Objektorientierte Programmierung

Bachelorstudiengang Informatik

Wintersemester 2021/22

Prof. Jan Rexilius

FH Bielefeld
University of
Applied Sciences

Bitte nehmen Sie an der
Modulbefragung / Kursbewertung teil!

Rückblick

1. **Statische Variablen und Methoden**
2. **Datumsangaben**
3. **Dateien**

Rückblick
Statische Variablen und Methoden

- Statische Variablen und Methoden benötigen keine Objekte, um eingesetzt zu werden. Sie werden in den Klassen selbst gespeichert
- Für die Definition einer statischen Variable oder Methode wird das Schlüsselwort `static` verwendet

```
public class MeineErsteAnwendung {
  public static void main (String[] args) {
    // hier könnte Ihr Code stehen
  }
}
```

Rückblick
Statische Variablen und Methoden

- Instanzvariablen: eine pro **Objekt**
- statische Variablen: eine pro **Klasse**

statische Variable:
eisInDerWaffel

Kind-Instanz eins

Kind-Instanz zwei

Rückblick
Statische Variablen und Methoden

- statische Variablen – Konstanten
- statische Methoden – können nicht überschrieben werden
- statische Klassen – können nicht erweitert werden (d.h. keine Unterklassen möglich)

```
public class Test {
  public final int FOO_X = 42;
}
```

```
public class Test {
  final void berechneWert() {
    // wichtige Sachen
  }}
```

```
final class Test {
  // kann nicht erweitert werden
}
```

Rückblick
Datum

- `String date = "18. October 2021";`
  `System.out.println(date);`

- Klassen in Java – bis Java 7:
  `java.util.Date` und `java.util.Calendar`

- Klassen in Java – ab Java 8:
  `java.time`

Rückblick
Dateien

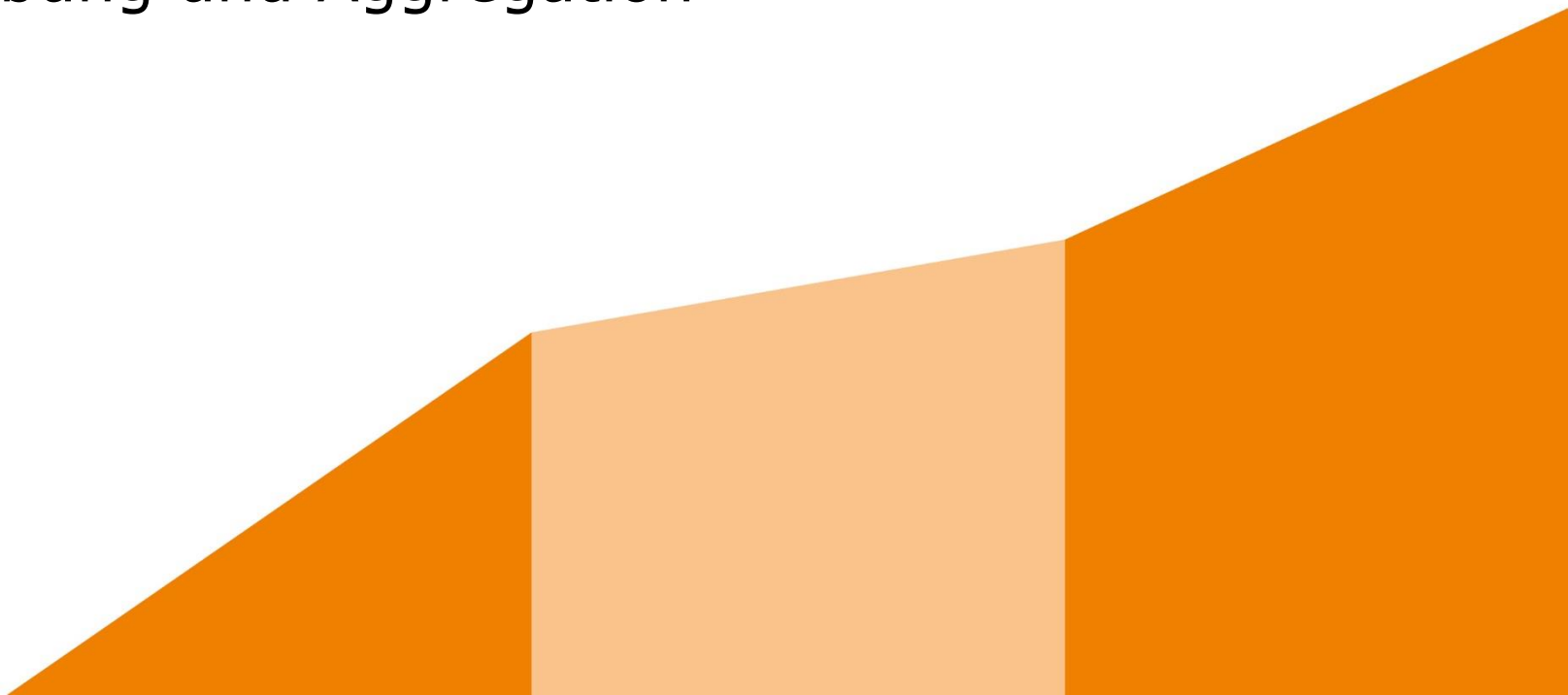
- Klassen in Java – bis Java 6: `java.io`
  - Beispiel: `java.io.File`

