Informatik I WS 07/08

Tutorium 24

24.01.08

Bastian Molkenthin

E-Mail: infotut@sunshine2k.de

Web: http://infotut.sunshine2k.de

Universität Karlsruhe (TH)
Forschungsuniversität · gegründet 1825

OOP - Vererbung

- Vererbung wird genutzt, um neue Klassen unter Verwendung bereits bestehender aufgebaut.

- So kann z.B. eine Klasse *Vehicle* für die Klassen *Cabrio* oder *Truck* als Grundlage dienen

- Vorteil: Allgemeine Attribute und Methoden wie *Hersteller, fahren, bremsen* oder *blinken* müssen dann nur einmal implementiert werden.

- Spezielle Attribute und Methoden wie *Laderaum oder Verdeck öffnen* können einfach ergänzt werden.

- Wenn Klasse A von Klasse B erbt, bekommt Klasse A alle Eigenschaften von Klasse B.

- Klasse A kann dann weiter spezialisiert werden.

- Java Schlüsselwort: extends

OOP - Vererbung

- Die Klassen Vehicle, Cabrio und Truck als UML-Diagramm:

Vehicle

color : String
hp : int
drivenDistance : int

drive()
brake()
startMotor()

Cabrio

topOpen : boolean

openTop()
closeTop()

Truck

loading : double

load()
unload()

OOP - Vererbung

```
class Vehicle {

    public String color;
    public int hp;
    public int drivenDistance;

    public Vehicle() {
        drivenDistance = hp = 0;
        color = "undefined";
    }

    public Vehicle(String color, int hp) {
        this.color = color;
        this.hp = hp;
        drivenDistance = 0;
    }

    public void drive(int distance) {
        drivenDistance += distance;
    }
}
```

```
class Cabrio extends Vehicle {

    public boolean topOpen;

    public Cabrio(String color, int hp) {
        super(color, hp);
        topOpen = false;
    }

    public void openTop() {
        topOpen = true;
}

    public void closeTop() {
        topOpen = false;
    }
}
```

```
class Truck extends Vehicle {
        double loading = 0;

        public void load() { loading = 1; }
        public void unload() { loading = 0; }
}
```

OOP - Vererbung

- Der Nutzen der Vererbung liegt darin, dass eine Unterklasse mit einer Oberklasse kompatibel ist.
  - jedes Programm, das in der Lage ist, mit Objekten der Oberklasse zu arbeiten, kann auch mit Objekten der Unterklasse arbeiten

- Die Kompatibilität gilt, weil jedes Objekt einer Unterklasse zugleich auch ein Objekt der Oberklasse ist.
  - "ist ein"-Beziehung (englisch : "is a")
  - Objekt der Unterklasse "ist ein" Objekt der Oberklasse
  - in umgekehrter Richtung gilt die "ist ein"-Beziehung nicht

- Einer Objektvariablen der Oberklasse kann ein Objekt der Unterklasse zugewiesen werden.

Aufgabe Vererbung

Welche Aufrufe sind gültig?

Vehicle myVehicle = new Cabrio();	:-)
myVehicle.drive(20);	:-)
if (myVehicle.topOpen) myVehicle.closeTop();	:-(
Vehicle myVehicle2 = new Truck();	:-)
myVehicle2.drive(20);	:-)
myVehicle2.load();	:-(
Truck myTruck = new Truck();	:-)
myTruck.load();	:-)
Vehicle myVehicle3 = new Vehicle();	:-)
Truck myTruck2 = (Truck)myVehicle3;	:-(

OOP – Abstrakte Klasse

- Häufig ist es sinnvoll zu verhindern, dass die "Mutterklasse" selbst zur Erzeugung eines Objekts herangezogen werden kann

- Eine solche Klasse wird mit dem Schlüsselwort abstract gekennzeichnet

- Methoden abstrakter Klassen können ebenfalls abstrakt sein – diese dürfen dann keinen Code enthalten und müssen in den (nicht abstrakten) Unterklassen überschrieben werden

- Dies dient dazu bestimmte Methoden zu erzwingen, deren konkrete Implementierung jedoch bei verschiedenen Unterklassen unterschiedlich ist

```
abstract class Building {
    int size;
}
...
Building b = new Building();
```

<------ geht nicht! Building ist abstract!

OOP – Abstrakte Klasse (2)

```
abstract class Building {
    int size;
    abstract String getType();
}
class Disco extends Building {

}
```

<-------- geht nicht! Muss getType() implementieren!

