



Vererbung



Programmieren 2

Kapitel 11: Vererbung

- 11.1 Motivation und Begriffsdefinitionen
- 11.2 Vorgehensweise und Implementierung
- 11.3 Arten von Vererbung
- 11.4 Konstruktoren
- 11.5 Abstrakte Klasse
- 11.6 Verschattung
- 11.7 Wurzelklasse Object
- 11.8 Zugriffsrechte und Sichtbarkeit
- 11.9 Schnittstelle



Abstrakte Klasse (1)

➤ Definition:

⊞ Klasse, die **nicht instanziiert** werden kann

➤ Zwei verschiedene Arten möglich:

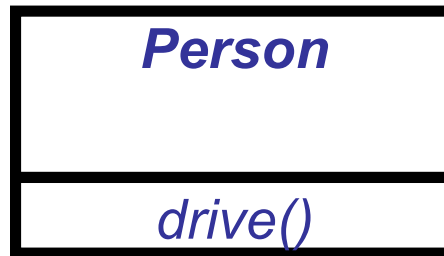
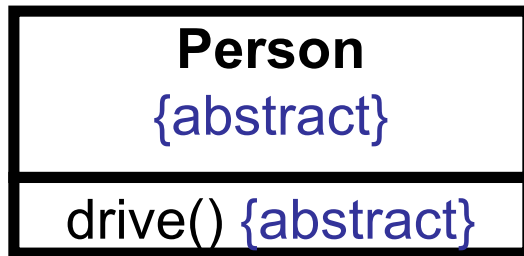
- (1) Alle Operationen werden – wie bei konkreten Klassen – vollständig implementiert
- (2) Mindestens eine Operation wird nicht implementiert (**abstrakte Operation**)
 - ⊞ Definiert nur Methodensignatur – Methodenrumpf ist leer
 - ⊞ Spezifiziert lediglich die Schnittstelle
 - ⊞ Abgeleitete Klassen müssen **alle** abstrakten Operationen der Oberklasse implementieren



Abstrakte Klasse (2)

➤ Notation in UML

- ⊞ Schlüsselwort `{abstract}`
- ⊞ Kursive Schrift
- ⊞ Bei handschriftlicher Darstellung besser mit Schlüsselwort!





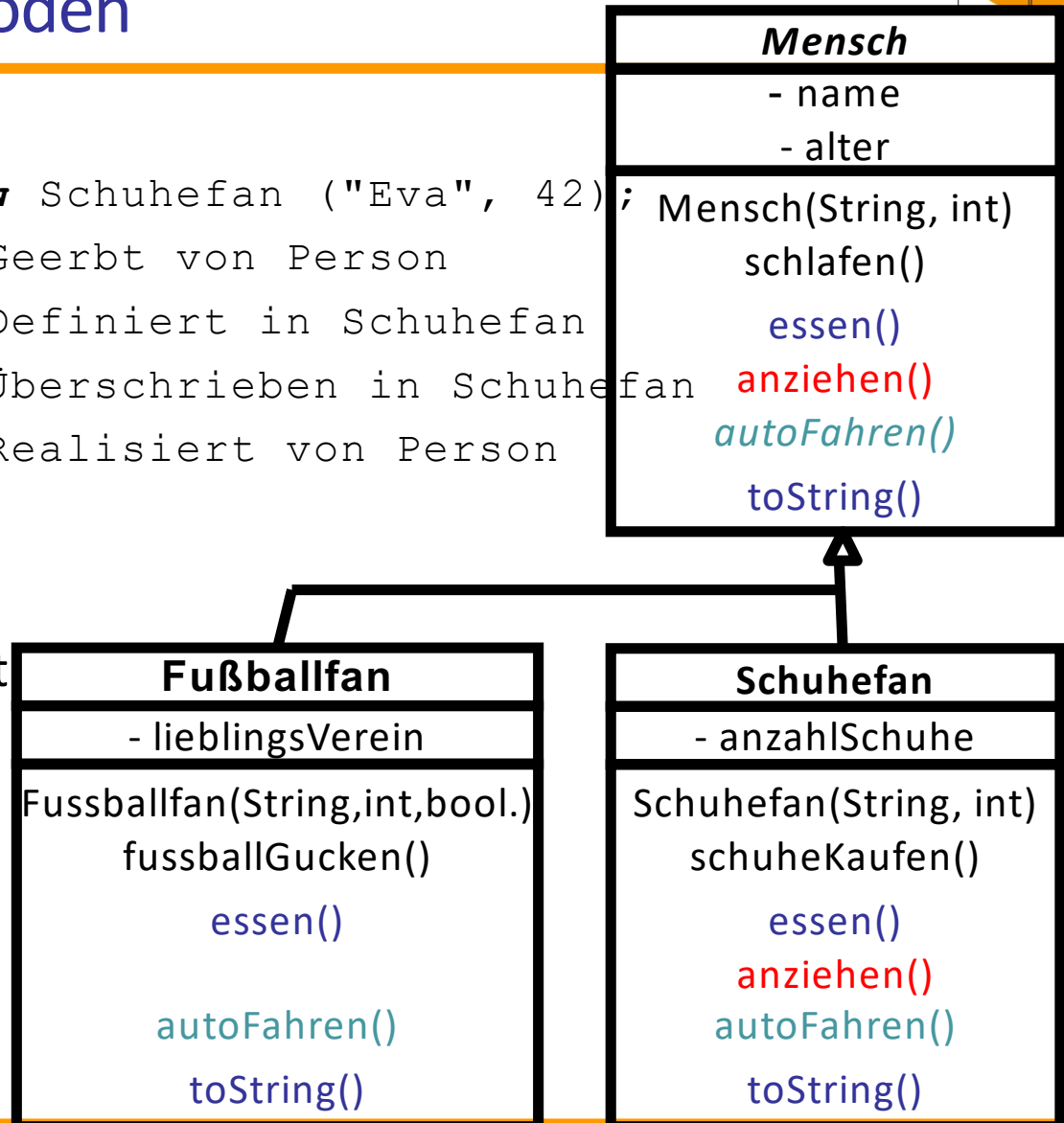
Aufruf ererbter Methoden

➤ Beispiel:

```
# Schuhefan eva = new Schuhefan ("Eva", 42);
# eva.sleep(); // Geerbt von Person
# eva.buyShoes(); // Definiert in Schuhefan
# eva.anziehen(); // Überschrieben in Schuhefan
# eva.drive(); // Realisiert von Person
```