- Klassen in Java – ab Java 7: `java.nio`
  - Beispiel: `java.nio.file.Files`, `java.nio.file.Path`

Inhalte dieser Vorlesung

1. **Vererbung und Aggregation**
2. **Polymorphie**
3. **Exceptions**

Vererbung und Aggregation



Vererbung und Aggregation
Superklassen – Wiederholung

```
public class Tier {
  private int anzahlBeine;
  public void setAnzahlBeine(…) {…}
  public void geraeuschMachen(…) {…}
  public void schlafen(…) {…}
}
```

Superklassen

- abstrahieren Gemeinsamkeiten mehrerer verwandter Unterklassen
- haben die gleichen Member wie alle anderen Klassen auch (Instanzvariablen und Methoden)

Vererbung und Aggregation
Unterklasse – Wiederholung

```
public class Nashorn extends Tier {
  // Alles was ein Tier kann, das kann ich auch
}

public class Loewe extends Tier {
  // Ich auch. Und das ohne eine eigene Zeile Code
}
```

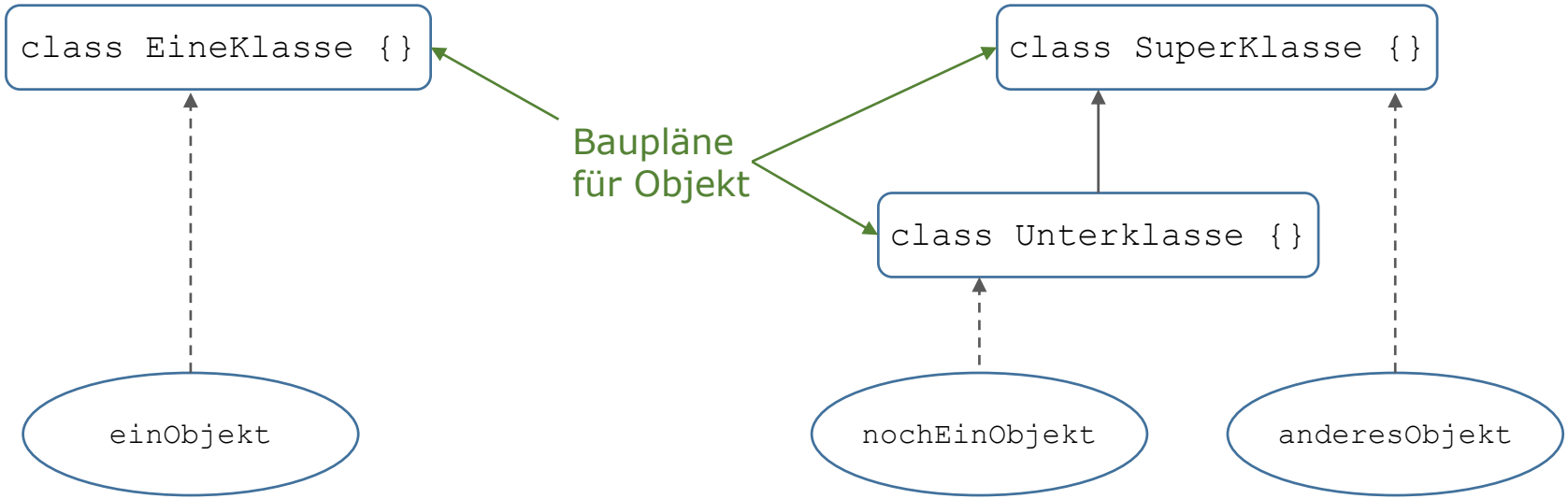
Unterklassen

- sind konkret
- erben die Member der Superklasse

Vererbung und Aggregation

Klassen und Objektstrukturen – Wiederholung

```
EineKlasse einObjekt = new EineKlasse();
Unterklasse nochEinObjekt = new Unterklasse();
Superklasse anderesObjekt = new Superklasse();
```



Vererbung und Aggregation
Einführung – Wiederholung

```
public class Tier {
  void geraeuschMachen() {
     // Sound laden, abspielen
  }
  void erzaehlWas() {
     // Informationen ausgeben
  }
} // class
```

