



# **Objektorientierte Programmierung**

## Bachelorstudiengang Informatik

Wintersemester 2021/22


Prof. Jan Rexilius



**FH Bielefeld**  
University of  
Applied Sciences

# Objektorientierte Programmierung

Bitte nehmen Sie an der  
Modulbefragung / Kursbewertung teil!

The bottom of the slide features a decorative graphic consisting of three overlapping orange shapes. From left to right, there is a dark orange triangle pointing upwards, a lighter orange trapezoid, and a dark orange shape that resembles a right-angled triangle or a portion of a larger shape, all creating a stepped, upward-sloping effect.

# Rückblick

- 1. Vererbung und Komposition**
- 2. Polymorphie**
- 3. Exceptions**

# Rückblick

## Vererbung

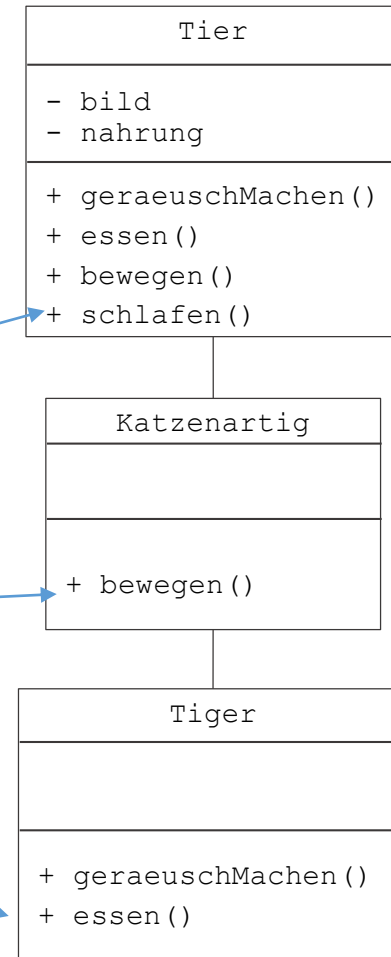
```
Tiger t = new Tiger();
```

```
t.schlafen();
```

```
t.bewegen();
```

```
t.essen();
```

```
t.geraeuschMachen();
```



# Rückblick

## Vererbung – Aufruf eines Basiskonstruktors

- Im Unterklassen-Konstruktor wird der Konstruktor der Oberklasse aufgerufen
- mit **super(...)** als 1. Anweisung im Block des Konstruktors
  - darin können Parameter an den Oberklassen-Konstruktor gegeben werden
- wenn **super** fehlt, dann setzt der Compiler automatisch den parameterlosen Standard-Konstruktor der Oberklasse durch **super()** ein
  - wenn dieser in der Oberklasse fehlt, tritt Compilerfehler auf

```
class Basis {
    int wert;

    Basis(int w) {
        wert = w;
    }
}

class Abgeleitet extends Basis {
    int wertAbgeleitet;

    Abgeleitet(int p1, int p2) {
        super (p2) ;
        wertAbgeleitet = p1;
    }
}
```

# Rückblick

## Vererbung – IS-A und HAS-A

- Unterklasse **IS-A** Superklasse
  - Tier IS-A Löwe? Nein
  - Löwe IS-A Tier? Ja
- Zoo **HAS-A** Nashorn, wenn Zoo eine Instanzvariable vom Typ Nashorn hat
  - **Keine** Vererbung
  - Beschreibt Verhältnis zwischen zwei Klassen

### Vererbung

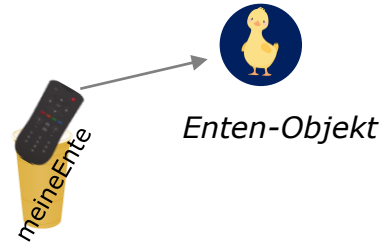
```
class X {  
    ...  
}  
  
class Y extends X {  
    ...  
}
```

### Komposition

```
class X {  
    ...  
}  
  
class Y {  
    X var;  
}
```

# Rückblick

## Polymorphie



```
Tier meineEnte = new Ente(); Tier
```

### Polymorphe Arrays

```
Tier[] tiere = new Tier[4];
tier[0] = new Ente();
tier[1] = new Loewe();
tier[2] = new Nashorn();
tier[3] = new Tier();

for (int i=0; i<tiere.length; i++) {
    tiere[i].essen();
    tiere[i].bewegen();
}
```

### Argumente und Rückgabetypen

```
class Tierarzt {
    public void spritzeGeben(Tier t){
        t.geraeuschMachen();
    }
}

class Tierbesitzer {
    public void start() {
        Tierarzt arzt = new Tierarzt();
        Ente e = new Ente();
        Nashorn n = new Nashorn();

        arzt.spritzeGeben(e);
        arzt.spritzeGeben(n);
    }
}
```

# Rückblick

## Exceptions

**try erfolgreich...**

```
public int division(int zaehler, int nenner) {  
    int ergebnis;  
    try {  
        // hier kommt der riskante Code  
        ergebnis = zaehler / nenner;  
        // hier kommt noch mehr Code  
    } catch (ArithmeticException ex) {  
        System.out.println(ex.getMessage());  
        ex.printStackTrace();  
        ergebnis = 0;  
    }  
    return ergebnis;  
}
```

**try scheitert...**



# Rückblick

## Exceptions

```
void test() throws IOException {  
    ...  
    if () {  
        throw new IOException();  
    }  
    ...  
}
```

```
void foo() throws IOException {  
    ...  
    test();  
    ...  
}
```