➤ Verarbeitung:

- ✚ Compiler stellt sicher, dass die JVM zur Laufzeit eine Implementierung findet
- ✚ JVM sucht erst in Klasse selbst, dann in Basis-klass, dann in deren Basisklasse, usw.





Übung – Abstrakte Klasse

➤ Live Übung

- ✚ Bearbeiten Sie **Aufgabe 4** des Blatts Live Übung „Vererbung“
- ✚ Sie haben 5 Minuten Zeit.





Programmieren 2

Kapitel 11: Vererbung

- 11.1 Motivation und Begriffsdefinitionen
- 11.2 Vorgehensweise und Implementierung
- 11.3 Arten von Vererbung
- 11.4 Konstruktoren
- 11.5 Abstrakte Klasse
- 11.6 Verschattung
- 11.7 Wurzelklasse Object
- 11.8 Zugriffsrechte und Sichtbarkeit
- 11.9 Schnittstelle



Verschattung (1)

- Bereits bekannt von „normalen“ Klassen
 - ⊞ Lokale Variablen bzw. Parameternamen verschatten Attribute
- Neue Art der Verschattung bei Vererbung
 - ⊞ Attribut der Unterklasse verschattet Attribut der Oberklasse
 - ⊞ Methode der Unterklasse verschattet Methode der Oberklasse
- Zugriff
 - ⊞ Auf verschattetes Element `x` der Oberklasse: `super.x`
 - ⊞ Auf verschattetes Element `x` der aktuellen Klasse: `this.x`



Verschattung (2)

➤ Oberklasse

```
public abstract class Person {  
    ...  
    public String eat() {  
        return "eat : Mmmmh, lecker.\n";  
    }  
}
```

➤ Unterklasse

```
public class Fussballfan extends Person {  
    ...  
    public String eat() {  
        return super.eat() +  
            "Kann ich einen Nachschlag haben?";  
    }  
}
```



Modifizieren der Unterklassen

➤ Erweitern

- ⊞ Etwas Neues hinzufügen
- ⊞ Unterklasse erweitert Oberklasse um weitere Attribute, Operationen und/oder Beziehungen

➤ Redefinieren

- ⊞ Sich ähnlich verhalten
- ⊞ In Unterklasse geerbte Methoden aus der Oberklasse bei Bedarf durch eigene spezifische Implementierung überschreiben
- ⊞ Ggf. dabei geerbte Implementierungen verwenden

➤ Definieren

- ⊞ Etwas Versprochenes realisieren
- ⊞ Abstrakt deklarierte Operationen der Oberklasse in Unterklasse implementieren