Superklasse

erbt

erbt

erbt

```
public class Nashorn extends Tier{
   ...
} // class
```

Unterklasse

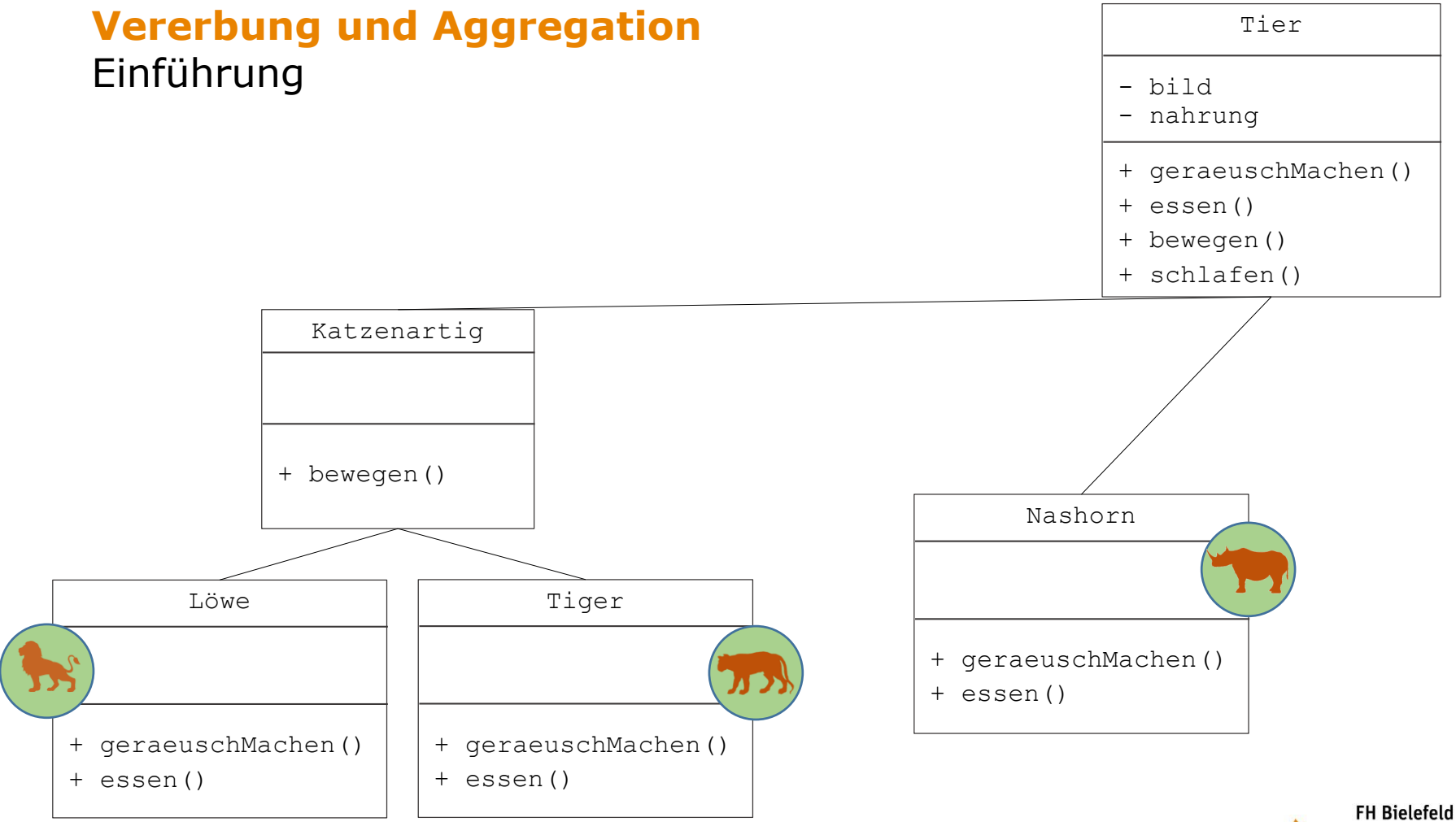
```
public class Löwe extends Tier {
   ...
} // class
```

Unterklasse

```
public class Elefant extends Tier
{
   ...
} // class
```

Unterklasse

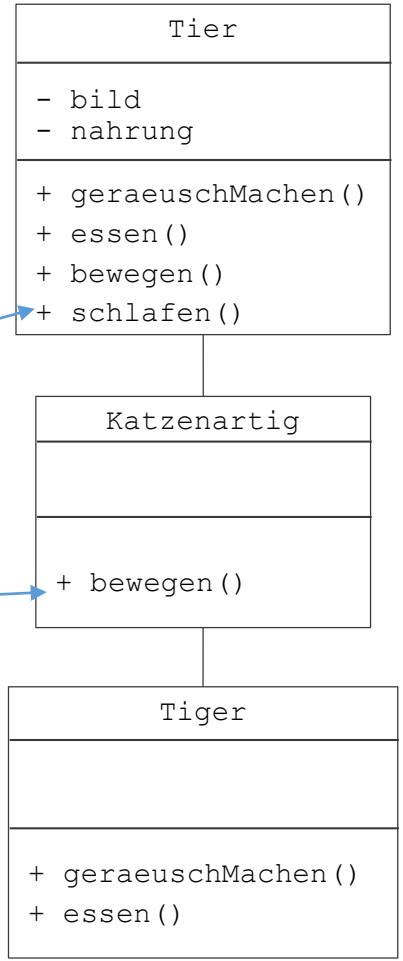
Vererbung und Aggregation
Einführung



Vererbung und Aggregation

Welche Methode wird aufgerufen?

```
Tiger t = new Tiger();

t.schlafen();

t.bewegen();

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



Vererbung und Aggregation
Vererbung – Beispiel

```
class Basis {
  int wert;

  void ausgeben() {
    System.out.println("Wert = " + wert);
  }
}

class Abgeleitet extends Basis {
  void erhoehen() {
    wert += 10;
  }
}

class Vererbung {
  public static void main(String[] args) {
    Abgeleitet obj = new Abgeleitet();
    obj.wert = 5;
    obj.erhoehen();
    obj.ausgeben();
  }}
```

```
Wert = 15
```

Ausgabe

Vererbung und Aggregation

Aufruf eines Basisklassenkonstruktors – `super`

- 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;
  }

}
```

