EML Research

http://www.eml-research.de/

Java - Part 4

Katja Wegner

katja.wegner@eml-r.villa-bosch.de

Überblick

⇒ Objektorientierung (2)

- Pakete
- Vererbung
- Zugriffsmodifikatoren
- Kapselung

Pakete

⇒ mehrere Klassen in einem Paket zusammenfassen (package)

```
package java.lang;          // class-Dateien in %JAVAHOME%/lang
package infFem.meinPaket;   // class-Dateien in /home/katja/infFem/meinPaket
```

⇒ Klassen anderer Pakete verwenden (import)

Importieren im Java-Programm	in der Arbeitsumgebung
`import java.util.Hashtable;`	
`import infFem.meinPaket.*;`	`CLASSPATH=/home/katja/`

Pakete in der API

⇒ package java.io: Ein- und Ausgabe

⇒ package java.lang: Basisklassen

⇒ package java.math: lange Zahlen

⇒ package java.text: Textverarbeitung

⇒ package java.util: Nützliches

⇒ ...

Vererbung

⇒ Wiederverwenden von Attribute und Methoden einer Basisklasse in Unterklassen

⇒ d.h. Weitergeben der definierten Eigenschaften einer Klasse

⇒ d.h. Ableiten der Unterklasse von der Basisklasse

⇒ d.h. Anpassen und Erweitern bestehender Klassen

⇒ Reduziert Programmieraufwand

⇒ abgeleitete Klasse = **Basis-** oder **Superklasse**

in Java:

⇒ alle Klassen sind Erweiterungen zu Basisklassen, d.h. sie sind **Unterklassen**

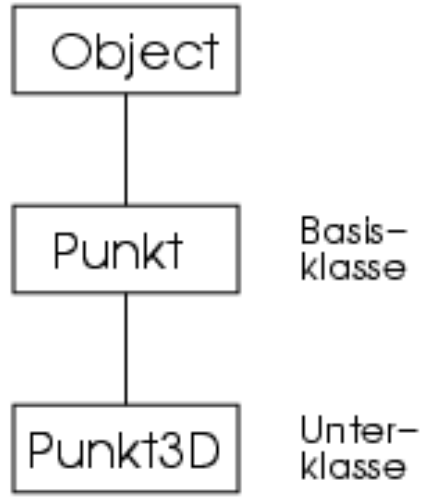
⇒ Basisklasse für alle Klassen: java.lang.Object

Anlegen einer Unterklasse

```
class Unterklasse extends Basisklasse {
        ...
}
```

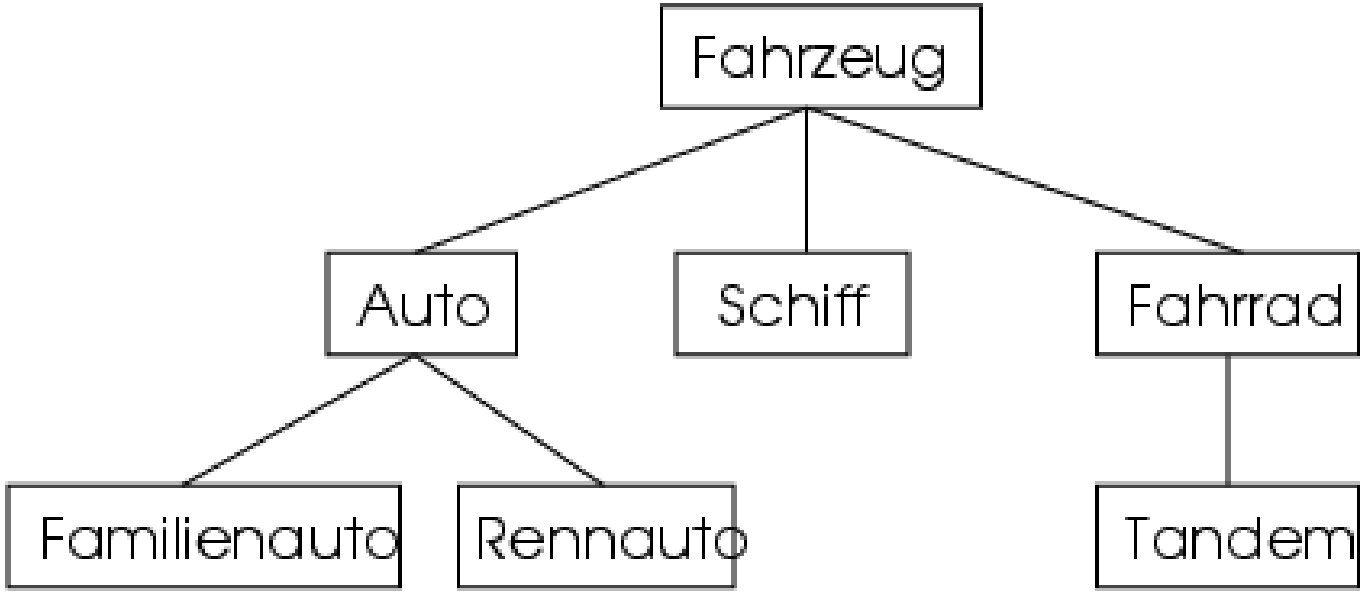
Beispiel:

```
class Punkt3D extends Punkt {
        int z;
        Punkt3D(int x, int y, int z) { // Konstruktor
                super(x, y);
                this.z = z;
        }
}
```