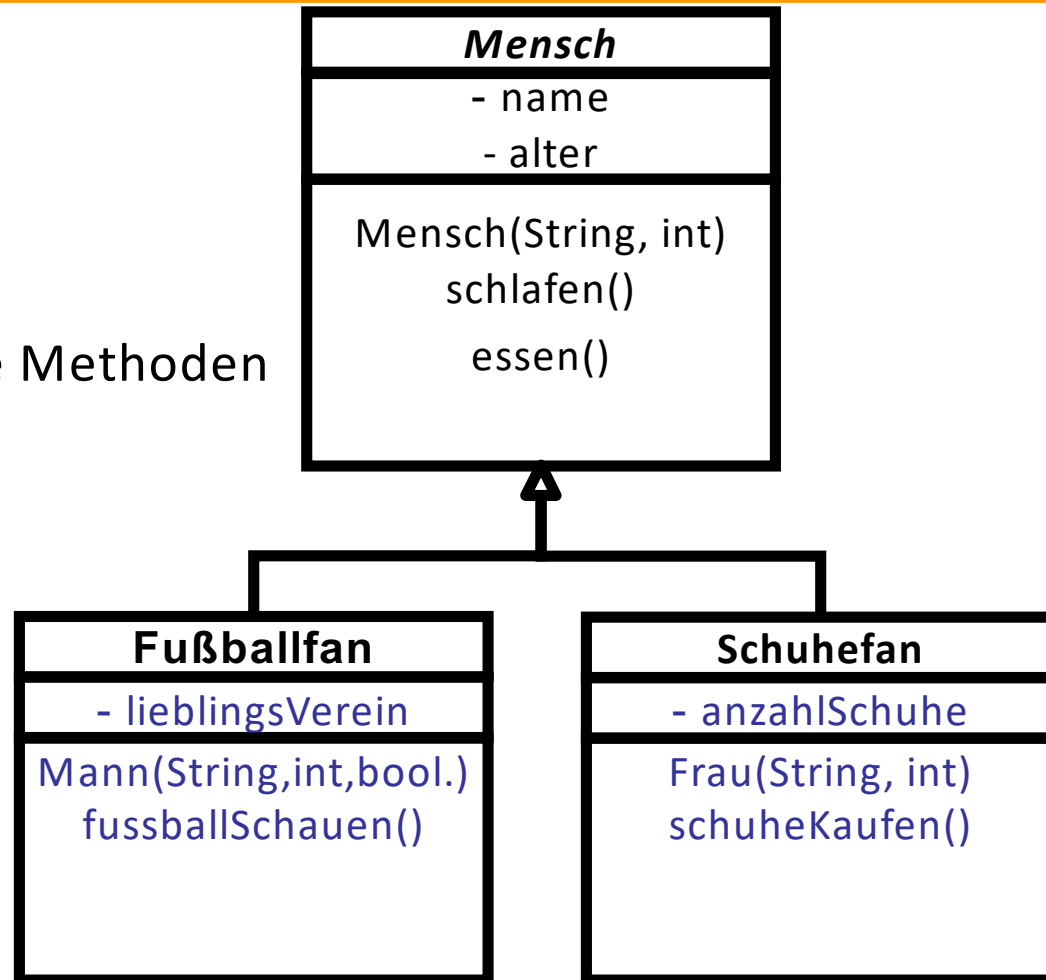
Eigenschaften in Unterklasse erweitern

➤ Ausgangsbasis

- ✚ Oberklasse Mensch
- ✚ Gemeinsame Attribute
- ✚ Grundlegende gemeinsame Methoden
- ✚ Konstruktor

➤ Erweiterung in Unterklassen

- ✚ Spezifische Attribute
- ✚ Spezifische Methoden
- ✚ Insbesondere eigene Konstruktoren





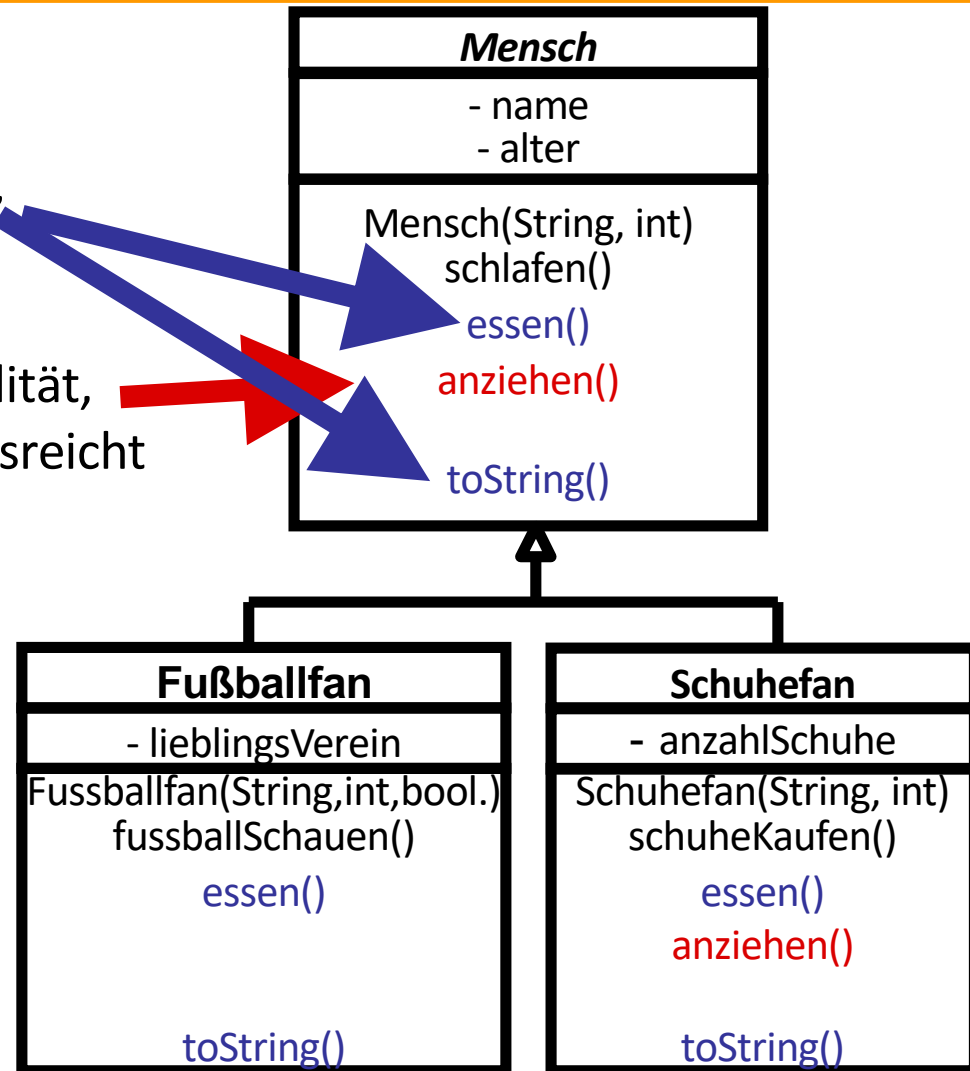
Eigenschaften in Unterklasse redefinieren

➤ Ausgangsbasis

- ⌘ Methode mit Basisfunktionalität, die in allen Unterklassen auftritt, aber ergänzt wird
- ⌘ Methode mit Standardfunktionalität, die für einige Unterklassen so ausreicht

➤ Redefinition in Unterklassen

- ⌘ **Überschreiben** der Methode aus der Oberklasse durch **spezifische Implementierung**
- ⌘ Einbinden der Implementierung aus der Oberklasse über **`super.<methodenname>()`**





Beispiel: Redefinieren

- Erweitern der Klasse Person um Methode anziehen()

```
public abstract class Person { ...  
    public String anziehen() {  
        return "anziehen: Unterhose und Socken";  
    } ...  
}
```

- Redefinieren der Klasse Schuhefan

```
public class Schuhefan extends Person { ...  
    public String anziehen() {  
        return „anziehen: Unterhose, Socken und Schuhe“;  
    } ...  
}
```