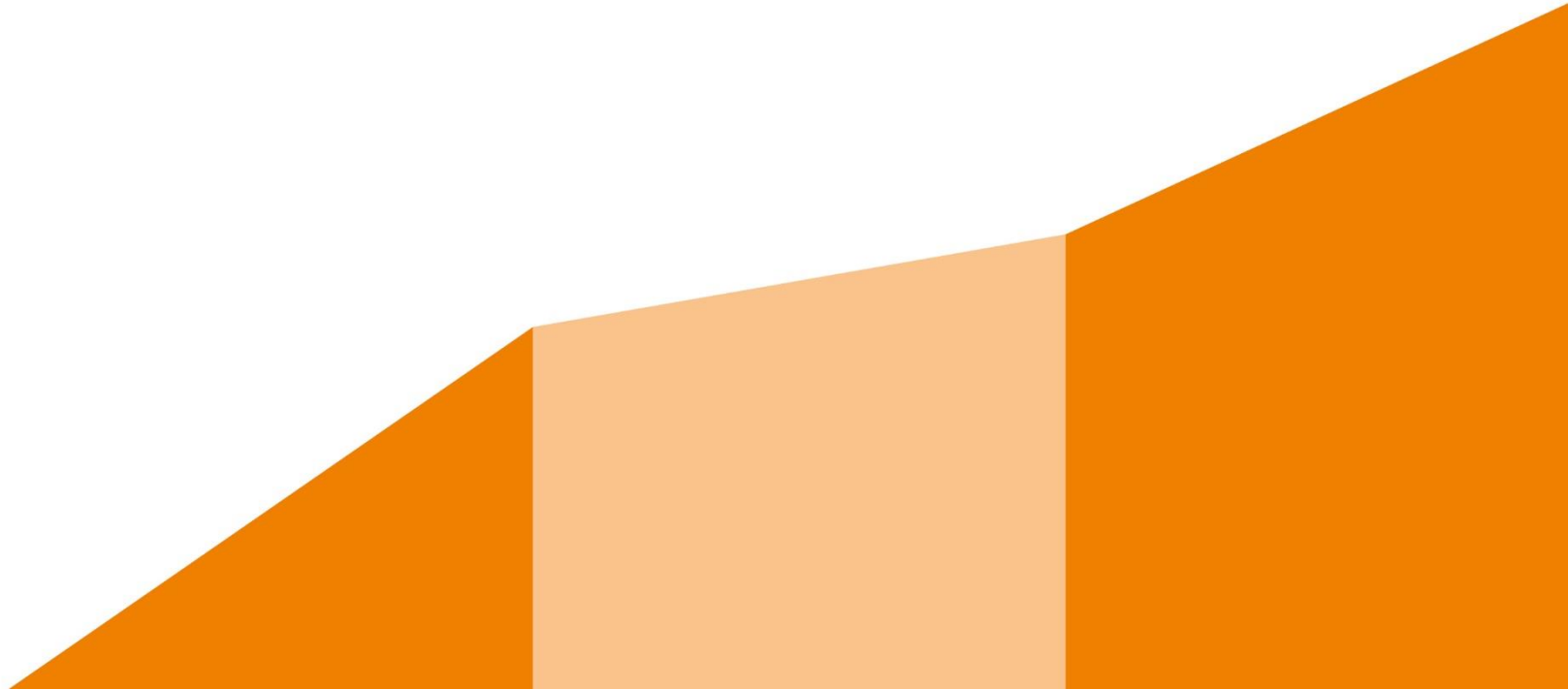
```
void method() {  
    try {  
        ...  
        foo();  
        ...  
    } catch (IOException e) {  
        ... // Fehlerbehandlung  
    }  
}
```

# Inhalte dieser Vorlesung

- 1. Abstrakte Klassen und Methoden**
- 2. Schnittstellen**
- 3. Generische Programmierung**

# Objektorientierte Programmierung

## Abstrakte Klassen und Methoden

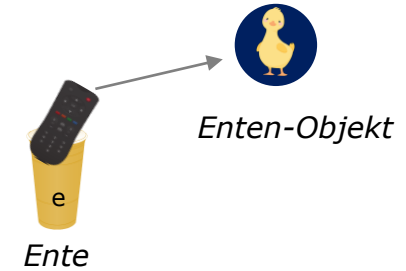


# Abstrakte Klassen und Methoden

## Einführung

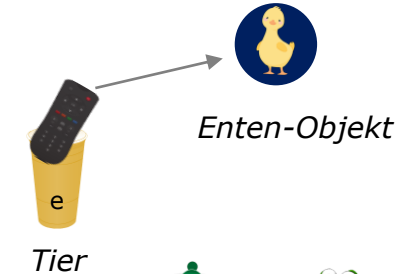
### 1. Referenztyp und Objekttyp gleich

```
Ente meineEnte = new Ente();
```



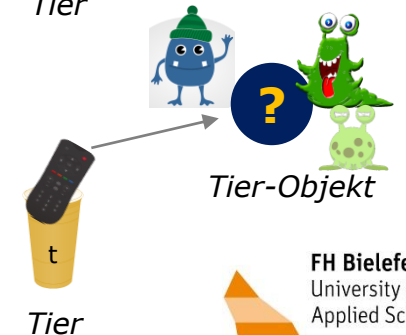
### 2. Referenztyp und Objekttyp nicht gleich

```
Tier meineEnte = new Ente();
```



### 3. Was passiert hier?

```
Tier tier = new Tier();
```



# Abstrakte Klassen und Methoden

## Abstrakte Klassen

```
abstract class Katzenartig extends Tier {  
    public bewegen() {}  
    public hatPfoten() {}  
}
```

```
public class KatzenartigTest {
```

```
    public void los() {  
        Katzenartig k;
```

```
        k = new Katzenartig();
```

```
        k = new Loewe();
```

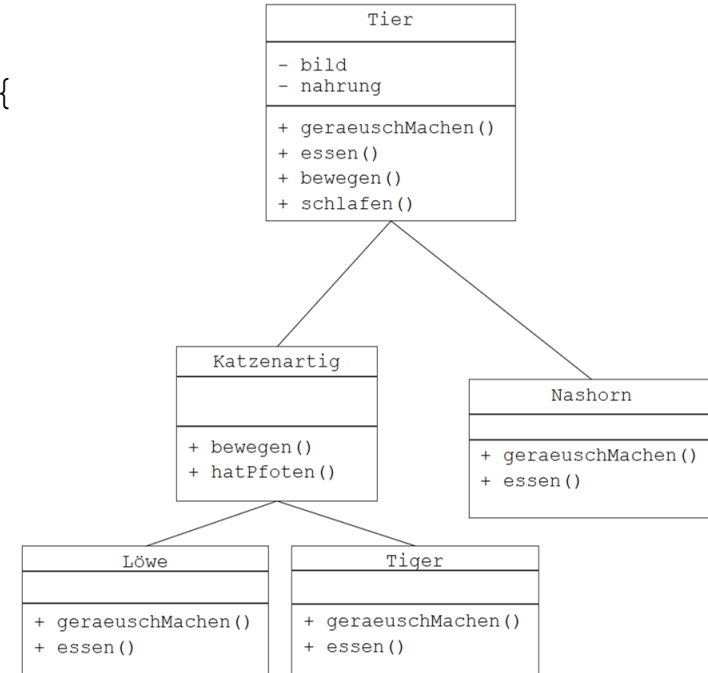
```
        k.bewegen();
```

```
    }
```

```
}
```