Vererbung und Aggregation
Verdecken, Überschreiben, Überladen

```
class Basis {
  String name;

  setName (String n) {
    this.name = n;
  }}

class Abgeleitet extends Basis {
  String name;

  setNewName(String n) {
    this.name = n;
  }}

class Verdecken {
  public static void main(String[] args) {
    Abgeleitet obj = new Abgeleitet();
    obj.setName("Basis");
    obj.setNewName("Abgeleitet");

    System.out.println(obj.name);
}}
```

```
Abgeleitet
```

Ausgabe

Vererbung und Aggregation
Verdecken, Überschreiben, Überladen

```
class Basis {
  String name = "basis";

  void getName() {
    return name;
  }
}

class Abgeleitet extends Basis {
  String name = "abgeleitet";

  void ausgeben() {
    System.out.println("eigener Name = " + name);
    System.out.println("eigener Name = " + getName());
    System.out.println("eigener Name = " + super.name);
}}

class Verdecken {
  public static void main(String[] args) {
    Abgeleitet obj = new Abgeleitet();
    obj.ausgeben();
  }
}
```

```
eigener Name = abgeleitete

eigener Name = basis

eigener Name = basis
```

Ausgabe

Vererbung und Aggregation

Verdecken, **Überschreiben**, Überladen

```
class Zaehler {
  int wert;

  public Zaehler () {
    wert = 0;
  }
  public void weiterdrehen() {
    wert++;
  }
  public int abfragen() {
    return wert;
  }
}
```

```
class ZaehlerP2 extends Zaehler {
  public ZaehlerP2 (int start) {
    wert = start;
  }
  public void weiterdrehen() {
    wert += 2;
  }
}
```

```
class ZaehlerP2Test {

 public static void main(String[] args) {

  ZaehlerP2 z = new ZaehlerP2();

  for (int i=0; i<=10; i++) {
    System.out.println(" " + z.abfragen());
    z.weiterdrehen();
  }
 }
}
```

Vererbung und Aggregation
Verdecken, **Überschreiben**, Überladen

- Anpassen von Methodenverhalten in Unterklassen:
  - gleicher Methodenname
  - gleiche Parameteranzahl und –typen
  - Annotation: **@override**

```
class Zaehler {
  ...

  public void weiterdrehen() {
    wert++;
  }
  ...
}
```

```
class ZaehlerP2 extends Zaehler {
  ...
  @override
  public void weiterdrehen() {
    wert += 2;
  }
  ...
}
```

Vererbung und Aggregation
Verdecken, Überschreiben, **Überladen**

```
class Basis {
  int wert = 1;

  void erhoehen() {
    wert++;
  }
}
```

```
class Abgeleitet extends Basis {

  void erhoehen(int betrag) {
    wert += betrag;
  }
}
```

```
class AbgeleitetTest {

 public static void main(String[] args) {

   Abgeleitet obj = new Abgeleitet();

   obj.erhoehen();
   System.out.println("Wert = "+obj.wert);
   obj.erhoehen(10);
   System.out.println("Wert = "+obj.wert);
  }
}
```

```
Wert = 2

Wert = 14
```

Ausgabe

Vererbung und Aggregation

IS-A und HAS-A

```
public class Tier {
  void geraeuschMachen() {
     // Sound laden, abspielen
  }
  void erzaehlWas() {
     // Informationen ausgeben
  }
} // class
```

Superklasse

IS-A

IS-A

```
public class Nashorn extends Tier{
  ...
} // class
```

Unterklasse

```
public class Löwe extends Tier {
  ...
} // class
```

Unterklasse

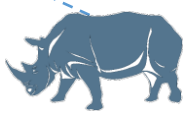
IS-A

IS-A

IS-A

IS-A

norbert



nepumuk

leo

lukas

FH Bielefeld
University of
Applied Sciences

Vererbung und Aggregation
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;
}
```

Vererbung und Aggregation

Einige wichtige Fakten

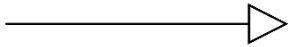
- Die Basisklasse vererbt alle Elemente mit Ausnahme der Konstrukturen und der `private`-Elemente
- Es gibt keine Möglichkeit, auf die Auswahl der vererbten Elemente einzuwirken
- Die Basisklasse wird durch die Vererbung nicht verändert
- Eine Java-Klasse kann nicht gleichzeitig von zwei oder mehreren Basisklassen abgeleitet werden.
- Klasse, die als `final` definiert sind, können nicht als Basisklasse verwendet werden.

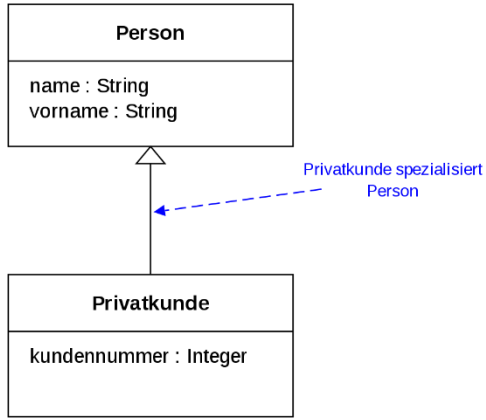
```
final class B {…}