Aber so…

```
abstract class Building {
    int size;
    abstract String getType();
}

class Disco extends Building {
    String getType() { return "Disco!"; }
}

public class BuildingDemo {
    public static void main(String[] args) {
        Disco d = new Disco();
        Out.println(d.getType());
    }
}
```

OOP – Interface

- Eine abstrakte Klasse, welche nur abstrakte Methoden und als Attribute nur Konstanten enthält entspricht einem Interface
(Schlüsselwort interface)

```
interface Iface {
   double PI = 3.14;
   double calc(double d);
}
```

- Die Methode calc ist hier automatisch public abstract
- PI ist automatisch public static final
- Es ist sichergestellt, dass alle Objekte, die vom Interface erben, über die vordefinierten Schnittstellen (Interfaces) verfügen
- Eine Klasse kann von mehreren Interfaces erben (aber nur von einer Klasse!)
- Java Schlüsselwort: implements

Interface Beispiel

```
interface Buyable
{
     double price();
}

interface Geometry
{
     double getVolume();
}

public class Chocolate implements Buyable, Geometry
{
    private area;
    public double price() {
        return 0.69;
    }
    public double getVolume() {
        return area * 1.5;
    }
}

...
Buyable schoki = new Chocolate();
Out.println(schoki.price());
```

Sichtbarkeit

Wo liegt der Unterschied zwischen public, protected und private?

sichtbar	**public**	**protected**	**default**	**private**
eigener Klasse	✓	✓	✓	✓
eigenem Paket	✓	✓	✓	
Kindklassen	✓	✓	✓	
Fremden Klassen	✓			

Klassen- und Objektattribute

- Klassenattribute existieren einmal pro Klasse.
  Zugriff über Klassennamen.
- Objektattribute existieren getrennt für jedes erzeugtes Objekt.
  Zugriff über Objektvariable.

- Klassenattribute werden mit Schlüsselwort static gekennzeichnet.
- Klassenattribute könne verwendet werden ohne ein Objekt der Klasse zu erzeugen.

```
class MyStatic {
    static int counter = 0;
    int value = 2;
}

public class statictest {
    public static void main(String[] args) {
        Out.println(MyStatic.counter);
        MyStatic.counter = 1;
        MyStatic c1 = new MyStatic();
        Out.println(c1.counter);
        Out.println(++c1.value);
        MyStatic c2 = new MyStatic();
        Out.println(c2.value);
        //Out.println("Value is: " + MyStatic.value);

    }
}
```

<------- Erzeugt Fehler!

Ausgabe:
0
1
3
2

Klassenmethoden

Analog zu Klassenattributen gibt es auch Klassenmethoden.

- Objektmethoden können auf Klassenattribute zugreifen, aber nicht umgekehrt.

Wozu Klassenmethoden?

Alle Methoden sind innerhalb einer Klasse, doch nicht jede muss auf Attribute der Klasse zugreifen, z.B *Character.isDigit()* oder *Math.random()*.

Somit nicht notwendig, Objekt der Klasse zu erstellen, um Klassenmethoden zu benutzen.

Überladen

Eine überladene Methode ist eine Funktion, die den gleichen Namen wie eine andere Funktion hat, sich jedoch in der Parameterliste unterscheidet.

```
static void print(int x) {  Out.println("Integer: " + x);  }
static void print(float f) {  Out.println("float: " + f);  }

public static void main(String[] args)  {
        print(5);
        print(5f);
}
```

Ausgabe: Integer: 5
float: 5.0

Unterschied in Parameterliste bedeutet, dass sich die Parameter in *Anzahl und/oder Datentyp* unterscheiden müssen. Der Rückgabewert gehört nicht zur Parameterliste!

```
static int foo(int  x) {  return 5;  }
static boolean foo(int  y) {  return false;  }
```

Geht nicht!!

Überschreiben

Überschreiben bedeutet, dass eine Klasse eine von einer Oberklasse geerbte Methode mit der gleichen Parameterliste neu deklariert.

```
class Disco {
    boolean checkAge(int age) {
        return (age >= 18);
    }
}

class TeenDisco extends Disco {
    boolean checkAge(int age) {
        return (age >= 13);
    }
}

public class DiscoDemo {
    public static void main(String[] args) {
        Disco coolClub = new Disco();
        Out.println(coolClub.checkAge(16));
        TeenDisco lousyclub = new TeenDisco();
        Out.println(lousyclub.checkAge(16));
    }
}
```

Ausgabe: false
true

Nochmal: Konstruktoren

- Standardkonstruktor ist immer vorhanden, auch wenn er nicht explizit angegeben ist.

```
class A { }
...
A a = new A();
```

:-)

- Konstruktoren können überladen werden, dann muss aber der Standardkonstruktor explizit angegeben werden falls benötigt!

```
class A {
    A(String s) { ...}
    A(int i) { ... }
}
...
A a = new A();
```

:-(

Fehler -> Standardkonstruktor nicht vorhanden!

- Konstruktoren werden nicht vererbt!

```
class A {
    A(String s) { ...}
}
class B extends A { }
...
B b = new B("hallo");
```

:-(

Fehler -> Konstruktor B(String) nicht vorhanden!