OK

Compiler  
error



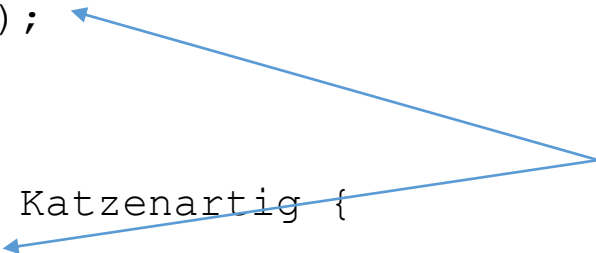
# Abstrakte Klassen und Methoden

## Abstrakte Methoden

- Unterklassen erben die abstrakten Methoden und müssen diese überschreiben (sonst müssen sie selber abstrakt definiert werden)

```
abstract class Katzenartig extends Tier {  
    public abstract bewegen();  
    public hatPfoten() {}  
}
```

```
public class Loewe extends Katzenartig {  
    public void bewegen() {  
        // Löwe, beweg dich!  
    }  
}
```



# Abstrakte Klassen und Methoden

## Zusammenfassung

- Eine abstrakte Klasse hat keine direkte Instanzen (eine nicht-abstrakte Klasse wird auch „konkrete Klasse“ genannt)
- Abstrakte Klassen werden mit dem entsprechenden Schlüsselwort `abstract` markiert

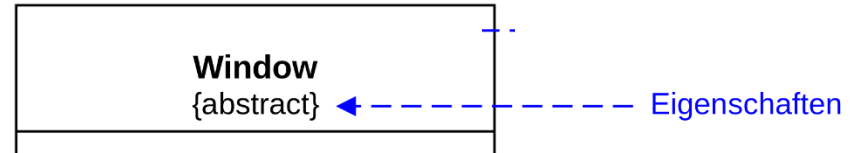
```
public abstract class Klassenname { ... }
```

- Eine abstrakte Klasse kann abstrakte und nicht-abstrakte Methoden haben.
- Abstrakte Methoden können nur in abstrakten Klassen angegeben werden. aber: abstrakte Klassen müssen nicht nur abstrakte Methoden haben.
- Der Compiler verbietet die Erzeugung von Objekten einer abstrakten Klasse

# Abstrakte Klassen und Methoden

## UML

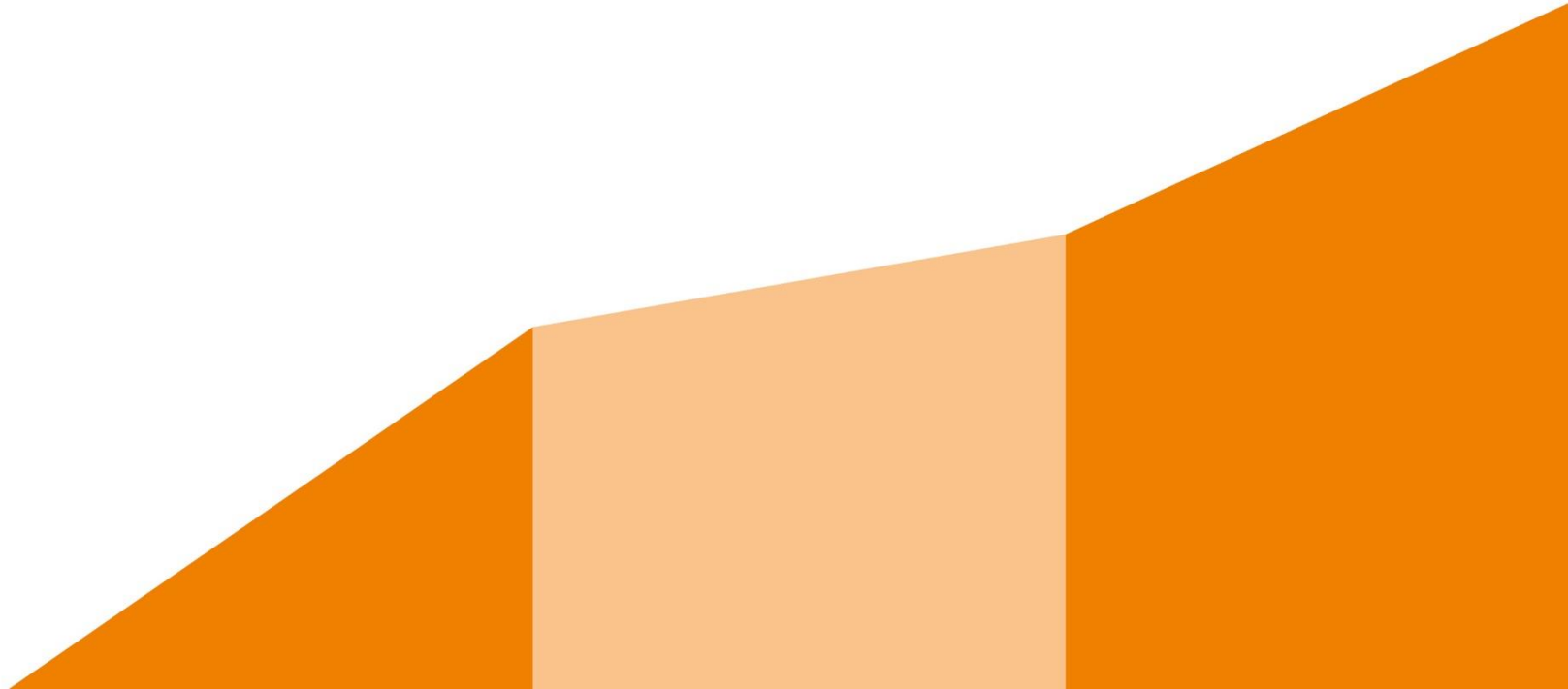
- {abstract}  
unter dem Klassennamen
- alternativ:  
Klassenname kursiv