Beispiel: Redefinieren, Basisfunktion nutzen (1)

➤ Veränderungen gegenüber obigem Beispiel

- ✚ Klassen `Person`, `Main` unverändert
- ✚ In Klassen `Schuhefan`, `Fussballfan` jeweils Methode `eat()` überschreiben

➤ Redefinieren der Klasse `Schuhefan`

```
public class Schuhefan extends Person {  
    ...  
    public String eat() {  
        String result = super.eat();  
        result = result + "\n        Wirklich schade,  
        dass das so viele Kalorien hat...";  
        return result;  
    } ...  
}
```



Beispiel: Redefinieren, Basisfunktion nutzen (2)

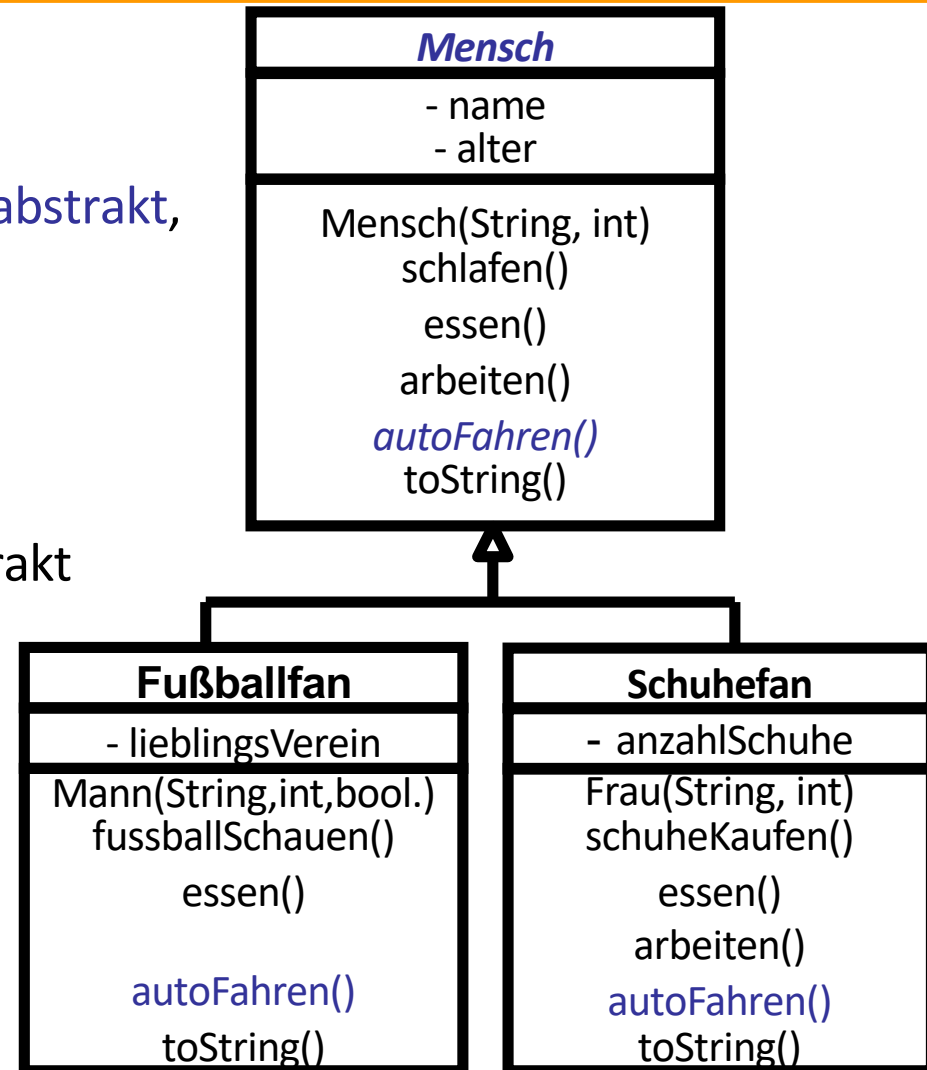
➤ Redefinieren der Klasse Fussballfan

```
public class Fussballfan extends Person {  
    ...  
    public String eat() {  
        String result = super.eat();  
        result = result + "\n Kann ich noch einen  
        Nachschlag haben?";  
        return result;  
    }  
    ...  
}
```



Eigenschaften in Unterklasse definieren

- Ausgangsbasis
 - ⊞ Operation *autoFahren()* ist **abstrakt**, d.h. definiert nur die Signatur
 - ⊞ Sichert damit die Existenz dieser Verhaltensweise
 - ⊞ **Keine Implementierung!**
 - ⊞ Oberklasse wird damit auch abstrakt
- Definition in Unterklassen
 - ⊞ Definieren zu den abstrakten Operationen **spezifische Implementierungen**
 - ⊞ In *jeder* nicht abstrakten Unterklasse erforderlich





Beispiel: Definieren (1)

- Veränderungen gegenüber obigem Beispiel
 - ⌘ Klasse `Main` unverändert
 - ⌘ Klasse `Person` definiert nur Schnittstelle der Methode `autoFahren()`; wird damit zur abstrakten Klasse
 - ⌘ Klasse `Schuefan`, `Fussballfan` jeweils erweitert um Implementierung der Methode `autoFahren()`

- Neue Version der Klasse `Person`

```
public abstract class Person {  
    ...  
    public abstract String drive();  
}
```



Beispiel: Definieren (2)

➤ Erweiterung von Fussballfan um Definition von drive()

```
public class Fussballfan extends Person {  
    ...  
    public String drive() {  
        return "Jetzt fahr schon, ich will zum Spiel!";  
    }...}
```

➤ Erweiterung von Schuhefan um Definition von drive()

```
public class Schuhefan extends Person {  
    ...  
    public String drive() {  
        return "Dann mal los zum Schuhladen!";  
    }...}
```



Übung – Methoden redefinieren

➤ Live Übung

- ✚ Bearbeiten Sie **Aufgabe 6** des Blatts Live Übung „Vererbung“
- ✚ Sie haben 5 Minuten Zeit.





