu komprimieren
→ Listen sind identisch, bis auf ihren Inhalt

Wie schon letztes Blatt gemerkt, ein Tool, um Gleiches zu komprimieren
→ Listen sind identisch, bis auf ihren Inhalt

```
public class SinglyLinkedList<E> {  
    private Node<E> head;  
  
    public void add(E object) {  
        Node<E> newNode = new Node<E>(object);  
        newNode.setNext(head);  
        head = newNode;  
    }  
}
```

Wie schon letztes Blatt gemerkt, ein Tool, um Gleiches zu komprimieren
→ Listen sind identisch, bis auf ihren Inhalt

```
public class SinglyLinkedList<E> {  
    private Node<E> head;  
  
    public void add(E object) {  
        Node<E> newNode = new Node<E>(object);  
        newNode.setNext(head);  
        head = newNode;  
    }  
}
```

„E“ ist dann durch euren gewünschter Datentyp ersetzbar, zB
`new SinglyLinkedList<Article>`

Wie schon letztes Blatt gemerkt, ein Tool, um Gleiches zu komprimieren
→ Listen sind identisch, bis auf ihren Inhalt

```
public class SinglyLinkedList<E> {  
    private Node<E> head;  
  
    public void add(E object) {  
        Node<E> newNode = new Node<E>(object);  
        newNode.setNext(head);  
        head = newNode;  
    }  
}
```

„E“ ist dann durch euren gewünschter Datentyp ersetzbar, zB

```
new SinglyLinkedList<Article>
```

Aufgabe: restliche Liste mit Generics implementieren