# Objektorientierte Programmierung

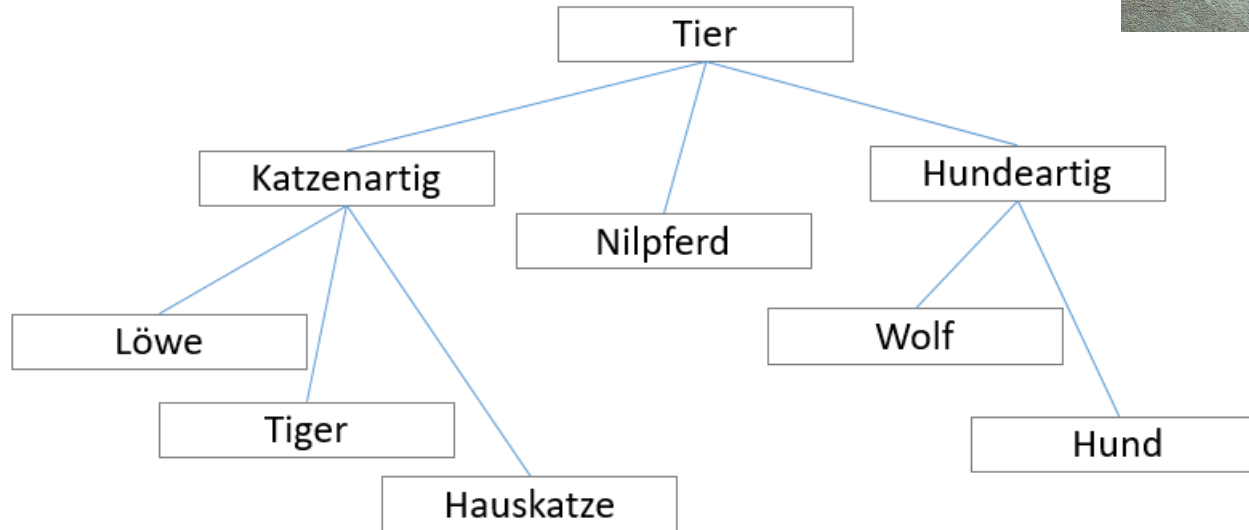
## Schnittstellen



# Schnittstellen

## Einführung – Zoohandlung

- Ziel: Neue Software für eine Zoohandlung
  - Tiere: kleine Tiere wie Hunde, Katzen, Vögel, etc.
  - Haustierverhalten: spielen()



# Schnittstellen

## Zoohandlung – Variante 1

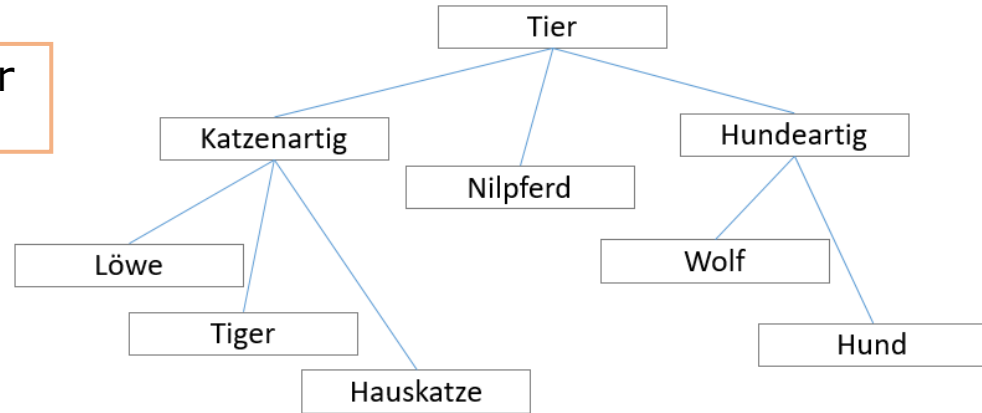
- Haustier-Methoden in Klasse Tier

### Vorteil:

- Alle Tiere erben neue Methoden
- Vorhandene Unterklassen nicht ändern

### Nachteil:

- Nicht jedes Tier ist ein Haustier
- Spezielle Implementierung für unterschiedliche Haustiere erforderlich



# Schnittstellen

## Zoohandlung – Variante 2

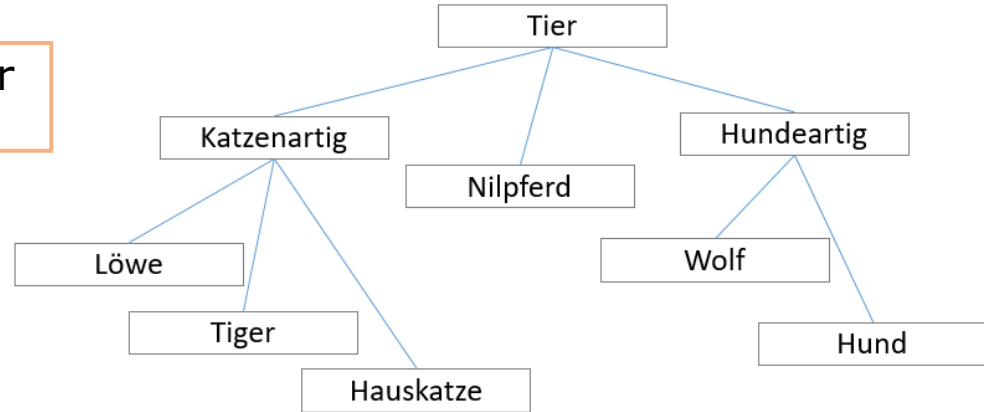
- Haustier-Methoden in Klasse Tier UND Methoden abstrakt machen