Programmieren 2

Kapitel 11: Vererbung

- 11.1 Motivation und Begriffsdefinitionen
- 11.2 Vorgehensweise und Implementierung
- 11.3 Arten von Vererbung
- 11.4 Konstruktoren
- 11.5 Abstrakte Klasse
- 11.6 Verschattung
- 11.7 Wurzelklasse Object
- 11.8 Zugriffsrechte und Sichtbarkeit
- 11.9 Schnittstelle



Klasse Object

- Klasse Object ist voreingestellte Basisklasse aller Klassen
- Äquivalent:

```
class ClassName  
{...}
```


und

```
class ClassName extends Object  
{...}
```
- Jede Klasse ist von einer anderen Klasse abgeleitet, außer Object
- Alle Klassen (abgesehen von Object) haben, direkt oder indirekt, Object als gemeinsame Basisklasse



Vordefinierte Methoden in Object

- Methoden von Object werden an jede Klasse vererbt
- Beispiele:

<code>public String toString()</code>	Lesbare Repräsentation
<code>public boolean equals(Object x)</code>	true wenn dieses Objekt und x gleich sind, false ansonsten
<code>public int hashCode()</code>	Kennnummer
<code>protected Object clone()</code>	Erzeugt ein Duplikat
<code>public Class getClass()</code>	Typobjekt dieses Objektes

- Object-Methoden bieten zum Teil nur minimale Funktionalität
- Methoden sollten in der Regel redefiniert werden!



Programmieren 2

Kapitel 11: Vererbung

- 11.1 Motivation und Begriffsdefinitionen
- 11.2 Vorgehensweise und Implementierung
- 11.3 Arten von Vererbung
- 11.4 Konstruktoren
- 11.5 Abstrakte Klasse
- 11.6 Verschattung
- 11.7 Wurzelklasse Object
- 11.8 Zugriffsrechte und Sichtbarkeit
- 11.9 Schnittstelle



Zugriffsrechte und Sichtbarkeit (1)

- Zusätzliche Zugriffsrechte und Sichtbarkeitsregeln durch Vererbungskonzept
- 4 Kategorien:
 1. `public (+)`: „weltweiter“ Zugriff sowohl von außen als auch von allen Nachfahren unabhängig von Paketzugehörigkeit
 2. `private (-)`: nur innerhalb der eigenen Klasse sichtbar; werden vererbt, sind aber von der Unterklasse aus nicht zugreifbar
 3. `protected (#)`: von allen Nachfahren darf darauf zugegriffen werden unabhängig davon, ob sich die Nachfahren im gleichen Paket oder in einem anderen Paket befinden und von allen Klassen im gleichen Paket
 4. **Implizit**: nur innerhalb des Pakets sichtbar, in dem die Klasse definiert ist; gilt für Nachfahren und Nicht-Nachfahren



Zugriffsrechte und Sichtbarkeit (2)

- Regeln zum Ändern der Zugriffskategorie beim Überschreiben:
 - Zugriffsrechte dürfen **nur erweitert**, aber **nicht weiter eingeschränkt** werden
 - `public`-Operationen müssen `public` bleiben
 - `private`-Operationen, die in Unterklassen neu definiert werden, dürfen eine beliebige Zugriffskategorie haben, da es sich um neue Operationen handelt
 - Operationen ohne explizite Zugriffskategorie können so bleiben oder als `protected` oder `public` überschrieben werden
 - `protected` darf als `public` überschrieben werden



Programmieren 2

Kapitel 11: Vererbung

- 11.1 Motivation und Begriffsdefinitionen
- 11.2 Vorgehensweise und Implementierung
- 11.3 Arten von Vererbung
- 11.4 Konstruktoren
- 11.5 Abstrakte Klasse
- 11.6 Verschattung
- 11.7 Wurzelklasse Object
- 11.8 Zugriffsrechte und Sichtbarkeit
- 11.9 Schnittstelle



Schnittstelle – Bedeutung

- Definition: **Schnittstelle, Interface**
 - ⊞ Spezielle Form von Klasse
 - ⊞ Keine Objekte direkt von Interface ableitbar
- Verhalten
 - ⊞ Definiert **nur abstrakte Operationen**, keine Implementierungen
 - ⊞ Legt also nur Anforderungen fest
 - ⊞ Keine ausführbaren Anweisungen (seit Java8: `static` und `default` möglich)
 - ⊞ Keine Konstruktoren
- Eigenschaften
 - ⊞ Enthält keine veränderbaren Attribute
 - ⊞ **Öffentlich sichtbare Konstanten als Attribute möglich**
- Alle Methoden / Datenelemente haben implizit Sichtbarkeit **public**!



Schnittstelle – Umsetzung in Java

➤ Bedeutung von Java

- ⌘ Ermöglicht klare Trennung von Implementierung und Schnittstelle
- ⌘ Mehrfachvererbung von konkreten Klassen in Java nicht erlaubt
- ⌘ Implementierung von mehreren Schnittstellen ist aber möglich!!!