class A extends B { -> FEHLER!
```

Vererbung und Aggregation

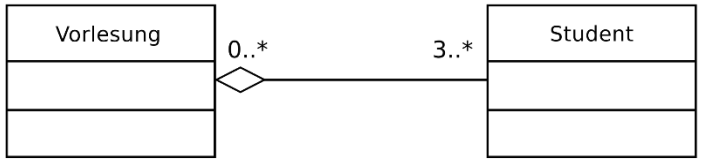
UML

- Vererbung (IS-A)





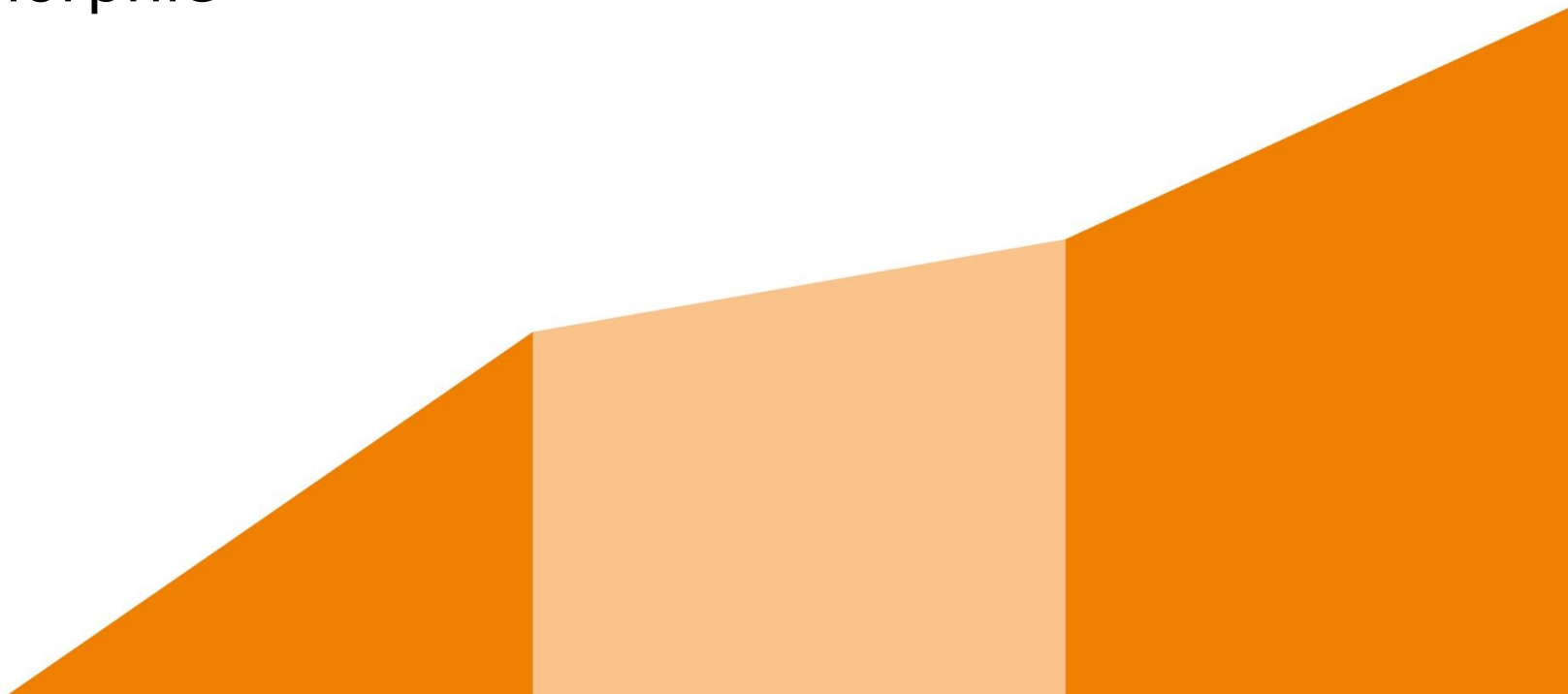
- Aggregation (HAS-A)



Quelle: wikipedia

Objektorientierte Programmierung

Polymorphie



Polymorphie

Die drei Schritte der Objektdeklaration und -zuweisung (Wdh.)

1. Eine Referenzvariable deklarieren

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

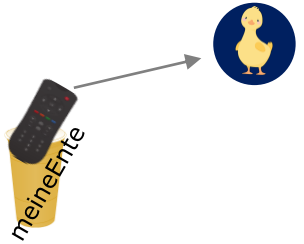
2. Ein Objekt erzeugen

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

Enten-Objekt

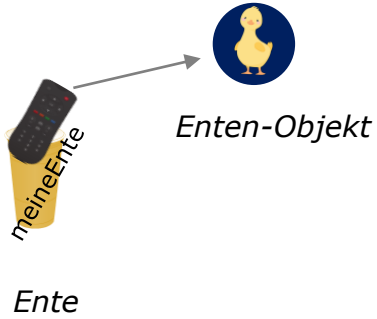
3. Objekt und Referenz verknüpfen

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



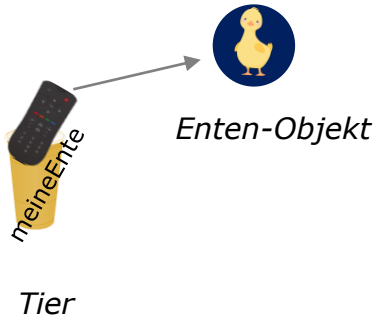
Polymorphie
Einführung

- Referenztyp und Objekttyp sind gleich



- Mit Polymorphie können Referenz und Objekt unterschiedlich sein!

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



Polymorphie
Upcasting und Downcasting

- **Upcast**