### Vorteil:

- Methoden machen für Nicht-Haustiere „nichts“

### Nachteil:

- alle! konkreten Klassen *müssen* die Methoden implementieren



# Schnittstellen

## Zoohandlung – Variante 3

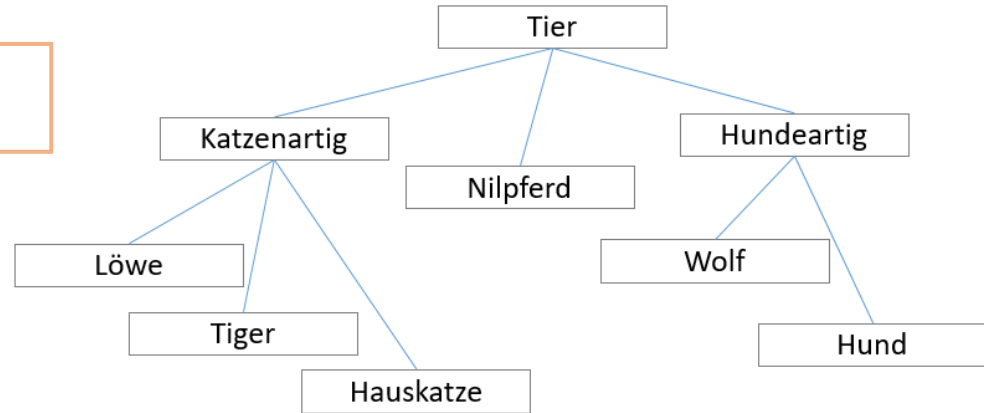
- Haustier-Methoden nur für Haustiere implementieren

### Vorteil:

- Methoden nur für Haustiere

### Nachteil:

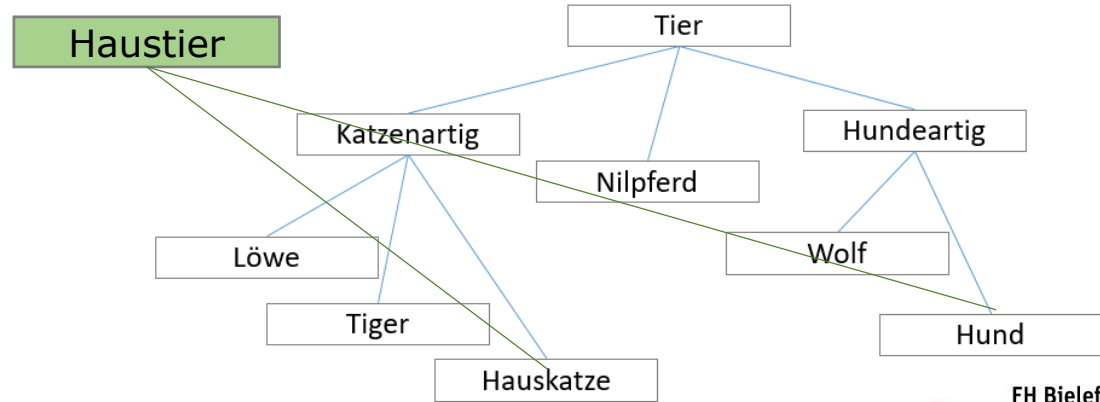
- Programmierer müssen die Methoden kennen, die für neue Haustiere implementiert werden müssen
- keinen Polymorphie für Haustier-Methoden



# Schnittstellen

Was wir gerne hätten...

- Haustier-Methoden nur in Haustier-Klassen
- alle Haustiere implementieren die gleichen Methoden
- Aber:
  - Mehrfachvererbung in Java nicht erlaubt
  - Klassen können nur von einer Superklasse erben



# Schnittstellen

## Einführung

- Idee: **alle** Methoden abstrakt machen
- Um ein Interface zu definieren:

```
public interface Haustier {  
    public void spielen();  
}
```

Methoden sind abstrakt, müssen  
also mit einem Semikolon enden

- Um ein Interface zu implementieren:

```
public class Hund extends Hundeartig implements Haustier  
{  
    @Override public void spielen() {...}  
}
```

Die Interface-Methoden  
müssen implementiert werden



# Schnittstellen

## Zusammenfassung

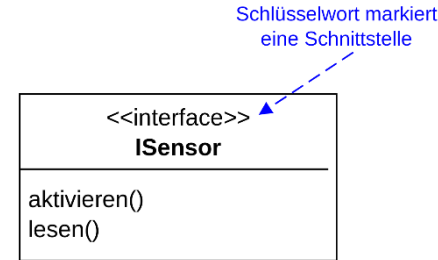
- Interfaces sind Beschreibungen die festlegen, **WAS** mit einer Schnittstelle gemacht werden kann (nicht **WIE**)
- Anstatt Schlüsselwort `class` wird `interface` verwendet
- Statt `extends` wird `implements` verwendet
- Standardfall: Methoden sind abstrakt und `public`
- Methode mit Implementierung: Schlüsselwort `default` erforderlich
- Implementierende Klasse muss entweder alle im Interface angegebenen Methoden definieren oder als `abstract` deklariert werden
- Eine Klasse kann beliebig viele Interfaces implementieren (Komma als Trennzeichen)



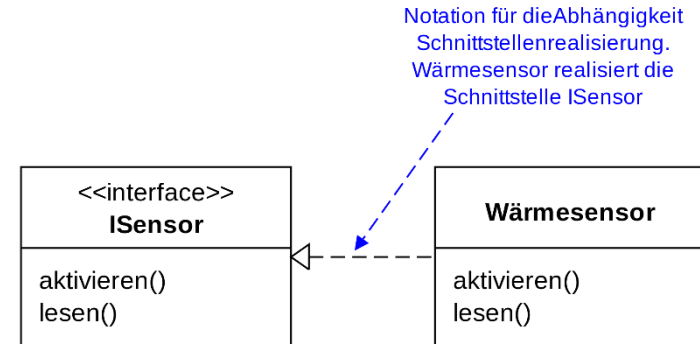
# Schnittstellen

## UML

- Interface-Klasse  
<<interface>>



- Abhängigkeiten zwischen Klassen



# Schnittstellen

## Übung – Wie sieht das Klassendiagramm aus?

- 1 

```
interface Foo {}  
public class Bar implements Foo {}
```
- 2 

```
interface Test {}  
public abstract class Maus implements Test {}
```
- 3 

```
public abstract class Hund implements Haustier {}  
public class Wuffi extends Hund {}  
public interface Haustier {}
```
- 4 

```
public class Gamma extends Delta implements Epsilon {}  
public interface Epsilon{}  
public class Alpha extends Gamma implements Beta {}  
public class Delta  
public interface Beta {}
```