➤ Umsetzung in Java:

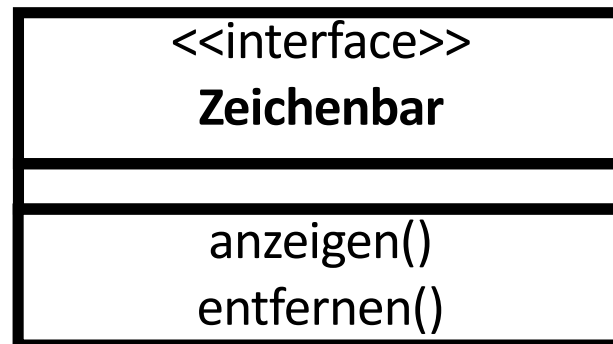
- ⌘ Reserviertes Wort `interface` (statt `class`)
- ⌘ Je Interface eigene `.java`-Datei, wird übersetzt zu `.class`-Datei



Schnittstelle in UML

➤ Schnittstelle in UML

- ⊞ Symbol analog zu Klasse
- ⊞ Stereotyp `<<interface>>` oberhalb des Klassennamens
- ⊞ Schnittstelle ist immer auch abstrakt, muss nicht explizit als abstrakt gekennzeichnet werden





Begriffe – Anbieter und Nutzer

➤ Anbieter einer Schnittstelle

- ⊞ Realisiert die Schnittstelle, d.h. implementiert die Operationen

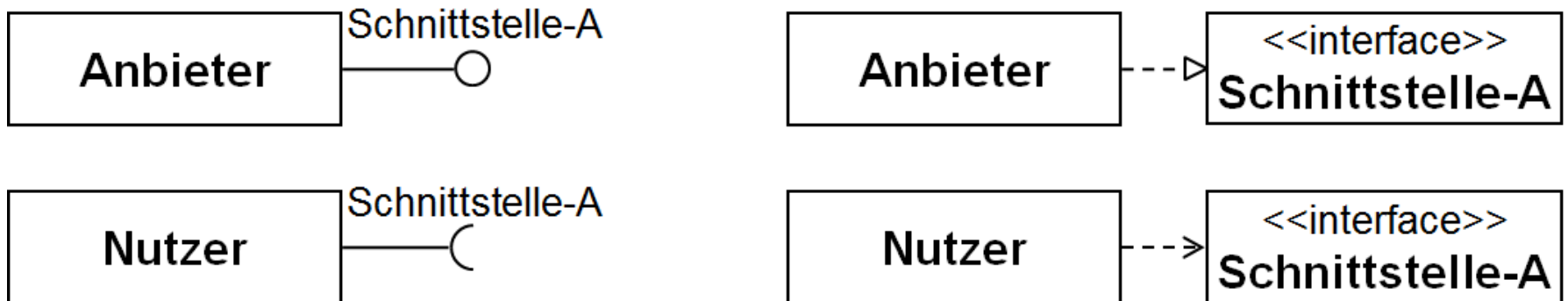
➤ Nutzer einer Schnittstelle

- ⊞ Verwendet die Schnittstelle, d.h. ruft die Operation auf
- ⊞ Kennt konkrete Implementierung nicht!

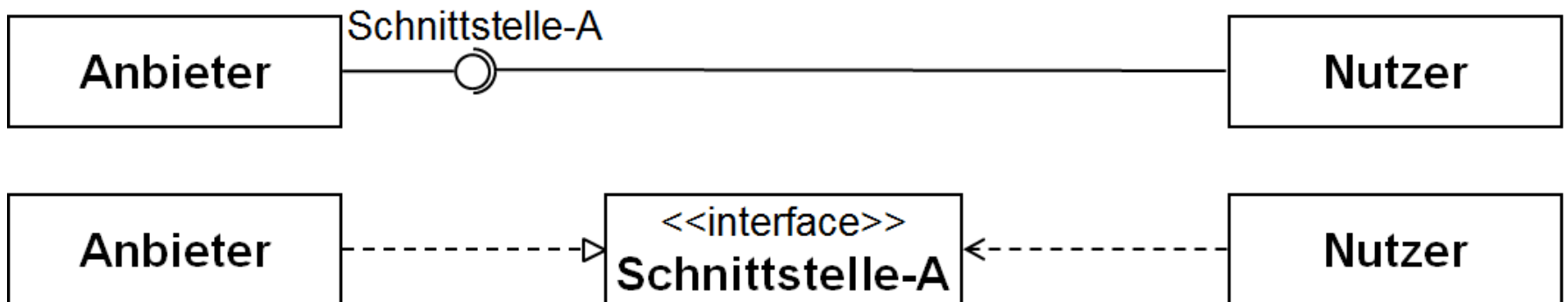


Anbieter und Nutzer in UML

Bereitstellung und Nutzung der Schnittstelle-A



Interaktion über Schnittstelle-A





Begriffe – Realisierung und Vererbung

➤ Realisierung

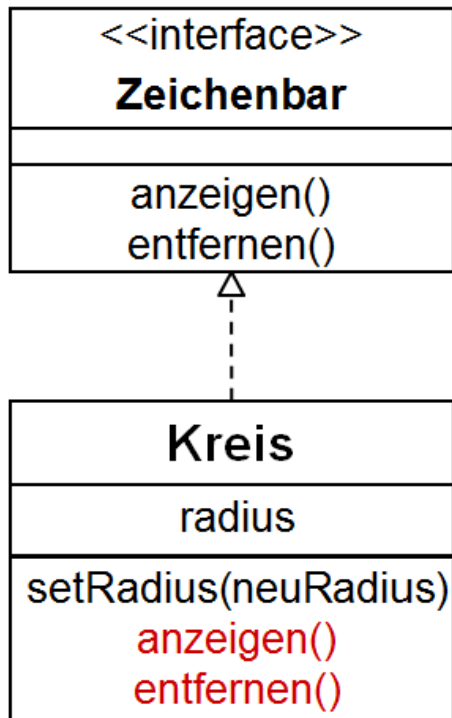
- ⊞ Schnittstelle alleine nicht ausführbar
- ⊞ Konkrete Klasse ist von Schnittstelle abgeleitet
- ⊞ Sprachgebrauch: „Konkrete Klasse implementiert das Interface“
- ⊞ Implementiert dabei alle definierten Operationen der Schnittstelle