```
Ente e = new Ente();
Tier t = e;
```

- **Downcast**

```
Tier t1 = new Tier();
Tier t2 = new Ente();
Ente e1 = t1;          // error at compile-time
Ente e2 = (Ente) t1;   // OK at compile-time. Error at runtime.
Ente e3 = (Ente) t2;   // OK
```

Polymorphie
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();
}
```

Array durchlaufen und Methoden der Klasse Tier aufrufen

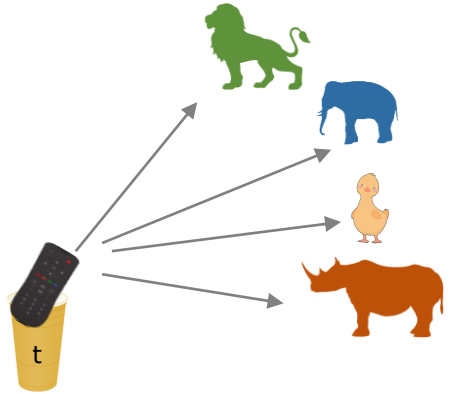
jedes Objekt ruft seine eigene Methoden auf

Polymorphie
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);
  }
}
```

geraeuschMachen() von Ente aufrufen

geraeuschMachen() von Nashorn aufrufen



Tier

Polymorphie
Statische / Dynamische Bindung

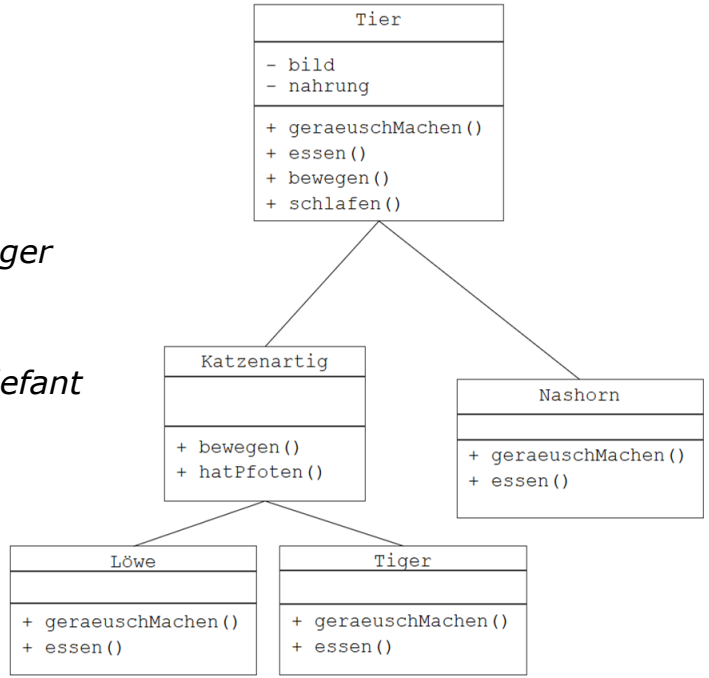
- Java gibt dem Typ des Objekts den Vorzug gegenüber dem Typ der Objektvariablen

```
Tier einTier;
einTier = new Tiger();
einTier.geraeuschMachen();
```

← *Objekt ist vom Typ Tiger*

```
einTier = new Elefant();
einTier.geraeuschMachen();
```

← *Objekt ist vom Typ Elefant*



- Codeanalyse nach dem richtigen Typ kann aufwendig werden. Beispiel:

```
Tier einTier;
if (Math.random()>0.5)
  einTier = new Ente();
else
  einTier = new Elefant();
einTier.geraeuschMachen()
```

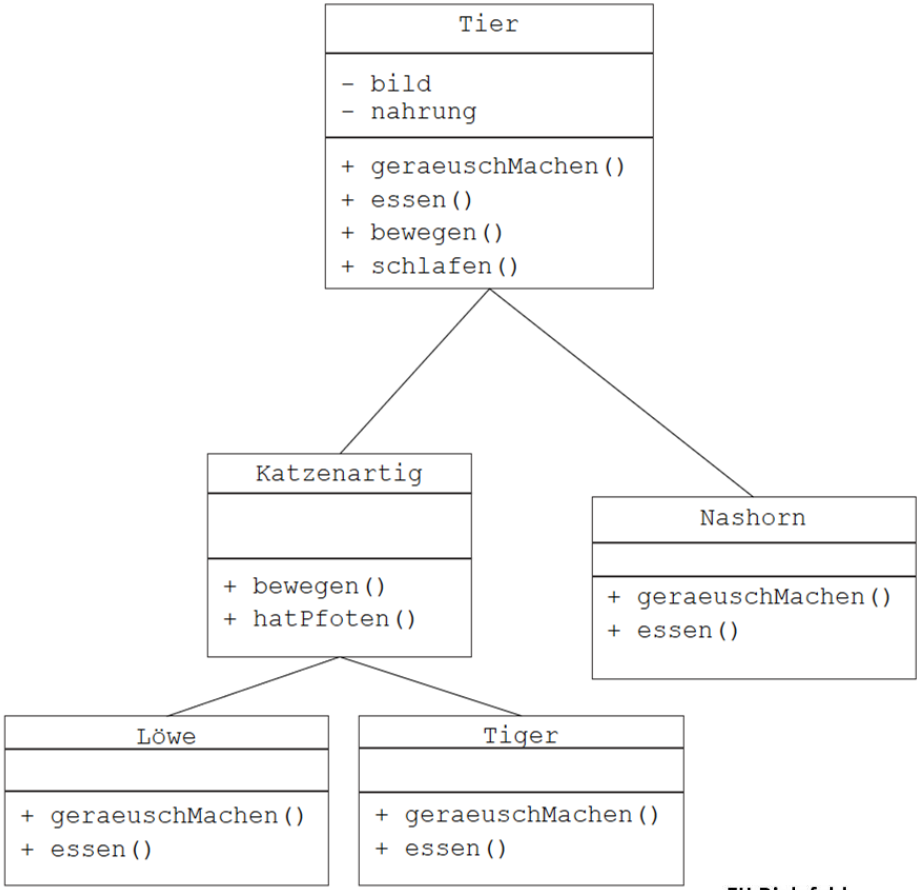
Polymorphie
Statische / Dynamische Bindung

- **Dynamische Bindung**
  Ein Methodenaufruf `obj.m()` wird erst zur Laufzeit (dynamisch) an die passende Methodendefinition gebunden
  - Basierend auf dem Typ des Objekts und nicht dem Typ der Objektvariablen
  - Bsp: `Tier einTier = new Tiger(); einTier.geraeuschMachen();`

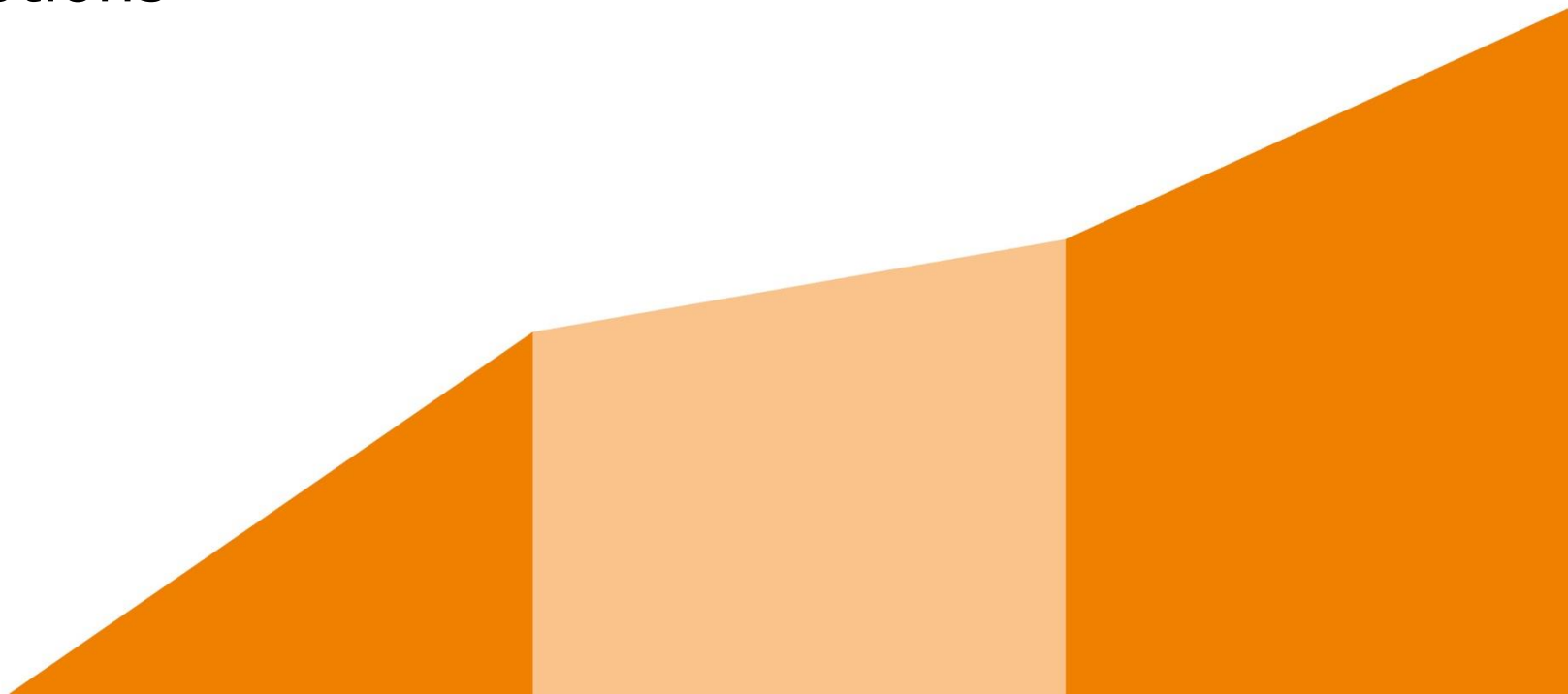
- **Statische Bindung**
  Typ wird bereits zur Entwurfszeit festgelegt
  - erfolgt bei
    - privaten, statischen und finalen Methoden
    - Überladen von Methoden

Polymorphie
Übung – Spielen Sie Compiler

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

tier[0].hatPfoten();
tier[1].schlafen();
tier[2].hatPfoten();
```