# Schnittstellen

## Übung – Wie sieht das Klassendiagramm aus?

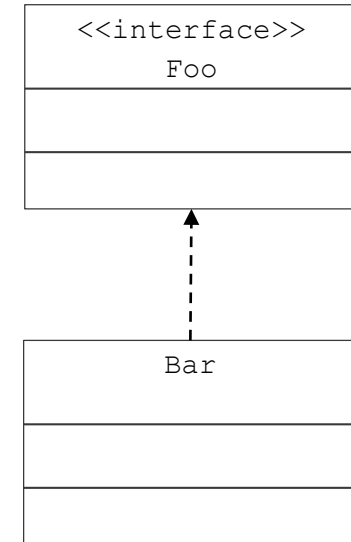
- 1 

```
interface Foo {}  
public class Bar implements Foo {}
```
- 2 

```
interface Test {}  
public abstract class Maus implements Test {}
```
- 3 

```
public abstract class Hund implements Haustier {}  
public class Wuffi extends Hund {}  
public interface Haustier {}
```
- 4 

```
public class Gamma extends Delta implements Epsilon {}  
public interface Epsilon{}  
public class Alpha extends Gamma implements Beta {}  
public class Delta
```



# Schnittstellen

## Übung – Wie sieht das Klassendiagramm aus?

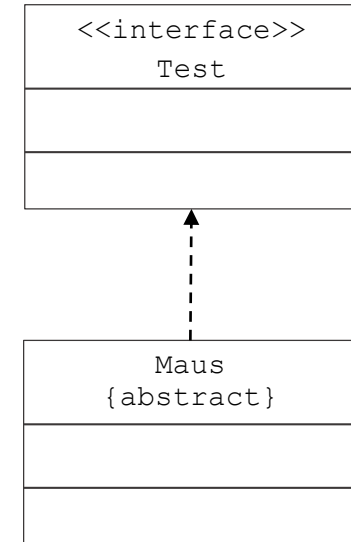
- 1 

```
interface Foo {}  
public class Bar implements Foo {}
```
- 2 

```
interface Test {}  
public abstract class Maus implements Test {}
```
- 3 

```
public abstract class Hund implements Haustier {}  
public class Wuffi extends Hund {}  
public interface Haustier {}
```
- 4 

```
public class Gamma extends Delta implements Epsilon {}  
public interface Epsilon{}  
public class Alpha extends Gamma implements Beta {}  
public class Delta
```



# Schnittstellen

## Übung – Wie sieht das Klassendiagramm aus?

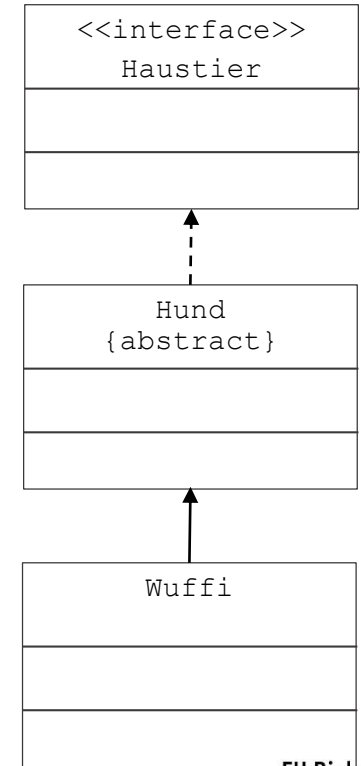
- 1 

```
interface Foo {}  
public class Bar implements Foo {}
```
- 2 

```
interface Test {}  
public abstract class Maus implements Test {}
```
- 3 

```
public abstract class Hund implements Haustier {}  
public class Wuffi extends Hund {}  
public interface Haustier {}
```
- 4 

```
public class Gamma extends Delta implements Epsilon {}  
public interface Epsilon{}  
public class Alpha extends Gamma implements Beta {}  
public class Delta
```



# Schnittstellen

## Übung – Wie sieht das Klassendiagramm aus?

1

```
interface Foo {}  
public class Bar implements Foo {}
```

2

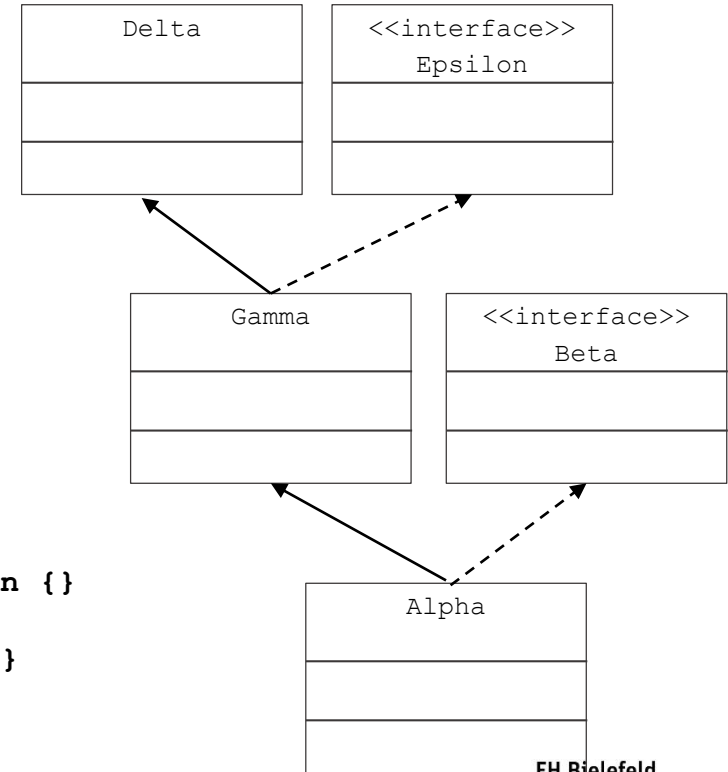
```
interface Test {}  
public abstract class Maus implements Test {}
```