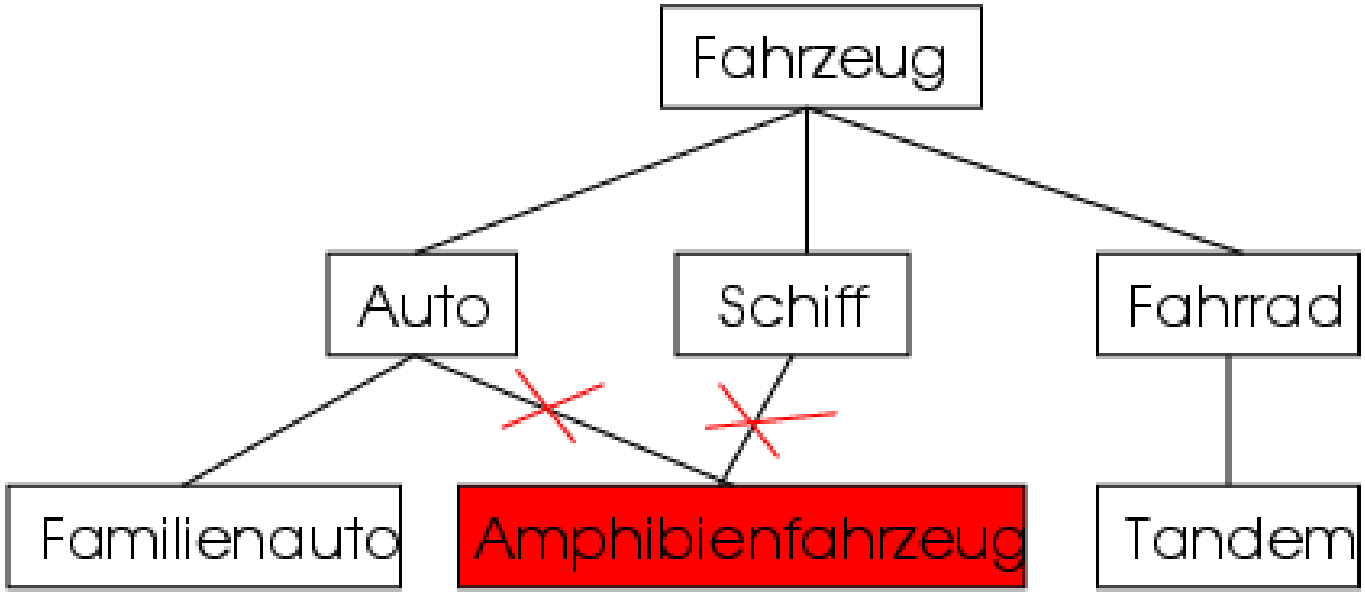
Vererbungshierarchie - Beispiel

Eine Hierarchie von Fahrzeugen:



Mehrfachvererbung

Mehrfachvererbung ist nicht erlaubt:



Überschreiben von Methoden

⇒ Definieren von Methoden in der Unterklasse

– mit gleichen Namen und denselben Parametern wie in der Basisklasse

```
class Punkt {
    int x,y;

    void skalieren(int faktor) {
        x = x * faktor;
        y = y * faktor;
    }
}
```

```
class Punkt3D extends Punkt {
    int z;

    void skalieren(int faktor) {
        x = x * faktor;
        y = y * faktor;
        z = z * faktor;
    }
}
```

Zugriffsmodifikatoren

⇒ **public**

- Zugriff durch Objekte aller Klassen

⇒ **protected**

- Zugriff durch Objekte der Klasse und allen Klassen aus demselben Paket und vererbten Unterklassen

⇒ **private**

- Zugriff nur durch Objekte derselben Klasse

⇒ <**friendly**>

- standardmäßig definiert, wenn kein Modifikator explizit angegeben wurde
- Zugriff nur durch Klassen desselben Pakets

Zugriffsmodifikatoren (2)

	Member-Sichtbarkeit			
Erreichbar für	**public**	**protected**	**friendly**	**private**
Definierte Klasse	Ja	Ja	Ja	Ja
Klasse im gleichen Paket	Ja	Ja	Ja	Nein
Abgeleitete Klasse in anderen Paketen	Ja	Ja	Nein	Nein
Nicht abgeleitete Klassen in anderen Paketen	Ja	Nein	Nein	Nein

Zugriffsrechte in Unterklassen

⇒ Unterklasse hat Zugriff auf alle als **public** und **protected** definierte Attribute und Methoden der Basisklasse

⇒ Geerbte Attribute und Methoden verhalten sich, als ob sie in Unterklasse selbst definiert wurden

⇒ **Geerbte Methoden** haben zusätzlich Zugriff auf **private** Elemente der Basisklasse

⇒ **ABER:** Konstruktoren sind nicht vererbbar

Kapselung (Information Hiding)

⇒ Verstecken von Attributen und Methoden

⇒ Verbergen interner Implementationsdetails einer Klasse

⇒ Vorteile:

- Einfache und klare Schnittstellen für die Verwendung einer Klasse
- Erhöhte Sicherheit, dass Daten stets gültige Werte haben
- Weitgehende Unabhängigkeit für die interne Programmierung
- Verhindert Programmierfehler beim Zusammenspiel der Klassen

Klasse Punkt mit Kapselung

```
class PunktKapselung {
    private int x,y;
    PunktKapselung(int x, int y) { // Konstruktor
        this.x = x;
        this.y = y;
    }
    PunktKapselung(Punkt punkt2) {
        this(punkt2.getX(), punkt2.getY());
    }

    public getX() { return x; }
    public setX(int x) { this.x = x; }
    public getY() { return y; }
    public setY(int y) { this.y = y; }
}
```

4.1. Übung

⇒ Erstellen Sie ein Paket **infFem.**<**IhrName**>, die entsprechenden Verzeichnisse und ändern Sie Ihren CLASSPATH.

⇒ Definieren Sie eine Klasse Knoten, die zu diesem Paket gehört und in der eine einfach Knotenstruktur eines Baumes definiert ist.

⇒ Diese soll drei Attribute besitzen: eines entspricht dem Namen des Knotens, eines repräsentiert die Elternknoten und das andere die Kinderknoten des Knoten.

⇒ Schreiben Sie Methoden zum Hinzufügen eines Elternknoten und eines Kindknoten zu diesem Knoten.

⇒ Überlegen Sie sich, welche Zugriffsmodifikatoren Sie für die Attribute und Methoden verwenden sollten und ergänzen Sie die dadurch notwendigen zusätzlichen Methoden.

4.2. Übung (Zusatz)

⇒ Schreiben Sie eine Klasse Baum mit einer Testmethode, in der drei Knoten erzeugt werden, von denen der erste Knoten der Elternknoten der anderen beiden ist.

⇒ Fügen Sie dem zweiten Knoten zwei Kinder hinzu.

⇒ Schreiben Sie eine Methode, um den Baum umzukehren und geben Sie die Namen aller Knoten aus.