Exceptions



Exceptions
Einführung

- Was könnte hier schief gehen?

```
public int division(int zaehler, int nenner) {
  int ergebnis = zaehler / nenner;
  return ergebnis;
}

division(12,4); // kein Problem
division(17,0); // da müssen wir aufpassen
```

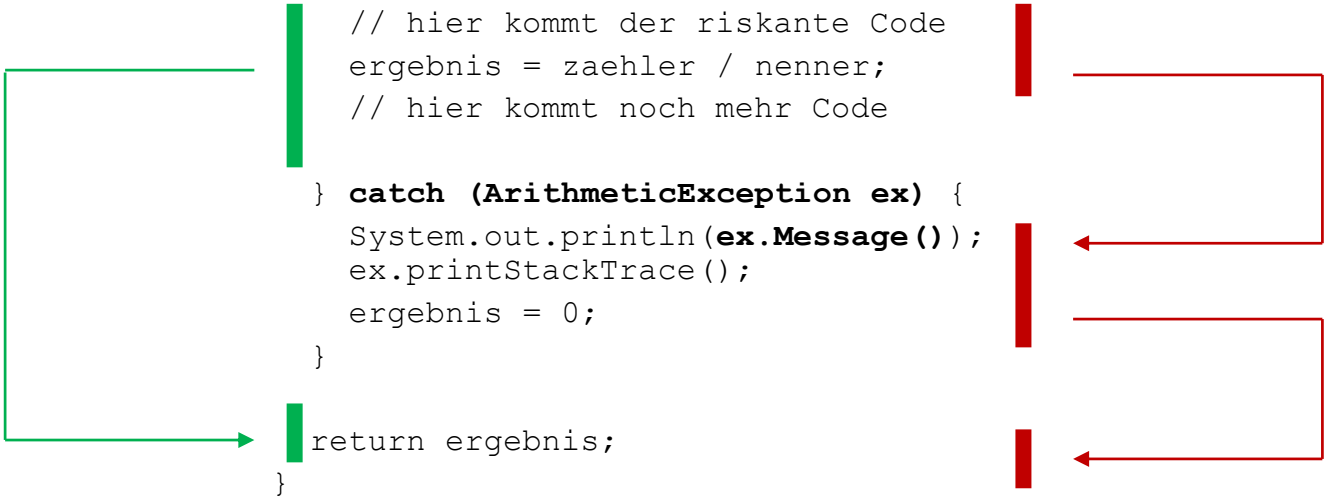
```
Exception in thread "main"
java.lang.ArithmeticException: / by zero
        at Test.division(Basis.java:11)
        at Ttest.main(Basis.java:42)
```

Exceptions
Try und Catch

`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.Message());
    ex.printStackTrace();
    ergebnis = 0;
  }

  return ergebnis;
}
```

`try` scheitert…



Exceptions
Try und Catch

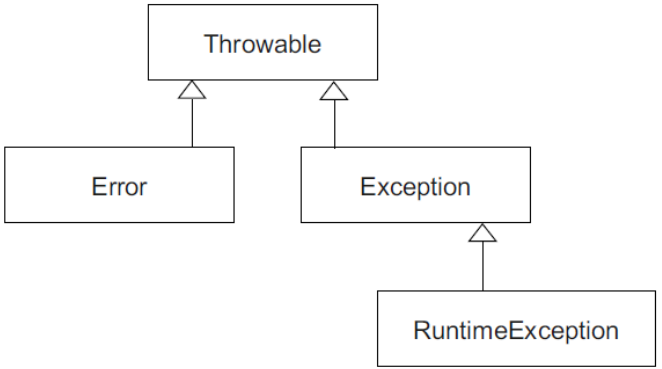
```
try {
   // hier kommt der riskante Code
} catch (ExceptionTyp1 ex) {
   // Behandlung von ExceptionTyp1
} catch (ExceptionTyp2 ex) {
  // Behandlung von ExceptionTyp2
} finally {
  // dieser code wird immer am Ende ausgeführt
  // egal ob eine Ausnahme eingetreten ist
}
```

„Multicatch“

```
...catch (ExceptionTyp1 | ExceptionTyp2 ex) {...}
```

Exceptions
Zusammenfassung

- Exceptions
  - behandeln Fehlersituationen (Ausnahmen) im Programmablauf
  - werden zur Laufzeit überprüft
  - dienen der Unterbrechung des normalen Kontrollflusses
- Exceptions sind Objekte
  - Enthalten Informationen zum aufgetretenen Fehler
  - `Throwable` ist Superklasse aller Fehler in Java
- wichtige Methoden von Java Exception-Klassen
  - String getMessage()
  - void printStackTrace()
  - StackTraceElement[] getStackTrace()



Quelle:Abts, Grundkurs Java, Springer

Exceptions
Gruppierung

- RuntimeExceptions **müssen nicht** abgefangen werden
  - ArithmeticException
  - ClassCastException
  - NullPointerException
  - IndexOutOfBoudnsExceptions

- Bestimmte Exceptions **müssen** abgefangen werden
  - IOException
  - FileNotFoudException
  - UnsupportedAude

Exceptions

Kontrollfluss – `throw` / `throws`

- Erzeugen / "werfen" einer Exception

```
throw obj;
```

- Einfachste Form

```
throw new Exception();
```

- Auch im Methodenkopf möglich

```
String readFromFile(String name) throws FileNotFoundException
{…}
```

Exceptions

Kontrollfluss – `throw` / `throws`

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

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

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