➤ Vererbung zwischen Schnittstellen

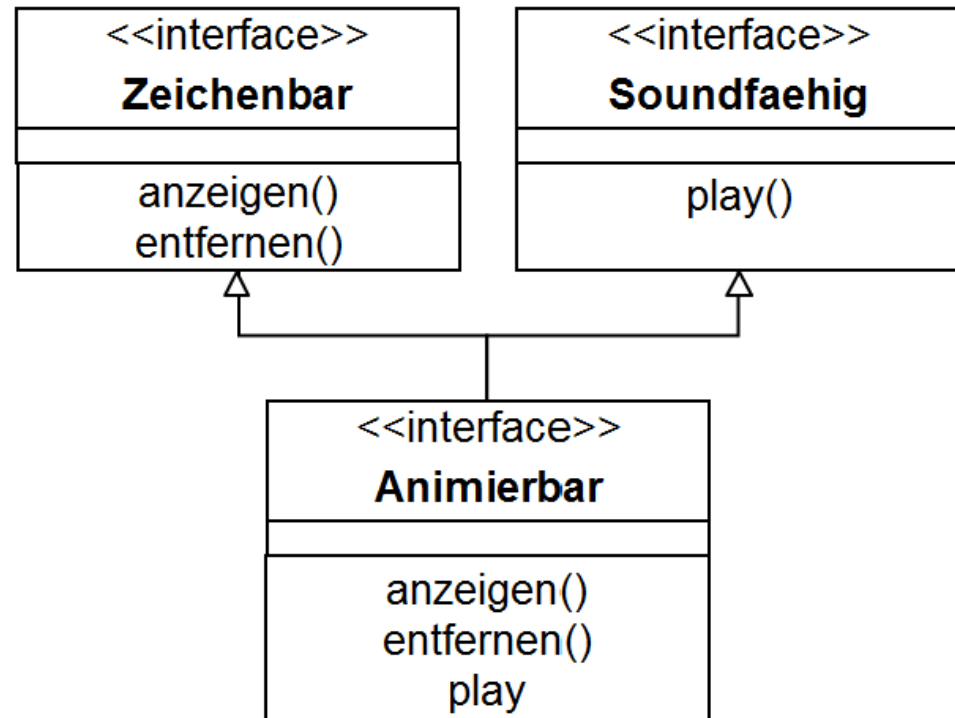
- ⊞ Neue Schnittstelle erweitert alte Schnittstelle
- ⊞ Dabei lediglich Hinzufügen von abstrakten Operationen
- ⊞ In Java: Interface kann mehrere Interfaces erweitern
- ⊞ D.h. Mehrfachvererbung zwischen Schnittstellen möglich!