3

```
public abstract class Hund implements Haustier {}  
public class Wuffi extends Hund {}  
public interface Haustier {}
```

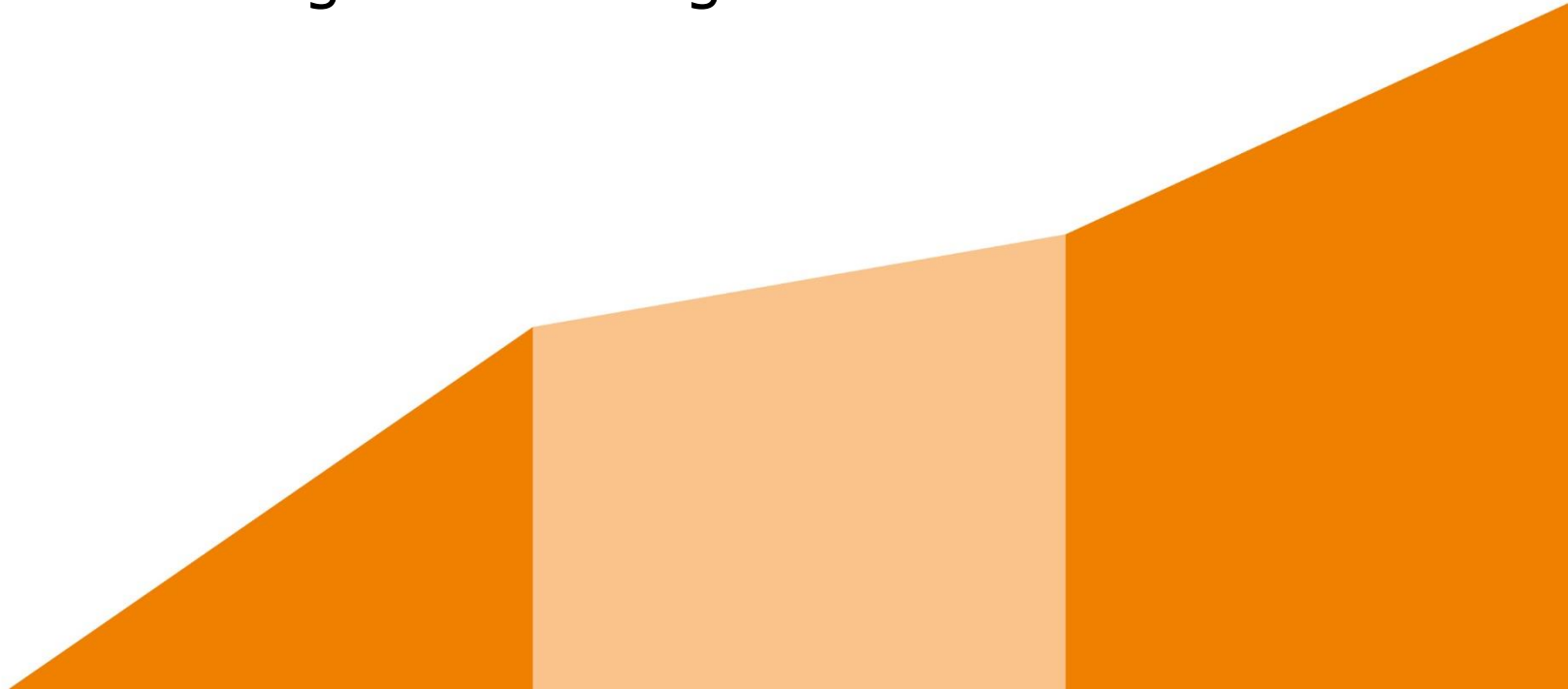
4

```
public class Gamma extends Delta implements Epsilon {}  
public interface Epsilon {}  
public class Alpha extends Gamma implements Beta {}  
public interface Beta {}
```



# Objektorientierte Programmierung

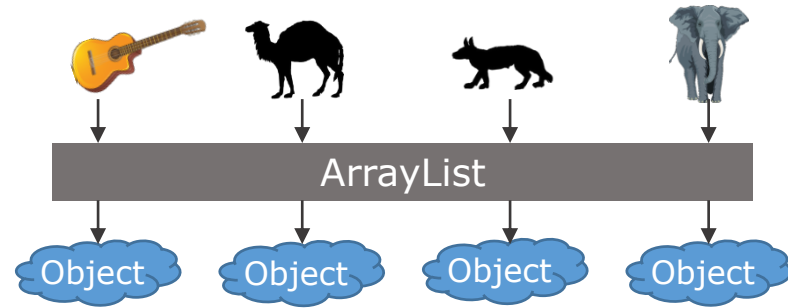
## Generische Programmierung



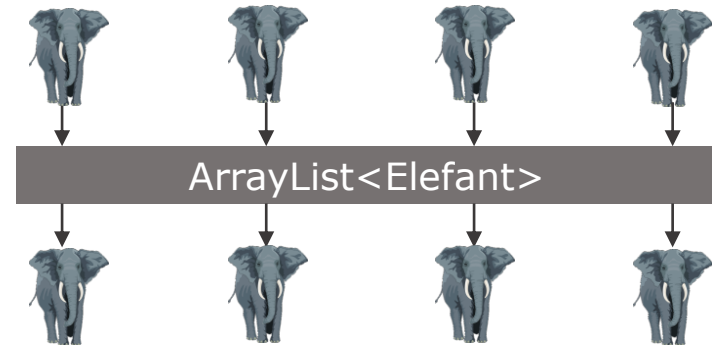
# Generische Programmierung

## Einführung – ArrayList

- Ohne Generics



- Mit Generics





# Generische Programmierung

## Einführung



```
class Schachtel {  
    Double  
    private Integer inhalt;  
  
    void hineinlegen(Double Integer obj) {  
        inhalt = obj;  
    }  
  
    Double  
    Integer herausnehmen() {  
        Double  
        Integer tmp = inhalt;  
        inhalt = null;  
        return tmp;  
    }  
}
```



```
class Schachtel<T> {  
    private T inhalt;  
  
    void hineinlegen(T obj) {  
        inhalt = obj;  
    }  
  
    T herausnehmen() {  
        T tmp = inhalt;  
        inhalt = null;  
        return tmp;  
    }  
}
```



```
Schachtel<Integer> box1 = new  
Schachtel<Integer>();
```

```
Schachtel<Double> box2 = new  
Schachtel<Double>();
```