Konstruktoren (2)

- Konstruktor einer Unterklasse ruft als erstes immer den Konstruktor der Oberklasse auf! Dies ist auch selbst möglich mit super(), muss aber 1. Anweisung im Konstruktor sein!

```
class A {
    A() { Out.println("Hello A"); }
}
class B extends A {
    B() { Out.println("Hello B");}  ⟺  B() { super(); Out.println("Hello B"); }
}
...
B b = new B();
```

Ausgabe: Hello A
Hello B

- Vorsicht beim Vererben und gleichzeitigem Überladen von Konstruktoren:

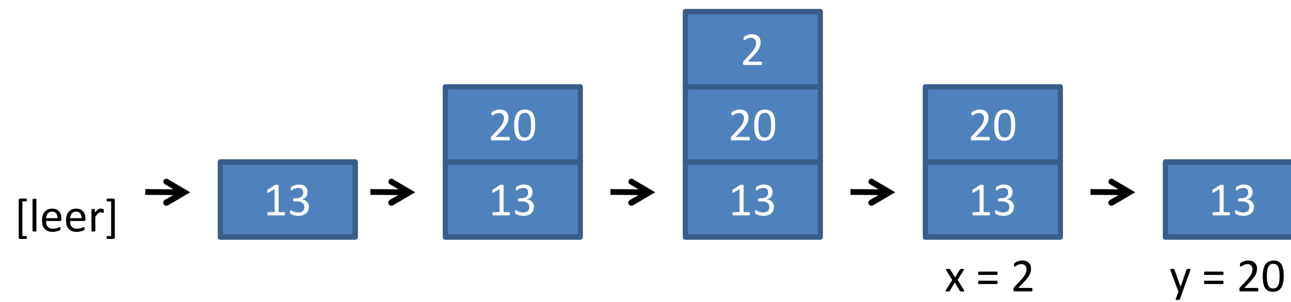
```
class A {
    A(String s) {
        Out.println("Hello A");
    }
}
class B extends A {
    B() { Out.println("Hello B"); }
}
...
B b = new B("hallo");
```

:-(Entweder Standardkonstruktor in A einfügen oder im Konstruktor von B explizit super("") als erstes aufrufen!

Stapel (Stack)

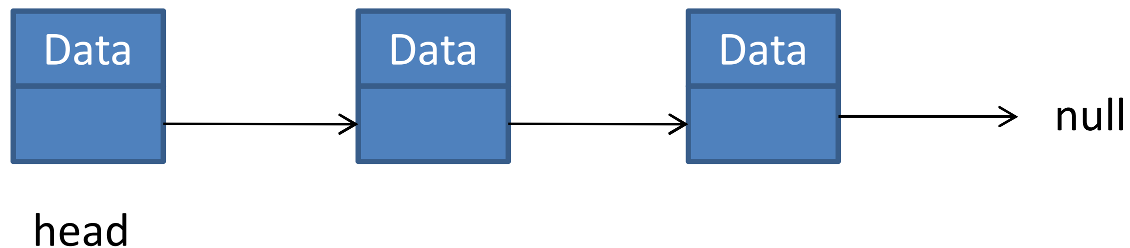
- Datenstruktur, die Zugriffsprinzip auf Elementmenge vorgibt.
- Zugriffsprinzip ist LIFO = Last in First out
  - Das zuletzt eingefügte Element wird als erstes wieder entfernt, d.h. man auch auch nur Zugriff auf dieses.
- Operationen:
  - push: Legt ein Element auf den Stapel
  - pop: Nimmt das oberste Element vom Stapel.
- Beispiel:

```
push(13)
push(20)
push(2);
x = pop();
y = pop();
...
```



Verkettete Liste

- Datenstruktur zur dynamischen Speicherung von beliebig vielen Objekten.
- Jedes Listenelement enthält einen Verwies auf das nächste Element.



- Vorteile / Nachteile:
  - \+ Kann beliebig wachsen und schrumpfen
  - \+ Einfügen von Elementen an beliebiger Stelle ist einfach
  - \- Höherer Speicherbedarf im Vergleich zu Arrays
  - \- Zugriff auf einzelne Elemente nur durch Durchlaufen der Liste

Beispiel: Liste

```
class ListElement {
   ListElement  next;
   String  data;

   ListElement (String  data)  {
      this.data = data;
   }
}

class MyList  {
   ListElement head;

   MyList()  {
      head = null;
   }

public void printall()  {
   ListElement elem = head;
   while (elem != null) {
      Out.println(elem.data);
      elem = elem.next;
   }
}

public void add (String  data)  {
   ListElement elem = new ListElement(data);

   if (head == null)
      head = elem;
   else {
      ListElement e = head;
      while (e.next != null)
         e = e.next;
      e.next = elem;
   }
}

public void addFirst(String data) {
   ListElement elem = new ListElement(data);
   elem.next = head;
   head = elem;
}
```

ÜBlatt: Stapel als doppelt verkettete Liste

= Stapel, bei dem die Elemente jeweils einen Zeiger auf ihren Vorgänger und Ihren Nachfolger haben. *topNode* zeigt immer auf das oberste Element!

```
push(“Party“)
push(“Ferien“)
push(“Relaxen“);
x = pop();
…
```

Fragen ???

Viel Spaß mit dem Übungsblatt!