Beispiel: Realisierung und Vererbung



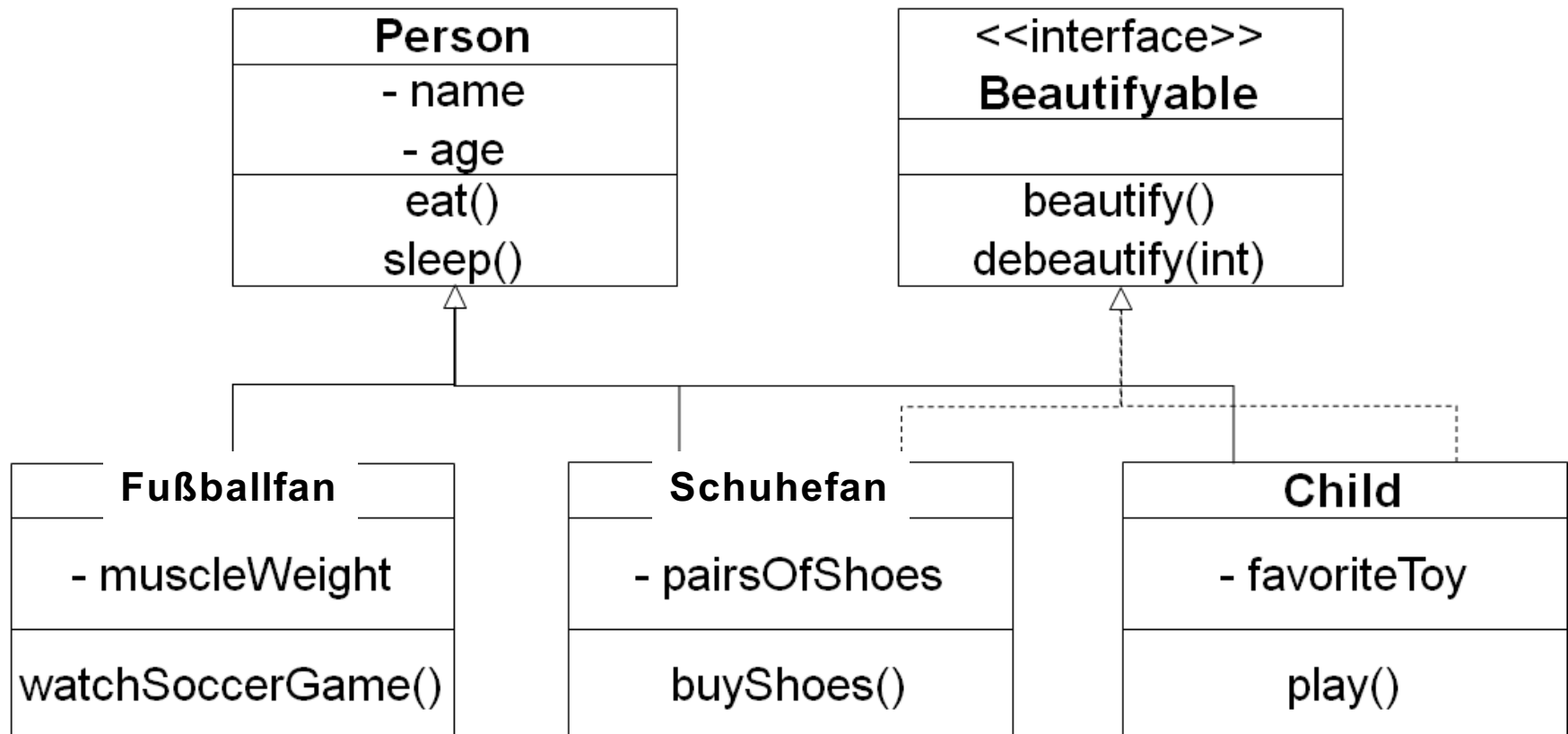
Realisierung



Vererbung



Beispiel





Implementierung des Interfaces

➤ Veränderungen gegenüber obigem Beispiel

- ⌘ Klassen `Fussballfan`, `Person` unverändert
- ⌘ Erweiterung der Klasse `Main`
- ⌘ Neues Interface `Beautifyable`
- ⌘ Klasse `Schuhefan` implementiert Interface `Beautifyable`
- ⌘ Neue Klasse `Child` implementiert Interface `Beautifyable`

➤ Neues Interface `Beautifyable`

```
public interface Beautifyable {  
    public String beautify();  
    public String debeautify(int minutes);  
}
```



Veränderungen der Main-Klasse

➤ Erweiterte Klasse Main

```
public class Main{  
    public static void main(  
        String[] args) {  
  
        Fussballfan adam = new Fussballfan();  
        Schuhefan eva = new Schuhefan();  
        Child kain = new Child();  
  
        adam.processPerson();  
        System.out.println(  
            adam.watchSoccerGame());  
        System.out.println();  
        ...
```

```
        ...  
        eva.processPerson();  
        System.out.println(  
            eva.buyShoes());  
        System.out.println(  
            eva.beautify());  
        System.out.println(  
            eva.debeautify(3));  
        System.out.println();  
  
        kain.processPerson();  
        System.out.println(  
            kain.play());  
        System.out.println(  
            kain.beautify());  
        System.out.println(  
            kain.debeautify(3));  
        System.out.println();  
    }  
}
```



Veränderung der Klasse Woman

➤ Erweiterte Klasse Schuhefan

```
public class Schuhefan extends Person implements Beautifyable{
    ...
    public String beautify() {
        return "Oh, mein Haar ist schon wieder durcheinander!.";
    }
    public String debeautify(int minutes) {
        return "Das schaff ich nie, mich in "
            + minutes + " Minuten zu stylen.";
    }
}
```



Implementierung der neuen Klasse Child

```
public class Child extends Person implements Beautifyable{
    private String favoriteToy;
    public String play() {
        return "play : Brrruummmm... Miau... Fiep...";
    }
    public String beautify() {
        return "Guck mal, ich bin ein Ritter" +
            " mit Prinzessinen-Krone!";
    }
    public String debeautify(int minutes) {
        return "Och, muss ich in "
            + minutes + " Minuten schon aufhören?";
    }
}
```



Ausgabe bei Ausführung von Main

Adam ist 21 Jahre alt und kann superfix Regale
zusammenschrauben.

sleep: Chrrrrr.... chrrrr...

eat : Mmmh, lecker.

play : Ja... JAA... TOOOOOOOR!!!

Eva ist 19 Jahre alt und hat 0 Paar Schuhe.

sleep: Chrrrrr.... chrrrr...

eat : Mmmh, lecker.

shop : DIE sind ja schick...; Paar Nummer 1

Oh, mein Haar ist schon wieder durcheinander!

Das schaff ich nie, mich in 3 Minuten zu stylen.

Kain ist 4 Jahre alt und hat als Lieblingsspielzeug Teddy

sleep: Chrrrrr.... chrrrr...

eat : Mmmh, lecker.

play: Brrruummm... Miau... Fiep...

Guck mal, ich bin ein Ritter mit Prinzessinen-Krone!

Och, muss ich in 10 Minuten schon aufhören?



Interfaces als Typ

➤ Interface als Typ

- ⊞ Definiert Referenztyp (analog zu Klasse)
- ⊞ Zulässig für Deklaration von Variable, Parameter, Rückgabetyp von Methode
- ⊞ Alle implementierenden Klassen kompatibel zum Interface

➤ Beispiel

```
Beautifyable b;  
b = new Child("Sabine", "Bobbycar");  
b.beautify();  
b = new Schuhefan("Peter", 42);  
b.beautify();
```

➤ Auswahl der Implementierung der Methode dynamisch zur Laufzeit!



Übung – Schnittstellen

➤ Live Übung

- ✚ Bearbeiten Sie **Aufgabe 7** des Blatts Live Übung „Vererbung“
- ✚ Sie haben 8 Minuten Zeit.





Typinformation zur Laufzeit

```
AbstrakteBasis ab1 = new NormaleBasis();  
AbstrakteBasis ab2 = new KonkreteA();  
NormaleBasis nb1 = new KonkreteB();  
KonkreteA ka1 = new KonkreteA();  
KonkreteB kb1 = new KonkreteB();
```

- Instanzen von Unterklassen können in Oberklassen gespeichert werden
- Laufzeitinformation über
 - ✚ **instanceof**: z.B. `if (ref instanceof MyClass) { ... }`
 - ✚ Via `Object.getClass()`