# Generische Programmierung

## Einführung

- Idee:  
Typ der Basisdaten ist **Parameter** der Datenstruktur (*nur Referenztypen*)
- Generics für Klassen/Methoden, die für verschiedene Datentypen immer die gleiche Aktion durchführt
- Mehrere Typvariablen sind möglich (z.B. `class Name<T,E>`)
- Vorteile
  - Typprüfung bereits zur Compilezeit
  - Keine redundanten Implementierungen
- Syntax

`class Name<T> { T info; ...}`

`interface Name<T> {...}`

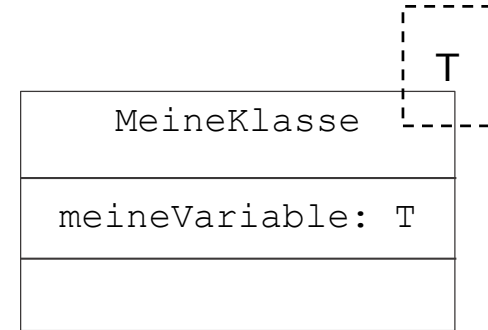
Typ-Parameter



# Generische Programmierung

## UML

```
class MeineKlasse<T>
{
    T meineVariable;
    ...
}
```



# Generische Programmierung

## Einführung

- Erzeugung von Instanzen

```
public class ArrayList<E> {...}
```

```
new ArrayList<Song>();
```

Typisierung wird bei der  
Instanziierung „nachgeholt“

- Variablendefinition

```
List<Song> songList = new ArrayList<Song>();
```

```
List<Song> songList = new ArrayList<>(); // geht auch
```

- Methoden

```
void add(T object) {...}
```

```
x.foo(wert);
```

„Diamant-Operator“

# Generische Programmierung

## Methoden Parametrisieren

- Außer Klassen können auch Methoden parametrisiert werden

```
class EineKlasse {  
  
    static <T> void einfuegen(Schachtel<T> box, T object) {  
        box.hineinlegen(object);  
    }  
}
```

```
Schachtel<Integer> box = new Schachtel<Integer>();  
EineKlasse.einfuegen(box, 42);
```

- Zusätzliche Platzhalter sind auch in parametrisierten Klassen möglich

```
class NochEineKlasse<T> {  
  
    <T> void methode(T param) {...}  
}
```

```
NochEineKlasse<Boolean> obj = new NochEineKlasse<Boolean>();  
obj.methode(42);
```

# Generische Programmierung

## Einschränkungen

- Die Typ-Parameter dürfen nur mit Referenz-Typen instanziiert werden (also keine primitiven Datentypen)
- Man kann die zulässigen Typen einschränken

```
class Flasche<T extends Getraenk> {...}
```

d.h. `Flasche<String>` geht dann nicht

- geht auch bei Interfaces

```
interface I<T extends I1 & I2 & I3> {...}
```

- Damit können Mindestanforderungen für eine Typvariable festgelegt werden

# Generische Programmierung

## Polymorphe Argumente mit Generics – Wildcards

- Polymorphie ist auch bei generischen Klassen anwendbar

```
public void los() {  
    Tier[] tiere = new Tier[2];
```

```
    tier[0] = new Nashorn();
```

```
    tier[1] = new Tiger();
```

```
    machGeraeusche(tiere);  
}
```

```
void machGeraeusche(Tier[] tiere) {  
    for (int i=0; i<tiere.length; i++) {  
        tiere[i].geraeschMachen();  
    }  
}
```

```
public void los() {  
    ArrayList<Tier> tierListe = new  
    ArrayList<>();
```

```
    tierListe.add(new Nashorn());
```

```
    tierListe.add(new Tiger());
```

```
    machGeraeusche(tierListe);  
}
```

```
void machGeraeusche(ArrayList<Tier> tiere) {  
    for (Tier tier : tiere) {  
        tier.geraeschMachen();  
    }  
}
```

jedes Objekt ruft seine  
eigene Methoden auf

# Generische Programmierung

## Polymorphe Argumente mit Generics – Wildcards

- Typkompatibilität ist nicht auf generische Klassen selbst anwendbar (durch generische Typen entsteht keine Vererbungshierarchie)

```
public void los() {  
    ArrayList<Tiger> tigerListe = new ArrayList<>();  
    tigerListe.add(new Tiger());  
    tigerListe.add(new Tiger());  
  
    machGeraeusche(tigerListe);  
  
    ArrayList<Tier> tierListe = new ArrayList<>();  
    tierListe = tigerListe;  
}  
  
void machGeraeusche(ArrayList<Tier> tiere){  
    for (Tier tier : tiere) {  
        tier.geraeuschMachen();  
    }  
}
```

The method machGeraeusche(ArrayList<Tier>) in the type TierTest is not applicable for the arguments (ArrayList<Tiger>)

Type mismatch: cannot convert from ArrayList<Tiger> to ArrayList<Tier>



# Generische Programmierung

## Polymorphe Argumente mit Generics – Wildcards

- Basistyp für alle Typen, die mit einer parametrisierten Klasse erzeugt werden können, ist der Wildcard-Typ "?"
- Platzhalter ? steht für einen beliebigen Typ
- Kann weiter eingeschränkt werden:

`class Name<? extends Klassenname> variablenName`

```
public void los() {  
    ArrayList<Tiger> tigerListe = new ArrayList<>();  
    tigerListe.add(new Tiger());  
    tigerListe.add(new Tiger());  
    machGeraeusche(tigerListe);  
}  
  
void machGeraeusche(ArrayList<?> tiere){  
    for (Object tier : tiere) {  
        ((Tier)tier).geraeuschMachen();  
    }  
}
```